Leveraging streaming for deterministic parallelization: an integrated language, compiler and runtime approach

Antoniu Pop

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

MINES ParisTech

École doctorale n°432:
Sciences des Métiers de l'Ingénieur

Doctorat européen ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

l'École nationale supérieure des mines de Paris

Spécialité "Informatique temps-réel, robotique et automatique"

présentée et soutenue publiquement par

Antoniu POP

le 30 septembre 2011

Exploitation du streaming pour la parallélisation déterministe
– approche langage, compilateur et système de runtime intégrée –

~ ~ ~

Leveraging Streaming for Deterministic Parallelization
– an Integrated Language, Compiler and Runtime Approach –

Directeur de thèse: **François IRIGOIN**

Jury

Philippe CLAUSS, Professeur, ICPS/LSIIT, Université de Strasbourg	Rapporteur
Albert COHEN, Directeur de Recherche, PARKAS, INRIA	Examinateur
François IRIGOIN, Maître de Recherche, CRI, MINES ParisTech	Directeur de thèse
Paul H J KELLY, Professeur, SPO, Imperial College London	Rapporteur
Fabrice RASTELLO, Chargé de Recherche, Compsys/LIP/ENS Lyon, INRIA	Examinateur
Pascal RAYMOND, Chargé de Recherche, Verimag, CNRS	Examinateur
Eugene RESSLER, Professeur, EECS, United States Military Academy	Président

T H È S E

MINES ParisTech
Centre de recherche en Informatique
35, rue Saint Honoré, 77305 Fontainebleau Cedex, France

Remerciements

Cette thèse a été préparée au *Centre de recherche en Informatique (CRI)* de l'École des mines de Paris (MINES ParisTech), entre septembre 2008 et septembre 2011, sous la direction de François Irigoin.

Je voudrais tout d'abord exprimer ma reconnaissance à François Irigoin (professeur au CRI), qui a encadré cette thèse, et à Robert Mahl (directeur du CRI) : cette thèse n'aurait pas été possible sans leur confiance et la liberté absolue qu'ils m'ont accordée, à la fois sur le sujet et sur les axes explorés. Avec l'aide de Pierre Jouvelot (maître de recherche au CRI), ils ont réussi à modérer mes excès d'enthousiasme et à apporter la rigueur nécessaire à ce travail.

Je remercie tout particulièrement Albert Cohen (directeur de recherche à l'INRIA, École Normale Supérieure) qui a pris intérêt à ma thèse dès les premiers mois, en a suivi l'évolution et m'a soutenu par sa collaboration pendant ces trois années ; à son insu, il m'avait déjà, dès 2005, contaminé par sa passion pour la recherche. Son enthousiasme, sa disponibilité et les nombreux échanges d'idées ont grandement contribué à cette thèse. Son aide et ses conseils ont été inestimables.

Je suis très reconnaissant envers les membres de mon jury, en particulier envers Philippe Clauss (professeur à l'ICPS/LSIIT, Université de Strasbourg) et Paul Kelly (professeur au SPO, Imperial College London) pour l'attention qu'ils ont portée à mes travaux, pour le soin avec lequel ils ont relu cette thèse et pour leurs conseils éclairés. Je remercie également Pascal Raymond (chargé de recherche CNRS à Verimag) et Fabrice Rastello (chargé de recherche INRIA à Compsys/LIP/ENS Lyon) pour leur participation à ce jury. Je remercie enfin Eugene Ressler (professeur au EECS, United States Military Academy) de m'avoir fait l'honneur de présider ce jury.

Je me rappellerai toujours avec un grand plaisir des passionnantes discussions avec Robert Mahl, et de l'excellente ambiance et la bonne humeur qui régnait au CRI. Je repense à tous les bons moments passés dans la compagnie des membres du centre, tout particulièrement Benoît Pin, Pierre Jouvelot et Samuel Benveniste, de Laurent Daverio, Corinne Ancourt, Fabien Coelho, Claire Médrala, Amira Mensi, Dounia Khaldi et bien entendu notre secrétaire Jacqueline Altimira dont l'efficacité et la bonne humeur ne m'ont jamais fait défaut.

Merci enfin à ma famille, et aux amis que je n'ai pas cités, pour leur soutien inconditionnel et leurs encouragements, avec une dernière pensée particulière pour ma compagne, Miryam.

Abstract

As single processing unit performance has reached a technological limit, the power wall, the past decade has seen a shift from the prevailing trend of increasing single-threaded performance to an exponentially growing number of processing units per chip. Higher performance returns on newer architectures are contingent on the amount of parallelism that can be efficiently exploited in applications, either exposed through parallel programming or by parallelizing compilers. However, uncovering raw parallelism is insufficient if a host of cores vie for limited off-chip memory bandwidth. Mitigating the memory wall, the stream-computing model provides an important solution for exploiting upcoming architectures.

This thesis explores streaming as a general-purpose parallel programming paradigm, rather than a model dedicated to a class of applications, by providing a highly expressive stream-computing extension to a de facto standard for shared memory programming, OpenMP. We rely on a new formal framework to investigate the properties of streaming programs, without the restrictions usually attached to dataflow models, and we prove that such programs benefit from deadlock and functional determinism, key assets in the productivity race. Finally, we focus on the efficient exploitation of our model, with optimized runtime support and compiler optimizations, through an implementation in the GCC compiler.

Résumé

La performance des unités de calcul séquentiel a atteint des limites technologiques qui ont conduit à une transition de la tendance à l'accélération des calculs séquentiels vers une augmentation exponentielle du nombre d'unités de calcul par microprocesseur. Ces nouvelles architectures ne permettent d'augmenter la vitesse de calcul que proportionnellement au parallélisme qui peut être exploité, soit via le modèle de programmation soit par un compilateur optimiseur. Cependant, la disponibilité du parallélisme en soi ne suffit pas à améliorer les performances si un grand nombre de processeurs sont en compétition pour l'accès à la mémoire. Le modèle de streaming répond à ce problème et représente une solution viable pour l'exploitation des architectures à venir.

Cette thèse aborde le streaming comme un modèle général de programmation parallèle, plutôt qu'un modèle dédié à une classe d'applications, en fournissant une extension pour le streaming à un langage standard pour la programmation parallèle avec mémoire partagée, OpenMP. Un nouveau modèle formel est développé, dans une première partie, pour étudier les propriétés des programmes qui font appel au streaming, sans les restrictions qui sont généralement associées aux modèles de flot de données. Ce modèle permet de prouver que ces programmes sont déterministes à la fois fonctionnellement et par rapport aux deadlocks, ce qui est essentiel pour la productivité des programmeurs. La deuxième partie de ce travail est consacrée à l'exploitation efficace de ce modèle, avec support logiciel à l'exécution et optimisations de compilation, à travers l'implantation d'un prototype dans le compilateur GCC.

Contents

List of Tables

List of Figures

Introduction en français

La performance des unités de calcul séquentiel a atteint des limites technologiques qui ont conduit l'industrie des microprocesseurs à développer des architectures multi-cœurs [2, 37,39,61], où plusieurs cœurs, ou unités de calcul, sont gravés sur la même puce. Alors que le nombre de transistors continue à augmenter de façon exponentielle, la complexité des unités de calcul stagne, voire même décroît, les fabricants de processeurs utilisant les transistors additionnels pour fournir davantage de cœurs simplifiés, moins gourmands en énergie et souvent avec une vitesse d'horloge plus faible. Par conséquent, les applications séquentielles ne verront pas leur performance augmenter avec ces nouvelles générations de processeurs de la même façon que par le passé. L'accélération des calculs dépend désormais essentiellement du parallélisme qui peut être efficacement exploité dans les applications, directement par programmation parallèle ou par parallélisation automatique avec des compilateurs paralléliseurs. Cependant, le parallélisme en soi n'est pas suffisant et même les applications massivement parallèles ne sont pas assurées d'obtenir des accélérations linéaires en fonction du nombre d'unités de calcul et encore moins de passer a l'échelle de manière portable, ce qui est essentiel dans un contexte où les architectures matérielles deviennent obsolètes très rapidement.

Paralléliser efficacement une application requiert en premier lieu d'exposer le parallélisme latent. Bien que ce problème ait été largement étudié, il s'est révélé être extrêmement difficile à résoudre, à la fois sous forme de paradigmes de programmation visant à faciliter l'expression du parallélisme par les développeurs et sous la forme de traduction automatique de codes séquentiels en codes parallèles. La première solution est très onéreuse en termes de ressources de développement et mène souvent à des problèmes de portabilité ou de portabilité des performances, alors que la deuxième solution semble mettre les compilateurs optimiseurs devant une difficulté insurmontable : choisir une transformation de code qui améliore les performances, sans même parler de transformation optimale, pour un modèle de machine donné.

En second lieu, il faut exploiter le parallélisme ainsi exposé sur une plateforme d'exécution tout en veillant à ce que les gains de performance puissent passer à l'échelle. La plus grande difficulté à cet égard vient d'un problème qui précède l'avènement des architectures multi-cœurs, le *memory wall*. Alors que les architectures mono-processeur sont fortement impactées par une latence élevée de l'accès à la mémoire, ces nouvelles architectures sont également soumises à une forte insuffisance de la bande passante mémoire. L'adoption de modèles de consistance mémoire relaxés, ainsi que le développement de hiérarchies de mémoires de cache complexes, ont permis de pallier les problèmes de latence et de bande passante pour les mono-processeurs, mais ces solutions deviennent

vite insuffisantes lorsqu'un nombre élevé d'unités de calcul se trouvent en concurrence pour une bande passante limitée vers la mémoire.

Motivation

Afin de faciliter l'exploitation d'architectures matérielles dont la complexité évolue rapidement et présentant un faible rapport de bande passante mémoire disponible par instruction, des langages de haut niveau ont été conçus pour exprimer l'(in)dépendance, les schémas de communication et la localité sans faire référence à une plateforme d'exécution particulière. Les compilateurs et systèmes de runtime sont responsables d'abaisser le niveau d'abstraction et d'orchestrer les threads d'exécution et la gestion de la mémoire. En particulier, le modèle de programmation par *stream* a récemment attiré beaucoup d'attention, étant un mode de programmation parallèle ayant un large champ d'application et qui garantit le déterminisme fonctionnel[1], un atout de taille pour améliorer la productivité. Ce modèle est aussi propice pour rendre le flôt de données explicite et pour structurer les programmes de façon à permettre d'exploiter simultanément le parallélisme de pipeline, de données et de tâches. Enfin le *streaming* permet également de réduire la sévérité du *memory wall* à travers deux effets complémentaires : (1) la latence d'accès à la mémoire est naturellement cachée par le découplage entre producteurs et consommateurs dans un pipeline ; et (2) la communication par streams favorise le transfert de données local à la puce, contournant la mémoire globale, ce qui réduit la pression exercée sur le système mémoire. C'est pourquoi le streaming paraît être une approche parfaitement adaptée aux problèmes soulevés par le *memory wall*.

Sous des formes diverses et variées, le streaming a été présent pendant près de cinquante ans, bien que le principal objectif d'organiser les calculs autour de streams de données ait beaucoup évolué durant cette période. Dans son étude du streaming [67], Stephens passe en revue l'histoire du développement de ce modèle de calcul, depuis la conception des modèles sémantiques pour formaliser le calcul avec des streams et jusqu'à l'analyse de flôts de données, au traitement du signal, aux systèmes réactifs ou à la conception et la vérification matérielles. Il classifie les systèmes de calcul en streaming sur la base de trois critères : la synchronie, le déterminisme et le type des canaux de communication. Les modèles de calcul par streams partagent, fondamentalement, la même structure, qui peut généralement être représentée sous forme de graphe, où des nœuds de calcul sont connectés à travers des arrêtes qui représentent des streams.

Le modèle de réseaux de processus de Kahn (*Kahn Process Networks (KPN)*) [36] est l'un des premiers modèles et aussi l'un des plus influents. Dans un KPN, des processus *data-driven* communiquent par des canaux de type FIFO non bornés, où les opérations d'écriture sont non-bloquantes alors que les opérations de lecture attendent qu'une quantité de données suffisante soit disponible. Les processus étant considérés être déterministes, l'une des propriétés les plus importantes de ce modèle est que le réseau dans son intégralité est également fonctionnellement déterministe par composition.

[1] Un programme est fonctionnellement déterministe s'il évalue une fonction de ses entrées, donc si ses sorties sont entièrement déterminées par ses entrées.

Cependant, les réseaux contenant des cycles peuvent mener à des interblocages, ce qui a motivé le développement d'une forme très restreinte de réseaux de processus de Kahn, le Synchronous Data-Flow (SDF) [43]. Contrairement aux processus d'un KPN qui s'exécutent de manière asynchrone et qui peuvent produire ou consommer une quantité arbitraire de données à chaque activation, les processus SDF ont un comportement déterminé statiquement. En connaissant les taux de production et de consommation de données lors de la compilation, il est possible de déterminer statiquement si l'exécution d'un SDF va mener à un interblocage et de construire un ordonnancement statique de l'exécution. Il est également possible de garantir l'absence d'interblocages dus au manque de ressources lors d'une exécution en mémoire bornée et de calculer précisément la place qu'occupent les canaux de communication. Certaines restrictions du modèle SDF sont relaxées dans le Cyclo-Static Data-Flow [13]. StreamIt [68] est l'incarnation la plus récente de ce modèle, se basant sur les fortes restrictions statiques du modèle sous-jacent pour rendre possible des optimisations agressives dans le compilateur. Il permet d'obtenir d'excellents résultats de performance et de portabilité de la performance sur diverses plateformes d'exécution [26] pour un ensemble restreint de programmes qui se conforment à ce modèle.

Les langages synchrones, qui comprennent LUSTRE [28], Signal [27] et Esterel [12], ont connu un grand succès dans les applications industrielles, en grande partie dus à leurs propriétés qui garantissent le déterminisme, l'absence d'interblocages et la sécurité. Fondés sur les modèles de systèmes réactifs ou de réseaux de traitement du signal, avec la distinction que les systèmes réactifs utilisent des canaux de communication bi-directionnels, ces langages ne requièrent pas de périodicité. Les processus y communiquent à travers des signaux, qui permettent aussi de définir une notion de temps et de causalité. Les signaux diffèrent des streams par le fait qu'ils sont échantillonnés plutôt que consommés.

Des langages flôt de données ont aussi été développés, où l'exécution est déterminée par les dépendances de données au lieu d'un flôt de contrôle. Ils s'apparentent au streaming par l'importance accordée aux dépendances de flôt de données, qui représentent les relations producteur-consommateur, au cœur des deux modèles. Le langage Lucid [7] utilise le mot-clé `next` à l'intérieur des boucles pour créer un effet semblable à l'avancement dans un stream de données, en consommant ou en produisant dans un canal, ou bien, dans le domaine des langages synchrones, à l'avancement des horloges sur les signaux. Les programmes Lucid sont, fondamentalement, des systèmes d'équations de récurrence. Le langage de programmation fonctionnelle Sisal [25] introduit la notion de stream de manière explicite, sous une forme naturellement très proche des listes. Si l'on comprend les systèmes de calcul avec streams comme une implantation concurrente des transformeurs de stream, qui est l'interprétation fonctionnelle d'un réseau de processus qui à un ensemble de streams d'entrée associe un ensemble de streams de sortie, alors tout langage fonctionnel peut être utilisé pour la programmation streaming. Cela correspond à une interprétation paresseuse des langages fonctionnels ; voir [18] pour une implantation de Lucid Synchrone [17] en Haskell [32].

Le modèle Communicating Regular Processes (CRP) [23], lui, se fonde sur un schéma de communication non-FIFO, où les processus écrivent et lisent dans les canaux de communication à des indices spécifiques, en respectant une propriété d'affectation dynamique

unique. Les processus consommateurs ne peuvent lire dans les canaux qu'à des indices ayant précédemment été écrits par le producteur, ce qui permet de faire respecter les dépendances de flôt pour chaque élément individuellement. Cette approche est bien plus flexible que les modèles basés sur le SDF, mais il serait très difficile d'implémenter cette méthode de manière efficace dans un langage de programmation streaming. Les propriétés des programmes écrits dans ce modèle ne peuvent être statiquement déterminées que lorsque les relations sont affines. Cependant, l'objectif dans ce cas est d'isoler les contraintes d'ordonnancement des différents processus afin de réduire la taille du problème au niveau du programme entier.

Chacune de ces diverses approches de la programmation streaming a le potentiel d'aider à réduire la sévérité du *memory wall*, mais cela n'est applicable qu'à des classes restreintes d'applications. Les programmes de streaming sont généralement considérés comme construits autour une notion de streams réguliers de données, ce qui s'adapte parfaitement aux modèles où les canaux de communication sont implémentés sous la forme de queues FIFO avec un seul producteur et un seul consommateur. Si l'on veut donner au streaming un rôle plus central dans la programmation d'applications pour les architectures multi-cœurs actuelles et les architectures many-core à venir, il faut adopter un modèle plus général, où les schémas de communication ne sont pas toujours réguliers et statiquement déterminés, mais où des schémas entièrement dynamiques peuvent apparaître et être dynamiquement exploités, en fonction de l'exécution du programme. L'intuition selon laquelle le flôt des données joue un rôle central dans tout programme n'est pas fausse, mais ce flôt de données se trouve souvent conditionné par du flôt de contrôle complexe, dû à des événements irréguliers, comme c'est le cas dans les programmes synchrones.

Un modèle général de programmation doit permettre d'exprimer le parallélisme sous toutes ses formes, et en particulier sous sa forme la plus générale, le parallélisme de contrôle, tout en donnant suffisamment d'informations au compilateur pour générer du code de streaming. Le parallélisme de tâches n'est pas toujours le meilleur choix en termes de performance, mais si le flôt de données entre les tâches est connu, alors la communication peut être implémentée avec des streams, ce qui permet d'appliquer un nombre d'importantes optimisations. Par exemple, lorsque le compilateur découvre que les invariants SDF sont satisfaits, il peut déclencher des transformations de boucle agressives pour adapter le parallélisme de tâches à l'architecture cible, permettant ainsi d'obtenir le même niveau de portabilité de performance que celui des langages basés sur le modèle SDF, ce qui est le cas dans StreamIt.

Il est essentiel, par ailleurs, que ces nouvelles approches du streaming préservent autant que possible les propriétés fortes que fournissent les modèles existants, comme le déterminisme fonctionnel ou l'absence d'interblocages. Le debugging d'applications concurrentes est bien plus compliqué que dans le cas séquentiel, principalement en raison de la nature non-déterministe des erreurs liées aux interblocages et aux courses critiques. Lorsque le comportement d'une application n'est pas facilement reproductible, les procédures de test deviennent statistiques et l'identification des erreurs devient difficile puisque tout outil de debugging impacte le comportement des applications.

Cependant, développer un nouveau langage de programmation serait inutilement coûteux en termes de formation des développeurs et cela requiert de fournir et maintenir de nouvelles chaînes de compilation et de debugging. Un nouveau langage ne protège pas pour

autant les programmeurs de la nécessité de mélanger différents styles de programmation et de constructions parallèles. Alors que plusieurs langages ont déjà été conçus pour exploiter le parallélisme autour du streaming, nous partons du principe qu'il est plus intéressant, et suffisant, d'introduire des modifications minimales et incrémentales dans un langage qui a fait ses preuves en programmation parallèle, ce qui permet de mettre à profit des interactions maîtrisées entre le streaming et les constructions séquentielles et parallèles fournies.

Dans cette thèse, nous étudierons le streaming en tant que modèle général de programmation parallèle, plutôt qu'un modèle dédié à une classe d'applications donnée, en élaborant une extension pour l'un des langages les plus couramment utilisés pour la programmation parallèle à mémoire partagée, OpenMP.

Objectifs de la thèse

L'objectif principal de cette thèse est de développer et de promouvoir l'usage du streaming comme un modèle général de programmation parallèle. Dans cette perspective, nous identifions les cinq principaux axes de recherche suivants :

1. Développer le streaming dans le cadre d'un langage de programmation parallèle bien établi, pour permettre de mélanger la programmation streaming avec d'autres formes de parallélisme dans un environnement intégré et entièrement contrôlé. Cela permet de réduire le coût de la barrière du langage pour les programmeurs, et de réutiliser des outils préexistants et largement disponibles pour la compilation.

2. Limiter autant que possible les restrictions habituellement associées à la programmation par streaming, telles que la synchronie, la périodicité ou la régularité. Ces restrictions réduisent fortement le champ d'application du streaming, et le confinent à quelques classes d'applications.

3. Montrer qu'il est possible de préserver les propriétés utiles des modèles de streaming, comme le déterminisme ou l'efficacité, malgré la relaxation des contraintes mentionnées ci-dessus.

4. Fournir une aide au debugging, à travers, par exemple, la détection automatique d'interblocages.

5. Permettre d'utiliser le streaming en mémoire partagée, mais aussi en mémoire distribuée. Les nouvelles architectures adoptent des modèles de consistance mémoire de plus en plus flexibles afin de réduire le coût de la cohérence de mémoire de cache entre un nombre de cœurs toujours plus grand, mais en fin de compte le passage à l'échelle est toujours plus limité dans un seul nœud de calcul qu'en augmentant le nombre de nœuds.

Les moyens à mettre en œuvre pour atteindre ces objectifs sont les suivants :

- Développer une modèle sémantique pour l'extension de langage streaming, en premier lieu en se basant sur des hypothèses fortes pour simplifier le modèle, puis en second lieu en relaxant ces hypothèses pour s'aligner avec la sémantique du langage de programmation parallèle sous-jacent.

- Implémenter une plateforme de démonstration complète pour motiver l'intégration de cette extension dans le standard du langage sous-jacent, en fournissant:

  - un compilateur capable de détecter les applications streaming régulières et de générer, le cas échéant, un code au même niveau d'optimisation que l'on pourrait obtenir en compilant le programme écrit dans l'un des langages de streaming plus restreints,
  - un système de runtime efficace, en particulier pour l'implantation des synchronisations,
  - des résultats expérimentaux validant l'efficacité de cette approche pour les applications streaming régulières, afin de montrer qu'il n'y a pas de conséquences négatives à adopter ce type de langage plus général pour la programmation streaming, ainsi que des résultats qui montrent l'utilité de cette extension pour des applications qui ne seraient pas programmables dans les langages de streaming plus classiques ou pour l'exécution en mémoire distribuée.

Ces objectifs étant trop ambitieux pour être entièrement atteints dans le cadre de cette thèse, nous avons décidé d'en limiter la portée. Nous laisserons les considérations d'exécution sur des systèmes à mémoire distribuée comme opportunité de recherche ultérieure et nous prendrons certains raccourcis dans l'implantation du prototype de compilateur, où nous réutiliserons, dans la mesure du possible, les fonctionnalités disponibles dans le compilateur GCC.

Organisation de cette thèse

Cette thèse est structurée en neuf chapitres. Le chapitre 2 présente notre extension au langage OpenMP permettant d'exprimer le flôt de données entre les tâches. Nous y décrivons, à travers un grand nombre d'exemples, les nouvelles constructions syntaxiques, qui s'intègrent avec les autres constructions du langage. Nous détaillons la sémantique de cette extension et montrons comment elle permet d'introduire le streaming de manière incrémentale dans OpenMP.

Afin de prouver les propriétés les plus importantes dont bénéficie cette extension, telles que le déterminisme d'interblocage et fonctionnel, la sérialisabilité et des garanties d'absence d'interblocage sous certaines conditions, nous développons et décrivons, dans le chapitre 3, un nouveau modèle de calcul Control-Driven Data-Flow (CDDF). Dans une première section, nous introduisons les notations et la caractérisation de certaines propriétés à l'égard de notre formalisme. Dans les sections suivantes, nous étudions les propriétés des programmes streaming qui peuvent être représentés dans le modèle CDDF

et nous prouvons des conditions suffisantes, et dans la mesure du possible décidables statiquement, pour l'absence d'interblocage et pour la sérialisabilité.

Dans le chapitre 4, nous affaiblissons les hypothèses faites dans le modèle CDDF présenté au chapitre 3, afin de réduire l'écart entre ce modèle abstrait et les contraintes plus pragmatiques de l'exécution de programmes en mémoire bornée ou de la nécessité de paralléliser le programme de contrôle.

Nous analysons notre extension du langage OpenMP dans la perspective de sa sémantique CDDF dans le chapitre 5. Nous y identifions les analyses statiques nécessaires à la génération de code optimisé et montrons comment les analyses de flôt de données existantes peuvent être adaptées aux besoins de programmes rentrant dans le cadre du modèle CDDF, évitant ainsi de développer un ensemble d'analyses spécifiques à ce modèle.

Le chapitre 6 décrit notre système de runtime. La première section propose une solution pour la synchronisation des streams et présente notre algorithme de synchronisation, ne faisant pas appel aux opérations atomiques ou aux barrières mémoire, avec une discussion détaillée sur les implications du modèle de consistance mémoire fourni par l'architecture sous-jacente. Les deux sections suivantes sont dédiées aux algorithmes d'ordonnancement de tâches, d'équilibrage de charge et de détection d'interblocage. L'interface cible pour la génération de code est décrite dans la quatrième section.

La passe de génération de code pour notre extension streaming d'OpenMP, dont nous avons implémenté un prototype dans le compilateur GCC, est présentée au chapitre 7. Les deux premières sections y décrivent la technique d'expansion des pragmas non optimisée, sur laquelle nous nous rabattons en cas d'échec des analyses statiques, et une troisième section présente la génération de code optimisé.

Nos résultats expérimentaux sont réunis au chapitre 8.

Nous concluons cette thèse dans le chapitre 9.

Chapter 1

Introduction

The design complexity and power constraints of large monolithic processors forced the industry to develop Chip-Multiprocessor (CMP) architectures [2,37,39,61], where multiple processor cores are tiled on a single chip. As the number of transistors per chip continues to experience an unabated exponential growth, the prevailing trend during the past decade has been towards providing an increasing number of simpler, more power-efficient and often slower cores per chip. The performance of single-threaded applications is therefore expected to stagnate or even decline with new generations of processors. Increased performance returns on new architectures are contingent on the amount of parallelism that can be efficiently exploited in applications, thanks to parallel programming or parallelizing compilers. However, uncovering raw parallelism is insufficient. Even massively parallel applications are not guaranteed to experience speedups linear with the number of cores and to scale in a portable way, an essential property with the fast obsolescence of current architectures.

The efficient parallelization of an application presents two fundamental problems. The first problem is to expose, or uncover, parallelism in the computation. Though a widely studied problem, it has proven to be extremely hard to solve, both in the form of support for programmers developing parallel applications and in the form of automatic translation of sequential code into parallel code. The former is expensive in terms of development and often leads to either portability or performance portability issues, while the latter appears to present an unsurmountable challenge to optimizing compilers: the choice of a performance improving code transformation, let alone an optimal transformation, for a given machine model.

The second problem is to exploit the parallelism on a specific platform and scalably improve performance. The most acute part of this problem, known as the memory wall, is a pre-existing condition. While uniprocessor architectures are strongly impacted by high memory latency, multiprocessors also experience severe bandwidth insufficiencies. Complex memory hierarchies and relaxed memory consistency models have been developed to deal with memory latency and bandwidth issues, but this is insufficient if a host of cores vie for the limited off-chip memory bandwidth.

1.1 Motivation

As architectures become increasingly difficult to exploit efficiently, because of their rapidly evolving complexity and their low ratio of off-chip memory bandwidth available per instruction, high-level languages are designed to express (in)dependence, communication patterns and locality without reference to any particular hardware. The compilers and runtime systems are responsible for lowering these abstractions to well-orchestrated threads and memory management. In particular, stream programming has recently attracted a lot of attention. It is a widely applicable form of parallel programming that guarantees functional determinism[1], a major asset in the productivity race. It is also conducive to making relevant data-flow explicit and to structuring programs in ways that allow simultaneously exploiting pipeline, data and task parallelism. Stream computations also help reduce the severity of the memory wall in two complementary ways: (1) decoupled producer/consumer pipelines naturally hide memory latency; and (2) they favor local, on-chip communications, bypassing global memory and therefore reducing the pressure on the memory subsystem. Thus streaming appears as a natural choice for approaching the problems raised by the memory wall.

In its various forms, streaming has been around for almost half a century, though the purpose of organizing computation around streams of data has varied along the way. In his survey [67] of stream processing, Stephens reviews the history of the development of streaming, from the design of semantic models formalizing stream computations to the analysis of data-flow, signal processing, reactive systems or design and verification of hardware. He classifies stream processing systems based on three criteria: synchrony, determinism and the type of communication channel. Fundamentally, stream-based models of computation all share the same structure, which can generally be represented as a graph, where computing nodes are connected through streaming edges.

Kahn Process Networks (KPN) [36] is one of the first, and most influential, models. Data-driven processes communicate through unbounded FIFO channels, which means that write operations are non-blocking while read operations will wait until sufficient data is available. As the processes are deterministic, a key property of this model is that the network as a whole is also functionally deterministic by composition. However, cyclic networks can lead to deadlocks, which has spurred the development of a restricted form of Kahn Process Networks, Synchronous Data-Flow (SDF) [43]. While processes in KPNs execute asynchronously and can produce or consume variable amounts of data, SDF processes have a statically defined behaviour. With rates of production and consumption known at compile time, it is possible to statically decide whether the execution is free of deadlocks and to statically schedule the execution. It can also guarantee the absence of resource deadlocks when executing on bounded memory, a realistic restriction. Extensions like Cyclo-Static Data-Flow [13] relax some of the constraints. StreamIt [68] is a recent instantiation of the synchronous data-flow model, building on the strong static restrictions of the underlying model to enable aggressive compiler optimizations, it achieves excellent performance and performance portability across a variety of targets [26] for a restricted set of benchmarks that properly map on this model.

[1] A program is functionally deterministic if it evaluates a function of its inputs, in other words, if its output is entirely determined by its inputs.

Synchronous languages, including LUSTRE [28], Signal [27] and Esterel [12], have been very successful in industrial applications, due to their determinism, deadlock-freedom and safety properties. Based on the reactive systems or the signal processing networks models, with the distinction that reactive systems rely on bidirectional communication channels, these languages do not require periodicity. Processes responding instantaneously communicate through signals, also used to define a notion of time and causality. Signals differ from streams in that they are sampled rather than consumed.

Data-flow languages have also been developed, where the execution is explicitly driven by data dependences rather than control flow. The connection between these languages and streaming comes from the focus on data flow dependences, which represent producer-consumer relationships. The Lucid [7] language relies on the `next` keyword within loops to achieve a similar effect to advancing in a stream of data, by consuming or producing in a channel, or, in the synchronous languages domain, to the advancement of clocks on signals. Lucid programs are fundamentally systems of recursion equations. The Sisal [25] functional language explicitly introduces the notion of stream, which is naturally very close to lists. If stream processing systems are understood as the parallel implementation of stream transformers, which is the functional interpretation of a process network mapping a set of input streams to a set of output streams, then any functional language can be used for stream programming. This corresponds to the lazy interpretation of functional languages; see [18] for a Haskell [32] implementation of Lucid Synchrone [17].

Communicating Regular Processes (CRP) [23] relies on a non-FIFO communication scheme, where processes write in channels at specific indexes with a Single Dynamic Assignment property. Consumers can only read channels at indexes that have previously been written, therefore enforcing flow dependences at the element level. While much more flexible than Synchronous Data-Flow models, this approach would be difficult to implement efficiently as a streaming language. Program properties, in this model, can only be statically determined when relations are affine. The goal, however, is to isolate the scheduling requirements of processes in order to reduce the size of the scheduling problem.

While all of these diverse approaches to stream programming have the potential to help mitigate the memory wall, they only apply to restricted classes of applications. Stream programs are generally considered built around regular streams of data, which fits the models where channels of communication are implemented as single-producer and single-consumer FIFO queues. If streaming is to take center stage in the development of applications for current and upcoming multi- and many-core architectures, a more general-purpose model is necessary, where communication patterns are not always regular and statically defined, but can occur and be exploited dynamically, depending on program execution. The insight that the flow of data plays a central role in all programs is not flawed, but data flow often needs to be predicated by complex control flow due to irregular events, as is the case in synchronous programs.

A general-purpose parallel programming model must express all forms of parallelism, including the most general form, control parallelism, while providing the compiler with sufficient information to generate streaming code. Task parallelism is not always the best answer for performance, but if the data flow between tasks is known, the communication can be implemented with streams and many powerful optimizations can be enabled. When

the compiler discovers that, for example, the SDF invariants are satisfied, it triggers aggressive loop transformations to adapt the task parallelism to the target, matching the performance portability of synchronous data-flow based languages, as is the case in StreamIt.

Importantly, new approaches to streaming should try to preserve the strong properties provided by some of the existing models, like functional determinism or deadlock-freedom. Debugging concurrent applications is a daunting task, orders of magnitude more complicated than for sequential programs, mostly because of the non-deterministic nature of races and deadlock-related errors. If reproducibility is elusive, testing becomes statistical and the identification of errors difficult when all debugging tools impact the behaviour of programs.

However, developing yet another new language is wasteful in terms of programmer expertise and requires building new compiler and debugging tool-chains. Yet new languages do not shield programmers from the necessity of mixing different programming styles and constructions. While many languages have been designed to exploit stream parallelism, we believe it is more interesting and sufficient to introduce minimal and incremental additions to an existing and well-established language, leveraging the interaction between streaming and the existing, proven, sequential and parallel constructs.

This thesis explores streaming as a general-purpose parallel programming paradigm, rather than a model dedicated to a class of applications, by providing a highly expressive stream-computing extension to a de facto standard for shared memory programming, OpenMP.

1.2 Statement of Purpose

Our overarching goal is to develop and promote the use of streaming as a model for general purpose parallel programming. To this end, we identify five research axes:

1. Develop streaming within a well-established and proven parallel programming language, to allow mixing stream programming with other parallel constructs in a controlled and integrated environment. This mitigates the language barrier, with respect to programmers, and allows reusing existing widely-available compilation frameworks;

2. Relax as many of the restrictions usually attached to stream programming, such as synchrony, periodicity or regularity, as possible. These restrictions reduce the applicability of streaming to a few classes of applications;

3. Show that it is possible to preserve the useful properties of streaming models, such as determinism or efficiency, notwithstanding the relaxation of the aforementioned restrictions;

4. Provide debugging support, for example with a deadlock detection scheme;

5. Support streaming on shared memory as well as on distributed memory. New architectures adopt relaxed memory models to reduce the cost of cache coherence

across an increasing number of cores, but scaling up (i.e., adding resources to a single node) is more limited than scaling out (i.e., increasing the number of nodes).

The means towards this end include: (1) developing a semantical model for the streaming extension, first with strong simplifying assumptions, and then generalized to fit the semantics of the underlying parallel programming language, and (2) implementing a full demonstration framework to motivate the integration of the streaming extension in the language standard. The latter means further requires providing: (a) a compiler that can detect regular streaming applications and optimizes them as would the compilation of more restrictive streaming languages, (b) an efficient runtime, and (c) experimental results that validate the efficiency of the framework for regular streaming applications, to show that there is no down-side in adopting this more general streaming language, and that also validate the usefulness of the extension for non-streaming applications or for execution on distributed memory systems.

As these objectives cannot be reasonably achieved within the framework of this thesis, we had to add some limitations to the scope, in particular by leaving the distributed memory considerations for future work, and take some shortcuts, in particular for the implementation of the compiler prototype where we reuse as much as possible what is available in the GCC compiler.

1.3 Thesis Outline

This thesis is organized in nine chapters. Chapter 2 presents our extension to the OpenMP language to enable expressing data flow between tasks. We explain the additional syntax, seamlessly integrated with the existing constructs, on a large set of coding patterns. We further detail the semantics of our extension and show how it incrementally enables stream-computing in OpenMP.

To prove that our streaming extension benefits from important properties, like deadlock and functional determinism, serializability and some form of deadlock-freedom, a new formal framework is developed in Chapter 3, the Control-Driven Data-Flow (CDDF) model of computation. The first section introduces the notations and characterizes a set of program properties with respect to our formalism. In the following sections, we investigate the properties of streaming programs that fit the CDDF model and prove sufficient, and when possible statically decidable, program conditions for different classes of deadlock-freedom and serializability.

Chapter 4 weakens the assumptions made in the original model, partially bridging the gap between the abstract model and the more pragmatic constraints of executing in bounded memory or parallelizing the control code.

We analyze our OpenMP extension in the perspective of its CDDF semantics in Chapter 5. After mapping the extension onto the CDDF model, we discuss the static analysis requirements for generating optimized code. We detail the way existing data-flow analyses can be adjusted to fit our needs, rather than developing a new ad hoc static analysis framework.

Chapter 6 presents our runtime system. The first section addresses the issue of the synchronization of streams and presents our atomic operation-free and memory fence-free

algorithm, with a strong emphasis on memory consistency issues. The second and third sections focus on scheduling, load-balancing and deadlock detection algorithms. The API we provide as a target for code generation is detailed in the fourth section.

Chapter 7 describes the code generation pass, implemented as a prototype in GCC, for our stream-computing extension. The first two sections provide the default expansion scheme, which serves as a fall-back strategy when all static analyses fail, and a third section presents the optimized code generation.

Our performance experiments are gathered in Chapter 8.

We conclude in Chapter 9.

Chapter 2

A Stream-Computing Extension to OpenMP

In this chapter, we introduce an extension to OpenMP3.0 enabling stream programming with minimal, incremental additions that integrate into the current specification while preserving backward compatibility. The stream programming model decomposes programs into tasks and makes the flow of data explicit among them, thus exposing data, task and pipeline parallelism. It helps the programmers to express concurrency and data locality properties, avoiding non-portable low-level code and early optimizations. We survey the diverse motivations and constraints converging towards the design of our simple yet powerful language extension, and we detail the semantics of the extension as well as the interaction with current OpenMP semantics.

Dans ce chapitre, nous présentons une extension au langage OpenMP permettant la programmation streaming avec un ensemble d'additions minimes et qui s'intègrent entièrement à la spécification actuelle, sans impact pour la compatibilité des programmes existants. Le modèle de programmation par streams décompose les programmes en tâches et rend explicite le flôt de données entre celles-ci, permettant ainsi d'exposer à la fois du parallélisme de données, de tâches et de pipeline. Ce modèle aide les programmeurs à exprimer les propriétés de concurrence et de localité de données des programmes, en évitant les constructions de bas niveau, non-portables, et les optimisations précoces. Nous détaillons les diverses motivations et contraintes qui ont dirigé nos choix dans l'élaboration de cette extension de langage et nous en décrivons la sémantique, ainsi que les interactions avec la sémantique des constructions OpenMP existantes.

2.1 Background

The performance of single-threaded applications is expected to stagnate with new generations of processors. Improving performance requires changing the code structure to harness complex parallel hardware and memory hierarchies. But this is a nightmare for programmers: translating more processing units into effective performance gains involves a never-ending combination of target-specific optimizations. These manual optimizations

involve subtle concurrency concepts and non-deterministic algorithms, as well as complex transformations to enhance memory locality. As optimizing compilers and runtime libraries no longer shield programmers from the complexity of processor architectures, the gap to be filled by programmers increases with every processor generation.

High-level languages are designed to express (in)dependence, communication patterns and locality without reference to any particular hardware, leaving compilers and runtime systems with the responsibility of lowering these abstractions to well-orchestrated threads and memory management. In particular, stream programming has recently attracted a lot of attention. It is a widely applicable form of parallel programming that guarantees functional determinism, a major asset in the productivity race. It is also conducive to making relevant data-flow explicit and to structuring programs in ways that allow simultaneously exploiting pipeline, data and task parallelism. Stream computations also help reduce the severity of the memory wall in two complementary ways: (1) decoupled producer/consumer pipelines naturally hide memory latency; and (2) they favor local, on-chip communications, bypassing global memory. While many languages have been designed to exploit pipeline parallelism, we believe it is more interesting and sufficient to introduce minimal and incremental additions to an existing and well-established language.

This chapter introduces an extension to the OpenMP3.0 language [69] to enable stream programming. It requires only minor additions that integrate in the current language specification without impacting the behaviour of existing OpenMP programs.

The remainder of this chapter is structured as follows. Section 2.2 discusses our motivation and related work. Section 2.3 details the analysis driving the design of our streaming extension. Section 2.4 presents the extension itself. Section 2.5 defines the semantics of the new constructs and validates the execution model. Section 2.6 further analyzes the interaction of this extension with the existing OpenMP specification. Finally, in Section 2.7 we introduce a function prototype annotation scheme for enabling modular compilation of streaming programs.

2.2 Motivation and Related Work

Many languages and libraries are being developed for the stream-computing model. Some are general purpose programming languages that hide the underlying architecture's specificities, while others are specifically targeted at accelerator architectures. While a complete survey is outside the scope of this thesis, we present a selection of the most related efforts in this field. We also discuss the motivations and constraints that drive our proposal.

Data-parallel execution puts a high pressure on the memory bandwidth of multi-core processors. There is a well known tradeoff between synchronization grain and private memory footprint, as illustrated by performance models for bulk-synchronous data-parallel programs [14]. But there are few answers to the limitations of the off-chip memory bandwidth of modern processors. Pipeline parallelism provides a more scalable alternative to communication through main memory, as the communication buffers can be tailored to fit in the caches, effectively making cores communicate through a shared cache or through the cache coherence protocol.

Furthermore, a stream-programming model naturally exposes data, task and pipeline parallelism through its high-level semantics, avoiding the loss in expressiveness of other parallel-programming models. A stream computation is divided in pipeline stages, or filters, where the producer-consumer relationships are explicit. Stages can be either sequential, if there is a dependence between successive executions of the stage, or parallel, in which case, the stage can be replicated at will for load balancing and/or exploiting data parallelism. The sequential filters are an issue that is shared with other forms of parallelism as it stems from the presence of state in the filter, or equivalently from a loop-carried dependence. Closely related to pipelining, doacross parallelization [19], can be used in such cases, but it is more restrictive and requires communication of the state between threads.

Because of this problem, the use of pipelining can be the only efficient solution for parallelizing some applications. As an example, the recent study [51] by Pankratius et al. of the parallelization of Bzip2 shows that this application is not only hard to parallelize, but more specifically that only pipelining allows it to be efficient and to achieve decent scalability levels. The authors of the study remark that OpenMP is not well suited for parallelizing Bzip2, but this was reversed by implementing FIFO queues to communicate between tasks, making it one of the best choices. We will elaborate on this aspect in the experimental chapters. Our objective is to show that it is neither necessary to develop a new language for streaming, nor to require developers to write the pipelining code by hand.

The StreamIt language [68] is an explicitly parallel programming language rooted in the Synchronous Data Flow (SDF) model of computation [43]. StreamIt provides three interconnection modes: the Pipeline allows the connection of several tasks in a straight line, the SplitJoin allows for nesting data parallelism by dividing the output of a task in multiple streams, then merging the results in a single output stream, and the FeedbackLoop allows the creation of streams from consumers back to producers. The channels connecting tasks are implemented either as circular buffers, or as message passing for small amounts of control information. Thanks to these static restrictions (periodicity, split-join structure), a single StreamIt source can be compiled very efficiently on a variety of shared and distributed memory targets [26]. But we believe these expressiveness restrictions are not necessary to achieve excellent performance, assuming the programmer is willing to spend a minimal effort to balance the computations and tune the number of threads to dedicate to each task manually. This is the pragmatic approach OpenMP has successfully taken for years. In addition, when the compiler discovers that the SDF invariants are satisfied, it may trigger aggressive loop transformations to adapt the task parallelism to the target, matching the performance portability of StreamIt.

The Brook language [15] provides language extensions to C with Single Program Multiple Data (SPMD) operations on streams. Unlike StreamIt, it is control-centric, with control flow operations taking place at synchronization points. Streams are defined as collections of data that can be processed in parallel. For example: "`float s<100>;`" is a stream of 100 independent floats. User defined functions that operate on streams are called kernels. The user defines input and output streams for the kernels that can execute in parallel by reading and writing to separate locations in the stream. Brook kernels are blocking and isolated: the execution of a kernel must complete before the next kernel

can execute. This is the same execution model that is available on graphics processing units (GPUs): a task queue contains the sequence of shader programs to be applied on the texture buffers.

Another interesting approach to generate the data transmission towards the accelerator boards is that of the CAPS enterprise: codelets are functions [60] whose parameters can be marked with input, output or inout. The codelets are intended to be executed remotely after the input data has been transmitted.

Dynamic data-flow principles [6,21,70] have regained popularity as pragma-based extensions to imperative languages. Based on CellSs [10], StarSs [53] defines a complete set of pragmas to program distributed-memory and heterogeneous architectures; it supports both data-flow and control-flow programming styles. SMPSs is one of the StarSs incarnations for shared-memory targets [46]. TFlux follows a similar approach [66], focusing on data flow and targeting the Data-Driven Multithreading (DDM) execution model [41]. StarSs and TFlux are closely related to streaming languages, but they differ from our approach in two fundamental aspects:

- their design and implementation assume a short-lived task execution model, relying on data-driven scheduling of lightweight user-level threads and work stealing; this is excellent for load-balancing, but induces significant overheads for finer grain tasks, as our experiments confirm;
- they are not compatible with OpenMP, but introduce other pragmas with different semantics; StarSs handles distributed memory and heterogeneous targets, unlike OpenMP, but is not as expressive on shared-memory targets; TFlux is currently restricted to nested loops.

Closest to our work is the Streaming Programming Model of the ACOTES project [16]. This model takes its inspiration from the OpenMP3.0 tasks, but is not compatible with OpenMP. It adds decoupled communication channels and pioneered the *persistent interpretation of tasks*, among other contributions. Our proposal derives from this experimental platform, but it is more expressive, it achieves a complete and incremental integration within OpenMP.

2.3 Design Goals of the Extension

Our primary design goal is to enable OpenMP programmers to exploit pipeline parallelism without explicitly handling communication and synchronization, which is both error-prone and time-consuming. We also want to offer highly efficient decoupled pipelined executions to programmers with no experience in shared-memory concurrency. To achieve these goals, we propose extensions to the OpenMP language, exposing the necessary information for the generation of pipelined parallel code, while ensuring this additional expressiveness does not introduce excessive complexity and does not break the semantics of the current specification.

More specifically, we deem the three objectives of expressiveness, efficiency and simplicity to be the most important ones to enable stream programming in OpenMP. In Section 2.4, we will show how to achieve these goals.

Expressiveness

The extension aims to provide a way for programmers to expose producer-consumer relationships. The current OpenMP specification lacks the capability to make the flow of data explicit, as the existing sharing clauses only allow shared and private data to be distinguished. In order to use task constructs in non-embarrassingly parallel problems, manual synchronization is required to communicate through shared memory.

The convenience of "peek" operations and non-unitary production and consumption rates is often provided in streaming frameworks. The manual implementation is cumbersome enough to deserve a simplified mechanism at the language level, which in addition allows the compiler to generate in-place operations in stream buffers, avoiding the explicit state management and the spurious dependences it induces, and avoiding the overhead of copying from streams to local buffers and back.

We also consider dynamic taskgraphs and split-join patterns to be important in order to exploit non-streaming applications, that present complex dependence patterns that do not fit the static producer-consumer model.

Productivity

One of the drawbacks of new stream-programming languages is that they come with a whole new programming environment, which lacks debugging support, interoperability and mature accompanying libraries. To this startup cost, one should often add the cost of target-specific tweaking required by the lower level languages (e.g., hard-wired offloading directives for accelerators). By extending a well-known parallel-programming language, OpenMP, with incremental, natural and target-independent constructs for streaming, the programmer's productivity is maximized. We believe backward compatibility is essential to avoid making the extension an additional burden on current developers. They should not change the way they are used to work with OpenMP for non-streaming applications. The semantics of the extension should therefore be incremental and any new interaction should, as far as possible, follow the existing rules. This practical constraint turns into a research challenge: building compositional data-flow constructs over an existing imperative, shared-memory semantics.

Efficiency

The execution model of OpenMP3.0 specification tasks is similar to that of coroutines or fibers. A task instance is created whenever the execution flow of a thread encounters a task construct, but the execution of the newly created task is contingent on the cooperative scheduling policy. No ordering of the execution of tasks can be assumed. Such an execution model is well suited for unbalanced loads, but the overhead of creating and scheduling tasks is significantly higher than synchronizing persistent tasks.[1] There is a strong case for relying on persistent tasks rather than on data-driven scheduling of short-lived tasks. We substantiate this claim in Chapter 6 where we compare the overhead incurred by our synchronization algorithm on persistent tasks with the overhead of scheduling short-lived tasks.

[1]Except when the target architecture offers some support for very lightweight thread scheduling [41].

2.4 Proposed Streaming Extension

This section introduces the syntactic constructs, as well as the modifications to the current OpenMP specification, we need to meet the design goals outlined in Section 2.3. The OpenMP specification is provided as an annotation scheme for sequential programming languages, where the underlying language can be any of C, C++ or Fortran. We generally use a C syntax although everything can easily be transposed to Fortran or C++.

2.4.1 Definitions

Let us first define a few terms we commonly use in the remainder of this thesis.

Stream A stream is an infinite sequence of typed elements, that can also be understood as a typed communication channel. Any type defined in the underlying program (scalar, structure or array) can be used as a base type for a stream. The type of each data element in the stream is the base type of the stream.

Horizon The horizon of a task in a stream is the amount of data that can be accessed by the task's activation in that particular stream. It represents the number of elements in the stream that can be accessed by the task at one time.

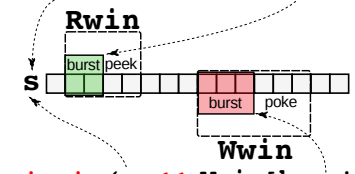
Burst The burst of a task in a stream is the number of elements effectively "consumed" or produced in the stream by the execution of one activation of the task.

Window A window is a stream accessor. It must have a compatible type with the stream it connects to, which is either the type of the stream's elements or an array of elements of this type. In the latter case, the window is accessed with a subscript in the range $[0 : horizon - 1]$.

2.4.2 Syntactic Extension to the OpenMP Language

```
input/output (list)
    list   ::= list, item
             | item
    item   ::= stream
             | stream >> window
             | stream << window
    stream ::= var
             | array[expr]
    expr   ::= var
             | value
```

```
int s, Rwin[Rhorizon];
int Wwin[Whorizon];
input (s >> Rwin[burstR])
```




```
output (s << Wwin[burstW])
```

Figure 2.1: Syntax for input and output clauses.

In order to make the data flow explicit between tasks and thus allow data dependence relations to be expressed between them, we propose to extend OpenMP3.0 with two additional clauses for task constructs, the input and output clauses presented on Figure 2.1. Both clauses take a list of items, each of which describes a stream and its behaviour w.r.t. the task to which the clause applies.

In the abbreviated item form, `input (stream)`, the stream can only be accessed one element at a time through an implicit window that uses the same name as the stream. In a more complete form, `input (stream >> window[burst])`, the programmer uses the C++ flavoured `<< >>` stream operators to connect a `window` to a stream, gaining access, within the body of the task, to `horizon` elements in the stream. The specification of `burst` is optional; if absent, in the case of the full syntax relying on the stream operators, it is considered to be equal to the horizon, which is the declaration size of the window, by default. Of course, in the simplified syntax, where we use no stream operators and an implicit window, the burst is always 1.

Though we also make some further adjustments to the semantics of existing constructs of the OpenMP language, this is the single syntactic modification required to provide streaming. We use this additional syntax to enable stream communication between tasks.

2.4.3 Stream Communication Between Tasks

Any variable (scalar or array) can be used as a stream by any task in the variable's scope. This is fully transparent for programmers: the compiler is responsible for allocating memory and for indexing stream buffers, thus converting the user-level stream access window to a view [2] inside the stream buffer. The horizon of a task in a stream is the number of contiguous stream elements the task can access at any point of its execution, and by extension it also denotes the size of the stream access window. The burst rate of a task is the number of stream elements that are consumed or produced by each execution of the task, therefore representing the number of elements the stream view needs to slide after the execution of the task.

Our programming model is more general than *scalar* data-flow: tasks compute *streams of values* and not individual values. To the programmer, streams are plain C variables, transparently expanded into streams by the compiler. An array declaration (in C) defines the sliding access window, which is directly accessible within the task, and its declaration size defines the *horizon*. Connecting a window to a stream in an input or output clause allows to indicate the *burst*, which is the number of elements by which the corresponding stream view is shifted after each activation. In Figure 2.1 the input window `Rwin` is shifted by two elements, while the output window `Wwin` is shifted by three elements. Scalar data-flow corresponds to `horizon` = `burstR` (resp. `burstW`). In the more general case where `horizon` > `burstR` (resp. `burstW`), the window elements beyond the burst are accessible to the task; for an output window, the values of these elements will only be committed and made accessible to consumers in subsequent activations. Task activation requires the availability, on each input stream, of *all* `horizon` elements in input windows.

The examples on Figure 2.2 illustrate the syntax of the input and output clauses for some common use-cases. On the left, the first task produces data on stream `x` without specifying a window for accessing the stream. An implicit window is provided that uses

[2] The stream view is a notion that does not directly concern programmers. A view is a range of elements in a stream that can be accessed by a task. It is very similar to the window and shares the same horizon. However, the window is a user-defined accessor to the stream, that is accessed by indices in the range $[0 : horizon - 1]$, while the view is the result of the mapping of a window on a stream buffer. We will elaborate on this in Chapters 3 and 7.

```
int x; // The stream
int v; // The access window

#pragma omp task output (x)
  {
    // Implicit window "x" on stream x
    // burst == horizon == 1

    x = ...;
  }

#pragma omp task input (x >> v)
  {
    // Explicit window "v" on stream x
    // burst == horizon == 1

    ... = ... v ...;
  }
```

```
int y;    // The stream
int V[3]; // The access window

#pragma omp task output (y)
  {
    // Implicit window "y" on stream y
    // burst == horizon == 1

    y = ...;
  }

#pragma omp task input (y >> V[1])
  {
    // Explicit window "V" on stream y
    // burst == 1, horizon == 3

    ... = ... V[0] ... V[2];
  }
```

Figure 2.2: Examples of input and output clauses.

the name of the stream, `x`, with implicit unitary burst and horizon values. The second task on the left consumes data from stream `x`, but uses an explicit scalar window, `v`. The use of the stream name within the task is therefore forbidden and all accesses to the stream must happen through the stream access window. In this case, the programmer did not specify the burst, which is then implicitly evaluated to be the same as the horizon, 1. Note the direction of the stream operator `>>` , that behaves exactly as the common C++ stream operator. The programmer perceives a copy semantics from stream `x` to the window `v`, but, as we discuss later, the compiler transparently replaces the window accesses with a sliding window, the view, directly inside the stream buffer, therefore incurring no overhead.

On the right of Figure 2.2, we show the behaviour of array windows, where the horizon is greater than one. The first task on the right is identical to the first task on the left, but the second task uses the access window `V` to access the stream `y`. This means that the task will not execute as long as there are less than `horizon` (here 3) values available on stream `y`. The stream operator `>>` connects the window `V` to the stream `y`. It provides access, within the task, to the next 3 elements in stream `y`, through the window `V` which can be indexed as `V[0] .. V[2]`. Once again, these accesses are directly translated to accesses in the proper place inside the stream buffer, and the read view corresponding to this window is shifted by `burst` elements (here 1) once the task completes. Any subsequent task with an input clause on this stream will only see the remainder of the values.

Note that the window array used to access the stream does not need to be allocated and will altogether be removed by the compiler and replaced with direct accesses to the stream buffer. Its syntactic presence is motivated by compatibility reasons and to ensure the code can compile and execute even if the OpenMP annotations are omitted. It also helps to improve code readability as it avoids overloading the stream symbol. Of course the validity of the sequential code is doubtful in this example and this highlights a more

general isue with respect to the serialization of streaming programs, which we address in Section 3.7. We prove conditions under which serialization can be achieved by ignoring the pragmas, as well as a more general condition under which serialization is statically possible with some additional code generation work. In the general case, ignoring the streaming annotations does not yield semantically equivalent code.

```
int S[K];      // Array of streams
int X[horizon]; // Access window


#pragma omp task \
         output(S[0] << X[burst])
  {
    // burst <= horizon
    for (i = 0; i < burst; ++i)
      X[i] = ...;
  }


#pragma omp task \
         input(S[1] >> X[burst])
  {
    // burst <= horizon
    for (i = 0; i < horizon; ++i)
      ... = ... X[i];
  }
```

```
int A[5];   // Stream of arrays
int V[2][5]; // Window on stream of arrays

#pragma omp task output(A)
  {
    // Each element is an array
    // burst == horizon == 1

    for (int i = 0; i < 5; ++i)
      A[i] = ...
  }

#pragma omp task input(A >> V[1])
  {
    // burst = 1, horizon = 2
    for (int i = 0; i < 5; ++i)
      ... = ... V[0][i];
    for (int i = 0; i < 5; ++i)
      ... = ... V[1][i];
  }
```

Figure 2.3: Using arrays of streams and streams of arrays.

The examples on Figure 2.3 illustrate the more complex syntax of using arrays of streams, on the left, and how it differs from streams of arrays, on the right. The possibility of using arrays of streams is very important as it brings a dynamic dimension to the pipelines that can be expressed. Without this notion, only static taskgraphs can be expressed in our model.

On the left hand side, we first declare two arrays, one that will be used as an array of streams of integer values, `S`, and the second will be used as an access window of size `horizon`. The first task on the left produces data on the first stream in the array and uses the window `X` to access the stream `S[0]` within its body. The second task consumes data from the same array of streams, but on stream `S[1]` instead. The two tasks are not connected to the same stream.

The right side of Figure 2.3, illustrates the syntax for using streams of arrays, which means that the base element of the stream is an array. The first task produces data on stream `A` using the implicit access window syntax. The task's horizon and burst on this stream are both unitary, though as the body of the task shows, it corresponds to arrays of 5 integers. The bottom right task shows the way windows can be used to access streams of arrays. The window is declared as a higher dimension array on the base type of the stream: the base type being an array type, a second dimension is added in front of the

base array type declaration. The rest of the syntax is identical to the scalar case once the type of the stream's elements is abstracted. Note that the type of the window must always be compatible with the stream base type; in this case, for instance `V[0]` has the same type `int [5]` as the stream. The same can be used to handle streams of arrays of any dimension or streams of structures.

```
int X[3]; // An array of streams
int x;    // A scalar window
int V[2]; // Array window

#pragma omp task output (x >> X)
  {
    // Explicit window "x" on array of streams X
    // burst == horizon == 1
    x = ...;
  }

#pragma omp task input (X[0] >> x)
  {
    // Explicit window "x" on stream X[0]
    // burst == horizon == 1
    ... = ... x ...;
  }

#pragma omp task input (X[1] >> V[2])
  {
    // Explicit window "V" on stream X[1]
    // burst == horizon == 2
    ... = ... V[0] ... V[1];
  }
```

Figure 2.4: Broadcasting an `output` window over an array of streams.

Finally, Figure 2.4 shows how data produced by a task can be broadcast over an array of streams. The first task connects, in its `output` clause, the scalar window `x` to the array of streams `X`. From the programmer's perspective, the data stored in the window is copied to each stream in the array. The second task consumes data from the stream `X[0]`, while the third task consumes from `X[1]`. For this reason, both tasks will see the same sequence of values and their execution is independent from one another. In practice, there is no need to replicate the streams and we implement this behaviour without incurring any copy overhead.

2.4.4 Stream Communication with the Enclosing Context

The term *enclosing context* is commonly used in the context of the OpenMP language to refer to the code immediately nesting a given OpenMP construct, such as a task.

We do not allow the syntax for communication between tasks to be used to communicate with the enclosing context, because it results in ambiguous communication patterns

and, in many cases, would not allow data produced within a task to be used by the outer context immediately after the task construct. Another reason is that all the necessary constructs for enabling communication with the enclosing context are already present in OpenMP.

The `firstprivate` and `lastprivate` clauses provide the proper semantics, but in OpenMP3.0 only the `firstprivate` clause is allowed on task constructs. Importantly, the reason that lastprivate clauses are not supported on task constructs in the OpenMP specification is not based on semantical grounds, but rather on pedagogical grounds. Indeed, the use of lastprivate would be misleading to programmers and of little use in the current OpenMP model as it would result in blocking the main program until the task construct bearing the lastprivate clause terminates. We illustrate this behaviour on Figure 2.5, where the first task is rendered useless, under current OpenMP semantics, by the presence of the implicit synchronization point required for the lastprivate clause. However, in our case this clause is not quite as useless, and we will discuss, in Section 2.4.5, an example substantiating the necessity of the lastprivate clause's semantics for our extension.

We therefore need to amend the specification to allow `lastprivate` clauses on OpenMP tasks. The syntax and semantics of both clauses are left unchanged, the firstprivate clause corresponding to a privatizing copy of the variable it applies to, and the lastprivate clause corresponds to a private memory where the last state is copied-out, upon completion of the OpenMP task, to the main program.

In the case of firstprivate and lastprivate clauses, the notions of window, horizon and burst are irrelevant as the semantics are fundamentally *scalar*, in the sense that only a single stream element can be produced or consumed in one activation. The addition of lastprivate clauses on task constructs by itself enables some restricted form of streaming. As we will see, this form of communication is mostly used for enabling hierarchical streaming.

We illustrate the way tasks can communicate with the enclosing context on Figure 2.5. In the current OpenMP specification, a task can only use `firstprivate` and `shared` variables to communicate with the enclosing context. This is sufficient for expressing consumer behaviour, through `firstprivate`, but it clearly falls short of allowing to express a producer behaviour. The only way currently allowed to communicate data out of a task is to use shared variables and explicit synchronization, as is shown in the second task of our example for variable `x` with a `taskwait` directive. This is ambiguous on the behaviour of the task w.r.t. the variable `x`, as it could be either defined or used by the task, or even both or none.

In contrast, the first task on Figure 2.5 shows our proposed extension, where the variable `y` is produced and annotated as `lastprivate`. The semantics of this annotation guarantee that there is no reason to synchronize accesses to this variable during the execution of the task as it is privatized, and it introduces an implicit synchronization point after the task which ensures that the value of `y` is made available in the enclosing context before proceeding.

It is easy to see that this implicit synchronization point serializes the execution of the task with that of its enclosing context. However, the lastprivate clause would only synchronize one task and not the full program, as would a barrier, or a part of the program,

```
int x, y;  // Scalar, structure or array

x = 5; y = 3;

#pragma omp task firstprivate (x) lastprivate (y)
  {
    // "y" is private to the task
    y = 2 * x;
  }
  // copy-out of the last value of "y" within the task
  // implicit synchronization point

if (condition ())
  {
#pragma omp task shared (x, y)
    {
      // "x" and "y" are shared with the rest of the program
      // synchronization may be required for atomicity and ordering
      x = y - 1;
    }
#pragma omp taskwait
  }
  // the value of "x" depends on the "condition()"
```

Figure 2.5: Streaming to and from the enclosing context of a task.

as would a taskwait. One of the main differences between the two solutions lies in the availability of precise information on what data is really being synchronized. Using a lastprivate clause, the programmer not only minimizes the amount of over-synchronization, but also provides important information to the compiler. Explicit synchronization directives, like `taskwait`, are anonymous; in the previous example, the last task's synchronization cannot be statically resolved as only bearing on the last task and for the variable x. Let us show an example, on Figure 2.6, of how the lastprivate clause allows avoiding over-synchronization.

In this example, data read from a file is processed by two independent tasks, but the file is not necessarily processed in its entirety. Depending on a `test` in Task 2, the processing may end before the end of the file is reached. Task 2 uses a lastprivate clause to make the result of the test, `flag`, visible in the outer context. Task 1 is computationally intensive, but Task 2 is relatively lightweight, which makes these two tasks unbalanced.

If this example were implemented using a shared variable for `flag` with a `taskwait` directive to synchronize, the program would wait, in each iteration of the loop, for the termination of both tasks. This means that the two tasks would be coupled and the heavy workload of Task 1 would be unnecessarily forced on the critical path. The main program cannot exit the loop and execute the rest of the program until all activations of both tasks complete. There can be no overlapping of computation between the computationally intensive Task 1 and the rest of the program, therefore leading to an important loss of concurrency.

On the other hand, the implementation using the lastprivate clause only synchronizes

```
int x, y;  // Streams
bool flag = true;

while (!eof (file) && flag)
  {
    x = read (file);

#pragma omp task firstprivate (x)              // Task 1
    {
      heavy_workload (x);
    }

#pragma omp task firstprivate (x) output (y) \ // Task 2
                    lastprivate (flag)
    {
      y = light_workload (x);
      flag = test (y);
    }
    // No ‘‘taskwait’’ required: no coupling between tasks 1 and 2
  }

  // Rest of the program
```

Figure 2.6: Avoiding over-synchronization with the lastprivate clause.

the activations of the second task, therefore allowing the decoupling of the execution of Task 1 from the rest of the program. The main program, along with Task 2 which is synchronized with it, can finish the loop and go on to execute the rest of the program while Task 1 is concurrently finishing up the outstanding activations.

2.4.5 Hierarchical Streaming

To enable hierarchical streaming, where streaming tasks and possibly whole pipelines are nested in other streaming tasks, it is necessary to use both types of communication described above. This is due to the fact that we do not alter the visibility scope of variables in the underlying sequential program. Let us consider the incorrect example on Figure 2.7, where a programmer tries to build a pipeline between two tasks at different hierarchical levels.

The problem in this example is that the variable `x` seen by Task 1 is not the same as that seen by Task 3, because the `firstprivate` clause on Task 2 creates a new variable to privatize `x`. The streaming clauses on Task 1 and Task 3 do not match the same stream. There is an additional issue on the second stream `y`, that cannot be seen from outside of Task 2.

There are two ways to propagate streams from one level to the next, either by using the `shared` clause or by using the `firstprivate` clause to bring data from the enclosing context. The first solution is illustrated on Figure 2.8, where the stream variable is transparently forwarded by the `shared` clause through Task 2. This solution is not always

```
int x, y;  // Streams

#pragma omp task output (x)         // Task 1
  {
    x = ...
  }

#pragma omp task firstprivate (x, y) // Task 2
  {
#pragma omp task input (x) output (y) // Task 3
    {
      y = ... x ...;
    }
  }
```

Figure 2.7: Incorrect usage of input and output clauses.

acceptable as it results in a decoupling between the control flow of the two streaming tasks (1 and 3), which leads to an out-of-order consumption of data on stream `x`. Depending on the application, this may be acceptable.

```
int x, y;  // Streams

#pragma omp task output (x)         // Task 1
  {
    x = ...
  }

#pragma omp task shared (x, y)      // Task 2
  {
#pragma omp task input (x) output (y) // Task 3
    {
      y = ... x ...;
    }
  }
```

Figure 2.8: Using a `shared` clause to connect tasks at different levels of nesting.

The second solution consists in respecting the canonical behaviour of stream communication and remaining within the same hierarchical level, then using the second communication scheme between a task and its enclosing context to forward the data to the nested task. This solution is presented on Figure 2.9, where the crossing of the nesting level is done using `firstprivate` and `lastprivate` clauses. In this example, the outer task behaves as a streaming task at its own level of nesting, but it forwards data from stream `x` to Task 3 and also from Task 3 back to stream `y`. It is easy to see that the two clauses of Task 3 use the implicit access windows stemming from the input and output clauses of Task 2 rather than the stream variables `x` and `y`.

```
int x, y;  // Streams

#pragma omp task output (x)                    // Task 1
  {
    x = ...
  }

#pragma omp task input (x) output (y)          // Task 2
  {
#pragma omp task firstprivate (x) lastprivate (y) // Task 3
    {
      y = ... x ...;
    }
  }
```

Figure 2.9: Using the `firstprivate` and `lastprivate` clauses to connect tasks at different levels of nesting.

The example on Figure 2.9 also serves as motivation for enabling lastprivate clauses on OpenMP tasks. While using shared memory communication with explicit synchronization would be semantically correct, it results in lost optimization opportunities. If we replace the `lastprivate(y)` clause by `shared(y)` and we add a `#pragma omp taskwait` after Task 3, the compiler will not necessarily be able to remove the synchronization and properly forward the data stream instead. In a more general way, we aim at reducing the usage of shared clauses, which do not provide data dependence information and allow data races. When it is possible, programmers should be shielded from such behaviour and should not have to resort to improvized synchronization schemes that are error prone both for correctness and for performance.

2.4.6 Execution and Memory Model

The efficient compilation and exploitation of our extension requires the following adjustments to the execution and memory models provided in the OpenMP specification.

Execution model

The OpenMP3.0 execution model states that, whenever a thread encounters a task construct, a task instance is generated from the code of the associated structured block. This task may either be scheduled immediately on the same thread or deferred and assigned to any thread in the team. The data environment of the task is created according to the data-sharing attribute clauses on the task construct and any defaults that apply.

Such a model is well suited for very unbalanced loads, but in most cases the overhead of creating and scheduling the tasks is significantly higher than synchronizing persistent tasks. This is particularly acute on streaming tasks with regular production/consumption rates, operating on large sliding windows of live data. More generally, streaming allows to exploit any amount of regularity that exists in the program's communication, by serializing the data in the communication channels, even in the case of dynamic streaming patterns.

This can also be understood as a prefetching scheme where producers gather the data in streams and the consumers benefit from ordered and contiguous reads. For this reason, the execution model should leverage this advantage and allow a regular execution on regular data.

We propose to make streaming tasks persistent in our OpenMP extension. We emphasize the fact that this change only affects the *execution model*: the semantics of OpenMP programs is not impacted. This choice puts a heavier load on the compiler: it needs to convert the dynamic scheduling of new instances of a task into data-driven synchronization (i.e., based on the availability of data in the input streams). The impact of this conversion, as well as the optimizations it enables in terms of coarsening, and the efficiency considerations, in particular with respect to load balancing, are described in Chapter 7.

In this new model, all streaming tasks are created at the beginning of the enclosing OpenMP context and they can only execute when sufficient data is present on all input channels. When a task has no input channels, but has output channels (thus qualifying it for streaming), an implicit input stream is created to carry the control-flow, thus converted to data-flow.

Memory model

The extension of the memory model to include streaming across distributed memory spaces is work in progress and is not addressed in this thesis. This is a natural extension when no shared memory clause is used, as demonstrated by CellSs for example [10]. But supporting arbitrary OpenMP applications efficiently on distributed platforms remains an open problem and is the topic of on-going research by Millot et al. in the STEP project [47].

The memory consistency model of OpenMP is relaxed-consistency and requires explicit flush operations and atomic operations to enforce stronger consistency properties. While the choice of this memory consistency model is motivated by performance concerns, in particular mitigating the memory wall, this model is not natural for programmers. In all but embarassingly parallel applications, the handling of potential data races is one of the most daunting tasks in parallel programming, even with strong memory consistency guarantees, and this problem is exacerbated by the need for programmers to also understand the additional constraints introduced by the relaxed consistency model.

As we prove in Section 3.6, our extension provides the most stringent form of memory consistency, strict consistency, for all read and write operations in streams. Along with the high level of determinism that this extension provides, see Section 3.5, this improves productivity and simplifies the understanding of parallel programs. As these guarantees only hold for stream communication, this also further motivates our choice of avoiding the use of the `shared` clause. Note that these strong memory consistency guarantees do not incur a memory access overhead, as properly implemented streams will favor local on-chip communications where only relevant data is actually transferred.

2.5 Semantics of the Extension

In this section, we elaborate on the semantical interpretation of our proposed extension. The OpenMP specification provides many illustrative examples that help understanding of the semantics and we also adopt this stance. This section mostly provides clarifications on the behaviour of our extension over some common and relevant use-cases.

2.5.1 Foreword on OpenMP Tasks

Before we delve into the semantics of our extension and the coding patterns that it enables, it is important to first understand how non-streaming tasks behave in an OpenMP program.

For instance, because of its execution model, the task construct is mostly used within the scope of a worksharing construct[3]. As every thread encountering a task construct will create a dynamic instance of the task, it is necessary to be able to discriminate the different instances, in a context where tasks can be scheduled anytime, anywhere (barring tied tasks and "if" clauses). For this reason the task construct will actually be meaningful only in cases where threads are already differentiated, like e.g., within a worksharing construct.

```
#pragma omp parallel num_threads (2)
  {

    for (i = 0; i < N; ++i)
      {
#pragma omp task
        {
          work (i);
        }
      }
  }
```

```
#pragma omp parallel num_threads (2)
  {
#pragma omp for
    for (i = 0; i < N; ++i)
      {
#pragma omp task
        {
          work (i);
        }
      }
  }
```

Figure 2.10: Task instances need to be differentiable. Multiple undifferentiated instances (left) and properly differentiated (right).

Figure 2.10 illustrates this issue. On the left, the task construct is used directly within the `parallel` directive. The duplicate instances of the task, one for each of the two threads executing this parallel section, are impossible to differentiate. This is not only a performance issue. We duplicate the work, but we also duplicate shared memory accesses, so that synchronization would be required if the task accesses any shared data: the duplicate instances would use the same shared memory locations. By contrast, on the right, the worksharing construct (the `omp for`) distributes loop iterations onto the available threads and ensures a single instance of the task will be activated for each value of `i`.

[3]OpenMP3.0 defines the following worksharing constructs: `loop`, `sections`, `single` and `workshare`.

The same semantics apply to streaming tasks. We do not forbid their use outside of a worksharing construct, but the alternative is to manually differentiate threads, which is seldom portable and often considered a bad coding practice.

Another important point is that, by nature, pipelining requires tasks to be part of a loop structure as filters are applied to a sequence of elements. It is possible to use streaming tasks outside of a loop nest, which then behaves as a loop with a single iteration. This is of little interest overall as it only incurs the overhead of creating streams for single elements.

In the remainder of this chapter, we will only discuss the cases where streaming tasks are enclosed by a loop, which itself must be nested within a worksharing construct. However, as these constructs can be part of a caller function, the callee can possibly only exhibit freestanding tasks, which is sometimes the case in our examples for brevity.

2.5.2 Coding Patterns

Though this is by no means an exhaustive list of the possible uses of streaming tasks or of their interaction with other OpenMP constructs, we will present in the following paragraphs how our extensions can provide the necessary building blocks for programming streaming applications.

Pipeline parallelism

To provide the fundamental basis for pipelining, we use the `single` worksharing construct for building a simple pipeline, as for example on Figure 2.11. The two task constructs communicate through stream `x`, which decouples their execution by privatizing the sequence of values of `x`.

```
#pragma omp parallel
#pragma omp single
  {
    for (i = 0; i < N; ++i)
      {
#pragma omp task output (x)
        x = ... ;

#pragma omp task input (x)
        ... = ... x ...;
      }
  }
```

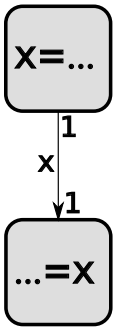


Figure 2.11: Pipeline using the `single` worksharing construct.

Parallel and sequential filters

The activations of filters can be executed concurrently, provided that there is sufficient data on input streams and that the filter has no self-dependences. The presence of self-dependences on a task denotes the presence of state carried-over from an activation to the next. Figure 2.12 illustrates this situation, where the first task is both producer and consumer of the same stream. This pattern is used to express statefullness and inhibit the concurrent execution of the task's activations.

```
int counter, x;
for (i = 0; i < N; ++i)
  {
#pragma omp task input (counter) output (x, counter)
    {
      counter++;
      x = ... ;
    }

#pragma omp task input (x)
    ... = ... x ...;
  }
```

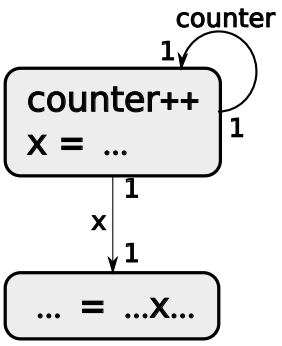


Figure 2.12: Sequential filter because of a self-loop.

In this example, the first filter must execute sequentially while the second could be data-parallelized. The ordering of the elements in the stream is preserved by the stream library implementation even if multiple threads concurrently process different activations of a task. The runtime focuses on scheduling data inside the stream rather than scheduling the activations, which minimizes ordering constraints on the execution of task activations.

This pattern is extremely common. It happens for instance when a filter reads or writes to a file. In that case, the file descriptor is both read and written at every access to the file and it should be considered as filter state. If a task does not properly specify inputs and outputs, the resulting behaviour is unspecified.

Note that the example on Figure 2.12 is incomplete as the self-dependence on the first task is unsatisfiable. In order to provide an initial value to the state, we will introduce a delay on the stream `counter` that encodes the state. The pattern used for delays will be presented later.

Mixing data and pipeline parallelism

While the runtime may exploit data parallelism within a pipeline by executing multiple instances of parallel filters (starting with the heaviest ones for load balancing), the programmer can decide to make data-parallelism explicit at the pipeline or individual filter level.

For example the code on Figure 2.13 creates parallel pipelines of filters by using the `loop` worksharing construct instead of `single`. We must note that the correctness of this parallelization depends on the behaviour of the tasks. The user is responsible for

```
int x, A[N];

#pragma omp parallel private (x)
#pragma omp for shared (A)
  for (i = 0; i < N; ++i)
    {
#pragma omp task output (x)
      x = ... ;

#pragma omp task input (x) shared (A)
      A[i] = ... x ...;
    }
```

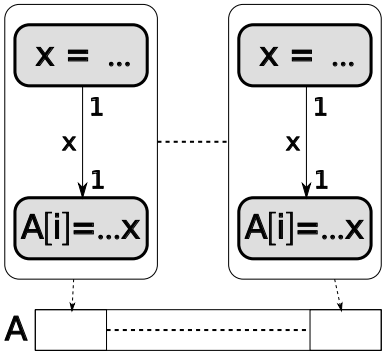


Figure 2.13: Parallel replicated pipelines with a worksharing construct.

ensuring no dependence is violated and for using proper synchronization if required. The programmer could even introduce a second level of data parallelism within a filter, by nesting a parallel loop within a streaming task as illustrated on Figure 2.14. However, this may result in unnecessary overhead if the implicit barrier at the end of the nested parallel loop cannot be removed by the compiler, as the cost of barrier synchronization is significantly higher than our stream synchronization scheme, presented in Chapter 6.

Critically in this example, the stream variable `x` is private to each thread, using the `private` clause on the `parallel` construct, which means that each thread executing the parallel loop will have its own stream and therefore build separate pipelines. The case where `x` is not private but shared, and therefore leading to a single stream that is used by all threads, will be discussed later. It then corresponds to a single pipeline where the computation in the enclosing context is parallelized, but not necessarily the execution of the filters.

```
int x,X[k];

#pragma omp parallel
#pragma omp single
    for (i = 0; i < N; i+=k)
      {
#pragma omp task output (x << X[k])
        {
#pragma omp parallel for
          for (j = 0; j < k; ++j)
            X[j] = ... ;
        }

#pragma omp task input (x)
        ... = ... x ...;
      }
```

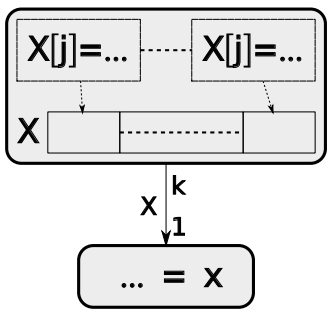


Figure 2.14: Parallel loop worksharing construct within a filter.

Figure 2.14 shows an alternative way of forcing data-parallelism inside a filter, albeit a rather inelegant and likely inefficient one. The first task uses an access window with horizon and burst of size `k` to concurrently produce multiple values on stream `x`. As the data is not committed before the activation completes, the second task will have to wait. Also note that in this example the first task produces $k \times N$ integers to stream `x`, while the second task only consumes N. As we will see, this type of behaviour is detrimental to bounding the amount of memory required for stream communication.

Exploiting data and pipeline parallelism

In the previous two paragraphs, we have seen how the programmer can introduce state in filters, therefore precluding all data parallelization of such filters, and how it is possible to explicitly mix data and pipeline parallelism. We must stress the fact that it is not necessary to use any parallel constructs in order to exploit any available data-parallelism. Indeed, as we have mentioned earlier, any stateless task is inherently data parallel, so the runtime system is allowed to dynamically schedule activations from such tasks on concurrent threads. Exploiting data parallelism in stateless filters is an essential part of any load-balancing strategy as it allows reducing bottlenecks in pipelines; when one filter is slower than all the others the throughput of the pipeline is equal to that of the slowest filter.

As a shorthand for expressing data-parallelism in streaming tasks, while also allowing programmers to specify the amount of data-parallelism for each task, we are considering to allow using the `num_threads` clause on task constructs. This would only serve as a hint to the compiler and runtime system that a specific task needs to be executed on a specific number of threads for best performance.

```
int x;

#pragma omp parallel num_threads (2)
#pragma omp for shared (x, A)
  for (i = 0; i < N; ++i)
    {
#pragma omp task output (x)
      x = ... ;

#pragma omp task input (x) shared (A)
      A[i] = ... x ...;
    }
```

```
int x;

for (i = 0; i < N; ++i)
  {
#pragma omp task output (x) \
          num_threads (2)
    x = ... ;

#pragma omp task input (x) \
          shared (A) num_threads (3)
    A[i] = ... x ...;
  }
```

Figure 2.15: Pipelines parallelized with a parallel construct (left) or with a `num_threads` clause (right).

The two examples presented on Figure 2.15 have the same semantics for the first task, the task's activations are executed on two threads, but the second task is, for example, more finely tuned with three threads. Note that this example differs from Figure 2.13 as the stream variable x is shared. As a result, there is only one pipeline in this example and a single stream x. The data-parallelism is exploited within the same pipeline.

Dynamic pipelines

In some cases, like the butterfly stages of an FFT, it is necessary to build a pipeline where the depth is parametric. The dynamic nature of a pipeline is captured by the array of streams construct that we added in the stream clauses. We can build dynamic pipelines by connecting tasks with successive streams in an array of streams as for example on Figure 2.16. The array of streams `A` is used to communicate through the pipeline for different instances of the second task.

```
int A[K];

#pragma omp parallel
#pragma omp single
  {
    for (i = 0; i < N; ++i)
      {
#pragma omp task output (A[0] << x)
        x = ... ;
      }

    for (j = 0; j < K-1; ++j)
      {
        for (i = 0; i < N; ++i)
#pragma omp task input (A[j] >> x) output (A[j+1] << y)
          {
            y = ... x ...;
          }
      }

    for (i = 0; i < N; ++i)
      {
#pragma omp task input (A[K-1] >> x)
        ... = ... x ... ;
      }
  }
```

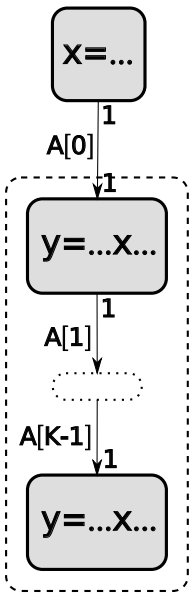


Figure 2.16: Dynamic pipeline of filters generated from a loop by using an array of streams.

In this example, the first task initiates the pipeline by producing some data on the first stream of the array `A`. The second task effectively builds the dynamic pipeline by accessing different streams depending on the value of `j` which is the induction variable of its outermost loop. At iteration j_0, each task instance reads data on stream `A[`j_0`]`, processes the data, then writes the output on the next stream in the array, `A[`j_0`+1]`.

This example also illustrates the dynamic nature of the taskgraph that is generated from our streaming extension as a single static task construct generates $K - 2$ actual pipeline filters. Until now, there was a one to one correspondence between task constructs and pipeline filters. With this pattern, we illustrate the need to identify pipeline filters

not only by a syntactic task construct, but also by the set of input and output streams it is connected to.

Our extension not only allows data-flow to be expressed between tasks, but also provides a way to statically specify how the dynamic taskgraph of an application is built.

Variable burst and horizon sizes

The use of variable burst and horizon sizes for stream access windows improves the expressiveness of the model, but it comes at a cost, both in terms of efficiency and on the properties that can be extracted from the program with static analysis.

In C99, it is possible to declare arrays whose size is only known when entering a block, which means that our syntax can lead to variable-sized bursts and horizons in streams. Variable horizons make it very problematic to determine the global buffer size for a stream variable. There are two options: either (1) the programmer provides the maximal horizon across all iterations to make this computation possible, as illustrated on Figure 2.17, but this reduces the concurrency between the two filters; or (2) the programmer uses variable horizon sizes, as shown on Figure 2.18, and can gain additional pipeline parallelism at the cost of a more difficult program to analyze. The difference in the amount of pipeline parallelism comes from the fact that the second task either waits for N values to be available on the input stream or just for i, as the stream synchronization needs to wait for enough data in the stream to fill the horizon of the window.

```
int X[N];

for (i = 0; i < N * (N-1) / 2; ++i)
  {
#pragma omp task output (x)
    {
      x = ... ;
    }
  }

for (i = N-1; i >= 0; --i)
  {
#pragma omp task input (x >> X[i])
    {
      // i <= N
      ... = ...X[0]...X[1]...X[i];
    }
  }
```

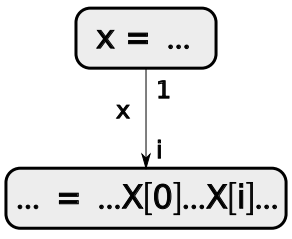


Figure 2.17: Restricting the horizon size to be a constant does not forbid a task to consume a variable number of elements.

Note that both approaches have the same expressiveness: bursts can be dynamic so each instance of a task can consume or produce a different number of elements, as illustrated on Figure 2.17, where the remainder of the horizon is left unused.

Let us take a closer look at these two approaches using the example on Figure 2.18 as reference. When horizons are variable, once the first task starts producing data, the second task is immediately able to start executing because it only requires one value on its input stream. If instead of a variable horizon, the user declares the access window with the maximum horizon value, the second task will need to wait for N values before starting. This is not the only issue, however, as the Figure 2.17 illustrates, using the maximum required horizon rather than the actual value will lead to a termination problem. The second task waits, in its last activation, for more data than is really necessary as it asks for N values whereas only one is needed. The programmer will have to add fake values at the end of the stream to compensate.

```
int x, y;
// int X[N];

for (i = 0; i < N; ++i)
  {
    int X[i+1];

#pragma omp task output (x)
    {
      x = ... ;
    }

#pragma omp task input (x >> X[0]) output (y)
    {
      y = 0;
      for (j = 0; j < i; ++j)
        y += X[j];
    }
  }
```

Figure 2.18: Partial sums computation using an access window with variable horizon and burst.

Programs that include either of these patterns, where some streams have access windows with variable burst or horizon sizes, do not allow the compiler to bound the amount of memory required for streaming. We provide a scheme for dynamically increasing the size of stream buffers, but that may not be a sufficient solution and we may decide to require programmers to declare the maximum amount of memory that should be allocated for such streams. Another issue is that this behaviour generally makes it undecidable whether a deadlock can occur, which will also lead to increased overhead from our runtime deadlock detection scheme.

2.5.3 Buffering Semantics of Stream Communication

The semantics of communication with the input and output streaming clauses is defined such that data is never discarded without being used. The semantics can be likened to that of a FIFO queue where output clauses push some amount of data and input clauses

pop some data. This leads to coding patterns that would not respect the sequential semantics[4] of the program, and that can therefore appear misleading. The following discussion on nesting of streaming tasks in arbitrary control flow will also allow us to emphasize the difference between streaming between tasks using input and output clauses and communicating with the outer context with firstprivate and lastprivate clauses.

Conditional execution

Streaming tasks can be nested into arbitrary control flow, like for instance the conditional blocks on Figure 2.19.

```
for (i = 0; i < N; ++i) {
  if (condition_1 (i)) {
#pragma omp task firstprivate (i) output (x)
    x = i ;
  }
  if (condition_2 (i)) {
#pragma omp task firstprivate (i) input (x)
    y = x + i ;
  }
}
```

Figure 2.19: Streaming tasks nested within arbitrary control-flow.

In the example presented on Figure 2.19, the values of x computed by the first task are buffered and consumed by the second task when it activates. Streaming clauses allow bypassing the control flow, so that the $k-$th activation of the second task consumes the value produced by the $k-$th activation of the first task, irrespectively of the truth values of the conditions.

```
for (i = 0; i < 2; ++i) {
  if (i == 0) {
#pragma omp task firstprivate (i) output (x)
    x = i ;
  }
  if (i == 1) {
#pragma omp task firstprivate (i) input (x)
    y = x + i ;
  }
}
```

Figure 2.20: Instantiation of the example on Figure 2.19.

This highlights the semantic difference between the input and firstprivate clauses, as the latter does not have buffering semantics and is control flow sensitive. The firstprivate clause simply forwards values from the enclosing context to the task in compliance with OpenMP3.0 semantics. As a result, depending on the conditions predicating both tasks,

[4]The semantics of the program where all OpenMP annotations are stripped.

the second task may not compute $y = i+i$ as it is possible that $x \neq i$. To clarify this issue, let us consider the case presented on Figure 2.20, where we instantiate the conditions. Each task is activated exactly once: when the first task activates, it uses $i = 0$ to define x and commits the value 0. When the second task is activated, we have $i = 1$, but the task still reads $x = 0$ on its input stream, therefore computing $y = 0 + 1$.

We motivate this choice of a buffering semantics for streaming clauses as follows.

- It offers maximal expressiveness, unlike a "sampling" semantics where in-flight data on input and output clauses would be implicitly discarded. Defining when this data should be discarded is also problematic since the task may be nested into arbitrary control flow.
- It simplifies the compilation: the conditional expression does not have to be sunk inside tasks, resulting in undesired if-conversion with spurious task activations when the condition is false.
- It is fully in line with the formal basis for data-flow and stream computing defined by Kahn [36], it matches directly the underlying execution model of our proposal, and it is compatible with OpenMP 3.0.
- But it is also a challenge for static buffer sizing. Programs with diverging queues of in-flight data can be written easily. This is a well known issue, solved with a type system (clock calculus) in data-flow synchronous languages [29], or with periodic restriction on the production/consumption rates [43]. Because of its expressiveness and modularity, we consider the type-based approach in further refinements of our streaming extension.

The ability to ignore pragmas or to generate code that can run sequentially is further discussed in Chapters 3 and 5. Ignoring pragmas is the canonical trivial serialization approach in OpenMP, but it is usually not possible with our extension.

The compilation of persistent tasks requires a way to enable the activation of a task only when the control dependences guarding the task are satisfied. This scheme is one of the main issues developed in Chapter 7. It relies on a stream of *task activations* that carries the control dependences and preserves the order of task activations in the serial schedule. This avoids sinking the task's enclosing control flow inside the persistent process and activating the task when there is no work to be done. It also allows to deterministically schedule the data inside streams, based on the order information on task activations.

firstprivate and lastprivate vs. input and output

These clauses are semantically very close, especially if we consider input and output clauses with unitary bursts and horizons. From a task's perspective, firstprivate and input both represent a privatizing copy-in of a value, while lastprivate and output represent copying out, upon termination, the last value of some variable.

Yet as discussed above, their semantics differ with regard to the control flow. This is easier to understand as an issue of reaching definitions or merging functions in the *Single Static Assignment (SSA)* representation [20]. To better understand this difference, let us

consider the SSA form of a similar example to that on Figure 2.20, which we present on Figure 2.21. The SSA form is built using a loop-closed SSA form distinguishing between loop-Φ and cond-Φ nodes. The latter take an additional condition argument, appearing as a subscript, to make the selection condition explicit.

```
x0 = 42;
y0 = 42;
z0 = 42;
for (i0 = 0; (i1 = Φ^loop(i0, i2)) < 2; i2 = i1 + 1) {
  // Writes to x and y are not visible in the main program's loop.
  z1 = Φ^loop(z0,z3);
  if (i1 == 1) {
#pragma omp task firstprivate (i1) output (x1) lastprivate (z2)
    {
      x1 = i1;
      z2 = i1;
    }
  }
  z3 = Φ^cond_(i1 == 1)(z1, z2);
  if (i1 == 0) {
#pragma omp task firstprivate (i1) input (x1) firstprivate (z3)
    {
      y1 = x1 + i1;
      y2 = z3 + i1;
    }
  }
}
```

Figure 2.21: SSA form for the instantiated example from Figure 2.20.

This example highlights the difference between these clauses. The first task produces new values for both x and z, the first using an output clause and the second with a lastprivate clause. One difference that can be seen right away is that only variable z has a loop-Φ node at the beginning of the loop because the writes to both y and z are never visible outside of the tasks where they occur. For z in particular, this shows the difference between the output and the lastprivate clause: the latter forwards the data to its enclosing context, thus constituting a definition operation on z that is captured by the SSA version $z2$, while the former only produces one value of x on a private channel that cannot be seen from the loop's body.

This difference becomes even more pronounced when considering the second task, that reads both the x and z variables, the first with an input clause and the second with a firstprivate clause. The variables' SSA versions once again differ between the two clauses, because of the same fundamental semantical difference. Variable z behaves as if the OpenMP annotations were absent: the firstprivate clause copies the value of $z3$ from the enclosing context, which is merged by the cond-Φ node, while the input clause reads directly from the private channel. As a result, $z3$ being read under the `i1 == 0` condition, the $\Phi^{\text{cond}}_{\texttt{i1 == 1}}$ node resolves the value to $z1$ which itself is $z0$. The streaming clauses output and input bypass altogether the control flow and there can be no Φ nodes on x.

The result of the computation, in this particular instantiation, is that the second task computes a first value of y that is defined using the first data element from the stream, which is therefore resolved to the data produced in the first task's first activation. This occurs for `i1 == 1`, so the first value on stream x is 1. The second task therefore computes $y1 = x1 + i1 = 1 + 0$ as this task executes when `i1 == 0`, therefore copying the value 0 from $i1$. On the other hand, the second value of y computed by this task is the one we could expect if we removed all OpenMP annotations: $y2 = z3 + i1 = z0 + i0 = 42 + 0$.

One important semantical consideration is that streams implicitly carry control flow information. The availability of data in a stream satisfies both data and control dependences of the consumer. We implement firstprivate clauses in a way that is similar to input clauses, with streams, as our choice of a persistent-task execution model means the expansion of the clause needs a stream to forward copy-in values to the thread in charge of executing the successive instances of the task.

2.5.4 Sampling Patterns

Sampling patterns are very frequent in digital signal processing applications, which are one of the main classes of programs that best fit the stream programming model. Sampling patterns can very easily be implemented with our OpenMP extension, both for static and regular patterns, like for example decimation patterns, or for dynamically variable downsampling patterns that allow, for example, to refine the sampling rate at runtime. We present, on Figures 2.22 and 2.23, two examples to illustrate this type of coding patterns.

```
for (i = 0; i < N; ++i)
  {
#pragma omp task output (x)
    x = ...;
  }

for (i = 0; i < N/4; ++i)
  {
#pragma omp task input (x >> X[4])
    use (X[0]);
  }
```

Figure 2.22: Implementation of a static decimation pattern.

The example presented on Figure 2.22 shows how a static decimation pattern can trivially be implemented with our extension. The second task consumes 4 elements on stream x at each activation of the task, but it only actually reads the first value, thus achieving a $4\times$ down-sampling of stream x.

The second example, presented on Figure 2.23, shows how a dynamic rate down-sampling can be achieved. This is a simple example where the rate does not depend on the data itself. The first task is simply a source of N data samples, the second task

```
for (i = 0; i < N; ++i)
  {
#pragma omp task output (x)
    x = ...;
  }

rate = ...;
sum = 0;
while ((sum += rate) < N)
  {
#pragma omp task input (x >> X[rate])
    use (X[0], X[1]);

    rate = ...;
  }
```

Figure 2.23: Dynamic down-sampling of a stream using variable burst rates.

processes this stream with a dynamically computed down-sampling rate. The programmer must use a dynamic burst rate, while ensuring that the task does not read too far ahead. In our example, the second task accesses two data elements at each activation, so the window's size should at least be 2 elements even if the burst rate falls below that.

```
int S[2];
int s;

for (i = 0; i < N; ++i)
  {
#pragma omp task output (s >> S)
    s = ...;

rate = ...;
sum = 0;
while ((sum += rate) < N)
  {
#pragma omp task input (S[0] >> X[rate])
    use (X[0], X[1]);

#pragma omp task input (S[1] >> X[rate]) lastprivate (rate)
    rate = compute_next_rate (X[0..rate-1]);
  }
```

Figure 2.24: Dynamic down-sampling of a stream using variable burst rates.

If the programmer needs a dynamic down-sampling rate that depends on the data within the stream itself, then there are two possible implementations, but we only present the most efficient one on Figure 2.24. This example is much more complex and requires a broadcast array of streams, S, with a scalar access window used by the source task, s. The

intuition is that we use an inspector task, the third task, that also reads the same data as the main computational task, the second task. The inspector decides on the amount of data that should be consumed on the next activation and pushes this new *rate* value in the enclosing context with a lastprivate clause. The inspector task is serialized with the main program because of this pattern, but its computational load can be low enough to not constitute a bottleneck. It is also possible to merge the second and third tasks, but this would result in introducing a lastprivate clause on the computational task, which would serialize the main workload.

2.5.5 Multi-Producer Multi-Consumer Streams

Unlike most streaming frameworks, our extension provides multi-producer multi-consumer streams that do not rely on static interleaving patterns. As such, our programming model is more versatile, in particular for expressing dynamic dependence patterns.

Multiple connections

It is possible to connect multiple filters to the same stream, both as input and as output. The semantics of multiple filters using the same stream is to interleave the stream accesses according to the sequential schedule. This interleaving scheme is a powerful way of defining fully dynamic *split* and *join* operations on streams. Our approach departs from the regular patterns and statically defined split-join models proposed in common stream programming languages like StreamIt [68].

We illustrate the way data from multiple producers can be dynamically merged into one stream on Figure 2.25 and how it can be conversely split between multiple consumers in Figure 2.26.

```
for (i = 0; i < N; ++i)
  {
#pragma omp task output (x)
    x = 0;

    if (i % 2)
    {
#pragma omp task output (x << X[2])
      X[0] = X[1] = 1;
    }

    // x = 0 0 1 1 0 0 1 1 ...
#pragma omp task input (x >> X[2])
      use (X[0], X[1]);
  }
```

Figure 2.25: Dynamically merging output data from multiple producer tasks in a single stream.

On Figure 2.25, the first task produces one 0 on stream x per iteration of the loop. The

second task only activates on odd iterations and produces two values on stream x that are interleaved with those produced by the first task. The sequential schedule provides the interleaving order which, in this case, yields the sequence of values 00110011... for stream x. The dynamic split pattern is very similar, as illustrated on Figure 2.26 where the first task produces data, here the loop counter, on stream x and the second and third tasks consume from x. As the consumers interleave their stream accesses, the second task always reads even values in the stream and the third task always reads odd values.

```
for (i = 0; i < N; ++i)
  {
#pragma omp task firstprivate (i) output (x)
    x = i;

    if (i % 2)
    {
#pragma omp task input (x)
      // x == i - 1

#pragma omp task input (x)
      // x == i
    }
  }
```

Figure 2.26: Dynamically splitting input data from a stream among different consumer tasks.

Note that it is also possible to enable multiple consumers to see the same data, which we call broadcast semantics. As we have seen in Section 2.4.3, this is achieved by producing data in an array of streams and having all consumers that need to access all the data produced read in a different stream in the array. The compiler only generates one stream, but the presence of the array of streams is important to properly identify whether a consumer task interleaves its accesses with another or if they both read the same data. To avoid statically undecidable cases, where the compiler needs to conservatively generate multiple streams for broadcast patterns and therefore incur the copy overhead, we do not allow mixing dynamic merging of data inside arrays of streams used for broadcast patterns, unless all of the tasks merging data use the same broadcast pattern. As Figure 2.27 shows, either all producers broadcast to the same array of streams, or there is no dynamic interleaving.

On the left, the first two tasks broadcast the value they compute in their access window x to all streams in the array of streams A. The third and fourth tasks read the same sequence of values. As they do not read from the same stream in the array, their accesses are not interleaved, but independent. On the right, the first task broadcasts the values computed in its access window x to the streams of the array A, but the second task only produces data to stream $A[0]$, which results in different data being produced in streams belonging to the same broadcast array. This is not semantically wrong, but we do not allow breaking-up broadcast arrays of streams as this does not allow to implement them as a single stream. This code will result in a compiler error in our implementation: all

arrays of streams used to broadcast data can only be used in broadcast mode in all output clauses.

```
int A[4];
int x;

for (i = 0; i < N; ++i)
  {
    if (i%2)
    {
#pragma omp task output (x >> A)
      x = 1;
    }
    else
    {
#pragma omp task output (x >> A)
      x = 0;
    }

#pragma omp task input (A[0])
    // x == (i%2)

#pragma omp task input (A[3])
    // x == (i%2)
  }
```

```
int A[4];
int x;

for (i = 0; i < N; ++i)
  {
    if (i%2)
    {
#pragma omp task output (x >> A)
      x = 1;
    }
    else
    {
#pragma omp task output (x >> A[0])
      x = 0;
    }

    // Compilation error ...
  }
```

Figure 2.27: Legal (left) and illegal (right) dynamic merge inside a broadcast array of streams.

Making the interleaving of stream accesses the default behaviour and requiring the use of a broadcast array to enable multiple consumers to access the same data differs with our previous work [56]. The initial model was asymmetric: multiple producers would interleave their data in streams while multiple consumers would all consume the same data, which appeared to be potentially misleading to programmers. This new semantics was first motivated by our attempt at unifying the syntax and semantics of the input and output clauses. It finally imposed itself when we started working on the code generation for more dynamic programs than the automatically generated examples in [56], as the former syntax proved to lack the information required for initialization and termination of stream programs, namely the number of broadcast consumers per stream was not decidable until program termination. This meant that stream buffer space could not be reused. In this new syntax, the number of broadcast consumers is known from the size of the broadcast array; even if it is only known at runtime, this is sufficient to generate the appropriate code.

Delays

Among stateful filters, *delays* play a central role. A unit delay prepends an initial value to the stream it takes as input, effectively delaying input values by one activation of

the task. Delays are used to break instantaneous dependence cycles in the SDF model of computation [43]; they take the form of the "pre" operator (akin to a synchronous register) in synchronous data-flow languages [29]. Their role is paramount in the modeling of hardware circuits and control-dominated embedded systems. Strangely, delays have not met the same success for parallel stream programming (yet), as illustrated by their absence from the StreamIt benchmark suite [68].

```
#pragma omp task output (x << A[k])
  for (i = 0; i < k; ++i)
    A[i] = ...;

  for (i = 0; i < N; ++i)
    {
#pragma omp task input (y) output (x)
      x = ... y ...;
#pragma omp task input (x) output (y)
      y = ... x ... ;
    }
```

Figure 2.28: Introducing delays on streams.

Delays can be implemented as a special case of the multi-producer pattern, through tasks producing the desired amount of data before the producer filters start executing. This is illustrated on Figure 2.28, where `k` initial elements are inserted in the stream `x` by the first task. The interleaving behaviour guarantees that this data is inserted at the beginning of the stream. Thanks to the multiple output semantics defined above, this first task implements a k-delay operator. If $k \geqslant 1$, this is sufficient to break the instantaneous dependence cycle among the two following tasks. No internal state is required to implement delays, the state is hidden in the stream, by storing the delay values, and outside in the control flow. This guarantees that delays do not waste data-parallelism by inducing spurious serialization due to internal state.

This pattern provides a stateless alternative to the "pre" operator. It also allows introducing delays dynamically, during the execution of a program, which gives even more flexibility.

2.5.6 Deadlocks and Dependence Cycles

The previous constructs can induce dependence cycles among tasks, through the input and output clauses. Delays can be used to break such cycles. Unfortunately, high expressiveness has a cost: with arbitrary control flow enclosing task activations and variable burst rates, detection of instantaneous (i.e., delay-free) dependence cycles is generally undecidable during compilation. We have to accept deadlocks as part of the semantics of the language extension, though we show in Chapter 3 that we can provide some guarantees and some coding rules that can eliminate them altogether.

At least, we know that if deadlocks occur, they will occur deterministically, independently of the number of threads or scheduling policy. This means that traditional test and debugging procedures for sequential programs are still applicable. We also provide,

in Chapter 3, a runtime deadlock detection algorithm that allows to gracefully exit the program and provide debug information on the deadlock state.

Although no complete method to avoid deadlocks can exist, conservative approaches have been very successful for embedded system design; they are based on control-flow and burst rate restrictions [43] (also adopted by StreamIt), or they rely on type systems of synchronous clocks [29]. Integrating some of these principles in the compiler could provide debugging help and support more aggressive optimizations; these research directions are left for future work.

2.5.7 Execution Model

Following are a few important considerations on the execution model underlying these language constructs. Our scope here is mainly what a programmer perceives, in terms of execution model, rather than a comprehensive analysis of our execution model, which will be further developed in Chapters 6 and 7.

Persistent tasks

To improve performance, we propose to adjust the execution model to make streaming tasks persistent. Instead of having one instance of the task for each point in the iteration space of the enclosing OpenMP context (worksharing construct or any other OpenMP construct), we have a single instance that traverses the full iteration space, consuming data on the input streams and producing on the output streams. We emphasize the fact that this modification of the execution model is not a requirement of the extension we propose. Under the right circumstances, in particular w.r.t. the target architecture support for lightweight scheduling [41], the compiler might still generate code that fits the current execution model for tasks.

To prove the validity of this transformation, let us consider the acceptable schedules of the task instances. In the old schedule, no ordering, no exclusion and no thread locality could be assumed. All schedules were therefore acceptable (without explicit locking). In the new execution model, the persistent task traverses the iteration space in a statically-defined partial order, thus restricting the possible schedules to a subset of the acceptable schedules. The particular case where a parallel filter is replicated to benefit from data parallelism means that the iteration space has been strip-mined and that the different instances will impose a local order for the execution. The resulting set of possible schedules is still a subset of the acceptable schedules in the old execution model.

Correctness is of course the burning question at this point. Overall, the transformation is always possible and correct when the only scheduling constraints are the data-driven ones imposed by `input` and `output` clauses. Obviously, introducing atomic sections within tasks will not interfere with task-level scheduling constraints. However causality problems may arise when combining our streaming extensions with arbitrary locking mechanisms, if the acquisition of a lock escapes outside task boundaries. In real applications, locking may be legitimate to handle other forms of concurrency unrelated with the parallelization itself (e.g., I/O or user interfaces). Conversion to persistent tasks forces the ordering of successive task instances. Whereas valid schedules may exist for independent, freely schedulable tasks, it is possible that none of them be compatible with the sequential

execution of dynamic instances of a given task. Without further precautions, conversion to persistent tasks may thus result into deadlocks (of the evil, target-dependent kind).

Because of the critical performance advantage of the persistent-task execution model, and because of the importance of compiler optimizations to tune the grain of task and pipeline parallelism [26], we choose to (minimally) constrain the usage of cross-task locking mechanisms. Since OpenMP encourages programmers to make the sequential execution a subset of the legal schedules of the parallel program [69], one may require cross-task locking to be compatible with the serial execution of tasks. When generating persistent tasks, the compiler can safely assume that the original schedule of task instances is deadlock-free. Regarding debugging and test, one only has to compile the program for serial execution to make sure it is deadlock-free.

Nesting

For all nesting purposes, we consider that the nesting of a streaming task within any OpenMP construct behaves in the same way standard task constructs behave. The iteration space taken into account for a streaming task is relative to the nearest enclosing OpenMP construct. However, the visibility of input and output clauses and therefore the visibility of the resulting task graph spans across all constructs within the nearest enclosing parallel region. This of course is also limited by the visibility scope of the stream variables.

Data parallelism

Data parallelism is typically exploited automatically at runtime as all tasks that do not belong to a dependence cycle (induced by input and output clauses) are fully data-parallel. Note that our choice to represent statefullness as a self-dependence, with both input and output clauses on the same task for a given stream, allows this generalization of data-parallelism. The programmer is also free to mix pipelining with other data-parallel OpenMP constructs: the compiler will generate broadcast, splitter and selector patterns to handle synchronization and stream buffer indexing.

Streaming tasks are data-parallel by nature, because they only read from and write to private memory (including their stream horizon). Unless the task uses shared memory with explicit synchronization or it is part of an inter-task dependence cycle, it is deemed parallel.

2.6 Interaction with Current OpenMP Semantics

We have already presented some of the ways our extension integrates and interacts with other OpenMP constructs in the previous sections and we further elaborate here on the interaction with a focus on the semantical implications. As streaming tasks are connected with ordered communication channels, nesting such tasks in other OpenMP constructs that lead to out-of-order execution of the task instances can result in non-determinism of the schedule of data in streams and sometimes require introducing some level of serialization. More specifically, we discuss the interaction of streaming tasks with

their environment when nested in concurrent contexts, like parallel loops, or when they are nested in non-streaming tasks, where the notion of order is lost.

As we have seen, load-balancing of streaming tasks can be achieved by exploiting data-parallelism within the pipeline as long as streaming tasks are stateless. However, the main program can also become a bottleneck if it cannot produce enough task activations to keep the worker threads busy. One solution is to exploit parallelism in the main program as well, either in the form of data-parallel loops or as task parallelism.

2.6.1 Streaming Constructs in Parallel Loops

When nesting streaming tasks in data-parallel loops, it is important to first understand that the visibility scope of stream variables plays a key role in distinguishing between a set of parallel pipelines within the parallel loop and a parallel loop being used to build a single pipeline of tasks that can independently exploit data-parallelism in stateless tasks.

```
int x;

#pragma omp parallel private (x)
#pragma omp for shared (A)
  for (i = 0; i < N; ++i)
    {
#pragma omp task output (x)
      x = ... ;

#pragma omp task input (x) shared (A)
      A[i] = ... x ...;
    }
```

```
int x;

#pragma omp parallel shared (x)
#pragma omp for shared (A)
  for (i = 0; i < N; ++i)
    {
#pragma omp task output (x)
      x = ... ;

#pragma omp task input (x) shared (A)
      A[i] = ... x ...;
    }
```

Figure 2.29: Parallel pipelines (left) and single pipeline built in parallel (right).

To clarify this issue, let us consider the example presented on Figure 2.29. The only difference between the left and right sides is that the stream variable x is either private or shared within the parallel loop. The result is that on the left side, where x is private, the variable used to define the stream connecting the two tasks is different between the different threads executing the parallel loop. For this reason, different threads build different pipelines and we have the same behaviour as that presented on Figure 2.13. On the right side, however, the stream variable x is shared, which means that each thread executing the parallel loop does indeed see the same stream connecting the two tasks. The result is a single pipeline built concurrently by all threads executing the loop. This is a desirable behaviour as it allows exploiting data-parallelism in the enclosing context, but it also requires more work from the compiler as the schedule of data written in the stream by the first task needs to be consistent with the schedule of the data read by the second.

In a more general case, illustrated on Figure 2.30, the schedule of data may need to be precisely the sequential schedule as the streaming tasks nested in the parallel loop are connected with tasks that are activated sequentially. We will always consider that

```
int x, y, z;

for (i = 0; i < N; ++i)
  {
#pragma omp task output (x)
    x = ... ;
  }

#pragma omp for
  for (i = 0; i < N; ++i)
    {
#pragma omp task input (x) output (y)
      y = ... x ... ;

#pragma omp task input (y) output (z)
      z = ... y ...;
    }

for (i = 0; i < N; ++i)
  {
#pragma omp task input (z)
    ... = ... z ...;
  }
```

Figure 2.30: Streaming tasks nested in data-parallel loops, connecting outside of the loop.

the schedule of data either respects the sequential schedule of the program or that it is unordered.

The example presented on Figure 2.30 requires the compiler to properly evaluate the placement of data in the streams in order to compile correctly. The first task produces data on stream x, where the data is placed according to the sequential schedule. When the second task consumes from x, the compiler evaluates, for each iteration, the position in the stream where the data corresponding to this iteration should be found. While this is simple in the case of constant burst rates, things become more complicated once bursts can be dynamic; this case would require pre-computing the partial sums of the bursts of all previous iterations before deciding on the index of a stream access.

2.6.2 Nesting in Non-Streaming Tasks

Another way to exploit parallelism in the main program is to use non-streaming tasks. Depending on the way tasks are implemented in the general case, it may be impossible to provide a deterministic schedule of data produced by a task nested in a non-streaming task, let alone consistent with the sequential program schedule. In fact the order information is lost due to the semantics of OpenMP tasks, and cannot be preserved. Let us consider the example on Figure 2.31, where a first non-streaming task is used to parallelize some non-streaming computation as well as the generation of the activations of a nested streaming task. In the existing OpenMP model, the first task can execute in any order, and as we

cannot rely on a loop counter[5] or any other means of relating to the serial schedule, we cannot generate the proper schedule of data on stream x. In this case, the consumer task is performing a reduction on integers where, assuming commutativity and associativity, the order is not important as long as the operation is atomic.

```
int x, y;
int a = 0;
int A[N];

for (i = 0; i < N; ++i)
  {
#pragma omp task shared (A, x) firstprivate (i) private (y)
    {
      y = foo (A[i]);
#pragma omp task output (x) firstprivate (y)
      {
        x = bar (y);
      }
    }

#pragma omp task input (x) shared (a)
    {
      #pragma omp atomic
        a += x;
    }
  }
```

Figure 2.31: Streaming task nested in non-streaming tasks.

However, one of our main objectives is to avoid the non-determinism that usually results from exploiting parallelism. We therefore consider that preserving the information on the serial schedule, and therefore preserving a deterministic schedule of data in stream x, takes precedence over performance concerns. In the example on Figure 2.31, a *control stream* (presented more in detail in Chapter 6) is required from the enclosing context to the non-streaming task. It represents essentially an if-conversion and consists, in this case, of a stream of the values of the induction variable i. Based on this information, the schedule of data in stream x is computed, in the general case sequentially across all instances of the enclosing task. This serializing constraint, which still allows some concurrency, can be relaxed in the case of *regular tasks* and provided that no dynamic control flow predicates the activation of the streaming task producing x within the body of its enclosing task. Regular tasks are defined and discussed in Section 5.3.1.

2.6.3 OpenMP Synchronization Constructs

Synchronization is very important in OpenMP as this model relies heavily on shared memory communication. In many cases, OpenMP constructs imply a barrier at the end

[5]To the OpenMP program, the loop counter i is just a variable without any specific semantics.

unless a `nowait` directive explicitly dismisses the synchronization point. We will only give here semantics for the two most relevant synchronization schemes for tasks in OpenMP: the `barrier` and `taskwait` directives.

OpenMP `barrier` directive

Barriers are the main synchronization tool in OpenMP and they are particularly important for parallel loops. The example we used above, on Figure 2.30, perfectly illustrates this: as we did not add a `nowait` clause on the `for` directive, an implicit barrier is added at the end of the parallel loop. The way streaming tasks interact with barriers is quite simple, all task activations generated before the program reaches the barrier must have executed, but the main issue is to ensure that no deadlock results from the lack of resources.

```
int x;

#pragma omp for
  for (i = 0; i < N; ++i)
    {
#pragma omp task output (x)
      x = ... ;
    }
    // Implicit barrier at end of parallel loop

for (i = 0; i < N; ++i)
  {
#pragma omp task input (x)
    ... = ... x ...;
  }
```

Figure 2.32: Streaming across a barrier.

Let us consider the simple example on Figure 2.32, where the first task is the producer on stream x and is in a parallel loop. The consumer is outside the loop, so the main program cannot reach it before passing the implicit barrier at the end of the parallel loop. This means that the consumer will not start consuming data until the producer has finished producing data for all iterations of the loop. Depending on the size of the iteration space, this may lead to a resource deadlock as at least N integers will need to be stored in the stream buffer of x.

OpenMP `taskwait` directive

A different form of barrier that is used to synchronize tasks without necessarily synchronizing the whole program is the `taskwait` directive. Its semantics in OpenMP is to wait for the completion of all outstanding tasks issued by the context in which the taskwait directive is encountered. The `taskwait` directive is a local barrier that only synchronizes the sons in the taskgraph, but not the siblings. The same applies for streaming tasks, and we also have the same issue, with potential resource deadlocks, as with barriers.

2.7 Modular Compilation

The separate compilation of functions containing streaming tasks requires additional information to identify which variables are used as streams. This could possibly be resolved at link time, but not without much complications. In order to make modular compilation easy, we introduce additional annotations at function declaration sites, that describe the streaming behaviour of the function. We do not require additional call-site annotations.

```
#pragma omp parallel
#pragma omp single
  {
    int a;

    for (i = 0; i < N; ++i)
      {
        foo (i, &a);

#pragma omp task input (a)
        {
          ... = ... a ...;
        }
      }
  }
```

```
void foo (int i, int *y)
{
  int x;

#pragma omp task firstprivate (i) output (x)
  {
    x = ... i ...;
  }

#pragma omp task input (x) output (*y)
  {
    *y = ... x ...;
  }
}
```

Figure 2.33: Streaming tasks in a function call.

Let us consider the example presented on Figure 2.33, where a function `foo` contains two streaming tasks. If the left and right side are compiled separately, the compiler cannot know that the `foo` function contains such tasks, which does not allow connecting these tasks with the consumer task on the left. This would result in a compiler error as, on the left, the variable a is used both as a stream and as a variable in the computation.

In order to provide the necessary information, we need to add annotations to the function prototype visible during the compilation, to describe how the function would integrate in a pipeline. The chosen annotation has been added on Figure 2.34. This annotation allows re-constructing the proper streaming structure. Starting from the code on the left, the annotation is sufficient to decide that a is an output stream of the task and to generate the equivalent annotation on the right.

2.8 Concluding Remarks

In this chapter, we presented an extension to enable stream programming in OpenMP. Our work is driven by the quest for increased productivity in parallel programming, which led us to propose a language extension rather than yet another new language, thus leveraging the existing knowledge and tool base of a de facto industry standard for shared memory parallel programming. The strong evidence that has been gathered on the importance of pipeline parallelism for mitigating the memory wall, and therefore gaining scalability and

```
#pragma omp output (*y)
void foo (int i, int *y);


#pragma omp parallel
#pragma omp single
  {
    int a;

    for (i = 0; i < N; ++i)
      {
        foo (i, &a);

#pragma omp task input (a)
        {
          ... = ... a ...;
        }
      }
  }
```

```
#pragma omp output (*y)
void foo (int i, int *y);

#pragma omp parallel
#pragma omp single
  {
    int a;

    for (i = 0; i < N; ++i)
      {
#pragma omp task firstprivate (i) output (a)
        { foo (i, &a); }

#pragma omp task input (a)
        {
          ... = ... a ...;
        }
      }
  }
```

Figure 2.34: Connecting a function containing streaming tasks to a pipeline.

efficiency, has further motivated and guided the development of our programming model. One key choice is to favor an execution model where the tasks are persistent: this choice often allows static scheduling and grain coarsening, which we discuss in Chapter 7 and enables lightweight, lock-free implementations for streaming communications, on which we elaborate in Chapter 6.

We discussed the design principles necessary to maximize the expressiveness and performance benefits of our extension, while also preserving backward compatibility. We presented a detailed description of the extension's semantics and of the coding patterns it enables, satisfying most common use cases both for regular, streaming applications, like digital signal processing, or for non-streaming applications, through the use of dynamic communication patterns and control flow.

As our long term goal is to integrate this extension in the OpenMP specification, we were especially cautious to ensure backward compatibility and to avoid introducing conflicts with the existing specification. We therefore chose to provide a purely incremental extension, that requires no modification to the existing semantics. In order to make a stronger case for our extension proposal, and substantiate our claims on the strong guarantees provided by our programming model, we analyze our programming model through the formalization of a general computational model for streaming, presented in Chapter 3, and the specialization of this formal model for our stream-computing extension, in Chapter 5.

An earlier version of part of the work presented in this chapter was published in [56].

Chapter 3

Control-Driven Data-Flow Model of Computation

Streaming and task-parallel programming are rapidly growing, in the current context of stalling processor frequencies and increasing hardware concurrency which continuously exacerbates the lack of off-chip memory bandwidth, as a performance and productivity oriented solution to parallel programming. Data-flow computing has been used in the past decades not only as a memory-efficient way of exploiting pipeline parallelism, but also because many important properties can be proven on these programs. Our stream-computing extension to OpenMP does not fit the existing data-flow models because of its generality. In this chapter we present a novel formal model of computation, that fits our programming model, and that allows us to prove deadlock-freedom properties, in Sections 3.3 and 3.4, functional and deadlock determinism, in Section 3.5, as well as serializability of streaming programs, in Section 3.7. We discuss the hypotheses necessary to prove these properties and the static analysis requirements. The generalization of the Control-Driven Data-Flow (CDDF) model to tackle issues raised by constraints of bounded memory execution and scalability concerns is presented in Chapter 4.

En raison de la tendance actuelle à l'augmentation du parallélisme des architectures, qui est généralement une réponse aux limitations en fréquence et en puissance des processeurs, la programmation parallèle par tâches ou en streaming s'installe comme une solution pour la programmation de ces architectures qui vise à la fois la performance et la productivité dans un contexte où la bande passante mémoire est le principal goulot d'étranglement. Le calcul sur flôts de données est utilisé depuis plusieurs décennies, non seulement pour son efficacité mémoire en exploitant le parallélisme de pipeline, mais aussi à cause des propriétés essentielles qui peuvent être prouvées sur ces programmes. Notre extension pour le streaming n'est pas bien modélisée par les modèles de calcul par flôts de données existants, en raison de sa généralité, ne respectant pas les contraintes imposées par ceux-ci. Nous présentons, dans ce chapitre, un nouveau modèle de calcul, répondant aux besoins de notre modèle de programmation, qui nous permet de prouver l'absence d'interblocages, dans les sections 3.3 et 3.4, le déterminisme à la fois fonctionnel et d'interblocage, dans la section 3.5, ainsi que la sérialisabilité des programmes en streaming, section 3.7. Nous détaillons les hypothèses nécessaires pour prouver ces propriétés, ainsi que les analyses statiques requises pour décider de la satisfaction de ces hypothèses.

La généralisation du modèle Control-Driven Data-Flow (CDDF) en mémoire bornée est étudiée au chapitre 4.

3.1 Introduction

In stream programming models, programs are structured as graphs of computational tasks, also called filters, connected with explicit communication channels, or streams. Each filter consists of a work function that is a semantically atomic execution unit, that consumes data on input channels and produces data on output channels. In general, filters regularly iterate the execution of the work function as long as there is data on the input channels. These programming models generally implement more or less restricted instances of Kahn Process Networks (KPN) [36].

The current interest in stream-computing has led to the development, in recent years alone, of many new stream programming models. Among others, we can cite StreamIt [68], based on the Synchronous Data Flow [43] model, where task graphs are built by combining straight pipelines of filters with feedback-loops and split-join patterns. While this could give the wrong impression that StreamIt supports multi-producer or multi-consumer streams, these operations rely on static patterns, which give an ad hoc order semantics to join operations and enforce determinism. Similar to StreamIt on these aspects, FastFlow [4] also provides dynamic multi-producer and, separately, multi-consumer streams, which are not order-preserving. To preserve the order, it relies on a secondary tag stream, which is not distributed in split-join patterns and that allows to reconstruct the order when joining the separate branches. Of course this order can be reconstructed because communication patterns are unitary, so the correspondence between data tokens and tags can be preserved.

Common to all of these models and, to the best of our knowledge, more generally to stream programming models, is the notion that the computation is driven by the availability of sufficient input data[1]. The OpenMP extension for stream-computing, presented in the first chapter of this thesis, extends the OpenMP task-parallel programming model in a way that is also similar, in certain aspects, to KPNs. Intuitively, the difference lies in the fact that, in our model, the main OpenMP program specifies both data-flow and control-flow, so instead of providing a work-function to be applied to data from some input streams, we know that this work-function must be applied a specific number of times. One way to handle this would be to rely on if-conversion, creating an additional input stream to encode the control-dependence, but this stream needs to be given a special status as it does not represent only an upper bound on the number of iterations to execute, but also a strict lower bound. This behaviour conflicts with Kahn semantics, as there can be a discrepancy between the required number of firings mandated by the control flow and the least fixed point solution to a feedback-loop a process belongs to. In our model, this generally leads to deadlocks, which we consider to be functional in that they result from discrepancies between the control-flow and the data-flow in the source program.

[1]With the exception of demand-driven models, where producers stall until a consumer requests data, like for example in Communicating Sequential Processes [31]. This model is non-deterministic and it does not fit our streaming extension to OpenMP.

Even more problematic is the generality of our model with respect to communication patterns. As we allow the arbitrary and dynamic interleaving of accesses to streams from different tasks, the functional determinism cannot be deduced from the monotonicity of processes as their composition is less restricted. In our model, even in the absence of feedbacks, the composition of monotonic processes may not be monotonic. This forces us to adopt a new definition of causality, stream causality, that possibly spans multiple processes, though we still rely in some cases on a weaker version, task causality, which is much closer to Kahn causality.

An OpenMP streaming program behaves as a dynamic directed task graph, where the vertices represent tasks that communicate and synchronize through streams modeled by the edges. This model diverges from KPNs in the following ways:

1. Streams can support multiple producers and multiple consumers that deterministically interleave their accesses, hence the task graph needs to be represented by a hypergraph in our model.

2. A *control program*, which is the OpenMP main program, orchestrates the schedule of data in streams to ensure determinism and enable the efficient synchronization of tasks.

3. The use of shared memory communication between tasks as well as explicit barrier synchronization are allowed.

4. The activation of tasks can be predicated by arbitrary control flow.

Our goal, in this chapter, is to introduce a new model of computation, the Control-Driven Data-Flow (CDDF) model, that captures the semantics of our programming model and can be used to prove that, under certain conditions, programs implementing this model are guaranteed to be: (1) free of spurious deadlocks (i.e., deadlocks introduced by the operational semantics of the model rather than functional deadlocks); (2) deterministic both functionally and for the state of the program when a deadlock occurs; and (3) serializable.

Our model is related to the Kahn process networks model, as well as to the tags system meta-model proposed by Lee et al. [44], but the similarities are limited to the reliance on streams viewed as either FIFO channels or tagged collections of events with an order relation on tags, integer indexes in our case, for communicating between processes. Some similarities can also be noted with Feautrier's communicating regular processes [23], but his model relies on a point-to-point synchronization scheme that we do not consider to be amenable to an efficient implementation. Furthermore, as we discuss in Section 3.4, the very notion of process is slightly blurred in our case due to the dynamic nature of our task graph, mostly introduced by coding patterns like the dynamic pipeline presented in Section 2.5.2. We also introduce a new concept, the control program, that cannot be accounted for in current models, either KPN or dataflow models.

The remainder of this chapter is organized as follows. Section 3.2 starts by presenting the CDDF model in its basic version, it discusses ordering constraints on the execution of tasks, it defines and characterizes the various classes of deadlocks that can occur in

CDDF programs. Section 3.3 introduces the notion of stream causality and analyzes the impact of stream causality on deadlocks. Section 3.4 introduces the notion of CDDF tasks, as well as task causality, and shows that some deadlock-freedom conditions can be guaranteed under this weaker form of causality. In Section 3.5, we prove that all CDDF programs, where the control program and the task work functions are deterministic, are guaranteed to be functionally deterministic and to deadlock deterministically. Section 3.6 briefly discusses the memory consistency model of stream communication. Section 3.7 shows that CDDF programs are serializable when they are deadlock-free and possibly even serializable in control program order under strict conditions. Finally, Section 3.8 summarizes the properties guaranteed for CDDF programs and the sufficient conditions proven in the previous sections.

3.2 Definitions and Notations

The CDDF model of computation is not exclusively meant as a modeling tool for our streaming extension to OpenMP, even though we specialize its semantics in a later stage to match our programming model, but rather as a general computational model for stream computing, that captures the semantics of many other stream based programming languages.

In our abstraction, we primarily focus on memory accesses and the implicit stream synchronization is only aimed at enforcing data-flow dependences. Before we generalize this model, we rely on the following simplifying assumptions: (1) the memory space is unbounded, which allows us to model stream accesses as *Single Dynamic Assignment* (SDA); and (2) the control program is a deterministic sequential process.

3.2.1 CDDF Program Structure

Our model focuses on the communication patterns in streams and on the scheduling constraints resulting from data dependences. For this reason, streams play a key role in this model, though only as identifiers in a first instance.

Definition 3.1 (Stream)**.** A stream is a symbol $s \in \mathcal{S}$, where $\mathcal{S}$ is an infinite set of symbols.

Stream communication is not based on the FIFO queue paradigm, as in many related frameworks, but rather on an indexing scheme that behaves, in an infinite memory model, as an injective function from $\mathcal{S} \times \mathbb{N}$ to the set of memory locations. Streams can be understood as infinite arrays of typed elements or even as typed communication channels. The key property is that there can be no overlap or alias issues: if $(s, i) \in \mathcal{S} \times \mathbb{N}$ and $(s', i') \in \mathcal{S} \times \mathbb{N}$ are two different stream locations, either $s \neq s'$ or $i \neq i'$, then the two stream locations cannot reach the same memory location.

Definition 3.2 (Stream access)**.** We define the set $\mathcal{X}$ of stream accesses, where we distinguish between read (R) and write (W) accesses:

$$\mathcal{X} = \{R, W\} \times \mathcal{S} \times \mathbb{N}$$

Contrary to existing streaming models, we do not give much importance to the notion of task, also often called filter in related work, which is usually presented as a work function iteratively applied to data present on input streams and producing data on output streams. As our model is inherently dynamic, it tends to be less focused on regular communication patterns between sequential or regular processes. Instead, our abstraction only keeps the notion of task activation, which is the fundamental atomic unit of work[2] in the form of one execution of a work function. As work functions can be called with different input and output streams, they cannot constitute or fully identify filters.

We further abstract the notion of task activations to only consider the stream accesses necessary for the task activation's execution, irrespectively of the work function applied.

Definition 3.3 (Task activation)**.** A task activation a is defined by the set of stream accesses it uses to read and write data in streams:

$$a \subset \mathcal{X}$$

The set $\mathcal{A}$ of task activations is the powerset of $\mathcal{X}$:

$$\mathcal{A} = \mathcal{P}(\mathcal{X})$$

This view is entirely focused on the sets of memory locations, or equivalently streams and stream indexes, that are read and written during the execution of the task activation. This represents the minimal information necessary to characterize the data dependences between task activations.

Definition 3.4 (Input and output streams)**.** We say that a stream $s \in \mathcal{S}$ is an input (resp. output) stream to a task activation $a \in \mathcal{A}$ and we write $s \in \mathcal{I}(a)$ (resp. $s \in \mathcal{O}(a)$) if the task activation a contains a read (resp. write) access to stream s.

$$\mathcal{I}(a) = \{s \in \mathcal{S} \mid \exists i \in \mathbb{N}, (R, s, i) \in a\}$$
$$\mathcal{O}(a) = \{s \in \mathcal{S} \mid \exists i \in \mathbb{N}, (W, s, i) \in a\}$$

The core of our model is the *control program*, a deterministic sequential process[3] which dynamically constructs a deterministic schedule of data in each stream. This schedule is based on the dynamic control flow of the control program.

As we are not interested in the semantics of the underlying programming language, we only require that this program's execution be non-blocking aside from synchronization barriers. We model the control program by its execution trace where we only keep two types of operations: activation points and barriers. All other operations are omitted from the control program trace in our model.

Activation points can be thought of as either a short-lived task spawn call or as a task activation scheduling call and its evaluation results in the creation of a new task activation

[2]Not to be mistaken with the notion of atomicity in this context. However, and we will develop on this in a latter section, task activations are atomic, in our model, with respect to stream accesses.

[3]We will later relax this hypothesis to avoid the bottleneck that can result from this process being sequential, at a cost either to the determinism of the model or requiring more complex code generation.

that needs to be executed. The barrier's informal semantics is to stall the control program until all the task activations it produced up to the barrier operation are executed.

Definition 3.5 (Activation point)**.** An activation point π is a control program operation that generates a task activation. It is modeled as a set of descriptors of stream access operations:

$$\pi \subset \{(u, s, b, h) \in \{R, W\} \times \mathcal{S} \times \mathbb{N} \times \mathbb{N}\}$$

where u determines if the activation will consume from or produce to stream s, b is the burst, or amount of data produced or consumed, and h is the horizon, which is the amount of data accessed in the stream. The horizon allows read accesses ahead in the stream, which is also called a *peek* operation in related work.

The set of activation points is defined as:

$$\Pi = \Bigg\{ \pi \in \mathcal{P}\Big(\{R, W\} \times \mathcal{S} \times \mathbb{N}^2\Big) \mid \big(\forall (u, s) \in \{R, W\} \times \mathcal{S}, \exists! (u, s, b, h) \in \pi \ \vee \ \nexists (u, s, b, h) \in \pi\big) \\ \wedge \big(\forall (u, s, b, h) \in \pi \, (b \leqslant h) \wedge ((u = W) \Rightarrow (b = h))\big) \Bigg\}$$

This means that an activation point can only have one descriptor of stream operations per stream s and operation type u, so only one instance of burst and horizon values. It also restricts the value of the burst to be lower than the horizon, and forces the equality of burst and horizon values for write operations.

Definition 3.6 (Control program trace)**.** The execution trace of the control program is a, possibly infinite, word on $\Pi \cup \{barrier\}$. The set $\mathcal{K}$ of possible execution traces is defined as:

$$\mathcal{K} = (\pi + barrier)^*$$

A CDDF program's execution is driven by the control program that generates task activations. Using the previous notations, we define the program state as the control program's execution trace and two sets of task activations, one for already executed activations and the other for outstanding task activations.

Definition 3.7 (CDDF program state)**.** We define the set Σ of possible program states:

$$\Sigma = \mathcal{K} \times \big(\mathcal{P}(\mathcal{A})\big)^2$$

A state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o) \in \Sigma$ is defined by the trace of the control program $\mathcal{K}_e$, which is the sequence of operations in $\Pi \cup \{barrier\}$ that have already been executed by the control program, and the sets $\mathcal{A}_e$ of executed task activations and $\mathcal{A}_o$ of outstanding task activations.

Definition 3.8 (Control program execution)**.** The execution of the control program is modeled by two functions: (1) an oracle function $NEXT : \mathcal{K} \to \Pi \cup \{barrier\} \cup \{\top\}$[4] that

[4]The additional possible result $\top$ marks the end of the control program.

models the sequence operator for the control program's execution and provides the next operation to be executed by the control program; and (2) the activation point evaluation function $\xi : \mathcal{K} \times \Pi \rightarrow \mathcal{A}$ that defines the way an activation point is evaluated to generate a task activation:

$$\xi(k,\pi) = \left\{ (u,s,i) \in \mathcal{X} \quad | \quad \exists (u,s,b,h) \in \pi \ \wedge \ i \in \big[\alpha, \alpha + h\big[, \text{ where } \alpha = \sum_{\substack{\pi' \in k \\ (u,s,b',h') \in \pi'}} b' \right\}$$

For a given control program trace k at the time of the evaluation of the activation point π, the function ξ computes the placement of data produced or consumed by the new task activation inside each stream. It relies on the past activation points that were executed by the control program, summing the bursts b' of all activations with the same operation type u on the same stream s. As the burst is the amount of data produced or consumed by a task activation, it determines the shift inside the stream required for the activation, and the sum of all such shifts since the beginning of the control program's execution determines the indexes of the stream accesses that will be attributed to the task activation being generated.

In order to clarify these notions, let us consider the following running example that shows how the control program evaluates activation points.

Example 3.2.1. *Let us consider the following pre-determined control program sequence* $(\pi_1, \pi_2, \pi_3, \pi_4)$*, which defines the function* $NEXT$ *in the scope of this example:*

$$\begin{cases} NEXT(\epsilon) & = \pi_1 = \{(R, s_1, 1, 2), (W, s_2, 2, 2)\} \\ NEXT(\pi_1) & = \pi_2 = \{(W, s_1, 2, 2)\} \\ NEXT(\pi_1.\pi_2) & = \pi_3 = \{(R, s_2, 0, 2), (W, s_2, 1, 1)\} \\ NEXT(\pi_1.\pi_2.\pi_3) & = \pi_4 = \{(R, s_2, 1, 2), (W, s_1, 1, 1)\} \end{cases}$$

The successive evaluation of these activation points by the ξ *function will yield the*

task activations (a_1, a_2, a_3, a_4)*:*

$$\begin{cases} a_1 = \xi(\epsilon, \pi_1) & = \{(R, s_1, i) \in \mathcal{X} \mid i \in [\alpha, \alpha+2[\wedge \alpha = \sum_{\substack{\pi \in \epsilon \\ (R,s_1,b,h) \in \pi}} b = 0\} \\ & \quad \cup \{(W, s_2, i) \in \mathcal{X} \mid i \in [\alpha, \alpha+2[\wedge \alpha = \sum_{\substack{\pi \in \epsilon \\ (W,s_2,b,h) \in \pi}} b = 0\} \\ & = \{(R, s_1, 0), (R, s_1, 1), (W, s_2, 0), (W, s_2, 1)\} \\ a_2 = \xi(\pi_1, \pi_2) & = \{(W, s_1, i) \in \mathcal{X} \mid i \in [\alpha, \alpha+2[\wedge \alpha = \sum_{\substack{\pi \in \pi_1 \\ (W,s_1,b,h) \in \pi}} b = 0\} \\ & = \{(W, s_1, 0), (W, s_1, 1)\} \\ a_3 = \xi(\pi_1.\pi_2, \pi_3) & = \{(R, s_2, i) \in \mathcal{X} \mid i \in [\alpha, \alpha+2[\wedge \alpha = \sum_{\substack{\pi \in \pi_1.\pi_2 \\ (R,s_2,b,h) \in \pi}} b = 0\} \\ & \quad \cup \{(W, s_2, i) \in \mathcal{X} \mid i \in [\alpha, \alpha+1[\wedge \alpha = \sum_{\substack{\pi \in \pi_1.\pi_2 \\ (W,s_2,b,h) \in \pi}} b = 2\} \\ & = \{(R, s_2, 0), (R, s_2, 1), (W, s_2, 2)\} \\ a_4 = \xi(\pi_1.\pi_2.\pi_3, \pi_4) & = \{(R, s_2, i) \in \mathcal{X} \mid i \in [\alpha, \alpha+2[\wedge \alpha = \sum_{\substack{\pi \in \pi_1.\pi_2.\pi_3 \\ (R,s_2,b,h) \in \pi}} b = 1\} \\ & \quad \cup \{(W, s_1, i) \in \mathcal{X} \mid i \in [\alpha, \alpha+1[\wedge \alpha = \sum_{\substack{\pi \in \pi_1.\pi_2.\pi_3 \\ (W,s_1,b,h) \in \pi}} b = 2\} \\ & = \{(R, s_2, 0), (R, s_2, 1), (W, s_1, 2)\} \end{cases}$$

As can be seen on this example, the burst values determine the shift of stream access indexes in each stream. This is illustrated on activations a_3 *and* a_4 *which both read the same indexes in* s_2 *because the activation point* π_3 *has a null burst on that stream, which means that no shift is made and no data is consumed. The horizon represents the number of indexes actually needed and accessed starting from a given beginning position, while the burst represents the shift of the beginning position needed for the scheduling of the indexes of the next task activation.*

Note that the order of evaluation of activation points, and therefore the underlying deterministic control program order, is essential in the attribution of stream access indexes to task activations. We refer to this behaviour as the scheduling of data in streams, which does not need to proceed in a satisfiable order: in our example π_1 *requests to read two data elements from stream* s_1*, which could not possibly be available as no operation has yet occurred. This request will be satisfied by* π_2 *which schedules the production of the desired elements in task activation* a_2*. As we see below, this data schedule allows to avoid relying on an execution schedule; the execution of task activations will be much less constrained.*

As we have seen, our model is centered around the data schedule computed by the control program. In order to model the *possible future* of the data schedule, if the control program were to continue past the already executed trace, we introduce a special task activation that captures and aggregates all possible continuations of the control program. This will play a key role in modelling, for example, barriers as it will represent all the stream accesses that cannot occur before the barrier completes.

We call this special activation the continuation activation. As it depends on the executed control program trace, it will be defined for a given trace $\mathcal{K}_e$ and noted $\mathcal{C}(\mathcal{K}_e)$. This activation cannot be executed: it models the possible future of the program after the control program trace $\mathcal{K}_e$. For that reason, it contains all stream accesses that could be attributed to a task activation in any continuation of the control program.

Definition 3.9 (Continuation activation)**.** In all CDDF program states $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$, the continuation activation $\mathcal{C}(\mathcal{K}_e)$ is implicitly added to $\mathcal{A}_o$. This continuation is defined by:

$$\mathcal{C}(\mathcal{K}_e) = \left\{ (u,s,i) \in \mathcal{X} \quad | \quad i \geqslant \sum_{\substack{\pi \in \mathcal{K}_e \\ (u,s,b,h) \in \pi}} b \right\}$$

The initial state of a CDDF program is therefore:

$$\sigma_{init} = (\epsilon, \varnothing, \{\mathcal{C}(\epsilon)\}) \qquad \text{where} \;\; \mathcal{C}(\epsilon) = \mathcal{X}$$

In the general case, this continuation activation models the stream accesses, and in particular the write accesses, that cannot occur before the control program makes progress. When the control program reaches a barrier, it models all write stream accesses that can only happen after the barrier passes. If for example a write stream access (W, s, i) belongs to the continuation activation and there is an outstanding activation containing (R, s, i), then unless the control program can make further progress, there is an unsatisfiable flow dependence and hence a deadlock.

Proposition 3.10. *The state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$ of a CDDF program satisfies:*

$$\bigcup_{a \in \mathcal{A}_e \cup \mathcal{A}_o} a = \mathcal{X}$$

This state invariant holds by construction of the continuation activation, which attributes to the continuation, on every stream, each stream access not yet scheduled in a task activation generated by the execution of the control program trace. There is also some overlap for read accesses as accesses in $\mathcal{C}(\mathcal{K}_e)$ start at the sum of bursts on each stream, which can be lower than the highest horizon. This property can also be rewritten as:

$$\forall (u,s,i) \in \mathcal{X}, \exists a \in \mathcal{A}_e \cup \mathcal{A}_o, \;\; (u,s,i) \in a$$

3.2.2 Ordering Constraints on Task Activations Execution

One of the key insights of our model is the separation of constraints between the control program, which executes a deterministic sequence of activation points, and the execution of task activations. We elaborate, in Section 3.3.1, on the issue of the order relation induced by $NEXT$ on activation points, and therefore by precedence in the control program trace, while in this section we discuss the ordering constraints that need to be enforced on the execution of task activations.

As we see below, the only fundamental order requirement for task activations execution is induced by the enforcement of flow dependences between activations. However, the implementation of our synchronization scheme, presented in Chapter 6, does not rely on hardware schemes that would allow for inexpensive synchronization of individual memory locations, which is the approach advocated by Feautrier in Communicating Regular Processes [23]. Instead, we rely on the synchronization of closed prefixes of stream

indexes, where indexes lower than a given threshold become read-only once all producers have moved past that point. In our view, this is a mandatory aspect, barring hardware support like full-empty bits, for efficient synchronization of stream communication. This means that our synchronization scheme over-approximates the real scheduling constraint requirements, which brings us to one of the principal issues tackled in this chapter: to which extent does this over-approximation induce new errors, like spurious deadlocks, and what conditions must be satisfied by a CDDF program to avoid these errors.

To model our synchronization scheme, we define the stream prefix order relation and the derived scheduling constraints enforced on the execution of task activations.

Definition 3.11 (Stream prefix order)**.** We define a binary relation $<$ in $\mathcal{P}(\mathcal{A}^2)$ on task activations as:

$$\forall (a, a') \in \mathcal{A}^2 : \quad a < a' \quad \triangleq \quad \exists (s, i) \in \mathcal{S} \times \mathbb{N}, \exists j \leqslant i,\ (W, s, j) \in a \ \wedge\ (R, s, i) \in a'$$

Which means that a' reads data in a stream s while a writes in stream s somewhere in the prefix of a''s read access. Note that this relation is not transitive as any relations $a < a'$ and $a' < a''$ can arise from different streams, nor reflexive or antisymmetric, so it is not an order relation.

We derive the relation $\ltimes \in \mathcal{P}(\mathcal{A}) \times \mathcal{A}$ that models our scheduling constraint for the execution of task activations, which we define as:

$$a \in \mathcal{A}, \mathcal{A}_0 \subset \mathcal{A} \quad \Big(\mathcal{A}_0 \ltimes a \quad \triangleq \quad \forall (R, s, i) \in a, \forall j \leq i, \exists a' \in \mathcal{A}_0,\ (W, s, j) \in a'\Big)$$

Intuitively, we want to say that, for a given program state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$, an outstanding task activation $a \in \mathcal{A}_o$ is in a relation $\mathcal{A}_e \ltimes a$, and therefore has all of its ordering constraints satisfied by the already executed activations in $\mathcal{A}_e$, if and only if there is no task activations $a'' \in \mathcal{A}_o$ such that $a'' < a$, recalling the role of $\mathcal{C}(\mathcal{K}_e) \in \mathcal{A}_o$ as an aggregator of possible futures. As there can only be one single write access operation per stream index, and as Proposition 3.10 ensures that no write access escapes $\mathcal{A}_e \cup \mathcal{A}_o$, this relation can be defined more concisely as:

$$\begin{aligned} \forall a \in \mathcal{A}_e \cup \mathcal{A}_o, \quad \mathcal{A}_e \ltimes a \quad &\triangleq \quad \forall a' \in \mathcal{A}_e \cup \mathcal{A}_o, a' < a \ \Rightarrow\ a' \in \mathcal{A}_e \\ &\triangleq \quad \nexists a' \in \mathcal{A}_o,\ a' < a \end{aligned}$$

For the second version of the definition, we note that under the hypothesis of single dynamic access to streams, there is a single write operation per stream index and Proposition 3.10 ensures this operation belongs to one of the task activations in $\mathcal{A}_e \cup \mathcal{A}_o$. Considering that $\mathcal{A}_e$ and $\mathcal{A}_o$ are disjoint, which we will see holds by construction of any valid program state, concludes the reasoning.

We finally give, on Figure 3.1, the operational semantics of the execution of CDDF programs, as well as an overview of the CDDF execution model on Figure 3.2. The execution rules have the following meaning:

(GEN) The activation generation rule states that the control program can execute an activation point as soon as it is reached by its oracle function, NEXT. The result of

$$\text{(GEN)}\ \frac{\pi := NEXT(\mathcal{K}_e) \qquad \pi \in \Pi}{(\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o) \longrightarrow (\mathcal{K}_e.\pi, \mathcal{A}_e, \mathcal{A}_o \cup \{\xi(\mathcal{K}_e, \pi), \mathcal{C}(\mathcal{K}_e.\pi)\} \setminus \{\mathcal{C}(\mathcal{K}_e)\})}$$

$$\text{(BAR)}\ \frac{\mathcal{A}_o = \{\mathcal{C}(\mathcal{K}_e)\} \qquad NEXT(\mathcal{K}_e) = barrier}{(\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o) \longrightarrow (\mathcal{K}_e.barrier, \mathcal{A}_e, \mathcal{A}_o)}$$

$$\text{(TERM)}\ \frac{\mathcal{A}_o = \{\mathcal{C}(\mathcal{K}_e)\} \qquad NEXT(\mathcal{K}_e) = \top}{(\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o) \longrightarrow (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)}$$

$$\text{(EXEC)}\ \frac{\mathcal{A}_o = \{a\} \cup \mathcal{A}'_o \qquad \mathcal{A}_e \ltimes a}{(\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o) \longrightarrow (\mathcal{K}_e, \mathcal{A}_e \cup \{a\}, \mathcal{A}'_o)}$$

Figure 3.1: CDDF execution rules.

its evaluation by ξ is added to $\mathcal{A}_o$ and the activation point is appended to the existing program trace $\mathcal{K}_e$. The old continuation, $\mathcal{C}(\mathcal{K}_e)$, activation is removed from $\mathcal{A}_o$ and a new one, $\mathcal{C}(\mathcal{K}_e.\pi)$, reflecting the restriction in possible futures of the program, is added.

(BAR) The barrier rule states that the control program only passes a barrier once all outstanding activations are executed, with the natural exception of the continuation activation. The barrier is also appended to the control program trace once cleared.

(TERM) The termination rule marks the end of the program as soon as the control program finishes, which is marked by the $\top$ operation, and no outstanding activations remain. It has similar semantics to the barrier rule, but does not modify the control program's trace. Termination of the CDDF program occurs as soon as the termination rule is executed once. This rule is used as a guard for program termination as it allows an infinite number of transitions once the program reaches termination.

(EXEC) The execution rule states that an outstanding activation a can only be executed once all the activations that need to be scheduled before it, in the stream prefix order, are executed, which as we have discussed is modeled by $\mathcal{A}_e \ltimes a$. Note that Lemma 3.17 guarantees that the continuation activation $\mathcal{C}(\mathcal{K}_e)$ cannot be executed.

Example 3.2.2. *To illustrate the stream prefix order and the resulting execution schedules, let us continue our running Example 3.2.1.*

We had successfully generated four task activations, (a_1, a_2, a_3, a_4)*, so the state of the*

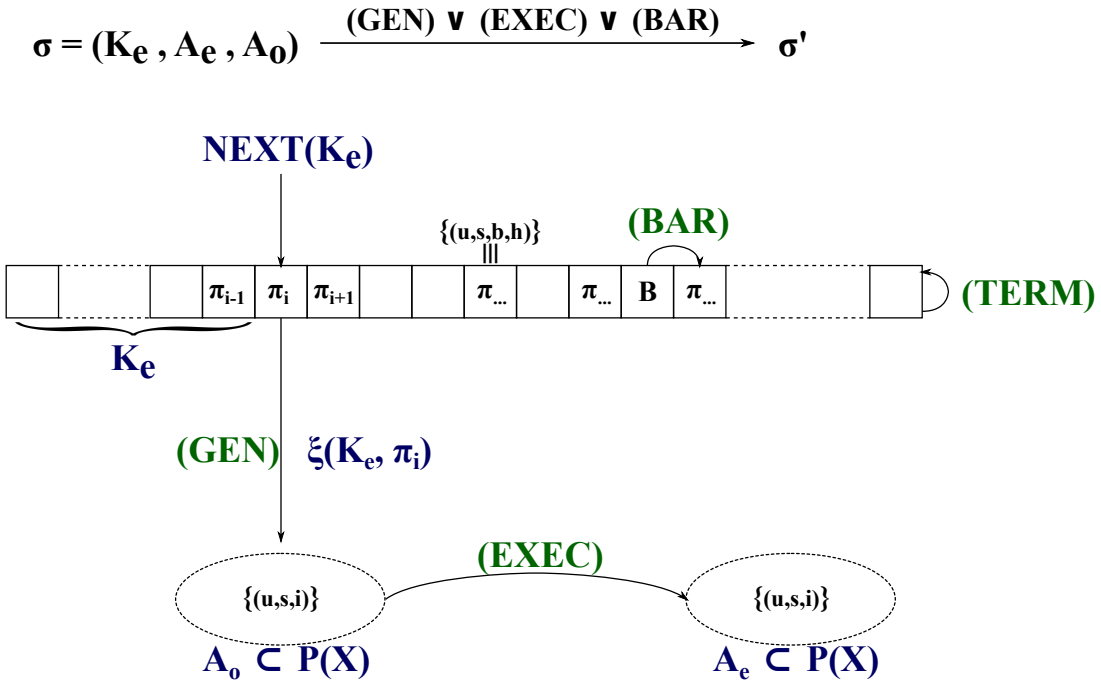


Figure 3.2: Overview of CDDF execution model.

program would be:

$$\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o) \text{ where } \mathcal{K}_e = \pi_1.\pi_2.\pi_3.\pi_4,\ \mathcal{A}_e = \varnothing,\ \mathcal{A}_o = \{a_1, a_2, a_3, a_4, \mathcal{C}_{(\pi_1.\pi_2.\pi_3.\pi_4)}\}$$

$$\text{and } \begin{cases} a_1 = \{(R, s_1, 0), (R, s_1, 1), (W, s_2, 0), (W, s_2, 1)\} \\ a_2 = \{(W, s_1, 0), (W, s_1, 1)\} \\ a_3 = \{(R, s_2, 0), (R, s_2, 1), (W, s_2, 2)\} \\ a_4 = \{(R, s_2, 0), (R, s_2, 1), (W, s_1, 2)\} \end{cases}$$

At this point, only the (GEN) rule has been applied four times, but no task activation has been executed. The rule (EXEC) is predicated by the stream prefix order relation, which requires all writes occurring in the prefix of any read access operation to a given stream to be scheduled before the task activation containing that read access.

The stream prefix order relations between these task activations are:

$$a_2 < a_1 \quad \wedge \quad a_1 < a_3 \quad \wedge \quad a_1 < a_4$$

As $\mathcal{A}_e$ is initially empty, the only task activation a_i that satisfies the predicate $\varnothing \ltimes a_i$, is a_2. As it has no read operations, it satisfies $\varnothing \ltimes a_2$, which is reflected by the fact that no activation is ordered before it on the stream prefix order. Once the execution of a_2 completes, the new state of the program will be:

$$\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o) \text{ where } \mathcal{K}_e = \pi_1.\pi_2.\pi_3.\pi_4,\ \mathcal{A}_e = \{a_2\},\ \mathcal{A}_o = \{a_1, a_3, a_4, \mathcal{C}_{(\pi_1.\pi_2.\pi_3.\pi_4)}\}$$

We can see that, in this new state, the next activation to be ready to execute will be a_1

as $\{a_2\} \ltimes a_1$*, then* a_3 *and* a_4 *will become ready at once, and could possibly be executed concurrently, as we have:*

$$\{a_1, a_2\} \ltimes a_3 \quad \wedge \quad \{a_1, a_2\} \ltimes a_4$$

3.2.3 Program Progress and Deadlocks

Based on the CDDF model, and in particular on the execution rules, we define a simple artificial measure of program progress that adds the length of the control program trace to the number of executed activations:

$$|(\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)| = |\mathcal{K}_e + |\mathcal{A}_e|$$

Note that all rules on Figure 3.1, except `(TERM)`, are strictly monotonically increasing the state of the program with respect to this measure.

Definition 3.12 (Program progress)**.** A CDDF program makes progress from state σ if any execution rule can be applied. The resulting state σ' satisfies $|\sigma| < |\sigma'|$ or the program has terminated. Termination of CDDF programs occurs as soon as the `(TERM)` rule executes once.

This measure and the definition of program progress could be construed as flawed if we do not accept as correct schedules where an infinite sequence of `(GEN)` occurs. While intuitively there is no point in generating an infinity of activations if they never get executed, a bounded memory model would convert such livelocks into resource deadlocks, but for the purpose of the CDDF model we admit such schedules.

The definition of deadlocks simply corresponds to the impossibility of progress from a given state.

Definition 3.13 (Program deadlock)**.** A CDDF program is in a deadlock in state σ, and we note $D(\sigma)$, if no execution rule can apply in that state.

From this definition, we deduce the following property on the state of a deadlocked CDDF program.

Lemma 3.14 (Deadlock state)**.** *The state* $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$ *of a CDDF program in a deadlock satisfies:*

$$D(\sigma) \quad \Leftrightarrow \quad \big(NEXT(\mathcal{K}_e) \notin \Pi\big) \ \wedge \ \big(\mathcal{A}_o \neq \{\mathcal{C}(\mathcal{K}_e)\}\big) \ \wedge \ \big(\forall a \in \mathcal{A}_o, \neg \mathcal{A}_e \ltimes a\big)$$

Proof. We derive, from the definition of the execution rules on Figure 3.1, the following equivalences in a given state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$:

$$\begin{aligned}
&\text{(GEN)} \quad \Leftrightarrow \quad \Big(NEXT(\mathcal{K}_e) \in \Pi\Big) \\
&\text{(BAR)} \quad \Leftrightarrow \quad \Big(\big(\mathcal{A}_o = \{\mathcal{C}(\mathcal{K}_e)\}\big) \ \wedge \ \big(NEXT(\mathcal{K}_e) = \mathit{barrier}\big)\Big) \\
&\text{(TERM)} \quad \Leftrightarrow \quad \Big(\big(\mathcal{A}_o = \{\mathcal{C}(\mathcal{K}_e)\}\big) \ \wedge \ \big(NEXT(\mathcal{K}_e) = \top\big)\Big) \\
&\text{(EXEC)} \quad \Leftrightarrow \quad \Big(\exists a \in \mathcal{A}_o \mid \mathcal{A}_e \ltimes a\Big)
\end{aligned}$$

Definition 3.13 states that a deadlock occurs when no rule can be fired in a given state of the program. We deduce that:

$$\begin{aligned} D(\sigma) \quad &\Leftrightarrow \quad \neg(\text{GEN}) \ \wedge \ \neg(\text{BAR}) \ \wedge \ \neg(\text{TERM}) \ \wedge \ \neg(\text{EXEC}) \\ &\Leftrightarrow \quad \begin{cases} \Big(NEXT(\mathcal{K}_e) \notin \Pi\Big) \\ \wedge \ \Big(\big(\mathcal{A}_o \neq \{\mathcal{C}(\mathcal{K}_e)\}\big) \ \vee \ \big(NEXT(\mathcal{K}_e) \neq barrier\big)\Big) \\ \wedge \ \Big(\big(\mathcal{A}_o \neq \{\mathcal{C}(\mathcal{K}_e)\}\big) \ \vee \ \big(NEXT(\mathcal{K}_e) \neq \top\big)\Big) \\ \wedge \ \Big(\forall a \in \mathcal{A}_o : \ \neg \mathcal{A}_e \ltimes a\Big) \end{cases} \end{aligned}$$

The second and third propositions can be simplified, given that:

$$NEXT(\mathcal{K}_e) \neq barrier \ \wedge \ NEXT(\mathcal{K}_e) \neq \top \ \Rightarrow \ NEXT(\mathcal{K}_e) \in \Pi$$

Which yields:

$$\begin{aligned} D(\sigma) \quad &\Leftrightarrow \quad \neg(\text{GEN}) \ \wedge \ \neg(\text{BAR}) \ \wedge \ \neg(\text{TERM}) \ \wedge \ \neg(\text{EXEC}) \\ &\Leftrightarrow \quad \begin{cases} \Big(NEXT(\mathcal{K}_e) \notin \Pi\Big) \\ \wedge \ \Big(\big(\mathcal{A}_o \neq \{\mathcal{C}(\mathcal{K}_e)\}\big) \ \vee \ \big(NEXT(\mathcal{K}_e) \in \Pi\big)\Big) \\ \wedge \ \Big(\forall a \in \mathcal{A}_o : \ \neg \mathcal{A}_e \ltimes a\Big) \end{cases} \end{aligned}$$

Merging the first and second propositions concludes the proof:

$$D(\sigma) \quad \Leftrightarrow \quad \begin{cases} \big(NEXT(\mathcal{K}_e) \notin \Pi\big) \\ \wedge \ \big(\mathcal{A}_o \neq \{\mathcal{C}(\mathcal{K}_e)\}\big) \\ \wedge \ \big(\forall a \in \mathcal{A}_o : \ \neg \mathcal{A}_e \ltimes a\big) \end{cases}$$

□

3.2.4 Deadlock Characterization

In order to define the different types of deadlocks that can occur in our model, we first need to introduce a flow dependence relation on task activations. We will use Bernstein's definition of data dependences [11] and adjust it to task activations. As stream accesses are single dynamic assignment and all reads come, by definition, after the unique write operation, only data-flow dependences are possible.

Definition 3.15 (Task activation dependence relation)**.** We define the data-flow dependence relation between task activations $\delta \in \mathcal{P}(\mathcal{A}^2)$ using the definition of flow dependences:

$$\forall (a,a') \in A^2 : a \ \delta \ a' \quad \triangleq \quad \exists (s,i) \in \mathcal{S} \times \mathbb{N},\ (W,s,i) \in a \ \wedge \ (R,s,i) \in a'$$
$$\triangleq \quad W(a) \cap R(a') \neq \varnothing \quad \text{where} \begin{cases} W(a) = \{(s,i) \in \mathcal{S} \times \mathbb{N} \mid (W,s,i) \in a\} \\ R(a') = \{(s',i') \in \mathcal{S} \times \mathbb{N} \mid (R,s',i') \in a'\} \end{cases}$$

As with the stream prefix order relation, we derive a relation $\triangleright : \mathcal{P}(\mathcal{A}) \times \mathcal{A} \rightarrow Bool$, that models the fundamental scheduling constraint corresponding to a set of task activations $\mathcal{A}_0$ satisfying all flow dependences for the execution of a task activation a:

$$\mathcal{A}_0 \triangleright a \quad \triangleq \quad \forall (R,s,i) \in a, \exists a' \in \mathcal{A}_0,\ (W,s,i) \in a'$$
$$\triangleq \quad R(a) \subset \bigcup_{a' \in \mathcal{A}_0} W(a')$$

The write access operations belonging to the task activations in $\mathcal{A}_0$ cover all the read operations in a, so all flow dependences of a are satisfied by the execution of the task activations in $\mathcal{A}_0$.

From this definition, it is easy to see that the stream prefix order we enforce in the CDDF model is much more restrictive than necessary. We do not detail the proof of the following proposition, as it directly results from Definitions 3.11 and 3.15.

Proposition 3.16. *Flow dependences are subsumed by the stream prefix order relation.*

$$\forall a, a' \in \mathcal{A} : \qquad a \delta a' \quad \Rightarrow \quad a < a'$$
$$\forall a \in \mathcal{A}, \forall \mathcal{A}_0 \in \mathcal{P}(\mathcal{A}) : \qquad \mathcal{A}_0 \ltimes a \quad \Rightarrow \quad \mathcal{A}_0 \triangleright a$$

This proposition also implies that enforcing the stream prefix order is sufficient to enforce flow dependences in CDDF programs. We also provide the following lemma, that guarantees that the continuation activation can never be a candidate for execution, irrespectively of the dependence relation used.

Lemma 3.17. *For a CDDF program in state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$, the continuation activation's dependences are never satisfied:*

$$\neg\big((\mathcal{A}_e \cup \mathcal{A}_o \setminus \{\mathcal{C}(\mathcal{K}_e)\}) \ltimes \mathcal{C}(\mathcal{K}_e)\big) \quad \wedge \quad \neg\big((\mathcal{A}_e \cup \mathcal{A}_o \setminus \{\mathcal{C}(\mathcal{K}_e)\}) \triangleright \mathcal{C}(\mathcal{K}_e)\big)$$

We do not explicit the proof of this lemma as it is trivial from the definitions of the relations and remarking: (1) that, by construction, $\mathcal{A}_e$ is a finite set of task activations, each containing a finite number of write operations by definition of ξ; and (2) that $\mathcal{C}(\mathcal{K}_e)$ contains an infinite number of read stream accesses. All read accesses of $\mathcal{C}(\mathcal{K}_e)$, and therefore its flow dependences, cannot be satisfied by a finite number of write accesses.

The earlier blanket definition of deadlocks can be further refined, based on the source of the deadlock, in three categories: functional deadlocks, insufficiency deadlocks and spurious deadlocks. We will respectively note $FD(\sigma)$, $ID(\sigma)$ and $SD(\sigma)$ when a functional, insufficiency or spurious deadlock occurs in state σ.

These types of deadlocks can be informally defined in the following way:

Functional deadlocks occur when all outstanding activations have unsatisfied flow dependences and the control program cannot make progress. This type of deadlocks corresponds to an algorithmic error as they correspond to situations where no schedule that would preserve flow dependences can exist, for example because of an unsatisfiable flow dependence cycle.

Insufficiency deadlocks occur when the control program cannot make progress, because of a synchronization point or because of its termination, and insufficient data has been scheduled to be produced to meet the requirements of consumers in some stream. The control program cannot generate any more task activations that could produce the missing data and there is at least one outstanding task activation that cannot execute. In our model, this happens whenever any outstanding activation depends on the continuation activation $\mathcal{C}(\mathcal{K}_e)$.

Spurious deadlocks are all remaining deadlocks, which are due to the over-synchronization scheme we enforce rather than a more fundamental issue. While insufficiency and functional deadlocks are a program correctness issue, arising from algorithmic errors, which cannot be resolved and are considered to fall under the programmer's responsibility, the presence of spurious deadlocks highlight inconsistencies in our model, induced by the over-synchronization of flow dependences. The existence of spurious deadlocks, and the conditions under which we can prove freedom thereof, are the price to pay for enabling an efficient implementation of stream synchronization on closed prefixes.

A fourth type of deadlocks, resource deadlocks, stem from the limitations of the amount of memory available for stream buffers or outstanding activations, so they are naturally absent from an unbounded memory abstraction; however, resource deadlocks are an issue in the generalized CDDF model that we present in Section 4.

Functional deadlocks

The definition of functional deadlocks is similar to that of CDDF deadlocks. They occur when no progress can be made in a program where the stream access ordering enforces only flow dependences (Definition 3.15) instead of our over-approximated stream prefix order (Definition 3.11).

Definition 3.18 (Functional deadlock)**.** A CDDF program is in a functional deadlock in state σ, and we note $FD(\sigma)$, if no execution rule can apply in that state, replacing rule (EXEC) with:

$$(\text{EXEC}_{\triangleright})\ \frac{\mathcal{A}_o = \{a\} \cup \mathcal{A}'_o \qquad \mathcal{A}_e \triangleright a}{(\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o) \longrightarrow (\mathcal{K}_e, \mathcal{A}_e \cup \{a\}, \mathcal{A}'_o)}$$

We can also prove the following lemma for functional deadlock states of CDDF programs. The proof is identical to that of Lemma 3.14.

Lemma 3.19 (Functional deadlock state)**.** *The state* $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$ *of a CDDF program in a functional deadlock satisfies:*

$$FD(\sigma) \quad \Leftrightarrow \quad \big(NEXT(\mathcal{K}_e) \notin \Pi\big) \ \wedge \ \big(\mathcal{A}_o \neq \{\mathcal{C}(\mathcal{K}_e)\}\big) \ \wedge \ \big(\forall a \in \mathcal{A}_o, \neg \mathcal{A}_e \triangleright a\big)$$

Insufficiency deadlocks

Insufficiency deadlocks represent deadlocks where the control program stopped too early and did not generate the task activations necessary for the completion of the existing outstanding task activations. They are therefore linked to the continuation of the control program, which is why we rely on the continuation activation to define them. Note, however, that we primarily define insufficiency deadlocks using the flow dependence relation rather than the stream prefix order, and that there is no equivalence between the corresponding deadlock states. However, we will get an equivalence once we relax the definition of deadlock states.

Definition 3.20 (Insufficiency deadlocks)**.** A CDDF program is in an insufficiency deadlock in state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$, and we note $ID(\sigma)$, when the control program cannot make progress and all outstanding task activations depend, transitively, on the continuation activation:

$$ID(\sigma) \quad \triangleq \quad \big(NEXT(\mathcal{K}_e) \notin \Pi\big) \ \wedge \ \big(\mathcal{A}_o \neq \{\mathcal{C}(\mathcal{K}_e)\}\big) \ \wedge \ \big(\forall a \in \mathcal{A}_o, \ \mathcal{C}(\mathcal{K}_e) \ \delta^+ \ a\big)$$

We also similarly define insufficiency deadlocks on the stream prefix order (Definition 3.11):

$$ID_<(\sigma) \quad \triangleq \quad \big(NEXT(\mathcal{K}_e) \notin \Pi\big) \ \wedge \ \big(\mathcal{A}_o \neq \{\mathcal{C}(\mathcal{K}_e)\}\big) \ \wedge \ \big(\forall a \in \mathcal{A}_o, \ \mathcal{C}(\mathcal{K}_e) <^+ a\big)$$

The following lemma shows that this definition is consistent with that of functional deadlocks.

Lemma 3.21. *Insufficiency deadlocks are functional deadlocks.*

For a CDDF program in a state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$*, we have:*

$$ID(\sigma) \qquad \Rightarrow \qquad FD(\sigma)$$

Proof. The proof relies on Lemma 3.17, which ensures that the continuation activation is never executable with (EXEC$_\triangleright$):

$$\neg\big((\mathcal{A}_e \cup \mathcal{A}_o \setminus \{\mathcal{C}(\mathcal{K}_e)\}) \triangleright \mathcal{C}(\mathcal{K}_e)\big)$$

By Definition 3.20, we have:

$$ID(\sigma) \qquad \Rightarrow \qquad \big(\forall a \in \mathcal{A}_o, \ \mathcal{C}(\mathcal{K}_e) \ \delta^+ \ a\big)$$

Which can be rewritten and expanded from the definition of flow dependences:

$$\begin{aligned} ID(\sigma) \quad &\Rightarrow \quad \big(\forall a \in \mathcal{A}_o, \exists a' \in \mathcal{A}_o,\ \mathcal{C}(\mathcal{K}_e)\ \delta^*\ a'\ \wedge\ a'\ \delta\ a\big) \\ &\Rightarrow \quad \big(\forall a \in \mathcal{A}_o, \exists a' \in \mathcal{A}_o, \exists (s,i) \in \mathcal{S} \times \mathbb{N},\ (W,s,i) \in a'\ \wedge\ (R,s,i) \in a\big) \\ &\Rightarrow \quad \big(\forall a \in \mathcal{A}_o, \exists (R,s,i) \in a, \forall a' \in \mathcal{A}_e,\ (W,s,i) \notin a'\big) \\ &\Rightarrow \quad \big(\forall a \in \mathcal{A}_o,\ \neg\big(\mathcal{A}_e \triangleright a\big)\big) \end{aligned}$$

By adding the remainder of the definition of $ID(\sigma)$, that $\big(NEXT(\mathcal{K}_e) \notin \Pi\big)\ \wedge\ \big(\mathcal{A}_o \neq \{\mathcal{C}(\mathcal{K}_e)\}\big)$, allows to conclude the proof. □

Spurious deadlocks

We finally define spurious deadlocks as deadlocks that are not functional.

Definition 3.22 (Spurious deadlock)**.** A spurious deadlock is a non-functional deadlock.
For a state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$, we note:

$$\mathrm{SD}(\sigma) \quad \triangleq \quad \mathrm{D}(\sigma)\ \wedge\ \neg\mathrm{FD}(\sigma)$$

This is the only type of deadlocks that are due to our over-approximation of data-flow dependences (Definition 3.11) and not to the program semantics. Our primary objective is to ensure that we avoid such deadlocks in CDDF programs.

Weak deadlock states

In many cases, we are able to show that the program will necessarily deadlock in the future of a given state, without knowing precisely when it will deadlock or whether the current state is already a deadlock state. A state from which all execution schedules lead to a deadlock will be called a weak deadlock state. The purpose here is to switch from universal quantifiers in the conditions of deadlock states to existential quantifiers in the conditions of weak deadlock states.

Definition 3.23 (Weak deadlock state)**.** A state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$ of a CDDF program is a weak deadlock state, for any type of deadlock, if there is a maximum state $\sigma' = (\mathcal{K}'_e, \mathcal{A}'_e, \mathcal{A}'_o)$ such that:

$$D(\sigma') \quad \wedge \quad |\sigma| \leqslant |\sigma'| \quad \wedge \quad \mathcal{K}_e = \mathcal{K}'_e$$

We note $WD(\sigma)$, $WFD(\sigma)$, $WID(\sigma)$ and $WSD(\sigma)$ when the state σ satisfies the weak deadlock condition respectively for a CDDF deadlock, a functional deadlock, an insufficiency deadlock or a spurious deadlock.

This definition is entirely directed at simplifying the definition of deadlock states. As we require $D(\sigma')\ \wedge\ \mathcal{K}_e = \mathcal{K}'_e$, we ensure that the control program cannot make further progress and therefore cannot generate new task activations:

$$NEXT(\mathcal{K}'_e) = NEXT(\mathcal{K}_e) \in \{barrier, \top\}$$

This condition is very important as it precludes a form of live-lock where only a part of the program is in a deadlock, the remainder progressing possibly indefinitely if the program is non-terminating.

It also preserves the continuation activation, as it depends only on the control program trace, and, considering that only the (EXEC) rule can apply, allows to know that:

$$\mathcal{A}'_o \subset \mathcal{A}_o$$

This ensures that $|\sigma'| - |\sigma|$ is finite, so the maximum number of transitions required to reach a deadlock from a weak deadlock state is finite.

Weak insufficiency deadlock state

This weaker definition of deadlock states allows to reason in terms of local deadlock conditions rather than global conditions, on all outstanding task activations. For example, a program is in an insufficiency deadlock state only if all outstanding activations depend on the continuation activation, but we show that it is sufficient to know that a single outstanding activation depends on the continuation $\mathcal{C}(\mathcal{K}_e)$ to conclude that the state is a weak insufficiency deadlock once the control program has reached the barrier.

This approach weakens the deadlock conditions sufficiently to show, for instance, the equivalence of weak insufficiency deadlock states when occurring due to the enforcement of flow dependences or of the stream prefix order.

Lemma 3.24 (Weak insufficiency deadlock state)**.** *A CDDF program in a state* $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$ *satisfies:*

$$\begin{aligned} WID(\sigma) \quad &\Leftrightarrow \quad NEXT(\mathcal{K}_e) \notin \Pi \quad \wedge \quad \big(\exists a \in \mathcal{A}_o,\ a \neq \mathcal{C}(\mathcal{K}_e) \ \wedge\ \mathcal{C}(\mathcal{K}_e)\ \delta\ a\big) \\ &\Leftrightarrow \quad NEXT(\mathcal{K}_e) \notin \Pi \quad \wedge \quad \big(\exists a \in \mathcal{A}_o,\ a \neq \mathcal{C}(\mathcal{K}_e) \ \wedge\ \mathcal{C}(\mathcal{K}_e) < a\big) \end{aligned}$$

Proof. First, we show that $WID(\sigma) \ \Rightarrow\ NEXT(\mathcal{K}_e) \notin \Pi \ \wedge\ \big(\exists a \in \mathcal{A}_o,\ a \neq \mathcal{C}(\mathcal{K}_e) \ \wedge\ \mathcal{C}(\mathcal{K}_e)\ \delta\ a\big)$.

By Definition 3.23 of a weak deadlock state, there is a state $\sigma' = (\mathcal{K}'_e, \mathcal{A}'_e, \mathcal{A}'_o)$ that satisfies:

$$ID(\sigma') \quad \wedge \quad |\sigma| \leqslant |\sigma'| \quad \wedge \quad \mathcal{K}_e = \mathcal{K}'_e$$

Which can be further expanded using the Definition 3.20 of insufficiency deadlocks on σ':

$$\Big(\big(NEXT(\mathcal{K}'_e) \notin \Pi\big) \quad \wedge \quad \big(\mathcal{A}'_o \neq \{\mathcal{C}(\mathcal{K}_e)\}\big) \quad \wedge \quad \big(\forall a \in \mathcal{A}'_o,\ \mathcal{C}(\mathcal{K}_e)\ \delta^+\ a\big)\Big) \quad \wedge \quad |\sigma| \leqslant |\sigma'| \quad \wedge \quad \mathcal{K}_e = \mathcal{K}'_e$$

As we know that $\mathcal{A}'_o \neq \{\mathcal{C}(\mathcal{K}_e)\}$, and we recall that the set of outstanding task activations $\mathcal{A}_o$ always contains at least the continuation activation, there is $a' \in \mathcal{A}'_o, a' \neq \mathcal{C}(\mathcal{K}_e) \ \wedge\ \mathcal{C}(\mathcal{K}_e)\ \delta\ a'$.

As $\mathcal{K}_e = \mathcal{K}'_e$, there can be no task activation generation in between the two states of the program, so $\mathcal{A}'_o \subset \mathcal{A}_o$, and therefore $a' \in \mathcal{A}_o$, which concludes the first part of the proof.

The second part must show: $WID(\sigma) \Leftarrow NEXT(\mathcal{K}_e) \notin \Pi \wedge \big(\exists a \in \mathcal{A}_o,\ a \neq \mathcal{C}(\mathcal{K}_e) \wedge \mathcal{C}(\mathcal{K}_e)\ \delta\ a\big)$.

Let $\sigma' = (\mathcal{K}_e, \mathcal{A}_e \cup \mathcal{A}_o \setminus \{\mathcal{C}(\mathcal{K}_e), a\}, \{\mathcal{C}(\mathcal{K}_e), a\})$. As $\{\mathcal{C}(\mathcal{K}_e), a\} \subset \mathcal{A}_o$, we deduce that $|\mathcal{A}_e| \leqslant |\mathcal{A}_e \cup \mathcal{A}_o \setminus \{\mathcal{C}(\mathcal{K}_e), a\}|$, which yields that:

$$|\sigma| \ = \ |\mathcal{K}_e| + |\mathcal{A}_e| \quad \leqslant \quad |\mathcal{K}_e| + |\mathcal{A}_e \cup \mathcal{A}_o \setminus \{\mathcal{C}(\mathcal{K}_e), a\}| \ = \ |\sigma'|$$

As we know that $NEXT(\mathcal{K}_e) \notin \Pi$ and both states have the same trace $\mathcal{K}_e$, we only need to show that $ID(\sigma')$ is true to conclude the proof. $\mathcal{A}'_o = \{\mathcal{C}(\mathcal{K}_e), a\}$ satisfies $\mathcal{A}'_o \neq \{\mathcal{C}(\mathcal{K}_e)\}$ and the only remaining activation in $a \in \mathcal{A}'_o$ satisfies $\mathcal{C}(\mathcal{K}_e)\ \delta\ a$, which concludes this second part of the proof.

Finally, to prove the equivalence of weak deadlock states defined by either flow dependence or stream prefix order enforcement, we need to show that:

$$\forall a \in \mathcal{A}_o,\ \mathcal{C}(\mathcal{K}_e) < a \quad \Leftrightarrow \quad \mathcal{C}(\mathcal{K}_e)\ \delta\ a$$

Proposition 3.16 yields:

$$\forall a \in \mathcal{A}_o,\ \mathcal{C}(\mathcal{K}_e) < a \quad \Leftarrow \quad \mathcal{C}(\mathcal{K}_e)\ \delta\ a$$

So we need only prove the reverse in order to conclude this proof. By definition of the stream prefix order:

$$\mathcal{C}(\mathcal{K}_e) < a \quad \Leftrightarrow \quad \exists s \in \mathcal{S}, \exists i, j \in \mathbb{N}, \quad j \leqslant i \ \wedge\ (W, s, j) \in \mathcal{C}(\mathcal{K}_e) \ \wedge\ (R, s, i) \in a$$

From the Definition 3.9 of the continuation activation, we deduce that:

$$\forall s \in \mathcal{S}, \exists \alpha \in \mathbb{N} : \quad \forall k \in \mathbb{N} \mid k \geqslant \alpha, (W, s, k) \in \mathcal{C}(\mathcal{K}_e)$$

Which allows us to conclude that:

$$(W, s, j) \in \mathcal{C}(\mathcal{K}_e) \quad \Rightarrow \quad j \geqslant \alpha$$

And as $j \leqslant i$, we further deduce that:

$$(W, s, j) \in \mathcal{C}(\mathcal{K}_e) \quad \Rightarrow \quad i \geqslant j \geqslant \alpha \quad \Rightarrow \quad (W, s, i) \in \mathcal{C}(\mathcal{K}_e)$$

By definition of the flow dependence relation, $(W, s, i) \in \mathcal{C}(\mathcal{K}_e) \ \wedge\ (R, s, i) \in a \ \Rightarrow\ \mathcal{C}(\mathcal{K}_e)\ \delta\ a$, which concludes the proof. □

Note that this equivalence of weak insufficiency deadlock states is possible only because we do not require a global condition on the set of outstanding activations. Indeed, the real insufficiency deadlock states are possibly different when enforcing either of the constraints, but the cause is the same: these deadlocks represent states where insufficient data can be produced on some streams, which are filled incrementally and therefore any data missing in the prefix of a stream index leads to data missing in the remainder.

We can show that a state σ satisfies:

$$ID_<(\sigma) \quad \Rightarrow \quad WID_<(\sigma) \quad \Leftrightarrow \quad WID(\sigma) \quad \Leftarrow \quad ID(\sigma)$$

This is sufficient to say that any insufficiency deadlock stems from an unsatisfiable flow dependence. Even though our over-synchronization scheme may cause the program to deadlock earlier, we do not introduce additional deadlocks.

Weak functional deadlock state

The characterizations of weak functional deadlock states and of general CDDF weak deadlock states are very similar, and their proofs identical, so we present them at the same time. Note that this definition is very general, thanks to the continuation activation modelling the future schedule of data in streams, and encompasses all types of deadlocks.

Lemma 3.25 (Weak (functional) deadlock state)**.** *All deadlocks result from the presence of a cycle between task activations based on the order relation corresponding to the deadlock type.*

For a CDDF program in state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$, we have:

$$\begin{aligned}
WD(\sigma) \quad &\Leftrightarrow \quad NEXT(\mathcal{K}_e) \notin \Pi \ \wedge \ \mathcal{A}_o \neq \{\mathcal{C}(\mathcal{K}_e)\} \\
&\qquad\qquad \wedge \ \exists a \in \mathcal{A}_o, a \neq \mathcal{C}(\mathcal{K}_e), \quad a <^+_{\mathcal{A}_o} a \ \vee \ \mathcal{C}(\mathcal{K}_e) <^+_{\mathcal{A}_o} a \\
WFD(\sigma) \quad &\Leftrightarrow \quad NEXT(\mathcal{K}_e) \notin \Pi \ \wedge \ \mathcal{A}_o \neq \{\mathcal{C}(\mathcal{K}_e)\} \\
&\qquad\qquad \wedge \ \exists a \in \mathcal{A}_o, a \neq \mathcal{C}(\mathcal{K}_e), \quad a \ \delta^+_{\mathcal{A}_o} \ a \ \vee \ \mathcal{C}(\mathcal{K}_e) \ \delta^+_{\mathcal{A}_o} \ a
\end{aligned}$$

Proof of Lemma 3.25. The proof of both propositions is very similar, so we only present the first one. We first show that, in a weak deadlock state, we can always either find a dependence cycle or a dependence chain starting at the continuation.

Let us consider state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$:

$$\begin{aligned}
WD(\sigma) \quad &\Rightarrow \quad \exists \sigma', \ D(\sigma') \ \wedge \ \mathcal{K}_e = \mathcal{K}'_e \\
&\Rightarrow \quad \exists \sigma', \ \big(NEXT(\mathcal{K}_e) \notin \Pi\big) \ \wedge \ \big(\mathcal{A}'_o \neq \{\mathcal{C}(\mathcal{K}_e)\}\big) \ \wedge \ \big(\forall a \in \mathcal{A}'_o, \neg \mathcal{A}'_e \ltimes a\big)
\end{aligned}$$

We use Definition 3.11 of the stream prefix order in $\forall a \in \mathcal{A}'_o, \neg \mathcal{A}'_e \ltimes a$, which yields:

$$\forall a \in \mathcal{A}'_o, \exists (R, s, i) \in a, \exists j \leqslant i, \ \forall a' \in \mathcal{A}'_e, \ (W, s, j) \notin a'$$

Proposition 3.10 ensures that all stream accesses are covered by an activation in $\mathcal{A}'_e \cup \mathcal{A}'_o$:

$$\bigcup_{a \in \mathcal{A}'_e \cup \mathcal{A}'_o} a = \mathcal{X}$$

This allows us to deduce that, since (W, s, j) is not present in activations from $\mathcal{A}'_e$, it

necessarily belongs to an activation in $\mathcal{A}'_o$:

$$\begin{aligned} & \forall a \in \mathcal{A}'_o, \exists (R, s, i) \in a, \exists j \leqslant i,\ \exists a' \in \mathcal{A}'_o,\ (W, s, j) \in a' \\ \Leftrightarrow \quad & \forall a \in \mathcal{A}'_o, \exists a' \in \mathcal{A}'_o : \quad a' < a \end{aligned}$$

We can recursively build an chain $... < a'' < a' < a$ by applying this last proposition to a, then a' which satisfies $a' < a$ and so on. As $\mathcal{A}'_o$ is a finite set, we deduce that either we stop once we reach the continuation activation or there must be at least one cycle. As $\mathcal{A}'_o \neq \{\mathcal{C}(\mathcal{K}_e)\}$, we can choose $a \neq \mathcal{C}(\mathcal{K}_e)$, which concludes the proof.

The proof of the reverse simply requires building a state σ' with $\mathcal{A}'_o = \{\mathcal{C}(\mathcal{K}_e), a_1, a_2, ..., a_n\}$, where $a_1 ... a_n$ are the activations forming the cycle or the dependence chain, which is a deadlock state as the cycle or the dependence on the continuation prevents any activation to be executable:

$$\forall a \in \mathcal{A}'_o, \left(\exists a' \in \mathcal{A}'_o,\ a' < a\right) \quad \Rightarrow \quad \neg\ \mathcal{A}'_e \ltimes a$$

□

Hierarchy of weak deadlock states

Based on Proposition 3.16 and the Lemmas 3.24 and 3.25, we can provide a hierarchy of weak deadlock states. It is important to stress the fact that any given state can satisfy the weak deadlock condition for multiple types of deadlocks, possibly in a completely independent manner. If such is the case, the stronger deadlock property is naturally taken into account.

Proposition 3.26 (Weak deadlock state hierarchy)**.** *Any state σ of a CDDF program satisfies:*

$$WID(\sigma) \quad \Rightarrow \quad WFD(\sigma) \quad \Rightarrow \quad WD(\sigma)$$

And the definition of spurious deadlocks is preserved for weak deadlock states:

$$WSD(\sigma) \quad \Leftrightarrow \quad WD(\sigma)\ \wedge\ \neg WFD(\sigma)$$

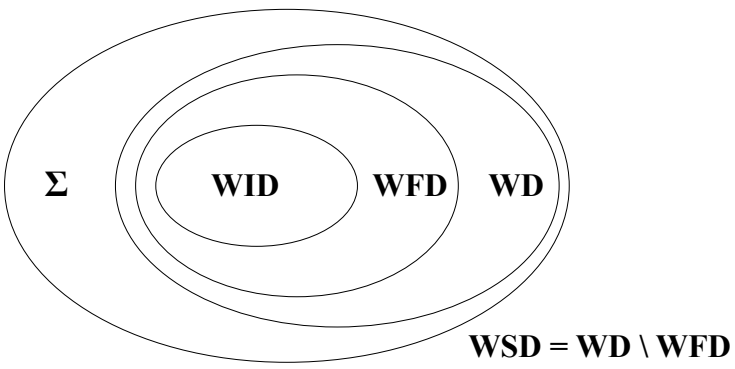


Figure 3.3: Hierarchy of weak deadlock states.

Figure 3.3 illustrates this hierarchy.

Proof. We prove Proposition 3.26 in two parts. First we show that $WID(\sigma) \Rightarrow WFD(\sigma)$, which is a direct result of Lemma 3.21:

$$WID(\sigma) \;\Rightarrow\; \exists\sigma', ID(\sigma') \;\Rightarrow\; FD(\sigma') \;\Rightarrow\; WFD(\sigma)$$

To prove that $WFD(\sigma) \Rightarrow WD(\sigma)$, we need only consider Lemmas 3.14 and 3.19, which respectively provide the characterization of a deadlock and a functional deadlock state.

$$\begin{aligned}
WFD(\sigma) &\Rightarrow \exists\sigma', FD(\sigma') \\
&\Rightarrow \exists\sigma', \big(NEXT(\mathcal{K}_e) \notin \Pi\big) \;\wedge\; \big(\mathcal{A}'_o \neq \{\mathcal{C}(\mathcal{K}_e)\}\big) \;\wedge\; \big(\forall a \in \mathcal{A}'_o, \neg\mathcal{A}'_e \triangleright a\big) \\
&\Rightarrow \exists\sigma', \big(NEXT(\mathcal{K}_e) \notin \Pi\big) \;\wedge\; \big(\mathcal{A}'_o \neq \{\mathcal{C}(\mathcal{K}_e)\}\big) \;\wedge\; \big(\forall a \in \mathcal{A}'_o, \neg\mathcal{A}'_e \ltimes a\big) \\
&\Rightarrow \exists\sigma', D(\sigma') \\
&\Rightarrow WD(\sigma)
\end{aligned}$$

On the third line, we use Proposition 3.16 which gives $\mathcal{A}'_e \ltimes a \Rightarrow \mathcal{A}'_e \triangleright a$ or conversely $\neg(\mathcal{A}'_e \triangleright a) \Rightarrow \neg(\mathcal{A}'_e \ltimes a)$.

□

In the remainder of this chapter, we analyze the properties of the CDDF model. In particular, we provide static and dynamic conditions under which our model does not introduce spurious deadlocks in programs and we prove the serializability and the determinism of programs in this model.

3.3 Stream Causality in CDDF Programs

This section starts by introducing the notion of stream causality and a characterization of stream causal programs based on the existence of stream causal schedules. In a second step, we prove that CDDF programs that are stream causal in each state whenever the control program reaches a barrier are free of all forms of deadlocks.

3.3.1 Stream Causality

As our model is asynchronous, we cannot use a global notion of time as a basis for the definition of causality. Instead, we can use streams as a set of independent local clocks. Each stream can be considered to define its own time based on access indices, each task activation representing a synchronization point between the clocks of the streams it writes to and, to a lesser extent, those it reads from. As write accesses to streams are exclusive in our single dynamic assignment model, we can define the *Stream Clock* (SC) of an activation as its set of write accesses. More importantly, this allows us to define a precedence relation between task activations producing data in the same stream. This relation is a subset of a total order relation, which is the control program order.

Definition 3.27 (Stream clock precedence relation)**.** We define a reflexive and antisymmetric binary relation $\preccurlyeq_{sc} \in \mathcal{P}(\mathcal{A}^2)$ on task activations as:

$$\forall (a, a') \in A^2,\ a \preccurlyeq_{sc} a' \quad \triangleq \quad \exists s \in \mathcal{S}, \exists i, j \in \mathbb{N},\ j \leqslant i\ \wedge\ (W, s, j) \in a\ \wedge\ (W, s, i) \in a'$$

In any state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$ of a CDDF program, we extend this relation, in the case of *sink* task activations (i.e., that contain no write accesses and would therefore not be in any relation wrt. stream clocks) in order to ensure that the continuation activation is a maximum:

$$\forall a \in \mathcal{A}_e \cup \mathcal{A}_o : \qquad a \preccurlyeq_{sc} \mathcal{C}(\mathcal{K}_e)$$

Remark 3.28. *This relation is antisymmetric by definition of the activation point evaluation function ξ (Definition 3.8), as it enforces the same order of attribution of write access indexes in all streams. The underlying order relation is the precedence of activation points in the control program trace: if task activations a, a' are the result of the evaluation of activation points π, π', then the order of occurrence of these activation points in the control program trace determines the direction of the precedence relation $\preccurlyeq_{sc}$:*

$$a \preccurlyeq_{sc} a' \quad \Rightarrow \quad \pi = \pi'\ \vee\ \pi \to \pi'$$

Which can be used to re-define the $\preccurlyeq_{sc}$ relation, using the notations from Definition 3.4:

$$a \preccurlyeq_{sc} a' \quad \Leftrightarrow \quad \mathcal{O}(a) \cap \mathcal{O}(a') \neq \varnothing\ \wedge\ (\pi = \pi'\ \vee\ \pi \to \pi')$$

Flow dependence chains between task activations can naturally be assimilated to causality chains, and we use them to this effect in our definition of causality, but they are not sufficient in our model. Flow dependences alone are sufficient when dealing with shared memory communication, but in the case of streaming it is necessary to also take into account the communication channels, or stream clocks in our case. We define stream causality in CDDF programs as the absence of time reversal inside causality chains.

Definition 3.29 (Stream causality)**.** A CDDF stream $s \in \mathcal{S}$ is causal in a state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$ of a program if the stream's local clock is positive along all causality chains:

$$\forall a, a' \in \mathcal{A}_e \cup \mathcal{A}_o,\ \big(a \neq a'\ \vee\ a' \neq \mathcal{C}(\mathcal{K}_e)\big), s \in \mathcal{O}(a) \cap \mathcal{O}(a'),\ \neg\Big(a \preccurlyeq_{sc} a'\ \wedge\ a'\ \delta^+_{(\mathcal{A}_e \cup \mathcal{A}_o)}\ a\Big)$$

The restriction $a \neq a'\ \vee\ a' \neq \mathcal{C}(\mathcal{K}_e)$ is necessary as the case $a = a' = \mathcal{C}(\mathcal{K}_e)$ always invalidates the expression. Indeed, $\mathcal{C}(\mathcal{K}_e) \preccurlyeq_{sc} \mathcal{C}(\mathcal{K}_e)$ and $\mathcal{C}(\mathcal{K}_e)\ \delta\ \mathcal{C}(\mathcal{K}_e)$ are always true, which validates our intuition that the continuation activation is inherently non-causal. However, as long as the continuation activation is only found at the end of causality chains, it does not preclude stream causality.

A CDDF program is stream causal in a state σ if each stream in the program is causal. We note:

$$SC(\sigma) \quad \triangleq \quad \forall a, a' \in \mathcal{A}_e \cup \mathcal{A}_o,\ \big(a \neq a'\ \vee\ a' \neq \mathcal{C}(\mathcal{K}_e)\big), \quad \neg\Big(a \preccurlyeq_{sc} a'\ \wedge\ a'\ \delta^+_{(\mathcal{A}_e \cup \mathcal{A}_o)}\ a\Big)$$

Remark 3.30. *It is important to note that stream causality is entirely determined by the schedule of data in streams and not by the execution schedule of task activations. For this reason, the causal nature of a CDDF program state only depends on the control program's trace in that state. This can be seen in Definition 3.29 of stream causality, where no difference is made between executed and outstanding activations.*

As stream causality only depends on the control program trace, this property is deterministic for a given program as long as the control program itself is deterministic.

Remark 3.31 (Barrier modelization)**.** *A second important remark is that our choice of modeling the behaviour of barriers through the continuation activation is entirely consistent with this definition of stream causality. Indeed, as we have noted above, the stream clock of this activation is greater than that of any task activation generated by the control program, which is consistent with its role as a model of the possible continuation of the program after the barrier passes. If any data produced before the barrier may be consumed afterwards, the order will be strict, which is consistent with the semantics of barriers in our model.*

The restriction, in the Definition 3.29 of stream causality, that we do not check the causality of the barrier itself, which appears as $a \neq a' \ \vee \ a' \neq \mathcal{C}(\mathcal{K}_e)$, means that the current state of the program does not allow us to deduce whether its possible future may or may not be causal.

Stream causality violations correspond to conflicts between flow dependences and stream clocks. In order to model causal program schedules, we define the minimal constraints for a scheduling function that respects both forward time in each stream and flow dependences.

Definition 3.32 (Stream causal schedule)**.** A scheduling function $\theta : A \to \mathbb{N}$ enforces a stream causal schedule iff:

$$\forall (a, a') \in A : \quad \begin{cases} a \ \delta^+ \ a' & \Rightarrow \quad \theta(a) < \theta(a') \\ a \preccurlyeq_{sc} a' & \Rightarrow \quad \theta(a) \leqslant \theta(a') \end{cases}$$

We use the existence of causal schedules, irrespectively of their capacity to enforce the stream prefix order, to characterize stream causality in CDDF programs. For this reason, we do not pay any attention to the task activations that have already been executed in a given state as the execution schedule up to that state may not be consistent with a causal schedule. In general, we only need to know whether a causal schedule exists, but we do not enforce it on the execution of task activations.

Proposition 3.33 (Causal CDDF program)**.** *A CDDF program is causal in a state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$ if and only if the program admits at least one causal schedule in that state.*

Proof. We first prove that program causality is necessary to the existence of a causal schedule, according to our definitions. Let us assume, by way of contradiction, that the program is not causal in a state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$:

$$\exists a, a' \in \mathcal{A}_e \cup \mathcal{A}_o, \ \big(a \neq a' \ \vee \ a \neq \mathcal{C}(\mathcal{K}_e)\big), \quad a \preccurlyeq_{sc} a' \ \wedge \ a' \ \delta^+_{(\mathcal{A}_e \cup \mathcal{A}_o)} \ a$$

Which means that any scheduling function θ, that enforces a causal schedule in this state (Definition 3.32), meets the following constraints:

$$(\theta(a) \leqslant \theta(a')) \ \wedge \ (\theta(a') < \theta(a))$$

This contradiction allows us to conclude that there can be no causal schedule for a non causal program.

Let us now show that program causality is sufficient to ensure the existence of a causal schedule. Before constructing a causal schedule, we verify that this kind of schedule is indeed consistent with stream causality. Assume the program is causal in state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$:

$$\forall a, a' \in \mathcal{A}_e \cup \mathcal{A}_o, \ \big(a \neq a' \ \vee \ a \neq \mathcal{C}(\mathcal{K}_e)\big), \quad \neg\Big(a \preccurlyeq_{sc} a' \ \wedge \ a' \ \delta^+_{(\mathcal{A}_e \cup \mathcal{A}_o)} \ a\Big)$$

This yields the following constraints on any causal scheduling function:

$$\neg\Big((\theta(a) \leqslant \theta(a')) \ \wedge \ (\theta(a') < \theta(a))\Big) \quad \Leftrightarrow \quad (\theta(a') < \theta(a)) \ \vee \ (\theta(a) \leqslant \theta(a'))$$

These constraints are always satisfiable, which ensures that a causal schedule is possible for a causal program in a state σ.

Let us consider a scheduling function θ that enforces all flow dependences. This schedule, which is obtained by using the $(EXEC_{\triangleright})$ transition rule, satisfies, by definition:

$$\forall a, a' \in \mathcal{A}_e \cup \mathcal{A}_o : \quad a \ \delta^+_{(\mathcal{A}_e \cup \mathcal{A}_o)} \ a' \quad \Rightarrow \quad \theta(a) < \theta(a')$$

The program is stream causal in state σ, $SC(\sigma)$, so there cannot be any flow dependence cycles between activations as $a \preccurlyeq_{sc} a$ is always true for activations that contain at least one write access, which is necessarily the case if an activation is the source of a flow dependence:

$$\exists a \in \mathcal{A}_e \cup \mathcal{A}_o, a \neq \mathcal{C}(\mathcal{K}_e), \ a \ \delta^+_{(\mathcal{A}_e \cup \mathcal{A}_o)} \ a \quad \Rightarrow \quad \neg SC(\sigma)$$

Furthermore, as all activations are in a stream clock precedence relation with the continuation activation:

$$\exists a \in \mathcal{A}_e \cup \mathcal{A}_o, a \neq \mathcal{C}(\mathcal{K}_e), \ \mathcal{C}(\mathcal{K}_e) \ \delta^+_{(\mathcal{A}_e \cup \mathcal{A}_o)} \ a \quad \Rightarrow \quad \neg SC(\sigma)$$

As there can be no cycles and no dependence chains containing the continuation activation, Lemma 3.25 allows us to conclude that this state is not a weak functional deadlock state, irrespectively of the fact that the control program has reached a barrier or not. This means that this schedule built on $(EXEC_{\triangleright})$ is able to schedule all task activations in $\mathcal{A}_e \cup \mathcal{A}_o$. As a side note, this already proves that a CDDF program that is stream causal every time the control program reaches a barrier cannot experience functional deadlocks.

Let us verify that, in a stream causal program state, this schedule is itself stream

causal. From the definition of stream causality, we have:

$$\forall a, a' \in \mathcal{A}_e \cup \mathcal{A}_o,\ \big(a \neq a' \ \vee \ a \neq \mathcal{C}(\mathcal{K}_e)\big), \quad a \preccurlyeq_{sc} a' \quad \Rightarrow \quad \neg(a' \ \delta^+_{(\mathcal{A}_e \cup \mathcal{A}_o)} \ a)$$

And by definition of the schedule:

$$\forall a, a' \in \mathcal{A}_e \cup \mathcal{A}_o : \quad a \ \delta^+_{(\mathcal{A}_e \cup \mathcal{A}_o)} \ a' \quad \Rightarrow \quad \theta(a) < \theta(a')$$

We deduce that:

$$\begin{aligned} \forall a, a' \in \mathcal{A}_e \cup \mathcal{A}_o,\ \big(a \neq a' \ \vee \ a \neq \mathcal{C}(\mathcal{K}_e)\big), \quad a \preccurlyeq_{sc} a' \quad &\Rightarrow \quad \neg(a' \ \delta^+_{(\mathcal{A}_e \cup \mathcal{A}_o)} \ a) \\ &\Rightarrow \quad \neg(\theta(a') < \theta(a)) \\ &\Rightarrow \quad \theta(a) \leqslant \theta(a') \end{aligned}$$

Which concludes the proof: θ, and any schedule enforcing flow dependences, is indeed a stream causal schedule for the program in state σ. □

3.3.2 Deadlock-Freedom in Stream Causal CDDF Programs

Without any restriction, spurious deadlocks may occur in CDDF programs due to the over-approximation, and therefore over-synchronization, of data dependences. As the model allows absolute freedom in the communication patterns between task activations, we can build a schedule that leads to a state where a CDDF program is in a deadlock, but not in a functional deadlock. This situation occurs, for instance, in the following example.

Example 3.3.1 (Spurious deadlock in a CDDF program)**.** *Consider a program state* $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$ *where:*

$$\mathcal{A}_e = \{a_1\} \qquad \textit{where} \qquad a_1 = \{(W, s_1, 1)\}$$

$$\mathcal{A}_o = \{a_2, a_3, a_4\} \qquad \textit{where} \qquad \begin{cases} a_2 = \{(W, s_1, 0), (R, s_2, 0\} \\ a_3 = \{(R, s_1, 1), (W, s_2, 0)\} \\ a_4 = \{(R, s_1, 0)\} \end{cases}$$

Figure 3.4 shows the true dependences δ as well as the additional constraints $<$ introduced by our synchronization scheme on stream prefixes. It also marks, for the discussion, the stream clock relation $\preccurlyeq_{sc}$ between activations a_2 and a_1.

The following flow dependences are present in this program state:

$$a_1 \ \delta \ a_3 \qquad a_3 \ \delta \ a_2 \qquad a_2 \ \delta \ a_4$$

As flow dependences allow the simple schedule (a_3, a_2, a_4) to complete the execution, considering that a_1 has already been executed, there is no functional deadlock. However, the addition of the stream prefix constraint leads to a cycle between activations a_2 and a_3, and therefore to a deadlock when the control program reaches a barrier:

$$a_1 < a_3 \qquad a_3 < a_2 \qquad a_2 < a_4 \qquad \textit{and} \qquad a_2 < a_3$$

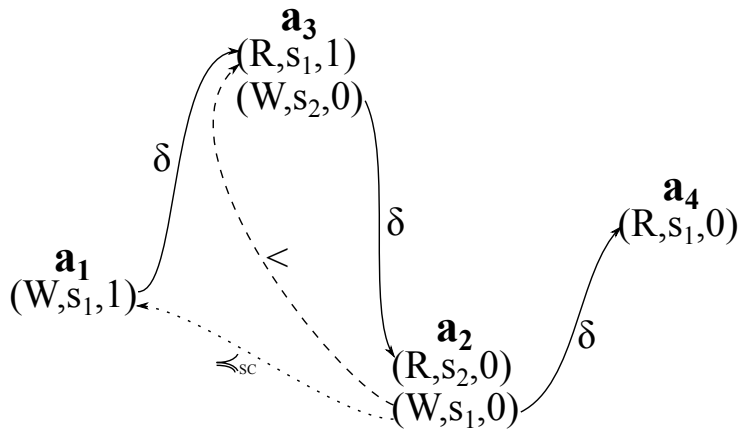


Figure 3.4: Example of a spurious deadlock in a CDDF program.

The intuition we get from this example is that a stream causality violation, between activations a_1 and a_2, was the necessary factor that allowed this spurious deadlock to occur:

$$\begin{array}{lllll} a_2 \preccurlyeq_{sc} a_1 & \wedge & a_1 \ \delta \ a_3 \ \delta \ a_2 & \Rightarrow & \theta(a_1) < \theta(a_2) \leqslant \theta(a_1) \\ & & & \Rightarrow & \neg SC(\sigma) \end{array}$$

□

Though stream causality is a strong condition that is neither amenable to static analysis, nor fit as a programming model restriction because of its complexity, it is a semantically important condition for deadlock-freedom, and for some strong forms of serializability, as we show in Section 3.7.

Theorem 3.34. *A CDDF program is free of all deadlocks if it is stream causal in each state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$ where the control program reaches a barrier or terminates, $NEXT(\mathcal{K}_e) \notin \Pi$.*

A CDDF program's state σ always satisfies:

$$SC(\sigma) \quad \Rightarrow \quad \neg WD(\sigma)$$

Proof. We prove this theorem by showing that $SC(\sigma) \ \wedge \ WD(\sigma)$ is impossible.

Let $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$ be the state of a CDDF program that satisfies $SC(\sigma) \wedge WD(\sigma)$.

From Proposition 3.10, we know that:

$$\bigcup_{a \in \mathcal{A}_e \cup \mathcal{A}_o} a = \mathcal{X} \quad \Rightarrow \quad \forall (u,s,i) \in \mathcal{X}, \exists a \in \mathcal{A}_e \cup \mathcal{A}_o, \ (u,s,i) \in a$$

We add this expression to the Definition 3.11 of the stream prefix order relation $<$, which allows us to rewrite the stream prefix relation in terms of the flow dependence and

the stream clock relations:

$$\begin{aligned}
&\forall a, a' \in \mathcal{A}_e \cup \mathcal{A}_o,\ a < a' \\
&\Leftrightarrow \quad \exists (s,i) \in \mathcal{S} \times \mathbb{N}, \exists j \leqslant i,\ (W,s,j) \in a \ \wedge\ (R,s,i) \in a' \\
&\Leftrightarrow \quad \exists (s,i) \in \mathcal{S} \times \mathbb{N}, \exists j \leqslant i,\ (W,s,j) \in a \ \wedge\ (R,s,i) \in a' \ \wedge\ (\exists a'' \in \mathcal{A}_e \cup \mathcal{A}_o, (W,s,i) \in a'') \\
&\Leftrightarrow \quad \exists a'' \in \mathcal{A}_e \cup \mathcal{A}_o,\ a \preccurlyeq_{sc} a'' \ \wedge\ a'' \ \delta\ a'
\end{aligned}$$

As the program is causal in this state, Proposition 3.33 guarantees the existence of at least one causal schedule in this state. Let θ be a scheduling function for such a schedule. By Definition 3.32 of a causal schedule, we have:

$$\forall a, a' \in \mathcal{A}_e \cup \mathcal{A}_o : \quad \begin{cases} a\ \delta^+\ a' & \Rightarrow \quad \theta(a) < \theta(a') \\ a \preccurlyeq_{sc} a' & \Rightarrow \quad \theta(a) \leqslant \theta(a') \end{cases}$$

We use our previous result and the definition of the causal schedule to determine the scheduling function constraints with respect to the stream prefix order:

$$\begin{aligned}
\forall a, a' \in \mathcal{A}_e \cup \mathcal{A}_o, \quad a < a' \quad &\Rightarrow \quad \exists a'' \in \mathcal{A}_e \cup \mathcal{A}_o,\ a \preccurlyeq_{sc} a'' \ \wedge\ a'' \ \delta\ a' \\
&\Rightarrow \quad \exists a'' \in \mathcal{A}_e \cup \mathcal{A}_o,\ \theta(a) \leqslant \theta(a'') \ \wedge\ \theta(a'') < \theta(a') \\
&\Rightarrow \quad \theta(a) < \theta(a')
\end{aligned}$$

As the state σ corresponds to a weak deadlock, Lemma 3.25 yields:

$$\begin{aligned}
WD(\sigma) \quad &\Rightarrow \quad \exists a \in \mathcal{A}_o, a \neq \mathcal{C}(\mathcal{K}_e), \quad a <^+_{\mathcal{A}_o} a \ \vee\ \mathcal{C}(\mathcal{K}_e) <^+_{\mathcal{A}_o} a \\
&\Rightarrow \quad \exists (a_1, ..., a_n) \in \mathcal{A}_o^n, a_1 \neq \mathcal{C}(\mathcal{K}_e), \big(a_1 < ... < a_n < a_1\big) \ \vee\ \big(\mathcal{C}(\mathcal{K}_e) < a_1 < ... < a_n\big)
\end{aligned}$$

As we have proved that $a < a' \ \Rightarrow\ \theta(a) < \theta(a')$, we use this constraint in the task activation dependence chains:

$$a_1 < a_2 < ... < a_n < a_1 \quad \Rightarrow \quad \theta(a_1) < \theta(a_2) < ... < \theta(a_n) < \theta(a_1) \quad \Rightarrow \quad \theta(a_1) < \theta(a_1)$$

$$\mathcal{C}(\mathcal{K}_e) < a_1 < ... < a_n \quad \Rightarrow \quad \theta(\mathcal{C}(\mathcal{K}_e)) < \theta(a_1) < ... < \theta(a_n) \quad \Rightarrow \quad \theta(\mathcal{C}(\mathcal{K}_e)) < \theta(a_n)$$

Both results are contradictory. The first result, $\theta(a_1) < \theta(a_1)$ is trivially impossible. The second result is contradictory because $a_n \preccurlyeq_{sc} \mathcal{C}(\mathcal{K}_e)$ is always true, which means that $\theta(a_n) \leqslant \theta(\mathcal{C}(\mathcal{K}_e))$, thus in contradiction with the result $\theta(\mathcal{C}(\mathcal{K}_e)) < \theta(a_n)$. These contradictions conclude the proof.

□

3.4 Task Causality and Sufficient Deadlock-Freedom Conditions

As we have seen, stream causality, in states where the control program reaches a barrier or terminates, is a sufficient condition for deadlock-freedom in CDDF programs. However, stream causality is, at best, very difficult to prove statically. Though less semantically important, we need to provide some alternative deadlock-freedom conditions, or at least spurious deadlock-freedom conditions, that: (1) fit the programming model and the semantic restrictions that come from the base languages, and (2) are more practical either for automatic static and dynamic analysis, or even for programming rules to be enforced by developers. In this section, we present some alternative conditions for deadlock-freedom and the formal framework for their analysis.

3.4.1 CDDF Tasks

We have avoided, until now, the additional complexity of introducing a definition of tasks in the CDDF model, as they do not play an important role. The definition we provide here is purposefully weaker than that of the programming model as we solely focus on communication and we overlook the work functions. The mapping of the OpenMP stream-computing extension onto the CDDF model is discussed in Chapter 5.

Definition 3.35 (CDDF task)**.** A task is an equivalence class in the set of task activations based on the following equivalence relation $\sim \in \mathcal{P}(\mathcal{A}^2)$:

$$\forall (a, a') \in \mathcal{A}^2 : \qquad a \sim a' \quad \triangleq \quad \begin{cases} \forall (u, s, i) \in a, \exists i' \in \mathbb{N},\ (u, s, i') \in a' \\ \forall (u, s, j') \in a', \exists j \in \mathbb{N},\ (u, s, j) \in a \end{cases}$$
$$\triangleq \quad \mathcal{I}(a) = \mathcal{I}(a') \ \wedge \ \mathcal{O}(a) = \mathcal{O}(a')$$

Two activations are equivalent in this relation iff they access exactly the same streams, both for input and for output. This definition disregards, in the OpenMP streaming instantiation of this model, the work function and therefore the task pragma that is associated with a task activation.

The set of tasks in a CDDF program in a state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$ is:

$$T(\sigma) = (\mathcal{A}_e \cup \mathcal{A}_o \setminus \{\mathcal{C}(\mathcal{K}_e)\})_{/\sim}$$

For a task activation a, we write $[a]_\sim$ its equivalence class on the $\sim$ relation. This equivalence class is the CDDF task to which a belongs.

Using this definition, we can build the task graph of a program state. We note $S(\sigma)$ the set of streams used by a program in state σ. It is simply defined as the set of streams

accessed by any activation in that state:

$$\begin{aligned} S(\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o) &\triangleq \Big\{ s \in \mathcal{S} \mid \exists u \in \{R, W\}, \exists i \in \mathbb{N}, \exists a \in \mathcal{A}_e \cup \mathcal{A}_o \setminus \{\mathcal{C}(\mathcal{K}_e)\},\ (u, s, i) \in a \Big\} \\ &\triangleq \bigcup_{a \in \mathcal{A}_e \cup \mathcal{A}_o \setminus \{\mathcal{C}(\mathcal{K}_e)\}} \Big(\mathcal{I}(a) \cup \mathcal{O}(a) \Big) \end{aligned}$$

Definition 3.36 (Task graph)**.** A CDDF program in a state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$ can be represented as a directed hypergraph:

$$H(\sigma) = \big(T(\sigma), S(\sigma) \big)$$

where tasks are vertices and streams are hyperedges connecting a set of producer tasks to a set of consumer tasks.

For a program in state σ and a stream $s \in S(\sigma)$, we define the sets of producer tasks $P(\sigma, s)$ and of consumer tasks $C(\sigma, s)$ as:

$$\begin{aligned} P(\sigma, s) &= \Big\{ [a]_\sim \in T(\sigma) \mid s \in \mathcal{O}(a) \Big\} \\ C(\sigma, s) &= \Big\{ [a]_\sim \in T(\sigma) \mid s \in \mathcal{I}(a) \Big\} \end{aligned}$$

Note that, despite the fact that, by definition, $\mathcal{C}(\mathcal{K}_e)$ is a producer and consumer of each stream, we do not count the continuation activation in any producer or consumer set.

This task graph clearly depends on the current state of the control program, and is therefore dynamic. While it could appear useless for static analysis, it is possible to build a static version that over-approximates the program taskgraph in all states of the program. We show, in Section 5.3.3, that this can easily be achieved in the case of our stream-computing extension to OpenMP. This type of static task graph will be presented in Chapter 5, where we further show how the deadlock-freedom conditions that use the dynamic task graph can be mapped onto the static one.

3.4.2 Task Causality

In our model, tasks are sets of activations that share the same input and output streams. We will derive, in the same way as stream clocks, a second precedence relation, called *task order*, on task activations from the control program order. The stream access indices of activations define a total order on the activations *within the task*. This order is induced by the definition of the activation point evaluation function ξ and the total order of activation points in the trace of the program. However, this relation's extension to the whole set of task activations would not be an order relation as it lacks transitivity.

Definition 3.37 (Task order)**.** Task activations belonging to the same task in a program state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$ are ordered by the relation $\preccurlyeq_T \in \mathcal{P}(\mathcal{A}^2)$:

$$\forall (a, a') \in \mathcal{A}_e \cup \mathcal{A}_o,$$
$$a \preccurlyeq_T a' \quad \triangleq \quad \big(a \sim a'\big) \ \wedge \ \big(\forall s \in \mathcal{I}(a) \cup \mathcal{O}(a), \exists i, j \in \mathbb{N}, \ j \leqslant i \ \wedge \ (u, s, j) \in a \ \wedge \ (u, s, i) \in a'\big)$$

This relation is transitive as we restrict it to $a \sim a'$, reflexive because we use $j \leqslant i$ and antisymmetric by construction of task activations in the evaluation function ξ.

This order relation may appear to be artificial at this point, especially if we were to require that the execution schedule of task activations respect this order. However, and we will develop this discussion later, this corresponds to a property provided by the semantics of the OpenMP language. Note also that this order relation is very similar to the stream clock relation, $\preccurlyeq_{sc}$, that we introduced in the previous section, which was used to define stream causality. We will use here the task order relation to define task causality in the same way.

Definition 3.38 (Task causality)**.** A CDDF task is causal in a state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$ of a program if the task's activations appear in task order along all causality chains:

$$\forall t \in T(\sigma), \ \forall a, a' \in t, \quad \neg\Big(a \preccurlyeq_T a' \quad \wedge \quad (a' \ \delta^+_{(\mathcal{A}_e \cup \mathcal{A}_o)} \ a)\Big)$$

A CDDF program is task causal in a state σ if each task in the program is causal in that state. We note:

$$TC(\sigma) \quad \Leftrightarrow \quad \forall a, a' \in \mathcal{A}_e \cup \mathcal{A}_o \setminus \{\mathcal{C}(\mathcal{K}_e)\}, \quad \neg\Big(a \preccurlyeq_T a' \quad \wedge \quad (a' \ \delta^+_{(\mathcal{A}_e \cup \mathcal{A}_o)} \ a)\Big)$$

Note that this definition is similar to process monotonicity in Kahn process networks, though here the equivalent notion of process is slightly different. Indeed, our definition requires that no flow dependence chain goes back in task order, which would mean that the execution of task activation a' enables the latter execution of a. Because $a \preccurlyeq_T a'$, the outputs of a are written in the prefix of the outputs of a', so the result of adding additional data on the input streams of task $[a]_\sim$ can result in more than just additional outputs, which violates Kahn causality.

It is easy to see that this property is weaker than stream causality, because the stream clock relation is defined across task activations producing data in a given stream, even if these activations are not in the same task:

$$\begin{aligned} \forall a, a' \in \mathcal{A} : \qquad a \preccurlyeq_{sc} a' \quad &\Rightarrow \quad \exists s \in \mathcal{S}, \exists i, j \in \mathbb{N}, \ j \leqslant i \ \wedge \ (W, s, j) \in a \ \wedge \ (W, s, i) \in a' \\ &\Rightarrow \quad \big(a \sim a' \ \Rightarrow \ a \preccurlyeq_T a'\big) \end{aligned}$$

The only exception are sink tasks, that are only consumers, and in which activations are only in a stream clock precedence relation with the continuation activation. We extended the stream clocks relation in that case, but task causality does not allow the same simplification. Indeed, as we have discussed above, the continuation activation does

not belong in any task, because tasks are defined as equivalence classes of task activations and the continuation activation is the only activation that has an infinite number of input and output streams. For this reason, the continuation activation can *never* be in a task order relation $\preccurlyeq_T$ with task activations.

Definition 3.39 (Task causal schedule)**.** A scheduling function $\theta : \mathcal{A} \to \mathbb{N}$ enforces a task causal schedule iff:

$$\forall a, a' \in \mathcal{A} : \quad \begin{cases} a \ \delta^+ \ a' & \Rightarrow \quad \theta(a) < \theta(a') \\ a \preccurlyeq_T a' & \Rightarrow \quad \theta(a) \leqslant \theta(a') \end{cases}$$

Proposition 3.40 (Task causal CDDF program)**.** *A CDDF program is task causal in a weak insufficiency deadlock free state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$ if and only if the program admits at least one task causal schedule in that state.*

Proof. The proof of this proposition is very close to that of Proposition 3.33, with a simple substitution of relations and properties. The only difference comes from the fact that contrary to stream clocks, task order does not allow ordering task activations with respect to the continuation activation. This means that the existence of a dependence chain starting at the continuation activation does not allow to conclude to a violation of causality:

$$\exists a \in \mathcal{A}_e \cup \mathcal{A}_o, a \neq \mathcal{C}(\mathcal{K}_e), \quad \mathcal{C}(\mathcal{K}_e) \ \delta^+_{(\mathcal{A}_e \cup \mathcal{A}_o)} \ a \quad \not\Rightarrow \quad \neg TC(\sigma)$$

For this reason, the application of Lemma 3.25 only allows to conclude that, in a task causal state σ, all functional deadlocks are insufficiency deadlocks. The existence of the schedule depends on the absence of the weak insufficiency deadlock condition $\mathcal{C}(\mathcal{K}_e) \ \delta^+_{(\mathcal{A}_e \cup \mathcal{A}_o)} \ a$. □

Even though task causality is not a sufficient condition, it constitutes a fair starting point for searching new deadlock-freedom conditions. As this property is required for program correctness in the programming model, it naturally falls under the programmer's responsibility.

3.4.3 Statically Analyzable Condition for Spurious Deadlock-Freedom

Theorem 3.41. *A CDDF program can only experience insufficiency deadlocks if it is task causal and each stream in the program is either single-producer or single-consumer. In each state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$:*

$$TC(\sigma) \ \wedge \ \Big(\forall s \in S(\sigma) : \ |P(\sigma, s)| = 1 \ \vee \ |C(\sigma, s)| = 1\Big) \quad \Rightarrow \quad WID(\sigma) \ \vee \ \neg WD(\sigma)$$

Proof. Let us consider the state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$ of a program that satisfies these conditions. We need to show that:

$$TC(\sigma) \ \wedge \ \Big(\forall s \in S(\sigma) : \ |P(\sigma, s)| = 1 \ \vee \ |C(\sigma, s)| = 1\Big) \ \wedge \ WD(\sigma) \quad \Rightarrow \quad WID(\sigma)$$

We use the results from Lemmas 3.25 and 3.24 instead of the definition to characterize weak deadlock states. Merging the two, we obtain:

$$WD(\sigma) \quad \Rightarrow \quad WID(\sigma) \quad \vee \quad \Big(\exists a \in \mathcal{A}_o, a \neq \mathcal{C}(\mathcal{K}_e), \quad a <^+_{\mathcal{A}_o} a\Big)$$

Merging the two expressions shows that our objective is to prove that no task activation cycle on the stream prefix order relation, $<$ (Definition 3.11), can exist in such states:

$$TC(\sigma) \wedge \Big(\forall s \in S(\sigma) : \ |P(\sigma, s)| = 1 \vee |C(\sigma, s)| = 1\Big) \quad \Rightarrow \quad \neg\Big(\exists a \in \mathcal{A}_o, a \neq \mathcal{C}(\mathcal{K}_e), \ a <^+_{\mathcal{A}_o} a\Big)$$

Let us assume, by way of contradiction, that the program is in a weak deadlock state σ, but not in a weak insufficiency deadlock, and therefore that $\exists a \in \mathcal{A}_o, a \neq \mathcal{C}(\mathcal{K}_e), \ a <^+_{\mathcal{A}_o} a$. As the program is also task causal in this state, it admits a task causal schedule. Let θ be such a schedule.

Let us now expand this stream prefix order cycle:

$$\exists a \in \mathcal{A}_o, a \neq \mathcal{C}(\mathcal{K}_e), \exists n \in \mathbb{N}, n > 0, \exists (a_1, ..., a_n) \in \mathcal{A}_o^n, \quad \big(a_1 = a_n = a\big) \ \wedge \ \big(a_1 < ... < a_n\big)$$

From the definition of the stream prefix order, we deduce that:

$$\forall k \in [1, n-1], \ a_k < a_{k+1} \quad \Rightarrow \quad \exists (s, i) \in \mathcal{S} \times \mathbb{N}, \exists j \leqslant i, \ (W, s, j) \in a_k \ \wedge \ (R, s, i) \in a_{k+1}$$

The stream s is either single-producer or single-consumer, so we will handle the cases separately.

If $|P(\sigma, s)| = 1$, then we look at the producer task activation a_k and its equivalence class. Either the precise write operation (W, s, i) belongs to a task activation in this class, or this operation is not yet scheduled and therefore belongs to the continuation activation:

$$\Big(\exists a'_k \in [a_k]_\sim, (W, s, i) \in a'_k\Big) \qquad \vee \qquad (W, s, i) \in \mathcal{C}(\mathcal{K}_e)$$

If the write operation is in the continuation activation, then we would have $\mathcal{C}(\mathcal{K}_e)\delta a_{k+1}$, which is a weak insufficiency deadlock condition and contradicts the hypothesis.

We can therefore assume that $(W, s, i) \in a'_k$ and $a'_k \ \delta \ a_{k+1}$. We deduce that:

$$a_k \preccurlyeq_T a'_k \ \wedge \ a'_k \ \delta \ a_{k+1} \qquad \Rightarrow \qquad \theta(a_k) \leqslant \theta(a'_k) \ \wedge \ \theta(a'_k) < \theta(a_{k+1})$$

We conclude that, in the case of a single-producer stream connecting a_k and a_{k+1}, we have:

$$a_k < a_{k+1} \qquad \Rightarrow \qquad \theta(a_k) < \theta(a_{k+1})$$

If $|C(\sigma, s)| = 1$, then we will look at the consumer task activation a_{k+1}, instead, and its equivalence class. In this case, as the higher read access operation (R, s, i) has already been scheduled to a task activation, the lower index operation (R, s, j) is guaranteed to have also been scheduled, so we know that:

$$\exists a'_{k+1} \in [a_{k+1}]_{\sim}, \quad (R, s, j) \in a'_{k+1}$$

So we have $a_k \ \delta \ a'_{k+1}$ and $a'_{k+1} \preccurlyeq_T a_{k+1}$, which trivially leads to:

$$\theta(a_k) < \theta(a'_{k+1}) \quad \wedge \quad \theta(a'_{k+1}) \leqslant \theta(a_{k+1}) \qquad \Rightarrow \qquad \theta(a_k) < \theta(a_{k+1})$$

By aggregating the results of the two cases, we conclude the proof with the contradiction:

$$a_1 < ... < a_n \quad \Rightarrow \quad \theta(a_1) < ... < \theta(a_n) \quad \Rightarrow \quad \theta(a) < \theta(a)$$

□

3.4.4 Condition Ensuring that Only Insufficiency Deadlocks Occur

In the absence of cycles in the program hypergraph, we can show that only insufficiency deadlocks can occur and Proposition 3.26 guarantees such deadlocks are never spurious deadlocks. While this condition may appear restrictive, it is satisfied by a wide class of streaming applications. Kudlur and Mahlke analyze a dozen such applications in [40] and find that the pattern never occurs and limit their study to acyclic task graphs.

Theorem 3.42. *A CDDF program can only have insufficiency deadlocks in a state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$ if its hypergraph $H(\sigma)$ contains no strongly connected components in that state:*

$$H(\sigma) \text{ contains no cycles} \quad \Rightarrow \quad WID(\sigma) \ \vee \ \neg WD(\sigma)$$

With the following definition, using Definition 3.36 for the set of tasks producing a stream s in state σ, $P(\sigma, s)$, and the set of consumer tasks $C(\sigma, s)$:

$$H(\sigma) \text{ contains no cycles}: \quad \forall n \in \mathbb{N}, n > 0, \nexists (s_1, ..., s_n) \in S(\sigma)^n \mid \begin{cases} C(\sigma, s_1) \cap P(\sigma, s_2) \neq \varnothing \\ C(\sigma, s_2) \cap P(\sigma, s_3) \neq \varnothing \\ ... \\ C(\sigma, s_n) \cap P(\sigma, s_1) \neq \varnothing \end{cases}$$

Proof. The proof of this theorem derives from Lemma 3.24, which characterizes a weak insufficiency deadlock state, and Lemma 3.25, characterizing weak deadlock states as dependence cycles. To prove this theorem, we show below that:

$$H(\sigma) \text{ contains no cycles} \ \wedge \ \neg WID(\sigma) \quad \Rightarrow \quad \neg WD(\sigma)$$

Let σ be a CDDF program state satisfying $H(\sigma)$ contains no cycles $\wedge$ $\neg WID(\sigma)$. Lemma 3.24 yields:

$$\neg WID(\sigma) \quad \Leftrightarrow \quad NEXT(\mathcal{K}_e) \in \Pi \quad \vee \quad (\nexists a \in \mathcal{A}_o,\ a \neq \mathcal{C}(\mathcal{K}_e) \ \wedge \ \mathcal{C}(\mathcal{K}_e) < a)$$

As $NEXT(\mathcal{K}_e) \in \Pi$ gives $\neg WD(\sigma)$, we only keep the condition $\nexists a \in \mathcal{A}_o,\ a \neq \mathcal{C}(\mathcal{K}_e) \ \wedge \ \mathcal{C}(\mathcal{K}_e) < a$, which means that $\mathcal{C}(\mathcal{K}_e)$ cannot appear in any task activation cycle in this state.

We need to show that the absence of cycles in the taskgraph $H(\sigma)$ means that there can be no task activation cycles in $\mathcal{A}_e \cup \mathcal{A}_o \setminus \{\mathcal{C}(\mathcal{K}_e)\}$. This is intuitively simple as the taskgraph is an over-approximation of the communication patterns in task activations where we lose the precise information on stream access indexes.

Let us assume by way of contradiction that there is a cycle of task activations in $\mathcal{A}_e \cup \mathcal{A}_o \setminus \{\mathcal{C}(\mathcal{K}_e)\}$:

$$\exists (a_1, ..., a_n) \in (\mathcal{A}_e \cup \mathcal{A}_o \setminus \{\mathcal{C}(\mathcal{K}_e)\})^n \quad | \quad a_1 < a_2 < ... < a_n \ \wedge \ a_n = a_1$$

Each stream prefix order relation $<$ translates, by Definition 3.11, into:

$$\forall k \in [1, n-1] : \quad a_k < a_{k+1} \quad \Leftrightarrow \quad \exists (R, s_k, i_k) \in a_{k+1}, \exists j_k \leqslant i_k,\ (W, s_k, j_k) \in a_k$$

From the previous relation, we deduce that the set of streams $(s_1, s_2, ...s_{n-1})$ satisfies:

$$\begin{cases} (R, s_1, i_1) \in a_2 \ \wedge \ (W, s_2, j_2) \in a_2 & \Rightarrow \ C(\sigma, s_1) \cap P(\sigma, s_2) = \{a_2\} \neq \varnothing \\ (R, s_2, i_2) \in a_3 \ \wedge \ (W, s_3, j_3) \in a_3 & \Rightarrow \ C(\sigma, s_2) \cap P(\sigma, s_3) = \{a_3\} \neq \varnothing \\ ... & \\ (R, s_n, i_n) \in a_n = a_1 \ \wedge \ (W, s_1, j_1) \in a_1 & \Rightarrow \ C(\sigma, s_n) \cap P(\sigma, s_1) = \{a_1\} \neq \varnothing \end{cases}$$

This contradicts the hypothesis of no cycles in $H(\sigma)$ and therefore concludes the proof. □

In the general case, this condition cannot be statically verified because of the dynamic nature of the task graph. We will see that building a static over-approximation of the task graph is generally easy in our streaming extension to OpenMP. If no strongly connected components are present in the static taskgraph, we also know that there are none in the dynamic taskgraph for any state of the program, so the result holds, as we show in Section 5.3.3.

3.4.5 Weaker Statically Analyzable Sufficient Condition for Spurious Deadlock-Freedom

The two conditions presented in Theorems 3.41 and 3.42 can be easily analyzed statically, even though in some cases only a conservative over-approximation may be evaluated at compilation time. One of the drawbacks of both conditions is that they restrict the expressiveness of the model, either excluding multi-producer multi-consumer streams or

excluding cyclic communication patterns. One way to mitigate this issue is to merge the results of the two theorems and require that only the disjunction of the two conditions be true overall. The resulting condition is much weaker as it only excludes the use of multi-producer multi-consumer streams for cyclic communication patterns. The additional condition of task causality is generally available by default in the case of OpenMP programs, but we keep this condition as tight as we can by requiring task causality only within strongly connected components of the program's taskgraph.

Corollary 3.43. *A CDDF program is free of all but insufficiency deadlocks if all tasks belonging to a strongly connected component of the program's taskgraph are causal and each stream belonging to a strongly connected component is either single-producer or single-consumer.*

A CDDF program in state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$ *satisfies:*

$$WID(\sigma) \ \vee \ \neg WD(\sigma) \ \Leftarrow \ \begin{cases} \forall (s_1, s_2, ..., s_n) \in S(\sigma)^n \ \mid \ \begin{cases} C(\sigma, s_1) \cap P(\sigma, s_2) \neq \varnothing \\ C(\sigma, s_2) \cap P(\sigma, s_3) \neq \varnothing \\ ... \\ C(\sigma, s_n) \cap P(\sigma, s_1) \neq \varnothing \end{cases} \\ \forall k \in [1, n] : \quad \Big(|P(\sigma, s_k)| = 1 \ \vee \ |C(\sigma, s_k)| = 1 \Big) \\ \forall t \in \bigcup_{k \in [1, n-1]} \big(C(\sigma, s_k) \cap P(\sigma, s_{k+1}) \big) \cup \big(C(\sigma, s_n) \cap P(\sigma, s_1) \big), \\ \qquad\qquad \forall (a, a') \in t^2 : \ \neg \Big(a' \preccurlyeq_T a \ \wedge \ (a \ \delta^+_{(\mathcal{A}_e \cup \mathcal{A}_o)} \ a') \Big) \end{cases}$$

Proof. The proof relies on the same principles as that of Theorems 3.41 and 3.42. We start, similarly to the proof of Theorem 3.41, by noting that applying Lemmas 3.25 and 3.24 allows to prove the following instead:

$$Hypothesis \quad \Rightarrow \quad \neg \Big(\exists a \in \mathcal{A}_o, a \neq \mathcal{C}(\mathcal{K}_e), \ a <^+_{\mathcal{A}_o} a \Big)$$

We assume, by way of contradiction, that the program is in a weak deadlock state, but not in a weak insufficiency deadlock, in state σ, and therefore that $\exists a \in \mathcal{A}_o, a \neq \mathcal{C}(\mathcal{K}_e), \ a <^+_{\mathcal{A}_o} a$. We expand this stream prefix order cycle:

$$\exists a \in \mathcal{A}_o, a \neq \mathcal{C}(\mathcal{K}_e), \exists n \in \mathbb{N}, n > 0, \exists (a_1, ..., a_n) \in \mathcal{A}_o^n, \quad \big(a_1 = a_n = a \big) \ \wedge \ \big(a_1 < ... < a_n \big)$$

Based on this expression, we use the same reasoning as in the proof of Theorem 3.42 to deduce the existence of a strongly connected component in the program's taskgraph based on the tasks $(t_1, t_2, ..., t_n) = ([a_1]_\sim, [a_2]_\sim, ..., [a_n]_\sim)$ to which belong the activations $(a_1, a_2, ..., a_n)$ and the set of streams $(s_1, s_2, ..., s_{n-1})$ that connect these tasks, from the definition of the stream prefix precedence relation $<$.

By hypothesis of our theorem, these sets of tasks and streams verify the following

expressions:

$$\forall k \in [1, n-1] : \quad \Big(|P(\sigma, s_k)| = 1 \ \vee \ |C(\sigma, s_k)| = 1 \Big) \tag{3.1}$$

$$\forall k \in [1, n], \forall (a, a') \in t_k^2 : \quad \neg \Big(a' \preccurlyeq_T a \ \wedge \ (a \ \delta^+_{(\mathcal{A}_e \cup \mathcal{A}_o)} \ a') \Big) \tag{3.2}$$

Our objective will be to show, in a similar manner to the end of the proof of Theorem 3.41, that the existence of the task activations cycle along with the single-producer or single-consumer property on the streams connecting these activations, from expression (3.1), lead to a violation of the task causality property (3.2).

We remark that if any a_k is actually the continuation activation, then the we directly obtain a contradiction as we would have an weak insufficiency deadlock condition. Furthermore, if a non-insufficiency functional deadlock occurs, then Lemma 3.25 allows to characterize the state by a cycle $\exists a \in \mathcal{A}_o, a \neq \mathcal{C}(\mathcal{K}_e), \ a \ \delta^+_{\mathcal{A}_o} \ a$, and as this can only happen in a strongly connected component of the taskgraph, this would violate the causality hypothesis as $a \preccurlyeq_T a$ is always true.

This ensures that, under the current hypotheses, the same schedule θ, that we have used previously, built on the $(EXEC_{\triangleright})$ rule, and that satisfies $\forall a, a' \in \mathcal{A}, a \ \delta \ a' \ \Rightarrow \theta(a) < \theta(a')$, is a possible schedule in this state. We can further deduce, from the task causality property (3.2) that:

$$\begin{array}{lll} \forall k \in [1, n], \forall (a, a') \in t_k^2, \quad a \preccurlyeq_T a' & \Rightarrow & \neg (a' \ \delta^+_{(\mathcal{A}_e \cup \mathcal{A}_o)} \ a) \\ & \Rightarrow & \neg (\theta(a') < \theta(a)) \\ & \Rightarrow & \theta(a') \geqslant \theta(a) \end{array}$$

The schedule is therefore causal at least for the tasks of the same strongly connected component. We can follow the remainder of the proof of Theorem 3.41, with the disjunction of the two cases, of single-producer or single-consumer streams, for all of the streams $s_{k,k\in[1,n-1]}$, which shows that, in both cases, we either get a contradiction because of a weak insufficiency deadlock condition or the following holds:

$$a_k < a_{k+1} \qquad \Rightarrow \qquad \theta(a_k) < \theta(a_{k+1})$$

Which concludes the proof with the contradiction:

$$a_1 < ... < a_n \quad \Rightarrow \quad \theta(a_1) < ... < \theta(a_n) \quad \Rightarrow \quad \theta(a) < \theta(a)$$

□

3.5 Functional and Deadlock Determinism

Determinism in programming environments is a major asset for productivity, but it is often missing in parallel programming environments. Sequential programming languages, like C, C++ and Fortran, the underlying languages of OpenMP, provide functional

determinism, while OpenMP itself does not. Indeed, race conditions due to shared memory communication are non-deterministic.

One of the most important properties of our computational model, and it is one of the design goals of our programming model, is to guarantee functional determinism. In this section we prove that our model guarantees not only functional determinism, but also that deadlocks occur deterministically, which is highly valuable for debugging.

Remark 3.44. *We focus in this section on data produced and consumed in streams, which means that functional determinism, in a way similar to Kahn process networks, is defined as producing the same output* inside streams *for a given input.*

In this section, we make the two following assumptions in order to prove determinism.

1. We assume that either no shared memory communication happens, or that it does not constitute a source of indeterminism (i.e. a race condition). This assumption is natural as we only allow shared memory communication as a convenience and under the programmer's responsibility.

2. We assume that a task that declares writing in a stream does indeed define all of the elements it declares, leaving no undefined values in streams. If a task declares producing a given amount of data for a given output stream, but either does not write this data or writes less than the amount specified, then undefined memory locations may be read by consumers, breaking all functional determinism guarantees.

3.5.1 Deterministic Task Activations

Until now, we have disregarded the work functions of tasks, modelling task activations as a set of stream accesses where the actual data stored inside the stream is unimportant. Work functions are relevant in the current context because determinism concerns both data placement and the data itself. As a simplifying assumption, we postulate that all work functions are inherently deterministic. This is not self-evident, in particular if we were to allow nested streaming constructions or in the case where tasks have dynamic burst rates, but it is of little interest here. The functional determinism of work functions is directly derived from that of the underlying sequential programming language.

We will therefore consider task activations to be deterministic in the sense that, irrespectively of the location of the stream accesses involved, their work functions are deterministic. As a result, for a given set of stream accesses characterizing the activation, if the data present in the input streams at the locations of all read accesses is the same, the data written on all output streams at the locations of write accesses will be the same.

The key insight is that if the data placement in all streams is deterministic, then the work functions' determinism will be sufficient to guarantee functional determinism for the program.

3.5.2 Deterministic Data Schedule in Streams

In our model, the schedule of data in streams is entirely determined by the control program. This choice is made in Section 3.2 specifically to guarantee determinism. As the

control program is a sequential and, by hypothesis, deterministic process, the schedule of data will also be deterministic.

Lemma 3.45 (Deterministic data schedule)**.** *For a CDDF program with a given input, the schedule of data is deterministic in all streams.*

Proof. We have modeled the control program's execution, see Definition 3.8, with the oracle function $NEXT$, which in the case of a sequential program is simply the sequence operator, and the evaluation function ξ that builds task activations from activation points, determining the stream access indices of each task activation and therefore the data schedule in streams.

Our axiomatic hypothesis is that the oracle function is deterministic[5], so for a given execution trace $\mathcal{K}_e$ the next operation that the control program executes, $NEXT(\mathcal{K}_e)$, is always the same. As a result, the control program trace of a CDDF program is always the same at termination when the control program executes sequentially.

We conclude the proof by remarking that, for a given CDDF program, the evaluation function ξ only depends on the control program's execution trace to build the next task activation.

Note that while the execution of task activations is only ordered by stream and barrier synchronization, the generation of task activations is sequential, which ensures that the same task activation will be assigned the production of data for a given stream location in all executions. □

3.5.3 Program Functional Determinism

CDDF programs are deterministic by design, relying on a sequential deterministic control program to orchestrate the schedule of data rather than that of the execution.

Theorem 3.46. *CDDF programs, where the control program executes sequentially, are deterministic.*

More specifically, for a given input, the data produced by the program in all streams is the same in a given program state σ, regardless of the execution schedule of task activations. The state of streams is a function of the program state and, in particular, the final state is a function of the initial state.

Proof. The proof of this theorem is based on Lemma 3.45 and the hypothesis that work functions are deterministic and that the two assumptions made at the beginning of this Section 3.5 hold.

One can notice that, as mentioned in Section 3.2, stream memory behaves as single dynamic assignment, where the only possible race condition is if a read operation could occur before the single write operation to a given location. The ordering requirement to avoid races are given by the flow dependence relation δ. As the stream prefix order subsumes flow dependence requirements, see Proposition 3.16, the order enforced by our stream synchronization scheme is stricter than necessary to avoid this type of races. It follows that there are no race conditions in a CDDF programs' stream communication, so this does not constitute a source of non-determinism.

[5]We relax this hypothesis when we parallelize the control program in Section 4.3.

Let us assume, by way of contradiction, that there is a state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$ where two different executions can lead to different data in some stream location. Let $(s, i) \in \mathcal{S} \times \mathbb{N}$ be the location where this occurs and $a_p \in \mathcal{A}_e$ the task activation that produced the data, so $(W, s, i) \in a_p$. Lemma 3.45 guarantees that the same task activation produces the piece of data for a given location since this is not dependent on the execution schedule.

As we consider that all work functions are deterministic, the execution of a_p can only produce different data for location (s, i) if its inputs differ between the two executions. So there exists $(R, s', i') \in a_p$ such that the value read by a_p at location (s', i') differs between the two executions.

As $a_p \in \mathcal{A}_e$, the execution rule (EXEC) guarantees that we have $\mathcal{A}_e \ltimes a_p$. By Definition 3.11, of the stream prefix order, there is another task activation $a'_p \in \mathcal{A}_e$ such that $(W, s', i') \in a'_p$ and trivially $a_p \neq a'_p$ or we would have a functional deadlock, which would contradict the hypothesis that $a_p \in \mathcal{A}_e$, as the activation cannot execute in that case.

We can recursively build an infinite chain of task activations, causally tracing upstream the source of the inconsistency, and all elements in this chain are in $\mathcal{A}_e$, which is a finite set, built by the execution of task activations. We deduce that there must be a cycle in the chain, which once again leads to an impossibility. Any cycle of flow dependences represents a functional deadlock, see Lemma 3.25, and all activations in the cycle cannot have executed, so they do not belong in $\mathcal{A}_e$. This contradiction concludes the proof. □

3.5.4 Deadlock Determinism

Theorem 3.47 (Deterministic deadlocks)**.** *CDDF programs deadlock deterministically: for a given input set, the program either does not deadlock or it deadlocks in the same state in all executions.*

Proof. To prove this theorem, it is sufficient to show that if a CDDF program admits, for a given input, an execution schedule in which a deadlock occurs, then that deadlock state is the maximum state that can and will be reached in all executions.

Let us consider such a program that deadlocks in state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$ with a schedule modeled by a scheduling function θ. Let us now assume, by way of contradiction, that there exists another schedule θ' that, irrespectively of deadlocks, allows the program to execute up to a state $\sigma' = (\mathcal{K}'_e, \mathcal{A}'_e, \mathcal{A}'_o)$ such that $|\sigma| < |\sigma'|$.

By definition, this means that $|\mathcal{K}_e| + |\mathcal{A}_e| < |\mathcal{K}'_e| + |\mathcal{A}'_e|$, which leads to the following two relevant cases to handle: (1) traces have different lengths, $|\mathcal{K}_e| < |\mathcal{K}'_e|$; or (2) the set of executed task activations is different, $|\mathcal{A}_e| < |\mathcal{A}'_e|$.

Case (1): $|\mathcal{K}_e| < |\mathcal{K}'_e|$. As σ is a deadlock state, we know from Lemma 3.14 that $NEXT(\mathcal{K}_e) \notin \Pi$. As we have argued in the proof of Lemma 3.45, the control program trace is necessarily deterministic for the same input, so $|\mathcal{K}_e| < |\mathcal{K}'_e|$ trivially implies that $\mathcal{K}_e$ is a prefix of $\mathcal{K}'_e$. We deduce that $NEXT(\mathcal{K}_e) = barrier$, as the trace continues, and therefore that in the schedule θ' the program was able to pass the barrier while in schedule θ, it was unable to make further progress.

From Lemma 3.14, we also know that $\mathcal{A}_o \neq \{\mathcal{C}(\mathcal{K}_e)\}$. As $\mathcal{K}_e$ is a prefix of $\mathcal{K}'_e$ and task

activations are created deterministically, we have:

$$\Big(\pi \in \mathcal{K}_e \quad \Rightarrow \quad \pi \in \mathcal{K}'_e\Big) \qquad \Leftrightarrow \qquad \Big(\quad a \in \mathcal{A}_e \cup \mathcal{A}_o \setminus \{\mathcal{C}(\mathcal{K}_e)\} \quad \Rightarrow \quad a \in \mathcal{A}'_e \cup \mathcal{A}'_o \setminus \{\mathcal{C}_{\mathcal{K}'_e}\}\Big)$$

In order to pass the barrier all activations in $\mathcal{A}_e \cup \mathcal{A}_o \setminus \{\mathcal{C}(\mathcal{K}_e)\}$ must be executed. As in the second execution the barrier has been able to pass, and $\mathcal{A}_o \neq \{\mathcal{C}(\mathcal{K}_e)\}$, we have:

$$\exists a \in \mathcal{A}_o \setminus \{\mathcal{C}(\mathcal{K}_e)\},\ a \in \mathcal{A}'_e$$

The rest of the proof is similar with the second case, where we first prove this same property before continuing.

Case (2): $|\mathcal{A}_e| < |\mathcal{A}'_e|$. If at this point $|\mathcal{K}_e| < |\mathcal{K}'_e|$, then we just consider this to fall in Case (1), so we consider here that $|\mathcal{K}_e| \geqslant |\mathcal{K}'_e|$. Using the same reasoning as above, we deduce that:

$$a \in \mathcal{A}'_e \cup \mathcal{A}'_o \setminus \{\mathcal{C}_{\mathcal{K}'_e}\} \qquad \Rightarrow \qquad a \in \mathcal{A}_e \cup \mathcal{A}_o \setminus \{\mathcal{C}(\mathcal{K}_e)\}$$

We also trivially have:

$$|\mathcal{A}_e| < |\mathcal{A}'_e| \quad \Rightarrow \quad \exists a \in \mathcal{A}'_e,\ a \notin \mathcal{A}_e$$

From these two expressions we also get the same property as in Case (1):

$$\exists a \in \mathcal{A}'_e, \quad a \in \mathcal{A}_o \setminus \{\mathcal{C}(\mathcal{K}_e)\}$$

Cases (1) and (2) continued.

The last property we have from Lemma 3.14 is that $\forall a \in \mathcal{A}_o, \neg(\mathcal{A}_e \ltimes a)$, which is true in particular for the activation we have been able to find in both cases:

$$\exists a \in \mathcal{A}_o \setminus \{\mathcal{C}(\mathcal{K}_e)\} \quad | \quad \big(a \in \mathcal{A}'_e\big) \ \wedge \big(\neg(\mathcal{A}_e \ltimes a)\big)$$

As $a \in \mathcal{A}'_e$, this activation was executed in the second execution schedule, so its dependences were satisfied by some subset of $\mathcal{A}'_e$ that does not contain a'. Let $\mathcal{A}''_e \subsetneq \mathcal{A}'_e$ be the smallest set of activations such that $\mathcal{A}''_e \ltimes a$. As $\neg(\mathcal{A}_e \ltimes a)$, we deduce that:

$$\exists a' \in \mathcal{A}''_e \quad | \quad a' < a \ \wedge \ a' \notin \mathcal{A}_e$$

If $a' \notin \mathcal{A}_o \ \wedge \ a' < a$ then the barrier was impossible to pass as the deadlock was an insufficiency deadlock. If $a' \in \mathcal{A}_o$, then we found the same property that allowed us to continue from cases (1) and (2):

$$\exists a' \in \mathcal{A}''_e \quad | \quad a' \in \mathcal{A}_o \setminus \{\mathcal{C}(\mathcal{K}_e)\}$$

We can recursively apply this reasoning and, as $\mathcal{A}''_e \subsetneq \mathcal{A}'_e$, build an infinite sequence of strictly decreasing sets of task activations, which is impossible and therefore concludes this proof.

The fact that this particular maximum state is actually reached in all executions is trivially deduced by swapping the role of states in the proof. If any execution cannot reach the state σ, but stops in state σ'' instead with $|\sigma''| < |\sigma|$, then the proof can be applied using σ'' as the base deadlock state. □

3.5.5 Determinism, Productivity and Portability

An important result of Theorems 3.47 and 3.46 is that, when debugging a CDDF program, deadlock reproducibility is absolute. In most parallel programming environments, running with some kind of code instrumentation, or in a debugger, leads to reduced deadlock or race reproducibility. In our model, this cannot happen.

Even more, our model guarantees that if a program experiences a deadlock-free run for a given input set, there can be no deadlocks in that program for that input set. This is true irrespectively of the underlying architecture, so application testing becomes equivalent to that of sequential applications: changing the level of concurrency (or communication latencies, or bandwidth, etc.) of the execution platform does not result in observing new spurious race conditions or deadlocks. A test suite that provides proper code coverage is therefore as efficient for uncovering errors in CDDF programs as it is for finding errors in sequential programs.

Lemma 3.45 and Theorem 3.46 further guarantee that stopping the control program (e.g. breakpoint) will lead to a deterministic whole program state. This property in itself is very useful to facilitate the debugging process. The programmer can rely on this guarantee to deterministically observe the program state by stopping the control program, and allowing the outstanding task activations to execute until quiescence is reached[6].

These properties therefore strongly improve the productivity and portability of applications that fit the CDDF model:

- Program testing is as effective as for sequential programs, independently of the architecture used for testing.
- Errors can be deterministically reproduced, like in sequential programs.
- Deadlocks occur deterministically, on all execution platforms.
- Programs can be interrupted deterministically for debugging.

3.6 Strict Consistency

Current architectures offer increasingly relaxed memory consistency models [3]. This choice is motivated by performance concerns, but it makes parallel programming all the more complicated. Current architectures do no longer offer sequential consistency (SC) [42], but more relaxed and complex memory models like x86 [34,50,64], Sun's total store order (TSO) or relaxed memory order (RMO) [65], or IBM's relaxed memory model

[6]Quiescence occurs when no further progress can be made. The time necessary to reach that point is undecidable, but finite as we have seen in the discussion of weak deadlock states.

for Power [63]. Programmers can no longer ignore these issues and must insert memory fences with varying patterns depending on the architecture [5], which makes parallel programming ever more complicated and races more elusive.

As we have discussed, in Section 2.4.6, OpenMP does not shield programmers from this additional complexity. Stream-computing, in general, allows to altogether hide this issue from developers. Because of the (perceived) *single dynamic assignment* behaviour of stream accesses, and the synchronization mechanisms enforcing flow dependences, all read operations in streams are guaranteed to return the last, and unique, value written at a given index in a stream. This is the strongest possible memory consistency guarantee, strict consistency [30], therefore allowing programmers to focus on algorithmic problems rather than on architecture-dependent issues.

In the case of CDDF programs, strict consistency is only guaranteed for stream accesses. As our model allows shared memory communication, it is still possible to be affected by relaxed memory consistency issues. Our position is that disallowing shared memory communication is too restrictive and would go against the OpenMP specification. Instead, we advocate only using shared memory if no stream communication patterns can be substituted.

3.7 Serializability

Serializability is a very important property for parallel programs as it allows efficient execution on non-concurrent platforms and sequential functional testing. In our case, as we have discussed in Section 3.5, the latter is much less relevant because of the strong determinism guarantees our model offers. Nevertheless, sequential execution of CDDF programs is relevant and we need to ensure it is possible.

Intuitively, this should mostly be a question of whether there is a possible interference between the implicit synchronization of stream accesses of different task activations that would require an interleaving of operations that cannot be achieved in a sequential schedule of the execution of task activations.

3.7.1 Dynamic Sequential Schedule

Proposition 3.48 (Serializability)**.** *All deadlock-free CDDF programs admit at least one sequential schedule.*

Proof. This property is almost self-evident from the definition of the execution rules on Figure 3.1, where the (EXEC) rule aggregates all the dependences of task activations in the $\ltimes$ relation, therefore only allowing the execution of task activations that have all their dependences satisfied before they can start executing. This means that a task activation behaves atomically, it cannot enable the execution of another task activation until all of its own dependences are satisfied.

Let us consider that, in a CDDF program in a state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$, there are two outstanding task activations $(a, b) \in \mathcal{A}_o^2$ that cannot be serialized, so it is impossible to schedule a before b or b before a. By definition of the scheduling constraints, we trivially

have $a <^+ b$ and $b <^+ a$, which means that there is a cycle $a <^+ a$ and therefore a deadlock according to Lemma 3.25. This contradicts the deadlock-freedom hypothesis.

A trivial sequential schedule, in an unbounded memory abstraction, consists in running the control program until it reaches a barrier or terminates, then executing all task activations in any order allowed by the $\ltimes$ relation, then repeat until termination. □

If we only consider correct programs, so excluding all programs that have functional deadlocks, we can therefore guarantee serializability under the same conditions as spurious deadlock-freedom. However, this type of serialization is more akin to the dynamic scheduling of some ad hoc user-level threads, or fibers. This scheme incurs scheduling runtime overhead and disables many compiler optimizations that would apply to a static sequential schedule.

3.7.2 Static Sequential Schedule

While serializability can be guaranteed without requiring strong conditions on CDDF programs, it is much harder to guarantee the existence of a static serial schedule, let alone providing an automatic way of building such a schedule. There is, however, a trivial static sequential schedule for which we can provide a necessary and sufficient condition: the control program order. This schedule is semantically important in the case of OpenMP, where this serialization strategy is the default strategy advocated in the specification.

Theorem 3.49. *A CDDF program can be sequentially executed in control program order if and only if it is stream causal in each state.*

Proof. Given the hypothesis that the program is stream causal in each state, we can add a barrier after each activation point in the control program and this barrier is guaranteed, by Theorem 3.34, to be able to pass, which means that each newly generated task activation can execute immediately.

Conversely, if the program can execute sequentially, with the execution of each task activation tied to the generation of the task activation, then the program can admit one barrier after each activation point without deadlocking. As the stream clock order is a subset of the order relation defined by the control program order, the execution of task activations was possible in strictly positive stream clock order for all streams in the program, while also respecting flow dependences, which is by definition a stream causal schedule, which guarantees, using Proposition 3.33, that the program is stream causal in each state. □

This result is of great practical relevance, in particular for OpenMP-based instantiations of this model, as it provides a necessary and sufficient condition for the default type of serializability, and this condition also provides the strongest deadlock-freedom property. It also provides closure with respect to OpenMP semantics and therefore establishes our definition of stream causality as founded, at least for our OpenMP extension incarnation of the CDDF model.

In practice, we will also rely on static analysis to determine, when possible, if the program falls in one of the categories where previous work provides ways to statically compute schedules, like in the case of synchronous [43] or cyclo-static [13] dataflow programs.

Remark 3.50. *This approach is the usual serialization strategy for OpenMP programs, where serialization is achieved by ignoring the OpenMP compiler directives during program compilation, therefore compiling the semantically equivalent underlying sequential program.*[7]

In the case of our OpenMP extension for streaming, merging the execution of an activation point and the execution of the activation is very similar to ignoring (or stripping) the program annotations. However, this is not possible for streaming programs, where streams are still required to buffer the data if the production/consumption rates are not a statically perfect match. Only the programs that verify synchronous data-flow conditions can be simply stripped of streaming annotations. Still, the necessary code generation is trivial if the annotations cannot be simply ignored.

3.8 Summary of the Properties and Associated Conditions

In order to give a better idea of what guarantees can be obtained in CDDF programs, we present on Table 3.1 the properties provided by each of the program conditions we have analyzed in this section.

Condition on state	Deadlock Freedom properties				Serializability		Determinism	Strict
$\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$	$\neg D(\sigma)$	$\neg ID(\sigma)$	$\neg FD(\sigma)$	$\neg SD(\sigma)$	Dyn. order	CP	Func^{al} & Dlock	Consist.
$TC(\sigma) \quad \wedge$ $\forall s \in SCC(H(\sigma)),$ $\neg MPMC(s)$	no	no	yes	yes	if $\neg ID(\sigma)$	no	yes	yes
$TC(\sigma) \quad \wedge$ $\forall s, \neg MPMC(s)$	no	no	yes	yes	if $\neg ID(\sigma)$	no	yes	yes
$SCC(H(\sigma)) = \varnothing$	no	no	yes	yes	if $\neg ID(\sigma)$	no	yes	yes
$SC(\sigma) \quad \vee$ $NEXT(\mathcal{K}_e) \in \Pi$	yes	yes	yes	yes	yes	no	yes	yes
$\forall \sigma, SC(\sigma)$	yes	yes	yes	yes	yes	yes	yes	yes

Table 3.1: Properties of CDDF programs.

Table 3.1 presents the conditions required to provide the various properties guaranteed in our model: $D(\sigma)$, $ID(\sigma)$, $FD(\sigma)$ and $SD(\sigma)$ respectively mean that the program can experience a deadlock[8], an insufficiency deadlock, a functional deadlock or a spurious deadlock. The conditions range from the weakest to the strongest. Note that each condition provides spurious deadlock freedom, which is one of our main objectives, as well as functional and deadlock determinism and strict consistency. In the following, we describe the conditions, which are presented in an abbreviated form in the table, and we discuss the results.

[7]The semantical equivalence is not actually a requirement, but a desirable property. When the annotations cannot be ignored without changing the semantics of a program, that program is considered non-serializable and the resulting behaviour unspecified.

[8]This general deadlock property is the disjunction of the three other properties.

1. The first, and weakest, condition comes from Corollary 3.43. It requires that no stream belonging to a strongly connected component (SCC) of the hypergraph $H(\sigma)$ be both multi-producer and multi-consumer (MPMC), though different streams can be either one or the other. It also requires that the program be task causal in that state, which is stronger than the original condition where only the tasks belonging to a SCC need to be task causal. Programs can only experience insufficiency deadlocks, and their absence would further guarantee serializability.

   Note that insufficiency deadlocks are easier to analyze as it is only necessary to verify the amounts of data produced and consumed on each stream, regardless of the indexes of stream accesses. We stress the fact that this condition does not provide a blanket freedom of functional deadlocks, but only freedom from the subset of functional deadlocks that are not insufficiency deadlocks.

2. The second condition corresponds to Theorem 3.41. It requires that no stream in the program be multi-producer and multi-consumer (MPMC) in that state and that the program be task causal, $TC(\sigma)$. Programs meeting these requirements can only experience insufficiency deadlocks and, similarly to the first condition, can benefit from serializability when no insufficiency deadlocks occur.

3. The third condition, from Theorem 3.42, requires the hypergraph $H(\sigma)$ to be free of strongly connected components. This condition is sufficient to avoid non-insufficiency deadlocks.

4. The fourth condition requires stream causality of the program state whenever the control program reaches a barrier or terminates. This strong condition is sufficient, in Theorem 3.34, to prove the absence of any type of deadlocks.

5. Finally, the last and strongest condition, from Theorem 3.49, requires stream causality $SC(\sigma)$ in each state σ of the program, providing all of the possible properties.

Remark 3.51. *Stream causality in each state of the program is the most restrictive condition, that also provides the most guarantees. While this condition might be construed as very restrictive to the expressiveness of CDDF programs, it is a natural condition in many cases of interest for our stream-computing extension to OpenMP. Indeed OpenMP is primarily designed to parallelize sequential programs, which necessarily have only forward dependences with respect to the program's control flow. Interestingly, this property guarantees stream causality in each state of the program as it means that the past of the program always produces the data required for its present. This condition is not always satisfied, but our experience shows that most algorithms are naturally expressed in this way rather than using more complicated, and counter-intuitive, backward dependence patterns.*

For instance, all the applications[9] *and micro-kernels that we have implemented using our extension, before developing this formal model, satisfy this condition. However, we believe that this may not necessarily be the case when the development does not start from a sequential implementation.*

[9] FMradio, 802.11a and FFT.

As a concluding remark, we consider that in order to ensure the correctness of the implementation of a programming construct formalized with the CDDF computational model, only spurious deadlock-freedom can be required. The other additional properties are beneficial for productivity and for enabling optimizations, but they are not necessary. The other deadlock types stem from algorithmic errors, serializability is always available in *correct*[10] programs, as are determinism properties and strict consistency.

3.9 Conclusion

In this chapter, we introduced a new computational model for streaming applications. This model heavily relies on the notion of control program which serves as a scheduler of data in streams. The deterministic schedule of data achieved in this manner allows to reduce the scheduling requirements, and the synchronization overhead, of streaming applications. We used this model to prove a set of important properties provided by programs implementing this model, like deadlock-freedom, determinism and serializability.

The main contributions of this chapter are as follows.

- A new, general model of computation for stream-computing programs.
- A set of conditions that guarantees, to varying degrees, deadlock-freedom of streaming programs and the proofs that such conditions are sufficient.
- The proof of functional and deadlock determinism of programs fitting in our model.

This shows that the streaming extension to OpenMP presented in Chapter 2 can be implemented efficiently, allowing to synchronize streams over prefixes without introducing new deadlocks, and has good debugging, and more generally productivity, properties: functional and deadlock determinism, independently of the execution platform, serializability and strict consistency. However, this model relies on a set of simplifying assumptions, notably unbounded memory, sequential control program and the absence of feedback from task activations to the control program. We need to relax these restrictions, in Chapter 4, in order to allow mapping our streaming extension onto the CDDF model, in Chapter 5.

[10] If we consider programs with functional deadlocks to be incorrect.

Chapter 4

Generalization of the CDDF Model

The CDDF model in its original form, presented in Chapter 3, makes some important simplifying assumptions that limit its applicability. The CDDF model assumes unbounded memory, sequential execution of the control program and the absence of communication between the control program and task activations. In this chapter, we relax these assumptions and introduce some practical notions that bring the model closer to its intended target, modelling our streaming extension to OpenMP. Our objective is to generalize the model while preserving the properties proved in the previous chapter or, when this is not possible, provide the conditions under which these properties still hold. In Chapter 5 we take this one step further and show how our programming model can map onto this computational model while integrating the semantics and properties of the underlying programming language.

Le modèle CDDF, tel qu'il est présenté au chapitre 3, fait un certain nombre d'hypothèses, qui simplifient le problème, mais en réduisent l'applicabilité. Dans ce chapitre, nous cherchons à prouver les conditions nécessaires pour garantir les mêmes propriétés, mais sans faire les hypothèses de mémoire non bornée, d'exécution séquentielle du programme de contrôle et d'absence de communication entre le programme de contrôle et les activations de tâches. Ces considérations d'ordre pratique sont nécessaires pour permettre d'utiliser le CDDF afin de modéliser les applications utilisant notre extension streaming au langage OpenMP. Au chapitre 5 nous montrerons comment ce modèle de calcul peut être appliqué à notre modèle de programmation tout en intégrant la sémantique et les propriétés du langage de programmation sous-jacent.

4.1 Introduction

This chapter presents an attempt at generalizing the CDDF model, by removing some of the simplifying assumptions that are the most constraining for any programming model that would implement this model of computation, and extending the proofs of sufficiency of either the same or stronger conditions to preserve the original properties.

We first introduce a communication scheme between task activations and the control program, modelling the firstprivate and lastprivate clauses in our streaming extension to

OpenMP. This type of communication introduces new synchronization constraints on the control program and hence new deadlocks. Secondly, we analyze the ordering constraints that must be enforced to preserve the program's semantics when evaluating activation points concurrently in the control program. This is an essential step towards achieving scalability. Finally we evaluate the impact of bounding the size of stream buffers on deadlock conditions and on determinism.

The remainder of this chapter is organized as follows. Section 4.2 relaxes the restriction on the communication between task activations and the control program, providing for the semantics of the OpenMP firstprivate and lastprivate sharing clauses. The new resulting deadlocks are characterized and the proofs of the deadlock-freedom properties in the CDDF model are extended to cover these new deadlocks. In Section 4.3, we analyze the ordering requirements for the evaluation of activation points that allow to preserve the CDDF determinism property while enabling concurrent execution of the control program. This section sets the stage for evaluating the static analysis requirements for parallelizing the control program, in Section 5.4. Finally, Section 4.4 evaluates the impact of bounding stream buffers, a necessary step towards enabling the execution of CDDF programs in bounded memory. The resulting resource deadlocks are characterized and we provide an algorithm for detecting and resolving resource deadlocks at runtime. The instantiation of this algorithm is discussed in Section 6.4.

4.2 Communication with the Control Program

The control program behaves as the driver of CDDF programs with respect to control flow. It is possible to write programs that do not communicate with the control program, simply initiating pipelines with a source task like in Kahn process networks, but this is a rather strong restriction that users may tend to circumvent using shared memory communication. Our goal is to eliminate the use of shared memory communication whenever possible, which allows to guarantee many important properties, like functional and deadlock determinism, beyond relying on programmer discipline, and possibly to compile the programs for distributed memory targets.

In Chapter 2, we used firstprivate and lastprivate clauses to enable communication between tasks and their enclosing context. We introduce here a model for this type of communication. While we could continue with our very general approach in modelling communication, in this particular case we do not allow as much freedom as for stream communication between tasks. This is necessary because of the strong constraints that this type of communication introduces in our model and is justified by the semantics of the firstprivate and lastprivate clauses, which are similar to that of synchronous data-flow:

- Production and consumption rates are always unitary, so production and consumption rates always match.
- The horizons are always unitary as well, no peek operations are possible.
- A single producer and a single consumer are allowed per stream. This is in no way restrictive as there is no real merging of data occurring because of the previous

restrictions, so a stream can be split in multiple streams without impacting the semantics.

- Each communication channel can be implemented with a buffer of size one, larger sizes are used only to allow decoupling and therefore tolerate latencies.

The communication from the control program to a task activation can be properly modeled by a stream where the aforementioned restrictions apply, so $burst = horizon = 1$ for both producer and consumer and the amount of data produced[1] is always identical to the amount consumed.

The communication between a task activation and the control program is, by definition, synchronous: as the control program is the generator of the task activation, this activation cannot start executing until the control program passes the activation point that generates the task activation. In the case of lastprivate clauses, the semantics of this communication requires that the data produced by the task activation be read by the control program just after the activation point. For this reason, there can never be more than one data element in the streams used to communicate data from task activations to the control program. Communicating between a task activation and the control program introduces a serializing synchronization and hence this type of communication is modeled as a stream with a buffer of size one, which behaves as a semaphore synchronization between the task and the control program.

For simplicity, and to keep the same terminology, we call communication from the control program to a task activation firstprivate communication and the reverse lastprivate communication. This naming convention does not mean that we restrict the applicability of the CDDF model to OpenMP and our extension.

4.2.1 Adjustment to the CDDF Model for Firstprivate and Lastprivate Communication

As firstprivate and lastprivate communication is restricted to synchronous semantics, the addition of this type of communication does not require major changes to the existing model. We introduce the following definitions.

Definition 4.1 (Firstprivate communication)**.** All task activations in a CDDF program can receive data from the control program using firstprivate communications. The control program produces exactly one value, during the generation of the task activation, on a dedicated stream connecting exclusively the control program and the task activation's *task*, as defined in Definition 3.35. When it executes, the task activation consumes exactly one value on that stream.

We define the function $FP : \mathcal{A}_{/\sim} \to \mathcal{P}(\mathcal{S})$ such that $FP([a]_\sim)$ is the set of firstprivate streams consumed by the task $[a]_\sim$.

[1]The number of times data is produced multiplied by the burst.

Definition 4.2 (Lastprivate communication)**.** All task activations in a CDDF program can send data to the control program using lastprivate communications. When the task activation a executes, it produces exactly one value on a dedicated stream connecting exclusively the task $[a]_\sim$ to the control program. The control program consumes exactly one value on that stream after executing the activation point that created the task activation.

We define the function $LP : \mathcal{A}_{/\sim} \to \mathcal{P}(\mathcal{S})$ such that $LP([a]_\sim)$ is the set of lastprivate streams produced by the task $[a]_\sim$.

Note that lastprivate communication introduces an additional operation in the control program's trace. But as it is always associated with, and executed immediately after, an activation point, we do not add this operation to the trace. This operation is equivalent to the execution of an activation in the sense that it represents a subset of $\mathcal{X}$, a set of read accesses to the lastprivate streams. As our semantics require this read to come exclusively from the activation generated by this activation point, this also represents a synchronization satisfied by waiting for the activation to execute. The case of firstprivate communication is somewhat similar, as it adds a set of write accesses to the firstprivate streams, but there is no synchronization semantics required for the control program.

To account for this synchronization point in the model, we need to adjust the execution rules from Figure 3.1 in case a task activation is generated, that uses lastprivate communication. We replace the previous set of execution rules by those presented on Figure 4.1. Once a task activation is generated that uses lastprivate communication, the control program cannot make progress, in the sense that all rules increasing the trace $\mathcal{K}_e$ are disallowed, until that task activation is executed. This should impact rules (GEN), (BAR) and (TERM), but in practice this condition is subsumed by the existing rule conditions for (BAR) and (TERM), so only (GEN) needs to be modified.

$$\text{(GEN)}\ \frac{\pi := NEXT(\mathcal{K}_e) \qquad \pi \in \Pi \qquad \forall a \in \mathcal{A}_o : LP([a]_\sim) = \varnothing}{(\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o) \longrightarrow (\mathcal{K}_e.\pi, \mathcal{A}_e, \mathcal{A}_o \cup \{\xi(\mathcal{K}_e, \pi)\})}$$

$$\text{(BAR)}\ \frac{\mathcal{A}_o = \{\mathcal{C}(\mathcal{K}_e)\} \qquad NEXT(\mathcal{K}_e) = barrier}{(\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o) \longrightarrow (\mathcal{K}_e.barrier, \mathcal{A}_e, \mathcal{A}_o)}$$

$$\text{(TERM)}\ \frac{\mathcal{A}_o = \{\mathcal{C}(\mathcal{K}_e)\} \qquad NEXT(\mathcal{K}_e) = \top}{(\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o) \longrightarrow (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)}$$

$$\text{(EXEC)}\ \frac{\mathcal{A}_o = \{a\} \cup \mathcal{A}'_o \qquad \mathcal{A}_e \ltimes a}{(\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o) \longrightarrow (\mathcal{K}_e, \mathcal{A}_e \cup \{a\}, \mathcal{A}'_o)}$$

Figure 4.1: CDDF execution rules with lastprivate communication semantics.

The (GEN) rule is modified to include the additional synchronization resulting from lastprivate communication. We add the condition that no task activation producing on

lastprivate streams can still be outstanding, in the set $\mathcal{A}_o$, when the rule is executed. Only one such task activation can be present in $\mathcal{A}_o$ at a time and it has to be the latest task activation generated by the control program.

4.2.2 Impact on the Properties of the CDDF Model

The modifications pertaining to communications with the control program are limited to the (GEN) execution rule and the simplifying restrictions on firstprivate and lastprivate streams. We will not go through all the proofs of the properties of the model with these adjustments as many are obvious. Determinism, both functional and that of deadlocks, cannot be impacted by this addition, nor can serializability. The only significant impact is on deadlock properties. The particular case of lastprivate communication and its control program synchronization semantics make the existing proofs obsolete.

Let us first start by reassessing, in the presence of lastprivate communications, the notion of deadlock program state, which we originally presented in Lemma 3.14, on page 69. Until now, the sources of deadlocks were barriers, for insufficiency deadlocks, dependence cycles based on the data-flow dependence relation, δ (Definition 3.15), for functional deadlocks and dependence cycles on the stream prefix order relation, $<$ (Definition 3.11), for spurious deadlocks.

Lastprivate communication introduces a new type of deadlock, similar to the barrier-induced insufficiency deadlocks in that a lastprivate communication also blocks the control program. This means that some or all of the outstanding task activations must be able to execute without additional data that would be produced by newer task activations not yet generated. Informally, when the data produced by all task activations generated by the control program *before* it reaches an activation point that has lastprivate communications is insufficient to satisfy the dependences of the task activation being generated, we have a lastprivate insufficiency deadlock.

Before defining this new type of deadlocks, we characterize the overall possible deadlocks in the model based on the execution rules of Figure 4.1. This characterization clarifies the changes that are required.

Remark 4.3 (Notation)**.** *In order to distinguish between the original model properties, as defined in the CDDF model before generalization, and the new properties resulting from this generalization, we will subscript the property name. For example, in the case of a deadlock state we replace $D(\sigma)$ with $D_L(\sigma)$ to show that the model includes lastprivate communication semantics.*

Lemma 4.4 (Deadlock characterization with lastprivate communication semantics)**.** *In the presence of lastprivate communications, the state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$ of a CDDF program in a deadlock satisfies:*

$$D_L(\sigma) \quad \Leftrightarrow \quad D(\sigma) \ \vee \ \big((\exists a \in \mathcal{A}_o, LP([a]_\sim) \neq \varnothing) \ \wedge \ (\forall a \in \mathcal{A}_o, \neg(\mathcal{A}_e \ltimes a))\big)$$

In the same way, the state of a CDDF program in a functional deadlock satisfies:

$$FD_L(\sigma) \quad \Leftrightarrow \quad FD(\sigma) \ \vee \ \big((\exists a \in \mathcal{A}_o, LP([a]_\sim) \neq \varnothing) \ \wedge \ (\forall a \in \mathcal{A}_o, \neg(\mathcal{A}_e \rhd a))\big)$$

Proof. The proof is similar to that of the original Lemma 3.14, on page 69.

We derive, from the definition of the execution rules on Figure 4.1, the following equivalences in a given state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$:

$$\begin{aligned}
&\text{(GEN)} \quad \Leftrightarrow \quad \Big(\big(NEXT(\mathcal{K}_e) \in \Pi\big) \ \wedge \ \big(\forall a \in \mathcal{A}_o,\ LP([a]_\sim) = \varnothing\big)\Big)\\
&\text{(BAR)} \quad \Leftrightarrow \quad \Big(\big(\mathcal{A}_o = \{\mathcal{C}(\mathcal{K}_e)\}\big) \ \wedge \ \big(NEXT(\mathcal{K}_e) = \mathit{barrier}\big)\Big)\\
&\text{(TERM)} \quad \Leftrightarrow \quad \Big(\big(\mathcal{A}_o = \{\mathcal{C}(\mathcal{K}_e)\}\big) \ \wedge \ \big(NEXT(\mathcal{K}_e) = \top\big)\Big)\\
&\text{(EXEC)} \quad \Leftrightarrow \quad \Big(\exists a \in \mathcal{A}_o,\ \mathcal{A}_e \ltimes a\Big)
\end{aligned}$$

Definition 3.13 states that a deadlock occurs when all rules are impossible in a given state of the program. We deduce that:

$$\begin{aligned}
D_L(\sigma) \quad &\Leftrightarrow \quad \neg(\text{GEN}) \ \wedge \ \neg(\text{BAR}) \ \wedge \ \neg(\text{TERM}) \ \wedge \ \neg(\text{EXEC})\\
&\Leftrightarrow \quad \begin{cases}
\Big(\big(NEXT(\mathcal{K}_e) \notin \Pi\big) \ \vee \ \big(\exists a \in \mathcal{A}_o,\ LP([a]_\sim) \neq \varnothing\big)\Big)\\
\wedge \Big(\big(\mathcal{A}_o \neq \{\mathcal{C}(\mathcal{K}_e)\}\big) \ \vee \ \big(NEXT(\mathcal{K}_e) \neq \mathit{barrier}\big)\Big)\\
\wedge \Big(\big(\mathcal{A}_o \neq \{\mathcal{C}(\mathcal{K}_e)\}\big) \ \vee \ \big(NEXT(\mathcal{K}_e) \neq \top\big)\Big)\\
\wedge \Big(\forall a \in \mathcal{A}_o,\ \neg(\mathcal{A}_e \ltimes a)\Big)
\end{cases}
\end{aligned}$$

By definition, the codomain of $NEXT$ is $\Pi \cup \{\mathit{barrier}\} \cup \{\top\}$.

We trivially deduce that $\big(NEXT(\sigma) \neq \mathit{barrier} \wedge NEXT(\sigma) \neq \top\big) \ \Leftrightarrow \ NEXT(\sigma) \in \Pi$, which allows us to simplify the second and third propositions, yielding:

$$D_L(\sigma) \quad \Leftrightarrow \quad \begin{cases}
\Big(\big(NEXT(\mathcal{K}_e) \notin \Pi\big) \ \vee \ \big(\exists a \in \mathcal{A}_o,\ LP([a]_\sim) \neq \varnothing\big)\Big)\\
\wedge \Big(\big(\mathcal{A}_o \neq \{\mathcal{C}(\mathcal{K}_e)\}\big) \ \vee \ \big(NEXT(\mathcal{K}_e) \in \Pi\big)\Big)\\
\wedge \Big(\forall a \in \mathcal{A}_o,\ \neg(\mathcal{A}_e \ltimes a)\Big)
\end{cases}$$

Finally, we notice that $\big(\exists a \in \mathcal{A}_o,\ LP([a]_\sim) \neq \varnothing\big) \ \Rightarrow \ \big(\mathcal{A}_o \neq \{\mathcal{C}(\mathcal{K}_e)\}\big)$, because the guard activation is not considered as a lastprivate synchronized activation even though it does contain write accesses to all streams, which allows us to eliminate $\big(NEXT(\sigma) \in \Pi\big)$ in the second proposition:

$$D_L(\sigma) \quad \Leftrightarrow \quad \begin{cases}
\Big(\big(NEXT(\mathcal{K}_e) \notin \Pi\big) \ \vee \ \big(\exists a \in \mathcal{A}_o,\ LP([a]_\sim) \neq \varnothing\big)\Big)\\
\wedge \Big(\mathcal{A}_o \neq \{\mathcal{C}(\mathcal{K}_e)\}\Big)\\
\wedge \Big(\forall a \in \mathcal{A}_o,\ \neg(\mathcal{A}_e \ltimes a)\Big)
\end{cases}$$

We finally use the same reasoning as above to distribute and simplify this expression,

which yields the desired result:

$$D_L(\sigma) \quad \Leftrightarrow \quad \begin{cases} \Big(\big(NEXT(\mathcal{K}_e) \notin \Pi\big) \ \wedge \ \big(\mathcal{A}_o \neq \{\mathcal{C}(\mathcal{K}_e)\}\big) \ \wedge \ \big(\forall a \in \mathcal{A}_o, \ \neg(\mathcal{A}_e \ltimes a)\big) \Big) \\ \vee \ \Big(\big(\exists a \in \mathcal{A}_o, \ LP([a]_\sim) \neq \varnothing\big) \ \wedge \ \big(\forall a \in \mathcal{A}_o, \ \neg(\mathcal{A}_e \ltimes a)\big) \Big) \end{cases}$$

□

Lemma 4.4 allows us to isolate the deadlocks created by the lastprivate communication pattern from all previously analyzed deadlock types and conditions. Based on the result of this lemma, we can now define what we will call lastprivate deadlocks.

Definition 4.5 (Lastprivate Deadlock state)**.** A CDDF program is in a lastprivate deadlock in a state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$, $LD(\sigma)$, if and only if there is an outstanding task activation $a \in \mathcal{A}_o$ that has lastprivate communications $LP([a]_\sim) \neq \varnothing$ and no outstanding task activation can be executed by the (EXEC) rule. We write:

$$LD(\sigma) \qquad \triangleq \qquad (\exists a \in \mathcal{A}_o, LP([a]_\sim) \neq \varnothing) \ \wedge \ (\forall a \in \mathcal{A}_o, \neg(\mathcal{A}_e \ltimes a))$$

Note the difference between $D_L(\sigma)$ and $LD(\sigma)$: the former represents a deadlock state when the model includes lastprivate communications, irrespectively of the origin of the deadlock, while the latter means that the deadlock is precisely induced by the presence of lastprivate communications.

Remark 4.6. *We can rewrite Lemma 4.4 with the new notation introduced in Definition 4.5:*

$$\begin{aligned} D_L(\sigma) \qquad &\Leftrightarrow \qquad D(\sigma) \ \vee \ LD(\sigma) \\ \neg D_L(\sigma) \qquad &\Leftrightarrow \qquad \neg D(\sigma) \ \wedge \ \neg LD(\sigma) \end{aligned}$$

The important result in this expression is that we can keep the existing deadlock property proofs from the original model and simply extend them to lastprivate deadlocks. Note that there is no interference between the types of deadlocks because they use different conditions on the control program. While we could merge the deadlock conditions by removing the last distribution step in the proof of Lemma 4.4, this would result in greatly over-approximating the deadlock-freedom conditions.

From this, we can deduce that requiring stream causality for each control program trace prefix ending with an activation point that generates lastprivate communications is a sufficient condition for deadlock-freedom as it guarantees that all activations in $\mathcal{A}_o$ can execute, but it is not necessary. In a more general way, all existing conditions are sufficient if a barrier is added after each such activation point. We could, for example, require that the program be stream causal not only when reaching a barrier as in Theorem 3.34, but also when reaching an activation point with lastprivate communication semantics.

The semantics of lastprivate communication is that of a *local* barrier, that synchronizes only the set of outstanding task activations belonging to causality chains leading to the activation using the lastprivate streams. Instead of the barrier behaviour we had until

now where all task activations in $\mathcal{A}_o$ needed to be executed, we now need to define the subsets of $\mathcal{A}_o$ that are *needed* for the control program to make further progress.

Definition 4.7 (Sets of *needed* task activations)**.** For a CDDF program in a state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$, we define the sets $\mathcal{A}_n^<$ and $\mathcal{A}_n^\delta$ of task activations that *need* to execute if the control program has generated a task activation using lastprivate communication:

$$\mathcal{A}_n^<(\sigma) \;=\; \Big\{ a \in \mathcal{A}_o \mid \exists a' \in \mathcal{A}_o, LP([a']_\sim) \;\wedge\; a <^* a' \Big\}$$
$$\mathcal{A}_n^\delta(\sigma) \;=\; \Big\{ a \in \mathcal{A}_o \mid \exists a' \in \mathcal{A}_o, LP([a']_\sim) \;\wedge\; a \;\delta^*\; a' \Big\}$$

Note that the definition of these sets includes the lastprivate task activation itself. The sets are empty if no lastprivate task activation is outstanding.

The real issue is to characterize the difference between *real* causality chains, that are based on flow dependences in streams, leading to a task activation using lastprivate communications and the over-approximated chains enforced by the stream prefix order. We do not detail and characterize every type of deadlock in this case, but only distinguish between functional and spurious deadlock states. Once again we rely on the notion of weak deadlock state to simplify the proofs.

In any given state σ, the functional soundness of the program requires that the activations in $\mathcal{A}_n^\delta(\sigma)$ be able to execute, while the synchronization we enforce require the activations in $\mathcal{A}_n^<(\sigma)$ to be able to complete. We can trivially show that $\forall \sigma, \mathcal{A}_n^\delta(\sigma) \subset \mathcal{A}_n^<(\sigma)$, so the problematic cases occur when the task activations in $\mathcal{A}_n^<(\sigma) \setminus \mathcal{A}_n^\delta(\sigma)$ cannot execute.

We first define a *weak lastprivate deadlock state* that characterizes the new possible deadlock states as well as its counterpart in terms of flow dependences, the *weak lastprivate functional deadlock state*. Based on these definitions, we then discuss how new spurious deadlocks can be characterized in case they do not coincide with existing deadlock conditions. Finally, we analyze the conditions necessary to prove the deadlock-freedom properties provided by the CDDF model.

Proposition 4.8 (Weak deadlock states with lastprivate communications)**.** *A CDDF program is in a weak lastprivate deadlock (resp. functional deadlock) in a state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$, and we write $WLD(\sigma)$ (resp. $WLFD(\sigma)$), if and only if its state satisfies:*

$$WLD(\sigma) \quad \Leftrightarrow \quad \mathcal{A}_n^<(\sigma) \neq \varnothing \quad \wedge \quad \Big(\exists a \in \mathcal{A}_n^<(\sigma),\; \mathcal{C}(\mathcal{K}_e) < a \;\vee\; a <^+ a \Big)$$
$$WLFD(\sigma) \quad \Leftrightarrow \quad \mathcal{A}_n^\delta(\sigma) \neq \varnothing \quad \wedge \quad \Big(\exists a \in \mathcal{A}_n^\delta(\sigma),\; \mathcal{C}(\mathcal{K}_e) \;\delta\; a \;\vee\; a \;\delta^+\; a \Big)$$

Proof. The proof of this proposition is not detailed as it simply requires noting that both deadlock states are indeed reached once all outstanding task activations that can be executed complete, but the task activations in $\mathcal{A}_n^<(\sigma)$ (resp. $\mathcal{A}_n^\delta(\sigma)$) that depend on either the continuation activation or on task activations within a cycle will never be executable as some of their dependences cannot be satisfied.

The reverse is trivial as deadlock states, where none of the outstanding activations can execute once a lastprivate task activation is outstanding, will at least have the outstanding lastprivate task activation in their needed sets. □

In order to define the new spurious deadlocks, we once again use the same basic definition as deadlock states that are not functional deadlocks:

$$WLSD(\sigma) \quad \approx \quad WLD(\sigma) \ \wedge \ \neg WLFD(\sigma)$$

In order to simplify this definition and provide a more convenient characterization, we rely on the insight that both deadlocks in Proposition 4.8 are due to the lastprivate task activation being impossible to execute because of dependences transitively originating from one or more task activations that cannot be executed, irrespectively of the execution schedule. This means that the lastprivate task activation either depends on an unsatisfiable cycle of task activations or on the continuation activation.

If an unsatisfiable cycle is present in the set of outstanding task activations, the conditions for a deadlock are met even if no barrier has been reached. If such a cycle happens to occur within $\mathcal{A}_n^<$, then the deadlock will be concretized earlier, but even in the absence of lastprivate communications it would finally lead to a deadlock state when the control program either reaches a barrier or terminates. For this reason, we will simplify the cases of weak lastprivate spurious deadlock states by only considering the insufficiency cases, where the lastprivate task activation depends on the continuation activation.

Definition 4.9 (Weak Lastprivate Spurious Deadlock state)**.** A CDDF program is in a weak lastprivate spurious deadlock in a state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$, and we write $WLSD(\sigma)$, if and only if its state satisfies:

$$WLSD(\sigma) \quad \triangleq \quad \exists a \in \mathcal{A}_o, \quad \big(LP([a]_\sim) \neq \varnothing\big) \ \wedge \ \big(\mathcal{C}(\mathcal{K}_e) <^+_{\mathcal{A}_o} a\big) \ \wedge \ \neg\big(\mathcal{C}(\mathcal{K}_e) \ \delta^+_{\mathcal{A}_o} \ a\big)$$

This definition is partial, but all missing cases are covered by weak spurious (non-lastprivate) deadlock conditions. For this reason, if the underlying program is free of such deadlocks, which is the case for all of the conditions we have studied, the definition is valid. The remaining cases are not subsumed by the existing spurious deadlock cases, despite the similarity of having task activations depend on the continuation in the stream prefix order but not based on flow dependences. This is due to the fact that a barrier may not have been reached and, therefore, that the continuation activation could change before this happens.

Theorem 4.10. *A CDDF program is free of lastprivate spurious deadlocks if it is free of insufficiency deadlocks, is task causal and each stream in the program is either single-producer or single-consumer. In each state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$:*

$$\neg WID(\sigma) \ \wedge \ TC(\sigma) \ \wedge \ \Big(\forall s \in S(\sigma), \ |P(\sigma, s)| = 1 \ \vee \ |C(\sigma, s)| = 1\Big) \quad \Rightarrow \quad \neg WLSD(\sigma)$$

Proof. This theorem is an extension of Theorem 3.41. It includes the additional lastprivate deadlock conditions. We are only interested in the absence of weak lastprivate spurious deadlocks, so we use as hypothesis the absence of weak insufficiency deadlock conditions.

Let us assume, by way of contradiction, that a CDDF program is experiencing a last-private spurious deadlock in a state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$, and that the theorem's conditions are satisfied in this state. We first expand the expression based on the Definition 4.9:

$$\neg WID(\sigma) \ \wedge \ TC(\sigma) \ \wedge \ \Big(\forall s \in S(\sigma) : \ |P(\sigma, s)| = 1 \ \vee \ |C(\sigma, s)| = 1\Big)$$
$$\wedge \ \Big(\exists a \in \mathcal{A}_o, \quad \big(LP([a]_\sim) \neq \varnothing\big) \ \wedge \ \big(\mathcal{C}(\mathcal{K}_e) <^+_{\mathcal{A}_o} a\big) \ \wedge \ \neg\big(\mathcal{C}(\mathcal{K}_e) \ \delta^+_{\mathcal{A}_o} \ a\big)\Big)$$

As the program is task causal in this state, and it is not in a weak insufficiency deadlock state, Proposition 3.40 allows to deduce that it admits a task causal schedule. Let θ be such a schedule. We expand the incriminating stream prefix precedence chain:

$$\exists a \in \mathcal{A}_o, a \neq \mathcal{C}(\mathcal{K}_e), \exists n \in \mathbb{N}, n > 0, \exists (a_1, ..., a_n) \in \mathcal{A}_o^n, \quad \big(a_n = a\big) \wedge \big(\mathcal{C}(\mathcal{K}_e) < a_1 < ... < a_n\big)$$

A set of streams $(s_1, s_2, ..., s_n)$ connect these tasks, based on the definition of the stream prefix precedence relation $<$. We can reuse the disjunction of the two cases, of single-producer or single-consumer streams, for all of the streams $s_{k,k\in[1,n]}$, from the proof of Theorem 3.41, and similarly show that we always either get a contradiction because of a weak insufficiency deadlock condition or the following holds:

$$\forall k \in [1, n-1], \quad a_k < a_{k+1} \qquad \Rightarrow \qquad \theta(a_k) < \theta(a_{k+1})$$

As the continuation activation is always considered to be acausal and is not taken into account for the property of single-producer or single-consumer streams, we cannot rely on this reasoning to further extend the chains to include $\mathcal{C}(\mathcal{K}_e) < a_1 \ \Rightarrow \ \theta(\mathcal{C}(\mathcal{K}_e)) < \theta(a_1)$. However, we rely on the proof of Lemma 3.24, which states:

$$\forall a \in \mathcal{A}_e \cup \mathcal{A}_o, \qquad \mathcal{C}(\mathcal{K}_e) < a \quad \Leftrightarrow \quad \mathcal{C}(\mathcal{K}_e) \ \delta \ a$$

In our case, this yields:

$$\mathcal{C}(\mathcal{K}_e) < a_1 \quad \Rightarrow \quad \mathcal{C}(\mathcal{K}_e) \ \delta \ a_1 \quad \Rightarrow \quad \theta(\mathcal{C}(\mathcal{K}_e)) < \theta(a_1)$$

$$\mathcal{C}(\mathcal{K}_e) < a_1 < ... < a_n \quad \Rightarrow \quad \theta(\mathcal{C}(\mathcal{K}_e)) < \theta(a_1) < ... < \theta(a_n) \quad \Rightarrow \quad \theta(\mathcal{C}(\mathcal{K}_e)) < \theta(a)$$

As the second part of the original expression to prove yields:

$$\neg\big(\mathcal{C}(\mathcal{K}_e) \ \delta^+_{\mathcal{A}_o} \ a\big) \quad \Rightarrow \quad \neg\big(\theta(\mathcal{C}(\mathcal{K}_e)) < \theta(a)\big) \quad \Rightarrow \quad \theta(\mathcal{C}(\mathcal{K}_e)) \geqslant \theta(a)$$

We conclude the proof with the contradiction:

$$\theta(\mathcal{C}(\mathcal{K}_e)) < \theta(a) \quad \wedge \quad \theta(a) \leqslant \theta(\mathcal{C}(\mathcal{K}_e))$$

□

4.2.3 Summary of the Deadlock-Freedom Properties

We present, on Table 4.1, a brief summary of the properties and conditions analyzed for deadlock-freedom after extending the model with lastprivate semantics. We explain the conditions and briefly discuss each case as follows.

1. The first, and weakest, condition comes from Corollary 3.43. It requires that no stream belonging to a strongly connected component of the taskgraph be both multi-producer and multi-consumer, though different streams can be either one or the other. It also requires that the program be task causal in that state, which is stronger than the original condition. This condition is insufficient to avoid lastprivate spurious deadlocks as it only focusses on cycles.

2. The second condition is an adaptation of the conditions proven for Theorem 4.10 and Corollary 3.43. This condition extends the first condition to also require all the streams from which a lastprivate task can be reached in the taskgraph be also either single-producer or single-consumer. This condition is expressed, for brevity, through the informal iterated closure of the function $\mathcal{I}$ which is meant to gather the set of input streams of a task activation a, then iteratively the input streams of the producer tasks. This adjustment provides the constraints required to ensure freedom of weak lastprivate spurious deadlocks and can be easily proved by noting that the additional streams that are covered correspond exactly to the set of streams $(s_1, s_2, ..., s_n)$ used in the proof of Theorem 4.10.

3. The third condition corresponds to the original Theorem 3.41 and its extended version 4.10, requiring that no stream in the program be multi-producer and multi-consumer (MPMC) and that the program be task causal. Programs meeting these requirements can only experience insufficiency and functional lastprivate deadlocks.

4. The fourth condition, from Theorem 3.42, requires the taskgraph to be free of strongly connected components. While this condition was sufficient to avoid non-insufficiency deadlocks in the basic version of the CDDF model, it is insufficient to avoid lastprivate deadlocks, which are closer to insufficiency deadlocks than to cycle-induced deadlocks.

5. The fifth condition, from Theorem 3.34, requires stream causality of the program state whenever the control program reaches a barrier. As this gives no additional information on the program state when reaching a lastprivate activation point, this condition is trivially insufficient to prevent lastprivate spurious deadlocks.

6. The sixth condition adjusts the fifth by adding this missing information, therefore requiring stream causality also when the control program reaches a lastprivate activation point. As we discussed in Remark 4.6, this condition is trivially sufficient.

7. Finally, the last and strongest condition, required by Theorem 3.49, requires stream causality in each state of the program. This condition still provides all of the possible properties, even in the presence of lastprivate communications.

Condition on state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$	Deadlock Freedom properties					
	$\neg D(\sigma)$	$\neg ID(\sigma)$	$\neg FD(\sigma)$	$\neg SD(\sigma)$	$\neg LD(\sigma)$	$\neg LSD(\sigma)$
$TC(\sigma) \ \wedge \ \forall s \in SCC(H(\sigma)), \neg MPMC(s)$	no	no	yes	yes	no	no
$\forall a \in \mathcal{A}_o, LP([a]_\sim) \neq \varnothing$, $\forall s \in \mathcal{I}^+(a) \cup SCC(H(\sigma)) \ \neg MPMC(s)$ $TC(\sigma)$	no	no	yes	yes	no	yes
$TC(\sigma) \ \wedge \ \forall s, \neg MPMC(s)$	no	no	yes	yes	no	yes
$SCC(H(\sigma)) = \varnothing$	no	no	yes	yes	no	no
$SC(\sigma) \ \vee \ NEXT(\mathcal{K}_e) \in \Pi$	yes	yes	yes	yes	no	no
$SC(\sigma) \ \vee \ NEXT(\mathcal{K}_e) \in \Pi$ $\vee \ \forall a \in \mathcal{A}_o, LP([a]_\sim) = \varnothing$	yes	yes	yes	yes	yes	yes
$\forall \sigma, SC(\sigma)$	yes	yes	yes	yes	yes	yes

Table 4.1: Deadlock-freedom conditions for CDDF programs extended with lastprivate communication semantics.

4.3 Parallelizing the Control Program

All of the major distinctions and advantages of the CDDF model over other streaming computational models come from the semantics of the control program, and in particular from its sequential semantics that allow for both determinism and a global understanding of the schedule of data in streams. However, all good things come at a cost, and in our case the cost is all too obvious: the control program's execution constitutes a perfect bottleneck if it cannot be parallelized itself.

Parallelizing the control program is not easy if we want to preserve all the program properties exhibited this far. In this section, we first analyze the weak spots where the control program's sequential execution is a crucial condition and isolate the specific serializing constraints that enable the properties' proofs. In a second part, we show how parallelization can preserve the necessary semantical information of the original sequential control program, either through ad hoc parallelization or through restrictions on the parallelism that can be exploited in the control program. Finally, we discuss how concurrent control program traces can be merged to build a sequential control program trace.

4.3.1 Control Program Concurrency Constraints

We model the control program's execution with the two functions $NEXT$ and ξ that respectively provide the next operation to be executed and evaluate an activation point. This means that we completely hide the execution of anything but activation points and barriers, so the concurrency constraints that we derive in this section can only be necessary conditions for the concurrent evaluation of activation points. If the control program's underlying program is inherently sequential, nothing can be done to parallelize its execution. However, as we will see, the evaluation of activation points can, to some

extent, be parallelized.

We restrict our study to the parallelization of the evaluation of activation points, which is the crux of our model. The possible parallelization of the underlying program results in a new oracle function $NEXT_n : \mathcal{K}^n \to (\Pi \cup \{barrier\} \cup \{\top\})^n$, where n is the number of threads executing the control program. This new function returns a set of next operations to execute, one for each thread. This issue is specific to the programming model, in Chapter 2, and is therefore discussed in Chapter 5, where we map our streaming extension onto the CDDF model.

We recall the definition of the activation point evaluation function, from Definition 3.8, on page 62:

$$\xi(\mathcal{K}_e, \pi) = \left\{ (u, s, i) \in \mathcal{X} \quad | \quad \exists (u, s, b, h) \in \pi \ \wedge \ i \in \big[\alpha, \alpha + h\big[, \text{ where } \alpha = \sum_{\substack{\pi' \in \mathcal{K}_e \\ (u,s,b',h') \in \pi'}} b' \right\}$$

It is immediately apparent that the main issue to concurrently evaluate multiple activation points comes from the explicit state embodied by the first parameter of this function, the control program trace. This trace is used to decide the placement of data read or written by the new task activation inside its input and output streams. However, the ξ function does not use all the operations in the program trace to compute the stream access indexes, but only a subset of the activation points in the trace.

We define the function $Use : \mathcal{K} \times \Pi \to \mathcal{P}(\Pi)$ which evaluates the set of activation points in a program trace that will be used by the activation point evaluation function ξ in order to generate a new task activation from an activation point. From the definition of ξ, we can define this function as:

$$Use(\mathcal{K}_e, \pi) = \Bigg\{ \pi' \in \mathcal{K}_e \quad | \quad \exists (u, s) \in \{R, W\} \times S, \ \exists b, b' \in \mathbb{N}, \ \exists h, h' \in \mathbb{N}^* : \\ (u, s, b, h) \in \pi \quad \wedge \quad (u, s, b', h') \in \pi' \Bigg\}$$

Intuitively, we can see that in the case of single-producer and single-consumer streams, this means that only the past activation points of the same task are required to evaluate the new activation of that task as (u, s) would be unique to a single equivalence class. In the general case of multi-producer and multi-consumer streams, this does not hold as the deterministic interleaving of data requires the knowledge of past activation points of more than the task itself.

The counterpart of the Use function which evaluates the set of activation points written by the evaluation function ξ, is the identity on the activation point: $Def(\mathcal{K}_e, \pi) = \{\pi\}$. From this, we can deduce the ordering requirements for the execution of activation points as the preservation of the sequential evaluation order: for two activation points π_1 and π_2 such that the sequential execution of the control program evaluates π_1 before π_2, if $\pi_1 \in Use(\mathcal{K}_e, \pi_2)$, then π_2 must be executed after π_1 in the concurrent schedule. This is equivalent to requiring that both activation points need to be in the same partition of

activation points, therefore evaluated by the same thread, and that the original sequential order of evaluation be respected.

Note that the fundamental property we have used is the order relation on activation points evaluation induced by the precedence of the occurrence of activation points in the deterministic, sequential control program trace. We used, for stream causality and for task causality, binary relations that are sub-relations of this order relation. In order to preserve the same properties, we need to ensure that any interleaving of activation points evaluation from concurrent control program traces preserves the sub-relations that we have used in our definitions of the stream clocks and of the task order.

If we consider a concurrent evaluation of activation points, where we do not need to synchronize between the threads evaluating activation points, then a thread can evaluate an activation point π only if it has itself evaluated all of the activation points used by the evaluation of π. Considering that n threads are used for this, and that $\mathcal{K}_e = \bigcup_{1 \leqslant i \leqslant n} \mathcal{K}_i$ where $\mathcal{K}_i$ contains the set of activation points evaluated by thread i, the condition for allowing π to be evaluated by thread i is:

$$Use(\mathcal{K}_e, \pi) \subset \mathcal{K}_i$$

The partition of the evaluation of activation points between threads, or a synchronization scheme, must preserve these ordering constraints for the observed behaviour of the control program to remain the same, which is to say that the schedule of data in streams remains deterministic and consistent with sequential execution of the control program and the sequential evaluation of activation points in control program order.

4.3.2 Ad Hoc Parallelization of the Control Program

A natural way of parallelizing the control program's execution, irrespectively of any parallelization effort on the underlying program[2], is to off-load the evaluation of activation points to different threads, while enforcing, or ensuring the preservation of, the ordering constraints highlighted in the previous section. The order can be preserved by scheduling the evaluation of all dependent activation points on the same thread, thus partitioning the activation points in independent groups that can be evaluated concurrently.

A maximal partition can be dynamically built using the algorithm presented on Figure 4.2. This algorithm increases the number of partitions as soon as an independent activation point needs to be evaluated. If the activation point depends on a single existing partition, it adds the new activation point to that partition. Finally, if the new activation point depends on multiple partitions, it needs to merge all such partitions into a single one so that the new activation point can be evaluated within a partition that satisfies all of its dependences. This final step reduces, when necessary, the number of partitions.

In this case, the control program is not parallelized itself, so the control program trace is kept sequential. As the partitions of activation points we build are all independent, there is no harm from concurrently evaluating activation points from different partitions.

This technique does not entirely solve the control program bottleneck issue, but does

[2]This type of parallelization must still provide some order information on the generation of activation points, but this is a programming model issue.

For a CDDF program in state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$, where $NEXT(\mathcal{K}_e) = \pi$, the following algorithm allows to decide on which thread π can be evaluated. This algorithm maximally partitions the evaluation of activation points on concurrent threads. The algorithm starts with $n = 0$.

- If $\forall i \in [1, n],\ Use(\mathcal{K}_e, \pi) \cap \mathcal{K}_i = \varnothing$ then evaluate π on a new thread $n + 1$ and update the number of threads $n := n + 1$.
- If $\exists i \in [1, n],\ Use(\mathcal{K}_e, \pi) \subset \mathcal{K}_i$ then evaluate π on thread i.
- If $\exists i_1, ..., i_k \in [1, n], \forall j \in [1, k],\ Use(\mathcal{K}_e, \pi) \cap \mathcal{K}_{i_j} \neq \varnothing$ then synchronize the threads $i_1, ..., i_k$, waiting for the completion of any scheduled evaluation of activation points on these threads, and merge them into a single thread at index i_1. Evaluate π on i_1 and update the number of threads $n := n - k + 1$, as well as all other thread indexes.

  This ensures that the condition we previously identified is satisfied for this merged partition i_1:

$$Use(\mathcal{K}_e, \pi) \subset \mathcal{K}_{i_1}$$

Figure 4.2: Dynamic partition of activation points for concurrent evaluation.

contribute to reduce the load of the control program, which is likely to become the critical path once load-balancing techniques, as described in Section 6.3.3, are applied. Importantly, it shows that the additional computation involved in pre-computing the data schedule, which is one of the overheads our model incurs, does not by itself constitute a scalability limitation in most cases. The programs where one partition needs to evaluate all activation points would correspond to a program where each stream is produced or consumed by all task activations, or more generally to programs where the task graph is strongly connected. In both cases scalability cannot be achieved, even disregarding this overhead.

4.3.3 Parallel Synchronous Execution of the Control Program

One way to simplify the partitioning constraints for the evaluation of activation points is to rely on a synchronous execution model for the control program. When such an execution is possible, the implicit order induced by synchrony on the control program traces will strongly restrict the possible interleavings of the local traces when merging them. Indeed, for each pair of events, either they happen at the same time or there is a precedence relation between them.

This restriction is equivalent to enforcing an order for the evaluation of activation points reached by different executions of $NEXT_n$. In other words, in a given state $\sigma =$

$(\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$, all activation points in $NEXT_n(\mathcal{K}_1, ..., \mathcal{K}_n)$ must complete before a new set of operations can be obtained with $NEXT_n(\mathcal{K}'_1, ..., \mathcal{K}'_n)$.

In this case, the only restriction on evaluation of activation points $(\pi_1, ..., \pi_n) = NEXT_n(\mathcal{K}_1, ..., \mathcal{K}_n)$ is that these activation points will not generate task activations reading or writing to common streams:

$$\forall i, j \in [1, n], i \neq j, \forall (u, s) \in \{R, W\} \times \mathcal{S} :$$
$$\left(\forall (b, h) \in \mathbb{N}^2, (u, s, b, h) \notin \pi_i\right) \quad \vee \quad \left(\forall (b, h) \in \mathbb{N}^2,\ (u, s, b, h) \notin \pi_j\right)$$

As the past of each thread is *known* to all threads in a synchronous execution, which ensures that the execution of all preceding activation points is complete on all threads before allowing to start evaluating new ones, we need only ensure that an activation point does not depend on activation points reached in the same clock tick.

4.4 Execution with Bounded Stream Buffers

Until now, we have relied on the strong simplifying assumption that CDDF programs executed in an infinite memory space, therefore allowing each stream to dispose of an infinite amount of memory. This has enabled our model to rely on the single dynamic assignment property for stream accesses, which can of course not be achieved on real architectures. The common approach for executing streaming applications in a bounded memory space is to implement streams as circular buffers instead of infinite arrays, while using a modulo indexing scheme that allows the program to still perceive single dynamic assignment. However, with a circular buffer implementation, the data dependence relations also need to take into account anti and output dependences. This greatly complicates the synchronization constraints, and the resulting deadlock conditions, to the extent that we will no longer be able to provide simple and non-restrictive program conditions for deadlock-freedom. We will call *resource deadlocks* the new class of deadlocks thus introduced.

There are two ways to handle this issue: (1) constrain the CDDF program to simpler classes of programs, like synchronous or cyclo-static dataflow programs, where it is possible to prove the boundedness or divergence of the amount of memory required by the execution and even to compute the bound; or (2) rely on a resource deadlock detection scheme and a dynamic resizing of stream buffers to resolve the deadlock.

The first solution provides the best performance results, but is less general as it not only imposes very stringent conditions on programs, but it may also fail in deciding on the boundedness or divergence of the amount of memory required for the program's execution. As the problem of finding an upper bound on the amount of memory necessary to buffer data in communication channels has been very extensively covered, we do not offer here any improvement over the existing work, neither for computing this bound nor for deciding on the bound's existence. For example, Parks provides an analysis and execution scheme for general Kahn process networks [52], first deciding whether the size of a program is bounded or diverges, then running with an approximate bound and resizing the buffers

when necessary. However, only a small set of CDDF programs fit in the KPN model, and only such programs would benefit from this result. Though this is not a priority, at a later stage of the development of our implementation, we may try to apply such techniques and statically ascertain that a program satisfies the necessary conditions and rely on its result. In our case, this approach is very difficult to use in practice as our model is entirely dynamic, so not even a runtime analysis of the taskgraph can conclude for the remainder of the execution, let alone a static analysis.

The second solution allows all programs to run with some arbitrary initial buffer sizes, then detects any resource deadlocks and dynamically re-adjusts the size of the buffer of the streams responsible for the deadlock. This allows to resolve deadlocks as long as the program does not run out of memory. If all resource deadlocks can be detected and resolved in such a manner, preferably without incurring a substantial runtime overhead, this solution becomes significantly more interesting than the first one, even in cases where the first approach yielded a theoretical bound. Indeed, this bound can be much higher than what an execution really requires and can exceed the system's memory size.

Note that this second approach, if tuned to increase conservatively the size of buffers only when it is the only alternative to resource deadlock, effectively computes a precise bound at runtime while implicitly also finding one of the schedules that minimize the size of stream buffers required. Of course, this choice is not efficient, as it also maximizes the overhead incurred. The buffer resizing policy should, in practice, rely on a geometrical progression. As we discuss below, this breaks determinism for resource deadlocks, but still preserves determinism for the other forms of deadlocks as long as there is sufficient memory to concretize the deadlock state.

In the remainder of this section, we take a closer look at the changes that are required in the CDDF model in order to account for the new way of modelling streams. We characterize resource deadlocks and we provide a resource deadlock detection scheme. The details of the implementation of this algorithm, as well as the buffer resizing scheme, are discussed in Chapter 6. Finally, we discuss the impact on determinism and in particular on deadlock determinism.

4.4.1 Characterization of Resource Deadlocks

In the CDDF model, the only ordering constraints enforced on the execution of task activations are those resulting from the stream prefix order, where the read operations block the execution of a task activation if data is missing in the prefix of the read access. The characterization of spurious deadlocks relies on the comparison of deadlock occurrences between this synchronization scheme and the minimal, necessary constraint of enforcing flow dependences. In this generalization step, we broaden the data dependences taken into account in the synchronization scheme to include the anti and output dependences introduced by the reuse of memory locations in buffers. As we still only use flow dependences as a basis for defining true synchronization requirements, all resource deadlocks will naturally fall in the spurious deadlock class[3].

[3]Except in case the user is responsible to provide a size for stream buffers. This is an option we keep as it allows to eliminate the overhead of running the resource deadlock detection algorithm.

Each stream is initially attributed an arbitrary amount of memory[4] which varies during program execution. We define the function $\mathcal{B} : \Sigma \times \mathcal{S} \to \mathbb{N}$ which returns the size of a stream's buffer in a given program state. In addition to the stream prefix order, our new synchronization scheme also enforces the anti-dependences from the reuse of memory space, which is commonly called the back-pressure, waiting for data to be consumed in a stream before new data can be produced.

Definition 4.11 (Stream buffer reuse order)**.** In a CDDF program state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$, we define a binary relation $\prec_\sigma \in \mathcal{P}((\mathcal{A}_e \cup \mathcal{A}_o)^2)$ on task activations as::

$$a \prec_\sigma a' \quad \triangleq \quad \exists s \in \mathcal{S}, \exists i, j \in \mathbb{N},\ i - j \geqslant \mathcal{B}(\sigma, s)\ \wedge\ (R, s, j) \in a\ \wedge\ (W, s, i) \in a'$$

A task activation a' that needs to write to index i, in a stream s, needs to wait until all task activations reading any index j that is less or equal to $i - \mathcal{B}(\sigma, s)$ have completed. This ensures that, in a stream implemented as a circular buffer of size $\mathcal{B}(\sigma, s)$, all read operations to the memory location to be written have been executed before overwriting. Note that the read operations in streams are prefix-closed, due to the construction of such operations in ξ, which allows to overlook output dependences which are all covered by flow and anti dependences in this case.

Using this additional constraint, we define the new ordering requirement for executing a task activation $\rtimes \in \mathcal{P}(\mathcal{A}) \times \mathcal{A}$ which models the fact that all dependences of a task activation in terms of the buffer reuse constraint are met by the execution of a set of task activations:

$$\forall a \in \mathcal{A}_o, \mathcal{A}_e \rtimes a \quad \triangleq \quad \forall (W, s, i) \in a, \forall j \leq i - \mathcal{B}(\sigma, s), \nexists a' \in \mathcal{A}_o,\ (R, s, j) \in a'$$

And finally we define the relation $\bowtie \in \mathcal{P}(\mathcal{A}) \times \mathcal{A}$ merging both the stream prefix order and the buffer reuse constraint:

$$\forall a \in \mathcal{A}_o, \mathcal{A}_e \bowtie a \quad \triangleq \quad \mathcal{A}_e \ltimes a\ \wedge\ \mathcal{A}_e \rtimes a$$

Once again, the role of the continuation activation is to ensure that all read accesses that could still be scheduled by the control program are represented within $\mathcal{A}_o$ as they could occur in the continuation of the program. If any such read operation is lagging too far behind, then write operations that would reuse the memory location are to be delayed until we know that there can be no further use of the value stored at that particular stream index, which makes the value dead and preserves the perceived single dynamic assignment property on stream accesses while allowing to reuse the memory.

The new deadlock conditions can be expressed in terms of the absence of progress possible in a state σ when replacing the task activation execution rule (EXEC) with:

$$(\text{EXEC}_{\bowtie})\ \frac{\mathcal{A}_o = \{a\} \cup \mathcal{A}'_o \qquad \mathcal{A}_e \bowtie a}{(\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o) \longrightarrow (\mathcal{K}_e, \mathcal{A}_e \cup \{a\}, \mathcal{A}'_o)}$$

[4]In the case of our implementation of the OpenMP streaming extension, this amount is computed during initialization to ensure that all stream buffers fit in cache.

Which allows to define resource deadlocks as deadlocks where no task activation can execute with ($\text{EXEC}_{\bowtie}$) while there still are task activations that can execute with (EXEC).

Definition 4.12 (Resource deadlock)**.** A CDDF program is in a resource deadlock in state σ, and we note $RD(\sigma)$, if no task activation is ready to execute based on the $\bowtie$ relation in that state, but some activations have all their dependences satisfied:

$$RD(\sigma) \triangleq \big(NEXT(\mathcal{K}_e) \notin \Pi\big) \wedge \big(\mathcal{A}_o \neq \{\mathcal{C}(\mathcal{K}_e)\}\big) \wedge \big(\forall a \in \mathcal{A}_o,\, \neg(\mathcal{A}_e \bowtie a)\big) \wedge \big(\exists a \in \mathcal{A}_o,\, \mathcal{A}_e \ltimes a\big)$$

This definition once again allows using the continuation activation to ensure that any deadlock condition is characterized by a cycle, like in Lemma 3.25, but this time on the stream prefix order and the buffer reuse constraints.

4.4.2 Resource Deadlock Detection and Resolution

The detection of resource deadlocks is easy to express in terms of global conditions, as presented in Definition 4.12, but this is not a satisfactory runtime solution because it requires reaching consensus and waiting for quiescence, both equally expensive and impractical. It also requires to wait for the control program to run until it reaches a barrier, or its termination, before any deadlock is concretized, which also does not fit the notion of a bounded memory space as the size of the set of outstanding activations, $\mathcal{A}_o$, may diverge.

Therefore, our objective is to provide a detection scheme that does not require global knowledge of the program state and detects and solves deadlock conditions prior to deadlock concretization. This means that a deadlock condition like a dependence cycle should be identified, and possibly broken if resizing the buffers is enough, even when the control program or another part of the program are still able to make progress. Finally, we also need to ensure that this detection scheme does not introduce undue overhead and pressure on the communication subsystem.

The key insight for our deadlock detection scheme is that any deadlock condition stems from the presence of a cycle of dependences between task activations, this time including both stream prefix order $<$ (Definition 3.11) and stream buffer reuse order $\prec_\sigma$ (Definition 4.11). This can easily be proved in a way very similar to Lemma 3.25. As we do not want to rely on global information, and in particular we want to avoid impacting the control program, we adopt a runtime state exploration pattern, where some condition triggers the exploration of the dynamic taskgraph. Since we are not interested in non-resource deadlocks, the triggering condition is the existence of an activation a that is impossible to execute because of memory reuse constraints in the original state σ, while all of its true dependences are satisfied in their over-approximated form:

$$\mathcal{A}_e \ltimes a \quad \wedge \quad \neg(\mathcal{A}_e \rtimes a)$$

In this case, we use the time that the task activation would be stalled waiting for space to become available in one of its output buffers to explore the taskgraph in search of a possible cycle. The search runs along all unsatisfied dependences, either the stream prefix

order or the memory reuse constraint. If such a cycle is found, all the stream buffers that are full and blocking the execution along this cycle are resized. The search results are scrapped if the state of any task on the dependences path from the original task activation changes.

We present, in this section, the high-level view of the *D*eadlock *D*etection and *R*esolution (DDR) algorithm, which is instantiated in Chapter 6. Many of the models we use here, like the continuation activation or the tasks viewed as equivalence classes on task activations, are instantiated through specific runtime information and data structures.

We consider that the overall system memory is able to fit the computation, which should fall under the programmer's responsibility. This assumption means that each "correct" CDDF program must admit a schedule that can execute within the bounds of the available memory.

In order to model bounded sets of outstanding activations, we extend the definition of the bound of stream buffers, $\mathcal{B}(\sigma, \blacksquare)$, to also provide a bound of the number of outstanding task activations allowed in a given state, $\mathcal{B}(\sigma, \mathcal{A}_o)$. We use the same notation despite the typing discrepancy because, in our case, the implementation of this set relies on streams.

The objective of the algorithm, presented on Figure 4.4, is to detect and resolve resource deadlocks while ensuring that:

1. The size of stream buffers is only increased when necessary. This is important for the efficient implementation of streams, as we try to always keep stream buffers in cache, which allows stream communication to be achieved through cache coherence traffic, on targets that support it with no external memory access.

2. The size of the outstanding activation set is not allowed to diverge needlessly. This constraint is less important and we choose to resolve resource deadlocks through increased outstanding activation sets rather than stream buffer resizing when possible.

3. The deadlock determinism property is preserved for all but resource deadlocks. This is the strongest and most difficult constraint for a concurrent algorithm as it requires to ensure that any non-resource deadlock will lead to the same final deadlock state, irrespectively of the execution schedule and the way resource deadlocks are resolved. This is only possible when the concretized deadlock state fits in memory.

The DDR algorithm is executed as soon as a trigger condition is met during the normal execution of the stream synchronization algorithm, which enforces the stream prefix order and the memory reuse order. The proof of correctness of the DDR algorithm relies on showing that, under certain assumptions, it converges in a finite number of steps towards either executing the task activation that triggered the DDR algorithm or finding a non-resource deadlock condition.

Proof of the resource deadlock detection and resolution algorithm on Figure 4.4. We make the following assumptions in order to prove the convergence of this algorithm for a task activation a.

The program is in a current state σ, where one thread attempts to execute task activation $a \in \mathcal{A}_o$. The following steps describe the workings of the deadlock detection algorithm.

- If $\mathcal{A}_e \ltimes a \ \wedge \ \mathcal{A}_e \rtimes a$: execute the task activation. This test is part of the synchronization scheme.
- Otherwise, if $\neg(\mathcal{A}_e \ltimes a)$: the task is not executable because of missing input data. Yield to the scheduler.
- Deadlock detection and resolution trigger condition: $\mathcal{A}_e \ltimes a \wedge \neg(\mathcal{A}_e \rtimes a)$ which means that all input data is available but the task cannot execute due to lack of space in output stream buffers. This condition will be re-tested at every step to ensure it still holds. If it is invalidated the algorithm stops as the task activation becomes executable.

  1. If a non-resource deadlock condition has already been identified, so if the step *Handle detected deadlock* has been executed once, we deduce from the trigger condition that:

     $$\exists a' \in \mathcal{A}_o,\ a' \prec_\sigma a \ \Rightarrow\ \exists s \in \mathcal{S}, \exists i, j \in \mathbb{N},\ i - j \geqslant \mathcal{B}(\sigma, s) \ \wedge\ (R, s, j) \in a' \ \wedge\ (W, s, i) \in a$$

     We increase the size of the buffer of stream s to fit the data produced by a, setting $\mathcal{B}(\sigma, s) := i - j + 1$.

  2. Otherwise, explore the outstanding activations depth-first along all unsatisfied dependence chains starting at task activation $a_0 = a$, building a chain of activations $(a_0, ..., a_k)$, where each consecutive pair is either in the $<$ or the $\prec_\sigma$ relation, until one of the following:

     2.1. One of the task activations a_i in the chain is executed by another thread. Restart the search at a_{i-1}. If a_0 is executed, abort the search.

     2.2. Reaching an executable task activation a_k. Execute this activation (or wait until another thread does). Restart the search for unsatisfied dependences at a_{k-1}.

     2.3. Reaching the continuation activation $a_k = \mathcal{C}(\mathcal{K}_e)$.

        2.3.1. If the control program can make progress, which changes $\mathcal{K}_e$ and therefore $\mathcal{C}(\mathcal{K}_e)$, allow the control program to execute and retry from a_{k-1}. We consider that at least one new activation is generated.

        2.3.2. If the control program cannot make progress:

           2.3.2.1. If $NEXT(\mathcal{K}_e) \notin \Pi$:

              2.3.2.1.1. $\mathcal{C}(\mathcal{K}_e) < a_{k-1}$ means an insufficiency deadlock. Execute the *Handle detected deadlock* step.

              2.3.2.1.2. $\mathcal{C}(\mathcal{K}_e) \prec_\sigma a_{k-1}$ is a hard anti-dependence. Resize the buffer of the stream along this dependence to fit the data from a_{k-1}. Retry from a_{k-1}.

           2.3.2.2. If $\exists a' \in \mathcal{A}_o, LP([a']_\sim) \neq \varnothing$:

              2.3.2.2.1. If $\exists a_i \in (a_0, ..., a_k), a_i = a'$, then either: (1) $\exists j \in [i, k-1], a_{j+1} \prec_\sigma a_j$ where we choose the highest such j, to resolve the issue closest to the continuation activation, and we increase the size of the corresponding stream buffer, then retry from a_j; or (2) $\forall j \in [i, k-1], a_{j+1} < a_j$ which is a lastprivate insufficiency deadlock: execute step *Handle detected deadlock*.

              2.3.2.2.2. Otherwise, restart the exploration step from a', to eagerly try and detect potential lastprivate deadlock conditions, and then retry executing a.

     2.4. Reaching a task activation a_k that is already present in $(a_0, ..., a_{k-1})$ at the index i, which means that there is a cycle. In this case, the cycle needs to be broken.

        2.4.1. If $\nexists a_j \in (a_i, ..., a_{k-1}), a_j \prec_\sigma a_{j+1}$ then this means that we have a cycle on the stream prefix order alone, which is a non-resource deadlock condition.

        2.4.2. Otherwise, increase the size of the buffers of all streams defining a $\prec_\sigma$ relation between two activations in the cycle.

Figure 4.3: Outline of the resource deadlock detection and resolution (DDR) algorithm (continued on next page).

- *Handle detected deadlock:* once a non-resource deadlock condition is detected, which is not necessarily a deadlock yet, we set a flag that changes the policy of stream buffer resizing to avoid converging to a final deadlock state that could occur earlier than in the infinite memory model.

  Additionally, at this point, the algorithm has a precise knowledge of the deadlock condition, so a warning or other form of feedback can be provided on the source of the problem, like an insufficiency of data produced on a given stream or an unsatisfiable cycle of dependences, causality violation, etc.

- Independently, the control program can increase the limit for the set of outstanding task activations if the program is quiescent $\forall a' \in \mathcal{A}_o,\ \neg(\mathcal{A}_e \bowtie a')$, and the set of outstanding task activations has reached its limit $|\mathcal{A}_o| = \mathcal{B}(\sigma, \mathcal{A}_o)$.

Figure 4.4: Outline of the resource deadlock detection and resolution (DDR) algorithm.

- We assume that that the number of task activations generated by the control program between the creation of a task activation containing a stream access (u, s, i) and the creation of any other task activation containing (u', s, i) is finite. This means that this property does not hold for programs that are non-terminating and that produce (resp. consume) data on some stream index, but never consume (resp. produce) that particular stream index.

- We further assume that a task activation cannot depend, transitively, on an infinite number of task activations. An infinite dependence chain leading to a task activation means that it will never be executable, so if there is no deadlock condition, our algorithm cannot converge. This case only occurs if the program generates an infinite number of streams, otherwise the number of stream accesses on which an activation can transitively depend is finite which also ensures that the number of task activations to which these accesses belong is finite.

We prove the finite convergence of this algorithm by showing that it takes our algorithm a finite number of steps to ensure that one of the task activations on which the current task activation a depends is executed or to conclude to a non-resource deadlock[5]. As we assume that the set of task activations on which it can transitively depend is finite and as the number of such task activations is strictly decreasing, the algorithm converges in a finite number of steps.

Once the algorithm is triggered, it first checks whether a deadlock condition has already been detected. If such is the case, the algorithm behaves in a relaxed mode, executing step (1), where it provides any stream with as much space as it requests, even if the program's execution could continue without increasing the size of stream buffers. This avoids converging to a local deadlock state before reaching the maximum deadlock state that could be reached in the case of unbounded memory.

If no deadlock condition has been found yet, the program will execute (2), where the algorithm explores the dependence chains leading to task activation a. As we suppose that there can only be a finite number of task activations on which a can transitively depend, each such dependence chain is finite. We also know that, as the set of outstanding activations $\mathcal{A}_o$ is finite, the exploration either finds the end of a chain or a cycle of dependences. We argue that each substep of (2) is finite and no infinite sequence of substeps can exist before a task activation, on which a depends, is executed.

(2.1) and (2.2) are the execution steps, which inherently decrease the number of outstanding task activations on which a depends.

(2.3.1) does not decrease the number of outstanding task activations on which a depends, but instead it serves to allow the control program to generate the missing task activations. In this case, we have $\mathcal{C}(\mathcal{K}_e) < a_{k-1} \vee \mathcal{C}(\mathcal{K}_e) \prec_\sigma a_{k-1}$, which can be rewritten,

[5]There is an additional possibility, if the buffer resizing scheme fails to allocate more memory, the algorithm terminates (gives up) but can still provide information on the sizes of streams, which can be used by developers to identify the source of the error, considering that the program's execution is supposed to fit in memory.

by definition, as:

$$\exists (s,i) \in \mathcal{S} \times \mathbb{N}, \quad \Big((R,s,i) \in a \ \wedge \ \big(\exists j \leqslant i, (W,s,j) \in \mathcal{C}(\mathcal{K}_e)\big)\Big) \\ \vee \ \Big((W,s,i) \in a \ \wedge \ \big(\exists j \leqslant i - \mathcal{B}(\sigma,s), (R,s,j) \in \mathcal{C}(\mathcal{K}_e)\big)\Big)$$

As the allocation of stream indices to task activations is prefix-closed, the generation of any (u,s,j), where $j \leqslant i$, occurs before that of (u,s,i). Finally, as we assume that only a finite number of task activations can be generated between any two task activations containing respectively (u,s,i) and (u',s,i), we deduce that only a finite number of iterations of step (2.3.1) are required before all dependences from $\mathcal{C}(\mathcal{K}_e)$ to a_{k-1} are materialized in dependences from generated task activations or the control program becomes unable to make progress. In the former case, the length of the dependence chains increase, which cannot occur infinitely as we argued above, while in the latter case the algorithm cannot execute this step any longer.

(2.3.2.1) corresponds to a case where the control program has reached either termination or a barrier. There are two subcases. The first one immediately concludes to a deadlock condition and the second one eliminates a dependence $\mathcal{C}(\mathcal{K}_e) \prec_\sigma a_{k-1}$ by resizing the buffer of the stream carrying that dependence. As we require task activations to be finite, with a finite number of streams, there can only be a finite number of such dependences, which ensures that a finite number of executions of this step will either allow a_{k-1} to execute, or make this step impossible to further execute.

In (2.3.2.2), the control program has generated a lastprivate task activation a' and can therefore not make progress until a' completes. The two subcases consider whether a' is in a dependence chain leading to a or not, artificially switching the focus of the algorithm to a' if there is no dependence chain between a' and a, thus implicitly making a depend on a'. As the dependence chains are finite in the case of a' as well, the same reasoning as for a will apply. This step takes a finite number of iterations to either conclude to a deadlock condition or to execute a task activation on which a depends.

Finally, (2.4) deals with the case where the continuation activation is not reached during the exploration, and it immediately concludes to either a non-resource deadlock condition or it resolves at least one dependence, leading in a finite number of steps to executing one task activation on which a depends. □

4.4.3 Deadlock Determinism

The only case where deadlock determinism could be impacted is if the memory available does not hold the *concretized* deadlock state, which requires that the control program be blocked by either a barrier, a lastprivate activation point or the program termination. This can be resolved by introducing an artificial maximal bound on the set of outstanding task activations, which would therefore serve as a concretization point, but this can introduce a new form of spurious resource deadlocks.

4.5 Conclusion

In this chapter, we generalized the CDDF model by removing some of the original simplifying assumptions. We started by allowing communication to occur between the control program and the streaming tasks, which led to the appearance of the new class of lastprivate spurious deadlocks, we proved that these deadlocks can be avoided under conditions summarized on Table 4.1. In a second step, we analyzed the ordering constraints required to allow the parallelization of the control program, and provided a simple runtime algorithm for partitioning the evaluation of activation points among multiple threads of execution while preserving the semantics of the program. This is not entirely sufficient to ensure scalability, but it provides a mitigating solution, that does not rely on static analyses that may fail, and it also sets the groundwork for designing static analyses, in Chapter 5, to enable a more satisfactory form of control program parallelization presented in Section 5.4. Finally, we removed, in Section 4.4, the assumption that memory is unbounded. This introduces a last form of deadlocks, resource deadlocks. In order to avoid this type of deadlocks, for which we cannot prove any sufficient freedom condition for general CDDF programs, we introduced a deadlock detection and resolution algorithm.

Chapter 5

Control-Driven Data-Flow Semantics of Dependent Tasks in OpenMP

At the confluence between the OpenMP stream programming model, presented in Chapter 2, and the control-driven dataflow model of computation, presented in Chapters 3 and 4, this chapter presents the application of CDDF properties to OpenMP streaming programs. We first show how our stream-computing extension to OpenMP implements the CDDF model and discuss the discrepancies. In a second step, we provide a static analysis framework for streaming annotations to apply the results proven in the CDDF model to OpenMP streaming programs. Finally, we show that this static analysis framework also enables relying on more efficient stream communication synchronization code (in Chapter 6), compiler optimizations, including control program parallelization, and the generation of optimized activation point evaluation code (in Chapter 7).

A la confluence entre notre modèle de programmation streaming, présenté au chapitre 2, et le modèle de calcul Control-Driven Data-Flow, présenté dans les chapitres 3 et 4, ce chapitre étudie l'application des propriétés CDDF aux programmes écrits avec l'extension streaming d'OpenMP. Nous montrons tout d'abord que cette extension est compatible avec le modèle CDDF, à quelques divergences près, que nous analysons. Nous décrivons ensuite un ensemble d'analyses statiques des directives de compilation dédiées au streaming, qui permettent d'appliquer les résultats prouvés dans le modèle CDDF aux programmes streaming OpenMP. Nous montrons enfin, dans ce chapitre, que ces analyses statiques permettent aussi : de faire appel à un algorithme optimisé pour la synchronisation des communications, présenté au chapitre 6, d'appliquer des optimisations de compilation, avec en particulier la parallélisation du programme de contrôle, et de générer du code optimisé pour l'évaluation des points d'activation des tâches, au chapitre 7.

5.1 Introduction

The objective of the CDDF model is primarily to represent OpenMP streaming programs in order to reason about their properties. This is an important step, that ensures that such programs have a well-defined behaviour and allows to prove that desirable behaviours

can be guaranteed by satisfying some simpler, possibly static, conditions. However, this does not show that such conditions are satisfied by a given program. This would require developing new, ad hoc, static analysis techniques, and implementing them, which is of little interest if we can avoid it. A better solution is to find common ground between the conditions we wish to infer from the program source and the conditions required by existing program optimization techniques. We adopt this approach to re-use and adapt existing static analysis techniques commonly used in optimizing compilers, like dataflow analysis [38] and array dataflow analysis [9,22], as well as existing compiler intermediate representations, in particular the single static assignment form [20]. Indeed, we show that small adjustments to these techniques let us analyze properly OpenMP programs using our streaming features from Chapter 2 and to infer whether the sufficient conditions from Chapter 3 are satisfied or not.

We further rely on these static analysis tools in order to enable both our own code generation in GCC and runtime optimizations, which we discuss in Chapters 6 and 7, and some existing compiler optimizations which allow, for example, to parallelize the control program. Central to our optimization framework is the notion of regular tasks, which we define as tasks that present a closed-form relation between their stream accesses and their activation indexes. Such tasks can use simpler, faster synchronization mechanisms and require less control program side code generation. Detecting regular tasks is therefore essential if we are to efficiently exploit streaming programs and bridge the gap between our highly expressive programming model and the performance of simpler, more restrictive, stream programming languages.

Our objective in this chapter is to set the groundwork for the efficient implementation of the necessary runtime support and code generation for our OpenMP streaming extension. The first issue we have to deal with is that, while the CDDF model is built in the perspective of modeling OpenMP streaming programs, the abstractions required to simplify its representation and allow the proof of properties has led to some discrepancies. The representation of an OpenMP streaming program as a CDDF program results in the loss of some information, more specifically the work function of an OpenMP task is abstracted away in the CDDF model. In order to benefit from the properties proven on the CDDF model for OpenMP programs, we first need to reconcile the model and the language.

This chapter is organized as follows. Section 5.2 presents the relation between our stream programming language and the CDDF computational model. It presents the simple mapping between the model and the language and discusses how their discrepancies are reconciled. Section 5.3 explores the requirements, in terms of static analysis, of our code generation framework and shows how this can be achieved with existing analysis tools. Finally, Section 5.4 details the case of control program parallelization.

5.2 OpenMP Streaming as an Instance of the CDDF Model

In this section, we describe the way our stream-computing extension to OpenMP implements the CDDF model. The mapping between our programming language and

our computational model is rather obvious for the most part, but some adjustment is necessary to account for some specificities of the semantics of the OpenMP language and our streaming extension. These specificities work both ways, in some cases we need to ensure that the hypotheses made in the CDDF model are satisfied by the semantics of OpenMP tasks, while in other cases the programming model's semantics allows us to provide stronger properties.

We recall that the CDDF model is entirely structured around the notion of control program, in which we only focus on activation points and barrier operations. The CDDF control program naturally models the main OpenMP program, which serves as a driver for the rest of the computation. Its execution is modeled by the oracle function NEXT, and the activation point evaluation function ξ, see Definition 3.8 in Section 3.2.1. In the remainder of this section, we first show that activation points reached by the NEXT function properly model OpenMP streaming task directives reached by the control flow of the main program. We then proceed to compare OpenMP streaming tasks and their model, as well as the semantics of streams. Finally, we discuss some of the important impacts of OpenMP semantics on this instance of the CDDF model, in particular showing that the explicit parallelization of the control program is greatly simplified.

5.2.1 From OpenMP Task Directives to CDDF Activation Points

We defined activation points, in Definition 3.5 on page 62, as sets of descriptors of stream access operations. These descriptors are tuples specifying for a given stream, and a given direction (read or write), the amount of data to be accessed and discarded, which is the shift of position in the stream before the next activation point. The semantics of the streaming clauses, input and output, that we added to OpenMP tasks, is exactly the same. The clause itself encodes the direction of the operations, read for input and write for output, the stream identifier is explicit in the clause, and finally the stream access window provides the information on the *burst* and *horizon* of this descriptor. The equivalence is illustrated by the example on Figure 5.1.

The direct translation of streaming clauses into stream access descriptors, presented on Figure 5.1, also matches the restrictions we imposed both in the programming model and the computational model on the accepted values for bursts and horizons. This defines the conversion of task directives and their streaming clauses to CDDF activation points.

Figure 5.2 provides an example of the CDDF model of a simple OpenMP streaming program. In this example, a first task produces data on a stream x which is read by two tasks that interleave their read accesses to this stream. All tasks are executed twice because of the iteration space of the enclosing loop. The equivalent CDDF control program trace that will be generated contains six activation points and a final implicit barrier that comes from the semantics of the OpenMP parallel region.

The first observation we make on this toy example, and more generally on OpenMP programs, is that unlike the general view of activation points in the CDDF model, activation points can be extremely regular, in particular when burst rates and horizons are constant. This allows defining closed-form expressions for the indexes of stream access

```
// General case:
int stream, window[horizon];

input (stream >> window[burst])  ⟶ (R, stream, burst, horizon)
output (stream << window[burst]) ⟶ (W, stream, burst, horizon)

// Shorthand versions:
int x;

input (x)        ⟶     (R, x, 1, 1)
output (x)       ⟶     (W, x, 1, 1)

firstprivate (x)     ⟶     (R, x, 1, 1)
lastprivate (x)      ⟶     (W, x, 1, 1)
```

Figure 5.1: Streaming clauses as descriptors of stream accesses in activation points.

patterns, therefore strongly simplifying the parallelization constraints for the control program by avoiding the stateful evaluation of activation points, as discussed in Section 4.3. Furthermore, the construction of activation points from streaming clauses ensures that their evaluation will lead to finite sets of stream accesses, thus finite task activations, which is one of the assumptions we relied on for the proof of the resource deadlock detection algorithm, in Section 4.4.

A second observation is that this example highlights one of the main differences between the CDDF model and OpenMP task directives: in the CDDF model we have completely abstracted away the work function, which corresponds, in the case of OpenMP streaming tasks, to the block of code which is annotated with the task directive. This work function is a completely syntactical notion, and both consumer tasks could even have the same code as in the example, yet those tasks are indeed differentiated in OpenMP. The names of the functions called within tasks, `bar` in our example, is irrelevant with respect to OpenMP as it considers each task construct to define its own work function, irrespectively of possible similarities with other tasks. This is of little consequence for the semantics of activation points and task activations, but it changes the notion of task. The resulting activation points in the CDDF model of the program, (π_2, π_3) and (π_5, π_6), are indistinguishable and would be evaluated into task activations that would fall in the same equivalence class, so the same CDDF task.

5.2.2 OpenMP Tasks vs. CDDF Tasks

Let us first remark that the CDDF definition of task activations is simply a set of stream access operations. This correlates directly with OpenMP streaming tasks, where we disregard all non-streaming specific information, like non-streaming clauses and the associated memory operations or the work-functions themselves.

As mentioned above, the main difference between CDDF and OpenMP tasks is that OpenMP tasks are syntactically differentiated, so two tasks are always distinct, even

```
int x, X[2];

#pragma omp parallel
#pragma omp single
{
  for (int i = 0; i < 2; ++i)
  {
#pragma omp task firstprivate (i) output (x << X[2])
    {
      X[0..1] = foo (i);
    }

#pragma omp task input (x)
    bar (x);

#pragma omp task input (x)
    bar (x);
  }
} // implicit barrier at end of parallel region

// Leads to a control program generating the trace
// (π1.π2.π3.π4.π5.π6.barrier)
```

$$\text{Where} \quad \begin{cases} \pi_1 = \{(R, i, 1, 1), (W, x, 2, 2)\} \\ \pi_2 = \{(R, x, 1, 1)\} \\ \pi_3 = \{(R, x, 1, 1)\} \\ \pi_4 = \{(R, i, 1, 1), (W, x, 2, 2)\} \\ \pi_5 = \{(R, x, 1, 1)\} \\ \pi_6 = \{(R, x, 1, 1)\} \end{cases}$$

Figure 5.2: Example of an OpenMP streaming program and its CDDF model.

if they are identical in all respects, while CDDF tasks are equivalence classes on task activations which disregard the work-function associated with the streaming task. For this reason, CDDF tasks can overlap multiple OpenMP tasks as is the case, in the example on Figure 5.2, for the second and third tasks. Indeed, in this case, the activations of these two tasks would be in the same equivalence class and therefore belong to the same CDDF task.

We identify three different, partially overlapping, notions of tasks:

OpenMP task construct – static task represents a purely syntactical and hence static notion of task. In OpenMP, it also implicitly defines the work function, which is the syntactical body of the task construct. Unless otherwise specified, we refer to such constructs as OpenMP tasks or static tasks.

OpenMP task instance is a task activation. It represents the dynamic instantiation of

an OpenMP task construct, spawned when the control flow of an execution thread encounters a task construct.

CDDF task from Definition 3.35, only looks at the stream communication. It is defined as an equivalence class on input and output streams that disregards the work function. For this reason, it can overlap multiple OpenMP task constructs, as is the case in the example on Figure 5.2 for the two last OpenMP task constructs.

It is important to note that the reverse is also true, and a single OpenMP task construct can overlap multiple CDDF tasks, when an array of streams is used and the subscript in this array is not static. Let us consider the following example.

```
int A[2];

for (i = 0; i < 2; ++i)
#pragma omp task output (a >> A[i])
  a = ...;
```

In this case, the static task construct translates to two activation points, each communicating on a different stream. The resulting task activations belong to two distinct equivalence classes on the $\sim$ relation, from Definition 3.35, and therefore they belong to two distinct CDDF tasks.

Streaming task is the pragmatic association of a CDDF task with a work function, and therefore with an OpenMP task construct. In the example above, the single OpenMP task construct leads to two streaming tasks, one communicating on stream `A[0]` and the other on `A[1]`. In the example on Figure 5.2, the last two task constructs are seen as a single CDDF task, but despite having a *similar* body, they are syntactically distinct and therefore produce two streaming tasks.

Streaming tasks are the prevailing notion in the implementation-oriented Chapters 6 and 7.

This difference between the notions of tasks constitutes the only major weakness of the CDDF model to closely model OpenMP streaming. Indeed, we have based the notion of task causality on the CDDF notion of task, which means that multiple OpenMP tasks would need to be *causal* together, so that there would be no causality chain going backwards in the task order when considering the control program order for task activations across all OpenMP tasks that belong to the same CDDF task. In other words, the control program order should be a viable schedule not only for the execution of instances of the same task, but for groups of tasks. On the previous example, on Figure 5.2, task causality should require that at least one schedule be admissible, where the task activations of the same OpenMP task execute in order (e.g., π_2 before π_5 for the second task). However, once we convert to the CDDF model, the requirement spans across both the second and third task, therefore requiring that π_2 can be scheduled before π_3 itself scheduled before π_5 and then π_6 in at least one admissible schedule. This much stronger constraint should be avoided.

Our objective is to keep the weaker task causality condition on OpenMP tasks, where the condition only applies to individual task constructs. This proves to be quite easy,

despite the discrepancy in the notion of task causality between the CDDF model and OpenMP, as we have always used the notion of task causality in conjunction with a restriction for such tasks to only communicate on streams that are either single-producer or single-consumer. This means that all OpenMP tasks that are both producers and consumers of data are equal to the CDDF task. This comes from the definition of CDDF tasks as equivalence classes, and noticing that the task being either single-producer or single consumer on some stream will lead to the activations of this task belonging to an equivalence class of their own. The only issue remaining is for tasks, as is the case on Figure 5.2, that are either only consumers or only producers. In this case, we can no longer have an equivalence between OpenMP tasks and CDDF tasks, but instead we can simply show that the existing properties guaranteed under task causality are trivially correct for *source* and *sink* tasks.

We do not detail the proof here, but it is easy to observe that source tasks, that only produce data, can only lead to resource deadlocks. This comes from the fact that their activations are ready to execute as soon as they are generated, being independent of all other task activations. As resource deadlocks only exist in the case of bounded stream buffers, and they are handled dynamically, this does not modify the deadlock-freedom results involving task causality. On the other hand, sink tasks only read data without producing anything. For this reason, no task activations can depend on activations of sink tasks, which means that any task activation of a sink task is at the end of a causality chain. Only insufficiency deadlocks can occur for sink tasks, which preserves the results involving task causality in the CDDF model.

Another simple way to show that source and sink tasks do not pose any problems for task causality is to note that their activations can only be at the beginning or at the end of causality chains, therefore precluding any CDDF task causality violation which requires task activations from the same equivalence class to be both at the beginning and the end of a flow dependence chain. For this reason, all schedules of source and sink tasks are causal according to Definition 3.39, on page 89, of task causal schedules.

5.3 Static Analysis of OpenMP Streaming Programs

Our static analysis of OpenMP streaming programs is not intended as a substitute for other program analyses, but rather as a way of understanding the information provided by OpenMP streaming clauses to determine some important properties either at the program level or more locally at the task level. This analysis will serve as a basis for deciding the appropriate code generation for each task and the optimizations that can be performed. In general, we ignore the underlying sequential C code and only focus on the OpenMP streaming annotations, which has the advantage of simplifying our analysis, relying on assumption that the compiler pragmas, inserted by the programmer, are correct and *consistent* with the code of OpenMP tasks.

In this section, our objectives are to decide statically whether a program satisfies the conditions exhibited in the CDDF model and to identify tasks or communication patterns that are susceptible to allow aggressive compiler optimizations. We first start by characterizing regular and irregular streaming tasks and we propose a static analysis technique for identifying regular tasks, even in the presence of dynamic communication

patterns. We then present a simple algorithm for building a static over-approximation of the program task graph, which we further use, as discussed in Chapter 3, to evaluate conservative conditions for some of the CDDF deadlock-freedom properties. Finally, we discuss some pragmatic ways to remove or optimize communications between tasks and the control program.

5.3.1 Regular and Irregular Streaming Tasks

Our programming model allows a very high level of freedom in the communication patterns that can be used, in particular because of the duality of our understanding of OpenMP tasks, simultaneously as the short-lived task model used in the current specification of OpenMP and which we model as a *task activation*, and also as a long-lived, persistent and regular process which is often called *filter* in related work. We model the latter as an equivalence class on task activations, which means that the same *syntactic* task construct in an OpenMP streaming program can actually represent multiple such filters, which we call streaming tasks. This is best illustrated by the pattern of dynamic pipeline of filters on Figure 2.16. The number of streaming tasks generated by a single OpenMP task construct is entirely dynamic, bounded only by the number of possible combinations of streams on which this construct can communicate. This number does not only depend on the control flow, but possibly even on program input.

Allowing completely dynamic tasks and communication patterns in our model allows for unchecked expressiveness, but also comes at a cost in terms of optimization opportunities. In the general case, the behaviour of our programs cannot be statically analyzed, which inhibits most of our performance optimizations. However, we show that this can be overcome, at least in the case of regular applications, by applying simple data-dependence analysis techniques that we slightly adapt to our needs. Note that such regular applications simply correspond to the applications that can be written in other stream programming languages where the communication patterns are restricted. In general, common streaming applications, like for instance digital signal processing, tend to be highly regular, as demonstrated by the reliance on less expressive programming models for such programs.

We define regular tasks as tasks that communicate through streams where the stream access indexes can be expressed as closed-form linear expressions of the number of past activation points belonging to the same CDDF task, which we call the task activation index. We also require that such patterns be detected during program compilation, which means that we conservatively reject as irregular all other tasks, where the pattern could not be found.

We write $\mathcal{K}'_e \sqsubseteq \mathcal{K}_e$ if $\mathcal{K}'_e$ is a prefix of the trace $\mathcal{K}_e$. We note ϵ the empty trace.

Definition 5.1 (Regular task)**.** A task t is regular in a CDDF program in a state $\sigma = (\mathcal{K}_e, \mathcal{A}_e, \mathcal{A}_o)$ if for each stream s it connects to, and for each direction $u \in \{R, W\}$ of such connections, there exist two positive integer constants $B_{\langle t,u,s\rangle}$ and $D_{\langle t,u,s\rangle}$ such that:

$$\forall \mathcal{K}'_e \sqsubseteq \mathcal{K}_e, \qquad \sum_{\substack{\pi \in \mathcal{K}'_e \\ (u,s,b,h) \in \pi}} b \quad = \quad B_{\langle t,u,s\rangle} \cdot \left| \left\{ \pi \in \mathcal{K}'_e \mid \xi(\epsilon, \pi) \in t \right\} \right| + D_{\langle t,u,s\rangle}$$

This expression relates directly to the definition of the activation point evaluation function ξ, from Definition 3.8, and more specifically to the computation of the indexes of stream accesses that constitute a task activation. The two constants represent the implicit *burst* perceived by the task on a given communication channel $B_{\langle t,u,s\rangle}$ and its *delay* $D_{\langle t,u,s\rangle}$.

In the general case, the computation of the schedule of stream access indexes in ξ requires computing a sum, on the set of past activation points, of all the bursts b where the direction u and the stream s match. For regular tasks, it is only necessary to know how many activation points of a given task t occurred in the trace prefix. This statelessness appears on the right side of the expression in the form of the ϵ, which is the empty trace, in the evaluation function ξ.

We say more generally that a task is regular in a program if it is regular in each admissible state of the program.

This definition relies on the stronger CDDF notion of tasks, but we can easily see that this has a limited impact and that we can replace it with OpenMP streaming tasks. When tasks communicate through streams identified by scalar variables, we identify four possible cases where we can infer task regularity from static analysis:

1. The simplest case corresponds to tasks that only communicate through single-producer single-consumer streams, with an exception that we will discuss shortly for delay patterns, and use constant bursts.

2. The second case allows multi-producer multi-consumer streams, but requires that all other tasks that interleave their stream accesses share the same control dependence, also using constant bursts.

3. The third case is a generalization of the second one. It does not require secondary producers or consumers to be in the same control dependence, but all tasks must be in a static control relative to each other, which allows to statically determine their relative interleaving patterns. Bursts should also be constant.

4. Finally, the bursts can be allowed to vary provided that this variation presents a cyclical pattern that can be identified statically.

Note that, in this context, we are not interested in the behaviour of producer/consumer relations, but rather producer/producer and consumer/consumer relations on tasks sharing the same streams. In our analysis, the body of a task is not taken into account and is considered to be a black-box for which the OpenMP directive entirely specifies the behaviour. The analysis will therefore exclusively bear on stream variables; access windows only serve as a syntactic tool within the bodies of streaming tasks.

Let us consider the example presented on Figure 5.3, where we illustrate the two first cases of regularity. First of all, note that all of the bursts used here are constant, so this condition is satisfied for all tasks.

The first regularity condition further requires that tasks be single-producer or single-consumer of the streams on which they communicate. This can be very easily verified statically, when streams are represented as scalar variables, using simple scope analysis on

```
int a, b, c; // Scalar streams
int w_in[3], w_out[2]; // Stream access windows
int delay[2]; // Window used to implement a delay operation

// Implement a delay of 2 values on stream ‘‘a’’
#pragma omp task output (a << delay[2]) // (t1)
{
  delay[0] = ...;
  delay[1] = ...;
}

for (i = 0; i < N; ++i) {
#pragma omp task output (a) firstprivate (i) // (t2)
  a = ... i ...;

#pragma omp task output (c) firstprivate (i) // (t3)
  c = ... i ...;

  if (condition()) {
#pragma omp task input (a >> w_in[1]) output (b << w_out[2]) // (t4)
    foo (w_in, w_out);

#pragma omp task input (a >> w_in[1]) output (c << w_out[2]) // (t5)
    bar (w_in, w_out);
  }

#pragma omp task input (b, c) // (t6)
  use (b, c);
}
```

Figure 5.3: Example of regular and irregular tasks in the case of scalar streams.

stream variables, to determine if, for instance, the stream variable a used in the first task's output clause is the same as in the second task. If a stream variable appears in more than one clause of a given type, then the stream is either multi-producer or multi-consumer. In our example, only the last task is regular based on this condition as streams b and c are consumed only by this task.

The case of the first and second tasks is particular in that the first task implements a delay pattern, providing two initial values on stream a. For this reason, when it is possible, based on stream variable scoping, to know that this task only has one activation for the lifetime of the stream variable, its semantics are captured by the constant shift in the linear expression with $D_{\langle t_2,W,a\rangle} = 2$ for the second task. If this can be determined, then the first and second tasks are also regular. We must also note that the semantics of firstprivate and lastprivate clauses is always regular, as they correspond to single-producer single-consumer streams by definition. A more common case of dynamic delays occurs when there is a barrier, as it guarantees that the prefixes of all streams up to the highest reached index are closed. A dynamic delay can then be used for the rest of the computation on all streams to model the past of the execution, so it is possible to have, after a barrier,

regular tasks communicating on streams that were not fit before the barrier.

The remaining third, fourth and fifth tasks are more complicated. Indeed, the third and fifth are both producers of stream c, while the fourth and fifth are consumers of stream a. To complicate matters further, the latter are both nested in dynamic control flow, predicated by the evaluation of a `condition` function. In this case, the second regularity condition applies to the fourth and fifth tasks with respect to stream a as they have the same control dependence. What happens in this case is that despite the impossibility to know statically when these tasks activate, we know that if one activates, so does the second one. This means that the linear closed-form relation allowing to compute the stream access indexes for the read operations in stream a can be simply determined. Indeed, the multiplier constant B, which we will call the perceived burst, is the sum of the bursts of tasks four and five and the delay D is null for the fourth task and equal to one burst of the fourth task for the fifth task. If we call i_4 the counter of task activations of the fourth task t_4 and i_5 that of t_5, then the base offsets are respectively $2 * i_4 + 0$ for t_4 and $2 * i_5 + 1$ for t_5. As stream a has a closed-form solution for these consumers, the fourth task is regular, being the only producer of the stream b. However, the fifth task is not regular because there is no chance of finding a closed-form pattern for its access indexes on stream c. Indeed, the interleaving on c between t_3 and t_5 is dynamic because of the predicate.

Even when they are nested in dynamic control flow, like tasks four and five in our previous example, regular tasks benefit from the buffering semantics of our stream communication, which avoids holes in the stream buffers. This means that the activations of such tasks are executed over a downsampled iteration domain with respect to the loop nest they belong to in the original code, but they only perceive the iterations where such tasks are indeed reached. On their local iteration space, they access streams in the form of contiguous arrays indexed linearly using the expression from Definition 5.1, which enables us to aggressively aggregate both data and work and to generate code that can be further optimized (e.g., vectorized across multiple task activations) by the compiler. These optimizations and opportunities for more efficient code generation are presented in Chapters 6 and 7. An important additional benefit of regular tasks is that the evaluation of activation points is stateless and can be implicitly done during the execution of the activation itself. Indeed, the control program can do as little as incrementing[1] a counter that dynamically expands the iteration domain of a given task. This allows to exploit parallelism, when available, in the control program, by simply making the incrementation operation atomic.

On the other hand, irregular tasks are much harder to optimize and we need to rely mostly on runtime techniques for orchestrating their execution. For this reason, we often distinguish between these two types of tasks and rely on separate runtime and code generation algorithms when dealing with irregular tasks.

The more complicated cases of regularity, cases 3 and 4, are in certain aspects similar to cyclo-static data-flow [13], where the strict regularity of synchronous data-flow programs

[1]This must be done atomically when the task is activated by concurrent control program threads. The evaluation of activation points is not entirely stateless in this regard, but the order of evaluation is no longer important as long as atomicity is ensured.

is relaxed by only requiring that regular patterns appear in the form of static cyclic behaviour. We do not address these cases in the scope of this dissertation.

While the static analysis required to decide the regularity of tasks is very simple in the case of scalar stream variables, this is not usually the case when the program uses arrays of streams or broadcast arrays for communication. In the next section, we discuss some simple cases where static analysis can determine the regularity of tasks using such communication patterns.

5.3.2 Arrays of Streams and Broadcast Arrays

The analysis of communication patterns involving arrays, either as arrays of streams or as broadcast arrays, is more complicated, in particular because a single task directive can represent multiple CDDF tasks depending on the subscripting expressions in arrays of streams. Exact array analyses exist, and we see below that they can be successfully applied in our case, but they are not absolutely necessary to decide the regularity of tasks. We discuss the application of array dataflow analysis [22] to OpenMP streaming tasks in Section 5.4.2.

In this section, our objective is to provide a simple static analysis technique that determines, in common cases, the regularity of tasks using arrays of streams. While less precise than exact array dataflow analysis, this technique is also less restricted as it covers cases where array subscripts are not affine.

Firstly, let us recall that task regularity is fundamentally a question of the possibility of expressing stream access indexes as closed-form affine expressions. This is true when constant bursts are used on single-producer single-consumer streams, and we restrict the current analysis to this simple case for the sake of brevity. The constant burst property is often statically decidable, so our main objective is to decide whether, for instance, the streams used to build a dynamic pipeline are all single-producer and single-consumer.

Let us consider the example on Figure 5.4, where an array of streams A is used to communicate between a statically undefined set of dynamic tasks generated by the inner loop on j and the indirect subscript expressions using the array B. At first sight, static analysis of this type of subscripts fails in general for dataflow analysis techniques. However, our case is slightly different, and we do not require precise information to conclude because flow dependences are implicitly synchronized by stream communication and we only need to know if it is possible that the second task construct would communicate in non-regular patterns. This would happen if, for example, the task reads on the same input stream and writes on different streams. It would imply that the two task activations belong to different CDDF tasks, therefore making the input stream multi-consumer.

From this, we deduce that the condition that must be verified is that the set of CDDF tasks $\mathfrak{T}$ covered by a given OpenMP task construct must have non-overlapping sets of input and output streams:

$$\forall t, t' \in \mathfrak{T}, \quad t = t' \quad \vee \quad \mathcal{I}(t) \cap \mathcal{I}(t') = \varnothing \quad \wedge \quad \mathcal{O}(t) \cap \mathcal{O}(t') = \varnothing \tag{5.1}$$

Indeed, this ensures that no stream is either produced or consumed by more than one CDDF task in this construct, recalling that we are only concerned here with a single,

isolated, task construct. The interaction with the rest of the program must be handled separately.

Proposition 5.2. *To guarantee property* (5.1)*, a necessary and sufficient condition is the existence of bijections between each pair of effective streams used in the streaming clauses of a task construct. In other words, if the stream used in one streaming clause of the task construct is set, then this stream uniquely determines the streams in all other streaming clauses. In the case of the task construct presented below, the CDDF tasks verify* (5.1) *iff* `S1[a]` *uniquely determines* `S2[b]` *and conversely* `S2[b]` *uniquely determines* `S1[a]`*.*

```
#pragma omp task input (S1[a] >> view1) output (S2[b] << view2)
```

Proof. To prove this, one needs only consider that in the absence of a bijection between any pair of streams in streaming clauses, one of the sets of effective streams is of lower cardinality and therefore leads to a re-use of the same stream between different CDDF task, which immediately invalidates property (5.1). In our example above, if there are two possible values of `S2[b]` for a given choice of `S1[a]`, then that specific stream `S1[a]` is multi-consumer as it belongs to two distinct CDDF tasks. The reverse is true by construction as the presence of such bijections immediately guarantees the independence of equivalence classes as it yields, by Definition 3.35 of the CDDF tasks as equivalence classes on the sets of input and output streams:

$$\forall t, t' \in \mathfrak{T}, \quad \Big(\mathcal{I}(t) \cap \mathcal{I}(t') \neq \varnothing \quad \wedge \quad \mathcal{O}(t) \cap \mathcal{O}(t') \neq \varnothing\Big) \quad \Rightarrow \quad t = t'$$

□

There is a particular case is of interest, where the same array of streams is used in multiple streaming clauses of the OpenMP task construct, in our example when `S1 == S2`. If such is the case, the condition from Proposition 5.2 becomes a requirement for bijections between the subscripts used in the different clauses relying on the same array of streams.

We must stress that this condition only applies to single task constructs and the presence, as in Figure 5.4, of additional task constructs communicating through streams in the same array, need to be handled separately. Also note that if different arrays of streams occur in different streaming clauses, the condition remains unchanged as it merely requires that the actual streams used be interdependent.

The problem with the condition from Proposition 5.2 is that it is hardly applicable to static analysis. However, most common cases can be covered by simply eliminating common subexpressions in stream array subscripts. As we only need a bijection between the image sets of both subscripts, the trivial bijection on Figure 5.4 is a function f defined on the set of values stored in the array B and with values in that same set where all values are incremented by one: $f(x) = x + 1$.

This ensures that, as far as the second task construct is concerned, there is no multi-producer or multi-consumer streams involved. In order to decide whether this is also the case when factoring in the first and third task, it is necessary to also analyze the interaction with other OpenMP task constructs, which is much harder. A rather ad hoc solution, for the common case illustrated by our example, is to consider the constraints

```
int A[K]; // Array of streams
int B[K]; // Array of subscripts (not necessarily a permutation)
int w_in[3], w_out[2]; // Stream access windows

B[0..K-1] = ...; // Compute a dynamic map of subscripts

for (i = 0; i < N; ++i) {
#pragma omp task output (A[0] << w_out[2]) firstprivate (i)
  foo (w_out, i);

  for (j = 0; j < K-1; ++j) {
#pragma omp task input (A[B[j]] >> w_in[2]) \
                 output (A[B[j] + 1] << w_out[2])
    {
      bar (w_in, w_out);
    }
  }

#pragma omp task input (A[K-1] >> w_in[1])
  use (w_in);
}
```

Figure 5.4: Identifying single-producer single-consumer streams in arrays of streams.

on the subscripts in the second task construct, originating from the bounds of accesses to array A:

$$\begin{cases} 0 \leqslant B[j] < K \\ 0 \leqslant B[j] + 1 < K \end{cases} \quad \Rightarrow \quad 0 \leqslant B[j] < K - 1$$

This implies that the set of output clauses of this task construct is limited to subscripts in $[1, K-1]$ and that the set of input clauses only touch subscripts in $[0, K-2]$. This allows to conclude that there is no overlapping with the first and third constructs, therefore validating that each stream in the array A is single-producer and single-consumer stream and all tasks are regular, even though the number of tasks cannot be known statically as it depends on the number of distinct values stored in B. Note that there is no need to know that each stream in the array is used, so if for example the array B only contains zeroes, then there is only one CDDF task associated with the second OpenMP task construct.

This type of analysis is very restricted, but it corresponds to many cannonical uses of the dynamic pipeline of filters pattern. In particular, this pattern apears in this form in one of our applications, FFT, which is presented in Section 8.4. This analysis can be applied to our implementation of FFT.

If the stream initiating the pipeline, which is $A[0]$ on Figure 5.4, or that exiting the dynamic pipeline, $A[K-1]$ here, rely on a subscript that cannot be statically evaluated, the analysis may fail. However, such cases should not be common. If there are intermediate results that come out of the pipeline, they would generally rely on a different array of streams. Finally, this analysis is only necessary for optimization. If it fails, we conservatively fall back to considering all tasks communicating over arrays of streams as

irregular. The program will still properly compile and execute, relying on the techniques presented in Sections 7.2 and 7.3, where we provide a fall-back code generation framework that relies on no static analysis.

5.3.3 Static Over-Approximation of the Dynamic Task Graph

One of the important program representations used in the CDDF model is the dynamic task graph, that is used to check deadlock conditions based on properties such as the absence of strongly connected components (SCC) or the reachability of a lastprivate task from a multi-producer and multi-consumer stream. The task graph naturally emerges from the underlying model by assimilating CDDF tasks to vertices and streams to edges. The problem with this approach is mainly that the task graph cannot be determined statically, in particular because of the possible presence of dynamic control flow predicating the activation of a task, which can lead to the task never existing during an execution, or because of the use of arrays of streams, which can generate a dynamic number of tasks with a statically undecidable connection pattern. A simple solution to this issue is to build an over-approximation of the dynamic task graph that covers all possible dynamic task graphs.

Our over-approximation scheme relies on hiding all dynamic communication patterns in opaque aggregated supernodes and superstreams. Supernodes typically correspond to groups of CDDF tasks that cannot be statically differentiated, as is the case in the example on Figure 5.4 for the second task construct, and superstreams are generally all streams within an array of streams. The only things that matters for such special vertices and edges is the way they connect with the rest of the graph and whether there is a self-loop. The latter is conservatively always present unless it can be proven to be otherwise. A second over-approximation consists in hiding the control flow enclosing OpenMP tasks, which could mean that some tasks are absent in the dynamic task graph at some point of the execution, yet in the static task graph all tasks are conservatively represented.

The static task graph is naturally represented, as presented in Definition 3.36, as a directed hypergraph:

$$H = \big(T, S\big) \qquad \text{where} \quad S \in \mathcal{P}(T)^2$$

In our context, the set of tasks T is not a set of CDDF tasks, but rather a set of syntactic OpenMP task constructs, which can correspond to either multiple CDDF tasks, when the task construct is really a supernode, or to only a part of a CDDF task, when multiple syntactic task constructs have identical communication channels. The set of streams is abstracted to only represent relations between tasks, a stream $s \in S$ is defined as a set of producer tasks $P(s)$ and a set of consumer tasks $C(s)$.

Figure 5.5 presents a very simple algorithm for building our static task graph over-approximation. It does not require any context information, relying on a purely syntactical analysis of each OpenMP task construct. Our main goal is to ensure that analyzing this static task graph allows us to guarantee the absence of strongly connected components in the real, dynamic, task graph for any state of the program. We verify this by showing that the presence of a cycle in the dynamic task graph is a sufficient condition to the existence of a cycle in the static task graph.

We first need to remark that, using notations from Definition 3.36, any cycle in the

Iterate over OpenMP task constructs:
`#pragma omp task input (`s_1`, ..., `s_m`) output (`s_{m+1}`, ..., `s_n`)`

Add a new vertice t in the set of tasks T. Further iterate over all input and output clauses, only considering the stream identifiers:

- If the stream identifier is a scalar s_k, then add t to $P(s_k)$ or $C(s_k)$ depending on the type of the clause it appears in.
- If the stream identifier is an access in an array of streams $s_k = A[expr]$, then a single superstream represents all streams in this array. Add t to $P(A)$ or $C(A)$. We will trivially have:

$$P(A) = \bigcup_{s \in A[]} P(s) \quad \wedge \quad C(A) = \bigcup_{s \in A[]} C(s) \tag{5.2}$$

Figure 5.5: Static task graph construction algorithm.

dynamic task graph $H(\sigma)$ is defined by the existence of a set of streams $(s_1, ..., s_n) \in S(\sigma)^n$ such that:

$$\begin{cases} C(\sigma, s_1) \cap P(\sigma, s_2) \neq \varnothing \\ C(\sigma, s_2) \cap P(\sigma, s_3) \neq \varnothing \\ ... \\ C(\sigma, s_n) \cap P(\sigma, s_1) \neq \varnothing \end{cases}$$

As the static task graph represents all streams in the program, either precisely in the case of scalar stream variables or aggregated in a superstream for arrays of streams, each of the streams in $(s_1, ..., s_n)$ have a unique static representative $(s'_1, ..., s'_n)$. Using relation (5.2) from Figure 5.5, we have:

$$\forall i, \quad C(\sigma, s_i) \cap P(\sigma, s_{i+1}) \neq \varnothing \qquad \Rightarrow \qquad C(s'_i) \cap P(s'_{i+1}) \neq \varnothing$$

□

This very coarse over-approximation can be refined, using more precise static analysis tools. The first refinement we consider relies on the analysis proposed in Section 5.3.2 for determining task regularity in the presence of arrays of streams. Indeed, let us consider the result of applying the algorithm from Figure 5.5 to the example on Figure 5.4. The algorithm would create three vertices (t_1, t_2, t_3) and a single edge $A = (P(A), C(A))$ such that $P(A) = \{t_1, t_2\}$ and $C(A) = \{t_2, t_3\}$. As we trivially have $C(A) \cap P(A) = \{t_2\} \neq \varnothing$, the static over-approximation concludes that this program contains a strongly connected component, which is not true.

Refinement strategies are left for future work, though some simple ad hoc solutions can be easily implemented to cover simple cases of dynamic pipelines of filters. In Section 5.3.2, we relied on finding bijections between the sets of subscripts used in arrays of streams appearing in streaming clauses in the same OpenMP task construct to determine task regularity. The same kind of approach can allow to conservatively determine the absence of cycles in a supernode of the static task graph. Indeed, a sufficient condition, though

by no means necessary, to the absence of cycles within a supernode, is that all subscripts used for arrays of streams in output clauses be consistently always strictly greater (resp. always strictly lesser) than the maximum (resp. minimum) of all subscripts used for input clauses. This monotonicity ensures the absence of cycles within the supernode, and allows to remove self-edges in the static task graph. In the previously discussed example, this does indeed allow to remove the self-loop on task t_2 as its output subscripts are always greater than input subscripts: the bijection $f : x \mapsto x + 1$ we found is monotonic.

When the size of an array is static, it may be possible to refine the superstream, and all resulting supernodes, by analyzing each stream separately. However, this does not appear necessary for our current uses of the static task graph. More complex cases can also be handled, in restricted cases like static control programs, by relying on array dataflow analysis techniques that determine static patterns of communication. This topic is further discussed in Section 5.4.2, but the application of such techniques is not within the scope of this thesis.

5.3.4 Statically Decidable Deadlock-Freedom Conditions

Based on the results from Sections 5.3.2 and 5.3.3, we can apply the deadlock-freedom results from Chapter 3 to statically determine deadlock-freedom of OpenMP streaming programs.

The first result comes from applying Theorem 3.42, on page 91, that ensures the absence of all but insufficiency deadlocks in CDDF programs in states where no strongly connected components are present in the dynamic task graph. As the absence of SCCs in the static task graph ensures the absence of SCCs in all states of the dynamic task graph, a streaming program benefits from this result over all executions.

Before discussing the next deadlock-freedom conditions, we need to verify that the notion of task causality is consistent not only across the discrepancy between CDDF tasks and OpenMP tasks, which is addressed in Section 5.2.2, but also in the case of supernodes. Indeed, our static task graph may contain vertices that aggregate multiple CDDF tasks, and we need to ensure that they are causal. This is not difficult as each supernode represents one OpenMP task construct. As task constructs are considered task causal as a whole, they admit a sequential control program order across all CDDF tasks they represent. This condition is sufficient to obtain causality of the underlying CDDF tasks, though it is much stronger than necessary. The supernode as a whole is task causal.

The second result comes from the application of Theorem 3.41, on page 89, which also ensures the absence of all but insufficiency deadlocks, if all tasks are causal and no stream is both multi-producer and multi-consumer. The latter can be determined with the analysis framework for task regularity from Sections 5.3.1 and 5.3.2, while the former is a global guarantee if the OpenMP semantics is respected.

Putting together the first and second deadlock-freedom results yields the third, which is the direct application of Corollary 3.43. This requires using both the regularity analysis and the static task graph to check that no stream is multi-producer and multi-consumer within a strongly connected component.

Finally the fourth result concerns lastprivate spurious deadlocks, and more precisely their impossibility when no multi-producer and multi-consumer streams can reach a

lastprivate task in the task graph, as proved in Theorem 4.10.

5.3.5 Optimizing Communications between Streaming Tasks and the Control Program

Common scalar dataflow analysis techniques, available in all current optimizing compilers, can help to decide when it is legal to eliminate, or optimize, some of the communications between tasks and the control program. Indeed, the control program is the most performance-sensitive part in our model and, for this reason, any communication involving the control program is a potential source of overhead on the critical path. In particular, the presence of communication towards the control program, in the form of lastprivate clauses, represents a blocking operation that can greatly hinder performance and scalability.

As this type of communication relies on non-buffering semantics, and therefore has the same semantics as a simple use of a variable for firstprivate clauses and a definition of a variable for lastprivate clauses, we can integrate the semantics of such clauses in existing static analysis schemes. The result is generally in the form of reaching definitions information, which gives the origin of possible definitions of a variable at the site of a use of that variable. In our case, such information is generally sufficient to decide whether we can optimize communications with the control program.

In order to make things easier, we rely once more on the SSA representation to reason about reaching definitions in our examples on Figures 5.6 and 5.7. We identify two main optimizations, the first one consists in removing lastprivate clauses when the definition they represent has no uses, and the second one consists in replacing a matching pair of lastprivate and firstprivate clauses by a pair of output and input clauses, when the control flow and reaching definitions allow relying on buffering communication semantics, therefore avoiding the involvement of the control program in the communication and removing a blocking synchronization on the likely critical path.

```
x0 = ...; y0 = ...;
#pragma omp task lastprivate (x1)
  x1 = ...;

x2 = ...;

#pragma omp task lastprivate (y1)
  y1 = ...;

#pragma omp task firstprivate (x2, y1)
  use (x2, y1);
```

```
#pragma omp task
  x1 = ...;

x2 = ...;

#pragma omp task output (y)
  y = ...;

#pragma omp task firstprivate (x2)\
                    input (y)
  use (x2, y);
```

Figure 5.6: Optimizing communication between tasks and the control program. Original code (left) and optimized (right).

The required information is entirely provided by representing the program in SSA form, as shown on Figure 5.6, where both optimization opportunities are present. The first task defines the variable x through the lastprivate clause, but this value is lost because of the subsequent definition of variable x, which yields the second SSA name x_2. As can

be seen on the third task that reads the value of x, only the second version is seen, so the definition by the first task is killed before it can be used. The lastprivate clause can therefore be removed.

The second opportunity is illustrated by the second and third tasks in the example, as the definition of variable y, in the form of the lastprivate clause on the second task, only reaches the third task.[2] As the control program is only involved in forwarding this value from the second to the third task, this case can be optimized to avoid involving the control program by replacing the firstprivate and lastprivate clauses by input and output, as shown on Figure 5.6.

This example intentionally simplifies many aspects, like the presence of control flow or uses of data in the control program, as complications usually inhibit these optimizations. Consider, for example, the code on Figure 5.7, where a variable x is defined alternatively by the first or the second task, depending on the evaluation of a predicate `condition`. This leads to the presence of a cond-Φ node that merges the possible definitions x_1 and x_2 of x in variable x_3. The reaching definition to the use site, which is the fourth task, is that of x_3, which prohibits forwarding the stream directly from the producer tasks. The second case is illustrated by the third task, which produces y_1. In this case, the reaching definition at the use site is the proper one, but it is not the only use. As the control program also needs the information produced by the third task, no optimization is possible.

```
if (condition()) {
#pragma omp task lastprivate (x1)
    x1 = ...;
} else {
#pragma omp task lastprivate (x2)
    x2 = ...;
}
x3 = Φ^cond_condition() (x1, x2);

#pragma omp task lastprivate (y1)
  y1 = ...;
use (y1);

#pragma omp task firstprivate (x3, y1)
  use (x3, y1);
```

Figure 5.7: Impossible optimization of the communication between tasks and the control program.

Current on-going work aims at improving the applicability of these optimizations by creating a new task to implement the cond-Φ node. This task simply relies on a stream of the values of the condition predicate to choose the proper value to forward to the consumer task. This work is still in progress and is not within the scope of this thesis.

[2] Assuming that there is no further use of y_1 in the rest of the code.

5.4 Control Program Parallelization in OpenMP

In the CDDF model, the control program is assumed to be sequential in order to guarantee the determinism of data schedules in streams. This property is used to prove several conditions provided by our model, like stream and task causality. They derive from the total order on the evaluation of activation points in the control program. However, as discussed in Section 4.3, requiring a total order, and therefore serial evaluation of activation points by the control program, is not a necessary condition. In fact, we characterize weaker, yet sufficient, conditions for preserving the same semantics, and the determinism, of CDDF programs in Section 4.3.

In this section, we explore two approaches for parallelizing the control program, the first relies on OpenMP compiler directives added to streaming programs to explicitly parallelize the control program, while the second consists in an extension to the static analyses used by existing automatic compiler optimization and parallelization schemes to properly integrate the additional semantics and constraints of OpenMP streaming tasks.

As previously discussed, the issue of control program parallelization is of utmost importance for scalability. A sequential control program is a perfect bottleneck, both in the sense of the computational load imbalance and in the way it centralizes communication with all other parts of the program, therefore leading to a high, centralized, pressure on the communication subsystem and a high level of synchronization.

5.4.1 Explicit Parallelization of the Control Program in OpenMP

As seen in Section 3.4, OpenMP tasks need to be nested in worksharing constructs in order to be properly differentiated. These constructs can have sequential semantics, as is the case for the OpenMP `single` construct, or parallel semantics in the case of `loop` or `sections`. In the general case, and matching the default behaviour in the CDDF model for the control program, we have considered that the worksharing construct was the sequential execution `single` construct. We now take a closer look at the semantics of OpenMP streaming programs that use the parallel types of worksharing constructs.

Among these constructs, a difference must be made between those that preserve sequential program order information in some form, as is the case with the `loop` construct, and those that do not, e.g. the `sections` construct. In the remainder of this section, we focus on the OpenMP `loop` construct[3] as it is both the most commonly used and because the induction variable of the loop is correlated with the sequential order.

The fundamental problem of the evaluation of activation points, presented in Section 4.3, is that it generally relies on the past control program trace, or equivalently on a more compressed carried-over state, which requires the serial execution of the evaluation of some subsets of activation points. In the case of irregular tasks, these subsets are determined by the interleaving patterns of stream accesses, from different tasks, in the same stream. The minimal ordering requirement in this case is that all activation points accessing one stream either for reading or for writing need to be in the same partition

[3]The `loop` construct is used to annotate DOALL loops and will appear as a `omp parallel for` construct in the C version of OpenMP annotations.

of the control program and be evaluated in sequential control program order. This is a strong constraint, which is difficult to check statically.

To clarify this, let us consider the example presented on Figure 5.8, where it is understood that the analysis has failed to determine whether the tasks are regular. In order to verify the validity of a control program parallelization directive with respect to activation point evaluation, we will rely on the automatic compiler parallelization scheme presented in Section 5.4.2, where we only need to consider the analysis of OpenMP task annotations. Indeed, in this example, the programmer guarantees that any work occurring in the loop nest is data-parallel with respect to the j loop, so this does not require further attention.

```
#pragma omp parallel for
  for (j = 0; j < K-1; ++j) {
    for (i = 0; i < N; ++i) {
      // do some work
#pragma omp task input (A[B[j]] >> w_in[2]) \
                 output (A[B[j] + 1] << w_out[2])
      {
        bar (w_in, w_out);
      }
    }
  }
```

Figure 5.8: Evaluating activation points of different tasks in parallel.

We need to determine whether some activation points, belonging to different iterations of the j loop, may use the same stream for either input or output. We do not have a better tool than the existing array dataflow analysis framework, which we extend in Section 5.4.2 for this purpose. On Figure 5.8, the control program parallelization is perfectly acceptable if B represents a permutation on $[0, K-1]$, for example if $\forall j \in [0, K-1], B[j] = j$, as this would imply that each partition of the iteration domain contains activation points accessing different streams. However, if the analysis fails, we need to conservatively execute the loop sequentially.

The case of regular tasks is more appealing, because the evaluation of activation points is stateless[4] and therefore easy to parallelize. In this case, a task activation needs only know its activation index in the task's activation space in order to evaluate, on its own, the indexes it needs to access in streams. A task's activation space is defined by the number of activations that have been generated for the task, so it is independent of the control program control flow.

If such a task has no firstprivate and lastprivate clauses, and its horizons are constant for each stream, then a simple task activation counter incrementation is sufficient to generate a new task activation. This must be done atomically if multiple threads of execution can generate activations for the same task.

The parallelization of loop nests where tasks communicate with the control program is more complicated. Such communication creates a relation between the iteration set of the

[4]Once the incrementation of the activation counter is made atomic.

```
#pragma omp parallel for
  for (i = 0; i < N; ++i)
    {
#pragma omp task firstprivate (i) output (x)
       x = foo (i);

     if (condition (i)) {
#pragma omp task firstprivate (i) output (y)
         y = foo (i);
     }

#pragma omp task input (x) lastprivate (z)
       z = foo (x);

   }
```

Figure 5.9: Control program parallelization and communication with the control program.

loop in the control program and the activation indexes of any task which communicates with the control program. Let us consider the three regular task constructs on Figure 5.9. The first task represents the favourable case: it is possible to evaluate its activation points concurrently. As the task's activation index is fully identified by the induction variable i, it is possible to preserve the order of the stream identified by i. This is not the case for the second task, which is nested in non-static control. Unless it is possible to know how many times the `condition ()` predicate evaluates to true over the set of iterations lower than i executed by other threads, it is impossible to decide the stream index where the value of i should be written by the control program when evaluating the activation point at iteration i. This task construct therefore forces the sequential execution of the whole i loop in the control program. Finally, the third task construct uses a lastprivate clause, which inherently serializes the execution of both the control program and the task.

While firstprivate and lastprivate communication is regular, with respect to our definition of regularity, it can represent major hurdles to control program parallelization. For now, we only allow such parallelization to occur when no lastprivate clause occurs and when no control flow downsamples the activation space of tasks communicating with the control program.

5.4.2 Streaming Programs with Static Control and Automatic Parallelization

Optimizing compilers rely on dataflow analyses to determine the validity of program transformations aiming at reducing the execution time in many different ways, such as improving locality or enabling parallel execution. Automatic program parallelization is of great interest for the control program required by the CDDF model as it can improve its performance and reduce its scalability issue.

Such techniques generally cannot understand the effects on memory of the opaque runtime calls generated by the expansion of compiler directives, such as OpenMP constructs. In order to improve the applicability of dataflow analyses to OpenMP programs,

we modify these analyses to include the proper semantics and constraints of OpenMP directives.

Dataflow analyses are rooted in the divergence between the mathematical, declarative, notion of a variable and their temporal values in imperative programs. The analysis of memory dependences requires the existence of a logical time to order events and commonly distinguishes between true and false data dependences. False data dependences, anti (write after read) and output (write after write) dependences, are sometimes eliminated by expansion techniques, that avoid reuses of memory. To sum up, the fundamental objective of any form of dataflow analysis is to determine the relations between statements that write a variable and those that read it. This allows to ensure that a program transformation does not modify the semantics of a program.

In our case, things are quite different. The semantics of stream communication naturally enforce all flow dependences within streams, so there is no need to worry about, for example, a consumer task occurring in the main program before its producer. On Figure 5.10, both codes have exactly the same semantics, because streams implicitly synchronize reads and block consumers until the appropriate value is written. Furthermore, as streams are perceived as infinite arrays (even though this is also handled through synchronization), privatization is already baked in, so there can be no issue with memory reuse.

```
#pragma omp task input (a)
  ... = a;

#pragma omp task output (a)
  a = ...;
```

```
#pragma omp task output (a)
  a = ...;

#pragma omp task input (a)
  ... = a;
```

Figure 5.10: Equivalent programs with buffering semantics for streams. Data flow dependences in streams are implicit.

However, while the usual data dependences are not an issue, a new type of dependences occur, which are due to the main specificity of our streaming model: the sequential order of the control program determines the schedule of data in streams. This effectively means that the flow dependences themselves could change if this order is not respected, in the spatial sense rather than in the temporal sense, which is a consequence of replacing the notion of time in the control program with the notion of position, or precedence, in a stream. As we have discussed in Section 4.3, the real issue is to ensure that the original schedule of data is preserved, which requires that stream operations of the same type and on the same stream occur in the sequential program order.

The adjustment required to common dataflow analysis techniques is simply replacing the usual definition of data dependences, only for the analysis of the streaming clauses in OpenMP task constructs, by input[5] and output dependences only, when considering that stream variables are written when they appear in output clauses and read when they appear in input clauses. The analysis for the remainder of the program is unchanged, except that the bodies of streaming tasks are simply considered as black-boxes that have

[5]We recall the definition of the often left aside input dependence, which is a read after read dependence. It is generally of no interest in dataflow analysis.

no memory effects. Other clauses, like firstprivate or lastprivate, are simply considered to be either a read or a write operation on the variable that appears in such clauses. The case of shared variables is more complicated and requires to also analyze the body of the task.

Enabling dataflow analyses for streaming programs, and properly incorporating the semantics of activation point evaluation, allows the control program to potentially benefit from all program transformations enabled by these analyses. This includes both common loop nest optimizations and automatic parallelization, with the obvious restrictions that expansion and privatization of streams is not possible in the general case. The study of stream communication patterns that would allow, in a certain sense, to de-multiplex the interleaving of data in streams, therefore allowing to statically decide to split (or privatize) streams is left for future work.

This first benefit is to mitigate the bottleneck effect of the control program's semantics in the OpenMP streaming extension's implementation, but we can go even further. In the restricted case of static control programs, or even just static control parts of programs, we can also rely on the powerful array dataflow analysis [22] and make the same adjustment to the dependences that need to be taken into account.

Array dataflow analysis has the same objective as the scalar version, to determine source functions which provide the origin, in a loop nest's iteration space, among the statements present and at a given time, of a variable's definition. The main difference is that in this case the variable is an element in an array.

In our case, this analysis can be applied to cases where arrays of streams are used for communication, and can allow determining whether the CDDF tasks present in a loop nest are regular. Indeed, if the schedule constraint derived from array dataflow analysis is affine and the source functions of all streams used by a task are identical[6], then the task is regular in the static control part analyzed. In order to ensure that regularity is not broken by past stream accesses, it would still be necessary to insert a barrier before the static control part, to allow considering the past of each such stream as a delay in Definition 5.1.

The generalization of dataflow analysis to stream data structures, instead of arrays, is a very complex problem in the general case. Restricted cases, for instance when all tasks are regular, may be more tractable, but are not covered in this thesis.

5.5 Conclusion

In this chapter, we presented the static analysis framework required to apply the results of Chapter 3 to OpenMP streaming programs and to enable their optimization. We started by showing how the semantics of our programming model maps onto the representation of control-driven dataflow programs. We discussed the differences between the two semantics, in particular the divergence on the notion of task, and concluded

[6]These functions are adjusted to consider input and output dependences only, so they provide the source of the previous occurrence of a given stream either as input or, independently, as output of a task. In CDDF terminology, this represents the previous activation point in the control program trace that accesses the same stream with the same (read or write) direction, which is precisely the information we require.

that this difference does not preclude applying the results proven in the CDDF model to OpenMP streaming programs. In a second step, we developed our static analysis framework, which heavily relies on existing analyses and optimizations. This allows us to determine whether streaming tasks are regular, and can benefit from optimized runtime support, which we present in Chapter 6, and code generation, presented in Chapter 7. These analyses also enable existing transformations like loop nest optimizations or even control program parallelization, transparently handled by adjusting their static analyses to properly account for the semantics of streaming tasks.

Some particular classes of streaming applications, like synchronous dataflow [43] or cyclo-static dataflow [13], have been extensively studied and present a high level of regularity that can be exploited in very efficient ways. A natural extension of our static analysis framework would attempt to determine when programs conform to the stricter conditions imposed in these models and allow to integrate the optimizations, as well as the guarantees, developed for these models in our code generation. The first step in this direction consists in recognizing regular tasks when there are multiple producers and consumers using the same streams, by finding cyclic patterns of stream access interleaving. A second step requires to show that the production and consumption rates are identical on some constant control program iteration range. The development of such techniques presents interesting opportunities for further research.

Chapter 6

Runtime Support for Streamization

In this chapter we present the design and implementation of our runtime support for streaming. We provide stream communication synchronization primitives functionally consistent with the semantics of the stream prefix constraints presented in Chapter 3, as well as an implementation of the activation point evaluation function and scheduling functions for task activations. We further present a dynamic load-balancing scheme and a runtime deadlock detection algorithm. Focused on performance, the implementation avoids atomic operations and memory fences on architectures that provide a total store order memory model, except in some uncommon situations.

Ce chapitre présente la conception et l'implantation de notre système de runtime pour le streaming. Il fournit des primitives de synchronisation des communications par streams fonctionnellement compatibles avec la sémantique des contraintes sur les préfixes de streams, voir chapitre 3, ainsi que les fonctions d'évaluation des points d'activation et d'ordonnancement des activations de tâches. Nous y présentons également un mécanisme d'équilibrage dynamique des charges et un algorithme de détection dynamique des interblocages. L'implantation cible tout particulièrement la performance, évitant l'utilisation d'opérations atomiques et de barrières mémoire sur toute architecture fournissant un modèle mémoire de type Total Store Order*, sauf dans des situations peu communes.*

6.1 Introduction

The natural focus of our runtime implementation is to avoid any unwarranted overhead on the critical path. In a streaming application, the critical path is generally materialized by the heaviest filter, though of course, in our programming model, this can also be the control program. As we argue in Section 6.3.3, the key insight is that, unless the task graph is perfectly balanced, one of the tasks is slower than the others and this leads to a pattern of communication where all of its input stream buffers are consistently full while all of its output stream buffers are consistently empty. As a result, this task never has to wait for input data or available output space, yet the synchronization necessary to ensure that the task can proceed still constitutes an overhead on the critical path.

Our primary objective is to ensure that the synchronization of such a task is as inexpensive as possible, considering that in the common case it should only require a

single load operation from a locally cached variable or a register and that, in the worst case, it should not represent more than a few load operations from non-local variables. We achieve this by relying on the monotonicity of all stream access indexes, which allows us to use a software cache scheme and to avoid memory fences and atomic operations on architectures where the memory model provides at least total store order consistency. To further ensure that this *critical task* runs unimpeded, even in the event that, for some extraneous reason, quiescence is requested on this task, we designed a quiescence algorithm that allows this task to continue working as long as it does not have down time, which leads to it being the last to converge to consensus and therefore the one that spends the least amount of time in quiescence.

Less pragmatically, we try to achieve a real decoupling of threads through stream communication, which means that all memory latency is tolerated rather than explicitly synchronized. As we know that indexes are always monotonically increasing, reading stale information about the state of another task cannot impact the correctness of our synchronization scheme, it can only delay the moment the synchronization primitive passes. By ensuring that information always only flows into one direction, with respect to each synchronization primitive, we avoid most memory consistency issues on our main target architecture, x86.

This chapter is organized as follows. Section 6.2 presents the synchronization algorithm used by our stream communication, with a particular focus on the impact of memory model relaxations on our algorithms and a discussion on the memory fences required to ensure correctness in the case of more relaxed architectures. Section 6.3 provides the necessary runtime support for evaluating activation points and scheduling task activations, including load balancing concerns. Section 6.4 presents the instantiation of our resource deadlock detection and resolution algorithm from Chapter 3, with the required schemes for achieving quiescence of all tasks connected to a stream and for dynamically resizing stream buffers. Finally, Section 6.5 presents the code generation interface used in Chapter 7.

6.2 Synchronization of Stream Communication

The synchronization algorithm for stream communication relies on four synchronization primitives each handling the specific synchronization requirements of producer or consumer tasks, with two distinct versions for regular and irregular tasks. Before we present the algorithms and discuss the different cases, we need to give a better view of the way stream communication is organized. Unlike the abstract CDDF view of infinite stream buffers, the implementation relies on circular buffers that require synchronization for both flow dependences and anti dependences (see Section 4.4).

As discussed in Chapter 3, our communication scheme over-synchronizes flow dependences by enforcing the stream prefix order, which synchronizes dependences over closed prefixes. This means that the data stored at a given stream index can only be read once all indexes equal or lower have been written. It results in a clear separation, in contiguous areas, between read and write areas in the stream. Figure 6.1 illustrates this behaviour. On the left-hand side, the frontier between the areas where read operations are allowed, which is commonly called pressure, is determined by the lowest stream index that has

not yet been written. All lower indexes having already been produced, they form a closed prefix in which read operations are allowed; but no write operations can occur anymore in this area.

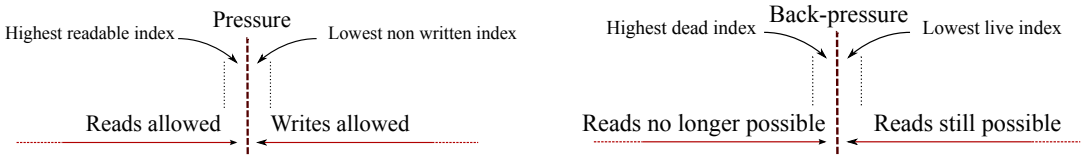


Figure 6.1: Pressure (left) and back-pressure (right) in stream communication.

The counterpart of pressure is back-pressure, illustrated on the right-hand side of Figure 6.1, which delimits the stream indexes that can no longer be read from those that are still live. Pressure and back-pressure are used to implement infinite streams with circular buffers: the space behind the back-pressure frontier is used to write new pieces of data. Figure 6.2 presents the circular buffer view, where the pressure and back-pressure frontiers define the live window, consisting in all stream indexes that are written, accessible to the consumers and still needed by some consumer task.

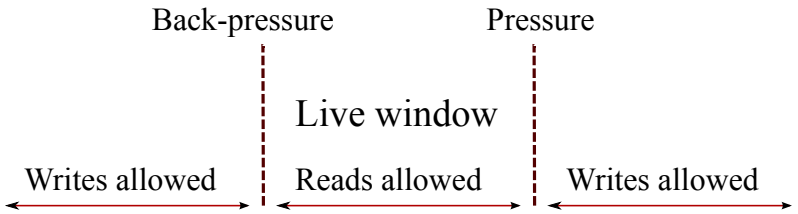


Figure 6.2: Stream circular buffer separation between read-exclusive and write-exclusive areas.

Our synchronization algorithm is based on this implementation of streams, with two synchronization primitives for each of the frontiers, one for the production side and one for the consumption side. Pressure is implemented through the `commit` and `update` pair, while back-pressure relies on `stall` and `release`. These primitives are informally defined as follows. All primitives take a stream and an index as parameters.

commit allows producers to make data produced available for consumption. We require strict monotony of stream accesses in each thread, so once an index is committed by a thread on a stream, this thread relinquishes all rights to write to any index equal or lower in that stream.

release allows consumers to relinquish the memory space in a stream after consuming data. This operation is also always monotonic and the consumer waives all rights to further access indexes in streams lower or equal to the highest index released.

update is a blocking operation that ensures, upon completion, that the caller thread can read up to a given index in a stream.

stall is also blocking and corresponds to the equivalent request for a producer. Upon completion, the caller may write up to the requested index in the stream.

In order to allow evaluating the frontier positions without re-constructing the stream index contiguity from scratch, all producers (resp. consumers) need to commit (resp. release) up to a given index for the prefix up to that index to be considered closed. Relying on this restriction, the decision of the highest readable index or highest dead index (which can be therefore re-used), as shown on Figure 6.1, requires only computing the minimum across all producers (resp. consumers) of the maximum committed (resp. released) index. Indeed, this minimum computation allows to find the lowest non-written index (resp. lowest live index), which is just across the frontier.

Remark 6.1. *A very important note: we consider all threads to communicate on monotonically increasing indexes. This plays a key role in our synchronization algorithm, and allows to avoid memory fences on architectures providing the* total store order *memory model [65], which is partially true for x86 architectures [34,50]. Total store order only allows reordering stores after loads, which does not invalidate our synchronization algorithm [3]. However, when this consistency model restriction is unavailable, we require store and load memory fences, as we discuss below.*

Note that, in our execution model, this monotonicity restriction does not mean that task activations need to be executed in sequential program order, but rather that on each thread executing task activations from a given CDDF task we preserve this order.

6.2.1 Synchronization Patterns

To give a better view of the way our runtime is organized, we provide, on Figure 6.3, a sequence diagram for the synchronization occurring between the control program and the threads responsible for executing task activations. Some of the constructs on this diagram, in particular the program trace after the expansion pass and the control streams, are only introduced later, in the remainder of this chapter as well as in Chapter 7. They have been partially simplified to make reading easier.

The code for this example, annotated with our extension, is at the top of Figure 6.3. It contains two streaming tasks, communicating through a stream x. The sequence diagram for the execution of this program shows the interaction between the control program and the two tasks, as well as in between the two tasks, with streams as intermediaries. Because they mediate interactions, streams are created before any synchronization operation can be issued. In our example, the data stream, x, is created at the very beginning of the control program's execution, before any activation point is evaluated. The control streams, one for each streaming task, are created just before the task itself.

The control program trace, on the left of the figure, presents the most relevant operations making up each of the four activation point evaluations shown on our example. The first evaluation of an activation point, for a given streaming task, also initializes the task's data structures, among which is its control stream, and launches its first thread[1]. Subsequent evaluations of activation points do not have these two first operations.

Aside from this initialization phase, the evaluation of activation points generates task activations, which are stored in the control streams. Hence the control program needs

[1]Because of the growing diagram complexity, we do not show teams of worker threads for tasks, nor multiple threads for the control program.

```
int x;
for (i = 0; i < N; ++i) {
#pragma omp task output (x) /* Task T1 */
  x = ...;
#pragma omp task input (x) /* Task T2 */
  use (x);
}
```

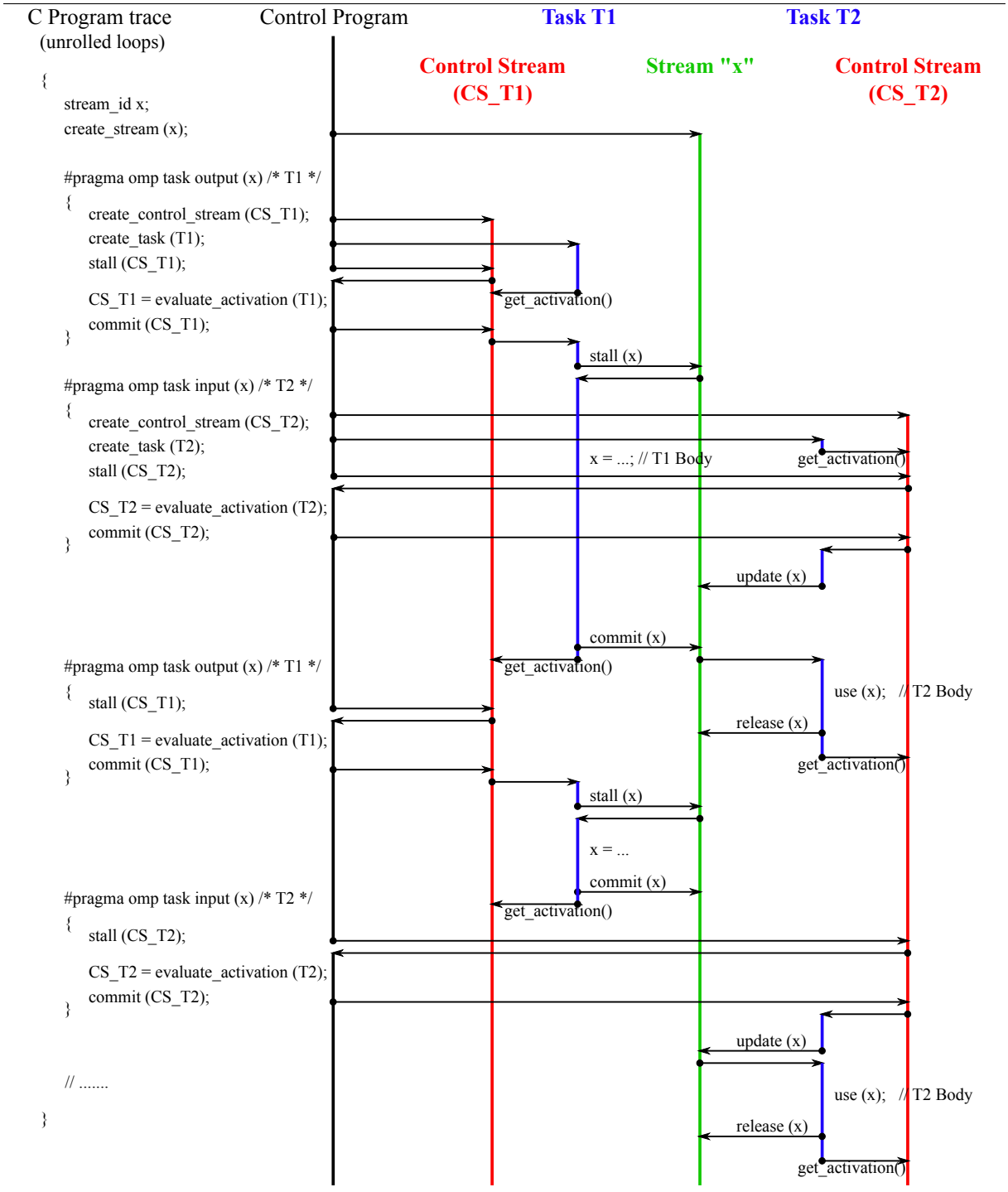


Figure 6.3: Sequence diagram (bottom) for synchronization in a streaming program (top).

to call the stream synchronization primitives `stall` and `commit`. As `stall` is a blocking operation, synchronizing the control program. Once the stall operation completes, the control program computes the stream indexes for the new task activation, with `evaluate_activation`, and commits this activation to the control stream. This same sequence of operations occurs for all subsequent evaluations of activation points.

Once created, task `T1` requests work from its control stream, `CS_T1`. This uses a different function, `get_activation`, than the `update` primitive as the control stream has a slightly different semantics than the data stream x. It carries the control flow information. The `get_activation` call is blocking and task T1 only resumes execution once the commit from the control program completes. Once the task has acquired an activation, it proceeds to acquire space in its output stream buffer with a call to `stall` on stream x. It can then execute the body of the task and finally commit the new value produced for x.

In the meantime, the control program creates the second task, and its control stream, and generates the first activation for task T2. This task acquires this first activation, then requests input data from x. It blocks until task T1 produces sufficient data elements on x, then it can execute its body, release the memory space in stream x and request a new activation from its control stream `CS_T2`.

6.2.2 Synchronization Algorithm for Regular Tasks

The synchronization algorithm for regular tasks relies on the assumption that the threads executing any outstanding task activations respect task order as defined in Definition 3.37. Each thread of execution is further limited to task activations of a given OpenMP streaming task, which means they all belong to the same CDDF task and share the same work function. This ensures that, on each individual thread, the order of accesses to all streams is monotonically increasing.

We present here the algorithm for regular tasks (see Definition 5.1 and their static identification in Section 5.3.1), where a closed-form expression allows the task to know the position where possible subsequent stream accesses would occur, while the general case of irregular tasks is presented in the next section.

The code generation required for the synchronization of stream communication is presented on Figure 6.4. It consists in issuing `update` calls for each input stream on the highest stream index read, in that stream, in the activation to be executed[2]. These blocking calls ensure that the necessary input data is already available on all input streams. Then, we need to similarly ensure that enough space is available in all output streams with calls to `stall`. After the work function of the task activation completes, calls to `commit` and `release` on all proper streams make the new data produced visible and allow reuse of stream buffer space occupied by data no longer needed. The indexes used for `release` and `commit` calls is based on the knowledge of the amount of data in streams that can be skipped due to regularity. These indexes are one less than the lowest index accessed by the next activation in this task. The precise code generation is detailed in Chapter 7. Note that we possibly commit and release more than the burst on each stream,

[2]As the synchronization is restricted to closed prefixes on stream indexes, the synchronization of a task activation always requires requesting access to the highest index it accesses in each stream.

which helps release resources in the stream buffer.

```
// Execute activation a:

∀s ∈ I(a) : update (s, max_(R,s,i)∈a (i));
∀s ∈ O(a) : stall (s, max_(W,s,i)∈a (i));

  execute_work_function (a);

∀s ∈ O(a) : commit (s, next_stall_index - 1);
∀s ∈ I(a) : release (s, next_update_index - 1);
```

Figure 6.4: Synchronization code required for stream communication of regular tasks.

At this point, we are not interested on how task activations are represented or how the indexes are computed, but rather on how the four synchronization primitives enforce dataflow pressure and back-pressure.

Each thread, called p to avoid confusion with tasks, stores information pertaining to its current stream access positions for each stream on which it communicates. This data is shared among all threads communicating on the same streams, but only the owner thread updates its own information. Our algorithm relies on a complete separation of load and store operations. Each pair of primitives, commit/update or release/stall, relies only on one-way communication from one group of threads (i.e., producers or consumers of a stream) to another group. It is important that different store operations from one given thread be perceived in the same order by load operations on threads from the group at the other side of the stream communication, but we do not assume any other preservation of the order of memory operations.

The data structure we use for simplicity is a globally visible map $M(p, u, s)$ indexed by thread identifiers, stream access directions (i.e., read or write) and stream identifiers. The data stored in this map are stream access indexes and it is initialized with zero values. All operations on this map occur within calls to synchronization primitives.

Figure 6.5 presents the pseudo-code of the four synchronization primitives. The first two are blocking primitives that only read data from the map M, while the last two are non-blocking and only write to M. In order to prove the correctness of this algorithm, we first remark that the `commit` and `release` primitives always monotonically increase the value in the map M, which ensures that any stale value that could be read in the `update` or `stall` primitives can only result in lower results of the minimum computation. This can induce delays before the primitives complete and allow the thread to make progress, but it only requires that the latest updates on M be seen at some point.

Critically, our algorithm requires that subsequent write operations performed by a given thread be seen in the original order by all other threads. This ensures that the write operations performed during the execution of the work function of the task are visible before the updates on map M are perceived. On the other side, we further require that load operations occur in program order, which ensures that the read operations in map M occur before any read operation in streams during the execution of a consumer's work function. Overall, these two constraints allow to ensure that a producer's store

```
update (s, index) {
  while (min_{p∈producers(s)}(M(p, W, s)) < index);
}

stall (s, index) {
  while (min_{p∈consumers(s)}(M(p, R, s)) < index);
}

commit (s, index) {
  M(th_id, W, s) = index;
}

release (s, index) {
  M(th_id, R, s) = index;
}
```

Figure 6.5: Synchronization primitives for regular tasks.

operations to a stream index and a consumer's load operations to that same index can only occur in the data-flow order.

6.2.3 Correctness of the Synchronization Algorithm with Relaxed Memory Models

In order to use architectures with weaker memory models than TSO, we introduce memory fences in the synchronization primitives to prevent relaxed memory ordering to modify the semantics of our algorithm. To that effect, we primarily rely on the set of memory ordering primitives provided by the memory models of the SPARC [65] architecture as they provide the most fine-grained control of ordering, though we also provide the requirements for x86 and POWER architectures as well as the sufficient Linux Kernel memory fences. We present the informal semantics of the fence operations, along with their conversion to Linux primitives and x86 instructions on Table 6.1. We discuss the POWER architecture subsequently, as its model does not provide for a trivial mapping. The lack of control on the granularity of ordering provided by the Linux primitives forces us to rely on much stronger fences than required, notably in the case of `load_load_fence` and `load_store_fence`, where the implementation leads to conservatively relying on the stronger x86 LFENCE and MFENCE instructions.

Note that the memory fences we add also work as compiler fences, preventing the compiler from re-ordering the memory operations across fences. When the memory fences are unnecessary on a given architecture, we replace them with a compiler fence, which uses the following pattern for the GCC compiler.

```
// Compiler fence
__asm__ __volatile__ ("" ::: "memory");
```

Before we provide an implementation of our synchronization algorithm with memory fences, let us first analyze the underlying communication patterns and the synchronization

	Informal semantics	Linux Kernel	x86	x86-CC
`load_load_fence`	Order prec. loads before subseq. loads	`smp_rmb ()`	`LFENCE`	`NOP`
`load_store_fence`	Order prec. loads before subseq. stores	`smp_mb ()`	`NOP`	`NOP`
`store_load_fence`	Order prec. stores before subseq. loads	`smp_mb ()`	`MFENCE`	`MFENCE`
`store_store_fence`	Order prec. stores before subseq. stores	`smp_wmb ()`	`SFENCE`	`NOP`

Table 6.1: Informal semantics of SPARC memory fences and their replacement in terms of Linux Kernel primitives, x86 instructions and for x86 with coherent write-back memory (x86-CC).

mechanisms required to enforce their correctness. Figure 6.6 illustrates the two communication patterns involved, where the sequentially consistent behaviour must be observed: the message passing (MP) pattern for pressure and the load buffering (LB) pattern for back-pressure.

Initial state: [stream_buffer] = A [commit_index] = 0 [release_index] = 0

Pressure (message passing pattern)			
Producer thread (P)		**Consumer thread (C)**	
Execution:	W[stream_buffer(idx)] = B	Update:	R[commit_index] = idx
	LWSYNC **SFENCE**		**ADDR or DATA** **LFENCE**
Commit:	W[commit_index] = idx	Execution:	R[stream_buffer(idx)] = B
Objective: (C:R[commit_index] = idx) => (C:R[stream_buffer(idx)] = B)			

Flow-dependence

Back-Pressure (load buffering pattern)			
Consumer thread (C)		**Producer thread (P)**	
Execution:	R[stream_buffer(idx)] = A	Stall:	R[release_index] = idx
	ADDR or DATA --------		**ADDR or DATA** --------
Release:	W[release_index] = idx	Execution:	W[stream_buffer(idx)] = B
Objective: (C:R[stream_buffer(idx)] = A)			

Anti-dependence

Figure 6.6: Communication patterns in pressure and back-pressure algorithms, with the required fences for POWER (green, left) and x86 (blue, right) architectures.

The pressure algorithm enforces a flow dependence, implementing a message passing pattern. In order to achieve the objective that if the consumer thread can read the value `idx` in the `commit_index`, then it necessarily also reads the new value, `B` when accessing `stream_buffer(idx)`, we need to add memory fences to prevent the reordering of write operations by the producer and that of read operations by the consumer.

The back-pressure algorithm enforces an anti-dependence, which fits the load buffering pattern issue. Indeed, if loads can be reordered after subsequent stores, then our objective, that the consumer reads the old value of the data in the stream buffer, before the producer

can overwrite this value, requires to add synchronization.

In order to guarantee that the synchronization proposed on Figure 6.6, we rely on the two analyses of the x86 and POWER architectures [63,64] by Sarkar et al. as well as on *Intel*'s white paper (IWP) on memory ordering [33].

For the x86 architecture, the LFENCE (load fence) and SFENCE (store fence) instructions used on Figure 6.6 correspond to the Linux implementation of memory fences, and the specification in the Linux Kernel. However, in the IWP, the first principle states that no load is reordered with other loads and that no writes are reordered with other writes, with the restriction to the case of coherent write-back memory, which per the IWP: "[Write-back] memory is what is typically used for memory made available by C *malloc* library calls [...]". For this reason, we do *not* rely on this type of fences in our implementation[3] and simply use compiler fences. The case of the back-pressure algorithm falls within the scope of the second principle stated within the IWP, and does not require synchronization.

The POWER architecture relies on a much weaker memory model, where memory operations can be reordered in any patterns, and the store operations may be propagated to other cores out of order. For this reason, we need stronger memory synchronization. The LWSYNC (lightweight sync) is a barrier instruction that will ensure that the order of propagation of stores is enforced across the barrier. It represents a much stronger synchronization than we require, but it is the weakest sufficient synchronization provided in the POWER model. In our case, for the pressure algorithm, it ensures that the second store cannot be propagated to any core that has not yet seen the first store. On the consumer side, a data or address dependence is required to ensure that loads are not reordered with each-other. Because of the existing control dependence in our algorithm, an ISYNC operation could be used, as argued in [63], but it is more expensive than simulating a dependence[4], by adding the value of `idx`, XORed with itself, to the pointer through which the stream buffer is accessed. For the back-pressure, it is sufficient to use the simulated dependence pattern in both producer and consumer code, though the authors of [63] note that this type of load buffering issue was impossible to observe in their tests on the PowerPC architecture, despite its presence in the specification.

When the architecture does not provide sufficient guarantees on memory ordering, we need to add fences to the synchronization primitives as shown on Figure 6.7. We also present here the ordering operations for POWER, which we will omit in the subsequent algorithms as we consistently rely on the same two communication patterns.

update requires a memory fence that prevents reordering load operations after other load operations. As this primitive guards the read accesses to streams, this prevents loads from M being executed after some data in streams has already been loaded, therefore enforcing flow dependences.

[3]Our view is further confirmed by the authors of [64] as they qualify the LFENCE and SFENCE instructions as "(perhaps costly) no-ops" for x86-CC.

[4]There is not always a real data dependence, in our implemented algorithm, between the address computed for accessing the stream buffer and the value of `idx`. Indeed, the computation of this address can rely, in the case of *regular tasks* as defined in Section 5.3.1, on the known values of constant burst values and the previous access address, which means that `idx` is not used in the address computation, as we show in Chapter 7. In such cases, a dependence on `idx` can be simulated.

```
update (s, index) {
  while (min_{p∈producers(s)}(M(p, W, s)) < index);
  load_load_fence (); // NOP on x86-CC
  // For POWER:
  // control dependence + ISYNC or either data or address dependence
}

stall (s, index) {
  while (min_{p∈consumers(s)}(M(p, R, s)) < index);
  load_store_fence (); // NOP on x86-CC
  // POWER: either data or address dependence
}

commit (s, index) {
  // POWER: LWSYNC barrier
  store_store_fence (); // NOP on x86-CC
  M(th_id, W, s) = index;
}

release (s, index) {
  // POWER: either data or address dependence
  load_store_fence (); // NOP on x86-CC
  M(th_id, R, s) = index;
}
```

Figure 6.7: Memory fences required on relaxed memory systems.

stall relies on a load-store fence that prevents loads from being scheduled after subsequent stores. This primitive guards against violation of anti dependences, so it needs to ensure that the data in M is read before anything can be altered in stream buffers.

commit needs a store-store fence where store operations cannot be reordered with respect to other store operations. Committing data is also part of enforcing flow dependences, and it is necessary to ensure that M is not updated before all data produced on output streams is stored.

release needs the same load-store fence as `stall`. It also enforces anti dependences, ensuring that read operations in streams have all completed before the update of M allows reuse of memory.

Most importantly, all these fences are unnecessary in total store order (TSO) and would only be required on x86 in special configurations, when running in the uncommon out-of-order stores mode or when using non-temporal instructions. Fences are, however, needed on architectures implementing the relaxed memory order (RMO), the POWER memory model or IA64. On the intermediate partial store order (PSO) model, only the store-store fence would be required in the `commit` primitive.

In a recent publication, Attiya et al. [8] show that the implementation of many common concurrent algorithms, like sets, stacks, mutual exclusion or, more importantly for us, queues, is impossible without relying on either a read-after-write (RAW) or an atomic write-after-read (AWAR) pattern. Both patterns inherently rely on memory fences or atomic operations, the common example being Dekker's mutual exclusion for RAW, where all non-sequentially consistent architectures would fail because of a store after load reordering, and a compare-and-swap lock for AWAR.

However, our algorithm does not fall within the scope of their proof because we do not exactly implement FIFO queues for stream communication. Despite the similarities, our algorithm does not have the fundamental issue of a multi-consumer or multi-consumer FIFO queue: there is no contention, no consensus must be reached between concurrent producers or concurrent consumers to a given stream. As stream access indexes are pre-computed, or pre-scheduled, by the control program, accesses only need a one-way communication between the set of producers to a given consumer for the pressure side or from the set of consumers to a given producer for the back-pressure side.

Another perspective on this issue is that the authors require an algorithm to rely on strongly non-commutative methods to obtain their result. A method is strongly non-commutative if its sequential execution by a thread can influence the result of the execution of another instance, of possibly the same method, by another thread, irrespectively of the order of execution of the two methods. Such is the case, for example, with an `add` or a `remove` method in a concurrent set implementation. Our algorithm contains no such methods. The `commit` and `release` primitives are exclusive writers of their respective locations in the map M, therefore precluding all interference, while `update` and `stall` only read from M.

Our original objective was to avoid expensive memory fences in the TSO model, which means that store-load fences should not be needed. Such fences represent the fundamental synchronization requirement in RAW patterns, as they force preceding stores to be ordered before subsequent loads. To avoid fences, our intuition was that a one-way communication scheme would allow to rely mostly on store-store and load-load ordering patterns. Our approach is validated by this result as our solution to avoid store-load fences has led us to avoid strongly non-commutative methods by relying on one-way communication. Indeed, each of our primitives either exclusively reads or exclusively writes to shared data structures, and the two primitives that write to shared memory use exclusive locations for each thread.

As for the AWAR pattern, we rely indeed on a write-after-read (WAR) pattern because of the load operations in `stall` followed by stores to the output stream synchronized by `stall` or equivalently in the case where the loads from an input stream must be ordered before the store operations in the subsequent `release` primitive. These cases, however, do not call for an *atomic* write-after-read as the write operations occur in memory locations guaranteed, by design, to be exclusively written by a single thread. As there is no race, the atomicity of the operation is unimportant: no external write operation can interleave between the thread's read and write operations.

6.2.4 Synchronization Algorithm for Irregular Tasks

In the case of regular tasks, the precise information on the stream access indexes required by future activations allows committing or releasing ahead of time all stream indexes that will not be used. This is necessary because a task may be assigned the production of index k in stream s, then be dormant for a long period of time. If all other producers to that stream make progress and the actual full prefix would allow consuming up to index $k' > k$, then the previous algorithm will fail and prevent all consumption past k. So if the task only commits the indexes actually produced, then it will hinder the progress of all other tasks connected to stream s.

However, this precise information is generally not available for irregular tasks, as their stream access patterns may not be decidable either statically or even dynamically, until the control program assigns their next access indexes. Our synchronization algorithm for irregular tasks must therefore be extended to allow disregarding the stream access indexes of tasks that lag behind because they are not activated.

In order to mitigate this issue, we rely once more on the information that can be obtained from the control program. While the future of that task remains undecidable, the control program can tell that no index in s has been scheduled to be produced by this streaming task up to at least index k', which should therefore allow consuming up to k' even though one of the producer tasks of s has not committed up to that point.

We present here the general case, where no information is available on the future of a task's stream access patterns. It is used for irregular tasks or when regularity is undecidable. The code generation required for the synchronization of stream communication, presented on Figure 6.8, is similar to that of regular tasks, except for the `commit` and `release` primitives where we conservatively only commit or release the index warranted by the original activation point, which is exactly *burst* elements from the lowest index accessed[5]. The additional *burst* information is considered available through outside means. We present the precise code generation in Chapter 7.

```
// Execute activation a:

∀s ∈ I(a) : update (s, max_(R,s,i)∈a (i));
∀s ∈ O(a) : stall (s, max_(W,s,i)∈a (i));

  execute_work_function (a);

∀s ∈ O(a) : commit (s, min_(W,s,i)∈a (i) + burst (a) - 1);
∀s ∈ I(a) : release (s, min_(R,s,i)∈a (i) + burst (a) - 1);
```

Figure 6.8: Synchronization code for stream communication of irregular tasks.

In addition to the map M that stores the latest, and by monotonicity also the highest, committed or released index of each thread, we need two additional data structures, also presented here as global maps for the sake of brevity. The first is the map $MaxSched(u, s)$ which stores the highest index scheduled, by the control program, for production or

[5]We recall that task activations always access intervals of indexes in streams to which they connect, so that the maximum and minimum expressions simply represent the bounds of the interval.

consumption in a stream. It is indexed by the stream access direction u and the stream identifier s. The second data structure is a map $MaxTaskSched(t, u, s)$ which stores the highest index scheduled for production or consumption, *by a given task* t, in a stream s. Both new data structures are exclusively written by the control program and only read by the concerned execution threads.

Before we expose the synchronization algorithm, we must stress the fact that this algorithm further requires that all task activations belonging to the same task be executed on the same thread. The generalization of this algorithm, allowing multiple threads of execution to concurrently execute multiple activations of the same task, is detailed in Section 6.2.6. For this reason, in the current context, a thread and a task are interchangeable ($t \equiv p$) and we allow indexing the map M by task identifiers instead of thread identifiers.

The synchronization algorithm for irregular tasks is presented on Figure 6.9. This synchronization algorithm relies on an *adjusted* computation of the minimum from the original algorithm. We focus on the algorithm for the `update` primitive, as the algorithm for `stall` is identical after a trivial substitution $W \to R$ and $producers \to consumers$.

This adjusted minimum computation uses the information provided by the control program on the maximum scheduled write access for stream s, $MaxSched(W, s)$, and the maximum scheduled write access by the task t for stream s, $MaxTaskSched(t, W, s)$. If a task, among the producers for stream s, has already successfully committed its highest scheduled write access (when $M(t, W, s) = MaxTaskSched(t, W, s)$), this means that it is not involved in writing to any stream index in the interval $[MaxTaskSched(t, W, s) + 1, MaxSched(W, s)]$. We can therefore replace its real highest committed index, $M(t, W, s)$, by its adjusted index $MaxSched(W, s)$.

The correctness of this algorithm on TSO architectures is rather complex to prove. We rely instead on the algorithm of Figure 6.10, where fences have been added to ensure proper execution on architectures allowing any kind of reordering of memory operations. As this algorithm, which we need to prove correct, contains no store-load fences, and the algorithm is identical when removing the fences, we deduce that the previous algorithm is valid on platforms providing the total store order memory model.

The proof of the algorithm on Figure 6.10 takes into account an indirect synchronization effect between the control program and all tasks. Since the control program stores the task activation after it updates the $MaxSched$ and $MaxTaskSched$ structures, and there is a store-store fence, the execution of the task activation necessarily starts after the updates relative to the activation's generation have been stored. As the activation further relies on either `update` or `stall` to synchronize on any streams, which contain either a load-load or a load-store fence that will also cover the activation's own load into memory for the thread to execute. The further reliance on `commit` or `release`, which also force the ordering of memory operations, finally ensure that the update of M occurs after that of $MaxSched$ and $MaxTaskSched$ with any possible information concerning that particular task activation. As all of these data structures only contain monotonically increasing values, we necessarily have $M(t, W, s) \leqslant MaxTaskSched(t, u, s)$ at any time. When they are equal, the load-load fence between the loads to $MaxSched$ and $MaxTaskSched$ in `update`, as well as the store-store fence between their updates by the control program, guarantee that the values read in $MaxTaskSched$ are more recent

```
// Evaluation of an activation point π by the control program:
{
  t = get_dynamic_task (static_task_id, π);
  for ((u, s, b, h) ∈ π) {
    start_index = MaxSched(u, s);
    MaxTaskSched(t, u, s) = start_index + b;
    MaxSched(u, s) = start_index + b;
    // ...
  }
  // rest of computation and memory store of the task activation.
}

update (s, index) {
  do {
    adjusted_min = ∞;

    for (t ∈ producers(s)) {
      task_index = M(t, W, s);
      adjusted_index = MaxSched(W, s);
      if ( task_index == MaxTaskSched(t, W, s) ) {
        task_index = adjusted_index;
      }
      if (adjusted_min > task_index) {
        adjusted_min = task_index;
      }
    }
  } while (adjusted_min < index);
}

stall (s, index) {
  // Similar to update, up to a substitution R/W ...
}

commit (s, index) {
  M(th_id, W, s) = index;
}

release (s, index) {
  M(th_id, R, s) = index;
}
```

Figure 6.9: Synchronization primitives for irregular tasks.

than those read in $MaxSched(u, s)$, which allows us to conclude that there can be no overshoot.

6.2.5 Optimized Cache Traffic Synchronization Algorithm

Our synchronization algorithm is based on spinning in the general case. As we consider that only one execution thread is created per hardware thread, this approach provides the best reactivity. Our experiments with other implementations, like futexes or POSIX condition variables, have proven unsatisfactory when no over-scheduling occurs.

However, spinning usually has a strong negative impact on cache traffic as each task checking for data on input streams, or checking for available space in output stream buffers, continuously queries the shared data structures for updates, continuously recomputing the minimum, thus generating a high volume of cache traffic. Our objective in this section is to minimize the amount of cache traffic generated by our algorithm. To achieve this, we use a secondary software cache to avoid requesting updates from the threads writing to the three shared data structures M, $MaxSched$ and $MaxTaskSched$.

Our algorithm relies on the idea that as long as the stream access index that is synchronized by `update` (resp. `stall`) is lower than a known value of the highest committed (resp. released) index of a given thread, then that known value already allows the `update` operation to succeed. A new, and by monotonicity greater, value would not change the result.

We add two new data structures, both corresponding to data entirely private to a thread, represented once more as maps. The first map $MCache(p, p', u, s)$ simply stores the last values loaded by thread p from the original, shared, map $M(p', u, s)$. As the values stored in M are monotonically increasing, we will have at all times $MCache(p, p', u, s) \leqslant M(p', u, s)$. The second map $MinCache(p, u, s)$ stores the latest minimum computed by thread p for stream s and direction u. We stress the fact that these two data structures are private to threads, therefore requiring no special care for memory ordering concerns. We do not present the memory fences that would be requires on very weak memory models as they are similar to the original algorithms.

We present the algorithms for synchronization of both regular and irregular tasks, but we only show the case of the `update` primitive. The `stall` primitive is handled in an identical manner, and the `commit` and `release` primitives require no modifications whatsoever. In the case of irregular tasks, the control program side remains unchanged as well.

Figure 6.11 presents the cache-optimized version of the synchronization algorithm (see Figure 6.5) for regular tasks. This algorithm minimizes the number of accesses to the shared data structure M, relying instead on the local data stored in $MinCache$ and $MCache$ whenever possible. The algorithm first checks whether any previous local evaluation of the minimum returned a value that is already satisfactory for the index currently synchronized. If indeed $MinCache$ contains a value higher than `index`, then there is no reason to recompute as the synchronization requirements are already met.

The second level of caching, in $MCache$, allows the simple computation of the minimum to use a possibly old value if this old value is high enough. Indeed as the data stored in M is monotonically increasing, older values can only be lower, therefore not impacting the validity of the computation. If for some thread the cached value is lower than required by the current synchronization, then a load is performed on the shared data structure and the cache is updated.

As a side note, we could also consider caching the set of producers that have already been verified to have committed beyond the synchronized index. This would allow to re-compute the minimum only on a subset of the producers, but in practice this set is too small to be of interest. Furthermore, all threads that would not be checked have values necessarily satisfactory in the local cache, so this would not save loads from the shared data structure.

The case of irregular tasks, presented on Figure 6.12, is very similar to the previous algorithm. The main difference is that the values stored in the $MCache$ data structure are not necessarily the original values read from M, but possibly the adjusted values read from $MaxSched$. This is inherently safe once again because of the monotonicity of updates in these data structures. Once a given index is deemed committed or released by a task, it is always safe to access lower indexes, at least with respect to that task. This behaviour ensures that data from tasks that were previously adjusted do not always generate cache traffic.

6.2.6 Synchronizing Data-parallel Tasks

As we have previously mentioned, stateless streaming tasks are naturally data-parallel, in the sense that there can be no dependences between different activations of the same task, with some simple exceptions. The first exception concerns statefulness, which we represent as a self-cycle where the task produces and consumes in the same stream, and more generally any task belonging to a strongly connected component of the task graph. In this case, task activations can be executed concurrently, but the implicit synchronization of flow dependences in streams will serialize their execution. The second exception comes from the possible reliance of OpenMP tasks on shared memory communication through the shared clause. However, the semantics of OpenMP tasks ensure that the programmer is fully responsible for any synchronization required by this communication, considering that no order of execution can be assumed for any task activation.

One of the most important optimizations of stream programs consists in executing task activations of computationally intensive tasks concurrently, to improve the load balance of the task graph. Indeed, the overall throughput of a pipeline is limited by that of the slowest filter. However, the concurrent execution of task activations means an additional level of concurrency for the stream synchronization algorithm. Until now, we have accounted for the concurrent execution of synchronization primitives on the same stream from a group of producer tasks and a group of consumer tasks. We add the possibility that any such task may actually correspond to multiple threads of execution.

In the case of regular tasks, our synchronization algorithm does not require any modifications as it is essentially independent of task semantics. For irregular tasks, however, the notion of task was central to adjust the computation of the minimum and we explicitly restricted the algorithm on Figure 6.9, as well as its cache-optimized version on Figure 6.12, to single threaded execution of individual streaming tasks. Because of the importance of exploiting data-parallelism in irregular tasks as well, we need to adapt this algorithm.

We identify two solutions to this problem. The first is essentially a static approach involving the control program, while the second is fully dynamic, but requires atomic

operations. In both solutions, the central point will be the way task activations are dispatched to one thread or another among the team of threads that executes a given task.

The first solution, presented on Figure 6.13, is almost identical to the original algorithm on Figure 6.12. References to tasks t are replaced with threads p in the `update` primitive, but the main difference lies in the fact that the control program has additional work, as shown by the call to `decide_thread_placement`. This method simply implements a dispatch strategy, for the task activation being generated, among the team of threads executing the task t. Such a dispatch strategy can be either very inexpensive in the case of static patterns, like a round-robin dispatch, but this may result in very poor load-balance among the team of threads. A more expensive strategy would consist in checking the progress status of each thread in the team before assigning the new activation. This solution can also be implemented through an off-load to a helper thread, that still serially dispatches the work. Because of the order enforced within the sequential dispatcher, and the fact that the shared memory communication pattern is still one-way, this algorithm is still devoid of memory fences.

To make things clearer, we could rename the $MaxTaskSched$ map to $MaxThreadSched$, but as the data structure preserves the same semantics in all other respects, we keep the old name.

The second solution requires the use of a concurrent multi-consumer FIFO scheduler queue, where the control program enqueues the new task activations and each thread of the team *atomically* acquires work *and* updates its own $MaxTaskSched$ entries for all task activations it acquires. We do not present the code for this case, as it is easy to derive from the previous algorithm.

The choice between the two solutions is very dependent on the application, in particular on the load of the control program, the load regularity of the task activations within the same task and the amount of concurrency that can be exploited compared to the number of available hardware threads. We do not propose a general answer, but we note that both regular and irregular tasks have the same fundamental problem with respect to the distribution of task activations to the threads of the team. Indeed, if the control program, or equivalently a dispatcher thread, is not involved in sequentially and unilaterally deciding on the dispatch strategy, then a consensus must be reached within the team on the attribution of work. This cannot be done without either of the aforementioned RAW or AWAR patterns.

The scheduling of task activations is presented in Section 6.3.2 where we further discuss this issue.

6.2.7 Evaluation of Stream Synchronization vs. Scheduling Overhead

To conclude this section on stream communication, we present an experiment to evaluate the difference in overhead between our optimized synchronization algorithm and the scheduling of lightweight tasks. We use a synthetic benchmark, called `exploration`, conceptually consisting of a sequential producer task generating values (e.g., from exploring a tree) and a consumer task performing a simple, inexpensive, operation on the

values. In the streaming version, the producer writes values in a stream, by groups of `burst` values at a time, and the consumer is a single task reading from that stream and synchronizing with the producer for every block of `burst` values. In the Cilk version the producer spawns a new task to process each block of `burst` values. The `burst` parameter allows to study the parallelization overhead as a function of the synchronization grain.

To ensure that the implementation of the lightweight task scheme is not at fault, we compare with a Cilk [24] implementation rather than with OpenMP tasks, which only provide rudimentary scheduling in the GCC implementation. We call our streaming version *persistent tasks* to contrast with the Cilk version, where tasks are short-lived. As our streaming tasks are seen as equivalence classes on their input and output streams, there are only two streaming tasks in this benchmark, and they are persistent throughout the execution of the program.

The results, obtained on an Intel Core2 Quad Q9550 with 4 cores at 2.83GHz, are presented on Figure 6.14. While the benefits of scheduling lightweight tasks for load-balancing are undeniable, the higher overhead of scheduling requires a significantly higher task granularity to amortize. In order to evaluate the granularity required to break even between persistent and short-lived tasks, we compare, on Figure 6.14, the execution time on the `exploration` synthetic benchmark. On one side we use persistent tasks, while on the other we have a Cilk implementation spawning short-lived user-level tasks [24]. Cilk is run with the `--nproc 4` option to generate parallel code, and with the `--nproc 1` option to specialize the code for sequential execution. The sequential Cilk version takes almost $7\,s$ for the finest synchronization grain, and $5\,s$ for larger ones. The parallel Cilk version with the finest synchronization takes $221.4\,s$ and the corresponding persistent task version takes $107.7\,s$. The performance gap widens significantly for bursts of intermediate size, and approaches $5\times$ when the persistent task version reaches its performance plateau. The most important figure, in practice, is that the persistent tasks break even for grain size $80\times$ smaller than Cilk. This demonstrates the need for data-flow interactions among long-lived, persistent tasks as an essential implementation mechanisms for scalable concurrency.

6.3 Activation Points and Task Activations

The focus of this section is the runtime support required for the evaluation of activation points, which naturally leads to the emergence of the dynamic task graph, and the way the execution of the resulting task activations is orchestrated. We discuss the synchronization requirements and show that it is often possible to avoid atomic operations and memory fences altogether. Exceptional events such as task creation, program initialization and termination or explicit barrier synchronization are clear exceptions. We further discuss the scheduling techniques for task activations and present our current approach to load balancing, which is still an open issue, with some improvements proposed for future work.

6.3.1 Evaluation of Activation Points

The control program is responsible for generating task activations from activation points, which we modeled as the ξ function in the CDDF model. The evaluation of activation points distinguishes between regular and irregular tasks, as defined in Section 5.3.1.

Regular tasks can benefit from a simplified generation algorithm. It is reduced to the incrementation of a counter and all stream access indexes have a closed-form representation as a function of this counter.

The activation point evaluation algorithm for regular tasks is presented on Figure 6.15. It relies on a dynamic lookup function `get_dynamic_task`, which identifies a streaming task based on the `static_task_id` and the set of input and output streams used by the task to communicate. The static task identifier can be understood as a pointer to the work function outlined from the body of the task. As such, this identifier is a purely syntactical item, linked to the OpenMP task construct. The set of input and output streams is obtained from the activation point, which we recall is a set of stream access descriptors in the form $(u, s, b, h) \in \{R, W\} \times \mathcal{S} \times \mathbb{N} \times \mathbb{N}$. The only information required from the control program is the number of times such a task activates, which defines the dynamic iteration set of the task, in the interval $[0, t.activation\_counter - 1]$. We introduce here the first internal field of the data structure representing streaming tasks in the runtime, which is the counter of activations. This data item is exclusively written by the control program, and therefore benefitting from the same memory fence free communication pattern.

The parallelization of the control program would require making the incrementation operation atomic, depending on the partition of the control program. If the generation of task activations of the same task is devolved to a single thread of execution, this operation can be non-atomic. This incrementation operation behaves in the same way as adding a task activation on a scheduler queue, except that the implicit knowledge of the task activation from the context of the task t allows to compress this implicit if-conversion.

Remark 6.2. *This implementation hides two important points. Firstly, the burst and delay constants, $B_{\langle t,u,s\rangle}$ and $D_{\langle t,u,s\rangle}$, that determine the closed-form expression for computing the stream access indexes, are not always contained in the activation point. When multiple tasks regularly interleave their accesses in a given stream, the perceived burst[6] is the sum of the actual bursts of each such task. Furthermore, the delay is strongly linked to the control flow context and to the possible presence of explicit delay patterns. This information is generated, possibly in the form of a parametric expression, by the compiler. It is made available to the runtime from the static task identifier.*

Secondly, the `get_dynamic_task` function hides the creation of new dynamic tasks when it is called for the first time with a given static identifier and a given set of input and output streams, contained in the parameter π. This operation then requires a consensus among all producers and consumers of all streams in π. This is an exceptional case; in the general case, the task has already had previous task activations and no synchronization is required. We present the corresponding algorithm in Section 6.4, on Figure 6.27, as it relies on the notion of stream level quiescence, itself only introduced in that section.

Figure 6.16 presents the algorithm evaluating the activation points of irregular tasks. This algorithm completes the partial view of the control program side of stream communication synchronization from Figure 6.9, so we reuse the same data structures. The main difference is that, because of the impossibility of relying on closed-form expressions for computing stream access indexes based solely on the activation counter, we need to

[6]Which we recall is the notion of shift in the stream before the next iteration can access the stream.

create a control stream, linked to the task, to provide the task activation with the start position of its stream accesses. For this, we build a data structure to hold, for each couple (u, s) of a stream and the direction of accesses, the start index of allowed accesses and the horizon h which gives the size of the access interval. This data structure is written to the control stream at the position corresponding to the current task activation in the task's iteration set. Note that this stream is itself always *regular* as it has a single producer, a single consumer and its burst is always unitary.

As a final note, the dynamic task graph is not purposefully built, but rather emerges from the interconnection of dynamic tasks through streams. These interconnections are registered dynamically whenever a new task is created. The abstract functions that we use in the stream communication synchronization algorithms, like `producers(s)` which returns the set of producers of stream s, can be used to traverse this task graph.

6.3.2 Scheduling of Task Activations

The scheduling of task activations considers that task causality is always respected, which means that there are no dependences to enforce from latter activations of a task to earlier activations of the same task. This ensures the validity of our most basic form of scheduling, which is a sequential schedule, in control program order, of all activations of a given task.

The scheduler is presented as a set of simple scheduling functions, that return either one point or a range of points in the iteration space of the task, which is a flattened space that hides the original control flow. This function is used in a form similar to that presented in Figure 6.17, where the `get_activation(t)` represents the scheduling function.

Note that it is possible to entirely execute a CDDF task graph as a pure runtime solution. We call it a task graph interpreter. The code generation effort is limited to the bare minimum: outlining work functions and issuing calls to the evaluation of activation points. However, this solution is rather heavy and misses many compiler optimization opportunities. We will rely on this technique, in future work, to develop runtime task-level optimizations like task fusion.

Sequential scheduler: a persistent regular process view

The basic sequential scheduler, presented on Figure 6.18, only consists in returning the next unassigned activation index, with an incrementation of the counter of assigned activations. The function waits until the activation counter, of the task from which an activation is requested, is greater than the highest assigned activation index. During the wait, it queries a termination flag associated with the task data structure to ensure that other activations are still possible. Of course this test is inexpensive as the variable is set only once at program termination.

Once termination is signaled, and it is understood that this may only be seen once the latest value of the activation counter is seen as well[7], a last check is required to ensure that no work is available. If this check fails, in the sense that there is no more work, the task can execute its termination sequence.

[7]This does trivially not require memory fences in the TSO model, for the same reasons as previously exposed, but it cannot avoid a fence in weaker memory models.

Enabling work aggregation

Aggregation is one of the most important performance optimizations, simultaneously reducing runtime overhead, through a reduced number of runtime calls, often nullifying false cache sharing, more generally reducing cache misses and enabling compiler optimizations, and notably, vectorization. This will be discussed more in detail in the code generation sections of Chapter 7.

Aggregation comes in two flavours. The aggregation of work consists in acquiring multiple units of work at a time, in our case task activations, and the aggregation of data consists in communicating bundles of data elements at a time, which can be for example aligned on cache line boundaries to avoid false sharing. The aggregation of data is entirely a code generation issue as our stream communication primitives already allow arbitrary data aggregation patterns.

We are interested here in the aggregation of work and hence of task activations, which is achieved with a scheduler function that returns more than one task activation at each call. Figure 6.19 presents a simple solution for such a scheduler in our runtime. It takes as parameter not only a task identifier, but also a limit on the amount of work it can return at a time. This feature is essentially necessary for fairness issues and will be discussed in the next paragraph.

This algorithm is almost identical to the previous one, but it returns a range of task activation indexes that can be executed.

Concurrent scheduling

The two schedulers presented above are clearly not meant for concurrent use, meaning that they are only safe to use for tasks where all activations are executed on a single thread. If we want to also exploit data-parallelism within a task, we need to make those algorithms thread safe.

Concurrently dispatching task activations requires either memory fences or an atomic operation. Indeed, this problem does fall in the scope of the result by Attiya et al. [8], which is discussed above, in Section 6.2. We use an atomic operation for simplicity and we only present the first algorithm for brevity. The second scheduler can be handled identically.

The concurrent algorithm is presented on Figure 6.20. The only difference is the use of the atomic operation for the incrementation of the assigned activation counter. Note that the task termination function requires further synchronization, in the form of a critical section, to reach consensus and ensure that de-allocation is performed by the last exiting thread.

Aside from the necessity of atomic operations, the main issue of concurrent schedulers is that work aggregation can lead to load imbalance by way of unfair acquisition of work. Indeed, if there are only a limited number of activations available and the first worker thread acquires all of them, then all other worker threads will be left waiting. This issue is generally solved through work-stealing, which consists in allowing work-less threads to take units of work not only from the scheduler, but also from other worker threads that have acquired task activations on which they are not currently working. This strategy is relatively simple to implement naively, but more advanced techniques must include

strategies to avoid excessive stealing events. We choose not to implement such techniques at this stage for these three main reasons.

1. Work-stealing requires consensus to avoid conflict between two threads that decide they can both start executing a task activation. This consensus requires each thread to re-check *atomically* that it is entitled to execute an activation before starting. This is equivalent in the number of synchronization, and therefore overall cost, to simply not aggregating at all and just atomically acquiring one unit of work at a time. This nullifies all benefits of our scheme, where the scarcity of memory fences and atomic operations is critical for performance.

2. We rely, in order to efficiently synchronize stream communication, on the monotonicity of stream access operations. Without monotonicity, our scheme would be trivially incorrect. If work-stealing were to be implemented, each work-stealing attempt would need to also ensure that the task activation being stolen by a thread is of a higher activation index (i.e., that it was generated later in sequential control program order) than the highest task activation it has yet executed. As task activations are not tagged in our system, and this allows to much more efficiently handle synchronization as there are implicitly no *holes* in the sequence of task activations. Work-stealing would come with expensive modifications to the current scheme.

3. Finally, the general case of streaming applications corresponds to regular tasks that iterate on large amounts of data, therefore on large iteration spaces. The particular case where some tasks only have a low number of task activations which can be acquired by a single thread at once does not appear, at the current time, as a sufficient motivation to incur the aforementioned overheads.

Our solution instead lies in the heuristic choice of a `max_aggregation_factor`, which is a parameter of the aggregating scheduler on Figure 6.19. This limits the number of task activations a worker thread can acquire at once and it appears to be a sufficient solution in our current applications. Finally, we note that this issue does not impact correctness. Lack of fairness may reduce performance, but it should not have a strong impact as tasks with few activations generally do not represent a high overall workload.

6.3.3 Load Balancing Through Dynamic Data-Parallelization

Load imbalance is inherent to most pipeline parallelization schemes in the sense that load balance can only be achieved if each stage of that pipeline has the same computational load and the same arithmetic intensity. Contrary to data-parallelization, where generally a loop's iteration space is distributed across multiple threads of execution and therefore each thread executes the same instructions on different data, each pipeline stage usually consists of a different work function. If pipeline stages are executed sequentially, then the highest possible speedup is obtained through Amdahl's law applied to the proportion of work found in the stage with the heaviest computational load.

One of the possible solutions is to exploit data-parallelism within pipeline stages, which also allows to increase the amount of parallelism exploited when the pipeline depth

is insufficient. As we have discussed, all stateless filters are inherently data-parallel and their iterations can be executed concurrently on multiple threads. In a certain sense, the analysis required to ensure the validity of data-parallelization is performed statically and completed, or enforced in the case of tasks belonging to strongly connected components of the task graph, dynamically.

To dynamically exploit data-parallelism in tasks, we need a scheme to add worker threads to help execute the filter representing a bottleneck in the pipeline. Even more importantly, we need a decision-making tool, that can dynamically choose in which tasks to exploit data-parallelism, without requiring global consensus, yet allowing to balance the execution of the task graph. We first present the simple runtime framework required to allow the dynamic addition of worker threads for a task, then we show that we can exploit the stream communication synchronization information to monitor the *relative* load balance of a task in a pipeline, with almost no overhead. Finally, we discuss the decision making heuristics by analyzing the case of lad-balancing the execution of the FMradio application and we propose guidelines for future developments.

Dynamically enabling the concurrent execution of tasks

In order to enable a dynamic switch from single threaded execution of a task to concurrent execution of its task activations, it is first necessary either to generate code that uses, by default, the stream communication synchronization and the scheduler functions that support concurrent execution, or to provide a way to dynamically switch between the sequential to the concurrent versions of these functions. However, dynamically switching requires that such functions be called through an indirection and also an atomic operation at each call to ensure that the proper version is used. There is little benefit overall, so we generate code that uses the concurrent versions from the start for all tasks that are susceptible of being data-parallelized. Of course all stateful tasks, or those in strongly connected components, would use the sequential versions as they cannot benefit from data-parallelism in general.

Adding new worker threads to help execute a task relies on the same mechanism as the creation of a new task. If the task to which this thread is added already relies on atomic task activation acquisition, its addition is inherently safe and does not require further attention.

Monitoring relative load imbalance in a pipeline

Monitoring load balance often rely on hardware counters or other means of evaluating the amount of work performed during a certain amount of time, which is then compared to the same information gathered in the rest of the program. This is more or less efficient, depending on the way it can be turned on and off, as well as on the difficulty of aggregating the information to make a decision. In our case, however, the pressure and back-pressure algorithms can provide us with a similar type of information for free. Indeed, the knowledge that input or output streams are either full or empty is enough. It provides the relative load (im)balance between a task and its neighbours in the pipeline and, as we discuss below, this transitively extends to the whole connected component of the taskgraph in which the task belongs.

This is simple to understand with the example presented on Figure 6.21, where a heavy, slower, filter creates congestion upstream and starvation downstream. In both cases, the heavier filter determines the throughput of lighter filters, by either forcing them to stall either for free space in an output stream buffer or for data in an input stream buffer. This kind of information is naturally local to some part of the task graph, though some influence can be propagated even from more remote parts, which is why we talk about relative imbalance. Our main objective is to define a test to decide in a purely local manner, at the task level, whether a task should be data-parallelized.

We present the case of obtaining such information for input streams, the handling of output streams being symmetrical. In order to know whether an input stream is mostly full or not, we can rely on the pressure based algorithm presented on Figure 6.22. We modify the algorithm from Figure 6.13 to test whether a call had to wait for input data or not. We simply accumulate the information, by incrementing a counter on stalling events and decrementing it on non-stalling invocations, in a data structure called $RelBalance(t, u, s)$. It stores, for each task, the aggregated history of the behaviour of a given input or output stream. Note that the data structure is shared by all threads executing a given task, but we do not synchronize the concurrent accesses as lost incrementations or decrementations have little impact on the overall trends.

This score-keeping is compared to a threshold value, which indicates whether the stream is mostly full or mostly empty. This knowledge is then used to locally decide whether a given task is rather heavier or rather lighter than its neighbours in the program task graph. The possible cases are presented on Table 6.2. Note that our monitoring scheme is only capable of precisely determining negative patterns, where input streams are consistently empty or output streams are consistently full, while the cases of full input buffers or empty output buffers would cost additional operations to determine. However, this is of little importance as we have precise information on what really matters to us. Indeed, if a single input stream is empty, then there is no point in improving the throughput of a task, and conversely if a single output stream is full, no throughput improvement can come from this task alone.

We therefore consider, as is apparent on Table 6.2, that it is sufficient for a single input stream to be consistently empty or a single output stream to be full in order to reject a task as a candidate for data-parallelization.

Decision heuristics: case study of FMradio

In order to give a better view of the benefits of such a scheme, and to discuss some concerns with respect to algorithm stabilization and convergence towards a well-balanced execution, let us consider the case of the software defined radio application FMradio. A streaming OpenMP annotated kernel is given on Figure 8.1, but we are only interested here in the application's task graph and its balance issues. We present, on Figure 6.23, the task graph of FMradio with a rough approximation of the overall computational load represented by each filter obtained by profiling. We represent the team of threads that execute the four heaviest tasks, with one thread each at the beginning of the program.

The first thing of importance on this task graph is the load imbalance. Indeed, executing this program with only pipeline parallelism cannot yield large speedups as Amdahl's law gives an upper bound of roughly 3.3× speedup because of the two heavy

Detected state	Relative workload	Interpretation
	Indeterminate	Insufficient upstream production Downstream irrelevant: blocked by empty input stream ⇒ Reject
	Low	Insufficient upstream production Insufficient downstream consumption ⇒ Reject
	High	Upstream production exceeds processing capabilities Downstream consumption exceeds processing capabilities ⇒ Candidate for data-parallelization
	Indeterminate	Upstream irrelevant: task blocked by full output stream Insufficient downstream consumption ⇒ Reject

Table 6.2: Interpretation of pressure and back-pressure information to deduce relative imbalance.

filters, each representing 30% of the computation. This is apparent in our performance results, reaching a maximum speedup of 3×, on all configurations from four hardware threads up. In order to gain load balance, and of course reduce the bottleneck effect of such heavy filters, our only option[8] is to exploit data-parallelism.

Let us now consider what happens in such a task graph once execution starts. Figure 6.24 shows the evolution of stream states with respect to pressure and back-pressure when we dynamically adjust the load balance through our scheme, adding new worker threads to the teams of all tasks that fit the condition on the input and output stream states. Let us consider this evolution step by step, as shown on Figure 6.24.

(a) After the program starts executing, the source filter `fm_quad_demod` is able to produce data much faster than any of its consumers. Once the stream buffers are full, it cannot make further progress and its throughput is naturally limited to the throughput of its slowest consumer. On Figure 6.24, we represent the state of each stream through the value of the relative balance monitoring variable, where red corresponds to full streams which stall for output operations and violet marks empty streams that often stall all consumers trying to read from them. So once the source filter is forced to produce at the pace of the slowest consumer, the data propagates through the task graph faster on all other paths, but it remains stuck at all confluence points between a fast and a slow path. There are four blocking streams in the first equilibrium state, in front of the slowest tasks and in front of confluence tasks that are blocked by absence of data from the slow paths.

In this configuration, once the threshold is reached, the two heaviest tasks make the decision to increase the size of their worker team as they fit our condition. Note that no other task in the graph does meet the condition. For all tasks that have teams represented, we provide the task's share of the total workload on top and a worker thread's share below.

(b) Once the two heaviest tasks added one worker thread each, all worker threads in the

[8] Of course, provided that such filters are stateless. Otherwise, the program cannot scale any further.

four heavy tasks have a workload share of 15%. For this reason, the new equilibrium is reached with each of their input threads full, and only another stream is full at a confluence point with a faster path. As the confluence point also has an empty input buffer, it is not a candidate for adding a thread to its team, but all four heavy `ntaps_filter_ffd` tasks are candidates for increasing their respective teams.

(c) Each of the four tasks has added a thread, and we get a situation similar to the first equilibrium, in state (a). Indeed, the two leftmost heavy filters have a thread-level load once again higher than all other threads in the program, at 10% of overall load, so the streams will present the same configuration. These tasks can add threads to their work teams. Note that this time the imbalance relative to their siblings, which have a load of 7.5%, is lower, so the dynamic equilibrium will take longer to reach.

(d) We have reached again the same state as in (b), with however a much lower difference between the heaviest and lightest thread loads. If the algorithm is allowed to proceed, it will reach a point where all threads have a load comprised between 1% and 2% in just another eight transitions. If for example the execution platform only disposes of 20 hardware threads, then this state is a good point for stopping the load-balancing process.

In order to obtain such a result, two important things are needed. Firstly, the threshold must be chosen large enough for the program to reach some form of dynamic equilibrium before any decision is made. Of course, we represent the decisions happening simultaneously, but this would not be the case in an actual execution. If one task makes the decision early, it does not impact the behaviour of the algorithm as the tasks making the decision later perceive an even greater relative load imbalance. In order to avoid hasty decisions, in particular as the program takes longer to stabilize once the sizes of worker teams grow and the difference between the highest and lowest loads decreases, the threshold should also increase, for example with some form of slow geometric progression, after each decision.

Secondly, this algorithm cannot converge. Unless a perfect load balance can be achieved, and the execution load is constant, the algorithm continuously refines the load balance with an implicit exploration of a sequence of dynamic equilibrium states. While the decision is made locally at a task level, the monitoring actually aggregates, through the flow of data in the task graph, relative pressure and back-pressure information for the whole task graph. In a certain sense, the flow of data in the task graph, or the lack thereof, that we monitor in our scheme, represents a perfect exploration mechanism for finding bottlenecks and the information is readily available to the concerned party, the critical task. To stop this refinement process, we need to rely on a global (or local to a node in more distributed systems) shared resource that is requested each time a new thread is created. This resource can simply be the identifier of a hardware thread that is still available, or a counter of such threads. Once no more resources are available, the refinement process stops, like in our example for a total number of 20 threads.

Future work

The current scheme is only designed to work in one way, which is sufficient for programs that have the same behaviour throughout their entire execution. For applications that have varying workloads or different computational phases, it is necessary to also be able to reduce teams of worker threads that become too large for a new, reduced load. This is quite simple to handle with a very similar approach where the full and empty stream buffers are reversed, but the difficulty is in finding a way of enabling team reduction without starting oscillatory phenomenons. We leave the handling of such cases for future work.

Another possible extension of this thesis concerns dynamic task fusion, which can be thought of, in the case where a more integrated fusion is impossible, as simply scheduling two tasks with low loads on the same thread. This does constitute an important challenge, to properly orchestrate their execution to avoid breaking ordering constraints and avoiding deadlocks.

6.4 Resource Deadlock Detection and Stream Memory Management

In the generalized CDDF model, it is not always possible to prove that no deadlocks can occur, nor that the amount of memory required for the execution can fit within the hardware limits. To mitigate these issues, and also provide enhanced debugging support for functional deadlocks, we introduced a runtime deadlock detection algorithm that is mostly meant for the correctness issue of resource deadlocks, but can also detect functional deadlocks. This algorithm uses a dynamic stream buffer resizing scheme to resolve resource deadlocks, when memory is available.

These techniques are not applied when it is possible to statically guarantee the good behaviour of the streaming program, but, when they are required, the design ensures minimal interference with the normal execution of the program. They are only activated when a task's execution is unable to proceed because of ordering constraints. Furthermore, our objective of avoiding all atomic operations or memory fences, assuming a TSO memory model, is preserved for the most frequent cases. Atomic operations are required to ensure correctness exclusively in the exceptional cases of resolving a resource deadlock or any operation that changes the structure of the program, like the dynamic creation of a new task, the addition of new worker threads to a task's execution team or resizing the buffer of a stream.

We first present the algorithm required to ensure the quiescence of all tasks connected to a given stream, with a slight adjustment to the implementation of the `stall` and `update` stream synchronization primitives, and we provide the implementation of the dynamic task creation algorithm which requires the stream quiescence algorithm, as mentioned in Remark 6.2. Then we show how stream buffers can be dynamically resized, and finally we discuss some implementation considerations for the resource deadlock detection and resolution algorithm, presented above on Figure 4.4.

6.4.1 Stream Level Quiescence

Stream level quiescence is a state where all threads executing tasks connected to a given stream have reached a consensus and stopped all operations. This consensus allows to make modifications to the local structure of the task graph without impacting the normal operation of communication. In order to achieve this, we rely on unsynchronized one-way notification of threads of a quiescence request, which is then acknowledged atomically by each thread. The performance and convergence delay aspects of quiescence are considered of lesser importance than the avoidance of atomic operations and memory fences on the common path of execution.

The algorithms are presented on Figures 6.25 and 6.26, with the algorithm for the thread requesting quiescence on the former and the adjustment to the `update`, and by symmetry to the `stall`, primitives on the latter.

The algorithm for requesting stream level quiescence is presented on Figure 6.25. The requesting thread first acquires a lock on the stream for which it requests quiescence. This step is necessary as multiple threads may attempt to modify the state of this stream (e.g., replace its buffer or add new threads as producers or consumers) and these modifications need to occur atomically. During the acquisition step, the thread cannot just wait as it could block another thread requesting quiescence from the current thread for another stream, which could lead to a deadlock. For this reason, each thread waiting needs to regularly check whether new quiescence requests have been posted in its queue. We rely on a queue of quiescence requests because it is necessary to know which threads have requested one's quiescence in order to properly acknowledge the event to the requester.

Once a thread has acquired the quiescence lock on a stream, it can issue quiescence requests to all threads connected to that stream. Note that the set of such threads cannot evolve as this would require another thread owning the quiescence lock. After issuing the requests, the thread waits until each request is acknowledged, while still responding to any incoming requests. The thread keeps a count of the number of requests it has received itself to know whether it needs to wait at the end of the quiescence phase.

After the thread achieves quiescence and performs the operations it intended on the stream, it calls the `resume_after_quiescence` function. This function posts a quiescence release to each thread from which it requested quiescence, then releases the quiescence lock on the stream. Finally, it ensures that it properly waits the release of each request it received since the beginning of the quiescence phase. On exit, it is possible that it may not have *seen* any last minute request, but as it is no longer blocked, the thread will see any such request during a `stall` or `update` operation.

Figure 6.26 presents the necessary updates for the stream communication synchronization primitives `update` and `stall`. As in Section 6.2, only the `update` primitive is presented, as the `stall` primitive is modified in the same way. We add a check for incoming quiescence requests. It is performed exclusively when threads fail to acquire the data from ingoing streams, or the space in outgoing stream buffers, they requested. If a request for quiescence has been made, the thread acknowledges it and recursively descends into quiescence for any new request. Once it receives a release notification for every acknowledged request, it exits quiescence. An important step after exiting quiescence is to ensure that all pointers the thread may have acquired in all streams are up-to-date. If the quiescence was used to replace a stream's buffer, some of the accesses obtained with

```
// Evaluation of an activation point π by the control program:
{
  t = get_dynamic_task (static_task_id, π);
  for ((u, s, b, h) ∈ π) {
    start_index = MaxSched(u, s);
    MaxTaskSched(t, u, s) = start_index + b;
    store_store_fence ();
    MaxSched(u, s) = start_index + b;
    // ...
  }
  store_store_fence ();
  // rest of computation and memory store of the task activation.
}

update (s, index) {
  do {
    adjusted_min = ∞;

    for (t ∈ producers(s)) {
      task_index = M(t, W, s);
      adjusted_index = MaxSched(u, s);
      load_load_fence ();
      if ( task_index == MaxTaskSched(t, u, s) ) {
        task_index = adjusted_index;
      }
      if (adjusted_min > task_index) {
        adjusted_min = task_index;
      }
    }
  } while (adjusted_min < index);
  load_load_fence ();
}

stall (s, index) {
  // Identical to update, with a trivial substitution R/W ...
  load_store_fence ();
}

commit (s, index) {
  store_store_fence ();
  M(th_id, W, s) = index;
}

release (s, index) {
  load_store_fence ();
  M(th_id, R, s) = index;
}
```

Figure 6.10: Maximum required memory fences for irregular task synchronization.

```
update (s, index) {
  if (MinCache(th_id, W, s) > index)
    return;

  do {
    current_min = ∞;
    for (p ∈ producers(s)) {
      task_index = MCache(th_id, p, W, s);

      if (task_index < index) {
        task_index = M(p, W, s);
        MCache(th_id, p, W, s) = task_index;
      }
      if (current_min > task_index) {
        current_min = task_index;
      }
    }
  } while (current_min < index);

  MinCache(th_id, W, s) = current_min;
}
```

Figure 6.11: Cache-optimized synchronization of regular tasks.

```
update (s, index) {
  if (MinCache(th_id, W, s) > index)
    return;

  do {
    adjusted_min = ∞;

    for (t ∈ producers(s)) {
      task_index = MCache(th_id, t, W, s);

      if (task_index < index) {
        task_index = M(t, W, s);
        adjusted_index = MaxSched(W, s);
        if ( task_index == MaxTaskSched(t, W, s) ) {
          task_index = adjusted_index;
        }
        MCache(th_id, t, W, s) = task_index;
      }
      if (adjusted_min > task_index) {
        adjusted_min = task_index;
      }
    }
  } while (adjusted_min < index);

  MinCache(th_id, W, s) = adjusted_min;
}
```

Figure 6.12: Cache-optimized synchronization of irregular tasks.

```
// Evaluation of an activation point π by the control program:
{
  t = get_dynamic_task (static_task_id, π);
  p = decide_thread_placement (t);
  for ((u, s, b, h) ∈ π) {
    start_index = MaxSched(u, s);
    MaxTaskSched(p, u, s) = start_index + b;
    MaxSched(u, s) = start_index + b;
    // ...
  }
  // rest of computation and memory store of the task activation.
}

update (s, index) {
  if (MinCache(th_id, W, s) > index)
    return;

  do {
    adjusted_min = ∞;

    for (p ∈ producers(s)) {
      task_index = MCache(th_id, p, W, s);

      if (task_index < index) {
        task_index = M(p, W, s);
        adjusted_index = MaxSched(W, s);
        if ( task_index == MaxTaskSched(p, W, s) ) {
          task_index = adjusted_index;
        }
        MCache(th_id, p, W, s) = task_index;
      }
      if (adjusted_min > task_index) {
        adjusted_min = task_index;
      }
    }
  } while (adjusted_min < index);

  MinCache(th_id, W, s) = adjusted_min;
}
```

Figure 6.13: Synchronization algorithm enabling data-parallelization of irregular tasks with control program dispatch.

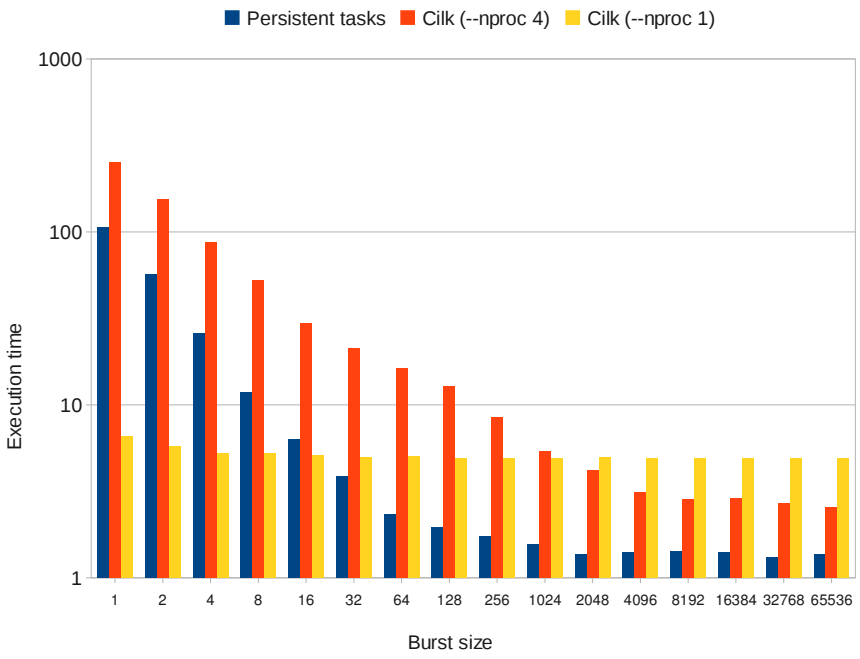


Figure 6.14: Exploration: persistent vs. short-lived tasks.

```
evaluate_regular_activation_point (static_task_id, π)
{
  t = get_dynamic_task (static_task_id, π);
  t.activation_counter += 1;
}
```

Figure 6.15: Evaluation of an activation point in a regular task.

```
evaluate_irregular_activation_point (static_task_id, π)
{
  t = get_dynamic_task (static_task_id, π);
  stream_indexes_map = build_map (π);

  for ( (u, s, b, h) ∈ π ) {
    start_index = MaxSched(u, s);
    stream_indexes_map(u,s) = (start_index, h, b);
    MaxTaskSched(t, u, s) = start_index + b;
    MaxSched(u, s) = start_index + b;
  }
  current = t.activation_counter + 1;
  add_to_stream (t.control_stream, current, stream_indexes_map);
  t.activation_counter = current;
}
```

Figure 6.16: Evaluation of an activation point in an irregular task.

```
t = get_self_task_identifier ();

while (activation_index = get_activation (t))
{
  // Irregular task: acquire access indexes from t.control_stream
  // Regular task: compute access indexes as
  //              B_⟨t,u,s⟩ * activation_index + D_⟨t,u,s⟩
  // issue update and stall calls

  execute_work_function ();

  // issue commit and release calls
}
```

Figure 6.17: Simple task executor loop.

```
get_activation (t)
{
  while (t.max_assigned_activation >= t.activation_counter)
  {
    if (t.finished == true)
      if (t.max_assigned_activation >= t.activation_counter)
        execute_task_termination ();
  }

  t.max_assigned_activation += 1;

  return t.max_assigned_activation;
}
```

Figure 6.18: Simple sequential scheduler using a blocking function.

```
get_activation_range (t, max_aggregation_factor)
{
  while (t.max_assigned_activation >= t.activation_counter)
  {
    if (t.finished == true)
      if (t.max_assigned_activation >= t.activation_counter)
        execute_task_termination ();
  }

  base_outstanding = t.max_assigned_activation + 1;
  available = t.activation_counter - t.max_assigned_activation;

  if (available > max_aggregation_factor) {
    available = max_aggregation_factor;
  }
  t.max_assigned_activation += available;

  return [base_outstanding, base_outstanding + available - 1];
}
```

Figure 6.19: Work-aggregating sequential scheduler.

```
get_activation (t)
{
  while (t.max_assigned_activation >= t.activation_counter)
  {
    if (t.finished == true)
      if (t.max_assigned_activation >= t.activation_counter)
        execute_task_termination ();
  }

  result = __sync_add_and_fetch (&t.max_assigned_activation, 1);

  return result;
}
```

Figure 6.20: Concurrent scheduler function without aggregation (see Figure 6.18).

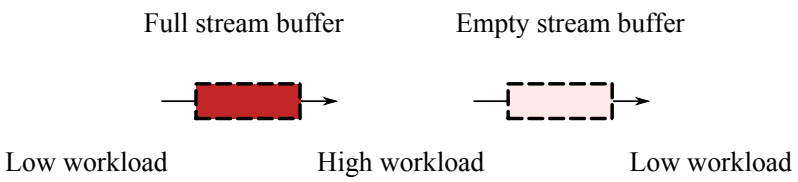


Figure 6.21: Imbalance induced by a heavy filter in a pipeline and effect on connecting streams.

```
update (s, index) {
  t = get_self_task_identifier ();

  if (MinCache(th_id, W, s) > index) {
    RelBalance(t, R, s) -= 1;
    return;
  }

  wait_flag = false;
  do {
    adjusted_min = ∞;
    // compute adjusted_min value as before ...

    if (adjusted_min < index) {
      wait_flag = true;
    }
  } while (adjusted_min < index);

  if (wait_flag == true)
    RelBalance(t, R, s) = max (0, RelBalance(t, R, s) + 1);
  else
    RelBalance(t, R, s) -= 1;

  MinCache(th_id, W, s) = current_min;
}
```

Figure 6.22: Monitoring the relative load balance of a task with respect to an input stream's pressure.

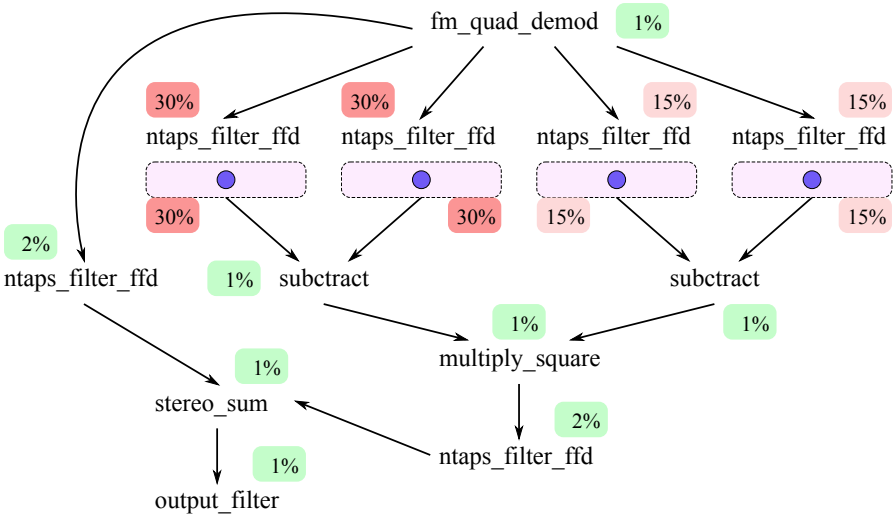


Figure 6.23: FMradio task graph and the proportion of computational load in each task.

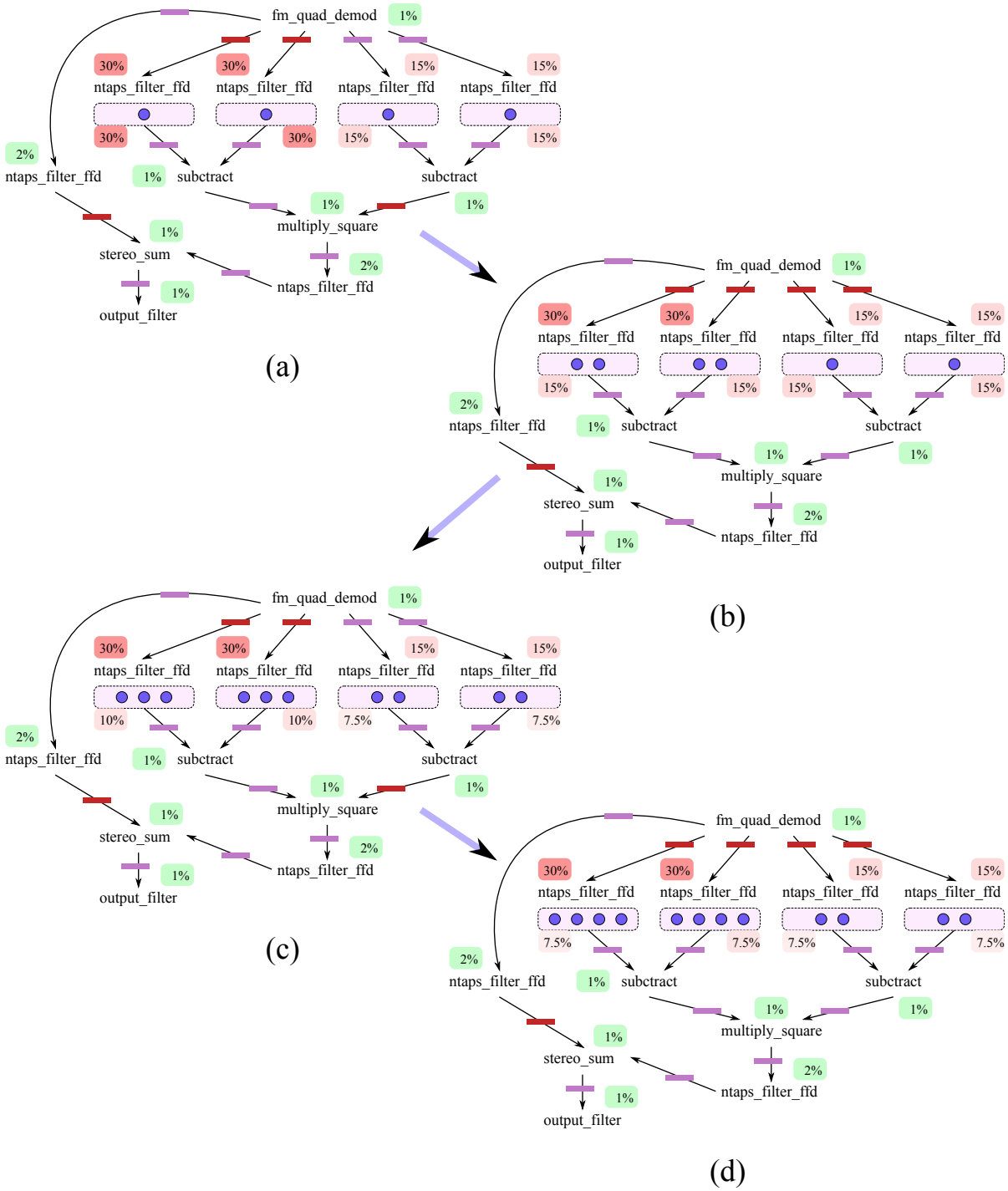


Figure 6.24: Evolution of stream pressure and back-pressure relative load with dynamic load-balancing in FMradio.

```
wait_quiescence (s) {
  q = get_self_thread ();

  // Acknowledge all requests while waiting to acquire stream lock
  while (! __sync_bool_compare_and_swap (&s.quiescence_lock, false, true)) {
    if (is_empty (q.quiescence_requests_queue) == false) {
      p = atomic_dequeue (q.quiescence_requests_queue);
      __sync_add_and_fetch (&p.acknowledged_requests, 1);
      q.received_requests += 1;
    }
  }

  // Notify all producers and consumers that they should stop
  // except the thread itself if it is connected to the stream
  for ( p ∈ producers(s) ∪ consumers(s) \ {q} ) {
    atomic_enqueue (p.quiescence_requests_queue, q);
  }

  // Wait until all acknowledge receipt of request
  while (q.acknowledged_requests < |producers(s) ∪ consumers(s) \ {q}|) {
    if (is_empty (q.quiescence_requests_queue) == false) {
      p = atomic_dequeue (q.quiescence_requests_queue);
      __sync_add_and_fetch (&p.acknowledged_requests, 1);
      q.received_requests += 1;
    }
  }

  // Quiescence achieved on stream s
}

resume_after_quiescence (s) {
  q = get_self_thread ();

  // Notify all producers and consumers of release
  for ( p ∈ producers(s) ∪ consumers(s) \ {q} ) {
    __sync_add_and_fetch (&p.quiescence_release, 1);
  }

  // Release stream lock
  s.quiescence_lock = false;

  // Handle received quiescence requests
  while (q.received_requests > q.quiescence_release) {
    if (is_empty (q.quiescence_requests_queue) == false) {
      p = atomic_dequeue (q.quiescence_requests_queue);
      __sync_add_and_fetch (&p.acknowledged_requests, 1);
      q.received_requests += 1;
    }
  }
  __sync_sub_and_fetch (&q.quiescence_release, q.received_requests);
}
```

Figure 6.25: Stream level quiescence algorithm, requesting thread side.

```
update (s, index) {
  p = get_self_thread ();

  while (min_{p∈producers(s)}(M(p,W,s)) < index) {
    if (is_empty (p.quiescence_requests_queue) == false) {
      q = atomic_dequeue (p.quiescence_requests_queue);
      quiescence_descent (q);

      // Repair damaged pointers upon emergence from quiescence
      for (s' ∈ input(p) ∪ output(p)) {
        // request new stream access pointers
      }
    }
  }
}

quiescence_descent (q) {
  p = get_self_thread ();

  // Acknowledge receipt of quiescence request to the originator
  __sync_add_and_fetch (&q.acknowledged_requests, 1);

  while (p.quiescence_release == 0) {
    if (is_empty (p.quiescence_requests_queue) == false) {
      r = atomic_dequeue (p.quiescence_requests_queue);
      quiescence_descent (r);
    }
  }
  __sync_sub_and_fetch (&p.quiescence_release, 1);
}
```

Figure 6.26: Stream level quiescence algorithm, adjustment to `stall` and `update` primitives.

previous `stall` or `update` operations are now invalid.

Note that a thread either notices quiescence requests before it starts executing the work function or[9] it executes the current activation to completion and have another chance to notice the request at the next iteration. This ensures that failure to notice a request cannot impair a thread's ability to execute properly, which means that the memory latency between the issuance of a request and its perception is unimportant and properly tolerated.

Remark 6.3. *This algorithm requires that quiescence requests be acknowledged by all threads in a timely manner, which may not be possible if external synchronization mechanisms are used. Each thread must regularly check quiescence requests*[10]*, which occurs in* `stall` *and* `update` *calls. Once quiescence is requested for stream s, the only case where high delays can be observed correspond to requests issued by a thread not involved in stream communication through s and if the production and consumption is perfectly balanced, so that none of the involved threads ever need to wait for data or for available space. Of course, this cannot occur when any of the threads involved at any level in the execution of tasks, or the control program, request quiescence as the requesting thread stops operations, therefore leading to either a* `stall` *or* `update` *miss.*

If, however, a synchronization is performed within *the body of tasks, so inside their work function, then it is necessary for the thread to continuously check for quiescence requests while waiting for the synchronization event. It is also necessary to ensure, with a* `store_store_fence` *or a* `LWSYNC` *memory barrier, that all store operations due to the execution of the work function are effective before acknowledging the request, therefore ensuring that the memory effects are moved along with the stream buffer data.*

Using stream level quiescence, we can now provide an algorithm for the dynamic creation of tasks, which we had postponed. The algorithm, presented on Figure 6.27, corresponds to the case where a request for identification of a dynamic task fails. This occurs when the control program attempts to generate a task activation for a task that had no activation points before. In such a case, the task lookup returns a null pointer. The algorithm attempts to acquire a lock on the static task identifier because it is possible that multiple threads that execute the control program see a null pointer if activation points of the same task can be evaluated concurrently. The first thread to acquire this lock is able to re-test the lookup function and still receive a null pointer, therefore creating the task. All threads subsequently acquiring the lock get a non-null pointer on their second test.

The implementation of the task creation itself is straightforward. Each connection between a task and a stream changes the structure of the task graph, notably the sets of producers and consumers defining the hyperedge representing the stream, so it is not sufficient to simply add the new task atomically, as it invalidates the computation of minima in the `stall` and `update` synchronization primitives. If only an atomic operation

[9] In the absence of extraneous synchronization within tasks.

[10] This does not imply a high overhead as the check is only performed during down time and it can be serviced from a local cache unless a request has already been issued. As requests are considered exceptional events, the common case of this check only represents a read of a *local* variable, and it is outside of the *critical path*.

```
get_dynamic_task (static_task_id, π)
{
  t = identify_task (static_task_id, π);

  while (t == NULL) {
    lock (&static_task_id.lock);

    t = identify_task (static_task_id, π);
    if (t == NULL) {
      t = build_task (static_task_id, π);
      for (s ∈ input(t) ∪ output(t)) {
        wait_quiescence (s);
        connect (t, s);
        resume_after_quiescence (s);
      }
      launch_task (t);
    }
    unlock (&static_task_id.lock);
  }
}
```

Figure 6.27: Dynamic task creation algorithm.

is used to add new tasks, then each minimum computation needs to occur atomically, which means making the common case more expensive and opposing the primary design objective of our runtime. We therefore rely on the stream level quiescence algorithm to connect this new task to the task graph.

6.4.2 Dynamic Stream Buffer Size Management

In order to resize a stream buffer, which often involves replacing it with another one, it is necessary to guarantee first that no producer or consumer is still accessing any data in the stream. A second requirement is that no thread has previously acquired handles (i.e., pointers) on data within the old buffer and that are still valid (i.e., not released or committed) upon completion of the operation. Both of these conditions are met, by design, if we rely on stream level quiescence.

The algorithm, presented on Figure 6.28, is entirely straightforward. Note that the `realloc` function copies the old data if the buffer is moved to a new memory location.

6.4.3 Towards a Low Overhead Runtime Deadlock Detection Algorithm

The algorithm presented in Chapter 3, on Figure 4.4, is purposefully designed for brevity while guaranteeing a finite convergence, but it is impractical to implement. In particular,

```
resize_stream_buffer (s, new_size)
{
  wait_quiescence (s);

  s.buffer = realloc (s.buffer, new_size);

  resume_after_quiescence (s);
}
```

Figure 6.28: Stream buffer resizing algorithm.

the form of work-stealing it uses, where a stalled task is allowed to search for and execute activations of other tasks that satisfy its own dependences, is cumbersome to implement in our scheme. As our task activations are not individually tagged but only counted instead, this would only further complicate the algorithm (see Section 6.3.2 for a discussion of the issues with work-stealing in our streaming framework). However, if we postulate that task activations that have their dependences satisfied will eventually execute on their own threads[11], then instead of executing the task activations found during exploration, a stalled task simply backs-off and allows the threads responsible for these activations to proceed. Thus the algorithm becomes slightly less complicated to implement, but it can only be proven to finitely converge under stronger hypotheses, notably on the fairness of the thread scheduler of the execution platform.

The original algorithm is instantiated as follows.

- Trigger condition: within the `stall` and `update` stream synchronization primitives, if the spinning loop exceeds a threshold count, check that the current thread is executing the lowest task activation index in the worker thread team. If such is the case, start exploring the dynamic task graph. Let `s` be the stream the current thread is waiting for, when the condition is satisfied, and let `idx` be the index that was requested.

- Exploration: as specified in Figure 4.4. To determine the dependences, the existing runtime information is sufficient. Indeed, in order to build the dependence chain, the thread polls the tasks at the other end of stream `s` (so the set of producers if the exploration was triggered in `update` or the set of consumers for `stall`). Each task knows the stream access indexes for a given task activation index, so this polling operation consists in finding the task activation index that produces `idx` or consumes `idx - s.buffer_size`.

  Once this task activation index is found, recursively check that if the stream access indexes it requires are themselves satisfiable.

- If a task activation is found to be executable, all lower activation indexes of the same

[11] Which we recall is supposed to always be possible in the control program order because of task causality.

streaming task are executable as well, so the exploration can safely back-off and let the responsible worker team make progress. Continue exploring other dependences.

- Else a deadlock condition is identified. If it is a functional deadlock, then it can be reported directly, without waiting for the whole program to stop. This kind of deadlock is not subject to an issue of interleaving of memory accesses, even if the data gathered by the exploring thread is stale or inconsistent with respect to the perceived order of memory operations from different threads.

  However, if the deadlock condition corresponds to a resource deadlock, then it is possible that recent activity, occurring on other threads during the exploration phase, may have changed the state and invalidated the condition. Rather than attempting to reach consensus among all the threads involved, we only re-check the whole dependence chain for any possible alterations. If the resource deadlock condition still holds, we conservatively consider that the identified blocking stream buffer should be resized. If there was no resource deadlock, the only drawback is the time lost in the quiescence algorithm required for resizing, but it is likely that a larger buffer, for a stream experiencing a high level of contention, will be beneficial to the execution.

- The continuation activation ($\mathcal{C}(\mathcal{K}_e)$ in the CDDF model) is not directly represented in the runtime. However, if the search for dependences yields a task activation index greater than the `activation_counter` of the task, then it is understood to belong to the continuation activation. If such is the case, the thread checks if the control program has reached a barrier, possibly concluding to an insufficiency deadlock.

- Deadlock condition reporting consists in printing the dependence chain, with task identifiers, task activation indexes and incriminating stream access indexes.

The overhead of such an algorithm is primarily due to the cache traffic generated by polling remote threads. As the exploration is only performed while waiting on a stream synchronization, it does not introduce overhead at that point. The only problematic case occurs when the system is over-scheduled, that is when more execution threads are used than available hardware threads. In such a case, the spinning stream synchronization algorithm must include a scheduler `yield()` call to allow other threads to be scheduled. The choice of the threshold for the trigger condition is the key factor, both for the reactivity of the detection scheme and for the overhead incurred.

6.5 Runtime API for Code Generation

The implementation of our runtime support for streaming extends the current implementation of the GCC runtime for OpenMP, libGOMP. We provide a set of primitives that are used by our code generation pass, also implemented in the GCC compiler. In this section, we detail the interface of our runtime and briefly present the functionality provided by each visible function.

6.5.1 Initialization and Termination

In our model, the task graph of a streaming application is dynamic. It is necessary to create dynamically new streams and tasks, which are then connected to the existing task graph. For this reason, the initialization phase is almost absent in our model. Some data structures must be initialized by the control program before they can be used, in particular if the control program is itself concurrent.

The first such data structure is the stream. It must be initialized with the following call prior to any evaluation of an activation point that uses it. In the case of parallel control programs, such a call must be issued before entering the parallel region to avoid duplicates.

```
stream_id
GOMP_stream_create_stream (element_type_size, array_size, broadcast_size);
```

This call allocates a new stream, or an array of streams of size `array_size`, where the underlying stream elements is of size `element_type_size`. When a broadcast array is used, no additional dimensions are created, but the runtime knows how many producers and consumers to wait for in the minimum computation. Scalar stream variables have `array_size == 1`, and non-broadcasting streams should use `broadcast_size == 1`.

Stream deallocation is performed dynamically by the last consumer to disconnect from a stream. The control program must mark the end of use of a stream with the following function.

```
void
GOMP_stream_finish_stream (stream_id);
```

This call needs to be executed by a single thread for a given stream, and it cannot happen before the last activation point that communicates on this stream has been evaluated.

The case of streaming tasks is particular because they are dynamically created. We need a lock to ensure that only one instance is created if the control program is concurrent. Instead of relying on a program-level lock, we initialize a lock for each OpenMP task construct at any non-concurrent point in the control program's control flow that dominates the task construct with the following function.

```
static_task_id
GOMP_stream_init_static_task_id (void);
```

The function returns a `static_task_id`, which only represents the syntactical task construct. We generate it in the runtime to facilitate modular compilation, but it is a purely static identifier.

Termination of a streaming program occurs cooperatively when all streams have been deallocated. Indeed, as streaming tasks cannot connect to new streams[12], once a task sees any of its connections broken, it can immediately either conclude to an insufficiency deadlock, if it has unexecuted activations, or it can terminate.

[12]If such a thing is necessary, this simply creates a new task, thus preserving the CDDF view of tasks as equivalence classes on input and output connections.

6.5.2 Stream Communication

The synchronization of stream communication is entirely handled by the four primitives presented in Section 6.2. The following functions implement them in our runtime:

```
void GOMP_stream_update (stream_id, stream_access_index);
void GOMP_stream_stall (stream_id, stream_access_index);
void GOMP_stream_release (stream_id, stream_access_index);
void GOMP_stream_commit (stream_id, stream_access_index);
```

In order to reduce the code generation effort, we also provide more aggregated methods that allow to either acquire all the necessary stream resources, including data on input streams and free space on output streams, or to release the previously acquired resources, in the following functions.

```
void
GOMP_acquire_stream_accesses (thread, activation_index_start, activation_index_end);
void
GOMP_release_stream_accesses (thread, activation_index_end);
```

The implementation of these functions is provided in Chapter 7, on Figure 7.9. The first parameter is the thread data structure associated with the current thread. These functions work on ranges of task activation indexes, requesting in bulk all stream resources required to execute all activations in the range.

To enable this behaviour, the compiler needs to handle stream access views, which as we will see behave as simple C pointers, and issue the following call to register views with the runtime.

```
void GOMP_stream_register_view (thread, &view_pointer, access_direction, stream_id);
```

This function is issued by each worker thread individually to register its view pointers. The second parameter is used to pass the addresses of the thread's views. The caller further needs to specify the stream and access direction, either read or write, for which this view will be used.

Relying on this information, the runtime is able to set the view pointers automatically for the next activation, with a call to the following function.

```
bool GOMP_stream_next (thread, current_activation_index);
```

This is also used, as we have discussed for the stream buffer resizing scheme, in Section 6.4, to allow re-setting the pointers that were invalidated by a stream buffer replacement.

6.5.3 Activation Point Evaluation and Scheduling of Task Activations

The evaluation of activation points is almost entirely handled by the following runtime functions. In the general case, the code generation only needs to handle the marshalling of firstprivate data, which is then passed to the two functions:

```
void
GOMP_stream_regular_activation (static_task_id, firstprivate_data,
                                        extended_stream_access_descriptors);
void
```

```
GOMP_stream_irregular_activation (static_task_id, firstprivate_data,
                                  stream_access_descriptors);
```

The first function handles the activation points of regular tasks. It only takes as parameters the static task identifier, the firstprivate marshalled data and a set of stream access descriptors, each extended with two additional fields storing the statically determined constants $B_{\langle t,u,s\rangle}$ and $D_{\langle t,u,s\rangle}$. We recall that a task's regularity is contingent on the static determination of these constants, even if they are parametric. Instead of the usual (u, s, b, h) stream access descriptors that make up activation points, a regular task t will have descriptors of the form $(u, s, b, h, B_{\langle t,u,s\rangle}, D_{\langle t,u,s\rangle})$.

The second function is similar, but uses the default stream access descriptors. It would be useful, however, to allow it to also rely on the additional information of extended stream access descriptors when some of the streams it connects to have regular accesses and only a subset of its connections requires the more complex, irregular, mechanisms.

The scheduling functions are introduced in Section 6.3:

```
bool
GOMP_get_activation (thread, &activation_index);
bool
GOMP_get_activation_range (thread, &activation_index_start,
                           &activation_index_end, max_aggregation);
```

The first function requests a new unit of work and blocks until work is available and returns true when a task activation is acquired. It returns false when the task does not have further activations. The second function acquires a range of task activations, limited by the `max_aggregation` cap.

6.6 Conclusion

In this chapter, we presented the runtime algorithms used to perform stream communication synchronization, activation point evaluation and task activation scheduling, as well as the mechanisms used to improve load balancing, to dynamically resize stream buffers and to detect and resolve deadlocks. We finally provided an API for the code generation pass presented in Chapter 7.

We have achieved our goal of minimizing the overhead incurred on the critical path. In the common cases of execution, tasks only require atomic operations in the scheduler functions and this, only when they are executed by a team of worker threads. The particular case of *critical tasks*, the slowest tasks that determine the throughput of the task graph, is specifically optimized to avoid unwarranted overheads. For such tasks, the usual cost of stream synchronization is a single test of a local variable and, in the worst case, a number of loads of non-local variables proportional to the number of threads executing tasks connected to it.

Chapter 7

Work-Streaming Compilation

This chapter presents our code generation framework for our stream-computing extension to OpenMP. We detail the code generation required for the control program and the way we build the worker thread functions that execute task activations. In a second step, we discuss the techniques used to generate optimized code, in particular in the case of regular tasks. We focus on optimizations that are conducive to threads accessing memory in regular patterns, improving cache locality and reducing false sharing, to reduce the strain on the memory subsystem. A second objective is to generate code that does not preclude further compiler optimizations, for instance by laying the groundwork for the vectorization of worker thread code.

Ce chapitre présente l'algorithme de génération de code utilisé pour compiler notre extension streaming au langage OpenMP. Nous y distinguons la génération de code nécessaire au programme de contrôle et la construction des fonctions de travail qui exécutent les activations de tâches. Nous développons ensuite les techniques utilisées pour générer du code optimisé dans le cas particulier des tâches régulières. Afin de réduire la pression sur la mémoire, nous ciblons tout particulièrement les optimisations qui favorisent des accès réguliers à la mémoire dans les threads, améliorant ainsi la localité des caches, et, en second lieu, nous évitons de générer du code difficile à optimiser par le compilateur, préparant le terrain, par exemple, pour la vectorisation du code des threads de calcul.

7.1 Introduction

Our code generation framework applies code transformations enabled by the static analyses presented in Chapter 5 and targeting the runtime support API presented in Chapter 6. The first part of this chapter details the default code generation that results from expanding the OpenMP streaming task constructs without any knowledge of the program. This approach represents the conservative fall-back code generation when all of the static analyses fail. As we remark in the first two sections, there is little hope to achieve significant performance improvements with this default code generation scheme, let alone scalability, barring some very high granularity of work and computational intensity in task

activations. It is important, however, to ensure that programs compile and can execute even if all analyses fail.

In a second step, we explore some of the essential optimized code generation patterns enabled by the static analyses outlined in Chapter 5. We detail the aggressive aggregation techniques that can be applied to task worker threads to reduce runtime overhead, improve cache behaviour and enable further compiler optimizations. We also discuss the optimizations that can be enabled for the control program, in particular for the evaluation of activation points and for regular tasks.

The remainder of this chapter is organized as follows. Section 7.2 presents the basic code generation algorithm for the expansion of streaming tasks in the control program, detailing the generation of initialization code, activation point evaluation and firstprivate communication. Section 7.3 describes the generation of worker thread code, including the outlining of the body of streaming tasks in functions, where the original accesses to variables are replaced with direct accesses in stream buffers, and the addition of a scheduler loop and of stream synchronization code. Section 7.4 focuses on the essential optimizations that can be enabled during code generation, in particular for regular tasks. Finally, Section 7.5 presents a technique that allows task worker threads to perceive stream buffers as continuous, enabling simple traversal of stream data as an array.

7.2 Control Program Code Generation

In this section, we present the code generation algorithm for the control program. In OpenMP terminology, this is also called the outer context with respect to a task, or the main program. In addition to the generation of code for evaluating activation points, we also take care of a few additional issues, like the initialization of streams, the generation of static task identifiers and the generation of communications between the control program and the tasks.

Throughout this section, we rely on a simple, yet representative, running example. Presented on Figure 7.1, the example exhibits all of the most important behaviours with respect to the necessary code generation patterns for the control program. It consists of two task constructs communicating through a stream x. The first task construct communicates with the control program through a firstprivate clause, then produces data on stream x, while the second task construct reads data from x and produces data on some stream within an array of streams A. The program does not include consumers for the streams within A which are used at some later point.

The array of streams A is declared outside the loop nest, which means that these streams are persistent throughout the execution of the loop nest. On the other hand, stream x is declared within the first loop, which means that its lifespan, determined by its scoping, is only one iteration of the outermost loop. For this reason, the first task construct generates K different tasks, each producing N values on one of the streams x relative to one iteration of this loop. The stream used by the firstprivate clause is always tied to the *streaming* task, so this will also generate different streams for the values communicated for j, one for each of the K streaming tasks.

The second task construct also generates K streaming tasks, both because of the

```
int A[K];

#pragma omp parallel
#pragma omp single
  for (i = 0; i < K; ++i) {
    int x;

    for (j = 0; j < N; ++j) {
#pragma omp task firstprivate (j) output (x)
      x = ... j ... ;

#pragma omp task input (x) output (A[i] << y)
      y = x;
    }
  }
}
```

Figure 7.1: Running example for the generation of control program code.

stream x, which is a different stream in each iteration of the outermost loop, and because of the array of streams A, which is subscripted by the induction variable of the outermost loop, i.

In the remainder of this section, we present the successive expansion steps, necessary for these two OpenMP task annotations, in the control program. We first generate initialization code, then we issue the calls to the runtime functions for evaluating the activation points and finally we handle the communication between the control program and the streaming tasks, resulting from the firstprivate clause on the first task construct.

7.2.1 Initialization

In our semantics, the task graph is generated by the execution of the control program. We have a small initialization phase, that is only required for concurrent control programs. It enables a simple synchronization scheme to avoid the creation of duplicate dynamic tasks. Our running example uses a sequential control program, relying on the OpenMP `single` worksharing construct, but we generate this code nonetheless as it does not introduce significant overhead and it may be needed if the control program is parallelized in a later stage of the compilation.

Figure 7.2 presents the code generated after the initialization pass for our running example. We replace the declaration of stream variables, here A and x, by stream identifiers, which are pointers to stream data structures. They are initialized just after their declaration with a call to the runtime. This allocates the stream buffers, with an initial size either provided as an execution parameter or computed at run time based on the size of the cache, and all necessary data structures. The array of streams A is not declared as an array of identifiers, though in fact the runtime does allocate the K streams[1]. As discussed, the stream x is initialized within the outermost loop, as its scope

[1]We may, in future work, relax this allocation policy and just allocate the data structures of streams, then allocate the buffers when the first producer requests an access with the `stall` primitive.

```
static_task_id t1, t2;
stream_id A;

t1 = GOMP_stream_init_static_task_id ();
t2 = GOMP_stream_init_static_task_id ();

A = GOMP_stream_create_stream (sizeof(int), K, 1);

#pragma omp parallel
#pragma omp single
  for (i = 0; i < K; ++i) {
    stream_id x; // Stream private to the iteration (visibility scoping)
    x = GOMP_stream_create_stream (sizeof(int), 1, 1);

    for (j = 0; j < N; ++j) {
#pragma omp task firstprivate (j) output (x) // Static id: t1
      x = ... j ... ;

#pragma omp task input (x) output (A[i] << y) // Static id: t2
      y = x;
    }
    GOMP_stream_finish_stream (x);
  }

// ... use of the streams in A[i]

GOMP_stream_finish_stream (A);
```

Figure 7.2: Generation of initialization code for our running example on Figure 7.1.

is restricted to one iteration. The streams thus created are marked for destruction at the end of their visibility scope. The destruction only occurs once the last task activation that accesses the stream completes. In the same way, the array of streams A is also marked once the variable A reaches the end of its scope and it can no longer be used by the control program to generate new task activations that would use it.

The second step consists in attributing static task identifiers to each task construct. This is a purely syntactical issue in general[2] but in order to avoid relying on a global program-level lock, we prefer to instantiate an individual lock for each task construct. This lock is needed when activation points of the same task can be evaluated by multiple threads, as the first thread to evaluate such an activation point is responsible for generating the dynamic task and launching its first worker thread. We therefore rely on a runtime function to generate unique static identifiers and initialize a lock for each task construct. These functions must be called in a non-concurrent context and before any activation points of the same tasks can be evaluated.

[2]In the case of modular programs, it is not possible to attribute contiguous integer identifiers, but a simple naming convention can be substituted.

7.2.2 Evaluation of Activation Points

The evaluation of activation points is, in the general case, performed by a runtime function, so the compiler needs only to issue the proper call. In some cases, which we discuss in Section 7.4, this is replaced by direct code generation to avoid the overhead of a function call and enable further optimizations.

```
static_task_id t1, t2;
stream_id A;

t1 = GOMP_stream_init_static_task_id ();
t2 = GOMP_stream_init_static_task_id ();

A = GOMP_stream_create_stream (sizeof(int), K, 1);

#pragma omp parallel
#pragma omp single
  for (i = 0; i < K; ++i) {
    stream_id x;
    x = GOMP_stream_create_stream (sizeof(int), 1, 1);

    for (j = 0; j < N; ++j) {
//#pragma omp task firstprivate (j) output (x) // Static id: t1
      GOMP_stream_regular_activation (t1, NULL, [(W,x,1,1,1,0)]);

//#pragma omp task input (x) output (A[i] << y) // Static id: t2
      GOMP_stream_irregular_activation (t2, NULL, [(R,x,1,1), (W,A[i],1,1)]);
    }
    GOMP_stream_finish_stream (x);
  }
}
```

Figure 7.3: Generated activation point evaluation code for our running example on Figure 7.1.

The code generated for evaluating activation points is presented on Figure 7.3. The first task construct is identified as corresponding only to regular tasks because it uses a single scalar stream variable, x, which is not produced by any other task. Note that firstprivate clauses do not need to be taken into account for this decision. The second task can be identified as regular, but we assume that the analysis fails for this task to show the code generation for irregular tasks.

The compiler issues a call to either the regular or the irregular version of the runtime function, and removes the original annotation as well as the body of the task. The parameters are the static task identifier, a pointer to a data structure built for communicating between the control program and the task[3] and a set of stream access descriptors defining the activation point. This set is aggregated in a data structure passed as argument, which we represent as a list in the parameter list.

Two additional parameters are added to the stream access descriptors for regular tasks:

[3]This data structure is introduced, and the parameter filled, in the next section.

the constants that define the closed-form expression for evaluating the stream access indexes in a given stream. In our example, and using the notation from Definition 5.1 on page 138, we have a perceived burst of 1 and a delay of 0 for task $t1$ on stream x with write accesses:

$$B_{\langle t1,W,x\rangle} = 1 \qquad \wedge \qquad D_{\langle t1,W,x\rangle} = 0$$

We recall (see Section 5.3.1) that the perceived burst is the required shift in the stream after each execution and it is identical to the actual burst if there is no other task interleaving accesses in the same stream. These two values appear in the last parameter of the activation point evaluation function, generated for the first task construct, on Figure 7.3.

7.2.3 Communication with Streaming Tasks

The firstprivate clause that appears on the first task construct is not translated yet. This relies on a mechanism similar to classical OpenMP expansion of non-streaming tasks, where firstprivate variables are marshalled in a data structure passed as argument to the task. We also rely on the same marshalling technique and pass the structure to the activation point evaluation function, which forwards it to the task. The forwarding is handled by a control stream, which is uniquely produced by the control program and consumed by the task to which it belongs. This stream is always guaranteed to contain the data required for the execution of a given activation at a stream access index equal to the activation index itself. In other words, does not require any index computation.

Figure 7.4 presents the result of this last step. An assignment of the firstprivate variable j is added just before the activation point evaluation call. The null pointer in the following call is replaced with a pointer to the marshalling data structure.

Now the code generation is complete in the control program and we handle the general, unoptimized, case of generating functions for worker threads.

7.3 Generating Task Worker Thread Functions

In this section, we present the generation of code for the tasks, which is also called the "inner context". In OpenMP, this consists in outlining the body of a task in a function and adding the marshalling and unmarshalling statements required to re-construct the execution environment of the code contained in the original task block. A pointer to this outlined work function is then packed along with the data that was copied from the outer context and placed on a scheduler queue. However, as our execution model imposes the monotonicity of activation indexes that can be executed by a given thread, general task schedulers cannot be used for streaming task activations. We do not generate work functions that correspond to executing one task activation, but rather build a function that contains its own scheduler loop, therefore representing a worker thread with a scheduler queue dedicated to a task and its team of workers. This reduces the contention due to a single centralized queue, or the need to balance distributed queues, therefore simplifying the implementation.

As above, we use a single running example, presented on Figure 7.5, to show the

```
static_task_id t1, t2;
stream_id A;

t1 = GOMP_stream_init_static_task_id ();
t2 = GOMP_stream_init_static_task_id ();

A = GOMP_stream_create_stream (sizeof(int), K, 1);

#pragma omp parallel
#pragma omp single
  for (i = 0; i < K; ++i) {
    stream_id x;
    x = GOMP_stream_create_stream (sizeof(int), 1, 1);

    for (j = 0; j < N; ++j) {
//#pragma omp task firstprivate (j) output (x) // Static id: t1
      t1_firstprivate_data.j = j;
      GOMP_stream_regular_activation (t1, &t1_firstprivate_data,
                                      [(W,x,1,1,1,0)]);

//#pragma omp task input (x) output (A[i] << y) // Static id: t2
      GOMP_stream_irregular_activation (t2, NULL, [(R,x,1,1), (W,A[i],1,1)]);
    }
    GOMP_stream_finish_stream (x);
  }
}
```

Figure 7.4: Generation of firstprivate communication code.

code generation patterns with respect to the inner context. This example has a single streaming task, performing a simple moving average computation on an input stream x, weighted by a coefficient, a, read from the enclosing context, and writing its output on a stream y. The stream x is viewed through a window X with a burst of 2 elements and a horizon of 3 elements. This means that the task can see 3 elements of the stream at a time, but only 2 elements are consumed.

```
int x, a;
int X[3];

#pragma omp task firstprivate (a) input (x >> X[2]) output (y)
  {
    y = a * (X[0] + X[1] + X[2]);
  }
```

Figure 7.5: Running example for the generation of task work functions.

The steps of this code generation are: (1) the outlining of the task body in a function, (2) the adition of unmarshalling and initialization code, (3) wrapping a scheduler loop around the original body of the task, and (4) the insertion of the necessary calls for stream synchronization and access to stream buffers through views.

7.3.1 Task Body Outlining

The generation of code for task worker functions starts with outlining the task body in a separate function. We replace all non-local variables, with the exception of shared variables, by pointers that are used to access data directly in the stream buffers connected to the task.

On Figure 7.6, the generated worker function takes as parameter a data structure that provides a pointer to the thread data structure. Calls to a registration function are issued for all pointers replacing non-local variables. These calls register the association between this pointer and the stream identifier to which they should provide access. This creates a view data structure in the runtime, associated to the current thread.

```
worker_function (void *parameters) {
  int *view_X;
  int *view_y;

  // Unpacking parameters and initialization
  p = parameters->thread;
  GOMP_stream_register_view (p, &view_X, R, parameters->stream_id_x);
  GOMP_stream_register_view (p, &view_y, W, parameters->stream_id_y);

  // Body of the task
  view_y[0] = p->control_view[0].a * (view_X[0] + view_X[1] + view_X[2]);
}
```

Figure 7.6: Outlined work function where stream accesses are converted to accesses through views in the stream buffer.

7.3.2 Scheduler Loop

The second step consists in generating a scheduler loop, traversing the iteration space of the task. This while loop is presented on Figure 7.7. If multiple worker threads are present in the task's team, then the `GOMP_get_activation` function will switch to assigning activation indexes atomically.

Note that the `GOMP_get_activation` function also synchronizes the control stream associated with the task and sets the control view, local to the thread, that can be accessed for any firstprivate data as well as control information when necessary.

7.3.3 Stream Synchronization

Finally, we add the stream synchronization calls and update the view pointers registered with the thread, to ensure that the access of stream data through the view pointers corresponds to the current activation. Figure 7.8 shows the two additional calls to `GOMP_acquire_stream_accesses` and `GOMP_release_stream_accesses`, which provide this functionality.

In order to reduce code generation for the unoptimized case, all the required calls to synchronization primitives, as well as any index computations, are aggregated in these

```
worker_function (void *parameters) {
  int *view_X;
  int *view_y;

  // Unpacking parameters and initialization
  p = parameters->thread;
  GOMP_stream_register_view (p, &view_X, R, parameters->stream_id_x);
  GOMP_stream_register_view (p, &view_y, W, parameters->stream_id_y);

  // Scheduler loop
  while (GOMP_get_activation (p, &act_idx))
    {
      // Body of the task
      view_y[0] = p->control_view[0].a * (view_X[0] + view_X[1] + view_X[2]);
    }
}
```

Figure 7.7: Addition of a scheduler loop on activation indexes.

```
worker_function (void *parameters) {
  int *view_X;
  int *view_y;

  // Unpacking parameters and initialization
  p = parameters->thread;
  GOMP_stream_register_view (p, &view_X, R, parameters->stream_id_x);
  GOMP_stream_register_view (p, &view_y, W, parameters->stream_id_y);

  // Scheduler loop
  while (GOMP_get_activation (p, &act_idx))
    {
      GOMP_acquire_stream_accesses (p, act_idx, act_idx);

      // Body of the task
      view_y[0] = p->control_view[0].a * (view_X[0] + view_X[1] + view_X[2]);

      GOMP_release_stream_accesses (p, act_idx);
    }
}
```

Figure 7.8: Addition of stream communication synchronization.

two functions, presented on Figure 7.9. In the case the code can be optimized, e.g. in regular tasks, the useful part of the functions is inlined in the worker function.

The first function aggregates all the blocking primitives used to wait for data available in input streams or for space in output streams. To work on ranges of task activations, it takes as parameters both the lowest and the highest indexes requested. The function traverses all the views registered with the thread p and computes the base stream access indexes for the lowest and highest task activation requested. If the thread p happens to

```
GOMP_acquire_stream_accesses (p, act_idx_low, act_idx_high) {
  t = p->task;
  span = act_idx_high - act_idx_low;

  for (v ∈ p->registered_views) {
    // compute the stream access index
    if (t->is_regular) {
      base_index_low = B(t,v->u,v->s) * act_idx_low + D(t,v->u,v->s);
      base_index_high = B(t,v->u,v->s) * act_idx_high + D(t,v->u,v->s);
      horizon = H(t,v->u,v->s);
    } else {
      base_index_low =
        p->control_view[0].stream_indexes_map(v->u,v->s).start_index;
      base_index_high =
        p->control_view[span].stream_indexes_map(v->u,v->s).start_index;
      horizon = p->control_view[span].stream_indexes_map(v->u,v->s).horizon;
    }

    // Request access to the highest accessible indexes
    if (v->u == R) {
      GOMP_stream_update (s, base_index_high + horizon);
    } else { // (v->u == W)
      GOMP_stream_stall (s, base_index_high + horizon);
    }
    // Set the view pointer in the stream buffer to the beginning
    *v->view_pointer = v->s->buffer + (base_index_low % v->s->buffer_size);
  }
}

GOMP_release_stream_accesses (p, act_idx) {
  t = p->task;

  for (v ∈ p->registered_views) {
    // compute the stream access index for the release
    if (t->is_regular) {
      release_index = B(t,v->u,v->s) * (act_idx + 1) + D(t,v->u,v->s) - 1;
    } else {
      release_index =
        p->control_view[0].stream_indexes_map(v->u,v->s).start_index;
      release_index += p->control_view[0].stream_indexes_map(v->u,v->s).burst;
    }

    if (v->u == R) {
      GOMP_stream_release (s, release_index);
    } else {
      GOMP_stream_commit (s, release_index);
    }
  }

}
```

Figure 7.9: Aggregated stream synchronization functions.

be working for a regular task, then this computation relies on the closed-form expression[4], otherwise the control stream is accessed through the control view to retrieve the starting point of stream accesses computed during the evaluation of the activation point (see Figure 6.16). Once all the indexes are computed, the function issues the `update` or `stall` call depending on whether the view is for reading or for writing to the stream. Finally, the view pointer, registered by the worker function with the thread, is set to the first accessible index. This pointer is updated, for the sake of brevity, with a modulo computation, which is too slow if the buffer size is not a power of 2. We further discuss this computation in Section 7.5, where we show the mechanism that ensures that all of the buffer elements for which a thread is granted access with this call are contiguous. This is essential to prevent the buffer wrap-around issue from occurring within the worker function code, which would hinder possible optimizations on its code.

The second function is similar, but computes the stream access index that can be released. It is understood that once a range of task activations is acquired with the first function, `GOMP_acquire_stream_accesses`, the upper bound of the range must be released with the second function, `GOMP_release_stream_accesses`. For regular tasks, the release or commit index for each stream is computed as one less than the known next access index that will be acquired. This knowledge is, however, absent in the case of irregular tasks. We need to get the starting point of the task activation that is released and add the burst to account for the elements discarded by the current activation. This does not allow to release all of the indexes that will not be accessed, due to the absence of knowledge on the future of an irregular task, and leads to the use of the adjusted minimum computation pattern presented on Figure 6.9.

7.4 Optimized Code Generation

The code generated so far does not look very appealing. A lot of overhead is introduced and the resulting code is quite hard to analyze by an optimizing compiler, due to the opaque runtime calls. Unless the granularity of work and the computational intensity are very high, this may not lead to any performance improvements. The code generation can be improved by using results obtained by the static analysis framework presented in Chapter 5.

In this section, we present some of the most important cases where optimized code can be generated. We first present two optimizations for the worker function, the aggregation of work and the aggregation of data. We discuss their implications both from the perspective of the positive execution behaviour they foster and in the way they enable further compiler optimizations of the worker code. In a second part, we focus on the optimizations on the control program side, first through sequential code optimizations and second through control program parallelization.

We use the same running examples as before. For the control program code generation, we use the example on Figure 7.1, but its second task construct is no longer considered irregular[5], to generate optimized code. For the generation of optimized worker function code, we reuse the example on Figure 7.5.

[4]Note that horizons are always constant in regular tasks.

[5]We recall that it was decided, in Section 7.3, to consider it irregular to exhibit the code generation

7.4.1 Work Aggregation

Worker functions can be optimized by acquiring multiple units of work, i.e. multiple task activation indexes, in a single call to the runtime. Figure 7.10 shows this pattern, where the `GOMP_get_activation` call is replaced by a call to `GOMP_get_activation_range`. A second loop must be added to iterate over the task activation indexes obtained. The rest of the code remains unchanged.

```
worker_function (void *parameters) {
  //Initialization code (same as before)

  // Scheduler loop
  while (GOMP_get_activation_range (p, &low_idx, &high_idx, max_aggregation))
    {
      for (act_idx = low_idx; act_idx <= high_idx; act_idx += 1)
        {
          GOMP_acquire_stream_accesses (p, act_idx, act_idx);

          // Body of the task
          view_y[0] = p->control_view[0].a * (view_X[0] + view_X[1] + view_X[2]);

          GOMP_release_stream_accesses (p, act_idx);
        }
    }
}
```

Figure 7.10: Enabling work aggregation in worker threads.

The first benefit of this scheme is that fewer calls to the runtime scheduler function are required. This can have a high impact on performance when multiple threads share the same scheduler as it reduces the number of atomic operations and reduces contention. However, as mentioned in Section 6.3, this raises some fairness issues when one thread acquires too many task activations at once and deprives of work other threads in the team. The solution introduced on Figure 6.19, in the form of a maximum work aggregation factor, is far from perfect as it relies on the heuristic choice of this factor. We have further argued, in Section 6.3, that work stealing, the common approach to this type of issue, is impractical to implement in our framework. One solution is to use an adaptive refinement of the aggregation factor. Such a scheme could be implemented by allowing threads stalling for work to inspect the other threads in the team and, if they have acquired significantly too much work, reduce their aggregation factor to prevent them from acquiring as much in the future. On the other hand, threads could also be allowed to increase their aggregation factor when the amount of available work is significantly greater than this factor multiplied by the size of the worker team. Note that this type of intervention requires no synchronization, as it does not impact correctness, and it can tolerate any amount of memory latency, or even being lost due to another thread overwriting the variable.

---

patterns for irregular tasks.

The second and more important benefit is that this transformation enables to further aggregate data. The scheduler loop ensures that the control dependences of the original task, before outlining from the control program, are satisfied. Once a task activation index is acquired, it is guaranteed to be executable, provided that stream synchronization allows it. For this reason, work aggregation is inherently safe. Note that this aggregation cannot block and wait for sufficient activations to be available to match the aggregation factor, as this could lead to a deadlock. Instead, our implementation returns immediately the lesser of the available task activations and the maximum aggregation factor.

7.4.2 Data Aggregation

Once work aggregation is performed, the acquisition of a range of task activations guarantees that either the data required to enable all of the activations is or will be made available in the future, or there is a deadlock in the program. It is very important, however, to note that synchronizing all of the stream accesses in the range of activations at once is not safe. It can introduce deadlocks if the task belongs to strongly connected components. Indeed the task not only requires a block of data that may not be available at once in a cycle, which depends on the delay or slack in the cycle, but it also withholds its `commit` and `release` operations, therefore doubling the amount of slack required in the cycle. In general, we do not perform this optimization for tasks belonging to strongly connected components of the static over-approximation of the task graph unless it is possible to statically prove that sufficient delay has been introduced in cycles to avoid deadlocks. A more interesting approach consists in adjusting this data aggregation factor at runtime. This would allow a much more precise decision on whether the task does indeed belong to a strongly connected component of the dynamic task graph and it would also allow to precisely adjust the data aggregation factor to the best fitting amount, given the delay present in the cycle(s) to which it belongs.

Figure 7.11 contains the code generated to aggregate data in worker functions. The loop introduced on Figure 7.10 for traversing the range of task activations acquired is blocked with factor, `block_size`, the data aggregation factor. The synchronization operations for the acquisition of stream resources are merged. An additional loop is generated to traverse each block, and we need to issue a call to an advancement function, `GOMP_stream_next`, which updates the view pointers for the next task activation. This function works like the `GOMP_acquire_stream_accesses` function, of Figure 7.9, with respect to the update of the view pointers. It also will be subject to further optimization.

This scheme firstly reduces the number of synchronization operations per stream communication. It also changes the way communication occurs. Indeed, if the aggregation factor allows to ensure that, in particular for regular tasks, the amount of data synchronized by each operation is an aligned multiple of the cache line size, this virtually eliminates false sharing[6] in stream communication. This constitutes one of the most important performance optimizations in our model.

[6]The only case where false sharing can still occur is for output streams where multiple producer tasks interleave their accesses in the same stream. If such is the case, then they will compete for exclusive ownership of the cache lines if their execution leads them to produce in the same range of indexes at the same time. Note that sharing between worker threads of the same team cannot occur as they acquire

```
worker_function (void *parameters) {
  //Initialization code (same as before)

  // Scheduler loop
  while (GOMP_get_activation_range (p, &low_idx, &high_idx, max_aggregation))
    {
      for (base_idx = low_idx; base_idx <= high_idx; base_idx += block_size)
        {
          upper_bound = min (high_idx, base_idx + block_size - 1);
          GOMP_acquire_stream_accesses (p, base_idx, upper_bound);

          for (act_idx = base_idx; act_idx <= upper_bound; act_idx += 1)
            {
              // Body of the task
              view_y[0] = p->control_view[0].a * (view_X[0] + view_X[1] + view_X[2]);

              GOMP_stream_next (p, act_idx);
            }

          GOMP_release_stream_accesses (p, upper_bound);
        }
    }
}
```

Figure 7.11: Enabling data aggregation in worker threads.

The `GOMP_stream_next` function call prevents the compiler from optimizing the innermost loop. We can generate more palatable code to replace this call, at least for regular tasks. We present, on Figure 7.12, the code of the `GOMP_stream_next` function for a regular task. The closed-form expression for evaluating the next activation's stream access indexes is used. One issue remains, which is not simple to address at this point: the computation of the next pointer position in the stream buffer needs to account for the wrap-around at the end of circular buffers. However, as we discussed in Section 7.3.3 for the Figure 7.9, the `GOMP_acquire_stream_accesses` function ensures that the block of data to which it grants access is contiguous and can be accessed sequentially. This strong guarantee is obtained thanks to the scheme we present in Section 7.5. This advancement function also advances the control view, which only requires an incrementation in all cases.

Thanks to this property, the wrap-around code can be omitted, as can most of the computation. The only thing left to do is to increase the current view pointer by the perceived burst of the task on that stream `B(t,v->u,v->s)`. We present the resulting code on Figure 7.13, where we generate this advancement operation directly in the body of the innermost loop. We do not require a loop on the set of registered views as it can trivially be statically unrolled as each view corresponds to a syntactically different clause on the original OpenMP task construct. Note that the perceived burst is a static information, even if it comes in the form of a constant parameter. In the case of our example, originally presented on Figure 7.5, the task is always single-producer and single-consumer of its streams, so the perceived burst is equal to the actual burst, 2 on stream

different task activation index ranges.

```
GOMP_stream_next (p, act_idx) {
  t = p->task;

  for (v ∈ p->registered_views) {
    base_index = B(t,v->u,v->s) * (act_idx + 1) + D(t,v->u,v->s);

    // Set the view pointers
    *v->view_pointer = v->s->buffer + (base_index % v->s->buffer_size);
  }
  p->control_view += 1;
}
```

Figure 7.12: Advancing in stream buffers in the case of regular tasks.

x and 1 on stream y.

```
worker_function (void *parameters) {
  //Initialization code (same as before)

  // Scheduler loop
  while (GOMP_get_activation_range (p, &low_idx, &high_idx, max_aggregation))
    {
      for (base_idx = low_idx; base_idx <= high_idx; base_idx += block_size)
        {
          upper_bound = min (high_idx, base_idx + block_size - 1);
          GOMP_acquire_stream_accesses (p, base_idx, upper_bound);

          for (act_idx = base_idx; act_idx <= upper_bound; act_idx += 1)
            {
              // Body of the task
              view_y[0] = p->control_view[0].a * (view_X[0] + view_X[1] + view_X[2]);

              view_y += 1;
              view_X += 2;
              p->control_view += 1;
            }

          GOMP_release_stream_accesses (p, upper_bound);
        }
    }
}
```

Figure 7.13: Elimination of the view pointer advancement function call.

Now the code is simpler and we can discuss the third benefit of this optimization. This last version of the generated code is compatible with static analysis, and can result in essential optimizations, like vectorization. Indeed, the innermost loop no longer depends on the activation index itself and is free of pointer arithmetic. The compiler can simply generate code similar to the common pattern for the innermost loops presented on Figure 7.14, if the body of the task is vectorizable.

```
worker_function (void *parameters) {
  //Initialization code (same as before)

  // Scheduler loop
  while (GOMP_get_activation_range (p, &low_idx, &high_idx, max_aggregation))
    {
      for (base_idx = low_idx; base_idx <= high_idx; base_idx += block_size)
        {
          upper_bound = min (high_idx, base_idx + block_size - 1);
          GOMP_acquire_stream_accesses (p, base_idx, upper_bound);

          // Automatically vectorized version
          for (i = 0; i < upper_bound - base_index; i += 4)
            {
              // Vectorized body of the task
            }

          // Fall-back version (remainder of the iterations)
          for (i = max (0, i-4); i < upper_bound - base_index; i++)
            {
              // Original body of the task
            }

          GOMP_release_stream_accesses (p, upper_bound);
        }
    }
}
```

Figure 7.14: Automatically vectorized innermost loop of the worker function.

This case does occur in GCC, which automatically vectorizes the innermost loops of some of the simpler streaming task worker functions we generate.

This concludes our work on optimizing the task worker functions. Figure 7.14 represents the end goal of our optimization framework at this stage. This depends on the successful identification of regular tasks that do not belong to strongly connected components of the task graph.

7.4.3 Optimization of the Code Generated for the Control Program

We now focus our attention on the control program. The unoptimized version of the code generated for the evaluation of activation points, on Figure 7.4 on page 211, is not appealing and will probably not be further optimized by the compiler. If the workload of tasks is not high enough, the control program becomes the limiting factor for performance. From a scalability perspective, the control program is definitely one of the weakest links of our scheme because of the strong order requirements on its execution. However, if tasks are regular, we can generate efficient and concurrent code for the control program.

Serial optimizations

For the optimized code generation in the control program, we reuse example of Figure 7.1. We assume, however, that the second task construct generates regular streaming tasks only. The code generated in that case is presented on Figure 7.15, where the code of the activation point evaluation functions is inlined and the initialization part that remains unchanged is ignored for the sake of brevity.

```
#pragma omp parallel
#pragma omp single
{
  for (i = 0; i < K; ++i) {
    stream_id x;
    x = GOMP_stream_create_stream (sizeof(int), 1, 1);

    for (j = 0; j < N; ++j) {
//#pragma omp task firstprivate (j) output (x) // Static id: t1
      t1_firstprivate_data.j = j;
      dyn_t1 = get_dynamic_task (t1, [(W,x,1,1,1,0)]);
      GOMP_stream_stall (dyn_t1->control_stream, dyn_t1->activation_counter);
      dyn_t1->control_view->view_pointer =
        dyn_t1->control_stream->buffer +
        (dyn_t1->activation_counter % dyn_t1->control_stream->buffer_size);
      dyn_t1->control_view->view_pointer[0] = t1_firstprivate_data;
      GOMP_stream_commit (dyn_t1->control_stream, dyn_t1->activation_counter);
      dyn_t1->activation_counter += 1;

//#pragma omp task input (x) output (A[i] << y) // Static id: t2
      dyn_t2 = get_dynamic_task (t2, [(R,x,1,1,1,0), (W,A[i],1,1,1,0)]);
      dyn_t2->activation_counter += 1;
    }
    GOMP_stream_finish_stream (x);
  }
}
```

Figure 7.15: Regular tasks: evaluation of activation points inlined in the control program.

At this point, the resulting code does not look much better than before. First, we need to resolve the dynamic task, from the static task identifier and the set of input and output streams, the latter provided by the activation point passed as second parameter to the `get_dynamic_task` function calls. As the first task construct also has a firstprivate clause, we need to synchronize the communication with the control stream, which is quite cumbersome as it requires two runtime calls and computing the view position. The second task is simpler, but both task constructs also require identifying the dynamic streaming task with a call to `get_dynamic_task`.

First of all, we want to avoid requesting continuously the identification of the dynamic task. As we know what this function does (see Figure 6.27), we understand that it is only necessary here to create the task on its first call for a given static task identifier and a given set of input and output streams. In all other cases, its behaviour would qualify it as a *pure* function, with a behaviour entirely determined by its parameters. As both calls

have parameters that are either entirely static or constant within the innermost loop, we can hoist them out of the loop without impacting the execution.

For the second task construct, the activation counter is only defined within the innermost loop and never used aside from its incrementation statement. Its incrementation can therefore be hoisted out of the innermost loop as well, as shown on Figure 7.16. While static analysis would not allow these transformations in general, the knowledge of the side-effects of each of the operations we generate is the enabling factor.

```
#pragma omp parallel
#pragma omp single
{
  for (i = 0; i < K; ++i) {
    stream_id x;
    x = GOMP_stream_create_stream (sizeof(int), 1, 1);

    dyn_t1 = get_dynamic_task (t1, [(W,x,1,1,1,0)]);

    dyn_t2 = get_dynamic_task (t2, [(R,x,1,1,1,0), (W,A[i],1,1,1,0)]);
    dyn_t2->activation_counter += N;

    for (j = 0; j < N; ++j) {
      t1_firstprivate_data.j = j;
      GOMP_stream_stall (dyn_t1->control_stream, dyn_t1->activation_counter);
      dyn_t1->control_view->view_pointer =
        dyn_t1->control_stream->buffer +
        (dyn_t1->activation_counter % dyn_t1->control_stream->buffer_size);
      dyn_t1->control_view->view_pointer[0] = t1_firstprivate_data;
      GOMP_stream_commit (dyn_t1->control_stream, dyn_t1->activation_counter);
      dyn_t1->activation_counter += 1;
    }
    GOMP_stream_finish_stream (x);
  }
}
```

Figure 7.16: Calls to `get_dynamic_task` and counter incrementation hoisted out of the innermost loop.

The last control program code generation optimization is similar to data aggregation in task worker functions. It also reduces the number of stream synchronization calls and false sharing. In general, we do not have a loop body as easy as the current one, on Figure 7.16, where only our own generated code is left and the firstprivate communication requires no index computation. A systematic transformation, that requires no analysis to apply, is presented on Figure 7.17. However, the resulting code only achieves the first two objectives of the transformation, vectorization remaining just as hard in this case. This transformation can be replaced by loop blocking in simple cases, which enables vectorization.

```
#pragma omp parallel
#pragma omp single
{
  for (i = 0; i < K; ++i) {
    stream_id x;
    x = GOMP_stream_create_stream (sizeof(int), 1, 1);

    dyn_t1 = get_dynamic_task (t1, [(W,x,1,1,1,0)]);
    t1_acquired_index = -1;

    dyn_t2 = get_dynamic_task (t2, [(R,x,1,1,1,0), (W,A[i],1,1,1,0)]);
    dyn_t2->activation_counter += N;

    for (j = 0; j < N; ++j) {
      t1_firstprivate_data.j = j;

      if (t1_acquired_index < dyn_t1->activation_counter) {
        t1_acquired_index = dyn_t1->activation_counter + block_size;
        GOMP_stream_stall (dyn_t1->control_stream, t1_acquired_index);
        dyn_t1->control_view->view_pointer =
          dyn_t1->control_stream->buffer +
          (dyn_t1->activation_counter % dyn_t1->control_stream->buffer_size);
      }

      dyn_t1->control_view->view_pointer[0] = t1_firstprivate_data;
      dyn_t1->control_view->view_pointer += 1;

      if (t1_acquired_index == dyn_t1->activation_counter || j + 1 >= N) {
        GOMP_stream_commit (dyn_t1->control_stream, dyn_t1->activation_counter);
      }

      dyn_t1->activation_counter += 1;
    }
    GOMP_stream_finish_stream (x);
  }
}
```

Figure 7.17: Data aggregation for firstprivate communication in the control program.

Parallelized control program

When the control program contains parallelizing directives, especially when it relies on the `loop` worksharing construct rather than on the `single` construct, we cannot, always allow the concurrent execution of the loop. Indeed, when irregular tasks are found within the loop, and there are streams in which multiple tasks interleave their accesses, the parallelization is only correct if all of the activation points of such tasks are in the same thread. This condition, which is presented in Section 4.3 on page 116, comes from the fact that we store a state, in the $MaxSched(u, s)$ data structure, as shown on Figure 6.16. This precludes parallelization if concurrent writes can occur for the same direction u and the same stream s. A solution is to perform the *actual* evaluation with a separate thread. It only performs the partial sums of the bursts of successive activation points

in the sequential control program order and provides the stream access indexes to tasks through the control streams. The control program would send to such a helper thread the sequence of bursts observed, which can be ordered with respect to the sequential traversal order of the loop nest, based on induction variables. Note that as streams allow writing out of order as long as the write index is known in some way, this is valid. We leave the development of such a solution for future work, and for now conservatively reject the parallelization in such cases.

However, in the example of Figure 7.1, we remark that, as each task is connected to at least one stream with a local visibility scope, each iteration of the outermost loop defines its own local pipeline. As explained in Section 2.5.2 on Figure 2.13, the stream identifier x is different for each iteration of the outermost loop, which means that each iteration necessarily generates activations for different tasks. This qualifies this loop nest for parallelization, with respect to streaming tasks, as each iteration is independent, both for the x streams and for the $A[i]$ streams.

We do not list code for this case. The only difference is the replacement of the OpenMP single construct by a loop construct on Figure 7.17, requesting parallelization of the outermost loop.

We can now verify that the code evaluating the activation points does not present any kind of dependence conflicts if the outermost loop was parallelized. Each iteration of the outermost loop works on tasks that are local to the loop iteration and the only shared data structures are the static task identifiers, $t1$ and $t2$, and the array of streams A. It is therefore not even necessary to make the operations on the `activation_counter`s atomic, the loop is indeed data-parallel.

7.5 Providing an Infinite Continuous Buffer View

To avoid the circular buffer wrap-around checks, we provide a mechanism ensuring that a range of stream access indexes, once acquired with a call to `GOMP_acquire_stream_accesses`, can be accessed in the same way as an array. Instead of verifying at every access whether the access view pointer is still valid or recomputing the pointer with a call to `GOMP_stream_next`, as shown on Figure 7.11, this scheme enables the generation of code relying on pointer incrementation for traversing the range of stream elements, as shown on Figure 7.13.

Our scheme allows stream accesses to overshoot the stream buffer limit, and resolves the out-of-bounds accesses that result. In fact, we allocate a larger buffer than what is used by the stream circular buffer. The stream accesses that should have wrapped around, at the beginning of the buffer, occur in this additional memory space.

To illustrate this, let us consider the case of a producer to a stream that acquires a range of 8 elements in a stream, as illustrated on Figure 7.18. This producer can proceed as there are more than eight elements of unused space in the stream buffer. However, as the first stream index to which this producer is scheduled to write is located four elements before the buffer wrap-around, it would normally write the first four elements at the end of the buffer and the last four elements at the beginning. This means that there should be a re-computation of its view pointer for this stream in the middle of its activation

or activation range. Instead, we allow the producer to write outside of the buffer, in a space reserved for such occurrences, then we copy the offending last four elements to their rightful place. This copy is performed during the `commit` phase, where we check whether an overshoot has occurred and handle the case accordingly.

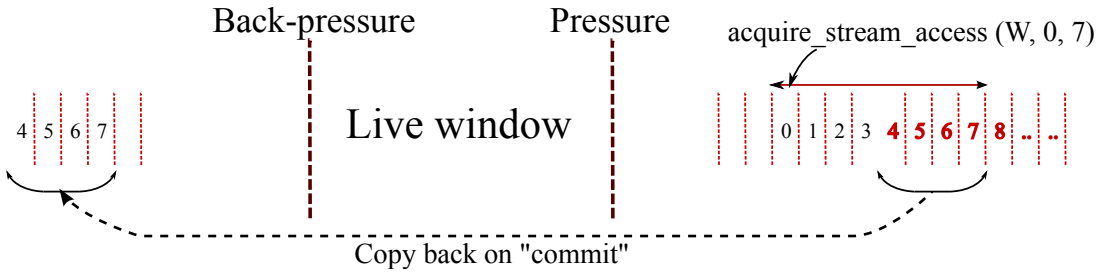


Figure 7.18: Allowing write accesses acquired as a range to overshoot the end of the stream buffer.

For a consumer requesting stream access indexes that cross the boundary of the buffer, we handle the case in a symmetrical way, but this time the copy operation needs to happen before the overshoot is allowed, as Figure 7.19 illustrates. We check if the set of accesses requested would normally wrap around the end of the stream buffer, and copy any missing data after the end of the stream buffer, therefore providing a continuous view of the range of stream elements that can be accessed through a simple C pointer or with a subscripted array.

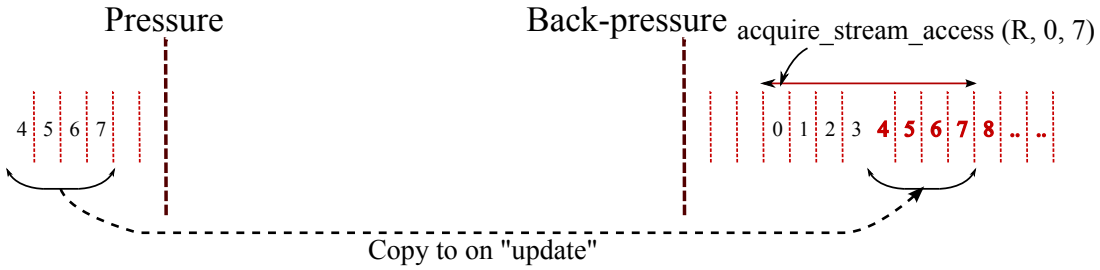


Figure 7.19: Allowing read accesses acquired as a range to overshoot the end of the stream buffer.

We present, on Figure 7.20, the algorithm used by our scheme. We integrate it to the aggregating functions for acquiring and releasing stream resources, `GOMP_acquire_stream_accesses` and `GOMP_release_stream_accesses`, but of course this could also be added directly in the `GOMP_stream_update` and `GOMP_stream_commit` synchronization primitives. Note that the copy operation is performed *after* the `update` operation completes, ensuring that the data was ready to be read in its original position, and that it is performed *before* the `commit` operation to ensure that the data at the beginning of the buffer is written before any consumer can access it.

To ascertain that there can be no conflict between copies to or from the beginning of the stream buffer or the overshoot area, one needs only consider that: (1) the stream access indexes being extended in the overshoot area are necessarily either read-only or write-only because the synchronization scheme applies to these indexes as well as to the beginning of the stream buffer, therefore ensuring exclusivity between producers and consumers to

```
GOMP_acquire_stream_accesses (p, act_idx_low, act_idx_high) {
  t = p->task;
  span = act_idx_high - act_idx_low;

  for (v ∈ p->registered_views) {
    // Compute the stream access index (same as before)
    // [...]

    // Request access to the highest accessible indexes
    if (v->u == R) {
      GOMP_stream_update (s, base_index_high + horizon);
      if ((base_index_low / v->s->buffer_size)
            < ((base_index_high + horizon) / v->s->buffer_size))
        {
          memcpy (v->s->buffer + v->s->buffer_size, v->s->buffer,
                    ((base_index_high + horizon) % v->s->buffer_size) + 1);
        }
    } else { // (v->u == W)
      GOMP_stream_stall (s, base_index_high + horizon);
    }
    // Set the view pointer in the stream buffer to the beginning
    *v->view_pointer = v->s->buffer + (base_index_low % v->s->buffer_size);
  }
}

GOMP_release_stream_accesses (p, act_idx) {
  t = p->task;

  for (v ∈ p->registered_views) {
    // compute the stream access index for the release (same as before)
    // [...]

    if (v->u == R) {
      GOMP_stream_release (s, release_index);
    } else {
      if ((v->current_base / v->s->buffer_size)
            < (release_index / v->s->buffer_size))
        {
          memcpy (v->s->buffer, v->s->buffer + v->s->buffer_size,
                    (release_index % v->s->buffer_size) + 1);
        }
      GOMP_stream_commit (s, release_index);
    }
  }

}
```

Figure 7.20: Preventing circular buffer wrap-around in acquired ranges of stream access indexes.

this area mirroring the beginning of the stream buffer; and (2) that any overwriting by another thread will rewrite the same data, but with possibly different lengths after the

end of the buffer. This overwrite cannot impact correctness.

Remark 7.1. *One of the main issues with this scheme is that it works best if there is a known bound for the highest possible horizon size for each stream, and how much aggregation may be enabled. Otherwise, it is impossible to decide how much overshoot space should be allocated. It is still possible to check that enough space is available in the same algorithm and trigger a stream buffer resizing event if the space is insufficient. In practice, if memory space is not an issue on the execution platform, and considering that this space will mostly not be used and therefore not waste cache space, we can simply allocate the double of the size of the circular buffer, and resize along with the buffer size. The overshoot can never exceed the buffer size without leading to a resource deadlock, which is then resolved, as a single request for more data that can fit the stream buffer is impossible to satisfy.*

7.6 Conclusion

In this chapter, we presented the code generation framework for our stream-computing extension to OpenMP. We provided a fall-back code generation scheme, that can be relied on when all static analyses fail, as well as optimized versions that are enabled by the static analyses presented in Chapter 5. The optimizations enabled by task regularity in worker threads not only result in reduced runtime overhead, but they also reduce the negative impact of streamization on the applicability of serial compiler optimizations.

In the current state of our implementation in the GCC compiler, we can only rely on ad hoc analyses because the OpenMP expansion pass occurs very early in the compilation flow. In fact it is performed just after parsing the program, which means that most of the static analysis information is unavailable. To overcome this issue, we have designed an intermediate representation [55] for OpenMP annotations that will be used, in future work, to preserve the high-level information through the compilation flow and postpone the OpenMP expansion until the latter compiler optimization passes.

Chapter 8

Experimental Evaluation

In this chapter, we present the experimental results we measured for three applications, FMradio, 802.11_a and 1D-FFT, on off-the-shelf x86 multicore machines. For each application, we first generate stream code by hand, relying on our runtime implementation, to evaluate the potential of our approach once the code generation is complete and all optimizations are applied. In a second step, we rely on the automatic generation of streaming code by our GCC prototype, from an OpenMP code annotated with our streaming extension. We present the speedups obtained by both methods relative to sequential code execution. For instance, we measure maximum speedups of 12.6×, 13× and 6.5× respectively for FMradio, 802.11_a and FFT on a 16 cores machine with hand-"compiled" code.

Nous présentons, dans ce chapitre, les résultats expérimentaux obtenus pour trois applications, FMradio, 802.11_a et la FFT unidimensionnelle, sur des architectures x86 grand public. Pour chacune de ces applications, nous avons écrit, à la main, une version streaming, faisant appel à notre système de runtime, qui nous permet d'évaluer le potentiel de notre approche dès lors que la passe de génération de code sera achevée avec toutes les optimisations que nous avons décrites précédemment. Nous évaluons une deuxième version, de chaque application, dans laquelle le code est écrit directement avec OpenMP, étendu pour le streaming, et compilée avec notre prototype basé sur GCC. Nous présentons les accélérations obtenues par ces deux versions relativement à une exécution séquentielle. Nous obtenons ainsi, par exemple, des accélérations allant jusqu'à 12.6×, 13× et 6.5× respectivement pour FMradio, 802.11_a et la FFT sur une machine dotée de 16 cœurs avec la version "compilée à la main".

8.1 Experimental Settings

The implementation of our streaming extension is under way in the `omp-stream` public branch of GCC. This implementation has followed the various stages [56–59] of the design of our extension which have converged towards the current extension, presented in Chapter 2. Earlier designs differ on some of the finer points, but the general approach, to make explicit the flow of data between tasks, has been the same throughout this evolution.

The results presented in this chapter correspond to the version we published in [56]. It only differs with this thesis on the semantics of broadcast over streams versus interleaving of read accesses. The discussion over this point in Section 2.5.5 reflects and motivates this choice, which mostly serves to simplify the work of the compiler by providing a missing piece of information. This is unlikely to have an impact on our performance results as our applications do not rely on dynamic interleaving patterns of read accesses to streams.

While the current implementation of the runtime and code generation framework is still incomplete, the essential building blocks are available to allow evaluating the performance potential of our streaming framework. Indeed, the runtime implementation is advanced enough to provide the necessary support for compiling by hand our applications without major limitations. Important parts of our algorithmic framework are still under development, in particular load-balancing and deadlock detection, but the essential part of our runtime is in the stream synchronization.

The code generation is less advanced, mostly because of design adjustments that have set us back along the way. In its current state, it is capable of generating code for FMradio and 802.11_a, but not for FFT. This difference is mostly due to earlier design choices, where the code generation was directly generating some of the functionality now provided by the runtime, in particular the evaluation of activation points and the forwarding of firstprivate communications. The relatively recent shift in design comes from a much stronger emphasis on dynamic codes, which has brought the requirement of the quiescence and stream buffer resizing schemes, leading to a stronger integration with the runtime. In prior versions, stream buffer sizes were set either statically or through execution parameters.

The comparison between the speedups measured on the hand-compiled and the automatically-compiled versions reflects the advancement of our code generation framework. As we see below, there is still a gap between the two versions, that we are currently working to fill.

8.1.1 Applications

For our experiments, we use three applications, FMradio, 802.11_a and FFT, briefly presented below.

FMradio is a kernel extracted from the GNU radio package[1]. This kernel was originally re-engineered to expose pipeline parallelism for the needs of the ACOTES project [1]. For this reason, the insertion of directives in the code required minimal effort. Some minor adjustments were required to remove needless state in some of the filters.

`FMradio` presents a high amount of data-parallelism, each filter with non-negligible load is in fact stateless. For this reason, and as discussed in Section 6.3.3, load balancing is easily achieved by exploiting the data-parallelism.

802.11a is an industrial code from Nokia [49], also a code from the ACOTES project, in which pipeline parallelism had been exposed by members of the project. However, in this case things were much more complicated. No care was taken in the original

[1] `http://gnuradio.org/redmine/projects/gnuradio`

code to avoid coding patterns that negatively impact parallelization and we had to extensively refactor the code in order to eliminate state in filters. After this step, annotating the program is straightforward.

This application is much more unbalanced, or rather balance is much more difficult to achieve, because some filters remain stateful.

1D FFT is based, for the parallelized versions, on our own unoptimized implementation of the Cooley-Tukey FFT algorithm, while the sequential version is a more optimized version which serves as a baseline of comparison in the StreamIt [68] benchmark suite.

The annotation of our algorithm is fairly straightforward and data-parallelism is naturally present. In fact, we used an unoptimized algorithm precisely because it does not impede data-parallelism.

Most importantly, we underline the fact that we do not evaluate FFT with a standard streaming approach. We evaluate the performance achieved on a *single* execution of FFT rather than the common streaming approach which uses a pipeline of successive FFTs applied to blocks of data in a stream. This is the case, as we discuss below, in the results reported for StreamIt in [26]. We use FFT in a single instance mode as an example of "non-streaming" code that can be more efficiently exploited with our streaming approach than other parallel constructs. Here, by *non-streaming* we mean that it does not fall in the usual category of streaming programs where a set of filters is applied to a regular stream of data.

While this small set of applications does not cover the full breadth of the expressiveness enabled by our streaming extension, we believe that it is sufficient to illustrate its potency. We still lack an application with truly dynamic dataflow patterns; as we discuss later, FFT is a dynamic task graph, but communication is entirely regular, FMradio and 802.11_a are inherently streaming applications, with static behaviour (synchronous dataflow in fact).

8.1.2 Experimental Platforms

We use the two following experimental platforms for our performance evaluation.

- 4-socket AMD quad-core Opteron 8380 (Shanghai) with 16 cores at 2.5 *GHz*, 64 *KB* L1 and 512 *KB* cache per core, 6 *MB* shared L3 cache and 64 *GB* of memory.
- 4-socket Intel hexa-core Xeon E7450 (Dunnington) with 24 cores at 2.4 *GHz*, 32 *KB* L1 cache per core, 3 *MB* L2 cache shared by 2 cores, 12 *MB* shared L3 cache and 64 *GB* of memory.

These targets are respectively called *Opteron* and *Xeon* in the remainder of this chapter.

8.2 Software defined radio: FMradio

As a synchronous dataflow application, FMradio fits the stream programming models. The kernel annotated with OpenMP and our streaming constructs is presented on Figure 8.1. The application is structured around a stream of data, read from a file in the main

program. This operation could also be made in a new task, but the file read operation predicates the main `while` loop. For this reason the task would require a lastprivate clause, which would serialize its execution with the control program and would only result in performance degradation.

```
// Implement delay on fm_qd stream
#pragma omp task output (fm_qd << fm_qd_pre[maxtaps_minus_one]) private (i)
  for (i = 0; i < maxtaps_minus_one; ++i)
    fm_qd_pre[i] = 0;

// Implement delay on ffd stream
#pragma omp task output (ffd << view_3[lp_3_taps_minus_eight]) private (i)
  for (i = 0; i < lp_3_taps_minus_eight; ++i)
    view_3[i] = 0;

  // Main loop
  while ((16 == fread (read_buffer, sizeof(float), 16, input_file))) {

#pragma omp task firstprivate (read_buffer) output (fm_qd << view8[8])
    for (i = 0; i < 8; i++)
      fm_quad_demod (&fm_qd_conf, read_buffer[2*i], read_buffer[2*i + 1], &view8[i]);

      for (i = 0; i < 8; i++) {
#pragma omp task input (fm_qd[0] >> view_11[1]) output (band_11) num_threads (3)
        ntaps_filter_ffd (&lp_11_conf, 1, &view_11[diff_11], &band_11);

#pragma omp task input (fm_qd[1] >> view_12[1]) output (band_12) num_threads (3)
        ntaps_filter_ffd (&lp_12_conf, 1, &view_12[diff_12], &band_12);

#pragma omp task input (fm_qd[2] >> view_21[1]) output (band_21) num_threads (6)
        ntaps_filter_ffd (&lp_21_conf, 1, &view_21[diff_21], &band_21);

#pragma omp task input (fm_qd[3] >> view_22[1]) output (band_22) num_threads (6)
        ntaps_filter_ffd (&lp_22_conf, 1, &view_22[diff_22], &band_22);

#pragma omp task input (band_11, band_12) output (resume_1)
        subctract (band_11, band_12, &resume_1);
#pragma omp task input (band_21, band_22) output (resume_2)
        subctract (band_21, band_22, &resume_2);

#pragma omp task input (resume_1, resume_2) output (ffd)
        multiply_square (resume_1, resume_2, &ffd);
      }

#pragma omp task input (fm_qd[4] >> view_2[8]) output (band_2)
      ntaps_filter_ffd (&lp_2_conf, 8, &view_2[diff_2], &band_2);
#pragma omp task input (ffd >> view_3[8]) output (band_3)
      ntaps_filter_ffd (&lp_3_conf, 8, view_3, &band_3);

#pragma omp task input (band_2, band_3) output (out)
      stereo_sum (band_2, band_3, &out.a, &out.b);

#pragma omp task input (out)
      {
        output_short[0] = dac_cast_trunc_and_normalize_to_short (output1);
        output_short[1] = dac_cast_trunc_and_normalize_to_short (output2);
        fwrite (output_short, sizeof(short), 2, output_file);
      }
  }
```

Figure 8.1: Annotated code of the main loop in FMradio.

We re-print, on Figure 8.2, the task graph graph of FMradio from Section 6.3.3, originally on Figure 6.23, for the convenience of the reader. This figure shows the structure of the application and helps understanding the code. There is a total of 14 task constructs,

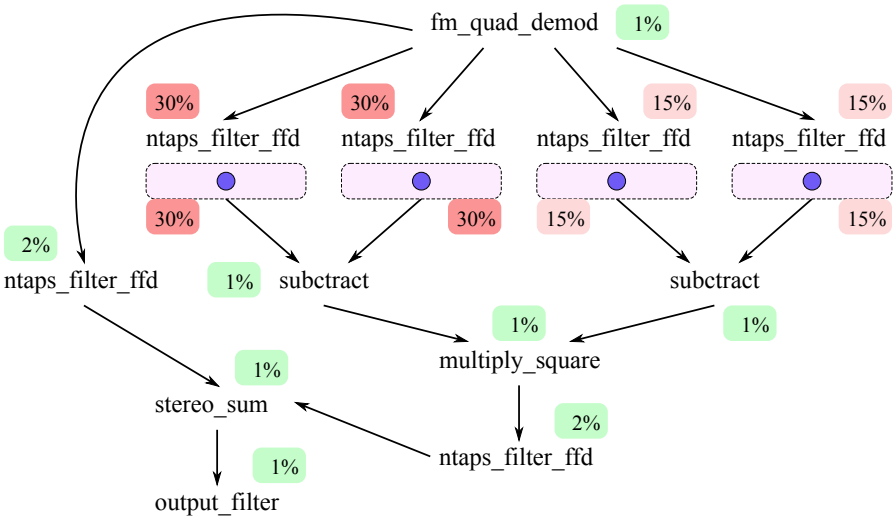


Figure 8.2: FMradio task graph.

12 are represented on the task graph.

The Tasks 1 and 2 on Figure 8.1 implement the delay pattern, presented in Section 2.5.5. The first task produces sufficient 0 values to prime the `fm_qd` stream. Note that this stream is a multi-producer stream, filled in by Tasks 1 and 3. The interleaving only occurs once, so this is considered as a delay on stream `fm_qd` rather than an interleaving. Task 2 also introduces a delay on stream `ffd`, while this stream is primarily produced by Task 12. These two tasks are not represented on the task graph of Figure 8.2.

The main loop of this program reads data from a file in a buffer, by chunks of 16 data elements. This `read_buffer` is passed on to Task 3 through a firstprivate clause. This task processes the data and produces 8 elements in its output window, `view8`, which is output on the `fm_qd` stream. On the task graph, Figure 8.2, this task is at the very top, marked with the name of the work function, and it provides data to five other tasks. This is achieved by a stream broadcast array pattern (see Section 2.4.3). The stream variable, `fm_qd`, is declared as an array of five elements. In the `output` clause of Task 3, it appears non-subscripted, which means that all consumers see the same data. Tasks 4 to 7, as well as Task 11, in the code on Figure 8.1, use their own "stream" in the array, differentiated by the subscript, to read from this broadcast stream. The code generated for this pattern uses only one physical stream with multiple consumers.

One of the features introduced by our extension is the `num_threads` clause for streaming tasks. This clause specifies the number of worker threads that should execute the activations of the task. As our runtime load-balancing scheme, presented in Section 6.3.3 is not yet available, we performed the load-balancing by hand, specifying the appropriate number of threads for the four most computationally intensive tasks. The code presented was used on the Xeon platform, where 24 hardware threads are available, and it lowers the execution time of the slowest thread to 5% of total execution. This corresponds to a maximum 20× speedup.

The code presented on Figure 8.1 is, with minor cosmetic modifications, the code that is used in the compiler-generated version. We did, however, adjust the broadcast

array `fm_qd` to reflect the current state of the streaming extension, but this is the only semantical change.

The results measured on this benchmark, using a hand-compiled version show a 12.6× speedup on Opteron (16 cores) and a 18.8× speedup on Xeon (24 cores). The Xeon result comes quite close to the limit speedup of 20× discussed above.

The speedup obtained with our compiler prototype is lower, with only a 10.5× speedup on Opteron, a 20% lower result than with the hand-generated code. *We do not present, at this time, the results from the compiler generated code on Xeon due to machine availability issues.*

We stress the importance of data-parallelization for load-balancing. Pipeline parallelism alone, with `num_threads = 1`, only provides a 3.1× speedup on Opteron and 2.9× speedup on a desktop machine equipped with an Intel Core2 Quad Q9550 with 4 cores at 2.83GHz. This is to be expected as the slowest filter, as shown on Figure 8.2, accounts for roughly 30% of the execution time, therefore bounding speedup by 3.3×.

8.3 Wifi: 802.11a

We do not review the details of this application as its code is excessively long and does not present any interesting new coding patterns. It is also a synchronous dataflow program, with a static behaviour that guarantees the regularity of all tasks.

For this application, the hand generated code reaches 13× on the Opteron platform (16 cores) versus a more modest 6× for the compiler generated code on this same platform.

We measured a speedup of 14.9× on Xeon (24 cores) with hand generated code, which shows the limits to scalability on this application. As discussed in Section 8.1.1, we were unable to remove the state in some of the filters. These cannot be data parallelized. For this reason, we do not expect performance to improve much further in the current implementation of this application.

8.4 1D FFT

The two applications presented above are inherently streaming applications, that perfectly fit the synchronous dataflow model. This is not the case for FFT. While the computation of multiple, independent, FFTs is an embarrassingly parallel problem where each thread can work on its own computation, the parallelization of a single FFT computation presents a limited amount of parallelism. When the size of the problem increases, the amount of parallelism increases in the inner stages. However, this additional parallelism is only available for decreasing portions of the total work. As we argue below, the overall parallelism available only scales logarithmically with the size of the problem.

We measure "real" speedups: the algorithm used as the sequential comparison baseline is the fastest one and differs from the parallelized ones. In order to expose parallelism, we privatize the data to eliminate "false" data-dependences and we also need to use the Cooley-Tukey algorithm with the reorder stages. Using computed array subscripts is faster for sequential implementations, but it results in array accesses in a much wider range for each iteration within a stage of the computation and hence increases false-sharing between

concurrent threads. These two differences introduce a lot of overhead, so we use instead an algorithm optimized for serial execution.

FFT cannot be compiled by our prototype and we only present results for hand generated code. We hand-tuned the amount of parallelism to achieve the best results, as explained below, and this leads to different configurations for different FFT sizes. In the results presented on Figures 8.5 and 8.6, we show separately the speedups achieved with a single configuration for all sizes and with the best configuration for each size. The single configuration version uses the configuration best suited for large sizes of FFT, therefore leading to the highest speedups only for a few of the FFT sizes.

We present the code of our implementation, as well as the resulting task graph, on Figure 8.4. The global structure is a linear pipeline of filters using two dynamic pipelines with an array of streams, `STR[]`. The first task construct, represented on the left of the task graph, serves as a source of data for the FFT. The second and third task constructs implement the dynamic pipeline of filters pattern, as presented in Section 2.5.2, and build the two dynamic pipelines of reorder stages and DFT stages. Finally the last task construct is a sink task that prints the results of the computation.

The second and third constructs are the most interesting ones. First, we can notice that we can easily apply our analysis scheme, presented in Section 5.3.2, on the subscripts in the array of streams `STR[]`, by eliminating the common subexpression `j` for the reorder stage and `j + log(N)` for the DFT stage, between the two streaming clauses of each task, which shows the regularity of each streaming task generated by these two constructs. Furthermore, we note that the horizons and rates are constant, for each stream in the array `STR`, but they vary across the different filters in the dynamic pipelines, as shown on the task graph at the top of Figure 8.4. This accounts for varying degrees of available data parallelism within the filters, which is a well-known issue for FFT.

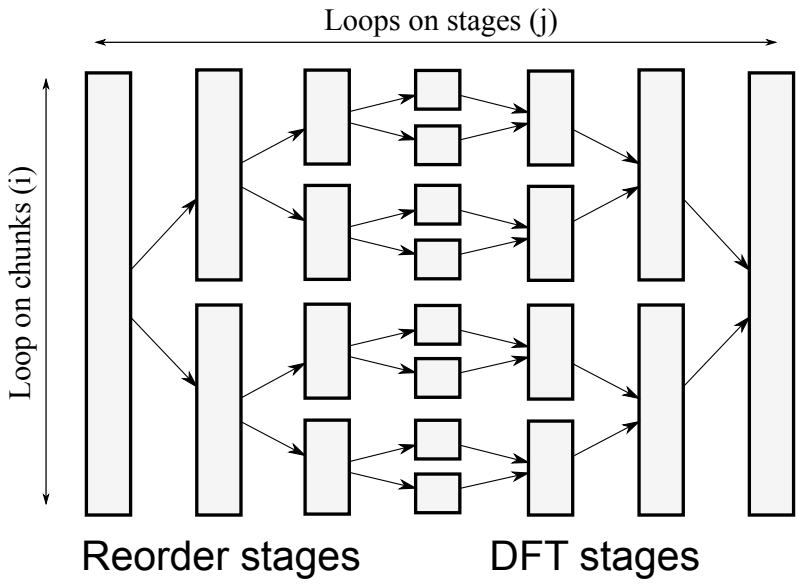


Figure 8.3: Data-flow graph for FFT.

Figure 8.3 presents the data-flow graph for our FFT implementation. The dependence

patterns show that data-parallelism is available in each stage, or vertical slice, of the data-flow graph. It is possible to exploit data-parallelism by parallelizing the loops on chunks, with the induction variable i, or to exploit task-parallelism, each chunk being exploited by an independent task, spawned as soon as its dependences are satisfied. This greedy parallelization pattern maximizes the amount of parallelism exploited. Pipelining relaxes the synchronization and enables wavefront parallelization. We control the granularity by varying the depth of the pipeline, thus changing the number of times the data is split.

Though we only present one version of the FFT algorithm, on Figure 8.3, we present, on Figures 8.5 and 8.6, the speedups measured of five parallelized versions compared to our sequential baseline:

Mixed pipeline and data-parallelism corresponds to our streaming implementation where we exploit both types of parallelism, with possibly multiple worker threads per task. On the data-flow graph (Figure 8.3) this corresponds to a wavefront traversal where we execute multiple chunks concurrently once we reach a given depth. The main idea is that pipeline parallelism does not hurt locality, but it is insufficient in and of itself. By exploiting pipeline parallelism only up to a given depth, we can reach a chunk size that properly fits in the shared cache of a processor, at which point we switch to exploiting data-parallelism.

Pipeline parallelism is our streaming implementation where a single worker thread is enabled per task: no data parallelism. This is a wavefront traversal of the data-flow graph, where all chunks in a vertical slice are serially executed. However, once a chunk is executed, both the chunk below and the chunks that get their dependences satisfied in the next stage (to the right on Figure 8.3) become executable.

Data-parallelism (OpenMP loops) corresponds to a simple data-parallel implementation exploiting the loop on chunks, with the induction variable i in our implementation, on Figure 8.4. In this case, we execute concurrently all the chunks belonging to vertical slices of the data-flow graph, with a barrier at the end of each stage.

Task-parallelism (OpenMP tasks) is an implementation exploiting task parallelism, but we limit the partition at a given depth, which improves performance by limiting the scheduling overhead. The traversal of the data-flow graph is only limited, in this case, by the data-flow dependences. This minimizes synchronization, but it leads to poor locality.

Task-parallelism (Cilk tasks) is similar to OpenMP tasks, but we do not limit the depth, therefore achieving a maximal partition, which the coding pattern advocated in Cilk [24]. This shows that task scheduling techniques alone are insufficient to properly exploit a maximal partition of the 1D FFT.

Speedups for `FFT` are presented on Figures 8.5 and 8.6. Combined pipeline- and data-parallelism achieve the best results, compared to pure data-parallelism or pure pipelining. We report two sets of results for each target: the single configuration results (top graph) correspond to speedups obtained when using the same tuning parameters (number of threads, granularity, etc.) for all the data sizes and for all the code versions; the best

configuration results (bottom graph) correspond to the optimal performance achieved with the best configuration for each individual data size point. The size of the machines and the associated cost of inter-processor communication set the break-even point around vectors of 256 elements and more. The maximum speedup of 6.5× on Opteron and 4.85× on Xeon is achieved for vectors of 2^{20} elements.

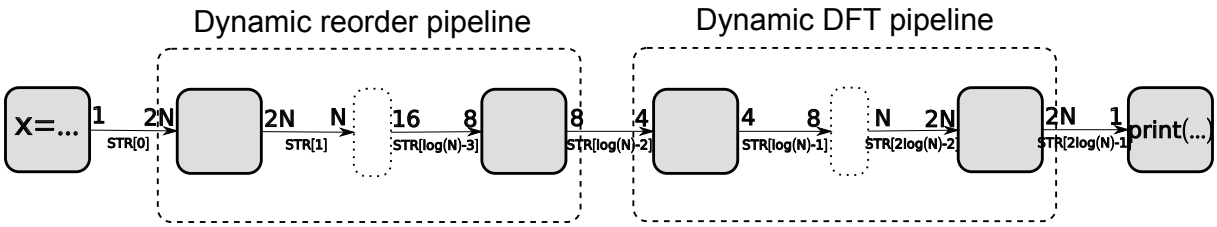
The analysis of the Opteron results, presented on Figure 8.5, shows a few noticeable points of interest. The most apparent one occurs in the single configuration graph (top) when the size of the FFT increases from 2^{19} to 2^{20}, in which case the speedup almost doubles. This is mostly due to the serial execution becoming significantly slower because it no longer fits in the L3 cache. Indeed, as the access patterns of FFT are non-linear, as soon as the data array no longer fits in one of the cache levels, the computation will necessarily experience increased memory latency as it needs to fetch data from lower levels of cache or from main memory. The same behaviour is observed on the Xeon platform, on Figure 8.6, where this occurs when the L2 cache is no longer sufficient to fit the FFT data, between FFT sizes of 2^{18} and 2^{19}. It also appears, to a less noticeable degree, between sizes 2^{12} and 2^{13} as the L1 cache size is exceeded.

Finally, the speedups of our streaming implementation (blue) decrease, on both machines, from FFT sizes 2^{20} to 2^{22}. This same phenomenon is observed in other places; it is very apparent on the bottom graph for Opteron results on Figure 8.5 when the FFT size increases from 2^{11} to 2^{13} of from 2^{17} to 2^{19}. All data points in each range actually correspond to the same configuration, before a new configuration becomes better suited as the size exceeds a threshold. This is very clearly visible on both "single configuration" graphs (at the top of both figures), as the speedups form a wave that ebbs once the "optimal" size is exceeded for the configuration. On the bottom graphs, the three *local* speedup optima occur for FFT sizes 2^{11}, 2^{17} and 2^{20} on Opteron and for sizes 2^9, 2^{16} and 2^{20} on Xeon. They exhibit the three configurations used to compose these graphs.

On a PRAM machine, is is possible to evaluate the theoretical speedup that can be reached and it is comprised, for a given problem `size`, between $\log_2(\text{size})/2$ and $(\log_2(\text{size}) + 1)/2$. This leads us to conclude that our highest speedup, reached for a problem size of 2^{20}, the highest PRAM speedup would be between 10× and 10.5×. Our 6.5× speedup on Opteron is therefore close to 65% of the theoretical optimum. In a certain sense it is higher, as we recall our sequential baseline is an optimized algorithm while the streaming results use an unoptimized one, but this is the natural price to pay for exposing parallelism. Note that the speedup of 5.9× achieved on the FFT of size 2^{17} represents close to 70% of the theoretical speedup on Opteron. This shows that non-streaming applications can also be efficiently executed with our extension.

8.5 Concluding Remarks

We evaluated our stream-computing framework on three applications: two streaming applications, FMradio and 802.11_a, and a non-streaming application, FFT. With speedups of up to 18.8× and 14.9× for FMradio and 802.11_a on a 24 cores platform, these applications show that our framework is capable of efficiently exploiting dataflow applications. The performance measured on FFT, a non-streaming application, shows that




```
float x, STR[2*(int)(log(N))];

for(i = 0; i < 2 * N; ++i)
#pragma omp task output (STR[0] << x)
  {
    x = (i % 8) ? 0.0 : 1.0;
  }

// Reorder stages
for(j = 0; j < log(N)-1; ++j) {
  int chunks = 1 << j;
  int size = 1 << (log(N) -j + 1);
  float X[size], Y[size];

  for (i = 0; i < chunks; ++i)
#pragma omp task input (STR[j] >> X[size]) output (STR[j+1] << Y[size])
    {
      // Y[0..size-1] = reorder (X[0..size-1]);
    }
}

// DFT stages
for(j = 1; j <= log(N); ++j) {
  int chunks = 1 << (log(N) - j);
  int size = 1 << (j + 1);
  float X[size], Y[size];

  for (i = 0; i < chunks; ++i)
#pragma omp task input (STR[j+log(N)-2] >> X[size]) \
                    output (STR[j+log(N)-1] << Y[size])
    {
      Y[0..size-1] = compute_DFT (X[0..size-1]);
    }
}

for(i = 0; i < 2 * N; ++i)
#pragma omp task input(STR[2*log(N)-1] >> x) input (stdout) output (stdout)
  {
    printf ("%f\t", x);
  }
```

Figure 8.4: FFT implementation using dynamic task pipelines and the corresponding task graph.

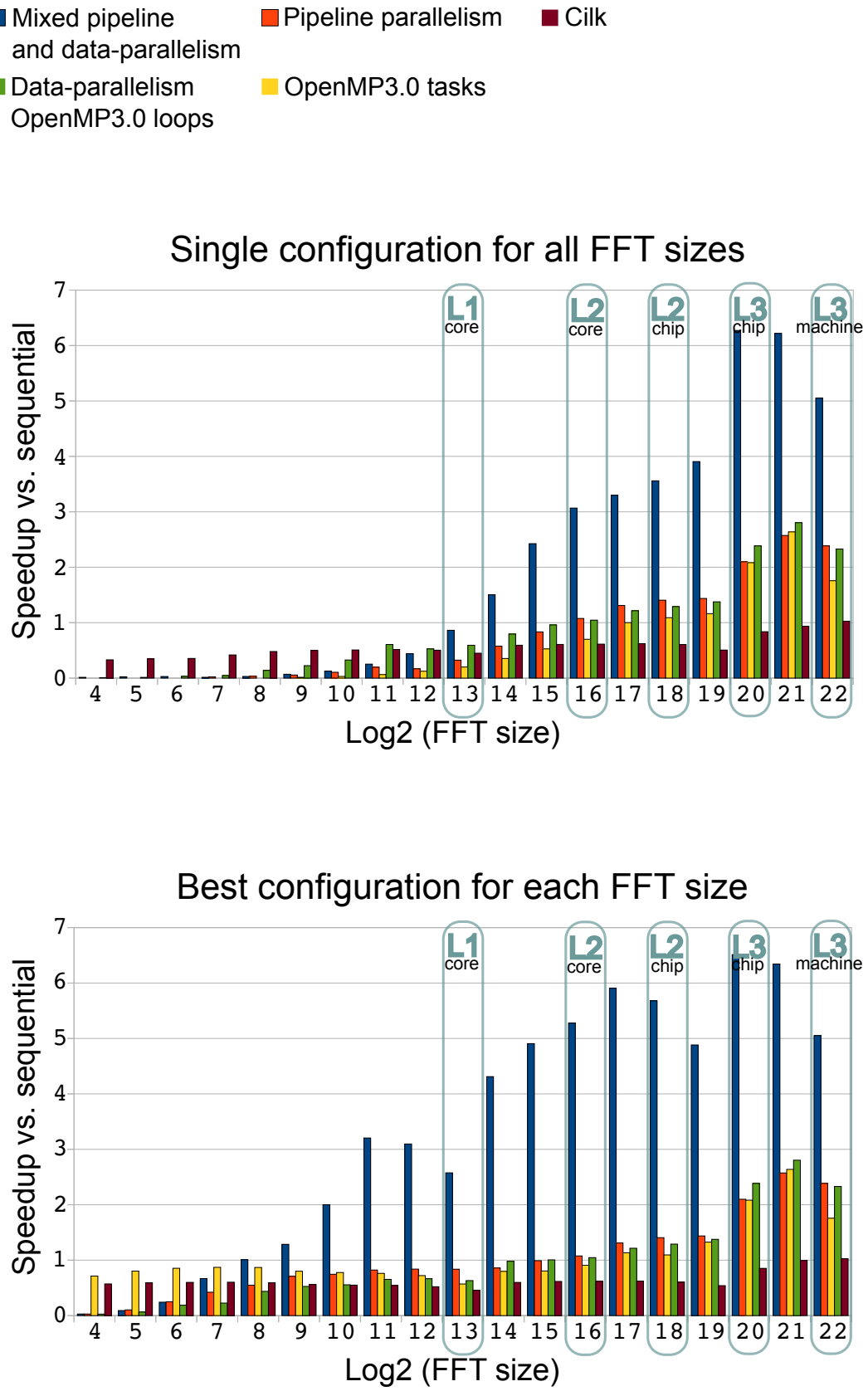


Figure 8.5: FFT performance on Opteron.

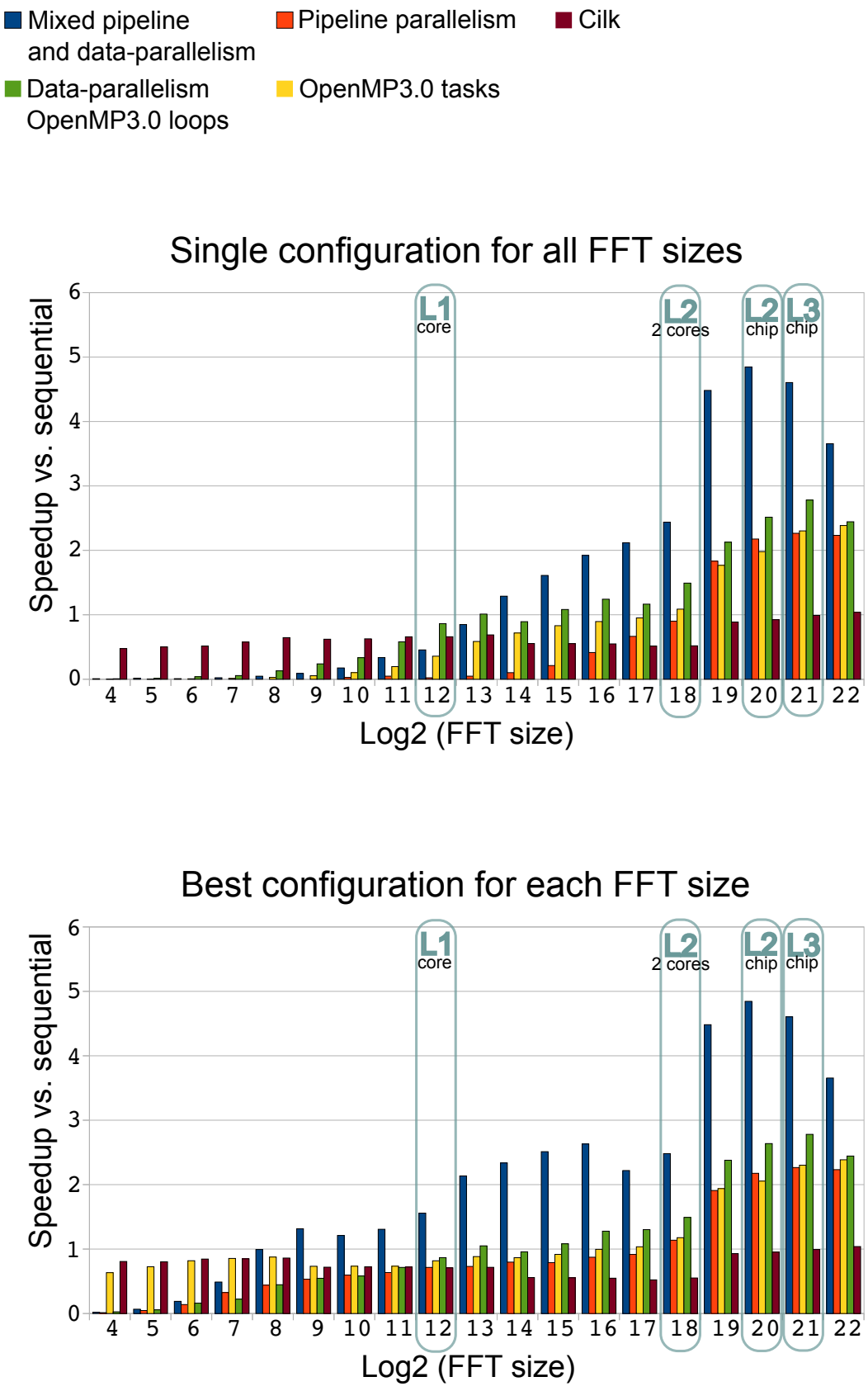


Figure 8.6: FFT performance on Xeon.

our streaming framework is able to exploit parallelism on multicore architectures more efficiently than data- or task-parallelism, showing more than twice higher speedups. The best speedup, for a very large one million elements configuration, is 6.5×, about 65% of the theoretical 1D FFT speedup on a PRAM machine. On very small sizes a slowdown is observed, but the break even point with sequential execution occurs for a FFT size of 256 elements, much earlier than for other forms of parallelism. This shows the applicability of our approach for general-purpose parallelization.

While still preliminary, the results presented in this chapter validate our approach and motivate the development of this streaming compilation in GCC, as well as the integration of our streaming extension in upcoming revisions of the OpenMP specification.

Chapter 9

Conclusion

To conclude this dissertation, we review the main contributions and results, we present ongoing and future work opportunities and we discuss some perspectives for our streaming framework.

9.1 Contributions

This thesis makes five main contributions: an extension of OpenMP to support stream computing, a formal model to prove its efficient implementation, a set of algorithms for the necessary runtime, a code generation framework with a prototype implemented in GCC and experiments.

Stream-Computing in OpenMP [56] The extension presented in Chapter 2, incrementally enables stream programming in OpenMP. This extension does not break the semantics of existing OpenMP programs and is compatible with existing compilers, notably with GCC. We provide a detailed description of the extension's semantics and of the most important coding patterns it enables.

A key property in our streaming extension is that, unlike other streaming frameworks, it does not require regular and static communication patterns, but can accomodate any type of dynamic connections, with a fully dynamic task graph, and allows streaming tasks to be nested in arbitrary control flow, thus resulting in a high level of expressiveness. Programmers expose task parallelism while providing the compiler with the dataflow information required to generate code that dynamically builds a streaming program. It allows to seamlessly exploit task, pipeline and data parallelism, while improving locality and fostering on-chip communication, therefore helping to reduce the severity of the memory wall.

The Control-Driven Data-Flow (CDDF) Model of Computation In order to prove that programs written with our streaming extension can be efficiently executed and benefit from essential properties that improve productivity, such as determinism, we developed in Chapter 3 a new model of computation that accounts for both control flow and data flow. By using a control program to deterministically schedule data in streams,

the scheduling of task activations is simplified and the synchronization of streams can be reduced to a form that allows an efficient implementation.

We prove that CDDF programs benefit from deadlock and functional determinism, that they are serializable under conditions that are satisfied by default in the current OpenMP semantics, and that statically decidable conditions are sufficient to provide freedom from spurious deadlocks, introduced by the synchronization scheme, or even in some cases from functional deadlocks, resulting from insatisfiable cycles of flow dependences.

To bring the CDDF model closer to our OpenMP streaming extension, Chapter 4 relaxes some of the restrictions made in Chapter 3. We prove that the CDDF model is compatible with bounded memory execution and that some level of parallelism can be exploited in the control program without loss of determinism.

Chapter 5 further shows that these results apply to our OpenMP extension and lays the groundwork for a static analysis framework for streaming programs. We address the question of statically verifying deadlock-freedom conditions and we provide a scheme for adjusting existing dataflow analyses to enable the generation of optimized code.

Runtime Deadlock Detection and Atomic Operation-Free/Fence-Free Synchronization Algorithms To address the issue of possibly remaining deadlocks, we designed a runtime deadlock detection algorithm, capable of detecting and resolving resource deadlocks and of providing a much needed support for program debugging. When the deadlock cannot be resolved, it can provide precise information on the tasks and streams involved in the deadlock as well as on the stream accesses that could not be satisfied. Along with the deadlock determinism property of our streaming programs, this simplifies testing and deadlock identification. In the absence of shared memory communication, no data races are possible. Debugging and testing such programs is deterministic and their behaviour is target independent.

In Chapter 6, we present a synchronization algorithm for streams that makes the best use of the Total Store Order (TSO) and x86 consistency models, providing stream synchronization with neither atomic operations nor memory fences. We show that our algorithm effectively uses all of the strengths of the TSO model, exploiting all non-relaxed ordering constraints. We further provide the equivalent algorithm under some of the most relaxed memory models, including IBM POWER and Relaxed Memory Order (RMO). We show that the overhead introduced by this synchronization scheme is significantly lower than scheduling lightweight tasks. A partial discussion of this work is published in [48].

Code Generation and Prototype Implementation in GCC [56,58,59] Chapter 7 presents our code generation algorithms. Designed for a complete integration in GCC's implementation of OpenMP, a simple code generation algorithm is provided that requires no static analysis, in conformance with the OpenMP expansion pass in GCC.

In addition to the default code generation scheme, we provide algorithms for generating optimized streaming code, by relying on the results of our static analysis framework detailed in Chapter 5.

While still incomplete, the implementation of our prototype of the code generation pass has reached a point where full applications like FMradio and 802.11_a can be compiled.

Experimental Results We use three applications, FMradio, 802.11_a and FFT, presented in Chapter 8. We first generate stream code by hand, relying on the runtime implementation, to evaluate the potential of our approach once the code generation is complete and all optimizations are applied. In a second step, we rely on the automatic generation of streaming code, by our GCC prototype, from an OpenMP code annotated with the streaming extension.

We present the speedups obtained by both methods when compared to sequential code execution, except for FFT which could not be automatically generated at this stage, showing significant speedups on several off-the-shelf multicore machines. For instance, using an AMD quad-core Opteron 8380 (Shanghai) with 16 cores at 2.5 *GHz*, we measure the following speedups for FMradio: 12.6× speedup with hand-compiled code and 10.5× with automatic generated code. For 802.11_a hand generated code reaches 13× versus a more modest 6× for the compiler generated code.

In the case of FFT, our results are obtained for a single FFT computation rather than the usual pipeline of successive FFT computations that better fit the streaming paradigm. The best performance, for a very large one million elements configuration, is a 6.5× speedup, close to 65% of the theoretical speedup on a PRAM machine. On very small sizes a slowdown is observed, but the break even point with sequential execution occurs for a reasonably small FFT size of 256 elements. This shows the applicability of our approach for general-purpose parallelization.

9.2 On-Going and Future Work

Many new ideas and opportunities for further research have popped up all along this thesis. Some of these ideas are already being investigated and in the development stage, while others present exciting perspectives and will provide the ground for new interesting studies.

Two compiler-related developments have been spurred by this thesis. The first one is related to one of our early approaches [57–59] to streamization, where we used streams as a replacement for memory expansion in the loop distribution pass of GCC, and to the decoupled software pipelining technique developed by Rangan et al. [62]. This development [45], led by Feng Li, a PhD student at INRIA, relies on the single static assignment (SSA) form to extract pipeline parallelism from sequential code, providing a partitioning algorithm based on the hierarchy of control dependences, and relies on our OpenMP streaming extension as a backend for code generation.

The second one stemmed from the need to rely on static analyses for the generation of optimized streaming code. It revealed the difficulty of obtaining static analysis information at the very early stages of the compilation flow in GCC, where OpenMP expansion occurs. We developed an intermediate representation scheme to preserve the high-level information of OpenMP constructs through the compilation flow, in order to perform both the OpenMP and the streaming expansion at later stages of the compilation flow, where the necessary static analysis information is available. This has many additional benefits as it avoids obfuscating the code, because the expansion introduces many opaque function calls and generates code that is not simple to analyze, before serial optimizations occur. An early paper [55] covers this topic.

In addition to this on-going work, many opportunities are open for further research. On the applications side, as our model allows to dynamically adopt new dataflow paths, applications from the fields of control systems and signal processing with dynamic mode switching, like dynamic video encoding, naturally fit in our streaming extension. The implementation of applications like low latency image processing for car and plane controllers present characteristics that could exploit our model.

Some questions are still open in the compiler and runtime, in particular with respect to the static analysis of streaming programs and the possibility to enable some of the optimizations we could not allow. We mentioned many such cases along the dissertation, for example with respect to the applicability of data aggregation in strongly connected components of the task graph, which could use a mixed compiler and runtime approach, where the runtime adapts the aggregation factor to the dynamic delay available in the streams involved in a cycle. The questions of refining our over-approximation of the static task graph and of providing a reversible load balancing strategy, which would improve the efficiency of exploiting fully dynamic task graphs, present some interesting challenges.

A possible use of our extension is as a target for the compilation of higher-level dataflow languages, for example for the generation of parallel, desynchronized, code from LUSTRE. An attempt in this direction by Léonard Gérard revealed that modular compilation support was required. While not yet implemented, this is supported by our compilation scheme.

As mentioned in Chapter 5, one of the most ambitious ideas is to use array dataflow analysis not only for deciding task regularity, but also to analyze the patterns of stream accesses. Powerful transformations like de-multiplexing the interleaving of data to privatize streams or re-scheduling streams for purposes like task fusion or loop nest optimizations would be within grasp. Along with a framework for detecting synchronous dataflow invariants, this may bring the highest performance returns.

Finally, as our communication scheme naturally maps onto MPI channels, tasks that do not rely on shared memory can be executed on different compute nodes. Extending the work by Millot et al. in the STEP project [47], which relies on the source-to-source optimizing compiler PIPS [35] to map OpenMP applications on distributed platforms, would present an interesting challenge.

9.3 Perspectives

One of our goals is to integrate our streaming extension into upcoming revisions of the OpenMP specification. We presented an early version of our work at IWOMP [54], the main venue for discussion on OpenMP developments. This is a lengthy process, where a full implementation is required before such an extension can be taken into consideration, and we plan on pursuing this direction as soon as our compilation framework is complete. This is of course not the only avenue for promoting our model. Indeed, our extension transparently maps on any task-parallel programming model, with some adjustments and accounting for the requirement of task causality which underlies many of our results. We are especially interested in the upcoming OpenHMPP [60] standard, where our streaming framework would naturally fit. Eventually this can extend to any language that uses tasks, like Cilk [24].

Conclusion en français

En conclusion de cette thèse, nous passons tout d'abord en revue les principales contributions et résultats, nous présentons les travaux en cours et les opportunités de recherche que cette thèse ouvre et enfin nous discutons des perspectives de notre modèle de streaming.

Contributions

Cette thèse apporte cinq principales contributions : une extension au langage OpenMP pour permettre le streaming, un modèle formel utilisé pour prouver l'efficacité de son implantation, un ensemble d'algorithmes de runtime, un prototype de générateur de code implémenté dans le compilateur GCC et un ensemble d'expériences visant à valider ces techniques.

Le stream-computing dans OpenMP [56] L'extension de langage que nous avons présentée au chapitre 2 introduit incrémentalement la programmation en streaming dans OpenMP. Cette extension ne casse pas la sémantique des programmes OpenMP préexistants et est compatible avec les compilateurs existants, notamment avec GCC. Nous décrivons en détail la sémantique de cette extension au travers d'exemples et des principaux schémas de programmation rendus possibles.

L'une des propriétés essentielles de ce modèle de programmation par stream est que, contrairement à d'autres modèles, il n'est pas nécessaire d'écrire des programmes communiquant de manière régulière et statique. En effet, notre modèle permet d'utiliser toute forme d'interconnexion entre tâches et de construire un graphe de tâches entièrement dynamique, où les tâches peuvent se trouver dans du contrôle de flôt arbitraire, ce qui confère à cette extension une expressivité complète. Le programmeur doit exposer le parallélisme de tâches en fournissant au compilateur les informations sur le flôt de données pour chaque tâche. Ces informations sont nécessaires, et suffisantes, pour permettre la génération du code qui construit dynamiquement le programme de stream. Il est possible d'y exploiter simultanément le parallélisme de tâches, de pipeline et de données, tout en améliorant la localité de cache et en favorisant les communications locales, qui ne sortent pas du processeur, réduisant ainsi la sévérité du *memory wall*.

Le modèle de calcul Control-Driven Data-Flow (CDDF) Afin de prouver que les programmes écrits avec l'extension streaming peuvent être efficacement exécutés, et qu'ils bénéficient d'un nombre de propriétés essentielles à la productivité telles que le déterminisme, nous avons développé, au chapitre 3, un nouveau modèle de calcul qui prend

en compte à la fois le contrôle de flôt et le flôt de données. En se basant sur un programme de contrôle pour établir un ordonnancement déterministe des données dans les streams, l'ordonnancement des activations de tâches peut être simplifié et la synchronisation de la communication par streams peut être réduite à une forme permettant une implantation particulièrement efficace.

Nous démontrons ainsi que les programmes CDDF sont déterministes, à la fois du point de vue fonctionnel et des interblocages, et sérialisables, sous réserve de conditions qui sont toujours satisfaites par défaut dans la sémantique OpenMP actuelle. Enfin, nous prouvons un ensemble de conditions suffisantes, et surtout statiquement décidables, pour l'absence d'interblocages non fonctionnels, qui sont introduits par le mode de synchronisation plutôt que par la sémantique du programme, et, dans certains cas, pour l'absence d'interblocages fonctionnels, qui sont dûs à des dépendances de flôt cycliques non satisfaisables.

Afin de rapprocher le modèle CDDF de notre extension pour le streaming dans OpenMP, nous étudions le modèle sous des contraintes plus faibles au chapitre 4, en particulier montrant ainsi que le modèle CDDF est compatible avec une exécution en mémoire bornée et qu'il est possible d'exploiter le parallélisme du programme de contrôle sans compromettre les garanties de déterminisme.

Le chapitre 5 montre comment l'extension d'OpenMP peut être modélisée dans CDDF et décrit les bases de l'analyse statique de programmes de streams. Nous y proposons une solution pour la vérification des conditions statiques d'absence d'interblocages et nous fournissons une approche, basée sur la réutilisation des analyses de flôt de données existantes, pour permettre de générer du code optimisé.

Détection dynamique d'interblocages et algorithme de synchronisation sans opérations atomiques ou barrière mémoire L'analyse statique ne pouvant détecter tous les cas d'interblocages fonctionnels, nous avons conçu un algorithme de détection dynamique d'interblocage, à même de détecter et de résoudre les interblocages dûs au manque de ressources — lorsque la taille des tampons mémoire des streams est insuffisante — et de fournir une aide au debugging dans le cas des interblocages fonctionnels. En effet, lorsqu'un interblocage ne peut être résolu, il est possible d'identifier très précisément les tâches et les streams impliqués dans l'interblocage, ainsi que les accès aux streams qui n'auraient pu être satisfaits. Couplée avec la garantie de déterminisme des interblocages, cette approche permet de simplifier le processus de test des programmes et l'identification des conditions d'interblocage. La vérification de ces programmes est déterministe et ne dépend pas de la plateforme de test.

Nous présentons, au chapitre 6, un algorithme de synchronisation, spécifique aux streams CDDF, qui fait un usage extensif de toutes les garanties d'ordonnancement des accès mémoire dans les modèles de consistance mémoire Total Store Order (TSO) ainsi que x86, permettant une synchronisation sans avoir recours aux opérations atomiques ou aux barrières mémoire. Nous montrons que cet algorithme utilise effectivement chacune des contraintes d'ordonnancement de ces modèles de consistance et fournissons un algorithme équivalent pour des modèles mémoire très peu contraints tels que l'IBM POWER et le Relaxed Memory Order (RMO), où nous sommes forcés d'avoir recours aux barrières mémoire. Nous montrons également que le coût de cette forme de synchronisation est significativement inférieur au coût d'ordonnancement de tâches légères. Une discussion

préliminaire de ce travail est publiée dans [48].

Génération de code et implantation d'un prototype dans GCC [56,58,59] Le chapitre 7 présente notre algorithme de génération de code. Conçue pour une intégration complète dans l'implantation d'OpenMP du compilateur GCC, une première version de cet algorithme ne requiert aucune analyse statique, se conformant ainsi à la passe d'expansion d'OpenMP de GCC. Une deuxième version se base sur les résultats des analyses statiques présentées au chapitre 5 afin de générer un code streaming optimisé.

Bien qu'encore incomplète, l'implantation du prototype a atteint un stade où des applications complètes peuvent être compilées, notamment FMradio et 802.11_a.

Résultats expérimentaux Nous évaluons notre approche streaming sur les trois applications présentées au chapitre 8: FMradio, 802.11_a et la FFT unidimensionnelle. Afin de mesurer le potentiel de notre approche, nous comparons l'exécution séquentielle de ces applications à la fois à la génération de code automatique à partir de notre extension du langage OpenMP et aux applications parallélisées à la main utilisant la même librairie de runtime. La génération de code à la main permet de montrer le potentiel de notre approche une fois que la passe de génération de code sera achevée.

Nous présentons les accélérations obtenues par ces deux méthodes par rapport à l'exécution séquentielle, à l'exception de la FFT qui n'a pu être générée automatiquement à partir de la version annotée avec l'extension d'OpenMP à ce stade du développement. Nos résultats montrent des accélérations importantes sur trois architectures grand public. Par exemple, sur une plateforme équipée de quatre processeurs AMD Opteron 8380 (Shanghai), chacun ayant quatre coeurs pour un total de 16 coeurs cadencés à 2.5 GHz, nous mesurons les accélérations suivantes pour FMradio: accélération de 12.6× avec la version écrite à la main et une accélération de 10.5× avec le code généré automatiquement. Pour l'application 802.11_a, l'écriture manuelle permet d'atteindre une accélération de 13× comparée à une accélération plus modeste, de 6×, pour la version générée automatiquement.

Pour la FFT, nos résultats correspondent au calcul d'une seule FFT unidimensionnelle, plutôt qu'à un pipeline de calculs de FFT successives qui est plus adapté au modèle de streaming classique. Le meilleur résultat, obtenu pour une très grande taille de FFT sur un million d'éléments, est une accélération de 6.5×, qui est proche de 65% du résultat théoriquement possible sur une machine PRAM. Sur les petites tailles de problèmes, nous observons des décélérations, avec un point d'équilibre situé aux alentours d'une taille de 256 éléments. Ceci tend à montrer que notre approche s'applique non seulement à des problèmes qui se prêtent au streaming, mais aussi pour la parallélisation d'applications plus générales.

Travaux en cours et opportunités

De nombreuses idées nouvelles et opportunités de recherche diverses se sont dégagées de cette thèse. Certaines sont déjà en cours de développement, d'autres présentent des perspectives encourageantes pour de nouvelles études.

Deux principaux axes de recherche en compilation ont été lancés par cette thèse. Le premier provient de l'une de nos premières approches [57–59] de la streamisation, dans laquelle nous utilisions les streams comme une forme optimisée d'expansion mémoire dans la passe de distribution des boucles du compilateur GCC, et est également inspiré des travaux de recherche sur le pipelining logiciel découplé (*Decoupled Software Pipelining*) par Rangan et al. [62]. Ce développement [45], mené par Feng LI, étudiant en thèse à l'INRIA, se base sur une représentation intermédiaire sous forme d'assignation statique unique (*Single Static Assignment*) pour extraire du parallélisme de pipeline à partir de programmes séquentiels, fournissant un algorithme de partition basé sur la hiérarchie des dépendances de contrôle et utilisant notre extension streaming au langage OpenMP comme cible de génération de code.

Le deuxième axe s'est dégagé de la nécessité à faire appel à des résultats d'analyses statiques pour la génération de code streaming optimisé, en montrant les difficultés pour obtenir ces informations dans les tout premières étapes du flôt de compilation dans GCC, où se déroule l'expansion du langage OpenMP. Pour répondre à ce problème, nous avons développé une représentation intermédiaire qui permet de préserver l'information de haut niveau encodée dans les directives OpenMP tout au long du flôt de compilation, ce qui permet d'effectuer l'expansion de ces directives dans un contexte où les résultats des analyses statiques de dépendances sont disponibles. Cela présente de nombreux autres avantages, en particulier en évitant une obfuscation précoce du code, due aux multiples appels de fonctions opaques introduits par l'expansion d'OpenMP qui empêchent le bon déroulement des optimisations classiques du code séquentiel. Une publication de résultats préliminaires [55] couvre ce sujet.

En plus de ces travaux en cours, il reste plusieurs opportunités de recherche. Du côté applicatif, comme notre modèle permet de changer dynamiquement les canaux de flôt de données, les champs applicatifs des systèmes de contrôle et du traitement du signal avec changement de mode dynamique, comme par exemple l'encodage vidéo dynamique, ont naturellement une bonne affinité avec notre extension pour le streaming. L'implantation d'applications comme le traitement d'images à faible latence pour les contrôleurs de voitures ou d'avions présentent des caractéristiques pouvant mettre à profit notre modèle.

Il reste également des questions ouvertes, à la fois du côté compilateur et du côté runtime, en particulier au sujet de l'analyse statique de programmes streaming pour permettre d'appliquer les optimisations que nous avons dû laisser de côté. Tout au long de cette thèse, nous avons mentionné des cas d'optimisations pour lesquels nous ne disposions pas des informations requises à leur mise en œuvre, choisissant toujours une approche conservatrice. Par exemple, l'agrégation des données dans les streams appartenant à des composants fortement connexes du graphe de tâches pourrait être appliquée avec une approche mixte compilateur-runtime, en faisant varier le facteur d'agrégation selon le délai dynamique disponible dans chaque stream appartenant à un cycle. Les questions relatives à l'affinage de notre sur-approximation statique du graphe de tâches et à la possibilité de fournir une stratégie réversible d'équilibrage des charges, ce qui permettrait d'améliorer l'efficacité pour les programmes très dynamiques, présentent des défis particulièrement intéressants.

Une utilisation possible pour notre extension consiste à en faire un langage cible pour la compilation de langages de flôts de données de haut niveau, par exemple pour la génération

de code parallèle et désynchronisé à partir du langage LUSTRE. Une tentative en ce sens par Léonard Gérard a montré qu'il était nécessaire d'avoir recours à une compilation modulaire. Bien que cela ne soit pas encore implémenté dans notre prototype, notre schéma de compilation le permet.

Comme nous l'avons mentionné au Chapitre 5, l'une des idées les plus ambitieuses serait de faire appel à l'analyse de flôt de données dans des tableaux non seulement afin de décider de la régularité des tâches, mais aussi afin d'analyser les patterns des accès aux streams. Des transformations très importantes pourraient alors être rendues possibles, comme le démultiplexage des entrelacements de données produites ou consommées par différentes tâches, permettant ainsi la privatisation des streams, ou le ré-ordonnancement des streams dans le but de permettre la fusion de tâches ou l'optimisation des nids de boucles. En complément d'analyses permettant de mettre en évidence les potentiels invariants de flôt de données synchrones, cela pourrait s'avérer être l'une des optimisations les plus importantes.

Enfin, comme notre mécanisme de communication se prête naturellement à une implantation sous forme de canaux MPI, les tâches n'utilisant pas la mémoire partagée pourraient être exécutées sur des nœuds de calcul distants, étendant ainsi les travaux de Millot et al. dans le projet STEP [47], où le compilateur optimiseur source-à-source PIPS [35] est mis en œuvre afin de compiler des applications écrites avec le langage OpenMP pour une exécution en mémoire distribuée.

Perspectives

L'un de nos objectifs est de promouvoir l'intégration de notre extension pour le streaming dans l'une des prochaines révisions du standard OpenMP. Nous avons présenté une version préliminaire de ce travail au workshop IWOMP [54], qui est le principal lieu de rencontre pour les discussions sur l'évolution de ce langage. L'intégration au standard est un processus long, qui requiert entre autres une implantation complète dans un compilateur de production, et nous prévoyons de poursuivre cette voie dès lors que notre prototype sera complet. Ceci n'est évidemment pas la seule option pour promouvoir ces idées, puisque notre extension s'applique de manière transparente à tout modèle de programmation fournissant des primitives permettant une programmation parallèle par tâches, moyennant quelques ajustements pour garantir la causalité des tâches, concept sous-jacent à un bon nombre de nos résultats. Nous nous intéressons tout particulièrement au prochain standard OpenHMPP [60], où notre extension pourrait très facilement être intégrée, mais nous pouvons envisager d'étendre tout langage permettant la programmation par tâches, comme Cilk [24].

Personal Publications

[1] F. Li, A. Pop, and A. Cohen. Advances in parallel-stage decoupled software pipelining. In F. Bouchez, S. Hack, and E. Visser, editors, *Proceedings of the Workshop on Intermediate Representations*, pages 29–36, 2011.

[2] C. Miranda, P. Dumont, A. Cohen, M. Duranton, and A. Pop. Erbium: a deterministic, concurrent intermediate representation for portable and scalable performance. In N. M. Amato, H. Franke, and P. H. J. Kelly, editors, *Proceedings of the 7th Conference on Computing Frontiers, 2010, Bertinoro, Italy, May 17-19, 2010*, pages 119–120. ACM, 2010.

[3] C. Miranda, A. Pop, P. Dumont, A. Cohen, and M. Duranton. Erbium: a deterministic, concurrent intermediate representation to map data-flow tasks to scalable, persistent streaming processes. In *Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '10, pages 11–20, New York, NY, USA, 2010. ACM.

[4] H. Munk, E. Ayguadé, C. Bastoul, P. Carpenter, Z. Chamski, A. Cohen, M. Cornero, P. Dumont, M. Duranton, M. Fellahi, R. Ferrer, R. Ladelsky, M. Lindwer, X. Martorell, C. Miranda, D. Nuzman, A. Ornstein, A. Pop, S. Pop, L.-N. Pouchet, A. Ramirez, D. Rodenas, E. Rohou, I. Rosen, U. Shvadron, K. Trifunovic, and A. Zaks. ACOTES Project: Advanced Compiler Technologies for Embedded Streaming. *International Journal of Parallel Programming*, pages 1–54, 2010. 10.1007/s10766-010-0132-7.

[5] A. Pop and A. Cohen. A Stream-Comptuting Extension to OpenMP. In *International Workshop on OpenMP (IWOMP'10)*, Tsukuba, Japon, 2010. Poster.

[6] A. Pop and A. Cohen. Preserving high-level semantics of parallel programming annotations through the compilation flow of optimizing compilers. In *Proceedings of the 15th Workshop on Compilers for Parallel Computers (CPC'10)*, Vienna Autriche, 07 2010.

[7] A. Pop and A. Cohen. A stream-computing extension to openmp. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 5–14, New York, NY, USA, 2011. ACM.

[8] A. Pop, S. Pop, H. Jagasia, J. Sjödin, and P. H. J. Kelly. Improving gnu compiler collection infrastructure for streamization. In *Proceedings of the 2008 GCC Developers' Summit*, pages 77–86, June 2008.

[9] A. Pop, S. Pop, and J. Sjödin. Automatic streamization in gcc. In *Proceedings of the 2009 GCC Developers' Summit*, 2009.

[10] J. Sjödin, S. Pop, H. Jagasia, T. Grosser, and A. Pop. Design of graphite and the polyhedral compilation package. In *Proceedings of the 2009 GCC Developers' Summit*, pages 33–45, Montréal Canada, 06 2009. A/406/CRI.

Bibliography

[1] ACOTES: Advanced Compiler Technologies for Embedded Streaming. `http://www.hitech-projects.com/euprojects/ACOTES/`.

[2] Advanced Micro Devices, Inc. White Paper: Multi-Core Processors – The next evolution in computing. 2005.

[3] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29:66–76, December 1996.

[4] M. Aldinucci, M. Torquati, and M. Meneghin. FastFlow: Efficient Parallel Streaming Applications on Multi-core. *CoRR*, abs/0909.1187, 2009.

[5] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in Weak Memory Models. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 258–272. Springer Berlin / Heidelberg, 2010.

[6] Arvind, R. S. Nikhil, and K. Pingali. I-Structures: Data Structures for Parallel Computing. *ACM Trans. on Programming Languages and Systems*, 11(4):598–632, 1989.

[7] E. A. Ashcroft and W. W. Wadge. Lucid, a Nonprocedural Language with Iteration. *Communications of the ACM*, 20(7):519–526, 1977.

[8] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. *SIGPLAN Not.*, 46:487–498, January 2011.

[9] D. Barthou, J.-F. Collard, and P. Feautrier. Fuzzy Array Dataflow Analysis. *J. on Parallel and Distributed Computing*, 40:210–226, 1997.

[10] P. Bellens, J. M. Pérez, R. M. Badia, and J. Labarta. CellSs: a programming model for the Cell BE architecture. In *SC*, 2006.

[11] A. Bernstein. Program analysis for parallel processing. *IEEE Transactions on Computers*, 15:757–762, October 1966.

[12] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[13] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-Static Data Flow. *Acoustics, Speech, and Signal Processing, IEEE International Conference on*, 5:3255–3258, 1995.

[14] R. H. Bisseling. *Parallel Scientific Computation: A Structured Approach using BSP and MPI*. Oxford University Press, Mar. 2004.

[15] The Brook Language. `http://graphics.stanford.edu/projects/brookgpu/lang.html`.

[16] P. M. Carpenter, D. Ródenas, X. Martorell, A. Ramírez, and E. Ayguadé. A Streaming Machine Description and Programming Model. In *SAMOS*, pages 107–116, 2007.

[17] P. Caspi, G. Hamon, and M. Pouzet. *Real-Time Systems: Models and verification – Theory and tools*, chapter Synchronous Functional Programming with Lucid Synchrone. ISTE, 2007.

[18] P. Caspi and M. Pouzet. Synchronous Kahn networks. In *Proceedings of the first ACM SIGPLAN international conference on Functional programming*, ICFP '96, pages 226–238, New York, NY, USA, 1996. ACM.

[19] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *Intl. Conf. on Parallel Processing (ICPP)*, Saint Charles, IL, 1986.

[20] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13:451–490, October 1991.

[21] J. B. Dennis and G. R. Gao. An efficient pipelined dataflow processor architecture. In *Supercomputing (SC'88)*, pages 368–373, 1988.

[22] P. Feautrier. Array Dataflow Analysis. In S. Pande and D. Agrawal, editors, *Compiler Optimizations for Scalable Parallel Systems*, volume 1808 of *Lecture Notes in Computer Science*, pages 173–219. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-45403-9-6.

[23] P. Feautrier. Scalable and Structured Scheduling. *International Journal of Parallel Programming*, 34:459–487, 2006.

[24] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *ACM Symp. on Programming Language Design and Implementation (PLDI'98)*, pages 212–223, Montreal, Quebec, June 1998.

[25] J.-L. Gaudiot, T. DeBoni, J. Feo, W. Böhm, W. Najjar, and P. Miller. The Sisal Model of Functional Programming and its Implementation. In *Proceedings of the 2nd AIZU International Symposium on Parallel Algorithms / Architecture Synthesis*, PAS '97, pages 112–, Washington, DC, USA, 1997. IEEE Computer Society.

[26] M. Gordon, W. Thies, and S. Amarasinghe. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct 2006.

[27] P. L. Guernic, A. Benveniste, P. Bournai, and T. Gautier. Signal – A Data Flow-Oriented Language for Signal Processing. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 34(2):362–374, 1986.

[28] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[29] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, Sept. 1991.

[30] M. P. Herlihy and J. M. Wing. Axioms for concurrent objects. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '87, pages 13–26, New York, NY, USA, 1987. ACM.

[31] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[32] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the programming language Haskell: a non-strict, purely functional language version 1.2. *SIGPLAN Not.*, 27:1–164, May 1992.

[33] Intel. 64 Architecture Memory Ordering White Paper. Technical report, Aug. 2007.

[34] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 3A, 2006.

[35] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: an overview of the pips project. In *Proceedings of the 5th international conference on Supercomputing*, ICS '91, pages 244–251, New York, NY, USA, 1991. ACM.

[36] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug. 1974. North Holland, Amsterdam.

[37] R. Kalla, B. Sinharoy, and J. Tendler. IBM Power5 chip: a dual-core multithreaded processor. *Micro, IEEE*, 24(2):40–47, Mar-Apr 2004.

[38] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.

[39] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded Sparc processor. *Micro, IEEE*, 25(2):21–29, March-April 2005.

[40] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 114–124, New York, NY, USA, 2008. ACM.

[41] C. Kyriacou, P. Evripidou, and P. Trancoso. Data-Driven Multithreading Using Conventional Microprocessors. *IEEE Trans. on Parallel Distributed Systems*, 17(10):1176–1188, 2006.

[42] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programm. *IEEE Transactions on Computers*, 28:690–691, 1979.

[43] E. A. Lee and D. G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Trans. Computers*, 36(1):24–25, 1987.

[44] E. A. Lee and A. Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17:1217–1229, 1998.

[45] F. Li, A. Pop, and A. Cohen. Advances in Parallel-Stage Decoupled Software Pipelining. In F. Bouchez, S. Hack, and E. Visser, editors, *Proceedings of the Workshop on Intermediate Representations*, pages 29–36, 2011.

[46] V. Marjanovic, J. Labarta, E. Ayguadé, and M. Valero. Effective communication and computation overlap with hybrid MPI/SMPSs. In *PPOPP*, 2010.

[47] D. Millot, A. Muller, C. Parrot, and F. Silber-Chaussumier. STEP: A Distributed OpenMP for Coarse-Grain Parallelism Tool. In R. Eigenmann and B. de Supinski, editors, *OpenMP in a New Era of Parallelism*, volume 5004 of *Lecture Notes in Computer Science*, pages 83–99. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-79561-2.

[48] C. Miranda, A. Pop, P. Dumont, A. Cohen, and M. Duranton. Erbium: a deterministic, concurrent intermediate representation to map data-flow tasks to scalable, persistent streaming processes. In *Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '10, pages 11–20, New York, NY, USA, 2010. ACM.

[49] H. Munk, E. Ayguadé, C. Bastoul, P. Carpenter, Z. Chamski, A. Cohen, M. Cornero, P. Dumont, M. Duranton, M. Fellahi, R. Ferrer, R. Ladelsky, M. Lindwer, X. Martorell, C. Miranda, D. Nuzman, A. Ornstein, A. Pop, S. Pop, L.-N. Pouchet, A. Ramirez, D. Rodenas, E. Rohou, I. Rosen, U. Shvadron, K. Trifunovic, and A. Zaks. ACOTES Project: Advanced Compiler Technologies for Embedded Streaming. *International Journal of Parallel Programming*, pages 1–54, 2010. 10.1007/s10766-010-0132-7.

[50] S. Owens, S. Sarkar, and P. Sewell. A Better x86 Memory Model: x86-TSO. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in*

Higher Order Logics, volume 5674 of *Lecture Notes in Computer Science*, pages 391–407. Springer Berlin / Heidelberg, 2009.

[51] V. Pankratius, A. Jannesari, and W. F. Tichy. Parallelizing Bzip2: A Case Study in Multicore Software Engineering. *IEEE Softw.*, 26(6):70–77, 2009.

[52] T. M. Parks. *Bounded scheduling of process networks*. PhD thesis, Berkeley, CA, USA, 1995. UMI Order No. GAX96-21312.

[53] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical Task-Based Programming With StarSs. *Intl. J. on High Performance Computing Architecture*, 23(3):284–299, 2009.

[54] A. Pop and A. Cohen. A Stream-Comptuting Extension to OpenMP. In *International Workshop on OpenMP (IWOMP'10)*, Tsukuba, Japon, 2010. Poster.

[55] A. Pop and A. Cohen. Preserving high-level semantics of parallel programming annotations through the compilation flow of optimizing compilers. In *Proceedings of the 15th Workshop on Compilers for Parallel Computers (CPC'10)*, Vienna Autriche, 07 2010.

[56] A. Pop and A. Cohen. A stream-computing extension to OpenMP. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 5–14, New York, NY, USA, 2011. ACM.

[57] A. Pop and S. Pop. A proposal for last private clause on OpenMP task Pragma. Technical report, MINES ParisTech, CRI - Centre de Recherche en Informatique, Mathématiques et Systèmes, 35 rue St Honoré 77305 Fontainebleau-Cedex, France, Jan. 2009.

[58] A. Pop, S. Pop, H. Jagasia, J. Sjödin, and P. H. J. Kelly. Improving GNU compiler collection infrastructure for streamization. In *Proceedings of the 2008 GCC Developers' Summit*, pages 77–86, June 2008.

[59] A. Pop, S. Pop, and J. Sjödin. Automatic streamization in GCC. In *Proceedings of the 2009 GCC Developers' Summit*, 2009.

[60] S. B. R. Dolbeau and F. Bodin. HMPP: A Hybrid Multi-core Parallel Programming Environment. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 2007.

[61] R. Ramanathan. Intel multi-core processors: Making the move to quad-core and beyond. *Technology@Intel Magazine*, 4(1):2–4, Dec 2006.

[62] R. Rangan, N. Vachharajani, M. Vachharajani, and D. August. Decoupled software pipelining with the synchronization array. In *13th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, Sept. 2004.

[63] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *PLDI 2011*, to appear.

[64] S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 379–391, New York, NY, USA, 2009. ACM.

[65] SPARC International, Inc., CORPORATE. *The SPARC architecture manual (version 9)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[66] K. Stavrou, M. Nikolaides, D. Pavlou, S. Arandi, P. Evripidou, and P. Trancoso. TFlux: A Portable Platform for Data-Driven Multithreading on Commodity Multicore Systems. In *Intl. Conf. on Parallel Processing (ICPP'08)*, pages 25–34, Portland, Oregon, Sept. 2008.

[67] R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[68] The StreamIt Language. `http://www.cag.lcs.mit.edu/streamit/`.

[69] The OpenMP Architecture Review Board. OpenMP Application Program Interface. `http://www.openmp.org/mp-documents/spec30.pdf`.

[70] I. Watson and J. R. Gurd. A Practical Data Flow Computer. *IEEE Computer*, 15(2):51–57, 1982.

Index

Exploitation du streaming pour la parallélisation déterministe :

approche langage, compilateur et système de runtime intégrée

Résumé :

La performance des unités de calcul séquentiel a atteint des limites technologiques qui ont conduit à une transition de la tendance à l'accélération des calculs séquentiels vers une augmentation exponentielle du nombre d'unités de calcul par microprocesseur. Ces nouvelles architectures ne permettent d'augmenter la vitesse de calcul que proportionnellement au parallélisme qui peut être exploité, soit via le modèle de programmation soit par un compilateur optimiseur. Cependant, la disponibilité du parallélisme en soi ne suffit pas à améliorer les performances si un grand nombre de processeurs sont en compétition pour l'accès à la mémoire. Le modèle de streaming répond à ce problème et représente une solution viable pour l'exploitation des architectures à venir.

Cette thèse aborde le streaming comme un modèle général de programmation parallèle, plutôt qu'un modèle dédié à une classe d'applications, en fournissant une extension pour le streaming à un langage standard pour la programmation parallèle avec mémoire partagée, OpenMP. Un nouveau modèle formel est développé, dans une première partie, pour étudier les propriétés des programmes qui font appel au streaming, sans les restrictions qui sont généralement associées aux modèles de flot de données. Ce modèle permet de prouver que ces programmes sont déterministes à la fois fonctionnellement et par rapport aux deadlocks, ce qui est essentiel pour la productivité des programmeurs. La deuxième partie de ce travail est consacrée à l'exploitation efficace de ce modèle, avec support logiciel à l'exécution et optimisations de compilation, à travers l'implantation d'un prototype dans le compilateur GCC.

Mots clés : streaming, parallélisation, runtime, langages, compilation

Leveraging Streaming for Deterministic Parallelization:

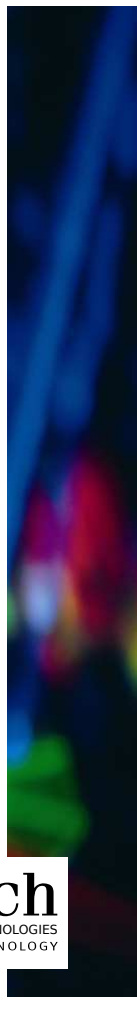
an Integrated Language, Compiler and Runtime Approach

Abstract:

As single processing unit performance has reached a technological limit, the power wall, the past decade has seen a shift from the prevailing trend of increasing single-threaded performance to an exponentially growing number of processing units per chip. Higher performance returns on these newer architectures are contingent on the amount of parallelism that can be efficiently exploited in applications, either exposed through parallel programming or by parallelizing compilers. However, uncovering raw parallelism is insufficient if a host of cores vie for limited off-chip memory bandwidth. Mitigating the memory wall, the stream-computing model provides an important solution for exploiting upcoming architectures.

This thesis explores streaming as a general-purpose parallel programming paradigm, rather than a model dedicated to a class of applications, by providing a highly expressive stream-computing extension to a de facto standard for shared memory programming, OpenMP. We rely on a new formal framework to investigate the properties of streaming programs, without the restrictions usually attached to dataflow models, and we prove that such programs benefit from deadlock and functional determinism, key assets in the productivity race. In a second part, we focus on the efficient exploitation of our model, with optimized runtime support and compiler optimizations, through an implementation in the GCC compiler.

Keywords: streaming, parallelization, runtime, stream-computing languages, compilation



MINES ParisTech

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY