Thèse

présentée pour obtenir le grade de docteur de l'École Nationale Supérieure des Télécommunications et de l'Universidade Federal de Campina Grande

Spécialité : Communications et Électronique

Daniel Cardoso de Souza

Algorithme de Partitionnement Appliqué aux Systèmes Dynamiquement Reconfigurables en Télécommunications

Soutenue le 13 Décembre 2006 devant le jury composé de

Raimundo Carlos Silvério Freire, Dr., UFCG	Président
Edna Natividade da Silva Barros, Dr., UFPE	Rapporteurs
Ivan Saraiva da Silva, Dr., UFRN	
Jean-François Naviner, Dr., ENST	Examinateurs
Marcelo Alves de Barros, Dr., UFCG	
Lírida Alves de Barros Naviner, Dr., ENST	Directeurs de Thèse
Benedito Guimarães Aguiar Neto, Dr., UFCG	

École Nationale Supérieure des Télécommunications

Universidade Federal de Campina Grande

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Programa de Pós-Graduação em Engenharia Elétrica

École Nationale Supérieure des Télécommunications
Département Communications et Électronique

Algoritmo de Particionamento Aplicado a Sistemas Dinamicamente Reconfiguráveis em Telecomunicações

Daniel Cardoso de Souza

Tese de Doutorado submetida à Coordenação do Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande, Brasil, e à École Nationale Supérieure des Télécommunications, França, como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências no domínio da Engenharia Elétrica.

Área de Concentração : Processamento de Informação

Orientadores
Prof. Benedito Guimarães Aguiar Neto, Dr., UFCG
Profa. Lírida Alves de Barros Naviner, Dra., ENST

Campina Grande, Brasil
Paris, França

Remerciements

Tout d'abord, il est mon devoir de remercier le peuple brésilien (par le biais du Gouvernement Fédéral) qui a permis de financer la réalisation de ce travail. Je remercie le Ministère de l'Education et son agence de support CAPES, par le support financier accordé sous la forme de la bourse de doctorat, au Brésil et en France, et aussi sous forme des billets d'avion et aides.

Je remercie les deux établissements où ce travail a été réalisé, l'UFCG et l'ENST, qui m'ont très bien accueilli et m'ont accordé l'infrastructure et les ressources nécessaires pour parvenir à ce document de thèse. Il a été une grande satisfaction d'avoir appartenu aux deux écoles, desquelles je garderai beaucoup de souvenirs et de leçons. Je remercie les professeurs docteurs Benedito Guimarães Aguiar Neto, de l'UFCG, et Lírida Alves de Barros Naviner, de l'ENST, pour l'opportunité de faire ce doctorat dans leurs institutions respectives, et aussi pour leur aide et attention tout au long du chemin.

Je remercie le personnel administratif des deux institutions, pour leur constante sollicitude et pour leur aide dans la résolution de tous les problèmes liés à la bureaucratie, les documents, le visa, l'immatriculaton et les démarches académiques. Pour leur bonne volonté et efficacité, mon merci beaucoup à Ângela de Lourdes, Pedro Alves e Lucimar Ribeiro, de l'UFCG ; et à Chantal Cadiat, Marie Baquero, Danielle Childz, Florence Besnard et Stéphane Bonenfant, de l'ENST.

Je tiens bien sûr à remercier le département COMELEC et le groupe SEN de m'avoir accueilli au sein de ses locaux, ce qui a largement contribué à la réussite de ce doctorat. L'action du groupe SEN se situe dans le contexte de la convergence des télécommunications, de l'informatique et de l'audiovisuel, dans le cadre général de l'émergence de la société de l'information. Le groupe SEN participe à la mise en oeuvre de la stratégie de Télécom Paris, ainsi qu'au renforcement de sa notoriété et de son image.

Merci beaucoup à Jesus Ubach, Jerôme Bacon et Vanderley Maia, stagiaires qui ont contribué à la réalisation de ce travail.

Je remercie beaucoup mes parents, Edilson et Guiomar, ma tante Zelinda et mon frère Ricardo pour leur amour inconditionnel et leur grand support, autant émotionnel que matériel, depuis le jour de ma décision de quitter mon travail à Belém pour partir faire un doctorat, jusqu'au jour de la soutenance de ma thèse. Grâce à eux j'ai eu la force pour m'encourager face aux difficultés.

Je laisse mon éternelle amitié et ma profonde gratitude à tous les amis exceptionnels et fantastiques que j'ai connus à Campina Grande et à Paris, qui ont partagé autant mes joies que mes désespoirs durant le long trajet de développement de ce travail. Pour la compagnie, pour tous les moments inoubliables, pour les fêtes, les cadeaux, l'encourage-

ment et pour l'amitié, j'enregistre ici les noms de Adoniran Judson, Amira Alloum, Ana Kosman, Anatilde Figueira, André Cardoso, Antonio Cipriano, Celina Takemura, Chantal Cadiat, Daniel Scherer, David Camarero, Denis Franco, Dimitri Edouart, Edroaldo Cavalcante, Emilio Strinati, Fabrício Braga, Felipe Silveira, Fernanda Streit, Fernando Rangel, Francisco Santos, Ghassan Kraidy (tu me cherches ?), Ghaya Rekaya, Gustavo Fischer, Ioannis Krikidis (grand merci cher "cousin" le grec latin), Jane Buzzi, João Negrão (big ! !), Juliana Leão, Karim Ben Kalaia (impeccable !), Korinna Lenz, Leonardo Nogueira, Luiz Antônio Neves, Luiz Brunelli, Luiz Gonzaga, Manel Romdhane, Márcia Costa e Silva, Maria Bosch, Maria do Carmo, Martin Schmidt, Maya, Mounia (Lydia) Lourdiane, Paulo Márcio, Qing Xu, Raquel Aline, Renata Simões, Rinaldo dos Santos, Roberta Barros, Robert Elscher, Rodrigo Souza, Rômulo do Valle, Sami Mekki, Sheng Yang, Soriana Lucena, Thiciany Matsudo, Vinícius Carvalho, Vinícius Licks, Will Almeida, et Zenilda Probst.

Merci beaucoup, avec de fortes accolades, à mon cousin Munir Sleiman et son épouse Sandrine, pour leur hospitalité, pour les heures agréables que nous avons passées ensemble, avec de bonnes conversations et inoubliables promenades.

Merci beaucoup aussi aux amis d'il y a longtemps, Ricardo Zaninelli, Patrícia Helena et Marilsa Peres, pour leur encouragement et pour les conversations.

Je remercie ma douce copine Nina Salakidou, qui m'a conquis avec son caractère, amour, compréhension, optimisme et patience, et m'a beaucoup encouragé durant la dernière année de la réalisation de la thèse.

Finalement, je rends grâce à Dieu pour la vie et pour la santé.

Résumé

Cette thèse a pour but de proposer un algorithme de partitionnement matériel/logiciel optimisé. On travaille sur l'hypothèse de que quelques caractéristiques spécifiques à certains algorithmes déjà publiés puissent être combinées de façon avantageuse, menant à l'amélioration d'un algorithme de partitionnement de base et, par conséquence, des systèmes hétérogènes générés par cet algorithme. L'ensemble d'optimisations proposées pour être réalisées dans ce nouvel algorithme consiste en : généralisation des architectures-cible candidates avec l'ajout de FPGA's pour le partitionnement, considération précise des coûts et puissances des fonctions allouées en matériel, ordonnancement de systèmes au matériel dynamiquement reconfigurable, et prise en compte de plusieurs alternatives d'implémentation d'un noeud d'application dans un même processeur. Ces optimisations sont implémentées en versions successives de l'algorithme de partitionnement proposé, lesquelles sont testées avec deux applications de traitement du signal. Les résultats du partitionnement démontrent l'effet de chaque optimisation sur la qualité du système hétérogène obtenu.

Resumo

Este trabalho tem como objetivo propor um algoritmo de particionamento *hardware/software* otimizado. Trabalha-se com a hipótese de que algumas características específicas de certos algoritmos já publicados possam ser combinadas vantajosamente, levando ao aprimoramento de um algoritmo de particionamento de base, e conseqüentemente dos sistemas heterogêneos gerados por ele. O conjunto de otimizações propostas para serem realizadas nesse novo algoritmo consiste de : generalização das arquiteturas-alvo candidatas com a inclusão de FPGA's para o particionamento, consideração precisa dos custos e potências das funções mapeadas em *hardware*, agendamento de sistemas com *hardware* reconfigurável dinamicamente, e consideração de múltiplas alternativas de implementação de um nó de aplicação em um mesmo processador. Essas otimizações são implementadas em sucessivas versões do algoritmo de particionamento proposto, que são testadas com duas aplicações de processamento de sinais. Os resultados do particionamento demonstram o efeito de cada otimização na qualidade do sistema heterogêneo obtido.

Abstract

This work's goal is to propose an optimized hardware/software partitioning algorithm. We work on the hypothesis that some specific features of certain published algorithms can be advantageously combined for the improvement of a base partitioning algorithm, and of its generated heterogeneous systems. The set of optimizations proposed for the achievement of this new algorithm encompass : generalization of candidate target architectures with the inclusion of FPGA's for the partitioning, precise consideration of functions' implementation costs and power consumptions in hardware, manipulation of systems with dynamically reconfigurable hardware, and consideration of multiple implementation alternatives for an application node in a given processor. These optimizations are implemented in successive versions of the proposed partitioning algorithm, which are tested with two signal processing applications. The partitioning results demonstrate the effect of each optimization on the achieved heterogeneous system quality.

Table des matières

Table des figures

Liste des tableaux

Liste des Définitions d'Acronymes

ASIC Circuito integrado de aplicação específica (do inglês *Application-Specific Integrated Circuit*) ;

ASIP Processador com conjunto de instruções de aplicação específica (do inglês *Application-Specific Instruction set Processor*) ;

BCS Busca binária da restrição (do inglês *Binary Constraint-Search*) ;

BIL Melhor valor imaginário (do inglês *Best Imaginary Level*) ;

BIM Melhor agenda imaginária (do inglês *Best Imaginary Makespan*) ;

BSB Bloco básico de agendamento (do inglês *Basic Scheduling Block*) ;

CCM Máquina de computação configurável (do inglês *Configurable Computing Machine*) ;

CDFG Grafo de fluxo de dados e controle (do inglês *Control/Data Flow Graph*) ;

CDMA Acesso múltiplo por divisão de código (do inglês *Code-Division Multiple Access*) ;

CLB Bloco lógico configurável (do inglês *Configurable Logic Block*) ;

CPU Unidade central de processamento (do inglês *Central Processing Unit*) ;

DHE Sistema embarcado heterogêneo e distribuído (do inglês *Distributed Heterogeneous Embedded*) ;

DRT Instante de disponibilidade dos dados (do inglês *Data Ready Time*) ;

DSP Processador digital de sinais (do inglês *Digital Signal Processor*) ;

ECST Menor instante de início da execução (do inglês *Earliest Computation Start Time*) ;

EPR Reconfiguração parcial antecipada (do inglês *Early Partial Reconfiguration*) ;

FPGA Arranjo (matriz) de portas lógicas programáveis no campo, isto é, pelo usuário (do inglês *Field Programmable Gate Array*) ;

GCLP Criticalidade global / fase local (do inglês *Global Criticality / Local Phase*) ;

HMP Arquitetura multi-processada heterogênea (do inglês *Heterogeneous MultiProcessor*) ;

IPC Comunicação interprocessadores (do inglês *InterProcessor Communication*);

LDPC Verificação de paridade de baixa densidade (do inglês *Low-Density Parity Check*);

LE Elemento lógico (do inglês *Logic Element*);

LUT Tabela de consulta (do inglês *Look-Up Table*);

MIBS Mapeamento e escolha da alternativa de implementação (do inglês *Mapping and Implementation Bin Selection*);

PDS Processamento digital de sinais;

PE Elemento processador (do inglês *Processing Element*);

RRT Instante de disponibilidade do recurso (do inglês *Resource Ready Time*);

RTR Reconfigurabilidade em tempo de execução (do inglês *Run-Time Reconfigurability*);

SoC Sistema em uma pastilha de circuito integrado (do inglês *System on a Chip*);

VHDL Linguagem de descrição de *hardware* do projeto VHSIC (do inglês *VHSIC Hardware Description Language*);

VHSIC Circuitos integrados de velocidade muito alta (do inglês *Very High Speed Integrated Circuits*);

XML Linguagem de marcação estendida (do inglês *eXtended Markup Language*).

Résumé Étendu

Cette thèse de doctorat a été réalisée dans le cadre d'un accord de co-tutelle entre l'Universidade Federal de Campina Grande – UFCG, à Campina Grande-PB, au Brésil, et à l'École Nationale Supérieure des Télécommunications – ENST, à Paris, en France. Ce doctorat, qui a reçu le soutien financier de CAPES (organisation gouvernementale brésilienne), faisait partie du programme de travail prévu dans l'accord de coopération CAPES/COFECUB n° 343/01.

Introduction

Cette thèse aborde le contexte du développement de systèmes embarqués hétérogènes, c'est-à-dire, ceux où le traitement de l'application est fait, coopérativement, par des composants appartenant aux domaines d'implémentation matériel et logiciel. Ces deux types d'implémentation possèdent chacun des atouts et des inconvénients, qui se complémentent [1, 2] :

1. L'implémentation d'un algorithme en matériel présente presque toujours une performance supérieure, en termes de vitesse d'exécution, par rapport à son implémentation logicielle.
2. Le coût pour une solution entièrement matérielle est habituellement beaucoup plus important que celui pour une solution logicielle, en ce qui concerne la taille du matériel utilisé, et le temps et le coût de développement.
3. Par rapport au matériel, l'implémentation logicielle mène habituellement à un temps de développement plus court et à un coût plus faible pour une même fonctionnalité, cela presque toujours au sacrifice de la performance.
4. Le matériel est traditionnellement fixe et inchangeable après sa fabrication, tandis que le logiciel offre l'avantage de la flexibilité, en raison d'être plus facilement modifiable par un développeur. Plus récemment, ce modèle a été changé pour plusieurs types de systèmes, avec l'emploi des technologies de matériel reconfigurable (dont les

principaux représentants sont les FPGA's), lesquelles apportent la flexibilité aussi au domaine matériel.

L'implémentation de systèmes embarqués de façon hétérogène permet donc de combiner les avantages spécifiques aux deux domaines d'implémentation et de placer la solution dans un point optimum de sa courbe coût x performance. Ce point est défini habituellement comme celui où le système atteint la performance demandée par sa spécification, ayant en même temps un coût total minimum, à la fois en termes monétaires et de durée de projet.

La complexité croissante des systèmes embarqués, en termes de fonctionnalités, a demandé le développement de méthodes intégrées, systématiques et concurrentes pour le projet du matériel et du logiciel pour un même système, en exploitant leurs synergies. Ces méthodes ont constitué le nouveau domaine de la **Conception Conjointe Matériel-Logiciel** (*HW/SW Codesign*), né au début des années 1990's. Dans toutes ces méthodes, l'étape la plus critique est celle où les décisions sur le **partitionnement** du système sont prises, c'est-à-dire les décisions sur quelles parties de la spécification du système seront réalisées en matériel et lesquelles seront réalisées en logiciel. Le partitionnement peut définir l'architecture finale du système et sa division entre les domaines matériel et logiciel, aussi bien que l'ordonnancement de l'exécution des tâches de l'application. Pour ces raisons, le partitionnement est le processus central et le plus relevant de n'importe quelle méthode de conception conjointe, ayant le plus grand effet sur le coût et la performance du système hétérogène.

Du fait de son importance stratégique, de nombreux algorithmes de partitionnement matériel/logiciel ont été publiés [1–16]. L'objet de la recherche réalisée au cours de la thèse est la phase de partitionnement du processus de conception conjointe, plus spécifiquement les algorithmes pour l'exécution de cette phase. La motivation pour cette recherche est le fait que le développement de nouveaux algorithmes de partitionnement, de plus en plus efficaces, est le chemin majeur menant à des systèmes hétérogènes avec un rapport qualité/coût optimisé.

Le travail de recherche décrit dans ce document part de l'hypothèse qu'il est possible d'intégrer des caractéristiques avantageuses spécifiques de divers algorithmes de partitionnement déjà publiés dans la littérature, pour le développement d'un algorithme optimisé. Cela devrait mener à une prise en compte efficace de la connaissance sur le développement des algorithmes de partitionnement, et finalement à la synthèse de systèmes hétérogènes aussi optimisés en coût et performance. La plupart des articles analysés pour cette thèse apportent des contributions originales au partitionnement, mais aucun d'entre eux n'a essayé cette démarche d'optimisation. Cela rend plus difficile l'évolution et la continuité de

la recherche dans le domaine, car de bonnes techniques ne sont pas forcément réutilisées dans les nouveaux algorithmes. Une deuxième hypothèse est qu'il est possible d'intégrer les besoins des algorithmes de partitionnement pour des systèmes à matériel fixe et à matériel reconfigurable. Ainsi, un seul algorithme pour les deux types de systèmes pourrait décider de l'exploitation ou non de la reconfigurabilité.

L'objectif principal de ce travail est de proposer et développer un algorithme de partitionnement matériel/logiciel optimisé, de façon à vérifier les deux hypothèses ci-décrites, et qui soit applicable à des systèmes génériques par rapport à l'architecture-cible, au nombre de modes d'opération et à la reconfigurabilité du matériel.

La démarche suivie pour la réalisation de cette thèse a commencé par l'étude et l'évaluation d'algorithmes de partitionnement réels de la littérature, ce qui a permis de définir un ensemble de critères de qualité pour ces algorithmes, issus de l'analyse faite de l'état de l'art :

- Généralité :
  - par rapport à l'architecture-cible : mono ou multiprocesseur, fixe ou synthétisée ;
  - par rapport au domaine d'application des systèmes à partitionner : algorithmes "multi-domaine" ou de domaine spécifique ;
  - par rapport au nombre de modes (applications) : systèmes mono-mode ou multi-mode ;
- Diversité de réquisits de qualité du partitionnement et du système :
  - prise en compte des coûts de communication entre le matériel et le logiciel (temps de communication et taille des interfaces) ;
  - prise en compte du parallélisme d'exécution entre des noeuds matériels et des noeuds logiciels ;
  - possibilité de partage de ressources matérielles entre deux noeuds ou plus de l'application ;
  - possibilité de choix parmi plusieurs alternatives d'implémentation, en matériel ou en logiciel, pour un même noeud ;
  - prise en compte, le cas échéant, de la reconfigurabilité en temps d'exécution (RTR) des FPGA's (seulement pour des systèmes au matériel reconfigurable) ;
  - possibilité d'exploitation, le cas échéant, de la reconfigurabilité partielle du matériel (seulement pour des systèmes au matériel reconfigurable).

Les caractéristiques ci-décrites correspondent à plusieurs façons de réduire la latence, le coût et la consommation d'un système hétérogène, et elles ont été déterminées pour l'évaluation de la qualité de chaque algorithme. Ensuite, le flot de travail employé a été le suivant :

1. Proposition et implémentation d'un algorithme optimisé avec quelques unes des propriétés ci-dessus.
2. Sélection des applications de test.
3. Test et évaluation de l'algorithme optimisé développé.

L'algorithme de partitionnement optimisé proposé dans cette thèse est implémenté et évalué par étapes, ce qui permet de suivre son évolution. Chaque implémentation d'une nouvelle caractéristique (optimisation) dans le code du partitionneur, qui le laisse dans un état fonctionnel, en donne une nouvelle version à tester. Les critères d'évaluation de la qualité de ces versions de l'algorithme sont : la complexité computationnelle du partitionnement et la performance et le coût des applications partitionnées, en termes de leurs vitesses d'exécution, leurs coûts en matériel et en mémoire occupés et leurs consommations de puissance.

Pour le test des versions améliorées de l'algorithme de partitionnement, il a été décidé d'utiliser des applications dans le domaine de télécommunications. Les raisons pour ce choix ont été : le fait que ce domaine a connu un remarquable progrès au cours des dernières décennies, possédant ainsi un grand intérêt académique et industriel ; et le fait que ce domaine emploie des algorithmes de traitement numérique du signal avec des exigences accrues de haute performance et faible coût, lesquels pourraient donc tirer grand parti d'une implémentation hétérogène. Par ailleurs, le champ des communications mobiles, dont les standards et algorithmes évoluent de façon assez rapide, s'avère prometteur pour l'emploi de composants au matériel reconfigurable, avec le but de viabiliser une mise à jour agile des produits. Pour toutes ces raisons, les applications de test choisies pour ce travail ont été un récepteur *Rake* pour des systèmes mobiles CDMA et un décodeur de canal LDPC. La plateforme hétérogène utilisée pour caractériser ces applications pour le processus de partitionnement a été la carte Nios II / Stratix II d'Altera.

Algorithme de partitionnement de base

L'algorithme choisi pour servir de base à l'implantation de l'algorithme optimisé est celui créé par H. Oh et S. Ha [16] pour leur outil de *Codesign* PEaCE [17]. En effet, cet algorithme satisfait à plus de critères de qualité (définis dans la section précédente) que les autres algorithmes étudiés. Par ailleurs, le code source de l'outil PEaCE est ouvert et accessible à l'utilisateur, ce qui permet son édition. L'algorithme de partitionnement de base travaille sur des architectures multiprocesseurs hétérogènes (HMP) génériques, et prend en compte le temps dû à la communication inter-processeurs. L'algorithme d'ordonnancement, appelé BIL [18], a pour but de minimiser la durée totale de l'agenda,

c'est-à-dire le temps d'exécution du graphe de l'application, et aussi de trouver l'Elément Processeur (PE) optimum pour chaque noeud. Le fonctionnement de ces deux algorithmes est décrit en détail dans [14, 16].

Trois fichiers d'entrée écrits en XML sont nécessaires pour le fonctionnement de l'algorithme de partitionnement. Pour un système "Exemple" à partitionner, ces fichiers s'appellent :

1. "`Exemple.xml`", qui contient les graphes hiérarchiques des applications du système à être partitionné. Les noeuds du graphe d'une application sont appelés des "tâches", et chaque tâche est aussi un graphe contenant les noeuds de fonctionnalité n_i, qui sont les éléments partitionnables. Un exemple de cette structure hiérarchique d'un mode est montré dans la figure 2.2, à la page 37. Les valeurs associées aux arches des graphes des applications sont aussi fournies.
2. "`Exemple_TimeCost.xml`" consiste d'une table de profils "Noeud-PE", où sont spécifiés tous les PE's candidats, en logiciel et en matériel, avec ses coûts monétaires par unité, les quantités disponibles de chaque type de PE et les temps d'écriture d'un mot de données sur le bus, pour chaque PE. Sous l'en-tête de définition de chaque PE candidat, tous les noeuds atomiques sont listés, avec leurs valeurs d'implémentation (temps d'exécution, taille et puissance) sur ce PE.
3. "`Exemple_mode.xml`" consiste d'une table spécifiant les contraintes imposées à chaque tâche de chaque mode du système. Ces valeurs sont indépendantes de la solution de partitionnement ou des PE's choisis pour la composer. Le fichier liste les noms des modes, contenant les noms des leurs tâches avec la période et l'échéance (*deadline*) de chacune.

L'algorithme de partitionnement original produit un fichier "`Exemple_sched.xml`" comme sortie, contenant les résultats du partitionnement et de l'ordonnancement. Dans ce fichier, les noeuds des graphes des applications sont alloués et ordonnés dans les PE's sélectionnés pour l'architecture du système, et l'instant d'exécution de chacun est défini.

Une analyse détaillée du code source C++ originel de l'algorithme de partitionnement de base, tel qu'il était disponible dans la version courante de l'outil PEaCE [17], a montré qu'il s'agissait en réalité d'un code assez incomplet, contenant plusieurs erreurs importantes de logique, et ne correspondant pas à la description faite en [16]. Ce code original a exigé un effort de programmation important pour le rendre opérationnel, pour qu'il puisse alors être pris comme une référence pour l'évaluation de l'impact des optimisations proposées sur la qualité d'un système hétérogène. Une lacune majeure trouvée a été l'absence d'une fonction de coût. Par ailleurs, seulement deux types de PE's étaient acceptés : "CPU" et "ASIC". Même si un FPGA était utilisé en réalité, il devrait être déclaré comme

du type “ASIC”, parce que les contraintes et coûts propres à un FPGA n'étaient pas pris en compte.

Outre les corrections nécessaires dans le code, nous avons décidé de le retirer de l'outil PEaCE, écrit pour le système d'exploitation Linux Red Hat, et de migrer le travail vers l'environnement Windows^{TM}, pour profiter de la disponibilité de Microsoft Visual C++^{TM} 2003. Cet outil rend beaucoup plus facile le travail de développement d'un logiciel. Après l'obtention de PEaCE sur internet [17], nous avons identifié les fichiers sources C++ responsables seulement par l'implémentation de l'algorithme de partitionnement, et ensuite nous avons travaillé exclusivement sur ces fichiers, avec le but de créer un logiciel de partitionnement indépendant de l'environnement PEaCE.

Dans le nouveau logiciel ainsi créé, l'utilisateur donne le nom et le type de système (multi-mode/multi-tâche ou mono-mode/mono-tâche) à partitionner au début de son exécution. Il choisit aussi la méthode de partitionnement à employer selon le type de système. Ce logiciel constitue la première version pour Windows de l'algorithme de partitionnement de PEaCE.

Pour parvenir à une version vraiment fonctionnelle de l'algorithme de partitionnement de base, celui-ci a été doté d'une fonction de coût basique. Pour alimenter cette fonction, des nouveaux champs ont été ajoutés au fichier d'entrée `TimeCost.xml`, pour informer les coûts et les contraintes des PE's de logiciel (CPU's) et de matériel (ASIC's). Le coût déclaré pour une CPU candidate est devenu son coût monétaire par unité, et les contraintes sont devenues les quantités disponibles de mémoire, qui est censée être divisée en mémoire de programme et de données. Le coût monétaire total de ces mémoires est aussi fourni. Le coût pour un ASIC candidat est devenu son coût monétaire par unité de surface (=coût/mm^2), et sa contrainte est la surface maximale permise pour lui. Le coût final d'un ASIC dépendra donc de sa surface totale et de la technologie.

Les coûts des noeuds des applications dans chaque PE ont passé à être calculés à partir des valeurs fournies par l'utilisateur dans le fichier `TimeCost.xml`. Un noeud alloué à une CPU consomme l'espace mémoire disponible pour celle-ci (lequel est une ressource élémentaire limitée). Ainsi, le coût pour un noeud en logiciel est devenu la quantité de mémoire (en octets), de programme et de données, occupée par le code du noeud. Cette taille mémoire est alors normalisée en la divisant par sa contrainte (la taille disponible). Ensuite, ce coût normalisé du noeud est multiplié par le prix de la mémoire, ce qui donne la valeur finale de son coût, exprimée dans une unité monétaire. Donc le coût d'un noeud alloué dans un PE de logiciel devient le coût de la fraction de mémoire effectivement occupée par le noeud.

Pour un noeud alloué dans un ASIC, le coût est défini par sa surface de circuit occupée (soit la mesure classique de la littérature, applicable seulement aux ASIC's), où l'unité

de surface a un coût fixe et dépendent de la technologie de fabrication. Cette surface de matériel occupée est multipliée par le coût du mm^2 de surface de cet ASIC.

La fonction coût élémentaire créée pour l'algorithme de partitionnement de base calcule le coût total d'une solution de partitionnement de la façon suivante : tous les noeuds d'un même type possèdent le même temps de exécution et le même coût, dans un PE donné. Les équations 0.1 et 0.2 expriment les coûts des noeuds sur les deux types de PE's, qui sont les coûts des ressources qui implémentent ces noeuds. Le coût d'une ressource dans un PE devient le coût d'un noeud du même type dans ce PE.

Pour un noeud n_i dans un PE de logiciel 'p' :

$$Cout(n_i, p) = \frac{TailleMem(n_i, p)}{TailleMemProg(p) + TailleMemDon(p)} \times CoutMem(p) \tag{0.1}$$

Pour un noeud n_i dans un PE de matériel 'p' :

$$Cout(n_i, p) = Surface(n_i, p) \times Cout\_mm^2(p) \tag{0.2}$$

À chaque itération de l'algorithme de base, celui-ci balaie la liste de ressources allouées pour l'architecture, identifie le PE auquel chaque ressource allouée appartient et additionne (accumule) les coûts de ces ressources par instance de PE. Ainsi, pour chaque PE dans le fichier `TimeCost.xml`, la fonction de coût obtient le coût total des ressources allouées appartenant à ce PE. Cela est fait parce que chaque PE a un coût d'utilisation différent, alors il n'est pas possible de additionner des coûts de ressources implémentées dans des PE's différents, même s'il s'agit de PE's du même type. Par exemple, deux ASIC's peuvent avoir des coûts par mm^2 différents, s'ils appartiennent à des technologies différentes, et alors les ressources allouées dans chacun de ces ASIC's doivent être totalisées séparément. Ensuite l'algorithme traverse les PE's et additionne les coûts totaux des ressources allouées dans chaque PE au coût monétaire du PE individuel (s'il y en a), et accumule ces résultats dans un coût total de l'architecture du système.

Dans la version de base de l'algorithme de partitionnement, les consommations de puissance des noeuds et des PE's ne sont pas prises en compte.

Après la création de cette fonction de coût basique, le partitionnement a finalement pu être exécuté correctement, avant l'implémentation des optimisations proposées par cette thèse. Cet algorithme de partitionnement ci-décrit est dorénavant nommé "version 1.0".

Proposition d'un algorithme de partitionnement optimisé

Cette section présente l'algorithme de partitionnement optimisé que nous proposons dans cette thèse, implémenté en trois versions successives à partir de la version 1.0. Les optimisations suivantes sont ajoutées à chaque version :

1. Prise en compte détaillée des contraintes de PE's et d'applications, des coûts liés à la consommation de ressources d'un FPGA, et des consommations de puissance par les noeuds des applications. Incorporation d'une fonction de coût complexe qui intègre des coûts et des contraintes dans la même expression.
2. Prise en compte du temps de reconfiguration du matériel dans l'ordonnancement des applications, pour des systèmes avec du matériel dynamiquement reconfigurable.
3. Prise en compte de plusieurs alternatives d'implémentation pour un noeud dans un même PE.

Ensuite, nous expliquons chacun de ces groupes d'optimisations. Nous faisons remarquer la nouveauté de l'intégration de ces optimisations dans un même algorithme de partitionnement.

Coûts, contraintes et puissances des PE's, des noeuds et de la solution

Le premier ensemble d'optimisations fait au code de la version 1.0 de l'algorithme de partitionnement a eu pour but de le préparer pour traiter des PE's du type "FPGA", en plus des PE's des types "CPU" et "ASIC" déjà reconnus. Le logiciel a été étendu pour :

1. Lire et manipuler les contraintes et les coûts propres à un FPGA et aux noeuds qui y sont implémentés.
2. Prendre en compte la consommation de puissance statique des PE's et de puissance dynamique des noeuds dans chaque PE, avec le but de calculer la consommation de puissance totale des partitions proposées par l'algorithme.
3. Lire et manipuler les nouvelles contraintes définies pour le système à partitionner, qui ont été ajoutées au fichier `mode.xml`.

Finalement, l'algorithme a été étendu avec une fonction de coût intégrant les coûts des noeuds et les contraintes des PE's dans une seule expression, comme démontré dans [19, 20]. La version de l'algorithme de partitionnement issue de ces modifications a été nommée "version 2.0".

Le processeur du type “FPGA” a été ajouté au format du fichier `TimeCost.xml`, avec son coût monétaire et ses contraintes. Le logiciel a donc été adapté pour le lire avec ses valeurs. Les contraintes définies pour un FPGA sont les quantités totales disponibles de ses ressources internes :

- LE's ou LUT's (Eléments Logiques) ;
- *bits* de RAM ;
- blocs multiplieurs ;
- blocs de PLL's ;
- broches d'E/S.

Ce sont les ressources élémentaires typiquement trouvés dans les FPGA's. Le coût de la mémoire de reconfiguration associée à un FPGA est aussi ajouté au fichier `TimeCost.xml`.

Pour chaque noeud d'application déclaré dans la table d'un FPGA, les quantités de chaque type de ressource élémentaire du FPGA occupées par ce noeud sont informées. À partir de ces quantités, nous pouvons calculer le **coût d'implémentation** de ce noeud dans ce FPGA. La plupart des algorithmes de partitionnement de la littérature utilisent uniquement la surface de matériel occupée par un noeud comme la mesure de son coût d'implémentation en matériel, mais cette mesure est adéquate seulement pour les ASIC's, où elle détermine directement le coût monétaire. Il n'a pas été trouvé, dans la littérature, la prise en compte des divers types de ressources internes d'un FPGA, et de leurs quantités, pour le partitionnement, ce qui constitue un indicatif de l'originalité de ce travail.

Dans la version 2.0, avant toute affectation d'un noeud à un PE, il est vérifié si l'occupation des ressources élémentaires de celui-ci par le noeud respecte les contraintes du PE. L'algorithme balaie les quantités de ressources élémentaires encore disponibles dans le PE, pour vérifier la validité de chaque allouement de noeud. Dans le cas d'une CPU, la ressource élémentaire est la mémoire disponible ; pour un ASIC, c'est la surface encore disponible ; et pour un FPGA, ce sont les types de ressources listés préalablement. Les quantités occupées de chaque ressource élémentaire sont soustraites des quantités disponibles, les mettant à jour pour les allouements suivants.

Les nouvelles contraintes pour une application, ajoutées au fichier `mode.xml` dans la version 2.0, consistent des valeurs maximales de performance (le temps d'exécution total), coût monétaire et consommation de puissance. Dans la version 1.0, ce fichier contient seulement les périodes des tâches.

Les fonctions de coût employées ont la forme générale suivante :

$$F(\wp) = \sum_i \kappa_i \times \frac{C_i(\wp)}{C_i} + \sum_i \kappa_{C_i} \times F_C(C_i(\wp)) \tag{0.3}$$

où :

– $\wp$ est la partition sous évaluation ;
– $C_i(\wp)$ est la valeur d'un coût particulier ;
– κ_i est le poids attribué à ce coût dans l'évaluation de la partition ;
– C_i est la contrainte de projet appliquée au coût $C_i(\wp)$ dans la partition $\wp$, et qui sert à la normalisation ;
– $\digamma_C(C_i(\wp))$ est un terme de correction ;
– κ_{C_i} est le poids pour ce terme de correction.

Les termes de correction $\digamma_C(C_i(\wp))$ pour la fonction de coût dans ce travail sont les termes de pénalité [19, 20], dont la formulation analytique est la suivante :

$$\digamma_C(C_i, C_i(\wp)) = \sum_i r^2[C_i(\wp), C_i] \tag{0.4}$$

où

$$r[C_i(\wp), C_i] = \max\left\{0, \frac{[C_i(\wp) - C_i]}{C_i}\right\} \tag{0.5}$$

Les calculs des coûts d'allouement des noeuds dans les PE's (consommation de ressources élémentaires), et des consommations de puissance totales dans chaque PE, sont faits ensemble. La fonction de coût intègre les coûts et les consommations de puissance dans une valeur unique, qui représente globalement la qualité de la partition. Tous ces calculs ont été implémentés de la manière suivante :

1. Les coûts des noeuds sont calculés en fonction des fractions de ressources élémentaires consommées. Chaque instance de noeud a son coût calculé dans chaque PE `p`, par le biais des équations suivantes :

Pour un noeud n_i dans un PE de logiciel `p` :

$$Cout(n_i, p) = \frac{TailleMemProg(n_i, p) + TailleMemDon(n_i, p)}{TailleMemProg(p) + TailleMemDon(p)} \tag{0.6}$$

Pour un noeud n_i dans un ASIC `p` :

$$Cout(n_i, p) = Surface(n_i, p) \tag{0.7}$$

Pour un noeud n_i dans un FPGA `p` :

$$Cout(n_i, p) = \frac{[\frac{LEs(n_i,p)}{LEs(p)} + \frac{RAMs(n_i,p)}{RAMs(p)} + \frac{Mults(n_i,p)}{Mults(p)} + \frac{PLLs(n_i,p)}{PLLs(p)}]}{4} \tag{0.8}$$

L'équation 0.8 totalise le coût d'un noeud implémenté dans un FPGA. La division par 4 a pour but de maintenir ce coût entre 0 et 1.

2. Un noeud alloué dans un PE est appelé une "ressource". Le coût et la puissance de cette ressource sont égaux à ceux du noeud dans ce PE.

Si type(ResAllou) = type(n_i) :

$$CoutResAllou(p) = Cout(n_i, p) \tag{0.9}$$

$$PuissResAllou(p) = Puissance(n_i, p) \tag{0.10}$$

3. Pour toute instance de PE dans le fichier `TimeCost.xml`, nous totalisons le coût et la puissance des ressources qui lui sont allouées. Par exemple, deux FPGA's différents sont totalisés séparément. Cela est fait parce que chaque PE a son propre coût monétaire et sa propre puissance statique, donc nous ne pouvons pas additionner des coûts et des puissances de ressources allouées dans des PE's différents, même s'il s'agit de PE's du même type.

$$CoutTotResAllou(p) = \sum CoutResAllou(p), \forall p, \forall ResAllou \tag{0.11}$$

$$PuissTotResAllou(p) = \sum PuissResAllou(p), \forall p, \forall ResAllou \tag{0.12}$$

4. En visitant chaque PE, l'algorithme de partitionnement vérifie sa nature et convertit le coût total des ressources allouées en un coût monétaire, par le biais des équations :

Pour une CPU `p` :

$$CoutTotResAllou(p) \leftarrow CoutTotResAllou(p) \times CoutMem(p) + Cout(p) \tag{0.13}$$

Pour un ASIC `p` :

$$CoutTotResAllou(p) \leftarrow CoutTotResAllou(p) \times Cout\_mm^2(p) \tag{0.14}$$

Pour un FPGA `p` *:*

$$CoutTotResAllou(p) \leftarrow CoutTotResAllou(p) \times Cout(p) + CoutMemReconf(p) \quad (0.15)$$

5. En visitant chaque PE, l'algorithme de partitionnement ajoute sa puissance statique à la valeur totale de sa puissance dynamique (due aux ressources allouées dans ce PE) :

$$PuissTotResAllou(p) \leftarrow PuissTotResAllou(p) + PuissStat(p) \quad (0.16)$$

6. Le coût total de tous les ressources allouées (coût de la partition) est l'addition des coûts totaux des ressources allouées par PE `p` :

$$CoutPart(\wp) = \sum_{p} CoutTotResAllou(p) \quad (0.17)$$

7. La puissance totale de tous les ressources allouées (puissance de la partition) est l'addition des puissances totales de chaque PE `p` :

$$PuissPart(\wp) = \sum_{p} PuissTotResAllou(p) \quad (0.18)$$

8. Nous appliquons les termes de correction au coût et à la puissance totaux de la partition, en employant les contraintes données dans le fichier `mode.xml`. L'équation finale de la fonction de coût est :

$$\begin{aligned} F(\wp) = {} & \kappa_{cout} \times \frac{CoutPart(\wp)}{ContrCoutMode} + \\ & \kappa_{puiss} \times \frac{PuissPart(\wp)}{ContrPuissMode} + \\ & \kappa_{C_{cout}} \times F_C(ContrCoutMode, CoutPart(\wp)) + \\ & \kappa_{C_{puiss}} \times F_C(ContrPuissMode, PuissPart(\wp)) \end{aligned} \quad (0.19)$$

où :

- *ContrCoutMode* et *ContrPuissMode* sont les contraintes de coût et puissance de l'application, respectivement ;
- κ_{cout} et κ_{puiss} sont les poids du coût monétaire total et de la puissance totale, respectivement.

Ces poids permettent de mettre en exergue le paramètre de qualité désiré, en lui attribuant un poids légèrement plus grand que les poids des autres paramètres. Les poids des termes de correction $\kappa_{C_{cout}}$ et $\kappa_{C_{puiss}}$ doivent recevoir des valeurs élevées, pour augmenter considérablement la pénalité de la fonction de coût quand il y a des violations des contraintes. Il est intéressant de faire de sorte que $\sum_i \kappa_i = 1$, car ainsi $F(\wp) = 1$ devient une valeur de référence, une fois que des violations des contraintes donneront des valeurs $F(\wp) \gg 1$; et des valeurs optimisées des paramètres donneront $F(\wp) < 1$.

La version originale du partitionneur de PEaCE produit comme sortie un fichier d'ordonnancement `Exemple_sched.xml`, où les noeuds des graphes des tâches sont partitionnés et ordonnés sur les PE's choisis pour l'architecture, et le temps d'exécution de chaque PE est informé. Nous avons ajouté à toutes les versions de l'algorithme de partitionnement la création d'un fichier de coûts `Exemple_costs.xml`, qui donne :

- Les quantités occupées de mémoire (CPU), surface (ASIC) et ressources élémentaires (FPGA), qui sont les coûts des noeuds ;
- Les coûts et consommations de puissance des partitions logicielle et matérielle ;
- Le coût et la consommation de puissance de la solution finale pour le système.

Reconfiguration dynamique du matériel

Les composants au matériel dynamiquement reconfigurable sont ceux qui peuvent être reconfigurés en temps d'exécution (RTR) et dans les mêmes systèmes où ils sont insérés, avec ou sans l'intervention humaine, mais sans interrompre l'opération de ces systèmes. RTR signifie la capacité de modifier la fonction implantée dans le matériel pendant l'exécution de l'application. Les différentes configurations du matériel peuvent rester stockées en mémoire et être changées par un logiciel superviseur du système.

La version 2.0 de l'algorithme de partitionnement peut manipuler des CPU's, des ASIC's et des FPGA's sans reconfiguration dynamique. La version 3.0 en est une extension, qui prend en compte ces trois types de PE's et en plus des FPGA's dynamiquement reconfigurables. Dans les algorithmes de partitionnement pour des systèmes hétérogènes au matériel dynamiquement reconfigurable, la modification principale, par rapport aux algorithmes pour des systèmes au matériel fixe, est la prise en compte de la latence de reconfiguration dans l'ordonnancement. Celle-ci peut représenter le temps pour reconfigurer un noeud sur une partie du FPGA, ou pour reconfigurer le FPGA tout entier. Dans le premier cas, la cible d'optimisation du partitionnement est de minimiser ou masquer cette latence, pour réduire les temps d'exécution des graphes et ainsi optimiser la performance de l'application partitionnée. Plusieurs techniques à ce but ont déjà été proposées dans la littérature [21].

Dans la version 3.0 de l'algorithme, nous avons implémenté le support à une reconfiguration totale et disruptive du FPGA. Du fait de sa nature, ce type de reconfiguration ne peut pas être ordonnée en parallèle avec les communications entre des noeuds de l'application, parce que toute opération dans le FPGA va s'arrêter durant une reconfiguration. Elle peut cependant être ordonnée en parallèle avec l'exécution de la partition logicielle du système, mais le temps de reconfiguration de toute la matrice du FPGA est presque toujours beaucoup plus grand que les temps d'exécution des noeuds d'application. Cela peut influencer décisivement le résultat du partitionnement.

Le logiciel a d'abord été adapté pour reconnaître le nouveau type de PE nommé "FPGADin" (pour indiquer qu'il s'agit d'un FPGA dynamiquement reconfigurable), à être déclaré dans le fichier `TimeCost.xml`. Malgré le fait que le temps de reconfiguration est une constante pour un même FPGA, nous avons décidé de déclarer sa valeur par noeud, au lieu de globalement par FPGA, en envisageant l'implémentation future du support à une reconfiguration partielle. Si le FPGA est dynamiquement reconfigurable mais pas d'une façon partielle, donc les temps de reconfiguration de tous les noeuds seront égaux au temps de reconfiguration du FPGA entier.

Nous ne pouvons pas prévoir si la reconfiguration dynamique sera faite pour un système, ni entre quels noeuds de l'application elle sera ordonée. Pour définir cela, l'outil d'ordonnancement BIL doit connaître les consommations de ressources du FPGADin par noeud, et suivre les quantités de chaque type de ressource encore disponibles dans le FPGADin après tout allouement. Ainsi, il pourra décider de l'emploi ou non de la reconfiguration dynamique et calculer le moment où l'ordonner. Normalement, c'est BIL qui choisit les PE's à utiliser dans le système partitionné, alors c'est à lui de choisir si le matériel sera fixe ou reconfigurable, ou même si tous les deux types seront utilisés. Quand il y a un FPGADin candidat pour l'architecture du système, donc avant tout allouement d'un nouveau noeud n_j à lui (un noeud d'un type encore inexistant dans la configuration courante du FPGA), l'algorithme d'ordonnancement BIL doit vérifier le respect à ses contraintes de ressources :

1. Si la taille du nouveau noeud n_j, en termes des quantités de ressources occupées, est plus petite que les quantités de ressources couramment disponibles dans le FPGA, donc l'algorithme alloue n_j dans la configuration initiale du FPGA, et donc l'ordonnancement d'une reconfiguration dynamique n'est pas nécessaire. Les quantités de ressources disponibles dans le FPGA sont mises à jour par la soustraction des ressources prises par le noeud n_j.

2. Si la taille de n_j demande plus de ressources que celles disponibles dans le FPGA, donc il est nécessaire d'ordonner la reconfiguration du FPGA, car celui-ci n'a aucun noeud déjà configuré qui soit du même type de n_j, et les noeuds déjà alloués ne laissent pas de

ressources libres suffisantes pour que n_j soit aussi alloué.

Si la situation "2" arrive, l'algorithme trouve le moment où ordonner la reconfiguration du FPGA : c'est l'instant où celui-ci devient disponible (ses noeuds finissent d'exécuter). Dans ce moment, il ordonne deux nouveaux types de noeuds créés dans la version 3.0 : un "noeud de sauvegarde de contexte" et un "noeud de reconfiguration". Le premier correspond à la tâche de sauvegarder en mémoire les résultats intermédiaires des noeuds d'application qui vont être déconfigurés, pour que les noeuds de la page de configuration suivante puissent poursuivre l'exécution de l'application. Le noeud de reconfiguration est affecté d'une valeur de temps d'exécution égale au temps de reconfiguration de n_j (qui est le temps de reconfiguration du FPGA). Ces deux types de noeuds sont ordonnés en séquence dans le FPGA, à partir de l'instant de début de la reconfiguration. Pour des fins d'évaluation de la technique, la valeur utilisée dans cette thèse pour la somme des temps de sauvegarde de contexte et de reconfiguration est d'une seconde (1 s). La consommation de puissance d'une reconfiguration est aussi prise en compte, étant définie arbitrairement en 500 mW. Ces valeurs sont incorporées au temps d'exécution et à la consommation de puissance totaux du système, dans le fichier de résultats `costs.xml`.

Après l'ordonnancement de la reconfiguration, les ressources (types de noeuds) qui existaient auparavant dans le FPGA sont marquées comme "non configurées" et les quantités de ressources disponibles sont réinitialisées à ses valeurs totales. Ensuite l'algorithme continue à ordonner les noeuds suivants de l'application, lesquels vont lire de la mémoire les résultats intermédiaires calculés par les noeuds de la configuration précédente. Le résultat final (l'application partitionnée et ordonnée) contient le noeud de reconfiguration, avec la durée totale de l'ordonnancement et la puissance du système augmentées du temps et de la puissance de reconfiguration. Le coût monétaire du système est potentiellement inférieur, car le matériel est multiplexé dans le temps, donc moins de LE's sont employés pour implémenter l'application, malgré le fait que le FPGA dynamiquement reconfigurable a un prix par unité plus élevé.

Plusieurs alternatives d'implémentation

La version 4.0 de l'algorithme de partitionnement est capable de traiter plusieurs alternatives d'implémentation pour un même noeud dans un même PE, que ce soit matériel ou logiciel, représentant ainsi la troisième optimisation proposée dans ce travail. Surtout en matériel, une même fonctionnalité peut être implémentée de façons diverses, correspondant par exemple à différentes architectures pour un noeud décrites dans une HDL, où chacune représente un compromis entre performance, taille et puissance. Pour l'algorithme de partitionnement, cela signifie que des instances différentes du même type de noeud (la

même fonction) puissent avoir des valeurs de coût et de performance différentes ; il faut que ces alternatives d'implémentation soient ajoutées au fichier `TimeCost.xml`, dans la table du processeur où elles sont possibles pour ce noeud.

Le cas le plus commun est le support à plusieurs alternatives d'implémentation en matériel. Nous prenons comme exemple l'opération de multiplication, qui est faite en logiciel toujours avec le même opérateur, mais qui peut être réalisée en matériel de plusieurs façons. Alors, tous les noeuds de multiplication doivent avoir le même nom de type, car il s'agit de la même opération. Le fichier `.xml` du graphe de l'application ne subit aucun changement, avec les noms d'instance déclarés de façon impartiale, comme `mulIX` par exemple. Dans les tables de noeuds pour des PE's de logiciel, dans le fichier `TimeCost.xml`, toutes les instances de noeuds de multiplication ont les mêmes noms d'instance du graphe, parce que il n'y a pas d'autres alternatives d'implémentation dans ce cas.

Les modifications ont lieu dans les tables de noeuds pour des PE's de matériel, où les alternatives d'implémentation **pour chaque instance** des noeuds de multiplication `mulIX` doivent être déclarées en séquence, en utilisant le même nom d'instance avec l'ajout d'un deuxième indice, comme dans : `mulI0_0`, `mulI0_1`, ..., `mulI0_n`. Les alternatives d'implémentation sont choisies indépendamment par instance de noeud, c'est-à-dire, un noeud de multiplication en matériel peut être implémenté par une alternative, en même temps qu'un autre noeud est implémenté par une autre alternative. L'algorithme de partitionnement lit les valeurs de temps d'exécution, taille, ressources occupées, puissance et temps de reconfiguration des alternatives d'implémentation, et remplit les matrices correspondantes d'un même objet noeud, déjà indexées par PE 'p' et par implémentation 'i'. Pour chaque instance du noeud dans le graphe de l'application, l'algorithme de partitionnement analyse chacune des alternatives d'implémentation disponibles pour lui, pour décider si cette instance doit être allouée en matériel, et choisir en même temps l'implémentation qui convient le plus. Le critère pour ce choix est donné par la fonction de coût utilisée.

Pour que le choix d'alternative d'implémentation fonctionne normalement, il faut que les différentes alternatives soient créées avec la même interface, de sorte qu'une puisse être remplacée par l'autre dans le graphe partitionné. La différence entre elles doit être limitée seulement à l'architecture.

Pour implanter le support à plusieurs alternatives d'implémentation, il faut adapter toutes les méthodes d'ordonnancement et partitionnement pour la manipulation de matrices au lieu de vecteurs. Par exemple, les déclarations `BIL[p]` et `BIM[p]` doivent être modifiées vers `BIL[p][i]` et `BIM[p][i]`, ce qui entraîne de modifications dans toutes les méthodes qui utilisent les valeurs de BIL et BIM d'un noeud. Partout où il y a des boucles qui parcourent les processeurs candidats, il faut ajouter des boucles internes pour

parcourir aussi les alternatives d'implémentation possibles de chaque noeud dans chaque processeur.

Complexité computationnelle de l'algorithme optimisé

La complexité computationnelle de l'algorithme de partitionnement original (versions de base et 1.0) est de $O(S.P^2.N_t)$ [16], où S est la complexité de l'outil d'ordonnancement BIL, P est le nombre total de PE's candidats et N_t est le nombre de tâches. $S = O(N^2.p.\log(p))$, où N est le nombre de noeuds du graphe d'une tâche et p est le nombre de PE's pris en compte pour l'ordonnancement [18].

Les versions 2.0 et 3.0 ont ajouté de nouveaux membres aux classes de PE's et de noeuds, de nouveaux tests conditionnels et les calculs de coûts et puissances de PE's, de noeuds et de solutions hétérogènes. Néanmoins, il n'y a pas eu l'ajout de nouvelles itérations en fonction des quantités d'éléments à traiter, et en conséquence la complexité computationnelle n'a pas été modifiée.

La version 4.0 a ajouté une nouvelle dimension au partitionnement, l'alternative d'implémentation, ce qui représente un choix de plus pour le partitionnement et l'ordonnancement. Néanmoins, des alternatives multiples sont disponibles seulement pour quelques uns des noeuds et dans quelques uns des PE's, et dans des quantités variables pour chaque paire (noeud, PE), rendant plus difficile la déduction d'une expression générale de complexité. La limite maximale est atteinte dans le cas où tous les N noeuds du graphe possèdent chacun I alternatives d'implémentation dans tous les PE's, parmi lesquelles i alternatives dans les PE's candidats, quand la complexité S de l'outil d'ordonnancement BIL devient $S = O(N^2.p.i.\log(p.i))$, et la complexité du partitionnement devient $O(S.(P.I)^2.N_t)$.

Résultats du Partitionnement d'Applications

Les quatre versions de l'algorithme de partitionnement expliquées précédemment ont été utilisées pour générer des solutions hétérogènes pour deux applications de traitement de signaux : un récepteur radio numérique *Rake* pour des systèmes CDMA et un décodeur de canal LDPC. Nous décrivons dans cette section seulement les résultats des partitionnements faits pour le *Rake*, parce que les résultats pour le LDPC mènent aux mêmes conclusions.

La plateforme hétérogène utilisée pour la caractérisation des noeuds des applications est la carte Nios II / Stratix II^{TM} d'Altera, sur laquelle les noeuds peuvent être exécutés et évalués. Cette carte est composée principalement d'un FPGA Stratix II EP2S60F672C5,

dans lequel peut être configuré un coeur processeur de logiciel Nios II. Il s'agit d'un processeur embarqué virtuel à l'architecture de 32 *bits*, qui peut être programmé dans les langages C/C++ par le biais de son environnement de développement Nios II IDE^{TM}.

L'évaluation des noeuds des applications en logiciel est faite sur leurs implémentations pour le processeur Nios II. La méthode d'estimation des temps d'exécution des noeuds utilise la fonction `alt_timestamp()` du langage C du Nios II [22]. Pour mesurer les tailles occupées dans la mémoire par les noeuds (ses codes exécutables), ceux-ci sont mis dans leurs propres fichiers C, ensuite compilés dans l'environnement Nios II IDE, et les tailles en octets de ces exécutables sont déclarées comme les occupations de mémoire par les noeuds, dans le fichier `TimeCost.xml`. Puisqu'il n'est pas possible de mesurer la consommation de puissance instantanée du Nios II seul, nous ne pouvons pas mesurer la puissance des noeuds en logiciel. Donc nous déclarons la valeur de puissance statique du Nios II (estimée par Quartus II) pour tous les noeuds en logiciel, dans le fichier `TimeCost.xml`.

Pour les paramètres des noeuds en matériel (temps d'exécution, nombres de ressources du FPGA occupées, puissance), il suffit de faire leurs synthèses suivies de simulation dans l'outil Quartus II, pour le modèle de FPGA présent dans la carte Nios II / Stratix II. Pour le partitionnement avec la version 1.0 de l'algorithme, qui traite seulement une mesure de surface occupée par un noeud en matériel, comme dans un ASIC, nous avons fait une addition pondérée des quantités de ressources du FPGA utilisées, de sorte à trouver une valeur de "surface équivalente" à déclarer dans le fichier `TimeCost.xml`.

Les temps d'exécution et les coûts d'implémentation (complexités) des noeuds d'interfaçage, en logiciel et en matériel, sont pris en compte dans le partitionnement, augmentant le temps d'exécution et le coût de la solution hétérogène finale.

Récepteur *Rake*

La première application utilisée pour tester les quatre versions de l'algorithme de partitionnement est un récepteur radio *Rake*, dont l'algorithme est disponible en langage C pour exécution en logiciel, et la description est disponible en VHDL pour exécution en matériel. Le graphe de l'application *Rake* a 21 noeuds au total, et chaque type de noeud est caractérisé en logiciel et en matériel, comme décrit dans la section précédente.

Version 1.0

Pour le test de la version 1.0, la contrainte de surface maximale spécifiée pour le *Rake* est de 500 mm^2 et le coût par unité de surface est arbitré en 500 \$/mm^2 (de valeurs typiques), menant à une contrainte de coût matériel maximal de 250.000 \$. Le tableau 4.6 (à la page 106) montre les propositions d'implémentations hétérogènes du *Rake* sur la

carte Nios II / Stratix II, faites par la version 1.0 et obtenues pour plusieurs valeurs de la contrainte de période du *Rake*, dans le fichier `mode.xml`.

La version 1.0 ne vérifie pas le respect aux contraintes de coût, respectant seulement la contrainte de période (la performance exigée). Les solutions proposées violent les contraintes de surface de matériel et donc de coût maximal pour des périodes jusqu'à 190 μs. Ce problème est corrigé à partir de la version 2.0 du partitionnement, avec l'ajout d'une fonction de coût, ce qui se constitue une des contributions de cette thèse.

Le graphe de la figure 0.1 montre que les valeurs de période du *Rake* hétérogène fournies par la version 1.0 suivent les valeurs de la contrainte de période et sont toujours inférieures à celles-ci. Le temps d'exécution du *Rake* seulement en matériel est de 639 ns, et son temps d'exécution en logiciel est de 3.121 μs, presque 5 mille fois plus grand. Tel est le gain de performance maximal (ou la réduction maximale de temps d'exécution) qui peut être obtenu pour le *Rake* avec la migration de fonctions du logiciel vers le matériel dédié.

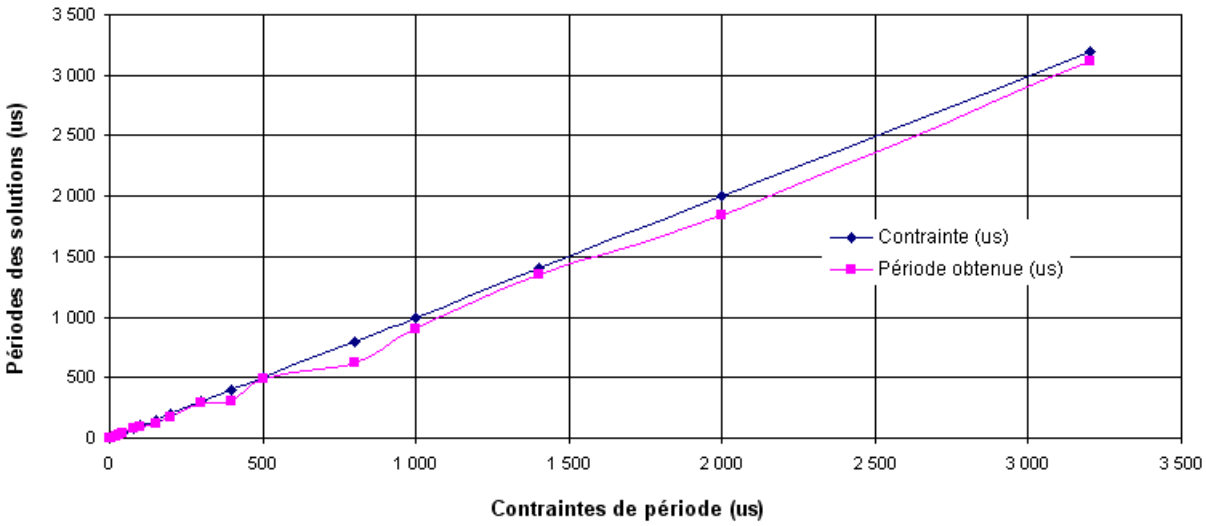


FIG. 0.1: Valeurs de période du récepteur *Rake* partitionné par la version 1.0.

Le graphe de la figure 0.2 montre les valeurs de surface du matériel et de coût monétaire des nombreuses solutions, en fonction de leurs valeurs de période, où il peut être observé que la surface et le coût diminuent de façon à peu près exponentielle avec l'augmentation du temps d'exécution du *Rake*. Le coût d'implémentation du *Rake* entièrement en matériel est de 468.000 $, et son coût entièrement en logiciel est de 103 $, une réduction de 4.540 fois. Le coût monétaire total inclut les prix du processeur (100 $), de la fraction de mémoire utilisée (entre 2 et 3 $) et de la surface du FPGA occupée, conformément aux équations 0.13 et 0.14. La combinaison de faible coût et haute capacité des mémoires actuelles permet à la partition logicielle d'avoir une contribution extrêmement faible (autour de

103 $) pour le coût d'une solution hétérogène, car celui-ci est dominé par le coût de la surface de matériel.

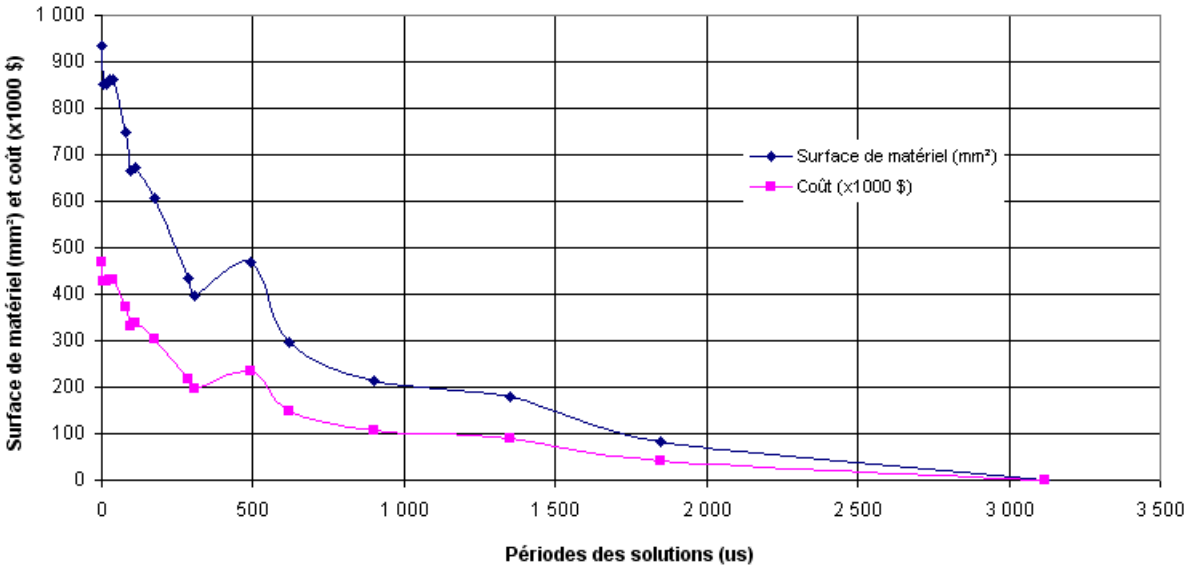


FIG. 0.2: Valeurs de coût du récepteur *Rake* partitionné par la version 1.0.

Version 2.0

Le tableau 4.7 (à la page 107) montre les propositions d'implémentations hétérogènes pour le *Rake* sur la carte Nios II / Stratix II, faites par la version 2.0 de l'algorithme de partitionnement. Ces solutions ont été obtenues pour plusieurs valeurs des contraintes de période d'exécution, coût et puissance du *Rake*, déclarées dans le fichier `mode.xml`. Nous pouvons remarquer que la version 2.0 respecte simultanément toutes les contraintes imposées par l'utilisateur au système partitionné, grâce à l'emploi de fonctions de coût intégrant les coûts et les contraintes dans une même expression.

Le temps d'exécution de l'application satisfait toujours sa contrainte, mais il n'est pas le temps minimum possible. Cela peut être observé quand la contrainte de période est augmentée : l'algorithme profite du fait que le temps d'exécution permis est relâché, en acceptant des valeurs plus élevées, pour essayer d'allouer un minimum possible de matériel dédié. Cela est fait parce que les poids de la fonction de coût dirigent l'algorithme vers la minimisation du coût total, lequel est largement dominé par la quantité de matériel employée. Ainsi, l'algorithme "découvre" que le coût peut être minimisé en utilisant moins de ressources du FPGA et en gardant plus de noeuds dans la mémoire (alloués au logiciel). Alors, un FPGA plus petit et moins cher que celui disponible pouvait être utilisé, réduisant ainsi le coût d'implémentation du matériel et du système tout entier. En réglant les valeurs des poids dans la fonction de coût, nous pouvons aussi bien faire de sorte que la

minimisation de la période ou de la puissance soit la priorité.

Le graphe de la figure 0.3 montre le comportement des périodes obtenues pour les nombreuses solutions, en fonction de leurs contraintes correspondantes, démontrant qu'il n'y a pas un effort de l'algorithme pour les minimiser, mais seulement pour les garder au dessous des contraintes, comme le fait la version 1.0.

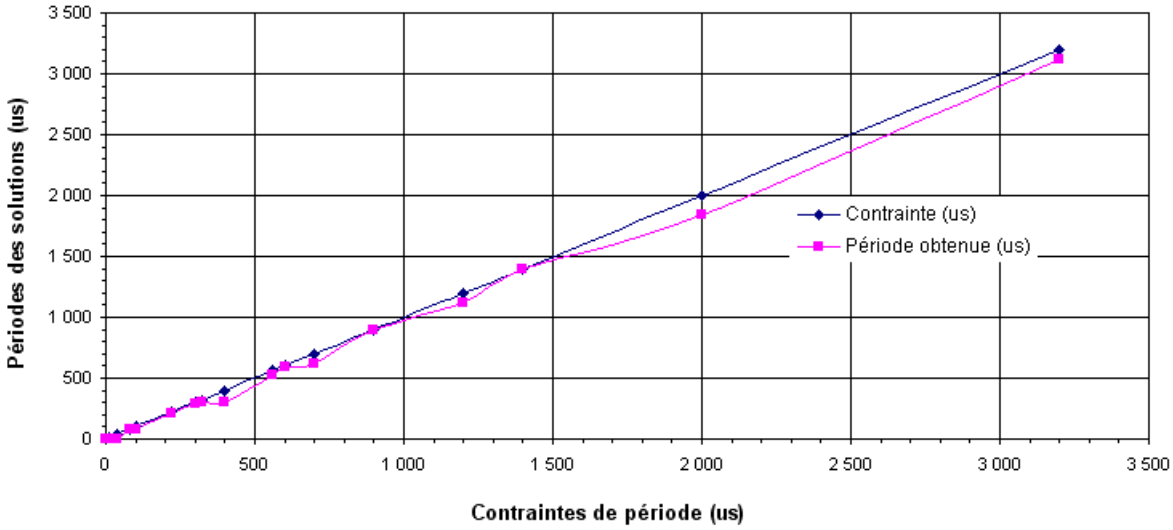


FIG. 0.3: Valeurs de période du récepteur *Rake* partitionné par la version 2.0.

Le graphe de la figure 0.4 montre les coûts monétaires des solutions avec leurs contraintes, en fonction des périodes des solutions, ce qui démontre l'efficacité de l'algorithme en minimiser ces coûts. La réduction de coût obtenue en passant de la solution entièrement matérielle (2.931,4 $) à la solution entièrement logicielle (103 $) est de 28,5 fois. Dans le tableau 4.7, l'utilisation de contraintes de coût beaucoup plus grandes que les coûts réels du système montre que l'algorithme de partitionnement n'explore pas la relâche permise au coût pour minimiser le temps d'exécution, mais plutôt il minimise le coût dans tous les cas.

Les valeurs de coût trouvées par la version 2.0 sont assez inférieures à celles fournies par la version 1.0, grâce à l'emploi d'un FPGA dans le processus de partitionnement, avec la prise en compte précise de l'occupation de ses ressources internes, tandis que la version 1.0 utilise un ASIC. Puisque des valeurs typiques de coût pour des ASIC's ont été employées, les résultats des deux versions reflètent le fait que le coût d'implémentation sur un ASIC est beaucoup plus important que celui sur un FPGA[1]. Mais le plus relevant dans le graphe de la figure 0.4 ne sont pas les valeurs numériques mais plutôt le comportement du coût des solutions en fonction des valeurs des contraintes.

[1]Pour des petits volumes de production

Le graphe de la figure 0.5 montre le comportement des valeurs de puissance des solutions générées, qui se concentrent un peu au-dessus de la somme des puissances statiques du FPGA (645 mW) et du processeur (62 mW), toujours présentes dans toutes les solutions hétérogènes. Seulement dans la solution entièrement logicielle la valeur de puissance est assez différente des autres, égale à 62 mW.

Le graphe de la figure 0.6 montre à la fois les valeurs de temps d'exécution, coût et puissance des solutions, normalisées par ses contraintes respectives dans chaque solution. Dans ce graphe, le rapport C/C_r est à peu près le moindre parmi les trois, malgré le fait qu'il présente encore des oscillations, tandis que les courbes T/T_r et P/P_r restent habituellement proches de la ligne limite coût/contrainte égale à 1.

Une comparaison entre les résultats des versions 1.0 et 2.0 du partitionnement permet de remarquer que les valeurs de période du *Rake* partitionné sont pratiquement égales pour les deux versions, comme nous pouvions espérer, car la différence entre les deux demeure dans le traitement des coûts et contraintes, et pas dans l'ordonnancement. Cela est démontré dans la figure 0.7. Les valeurs de coût produites par les deux versions, malgré leur grande différence numérique expliquée auparavant, possèdent à peu près le même comportement de décroissance exponentielle ; néanmoins, la courbe de coûts de la version 2.0 est plus stable que celle de la version 1.0, ce qui peut être aperçu dans la figure 0.8.

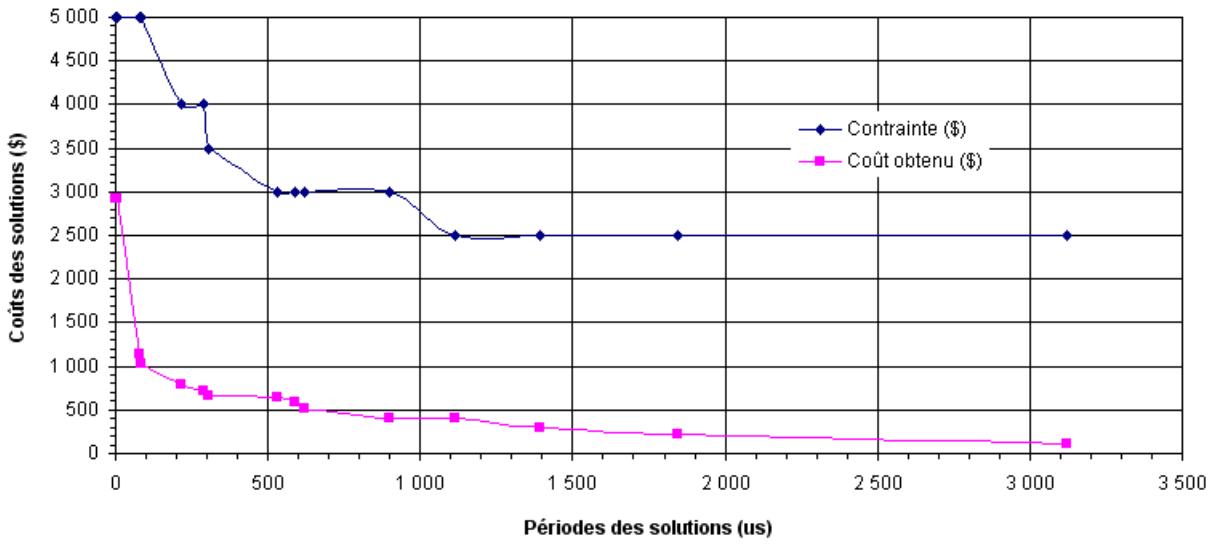


FIG. 0.4: Valeurs de coût du récepteur *Rake* partitionné par la version 2.0.

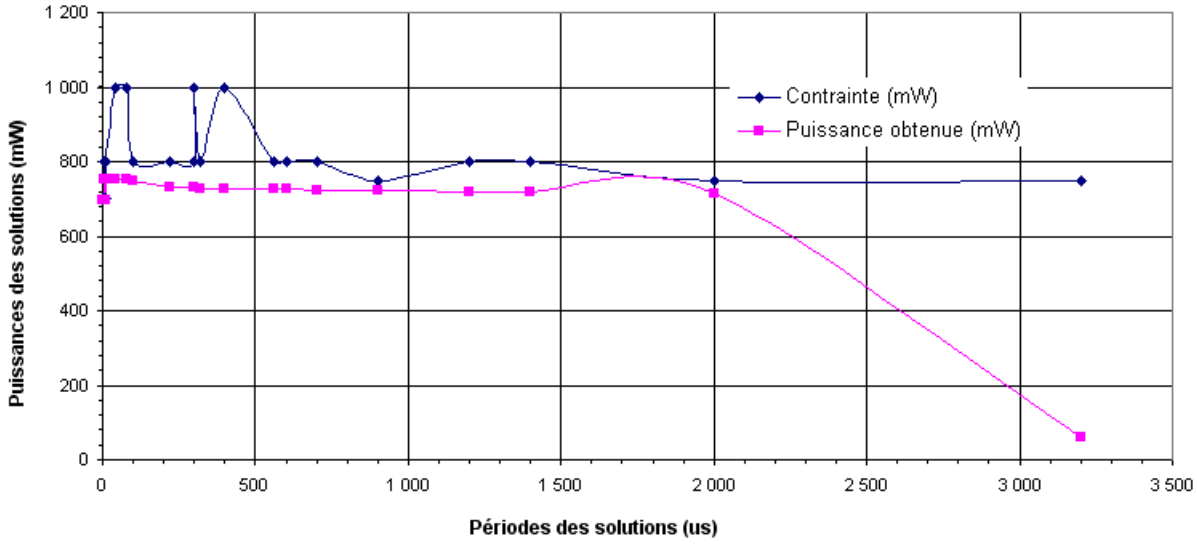


FIG. 0.5: Valeurs de puissance du récepteur *Rake* partitionné par la version 2.0.

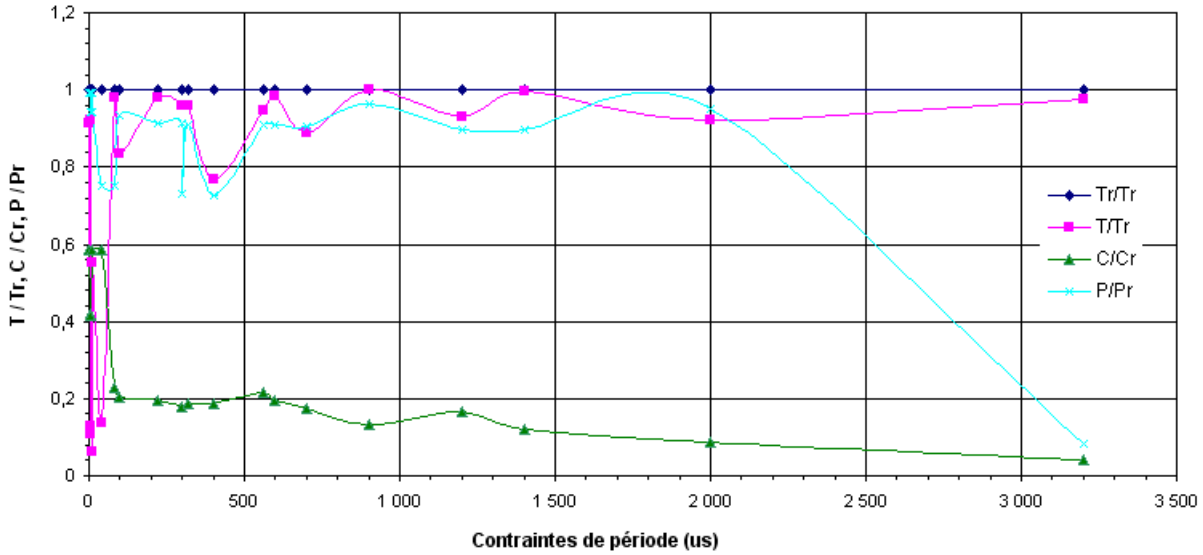


FIG. 0.6: Valeurs normalisées de période, coût et puissance pour le *Rake*, obtenues par la version 2.0.

Version 3.0

Pour le test de la version 3.0 de l'algorithme de partitionnement avec l'application *Rake*, il a été nécessaire de modifier la taille déclarée pour le FPGA dans le fichier `TimeCost.xml`. Puisque le *Rake* est trop petit par rapport à la taille réelle du FPGA, nous avons donc déclaré un PE "FPGADin" avec une quantité assez plus petite de LUT's que la réelle (juste 500), de sorte que juste quelques noeuds du *Rake* (qui prend 824 LUT's) suffisent pour les occuper tous. De cette façon, la condition où un noeud à être alloué demande plus de ressources que celles disponibles dans le FPGA est atteinte durant

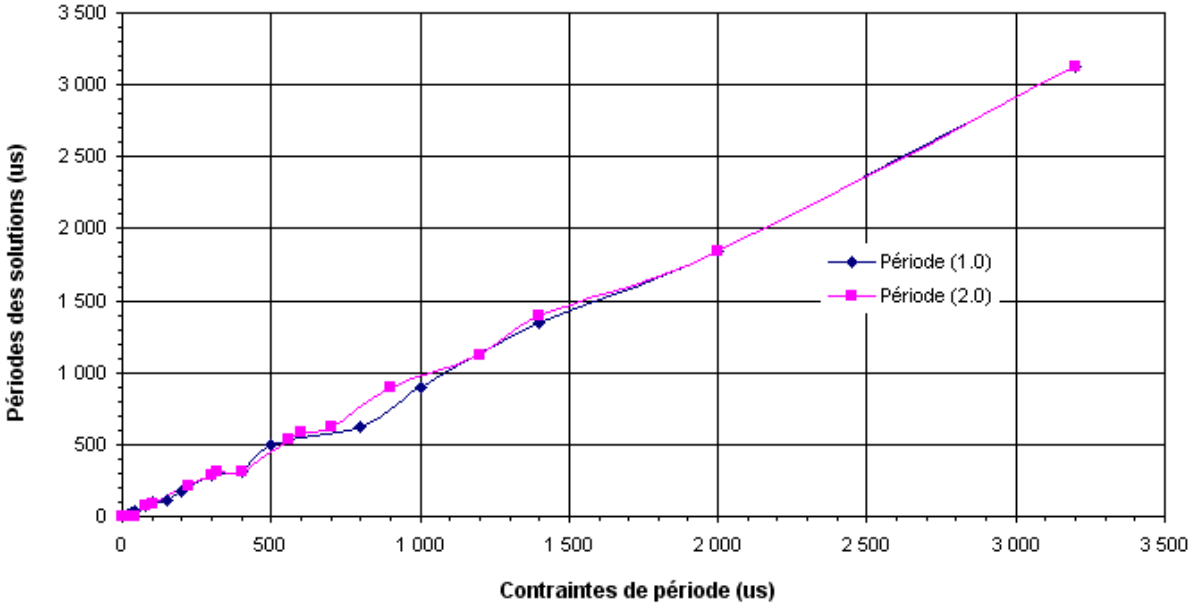


FIG. 0.7: Valeurs de période du récepteur *Rake* partitionné par les versions 1.0 et 2.0.

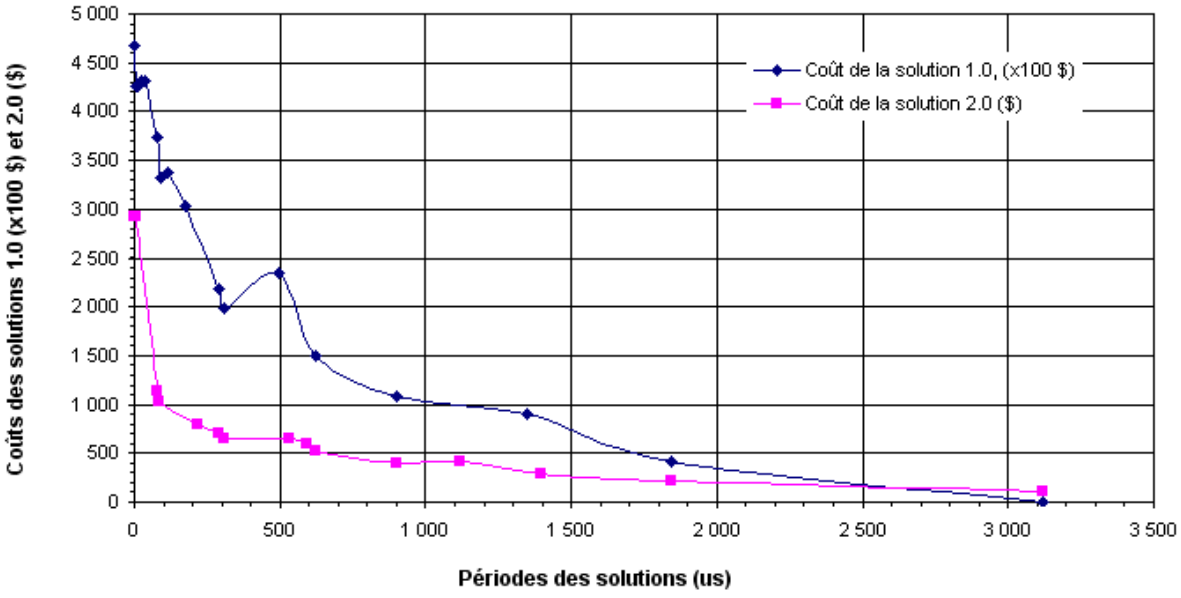


FIG. 0.8: Valeurs de coût du récepteur *Rake* partitionné par les versions 1.0 et 2.0.

l'ordonnancement, ce qui entraîne le besoin d'ordonner une reconfiguration dynamique. Le temps de reconfiguration est arbitré à 1 s et sa consommation de puissance est fixée à 500 mW. Après exécution du partitionnement pour le *Rake* avec des contraintes de période, coût et puissance de 700 ns, 5.000 $ et 700 mW respectivement (pour forcer l'allouement de l'application en matériel), le résultat final a montré l'ordonnancement du noeud de reconfiguration entre deux noeuds du graphe du *Rake*, en raison de la limitation de LUT's du FPGA. Donc seulement 500 LUT's ont été employés, de façon multiplexée

dans le temps, au lieu de 824. Néanmoins, à cause de la durée et de la consommation de puissance de la reconfiguration, les contraintes du fichier `mode.xml` n'ont pas été respectées : l'ordonnancement total calculé pour le *Rake* a été de 1.000.000.638 ns, et la consommation de puissance a été de 1.194,73 mW.

Version 4.0

Dans l'exemple du récepteur *Rake*, il y a un noeud `tri_mult` qui représente un multiplieur spécifique, pour un cas particulier de multiplication d'une variable complexe par une constante complexe. Dans la version en matériel du *Rake*, ce noeud est implémenté de façon beaucoup plus simple qu'une multiplication générique, laquelle est réalisée par un noeud de type `mul_c`. Mais en logiciel, les deux types de multiplications sont réalisés par le biais de l'opérateur standard de multiplication, et donc elles constituent des noeuds du même type générique `mul_c` ; en logiciel il n'y a donc pas d'alternatives d'implémentation pour ce noeud.

Au cas où l'algorithme décide d'allouer et d'ordonner cette opération de multiplication particulière en matériel, nous pouvons lui donner l'option de choisir entre le noeud générique `mul_c` et le noeud spécifique `tri_mult`. Pour cela il suffit de déclarer, dans la table du FPGA dans le fichier `TimeCost.xml`, deux instances consécutives d'un noeud de type `mul_c` : une appelée `mul_cI0_0` avec les valeurs caractéristiques du noeud `mul_c`, et une autre appelée `mul_cI0_1` avec les paramètres du noeud `tri_mult`. En trouvant deux instances consécutives d'un même type de noeud, dont les noms soient différents seulement par le chiffre après le deuxième caractère de soulignement ('_'), l'algorithme saura qu'il s'agit d'alternatives d'implémentation mutuellement exclusives, c'est-à-dire, parmi lesquelles seulement une pourra être ordonnée. En partitionnant le *Rake* avec la même fonction de coût de la version 2.0, l'analyse de la solution montre que l'algorithme choisit l'implémentation `mul_cI0_1`, dû au fait que celle-ci prend moins de ressources et possède un moindre coût que l'alternative `mul_cI0_0`. Alors la version 4.0 est dotée de la capacité de choisir parmi plusieurs alternatives d'implémentation pour un même noeud, selon la priorité de la fonction de coût.

Conclusions

L'objectif de cette thèse de doctorat a été de proposer, développer et valider un algorithme de partitionnement matériel/logiciel amélioré, à partir de l'intégration de techniques développées dans des recherches antérieures sur ce domaine, ce qui constitue une stratégie de travail nouvelle, au vu de ce qui avait été publié dans la littérature [23]. L'ana-

lyse de plusieurs algorithmes de partitionnement déjà publiés a mené à la détermination de critères de qualité pour cette classe de logiciels, et de trois optimisations relevantes à implanter et évaluer dans un algorithme de partitionnement choisi comme référence. Cet algorithme de base, proposé par H. Oh e S. Ha [16] et intégré à l'outil de *Codesign* PEaCE [17], a été expliqué en détail dans la Section 2.4.3. Pour rendre possible le développement d'un travail autonome et concentré exclusivement sur l'algorithme, son code source a été extrait de PEaCE et migré vers l'environnement Microsoft Visual C++.

Après la correction des principaux problèmes existants dans l'algorithme de partitionnement de base, nous l'avons rendu opérationnel et nous l'avons nommé "version 1.0". Nous avons réalisé ensuite les successives optimisations proposées dans cette thèse, décrites ci-dessus.

Les tests des versions optimisées de l'algorithme de partitionnement de base démontrent les profits obtenus avec l'intégration de caractéristiques de qualité d'autres algorithmes de partitionnement proposés dans la littérature. La version 2.0 s'est avérée efficace pour minimiser le coût des solutions partitionnées, par le biais de l'ajout d'une fonction de coût englobant des coûts et des contraintes et évaluée à chaque itération du partitionnement. Cette version peut aussi être adaptée pour minimiser le temps d'exécution ou la consommation de puissance des solutions, selon les valeurs attribuées aux poids de la fonction de coût. Les versions 3.0 et 4.0 ont ajouté de nouvelles fonctionnalités à l'algorithme initial, lui permettant de prendre en compte la reconfiguration dynamique du FPGA dans l'ordonnancement et de choisir l'alternative d'implémentation pour un noeud de façon à mieux satisfaire la fonction de coût.

Le temps de partitionnement dépend non seulement de la complexité de l'application, en termes de quantité de noeuds et densité du graphe, mais aussi des contraintes : comme l'algorithme partitionneur part toujours d'une solution entièrement logicielle, il résulte que de contraintes plus agressives (qui exigent que la majorité des noeuds soit désignée au matériel) demandent un temps de partitionnement plus élevé que de contraintes plus conservatives (qui soient satisfaites avec peu de noeuds en matériel). Une solution entièrement logicielle est obtenue dans une seule itération de partitionnement. Les temps d'exécution des partitionnements pour les applications *Rake* et LDPC ont varié entre 1 s et 8 s. Même si la version 4.0 a contribué à la complexité de l'algorithme initial, cela n'a pas engendré une augmentation remarquable du temps de partitionnement pour les exemples de test.

L'hypothèse principale de ce travail a ainsi été prouvée, selon laquelle il serait possible d'intégrer des caractéristiques avantageuses spécifiques de quelques algorithmes, à un algorithme de base, de façon cumulative et mutuellement compatible, pour le développement d'un seul algorithme de partitionnement amélioré. Il a été aussi démontré que

cette démarche peut mener à des améliorations de fonctionnalité dans l'algorithme, en fournissant un meilleur contrôle sur les systèmes hétérogènes résultants du partitionnement, qui auront potentiellement une meilleure performance et un moindre coût que ceux issus de l'algorithme de base. L'autre hypothèse vérifiée est celle selon laquelle il serait possible de créer un seul algorithme applicable au partitionnement de systèmes avec du matériel fixe et du matériel reconfigurable, qui déciderait sur l'exploitation ou pas de la reconfigurabilité de matériel, selon l'intérêt.

Les résultats obtenus pour les deux applications de test utilisées suggèrent la compatibilité entre l'algorithme choisi comme point de départ pour le travail et les trois techniques d'optimisation du partitionnement implantées. La proposition, l'implémentation et l'évaluation des effets des trois optimisations proposées, faites dans l'algorithme de partitionnement original, constituent la contribution de cette thèse de doctorat.

Capítulo 1

Introdução

1.1 Implementação de sistemas heterogêneos

Os sistemas eletrônicos digitais embarcados (*embedded systems*) desenvolvidos atualmente apresentam, como característica principal, a utilização de *software* (SW) em conjunto com o *hardware* (HW), enquadrando-se portanto na categoria de sistemas computacionais. Porém, esses sistemas são de natureza dedicada, isto é, voltados a uma aplicação específica e com um grau limitado de programabilidade pelo usuário da aplicação, diferenciando-se assim dos sistemas de computação de propósito geral, como os computadores tradicionais, nos quais a aplicação pode ser inteiramente programada pelo usuário final [24].

Os sistemas embarcados onde o processamento de uma aplicação qualquer é feito conjuntamente em componentes dos domínios de implementação físico e lógico, são denominados de **sistemas heterogêneos** ou **sistemas híbridos**. De modo geral, podem-se fazer as seguintes comparações entre as duas implementações [1, 2]:

1. A execução de um algoritmo em *hardware* dedicado apresenta quase sempre um **desempenho** (velocidade) superior em comparação à sua execução em *software*. Por exemplo, geralmente é mais eficaz implementar operações sobre *bits* em *hardware* dedicado, pois elas tomam muito tempo para serem executadas como seqüências de instruções, na maioria dos processadores de *software*.

2. Por outro lado, o **custo para uma solução inteiramente em *hardware*** dedicado costuma ser muito mais alto, em termos do tempo e do custo de projeto. O custo total da implementação de um sistema pode ser reduzido utilizando-se processadores de *software* de uso geral previamente disponíveis, reduzindo-se ao mesmo tempo o tamanho e a complexidade do(s) componente(s) de aplicação específica. Por

exemplo, operações que envolvem acesso à memória e interface com o usuário tipicamente não têm restrições, de modo que elas podem ser relegadas a um programa executando em um microprocessador, reduzindo assim a quantidade de *hardware* dedicado necessária para a implementação do sistema.

3. A implementação em *software* tende a apresentar **menor tempo de desenvolvimento e menor custo** para a mesma funcionalidade, porém quase sempre com desempenho inferior em comparação com uma implementação em *hardware*. Portanto, as partes do sistema que exigem alto desempenho devem ser implementadas sob a forma de componentes de *hardware*.

4. O *hardware* tradicionalmente é fixo e imutável após sua fabricação, enquanto o *software* oferece a vantagem da **flexibilidade**, por ter uma grande facilidade de modificação, facilitando assim a evolução do sistema. Assim, as partes do sistema que necessitarem de maior flexibilidade e facilidade de reprogramação devem ser implementadas como componentes de *software*.

A implementação de sistemas de forma heterogênea, isto é, em *hardware* e *software*, já se mostrou indispensável por permitir aliar-se a propriedade de flexibilidade intrínseca às soluções em *software* aos altos desempenhos obtidos pelas soluções baseadas em *hardware* dedicado. A principal motivação para o desenvolvimento de tais sistemas é buscar posicionar o sistema embarcado em um ponto ótimo da sua curva de custo x desempenho. Comprovando a importância estratégica desse novo paradigma, pode-se encontrar uma grande variedade de métodos de particionamento *hardware/software* na bibliografia [1–16].

Recentemente, o paradigma apresentado no item 4 acima vem sendo quebrado para vários tipos de sistemas, com a utilização de circuitos integrados de *hardware* reconfigurável em campo pelo usuário (representados principalmente por FPGA's), que trazem a flexibilidade também para o domínio do *hardware*. Tais componentes abrem muitas possibilidades inéditas de gerenciar-se a flexibilidade e o desempenho de uma aplicação, pois permitem alterar-se as partes do *hardware* de um sistema com a mesma facilidade com que se altera seu *software*. Dessa forma, eles oferecem um compromisso entre desempenho e flexibilidade que é impossível de ser obtido com *hardware* comum.

A questão principal no projeto de sistemas heterogêneos é encontrar um ponto ótimo entre uma solução totalmente em *hardware* e uma solução totalmente em *software*, onde o sistema alcance o desempenho exigido na sua especificação, mas com a mínima utilização de área de *hardware* dedicado, o que contribuirá decisivamente para um custo total mínimo do sistema, tanto em termos monetários quanto em tempo de desenvolvimento. O tamanho do código de *software* tem muito pouco impacto sobre a área do sistema por

causa da alta densidade alcançada pelas memórias atuais, que não se constituem em um *hardware* dedicado. Também devido às pressões de tempo para o lançamento do produto no mercado, bem como a uma possível melhor adequação de algumas partes das aplicações para serem implementadas em *hardware* ou em *software*, já se tornou muito comum que sistemas embarcados tenham implementações mistas de *hardware* e *software*.

A área de *Hardware/Software Codesign* surgiu no início da década de 1990, a partir da percepção de que era necessário desenvolver-se uma forma integrada e concorrente para se projetar o *hardware* e o *software* para um mesmo sistema, explorando a sinergia entre ambos. Com o uso de um método de *Codesign*, os desenvolvimentos do *hardware* e do *software* passam a ser feitos em conjunto e seguindo um mesmo fluxo de trabalho unificado, quando antes eram feitos por caminhos separados. Desde então, a área de *HW/SW Codesign* tem sido objeto de muitas pesquisas, acadêmicas e industriais, com o objetivo principal de auxiliar os projetistas na etapa mais crítica do projeto de sistemas heterogêneos: o particionamento. Nesta etapa são tomadas as decisões sobre quais partes da especificação do sistema serão realizadas em *hardware* e quais serão realizadas em *software*, levando a implementações heterogêneas para os sistemas. Esse processo, além de definir a arquitetura final do sistema e a sua divisão entre os domínios de *hardware* e *software*, também define a seqüência de execução das tarefas que compõem a aplicação. Por causa disso, o particionamento é o processo central e mais importante de qualquer método de *Codesign*, constituindo-se na etapa cujos resultados têm o maior impacto sobre o custo e o desempenho finais do sistema sendo projetado.

Atualmente encontram-se na literatura os mais variados tipos de algoritmos para particionamento automático em *hardware* e *software*, para os mais diversos tipos de aplicações e arquiteturas de sistemas computacionais. Uma visão geral do *HW/SW Codesign*, detalhando as etapas do projeto típico de um sistema heterogêneo, é dada em [25].

1.2 O objeto da pesquisa

O tema da pesquisa realizada nesta tese de doutorado é a etapa de particionamento do processo de *Hardware/Software Codesign*, mais especificamente os algoritmos para a execução dessa etapa. São estudados algoritmos heurísticos baseados na análise de agendamento.

O levantamento bibliográfico do estado da arte revelou que não tem havido uma preocupação sistemática das comunidades de desenvolvedores de algoritmos de particionamento em gerenciar os conhecimentos já produzidos em pesquisas anteriores. Mais especificamente, a literatura consultada não informou a respeito da realização de análises sobre

as compatibilidades entre diferentes algoritmos e diferentes técnicas de otimização de sistemas heterogêneos. Em decorrência disso, identificou-se como problema uma carência de estudos sobre a possibilidade de integração dos conhecimentos de particionamento em um único algoritmo. Foram identificadas as seguintes questões não respondidas nos trabalhos representativos do estado da arte:

1. Quais são as boas características de qualidade, dos diversos métodos de particionamento, que são candidatas a serem implantadas como características de um algoritmo ótimo?

2. Quais são os benefícios obtidos dessa integração de características de qualidade de vários algoritmos de particionamento já propostos na literatura?

3. Como verificar (testar) o método assim obtido?

A grande maioria dos artigos revisados nesta tese de doutorado trouxe contribuições originais, e alguns introduziram novos recursos e técnicas inovadoras de otimização do particionamento, mas nenhum deles procurou integrar características de trabalhos anteriores para uma acumulação eficiente do conhecimento sobre o desenvolvimento de algoritmos de particionamento. Com isso, contribuições promissoras feitas no passado muitas vezes não foram reutilizadas em trabalhos posteriores, dificultando a evolução e a continuidade da pesquisa na área. Se essas contribuições tivessem sido incorporadas a novos algoritmos sempre que possível, isto é, sempre que não houvesse conflitos entre elas, isso poderia ter levado a algoritmos melhorados, que forneceriam como resultado sistemas de qualidade superior.

1.3 As hipóteses

A hipótese principal deste trabalho é justamente de que seja possível combinar ou integrar algumas características vantajosas específicas, oriundas de algoritmos de particionamento já publicados na literatura, para o desenvolvimento de um algoritmo único e de melhor qualidade. Tal algoritmo de particionamento aprimorado deve levar à síntese de sistemas heterogêneos também melhorados nos aspectos de desempenho e de custo[1]. Outra hipótese a ser verificada é a de que seja possível integrar os requisitos de algoritmos de particionamento para sistemas com *hardware* fixo e com *hardware* reconfigurável, criando um único algoritmo aplicável aos dois tipos de sistemas. Esse algoritmo de particionamento deve decidir pela exploração ou não da reconfigurabilidade de *hardware*, conforme for mais vantajoso.

[1] O custo para um sistema é definido nos capítulos seguintes

A busca da verificação das hipóteses supracitadas, ainda não comprovadas na literatura, é o indicativo da originalidade da presente tese.

1.4 Objetivos deste trabalho

O objetivo geral desta tese de doutorado é propor e desenvolver um algoritmo de particionamento *hardware/software* aprimorado em relação aos demais algoritmos da literatura, de modo a verificar a validade das duas hipóteses descritas na seção anterior. Tal algoritmo deve ser aplicável a sistemas genéricos em termos de arquitetura-alvo, número de modos de operação e reconfigurabilidade do *hardware*.

Para a execução desse objetivo geral, foram definidos os seguintes objetivos específicos, norteando o fluxo de trabalho a ser seguido durante a realização desta tese.

1. Fazer pesquisa bibliográfica para levantar o estado da arte do particionamento *HW/SW* de sistemas heterogêneos, analisando os vários algoritmos propostos na literatura com essa finalidade, para sistemas de *hardware* fixo e de *hardware* reconfigurável, e identificando suas qualidades e limitações.

2. Definir os critérios de qualidade desejáveis para um dado algoritmo de particionamento.

3. Propor e implementar um algoritmo de particionamento aprimorado, por meio da integração, em um algoritmo de base escolhido, de algumas características vantajosas dos algoritmos de particionamento já publicados.

4. Selecionar as aplicações de teste.

5. Testar e avaliar o método de trabalho e o algoritmo de particionamento realizado, por meio das aplicações de teste, demonstrando os seus aprimoramentos e vantagens em relação ao algoritmo de base inicial.

1.5 Justificativa

Os resultados dos diversos trabalhos descritos na literatura demonstram que o custo e o desempenho finais do sistema heterogêneo particionado dependem, principalmente, da qualidade do algoritmo empregado no seu particionamento. Portanto, a principal justificativa para a realização desta tese é o fato de que a pesquisa e o desenvolvimento de algoritmos de particionamento cada vez mais eficientes são o principal caminho para

obter-se sistemas heterogêneos de *hardware* e *software* com relação custo/benefício cada vez melhor, advindo daí a grande relevância desses algoritmos.

Outra motivação desta tese é alertar para a necessidade de acumular (reutilizar), em algoritmos de particionamento mais novos, técnicas geradas em trabalhos anteriores sobre particionamento, e cuja eficácia já foi demonstrada. Assim procedendo, capitalizam-se conhecimentos e experiências na pesquisa e no desenvolvimento de métodos de particionamento, maximizando os ganhos da contribuição de cada método ao estado da arte. Não consta da literatura pesquisada nesta tese um algoritmo que integre todos os recursos e forneça melhores resultados do que todos os demais algoritmos publicados, para todos os domínios de aplicação.

1.6 Método de pesquisa

O método de pesquisa empregado considera o conjunto de requisitos de qualidade de algoritmos de particionamento pré-existentes como base para a proposta de um algoritmo aprimorado, isto é, reunindo características desejáveis para fornecer um sistema heterogêneo aprimorado. A pesquisa não ficou limitada a uma forma de implementação particular dos sistemas heterogêneos, mas foi independente da mesma, englobando algoritmos para o particionamento de sistemas com *hardware* fixo e com *hardware* dinamicamente reconfigurável. O levantamento bibliográfico do estado da arte [23] permitiu determinar os seguintes critérios de qualidade, normalmente utilizados na literatura, para a classificação e análise comparativa de algoritmos de particionamento *HW/SW* [1–16]:

- Generalidade:
  - quanto à arquitetura-alvo: mono ou multiprocessada, predefinida ou sintetizada;
  - quanto ao domínio de aplicação dos sistemas a particionar: algoritmos "multi-domínio" ou de domínio específico;
  - quanto ao número de modos (aplicações): sistemas mono-modo ou multi-modo;
- Diversidade de requisitos de qualidade do particionamento e do sistema:
  - consideração dos custos de comunicação entre *hardware* e *software* (tempos de comunicação e área das interfaces);
  - consideração de paralelismo de execução entre nós de *hardware* e de *software*;
  - compartilhamento de recursos de *hardware* entre dois ou mais nós da aplicação;

- escolha entre múltiplas alternativas de implementação, em *hardware* ou em *software*, para um mesmo nó;
- capacidade de reconfiguração em tempo de execução (RTR) dos FPGA's (somente para sistemas de *hardware* reconfigurável). Implica no compartilhamento de recursos de *hardware* entre nós do grafo da aplicação;
- capacidade de explorar a reconfiguração parcial do *hardware* (somente para sistemas de *hardware* reconfigurável).

Os critérios de qualidade acima correspondem a diversas formas de reduzir latência, custo e potência de um sistema heterogêneo, e permitem a avaliação da eficácia e da qualidade de cada algoritmo. Foram analisados os resultados obtidos pelos algoritmos de particionamento pesquisados, as condições em que esses resultados foram alcançados e as características mais relevantes (pontos fortes) identificadas em cada algoritmo. Com base nessa análise, escolheu-se um dos algoritmos de particionamento do estado da arte para servir como base para o desenvolvimento da tese, pelo fato de ele satisfazer a mais critérios de qualidade que os demais [23] e ser aplicável a uma arquitetura de sistema genérica (multi-modo e multiprocessada). Nesta tese propõem-se e implementam-se melhorias a esse algoritmo de base escolhido, por meio da incorporação de novas funcionalidades, que consistem em:

1. Consideração detalhada das restrições de PE's e de aplicações, dos custos associados ao consumo de recursos de um FPGA, e dos consumos de potência pelos nós das aplicações. Incorporação de uma função de custo complexa que integre custos e restrições na mesma expressão.

2. Consideração do tempo de reconfiguração do *hardware* no agendamento das aplicações, para sistemas com *hardware* dinamicamente reconfigurável.

3. Trabalho com múltiplas alternativas de implementação para um nó em um mesmo PE.

1.6.1 Experimentação

O algoritmo de particionamento otimizado proposto nesta tese é implementado e avaliado por etapas, permitindo acompanhar-se a sua evolução. Cada implementação de uma nova característica (otimização) no programa particionador, que o deixe em um estado funcional, resulta em uma nova versão do mesmo, para ser testada individualmente. Os critérios de avaliação da qualidade do algoritmo são: a complexidade computacional do

particionamento e o desempenho e o custo das aplicações particionadas, em termos de tempo de execução, custo monetário total e consumo de potência.

Para o teste das versões aprimoradas do algoritmo de particionamento, decidiu-se utilizar aplicações no domínio de telecomunicações. As razões para esta escolha foram: o fato de este domínio ter apresentado uma evolução muito rápida nas últimas décadas e assim possuir um grande interesse acadêmico e industrial atualmente; e o fato de ele empregar algoritmos de Processamento Digital de Sinais (PDS) com exigências fortes de alto desempenho e baixo custo, que portanto poderiam se beneficiar grandemente de uma implementação heterogênea. Além disso, a evolução acelerada dos padrões e algoritmos no campo das comunicações móveis torna-o promissor para o emprego de componentes reconfiguráveis, com o objetivo de viabilizar a atualização rápida dos produtos. Por essas razões, as aplicações de teste escolhidas para este trabalho foram um receptor *Rake* para sistemas de comunicações móveis CDMA e um decodificador de canal do tipo LDPC. A plataforma heterogênea considerada para o particionamento foi a placa de desenvolvimento Nios II / Stratix II da Altera, sobre a qual as aplicações precisariam ser caracterizadas antes de serem particionadas.

Com a conclusão da implementação do algoritmo de particionamento e sua avaliação com aplicações reais, é então possível responder às perguntas-objetos da pesquisa, e testar a hipótese de que "o novo método assim gerado levará a sistemas heterogêneos otimizados".

1.7 Considerações finais

Os objetivos 1 e 2, descritos na Seção 1.4, foram concluídos nos dois Projetos de Pesquisa [21,25] apresentados ao Programa de Pós-Graduação em Engenharia Elétrica da UFCG. Os trabalhos desenvolvidos até o presente foram publicados em nove artigos de conferências, sendo quatro em congressos nacionais [26–29] e cinco em congressos internacionais [30–34].

1.8 Organização do trabalho

Este trabalho está organizado da seguinte maneira:

O Capítulo 2 aborda a etapa de particionamento e seus principais aspectos: a representação do sistema, as tarefas realizadas e a comunicação entre as partições de *hardware* e *software*. Apresenta-se um resumo do estado da arte de algoritmos de particionamento, para sistemas com *hardware* fixo e com *hardware* reconfigurável. Em seguida, faz-se uma descrição detalhada do algoritmo de particionamento da ferramenta de *Codesign*

acadêmica PEaCE [17], escolhido como base para a implementação das otimizações propostas nesta tese. Explicam-se as razões para a escolha deste algoritmo e o método de trabalho adotado com o código. Cria-se uma função de custo simples para ele, visando atingir um nível mínimo de funcionalidade que permita a avaliação do impacto das otimizações. Esta versão do programa passa a ser chamada de "versão 1.0".

O Capítulo 3 apresenta as otimizações a serem realizadas na versão 1.0 do algoritmo de particionamento, levando a três versões sucessivas. Dentre elas, as mais importantes são as capacidades de manipular o consumo de recursos internos e o tempo de reconfiguração de componentes de *hardware* dinamicamente reconfigurável, como FPGA's, para o particionamento de sistemas embarcados com esses componentes.

O Capítulo 4 mostra os resultados da aplicação das quatro versões do algoritmo para particionar duas aplicações de PDS. Explicam-se os detalhes da criação dos arquivos de entrada para o particionamento e as análises das soluções heterogêneas geradas para as aplicações.

O Capítulo 5 apresenta as conclusões deste trabalho e as sugestões para sua continuidade.

Capítulo 2

Algoritmo-Base de Particionamento

2.1 Introdução ao particionamento *HW/SW*

O particionamento é o processo principal de qualquer método de *HW/SW Codesign*, e em princípio sua meta é obter uma partição ótima da funcionalidade de um sistema entre *hardware* e *software*. Isso significa buscar uma partição que leve o sistema ao máximo desempenho dentro de um dado limite de custo, ou ao mínimo custo tal que ainda atenda às exigências de desempenho. Este último objetivo é o mais freqüentemente adotado para os algoritmos de particionamento descritos na literatura; em outras palavras, manter o desempenho do sistema acima das restrições dadas, buscando ao mesmo tempo a minimização do uso de recursos de *hardware* dedicado. O projetista do sistema deve tentar implementar tanta funcionalidade quanto possível em *software*, mas ainda atendendo às restrições de desempenho do sistema. Na prática, geralmente o particionamento é feito com o objetivo de encontrar apenas uma solução satisfatória para a implementação do sistema, isto é, uma solução que atende às restrições impostas mas não é necessariamente a partição **ótima**.

O particionamento é a etapa do *Codesign* que decide quais funções do sistema serão realizadas em *hardware* e quais serão realizadas em *software*, além de poder definir a arquitetura final do sistema e o agendamento de execução das tarefas da aplicação. Por ser a etapa que mais afeta o custo e o desempenho finais do sistema sendo projetado, a qualidade e a eficiência do algoritmo de particionamento são determinantes para a qualidade e eficiência do processo de *HW/SW Codesign* como um todo. Inúmeras propostas de algoritmos de particionamento têm sido publicadas, com os mais variados enfoques e abordagens, e todas têm levado a resultados satisfatórios ou ótimos dentro de seus sub-domínios, sem que tenha sido proposto, até o presente, um algoritmo ou método totalmente superior aos demais, em todos os domínios de aplicação. O desenvolvimento de

algoritmos de particionamento cada vez melhores é o principal caminho para se chegar a sistemas heterogêneos com qualidade e custo ótimos.

O problema do particionamento entre *hardware* e *software* pode ser definido de forma ampla como segue [35]: tomando-se como entrada uma especificação de sistema consistindo de processos interagentes, produz-se como saída uma arquitetura, uma atribuição de funções para componentes e o agendamento da execução dos mesmos, sob limitações de projeto específicas. A solução gerada pelo algoritmo de particionamento nunca pode violar restrições de capacidade de recursos (área máxima de *hardware*, tamanho máximo de *software*), isto é, uma implementação só deve ser escolhida se existirem recursos em quantidade suficiente para implementá-la.

A teoria do particionamento é abordada de modo geral na Seção 2.2, tratando-se dos seus principais aspectos: a representação do sistema, a arquitetura-alvo, o mapeamento, o agendamento e a comunicação entre as partições de *hardware* e *software*. A Seção 2.3 apresenta o estado da arte de algoritmos de particionamento para sistemas com *hardware* fixo e com *hardware* reconfigurável. Em seguida, na Seção 2.4, descreve-se em detalhes o algoritmo de particionamento da ferramenta de *HW/SW Codesign* acadêmica PEaCE [17], proposto por H. Oh e S. Ha [16], escolhido como base para a realização desta tese. A escolha de utilizar este algoritmo é justificada, seus arquivos de entrada são descritos e o seu funcionamento é explicado de forma detalhada, enfatizando o algoritmo de agendamento BIL [18], que é o seu núcleo principal. As explicações são ilustradas por meio de exemplos de execução.

Finalmente, a subseção 2.4.6 descreve a forma escolhida para trabalhar-se com o código do algoritmo de particionamento de PEaCE, para os fins desta tese. Explica-se a principal correção realizada no código original para atingir-se um nível mínimo de funcionalidade, que permitisse a ele ser tomado como ponto de partida para a implementação das otimizações propostas neste trabalho. Uma descrição detalhada de todas as limitações e erros encontrados no algoritmo original de PEaCE, juntamente com as melhorias e correções realizadas, é dada em [23].

Neste trabalho, adotam-se as seguintes definições:

Nó É uma unidade computacional de um grafo, tarefa ou aplicação qualquer. Um nó realiza uma operação de qualquer complexidade, definida pelo usuário. Nós podem ser **atômicos**, isto é, podem representar as menores operações computacionais de um grafo, ou **hierárquicos**, quando contém subgrafos internos de nós, formando uma hierarquia.

Tarefa Tarefas são processamentos de sinal de alto nível, como por exemplo FFTs ou DCTs, utilizados em algoritmos. Uma tarefa é, ela mesma, um nó hierárquico do

grafo de uma aplicação. Uma tarefa, por sua vez, consiste em um ou mais grafos de nós computacionais logicamente relacionados, que produz um resultado útil para a execução de uma aplicação.

Aplicação ou Modo É um processamento bem definido e que produz um resultado útil do ponto de vista do usuário do sistema. Neste trabalho, uma aplicação é um algoritmo completo de processamento de sinais, constituída por um grafo composto de uma ou várias tarefas, segundo o modelamento feito pelo usuário, e essas tarefas, quaisquer que sejam suas complexidades internas, são formadas por grafos de nós até o nível mais baixo de hierarquia. Exemplos de aplicações são algoritmos de compressão de imagem ou vídeo, como JPEG ou MPEG.

Sistema É o conjunto de uma ou mais aplicações que se deseja particionar, com vistas a uma implementação heterogênea. O grafo do sistema é fornecido como entrada para o algoritmo de particionamento. O sistema também pode denotar a entidade física que executa esse conjunto de aplicações. Um sistema que pode executar múltiplas aplicações é dito ser um **sistema multi-modo**.

Elemento Processador Também chamado de PE (do inglês *Processing Element*), um elemento processador pode significar tanto um processador de *software* de uso geral (microprocessador, CPU ou núcleo processador embarcado) ou de aplicação específica (ASIP ou DSP), quanto um processador puramente em *hardware* (ASIC ou FPGA). Os PE's são os componentes processadores autônomos disponíveis, que são candidatos a compor a arquitetura de um sistema.

Recurso Elementar É um bloco interno individual de um PE. Um processador de *hardware* (ASIC ou FPGA) pode possuir múltiplos recursos elementares internos, que podem ser alocados individualmente aos nós das tarefas sendo particionadas. Os elementos lógicos, *bits* de memória, multiplicadores e outros blocos internos preexistentes em um FPGA são exemplos de recursos elementares: não são blocos funcionais da aplicação (nós), mas sim os blocos físicos básicos usados na implementação de qualquer bloco funcional mapeado no FPGA.

Recurso Designa um recurso computacional funcional do ponto de vista da aplicação, que tanto pode ser um PE quanto a implementação de um nó em um PE, a qual consome recursos elementares desse PE. Portanto, um par (nó, PE) do algoritmo de particionamento é considerado um recurso. Todo PE é um recurso, mas nem todo recurso é necessariamente um processador autônomo.

Custo de um Nó O conceito de custo da implementação de um nó em um PE é muito amplo, podendo ser muito difícil de precisar. Para esta tese, define-se o "custo de um nó em um PE" como a somatória das quantidades de recursos elementares desse PE consumidas pelo nó, normalizadas pelas quantidades totais disponíveis desses recursos, com o resultado multiplicado pelo custo monetário (preço) do conjunto de recursos.

2.2 Etapas do particionamento

2.2.1 Descrição do sistema

Na grande maioria dos algoritmos de particionamento propostos na literatura, a aplicação a ser particionada é inicialmente descrita pelo projetista em uma linguagem de *software* (C, C++), ou em uma HDL (VHDL, Verilog), ou em uma linguagem de modelamento de sistemas (SDL, SystemC). Para se compilar um programa descrito em uma linguagem de programação seqüencial, como C, para execução em *hardware*, a descrição comportamental de entrada do sistema precisa ser primeiro convertida para um formato intermediário na notação de grafos, denominado CDFG (Grafo de Fluxo de Dados e Controle). Os CDFG's constituem um modelo de computação orientado a atividades, isto é, que se concentra na definição da seqüência de transformações necessárias sobre os dados, para a geração do resultado [36].

Um grafo, denotado por $G = (N, E)$, é formado por dois conjuntos, N e E, onde N é um conjunto de **vértices**, também chamados de **nós** em um contexto de redes; e E é um conjunto de pares ordenados de vértices, chamados de **arcos** (**conexões** ou **enlaces**), com cada arco ligando dois e somente dois nós. O número de vértices do grafo, indicado por $|N|$, é a **ordem do grafo**. Os grafos de interesse para a representação de algoritmos são os **grafos orientados**, aos quais pode-se associar uma relação de dependência entre os vértices, isto é, uma seqüência de execução e de passagem de mensagens entre seus nós. Os CDFG's são sempre grafos orientados.

Nos grafos utilizados para o modelamento de algoritmos, cada nó n_i representa uma operação ou conjunto de operações (um cálculo, processo, bloco de código, instrução simples), de acordo com a granularidade do grafo, e cada arco a pode ser designado sem ambigüidade por um par ordenado de nós $(n_i,\ n_j)$, com extremidade inicial (origem) no nó n_i e extremidade final (destino) no nó n_j. Dessa forma, o arco a especifica uma dependência de dados e/ou controle entre esses dois nós e representa as relações de entrada e saída entre eles. Para o arco $a = (n_i,\ n_j)$, n_j é um nó sucessor de n_i, e n_i é um nó antecessor de n_j, e diz-se então que n_i e n_j são nós vizinhos ou adjacentes (possuem um

arco em comum). O número total de arcos que entram e saem de um nó é chamado **grau do nó**.

Dois nós quaisquer são conectados por no máximo um arco. Por isso, o número de arcos com origem em n_i torna-se igual ao número de nós sucessores de n_i, e o número de arcos com destino em n_j torna-se igual ao número de nós antecessores de n_j. Portanto, uma outra forma de representar um grafo *G = (N,E)* é substituir o conjunto E de arcos pelas listas $L_{suc}(n_i)$ e $L_{ant}(n_i)$ que representam os conjuntos de nós sucessores e antecessores para cada nó n_i de *N*. Dessa notação pode-se deduzir facilmente o conjunto E de arcos: n_j está em $L_{suc}(n_i)$ se e somente se o arco $(n_i,\ n_j)$ pertence a *E*. Essa notação para um grafo, baseada em **listas de adjacência**, é a notação empregada neste trabalho para a codificação do algoritmo de particionamento em linguagem de programação.

Um **caminho** P de comprimento l é uma seqüência de l nós $\{n_1, n_2, ..., n_l\}$, n_i sendo o nó sucessor de n_{i-1} (se $i > 1$) e o antecessor de n_{i+1} (se $i < l$), ou seja, um caminho é uma seqüência de nós ligados por arcos. Um caminho fechado constitui um **ciclo**. Um nó n_j é dito ser **descendente** de n_i se ele está situado em um caminho qualquer de origem em n_i. Nesse caso, diz-se que n_i é ascendente ou ancestral de n_j. O **fechamento transitivo** de n_i é o conjunto formado por n_i, seus sucessores, os sucessores de seus sucessores, e assim sucessivamente até o final do grafo. Trata-se portanto do conjunto de nós situados nos caminhos de origem em n_i, ou ainda do conjunto de todos os descendentes de n_i.

Um grafo pode também ser **ponderado**, recebendo valores ou pesos em seus arcos, atribuídos por meio de uma aplicação $C : E \rightarrow \Re$. Esse grafo fica então denotado por *G = (N,E,C)*. O peso ou valor numérico C_{ij} associado ao arco (n_i,n_j) representa algum tipo de custo ligado à transição de n_i para n_j: distância, custo de transporte ou de comunicação, tempo de percurso ou probabilidade de transição. A aplicação C é geralmente codificada, nos algoritmos de *HW/SW Codesign*, por meio de uma tabela a ser lida pelo algoritmo. Quando G é ponderado, pode-se definir o **custo de um caminho** entre dois nós de um grafo como a soma dos custos dos arcos que o constituem. Um domínio importante de aplicação é o cálculo do caminho de custo mínimo (ou caminho mais curto) entre dois nós. A busca dos caminhos de custo mínimo geralmente é restrita aos caminhos elementares do grafo. Pode existir, no pior caso, um número enorme de caminhos de mesmo custo ligando dois nós; seria impossível considerar todos. Por isso, os algoritmos clássicos de grafos constroem apenas um dos caminhos possíveis entre aqueles de custo mínimo ligando dois nós [37].

No contexto de representação de algoritmos de sistemas, além do grafo puro e simples, deve-se fornecer também algumas informações adicionais sobre seus nós e arcos, que vai ser usada pelo algoritmo de particionamento para determinar se um nó deve ser colocado em *hardware* ou *software*. Cada nó n_i deve trazer consigo os seguintes valores: estimativas de

área (ah_i) e tempo de execução (th_i) para a sua implementação em *hardware*; tamanho do código compilado (as_i) e tempo de execução (ts_i) para a sua implementação em *software*; o número médio de vezes que n_i é executado; e os conjuntos de variáveis de leitura e escrita de n_i, chamados de *read-set*(n_i) e *write-set*(n_i). Cada arco (n_i,n_j) deve trazer as seguintes informações: o número de amostras de dados N_{ij} enviadas de n_i para n_j; os custos da comunicação de uma amostra de dados entre n_i e n_j através da fronteira HW-SW, especificados pela área de *hardware* de comunicação $ah_{comm}(n_i,n_j)$, pela área de *software* de comunicação $as_{comm}(n_i,n_j)$, e pelo tempo de comunicação $t_{comm}(n_i,n_j)$; e o atraso total de execução máximo permitido para o sistema, D. As quantidades finitas dos recursos de *hardware* e *software* constituem restrições adicionais para o problema.

Finalmente, as razões para se utilizar CDFG's na execução do particionamento são as seguintes: (1) eles conseguem capturar a funcionalidade e as exigências de temporização e área do sistema em um modelo gráfico, no qual os compromissos de particionamento podem ser explorados sistematicamente [2]; (2) eles permitem aplicar-se técnicas de estimação do desempenho do sistema, por meio de agendamento, e verificar-se a consistência das restrições de projeto; (3) eles são independentes de qualquer implementação específica, seja em *software* ou *hardware*, o que permite uma modelagem homogênea do sistema e uma exploração imparcial do espaço de projeto; (4) eles tornam explícita a concorrência inerente da especificação de entrada, e facilitam o raciocínio sobre as propriedades desta; (5) o método de particionamento fica independente da escolha da linguagem de entrada específica, e outras linguagens podem ser utilizadas para a especificação e depois traduzidas para CDFG's; (6) do CDFG obtém-se informações de síntese de *hardware* e *profiling* (perfilamento) de *software* para o particionamento; (7) os grafos suportam a geração de uma descrição de *hardware*, para os blocos movidos para esse domínio de implementação.

2.2.2 Seleção de componentes da arquitetura-alvo

A arquitetura-alvo pode ser sintetizada durante o particionamento pela análise das partições de *software* e *hardware*, ou pode ser imposta pelo projetista ou pelo algoritmo antes do particionamento. Um meio-termo consiste em fazer previamente as escolhas mais estratégicas, definindo a estrutura geral da arquitetura-alvo, e só depois definir as necessidades arquiteturais mais detalhadas, com base nas duas partições. Uma forma de classificar os algoritmos de particionamento é quanto à arquitetura-alvo ser fixa (escolhida *a priori*), ou sintetizável por uma ferramenta de particionamento automático. Um bom algoritmo de particionamento não pode ficar limitado a uma única arquitetura, mas deve ser flexível para contemplar arquiteturas mono e multiprocessadas. O ideal é que a arquitetura-alvo seja genérica e parametrizável, podendo ser constituída de qualquer

quantidade e tipo de processadores de uso geral, DSP's, ASIP's, ASIC's e/ou FPGA's.

Quando há várias alternativas de arquiteturas-alvo possíveis, uma forma de explorar o espaço de projeto é encontrar o melhor particionamento funcional para cada uma, em termos de tempo de execução, área de *hardware*, etc., e depois comparar as arquiteturas-alvo levando em conta outros critérios como preço, consumo de potência, etc., de modo a escolher aquela que for mais adequada. O algoritmo pode gerar vários compromissos arquiteturais diferentes para a mesma aplicação, mudando-se o valor da restrição para o custo total do sistema e então otimizando-se seu desempenho global. Outra forma eficiente de explorar o espaço de projeto é considerando várias curvas de compromisso entre custo / paralelismo / tempo de execução para cada tarefa de uma aplicação, para um projeto ao nível de sistema [38]. O CDFG de cada tarefa é agendado para várias restrições de tempo, gerando as curvas de compromisso.

Segundo [14], há duas grandes classificações para as arquiteturas contempladas pelos métodos de *Codesign*. Os sistemas embarcados heterogêneos distribuídos (DHE) consistem de vários tipos diferentes de PE's, enquanto nos sistemas em um *chip* (SoC), os PE's são representados por núcleos processadores e blocos de *hardware* a serem integrados na mesma pastilha. Ambos os grupos lidam com o mesmo problema: dados os grafos de tarefas de entrada, determinar a arquitetura ótima dentro das restrições de projeto. A diferença está na forma de lidar com os componentes de *hardware*: enquanto o SoC considera muitas possibilidades de implementação de uma mesma tarefa no *hardware* do *chip*, o DHE tende a considerar cada nova possibilidade de implementação de um nó em *hardware* como um PE separado para a tarefa. A granularidade das tarefas no trabalho com DHE é maior do que no trabalho com SoC. A grande largura de banda da comunicação *on-chip* torna o *HW/SW Codesign* bastante vantajoso para SoC's, pois reduz a perda com o tempo de comunicação HW/SW. Outra vantagem dos SoC's é que o núcleo processador pode ser facilmente customizado com relação à aplicação, e também otimizado com coprocessadores de granularidade fina, sem acarretar atrasos de comunicação [38].

2.2.3 Mapeamento

O mapeamento determina em que PE, de *hardware* ou de *software*, escolhido na etapa anterior para fazer parte da arquitetura, cada nó de uma tarefa é mapeado. É o próprio problema de particionamento, em um sentido estrito: encontrar um mapeamento dos módulos funcionais para os PE's.

A principal meta do particionamento HW/SW é mapear todos os blocos da especificação do sistema, decidindo por uma implementação em *hardware* ou em *software* para cada um. Se a especificação for um CDFG, o particionador deve propor um mapeamento

para todos os nós desse CDFG, sempre observando que não é possível dividir um nó, isto é, não é permitida uma implementação mista de *hardware* e *software* para um mesmo nó: cada nó possui um e apenas um mapeamento. A granularidade dos nós é livremente definida pelo usuário, podendo ser variável. O problema do mapeamento é uma função combinatória de ordem 2^N, onde N é o número de nós. A decisão de mapeamento de um nó deve ser baseada em uma avaliação das métricas de interesse para todo o sistema: depende do custo local de cada implementação possível (área, desempenho, potência, ...) e da influência deste mapeamento sobre todos os outros nós.

2.2.4 Agendamento

Agendar significa encontrar o instante de início de execução para cada nó, para uma dada arquitetura-alvo, determinando a ordem de execução de nós em cada PE. Nos algoritmos mais simples de agendamento, a seqüência de execução é determinada estaticamente e fixada, em tempo de execução, em um processador. O problema do agendamento pode ser definido como segue: "Dado o número máximo de passos de tempo, encontrar uma agenda de custo mínimo que satisfaça o conjunto de restrições dadas". O custo a que a definição se refere é o custo de uma solução heterogênea. O agendamento pode visar a minimização de diferentes parâmetros: dos operadores usados pelos nós, das comunicações HW/SW, do tempo ocioso dos recursos, ou do tempo de execução; o objetivo principal do agendamento vai depender do objetivo do particionamento.

O algoritmo de agendamento mais utilizado nos trabalhos de particionamento da literatura é o agendamento de lista (*list scheduling*). Esse tipo de algoritmo agenda os nós executáveis em ordem decrescente de prioridade (um nó é dito "executável" quando seus antecessores já tiverem sido agendados). Para nós de *hardware*, o *list scheduling* prioriza um nó que tenha a maior soma de seu próprio atraso com os atrasos de todos os nós sucessores, permitindo assim que o nó de *hardware* mais crítico seja agendado primeiro. Para nós de *software*, a prioridade é dada a um nó que tenha um sucessor de *hardware* com a mais alta prioridade, permitindo assim que o nó de *software* que leve ao nó de *hardware* mais crítico seja agendado primeiro. A prioridade de um nó para ser executado é também calculada de modo que tantos nós de *software* quanto possível possam rodar em paralelo com um nó de *hardware*.

Outra definição para as prioridades de agendamento dos nós de um CDFG consiste em calculá-las como o inverso do *deadlines* desses nós [38]; nós que têm que encerrar suas execuções em menos tempo (menor *deadline*) recebem maior prioridade. Feito o agendamento dos nós do grafo do sistema, pode-se calcular a latência desse sistema e estimar a concorrência em potencial entre seus nós.

Comunicação e sincronização

No contexto de *HW/SW Codesign*, os dois domínios de implementação coexistem ao mesmo tempo, cada um respondendo por uma parte da execução da aplicação, donde decorre a necessidade de comunicação entre eles. Entretanto, a comunicação de dados e a sincronização entre as partições de *hardware* e de *software* do sistema exigem blocos de interface, que geram atrasos de tempo adicionais. Conseqüentemente, todo algoritmo de particionamento precisa levar em conta os custos dessas comunicações, tanto em termos das interfaces necessárias quanto do tempo extra gasto para ela ocorrer, para uma estimação precisa do custo e do desempenho globais de uma dada solução, principalmente em particionamentos com granularidade fina. Para tanto, é necessário conhecer os tempos da comunicação HW-SW de uma palavra, que dependem dos mecanismos de comunicação e da organização de memória.

Para N nós, 2^N - 2 partições heterogêneas são possíveis; portanto, o pré-processamento dos custos de comunicação exatos não é prático. Ao invés disso, pode-se estimar custos somente para nós adjacentes nos grafos, o que evita uma análise de fluxo de dados global. Como regra geral, reduzir o volume da comunicação entre as partições de *hardware* e de *software* melhora o desempenho global do sistema, mesmo que o PE de *hardware* possa rodar simultaneamente com o PE de *software*.

O tipo e a quantidade de dados a comunicar são estimados pela construção de grafos de fluxo de dados, a partir da especificação. Mas essas trocas de dados não são realmente conhecidas até que o particionamento seja efetuado. O problema é, portanto, retroativo, pois somente após o particionamento é feita a análise da comunicação inter-processos e serão conhecidas as reais necessidades de troca de informação que afetarão os tempos de execução do sistema particionado, podendo recolocar em questão as próprias escolhas do particionamento.

Particionando um mesmo sistema várias vezes, sob diferentes condições iniciais, observa-se que nem sempre os blocos que são os mais executados serão movidos para *hardware*. Blocos que são pouco chamados podem ser deslocados para *hardware* com mais freqüência do que outros que são iterados várias vezes, pois pode acontecer que o ganho de velocidade obtido com o *hardware* não seja grande o bastante para compensar pelo tempo extra de comunicação, a qual também será repetida várias vezes. O que se pode prever é que a maioria dos blocos extraídos para *hardware* corresponda a trechos de cálculo intensivo, se o tempo de comunicação HW/SW for suficientemente pequeno em relação ao ganho de tempo de operação.

Além do tempo de comunicação, há que se considerar também o incremento de área, ocupada pelos elementos de interface necessários para fazer a comunicação, como memórias

intermediárias e roteamento extra. Quando dois nós comunicantes n_i e n_j são particionados em implementações opostas (um em *hardware*, outro em *software*), será necessário inserir um nó de comunicação $nc_{i,j}$ entre eles, desempenhando a função de interface HW/SW.

2.3 Estado da arte

2.3.1 Algoritmos de particionamento com *hardware* fixo

Os primeiros algoritmos para particionamento automático em *hardware* e *software* foram publicados no início da década de 1990, quando também foi criado o termo "*Hardware/Software Codesign*". A seguir é feita uma análise dos principais algoritmos de particionamento para sistemas heterogêneos de *hardware* fixo publicados na literatura e, no final desta seção, a tabela 2.1 mostra uma classificação comparativa dos mesmos segundo diversos critérios, evidenciando os pontos fortes e fracos de cada um.

Gupta e De Micheli [1,2] propõem um particionador automático que, partindo de uma implementação inicial inteiramente em *hardware*, vai gradualmente deslocando os nós da descrição do sistema para *software*, um de cada vez, enquanto as exigências de desempenho continuarem a ser satisfeitas, visando reduzir o custo da solução inicial. A arquitetura-alvo do sistema é fixa e pré-definida, consistindo de um microprocessador, um ou mais ASICs e uma memória compartilhada para a comunicação SW/HW. O método de particionamento permite que *hardware* e *software* executem em paralelo, explorando o paralelismo próprio do sistema, dado pela descrição de entrada. O algoritmo de particionamento *min-cut* adotado possui a limitação de poder ficar preso facilmente em um mínimo local, apesar de sempre propor uma solução se existir uma, mesmo que não seja a solução ótima, pois trata-se de um algoritmo "ganancioso" (*greedy*).

Ernst, Henkel e Benner, com sua ferramenta de particionamento automático COSYMA (*COSYnthesis for eMbedded Architectures*), propõem uma abordagem orientada a *software* e em granularidade fina para o particionamento [4]. A partir da solução inicial totalmente em *software*, as operações que violam as exigências temporais são identificadas por simulação e análise dos tempos de execução no COSYMA. Essas operações são então migradas para uma realização em *hardware*, usando um algoritmo de recozimento simulado para o particionamento, de modo a obter-se um ganho de velocidade e a conseqüente redução do tempo de execução total. Portanto, o tempo de execução é o único atributo do projeto que é otimizado. O modelo de execução do *software* e do *hardware* é baseado no princípio da exclusão mútua, o que significa que este método não permite a execução paralela de componentes de *hardware* e de *software*; só um dos dois tipos de componentes pode estar

executando, em um dado instante.

Barros, Rosenstiel e Xiong apresentam um algoritmo [5, 6] para particionar especificações de sistema, escritas em linguagem UNITY, em partes de *hardware* e *software*. A linguagem UNITY permite modelar computações paralelas e é independente da arquitetura-alvo. É construída uma implementação inicial de referência das declarações em UNITY (uma árvore de blocos), e a função de custo reflete os critérios para encontrar-se a melhor linha de corte dessa árvore, que define a alocação inicial dos blocos em *software* e *hardware*. O algoritmo de *clustering* é baseado essencialmente na melhoria do desempenho. O compartilhamento de recursos de HW entre elementos com um comportamento similar é utilizado, se isso minimizar o custo área/atraso. Diferentes possibilidades de implementação de componentes de *hardware* são consideradas durante o particionamento. A arquitetura-alvo adotada consiste de um processador de SW, responsável pelo controle central, e de processadores em *hardware* dedicados (ASICs, FPGAs).

Kalavade e Lee propõem um algoritmo [7] denominado GCLP (Criticalidade Global / Fase Local), cuja característica principal é a de escolher entre dois objetivos, de acordo com a criticalidade do tempo global e com as características particulares do nó sendo mapeado: se o tempo global é crítico, a função-objetivo tende a ser a redução do tempo de execução, caso contrário ela tende a ser a minimização do uso dos recursos de *hardware* ou de *software*. Ao invés de usar uma função-objetivo fixa, GCLP escolhe entre essas duas funções-objetivo a cada etapa do particionamento, de acordo com duas medidas: Criticalidade Global (GC), uma estimativa global da criticalidade do tempo em cada passo do algoritmo; e Fase Local (LP), uma medida das características peculiares de cada nó e de suas preferências. A arquitetura-alvo presumida consiste de um processador de *software*, *hardware* customizado e comunicação mapeada em memória compartilhada. A implementação de GCLP demonstra experimentalmente que a consideração das características locais de cada nó leva a uma redução significativa de área de *hardware*, com correspondente aumento da "área" de *software*.

Em [8], Kalavade e Lee estendem o algoritmo GCLP original com a seleção da alternativa de implementação, formando um novo método chamado de MIBS (Mapeamento e Escolha da Alternativa de Implementação). Este novo algoritmo leva em conta que um nó pode ter muitas alternativas de implementação diferentes para um mesmo mapeamento, as quais diferem em características de custo (área) e tempo. MIBS não apenas particiona os nós, mas também encontra a melhor alternativa de implementação para eles, tanto de *hardware* como de *software*, de modo a minimizar o uso dos recursos. [8] demonstra que a capacidade de fazer essa seleção, ao invés de apenas usar uma única implementação para todos os nós, reduz significativamente a área de *hardware* global: a solução MIBS é muito superior àquela obtida apenas com GCLP, em termos de economia de área de *hardware*.

Vahid et al. [39] propõem o acréscimo de um laço externo a um algoritmo de particionamento qualquer, baseado em busca binária da restrição (BCS, Busca Binária da Restrição), para tratar exclusivamente da minimização do *hardware*. O algoritmo de particionamento propriamente dito fica voltado apenas para tentar satisfazer as restrições de desempenho do sistema, e é liberado de tentar simultaneamente minimizar o tamanho do *hardware* empregado, pois as duas metas competem diretamente entre si. Este método resulta em menor área de *hardware*, com o sistema ainda satisfazendo as exigências de desempenho, reduzindo os custos de implementação.

Madsen et al. [9] propõem o algoritmo de particionamento automático PACE, implementado na ferramenta de *codesign* LYCOS (*LYngby CO-synthesis System*). A especificação de um sistema digital pode ser fornecida em C ou VHDL, sendo traduzida para um formato interno baseado em CDFG's, para a ferramenta de particionamento. O CDFG da aplicação é dividido em BSB's (Blocos Básicos de Agendamento), que podem ser movidos entre *hardware* e *software*. PACE calcula as áreas e tempos de execução de todas as sub-seqüências possíveis de BSB's e escolhe a melhor combinação de seqüências de blocos, sem BSB's em comum, a ser realizada em *hardware*, de modo a alcançar o maior ganho de velocidade possível do sistema com uma área total de *hardware* menor ou igual que a área disponível. As interações com BSB's vizinhos são consideradas por meio do ganho de velocidade extra decorrente do fato de dois BSB's serem capazes de se comunicar diretamente entre si quando ambos são colocados no mesmo domínio de implementação. No agendador do algoritmo PACE, dois BSB's não podem executar em paralelo, mesmo que um esteja em *hardware* e outro em *software*, limitando o ganho de desempenho do sistema.

LYCOS pode ser utilizada para fazer particionamento HW/SW em uma arquitetura-alvo consistindo de um microprocessador e um ASIC, comunicando-se por meio de E/S mapeada em memória.

Jeon e Choi [10] apresentam um algoritmo de particionamento HW/SW que explora o paralelismo heterogêneo próprio do sistema para minimizar o custo de *hardware* sem violar a restrição de desempenho. O algoritmo parte do princípio de que, quanto maior o paralelismo de um nó com outros nós no CDFG, maior o ganho de velocidade obtido com o mapeamento desse nó para *hardware*. Isso se deve à redução no tempo de espera do *software*, que pode executar outros nós paralelos enquanto o *hardware* está executando. Seguindo esse critério, o algoritmo se aproxima mais rápido de atender a restrição de desempenho, mapeando proporcionalmente a quantidade mínima possível de nós para *hardware*, minimizando assim o custo da solução.

Esse algoritmo emprega o conceito de "força" como uma medida de paralelismo, significando o número de nós de *software* que podem rodar em paralelo com um nó candidato

a *hardware*. A cada iteração de particionamento, o nó com a máxima força é mapeado para *hardware*, e os custos de comunicação são atualizados. Dois nós podem compartilhar o mesmo recurso de *hardware*.

Jeon e Choi propõem um novo algoritmo de particionamento em [11] e, ao mesmo tempo, o emprego da técnica de *loop pipelining* para aumentar o paralelismo dentro dos laços da descrição da aplicação, visando uma implementação mais eficiente da mesma. *Loop pipelining* é uma técnica de otimização de laços para computação paralela, que consiste na sobreposição no tempo da execução de blocos de instruções em diferentes passos de iteração, para aumentar o paralelismo dentro de um laço.

A etapa de *loop pipelining* precede o particionamento HW-SW; este visa minimizar o custo de *hardware*, ao mesmo tempo mantendo o desempenho acima do exigido. O algoritmo de particionamento toma as decisões de mapeamento considerando o compartilhamento de recursos de *hardware* entre dois ou mais nós, e a existência de várias alternativas de implementação de um nó de *hardware*, para selecionar aquela que reduza ao máximo o custo total de *hardware*. A arquitetura-alvo consiste de um único processador de *software* de uso geral e múltiplos ASIC's. Considera-se um modelo de comunicação mapeada em memória, onde não há *hardware* dedicado para comunicação, i.e., o *software* é bloqueado até que a comunicação se encerre. Os autores demonstram que o uso de *loop pipelining* é eficaz em melhorar tanto o desempenho quanto o custo do sistema final, e que o compartilhamento de recursos de *hardware* e a escolha da forma de implementação são eficazes em reduzir o custo total de *hardware* do sistema particionado.

Rousseau et al. [3, 12] apresentam um algoritmo de particionamento de domínio específico, voltado somente para sistemas de fluxo de dados, como por exemplo, sistemas de telecomunicações e de processamento de sinais. É permitida a execução simultânea de partes de *hardware* e de *software*. Para cada nó é determinado um quadro de tempo, por meio de um agendamento ASAP (*As Soon As Possible*) e de um agendamento ALAP (*As Late As Possible*). Um nó pode ser agendado para qualquer passo de tempo entre os instantes ASAP e ALAP. Para cada nó, um passo de tempo é escolhido para o qual a função de custo seja mínima e se esta atribuição envolver a restrição mais fraca sobre os outros nós, pois agendar um nó diminui os quadros de tempo efetivos dos outros nós. As influências dos outros nós sobre o agendamento do nó corrente, em *software* ou em *hardware*, são representadas por "forças" de repulsão. O cálculo de todas essas forças para um nó n_i expressa a soma das restrições induzidas por este nó sobre todos os outros nós. No final de cada iteração, o algoritmo mapeia e agenda o nó com a menor força, porque este é o nó que induz as restrições mais fracas.

A arquitetura-alvo consiste de um único processador, onde a partição de *software* é executada, e um único ASIC, que implementa a partição de *hardware*. São ignorados todos

os tempos e custos de comunicação entre as partes de *hardware* e de *software*, dentro da suposição de que o tempo de execução do menor nó é maior que o tempo estimado das comunicações. Outro ponto fraco deste algoritmo é não trabalhar com compartilhamento de blocos funcionais comuns entre nós diferentes.

Prakash e Parker [40] estabelecem pela primeira vez um modelo formal e genérico para auxiliar a síntese automática de arquiteturas MultiProcessadas Heterogêneas (HMP), e também um método para mapear e agendar os nós de uma aplicação nos processadores da nova arquitetura criada, ao invés de apenas mapear nós sobre uma arquitetura fixa. Os autores utilizam programação linear para resolver o modelo, com o objetivo de atender às restrições de custo e desempenho da aplicação. A arquitetura HMP é genérica, isto é, ela pode utilizar vários tipos diferentes de processadores (PE's), em termos de funcionalidades, custo e desempenho.

O modelo matemático apresentado em [40] para o problema de síntese, define formalmente diversos parâmetros que caracterizam tanto o grafo da aplicação quanto a implementação do mesmo em uma arquitetura HMP qualquer. Este método recebe, como entradas, o grafo da aplicação a ser particionada e agendada na arquitetura HMP sintetizada; e uma tabela de nós versus PE's, contendo as características e os custos dos tipos de PE's disponíveis para sintetizar a arquitetura, e os tempos de execução de cada nó do grafo de entrada em cada um dos PE's. O método de [40] fornece como resultado: a arquitetura HMP sintetizada para o sistema, especificada em termos dos PE's selecionados para fazerem parte dela e de suas interconexões (topologia), o mapeamento e o agendamento da execução dos nós nos PE's, e também as comunicações de dados entre os nós através dos enlaces de comunicação dos PE's.

Em um trabalho posterior [41], Prakash e Parker estudam o impacto da comunicação entre os nós do grafo da aplicação, na síntese dos sistemas. Eles concluem que, à medida em que aumenta o volume de dados comunicado entre os nós, as arquiteturas mais distribuídas (com maior número de PE's e portanto com mais comunicações entre eles) vão apresentando um desempenho cada vez pior.

H. Oh e S. Ha [18] apresentam um algoritmo para o agendamento de CDFG's em arquiteturas HMP, chamado de BIL (Melhor Valor Imaginário), o qual incorpora os efeitos da heterogeneidade dos processadores disponíveis e do tempo devido à comunicação interprocessadores (IPC). O algoritmo de agendamento BIL é baseado na técnica de agendamento de lista, em que cada nó recebe uma prioridade, que é o seu índice BIL. O objetivo do agendamento é minimizar a agenda total (ocupação do tempo), do grafo de entrada.

O valor BIL do nó n_i agendado no processador P_j, BIL(n_i, P_j), é calculado com base na suposição de que todos os nós sucessores de n_i podem ser agendados em seus tempos ótimos, e indica o menor comprimento do caminho crítico (tempo de conclusão

mais curto a partir do nó n_i até o fim do grafo), incluindo o tempo extra de IPC. Uma vez que ele considera o custo da comunicação em seu cálculo, o BIL de um nó pode ser considerado como a informação global de todos os nós sucessores, dentro da suposição otimista. Isso significa que o agendamento de um nó n_i em um PE considera o efeito até sobre seus sucessores mais distantes, que entra no cálculo do BIL de n_i. Dessa forma, o efeito global da decisão de agendamento é medido adequadamente. A técnica BIL produz o agendamento ótimo se o CDFG é linear [18]. No final do algoritmo, é determinado o PE ótimo para o nó.

Hou e Wolf propõem um algoritmo de particionamento sobre arquitetura HMP [13], baseando-se nos trabalhos de Prakash e Parker sobre alocação de processos em arquiteturas heterogêneas distribuídas. É dado um algoritmo para particionar múltiplos grafos de tarefas do sistema em arquiteturas distribuídas compostas por diferentes tipos de PE's, informados pelo usuário com seus custos e o tempo de execução de cada nó dos grafos em cada PE. O algoritmo permite paralelismo de execução entre PE's, minimiza os custos de comunicação, e explora a vantagem específica de cada PE. Vários processos podem compartilhar o mesmo PE.

Em um trabalho posterior, H. Oh e S. Ha desenvolvem [14] um algoritmo de particionamento para uma arquitetura-alvo HMP, também baseado nos trabalhos de Prakash e Parker [40] e similar ao de [13], que cobre desde a escolha da arquitetura e da forma de implementação de nós de *hardware*, e o problema do compartilhamento de recursos em projetos de SoC's, até problemas de escolha de PE's em projetos de sistemas heterogêneos distribuídos. O agendamento é feito pelo algoritmo BIL. As entradas para o algoritmo são: o grafo da aplicação a ser particionada e agendada na arquitetura HMP sintetizada; e uma tabela de perfis nós x PE's, contendo as características e os custos dos tipos de PE's disponíveis para sintetizar a arquitetura, e os tempos de execução e custos de cada nó do grafo de entrada em cada um dos PE's, da mesma forma que em [13]. O número de PE's pode ser muito grande, porque cada alternativa de implementação em *hardware* é considerada como um PE separado, tendo seu próprio custo. O resultado fornecido é a arquitetura HMP escolhida para o sistema (especificada em termos dos PE's selecionados para ela), o mapeamento e o agendamento da execução dos nós da aplicação nos PE's, e também as comunicações de dados entre os nós nos enlaces entre os PE's. Comparado com uma solução por programação linear inteira, este algoritmo é muito mais rápido em execução, com um acréscimo de custo do sistema de apenas 5%.

Esse algoritmo resolve o dilema de ter que optar entre uma CPU cara e rápida, tal que o sistema demande pouco ou nenhum *hardware* adicional, e optar por uma CPU barata e lenta, que leve o sistema a demandar uma certa área de *hardware* dedicado. Na Seção 2.4, são explicadas as razões pelas quais este algoritmo é escolhido como base para a

implementação do algoritmo de particionamento otimizado, proposto nesta tese.

Kalavade e Subrahmanyam propõem pela primeira vez um algoritmo de particionamento para sistemas multi-modo [15], aqueles que suportam múltiplas aplicações (modos) das quais somente uma pode ser executada em qualquer instante, dependendo de fatores externos. Os múltiplos modos são uma técnica cada vez mais utilizada em sistemas embarcados para suportar a evolução dinâmica de ambientes, padrões, algoritmos e novos serviços. Quando se projeta um sistema multi-modo é importante considerar todas as aplicações simultaneamente, ao invés de projetar para aplicações individuais.

No algoritmo proposto em [15], trabalha-se com um conjunto de k aplicações AP = {A1, A2, ..., Ak}, com o objetivo de sintetizar a arquitetura heterogênea multiprocessada do sistema que as suporte todas, e que também seja otimizada para elas. Qualquer aplicação pode estar ativa em tempo de execução e suas restrições de temporização devem ser atendidas. O objetivo de projeto é minimizar a área de *hardware* global, o que se reflete no custo global do sistema.

O conjunto de aplicações especificadas é inicialmente analisado para extrair-se as medidas dos nós em comum entre elas. O algoritmo de particionamento é então executado para cada aplicação, e usará essas medidas para orientar os mapeamentos dos nós que são comuns entre as aplicações do conjunto AP. O algoritmo utilizado é o GCLP [8], originalmente criado para particionar uma única aplicação, que é modificado de duas formas diferentes para poder ser aplicado ao particionamento multi-aplicações:

- Se um mesmo tipo de nó é repetido em várias aplicações, seu mapeamento é orientado para *hardware*.

- Quando se mapeia nós que se repetem em várias aplicações, todas as suas instâncias devem ser levadas para *hardware* ou *software*. O mapeamento de nós comuns é orientado para que mantenha a consistência ao longo de todas as aplicações, mas também considerando as exigências específicas da aplicação em questão.

O segundo método teve resultados muito superiores ao primeiro, para o particionamento de um sistema multi-modo de processamento de sinais.

Oh e Ha propõem [16] uma generalização de seu algoritmo de particionamento anterior [14] para o caso de sistemas multi-modo. No seu modelamento, um sistema multi-modo Π é definido como um conjunto fixo de aplicações ou modos $\{\Pi_i\}$, onde $\Pi = \{\Pi_1, \Pi_2, \ldots, \Pi_N\}$. Cada modo Π_i é um nó que possui um grafo interno cujos nós são as tarefas $\{\tau_j\}$. A Seção 2.4.2 fornece mais detalhes sobre a estrutura interna de um modo. Considera-se que o sistema possa executar um único modo de cada vez, e cada modo, enquanto está ativo, representa o sistema global. Diferentes modos de um sistema são distinguidos pelas

tarefas que os compõem, mas podem ter tarefas em comum. Considera-se a possibilidade de compartilhamento de tarefas entre modos de operação e de compartilhamento de recursos de *hardware* entre tarefas. Cada tarefa é especificada como um CDFG acíclico, cujos nós também podem ser compartilhados entre tarefas diferentes. São permitidos múltiplos grafos de tarefas de entrada acíclicos, com diferentes períodos de iteração e sem dependências condicionais; os grafos podem ter a granularidade desejada pelo usuário, e mesmo uma granularidade mista. O particionamento de [16] procura encontrar uma arquitetura de sistema de custo mínimo que satisfaça as restrições de agendabilidade de cada tarefa.

As entradas para esse método são: os grafos das tarefas do sistema; uma biblioteca de PE's candidatos, que deve incluir microprocessadores, implementações em ASIC's de nós das aplicações e FPGA's; e uma tabela de perfis nó-PE, que mostra, para todos os nós de todas as tarefas, o custo e o tempo de execução em cada PE candidato disponível, seja em *hardware* ou em *software*. Portanto, para cada PE, são dados seu custo e o tempo de execução de pior caso de cada nó nesse PE. Cada implementação em *hardware* é considerada como um PE separado, tendo seu próprio custo.

O algoritmo escolhe o PE, dentre os PE's candidatos de entrada, para uma tentativa inicial de solução. Se o objetivo de projeto for a minimização do custo, então escolhe-se o processador mais barato primeiro. O grafo de cada tarefa é agendado nesse PE, buscando minimizar a duração da agenda. O agendador BIL é utilizado [18]. O passo seguinte é a avaliação de desempenho, que verifica se as restrições de agendabilidade são atendidas. Caso positivo, encerram-se as iterações e exibe-se o resultado do particionamento. Se qualquer restrição de projeto for violada, os resultados do agendamento e a informação de violação são passados para o controlador de alocação, para este escolher outros PE's de custo mais alto, que tenham mais condições de satisfazer a agendabilidade.

O algoritmo de particionamento proposto é aplicado a um sistema multi-modo. Em um primeiro momento, ele é aplicado a cada modo de operação separadamente, e os custos dos sistemas são somados no final. Em seguida, ele é aplicado a todos os modos juntos, resultando em um custo 10% menor do sistema [16].

Na tabela 2.1 (página 52) é mostrada uma classificação dos diversos algoritmos de particionamento pesquisados, listando as características de qualidade mais relevantes identificadas em cada algoritmo.

2.3.2 Algoritmos de particionamento com *hardware* reconfigurável

A introdução de circuitos integrados de *hardware* reconfigurável em sistemas embarcados acarretou a flexibilidade do *hardware*, em um nível próximo ao da flexibilidade do *software* de um microprocessador, permitindo a adaptação da plataforma embarcada para as exigências da partição de *hardware* da aplicação. Posteriormente, surgiram os componentes de *hardware* **dinamicamente** reconfigurável, isto é, que podem ser reconfigurados em tempo de execução (RTR), com ou sem a intervenção humana, e dentro dos sistemas nos quais estão inseridos, sem a necessidade de interromper a operação desses sistemas. RTR é a capacidade de modificar o *hardware* enquanto a aplicação está executando; as configurações do *hardware* costumam ficar armazenadas em memória e ser trocadas por um *software* do sistema. Os sistemas embarcados reconfiguráveis dinamicamente oferecem adaptabilidade a requisitos variáveis de sistema, a um custo baixo e com melhor desempenho. A exploração da reconfiguração dinâmica do *hardware*, nos sistemas que a possuem, equivale ao compartilhamento de recursos de *hardware* entre nós, nos sistemas de *hardware* fixo.

Tradicionalmente, os circuitos com RTR são sempre configurados por completo e de uma só vez, a cada execução de uma reconfiguração. Entretanto, alguns permitem ainda uma **reconfiguração parcial**, isto é, somente das suas regiões internas que precisam ser modificadas, o que se constitui em um método para reduzir o "custo" adicional representado por uma reconfiguração. A reconfiguração parcial permite que somente uma porção da matriz reconfigurável seja modificada, ou seja, ela permite a alteração seletiva de segmentos do *hardware*. Isso significa que, quando dois estágios consecutivos de uma mesma aplicação têm estruturas similares, parte do *hardware* já carregado na matriz pode ser reutilizado no estágio seguinte. Cada bloco lógico de um estágio anterior que não precisa ser reconfigurado para o próximo estágio representa menos informação de configuração que tem que ser enviada da memória para o dispositivo reconfigurável.

A reconfigurabilidade parcial reduz consideravelmente o tempo e o consumo de energia da reconfiguração do dispositivo, para estágios computacionalmente similares, porque as atualizações de *hardware* são altamente localizadas. A reconfiguração parcial é uma capacidade própria de apenas alguns modelos de FPGA's e, para aqueles que a possuem, o tempo de reconfiguração para um par de nós depende do número de LE's usados por cada um e da similaridade estrutural entre os nós. No caso de um FPGA que não suporta reconfiguração parcial, cada nova reconfiguração requer que o FPGA inteiro seja programado, logo o tempo de reconfiguração é fixo, e não é afetado pelo tamanho do nó.

A reconfigurabilidade dinâmica do *hardware* requer a adaptação dos algoritmos de par-

ticionamento *HW/SW* para explorá-la efetivamente. O *HW/SW Codesign* agora tem que determinar quais porções da especificação do sistema devem ser mapeadas para a lógica reconfigurável, para uma possível lógica fixa e para o processador de *software*. Quando o *hardware* do sistema híbrido é reconfigurável estaticamente (com interrupção da operação do sistema), o algoritmo de particionamento não difere daqueles para sistemas de *hardware* fixo. Nos algoritmos de particionamento para sistemas heterogêneos em que o *hardware* é reconfigurável dinamicamente, a principal modificação observada em relação aos algoritmos para sistemas de *hardware* fixo é a consideração da latência de reconfiguração do *hardware* no cálculo do desempenho global da aplicação particionada [21]. Uma nova meta de otimização do particionamento passa a ser a de minimizar ou mascarar essa latência: O particionador toma a decisão de mapear ou não um nó para *hardware* baseado não apenas na estimativa do tempo de execução dele no FPGA, mas também na estimativa de sua latência de reconfiguração, que pode afetar o tempo de conclusão das tarefas do sistema. Esse é o principal impacto da introdução da reconfigurabilidade do *hardware* sobre os algoritmos de particionamento.

Várias técnicas já foram propostas na literatura para minimizar o efeito da latência da reconfiguração sobre o agendamento [21]. A técnica mais comum consiste em tentar executar a reconfiguração dinâmica do *hardware* em paralelo (sobreposta no tempo) com a execução de outro nó da aplicação, principalmente com o processamento em *software* [42–46]. Na prática, isso só é vantajoso se o tempo de reconfiguração for da mesma ordem de grandeza do tempo de execução médio dos nós em *software*, o que ocorre quando a reconfiguração parcial é utilizada.

Comparando algoritmos de particionamento para sistemas de *hardware* reconfigurável e para sistemas de *hardware* fixo, observa-se que a exploração, por um algoritmo, da capacidade de reconfiguração dinâmica do *hardware*, nos sistemas que a possuem, equivale ao compartilhamento de recursos de *hardware* entre nós, para os sistemas de *hardware* fixo.

As soluções de particionamento propostas nos diversos trabalhos pesquisados são, em alguns casos, muito dependentes das arquiteturas de sistema empregadas, porque a diversidade de arquiteturas reconfiguráveis é muito maior do que no caso de sistemas heterogêneos com *hardware* fixo. As soluções dependentes de arquitetura são tratadas na subseção seguinte.

Soluções para arquiteturas específicas

Vincentelli et al. [47] propõem um método para uma arquitetura bem específica, mas muito poderosa, de plataforma CSoC, com um processador de *software* cujo conjunto de instruções é reconfigurável. Um novo algoritmo de particionamento automático é pro-

posto, que rotula os trechos críticos do código e mapeia-os sobre a matriz reconfigurável, para se tornarem "instruções em FPGA", podendo ser utilizadas na programação em linguagem C. [47] conclui que o uso do FPGA melhora significativamente o tempo de execução das aplicações, em comparação a usar somente um processador. Este algoritmo é voltado a aplicações de controle, entretanto os resultados experimentais mostram que o máximo ganho de desempenho é obtido com a implementação dos nós de dados no FPGA, e não dos nós de controle. Nesta arquitetura, como o arranjo reconfigurável faz parte da via de dados do processador, o tempo para a comunicação HW-SW é mínimo em comparação com as outras arquiteturas, sendo esta a principal vantagem desse tipo de sistema heterogêneo.

Noguera e Badia [42–45] usam uma topologia "clássica", porém com um *chip* dinamicamente reconfigurável de arquitetura personalizada, contendo dois agendadores, o arranjo de lógica dinamicamente reconfigurável (DRL) e memórias dedicadas. A aplicação das técnicas propostas pelos dois autores é vinculada à implementação dessa arquitetura reconfigurável concebida por eles. O algoritmo ataca o problema da latência de reconfiguração utilizando os dois agendadores implementados em *hardware*, dentro do *chip*. Os dois rodam em paralelo durante a execução da aplicação, sendo que um agendador serve para decidir dinamicamente a ordem de execução dos eventos, visando atender as restrições de desempenho da aplicação, e o outro agendador decide que célula do arranjo DRL deve ser reconfigurada e com que contexto de reconfiguração. O objetivo global dos dois agendadores é minimizar a latência de reconfiguração pela execução paralela dos eventos e das reconfigurações dinâmicas do *hardware*. O algoritmo é responsável por definir os contextos de reconfiguração. A solução de Noguera e Badia é voltada para satisfazer as necessidades de sistemas embarcados complexos onde a seqüência de execução de nós de *hardware* e *software* pode mudar dinamicamente.

Em [43], Noguera e Badia provam que usar dispositivos de reconfiguração mais rápida não garante os melhores resultados, que dependem fortemente do algoritmo de particionamento; o algoritmo de [43], utilizado com células de reconfiguração lenta, fornece quase o mesmo resultado que um algoritmo inferior com células de reconfiguração duas vezes mais rápida (e mais caras). Esse resultado mostra a importância de procurar-se otimizar os algoritmos de particionamento, agregando fatores de qualidade de trabalhos já publicados, para o reuso de conhecimentos e experiências. É demonstrado também que, quando se aumenta o número de células de DRL no sistema, o tempo de execução deste converge para o tempo obtido usando-se *hardware* estático (sem latência de reconfiguração) [43], o que é compreensível, pois com muitas células de *hardware*, cada nó vai poder ficar em sua própria célula o tempo todo, e portanto a reconfiguração dinâmica torna-se desnecessária.

Dentre os algoritmos de particionamento publicados para sistemas embarcados de

arquitetura distribuída, os algoritmos das ferramentas CORDS [48] e CRUSADE [49] mostraram-se os mais genéricos e completos. CORDS possui as seguintes características:

1. Aloca recursos automaticamente dentre um conjunto de múltiplos PE's genéricos (como FPGA's ou processadores de *software* de uso geral) e recursos de comunicação;
2. Sintetiza sua própria arquitetura distribuída, utilizando múltiplos FPGA's se necessário e determinando a conectividade dos recursos;
3. Agenda dinamicamente os nós e os eventos de comunicação;
4. Calcula o custo de cada arquitetura;
5. Permite a escolha da alternativa de implementação , para nós já mapeados em *hardware* ou *software*. A maioria dos outros algoritmos para arquiteturas distribuídas assume que a implementação em *hardware* será feita em um FPGA pré-escolhido;
6. Otimiza a seqüência de nós nos FPGA's, considerando suas prioridades dinâmicas individuais no agendamento; com isso, o atraso total de reconfiguração é ajustado dinamicamente durante o agendamento, buscando-se sua redução sem causar perda de deadlines;
7. Explora a reconfiguração parcial dos FPGA's quando necessário.

A característica descrita no item 5 acima, explorada em algoritmos para sistemas com *hardware* fixo, poderia ser integrada aos algoritmos de particionamento para sistemas de *hardware* reconfigurável, não só para escolher uma alternativa de implementação de nós de *software*, mas também para a possibilidade de escolher entre FPGA's de diferentes modelos/fabricantes, conforme as características de cada um, ou mesmo para escolher entre uma implementação em *hardware* fixo ou em *hardware* reconfigurável.

A ferramenta CRUSADE [49] faz uma generalização ainda maior que CORDS, pois utiliza uma biblioteca de PE's genéricos com microprocessadores, ASIC's e FPGA's/CPLD's, de vários fabricantes. Com isso, CRUSADE realmente é um algoritmo de particionamento, agendamento e co-síntese de sistemas DHE aplicável a sistemas tanto de *hardware* fixo como de *hardware* reconfigurável, embora não trabalhe com reconfiguração parcial do *hardware*.

Algoritmos de particionamento independentes da arquitetura-alvo

Nos algoritmos de particionamento em que a arquitetura-alvo heterogênea é a "clássica", utilizando FPGA's comerciais, as conclusões obtidas são mais facilmente generalizáveis. A principal meta de otimização dos algoritmos de particionamento para sistemas

heterogêneos dinamicamente reconfiguráveis é a de minimizar ou mascarar a latência de reconfiguração de cada nó mapeado em *hardware*, visando minimizar o tempo de execução dos grafos e assim otimizar o desempenho da aplicação. As técnicas propostas para essa finalidade são listadas a seguir:

1. Executar a reconfiguração dinâmica das células de *hardware* em paralelo com outra tarefa do sistema, que pode ser principalmente o processamento de eventos da aplicação, em *hardware* ou em *software*, o que é posto em prática por Noguera e Badia [42–45] e Choi et al. [46]. Os algoritmos propostos por esses pesquisadores fazem a pré-busca da configuração, que consiste em carregar-se antecipadamente o próximo contexto de reconfiguração para o FPGA, sobrepondo assim no tempo, se possível, o processamento no processador de *software* ou em uma célula do *hardware* reconfigurável, com a reconfiguração de outra célula. Em [50], a reconfiguração do *hardware* ocorre durante os enlaces (comunicações) entre nós do grafo.

2. Posicionar, por meio do agendamento no FPGA, tarefas do mesmo tipo ou de tipos similares adjacentes umas às outras no tempo, visando reduzir a quantidade de reconfigurações necessárias para um FPGA, minimizando assim o tempo total de reconfiguração na operação do sistema [48]. Com esse procedimento, a reconfiguração do FPGA somente ocorre quando da sua mudança de uma tarefa para outra de um tipo diferente. O FPGA não precisa ser reconfigurado para executar outra tarefa do mesmo tipo daquela para a qual ele já se encontra configurado.

3. Compartilhar variáveis entre reconfigurações consecutivas do FPGA [51].

4. Utilizar FPGA's dotados de reconfiguração parcial, explorando os pontos em comum entre projetos candidatos ao FPGA para reduzir a quantidade de dados de reconfiguração necessários para uma dada tarefa, e portanto reduzir o tempo necessário para cada reconfiguração [46, 51]. É analisado com qual das tarefas já existentes no FPGA a tarefa seguinte compartilha mais dados de configuração, e são reconfigurados somente os bits dessa tarefa que diferem dos bits de configuração da anterior. O tempo para reconfiguração total de um nó no FPGA é multiplicado pela porcentagem de reconfiguração incremental a ser executada para aquele nó [46]. Essa técnica de reconfiguração incremental de tarefas, que explora a reconfiguração parcial, deve ser utilizada somente quando necessário, devido ao maior custo desses FPGA's. Com esse tipo de FPGA, a técnica proposta no item 2 também é eficaz para reduzir o tempo necessário para cada reconfiguração, além da quantidade de reconfigurações. Os resultados experimentais obtidos com a ferramenta CORDS [48]

mostram que em alguns casos a reconfiguração mais rápida dos FPGA's com reconfiguração parcial permite a satisfação de especificações de sistema que não poderiam ser atendidas com FPGA's sem o recurso. Esse fato é especialmente verdadeiro para sistemas nos quais o tempo de computação é reduzido e torna-se similar ao tempo de reconfiguração.

5. Utilizar a técnica de reconfiguração parcial antecipada (EPR) apresentada por Choi et al. [46], que faz a configuração de um nó, em um FPGA, tão cedo quanto possível e antes do seu processamento ser necessário. Com a EPR, enquanto os nós de *software* estão sendo executados pela CPU, o FPGA já vai sendo reconfigurado para os seus próximos nós, o que mascara o tempo gasto nessa reconfiguração. A execução de um nó não precisa começar imediatamente após a sua configuração no FPGA. O FPGA também pode estar executando um nó e ao mesmo tempo estar sendo reconfigurado para o nó seguinte, contanto que esses dois eventos ocorram em setores físicos diferentes do dispositivo. Essa técnica nada mais é do que a mesma proposta no item 1, agora com reconfiguração parcial.

6. Modificar o estimador de tempo de execução em *hardware* de um nó, no algoritmo de particionamento, somando-se, ao seu tempo de execução, o tempo extra de reconfiguração [43].

7. Facilitar a identificação de grupos de nós cujas janelas de execução não se sobrepõem no tempo, isto é, que nunca executam em paralelo, e portanto podem ser atribuídos seqüencialmente ao mesmo conjunto de recursos de FPGA, compartilhando-os e formando os contextos de reconfiguração; caso contrário, eles devem ser atribuídos a conjuntos independentes de recursos de FPGA [49].

Alguns pesquisadores demonstram a redução de custos dos sistemas sintetizados com o emprego de *hardware* reconfigurável dinamicamente, em relação às arquiteturas sintetizadas sem esse recurso [49]. Essas reduções de custo significam um menor número de PE's e de enlaces na arquitetura do sistema. Outros fortes argumentos a favor do uso de *hardware* com RTR são dados pelos resultados em [46, 52], que demonstram a possibilidade de aumento da velocidade de execução da aplicação para a mesma restrição de área, quando a reconfiguração dinâmica é empregada.

Comparação entre os algoritmos de particionamento analisados

A tabela 2.2 (página 53) apresenta a classificação e comparação dos algoritmos de particionamento HW/SW para sistemas com *hardware* dinamicamente reconfigurável que

foram pesquisados. Os critérios de qualidade são os mesmos utilizados para os algoritmos para sistemas de *hardware* fixo, juntamente com outros específicos para sistemas de *hardware* reconfigurável: a capacidade ou não de RTR e o suporte ou não à reconfiguração parcial do *hardware*. Na tabela 2.2, uma arquitetura-alvo chamada de "coprocessador" é aquela em que o *hardware* do sistema é uma placa coprocessadora baseada em FPGA's, conectada a um microcomputador que roda os nós de *software*. A arquitetura chamada "clássica" consiste de um processador de *software* e um FPGA compartilhando uma memória.

2.4 Algoritmo-base de particionamento

O levantamento do estado da arte feito para esta tese apontou o algoritmo de particionamento de H. Oh e S. Ha [16], implementado na ferramenta de *HW/SW Codesign* PEaCE [17], como aquele que melhor atende aos critérios de qualidade da Seção 1.6. Além disso, um estudo das ferramentas de *Codesign* acadêmicas disponíveis na internet também levou à escolha de PEaCE para a realização desta tese. Os motivos para esta escolha foram: o fato de ser PEaCE a única ferramenta cujo código-fonte é totalmente aberto e acessível ao usuário; o fato de este código ser orientado a objeto, o que facilita sua manutenção e evolução por um programador; e a facilidade de contactar-se os seus desenvolvedores. Por todas essas razões, tanto de ordem técnica quanto prática, selecionou-se o algoritmo de particionamento acadêmico de [16] para servir como referência para a implementação das otimizações propostas neste trabalho.

A Seção 2.4.1 descreve os arquivos de entrada necessários para a execução de um particionamento por este algoritmo, e a Seção 2.4.2 descreve a estrutura de dados utilizada pelo mesmo. Em seguida, a Seção 2.4.3 explica detalhadamente o funcionamento do algoritmo de H. Oh e S. Ha, com destaque para o algoritmo de agendamento BIL [18]. São fornecidos exemplos de sua execução para sistemas simples. Finalmente, a subseção 2.4.6 descreve a forma de trabalho com o código desse algoritmo e a implementação de uma função de custo simples.

2.4.1 Arquivos de entrada para o algoritmo de H. Oh e S. Ha

O algoritmo de particionamento de H. Oh e S. Ha [16] propõe automaticamente uma implementação heterogênea (i.e., *hardware/software*) para uma aplicação, a partir de dados de entrada fornecidos. Esses dados de entrada são de quatro tipos, organizados em três arquivos, e estão mostrados na figura 2.1. Em primeiro lugar, o algoritmo lê um arquivo com o grafo do sistema, contendo um ou mais grafos de aplicações, cada um

dos quais contém todas as funções (nós) de uma aplicação e as trocas de dados entre elas (modelo de fluxo de dados). Em seguida, o algoritmo de particionamento precisa conhecer os valores de desempenho e de custo para cada nó das aplicações em cada um dos PE's candidatos, valores esses que são lidos de um segundo arquivo de entrada. Enfim, o algoritmo lê um terceiro arquivo contendo as restrições impostas à aplicação como um todo, que ela precisará respeitar independente do resultado do particionamento.

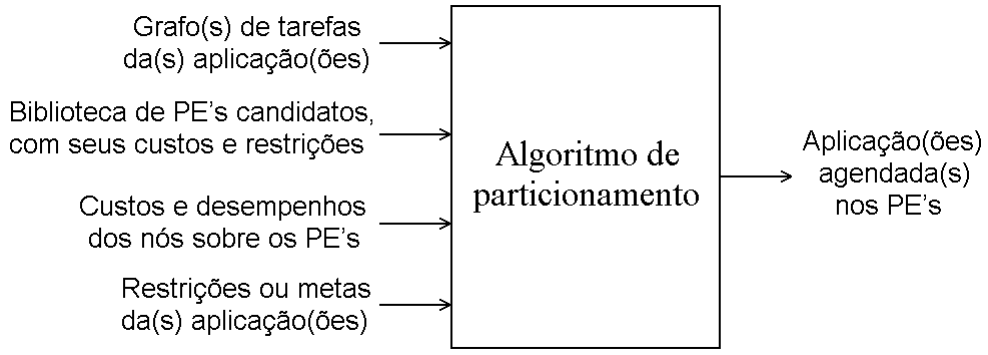


Figura 2.1: Entradas e saída do algoritmo de particionamento proposto.

Os três arquivos de entrada mencionados especificam portanto uma ou mais aplicações e um conjunto de PE's candidatos a implementá-las. Esses dados capacitam o algoritmo a propor uma solução, que consiste em uma implementação da(s) aplicação(ões) como um sistema heterogêneo. Os arquivos de entrada para o algoritmo de particionamento precisam ser escritos manualmente como arquivos de texto puro (ASCII) utilizando a sintaxe da linguagem XML, podendo-se empregar para isso um editor de textos qualquer[1]. Os conteúdos para esses três arquivos de entrada são detalhados a seguir, para um sistema chamado "`Exemplo`":

1. O arquivo "`Exemplo.xml`" contém os grafos hierárquicos de cada uma das aplicações do sistema a ser particionado. Uma aplicação é um grafo cujos nós são tarefas de alto nível $\{\tau_j\}$, e cada uma das tarefas é um grafo contendo os nós funcionais n_i, que são os elementos particionáveis. Os valores associados aos arcos dos grafos das aplicações também são fornecidos.

2. O arquivo "`Exemplo_TimeCost.xml`" consiste de uma tabela de perfis "Nó-PE", especificando todos os PE's candidatos, de *software* e de *hardware*, com seus custos monetários por unidade, as quantidades disponíveis de cada tipo de PE e os tempos de escrita de uma palavra no barramento, para cada PE. Abaixo do cabeçalho de

[1] utilizou-se Microsoft Word 2000^{TM}, pelas suas facilidades de edição.

definição de cada PE candidato, são listados todos os nós atômicos, de todas as tarefas e modos, com os valores de implementação de cada nó (tempo de execução, área e potência) naquele PE. A granularidade dos nós é livremente definida pelo usuário, podendo ser variável. Portanto, este arquivo contém a relação de PE's considerados para compor a arquitetura, com todos os valores associados ao desempenho e aos custos para os nós do sistema.

3. O arquivo "`Exemplo_mode.xml`" consiste de uma tabela especificando as restrições impostas para cada tarefa em cada modo do sistema. Esses valores independem da arquitetura ou dos PE's que forem escolhidos para compor a solução do particionamento. O arquivo lista os nomes dos modos, contendo os nomes das tarefas que os compõem e, para cada uma, seu período e seu *deadline*.

A sintaxe XML para os três arquivos de entrada é explicada no manual do PEaCE [54]. O programa particionador de PEaCE lê esses três arquivos de entrada e produz como resultado um arquivo de agendamento "`Exemplo_sched.xml`", onde os nós dos grafos das aplicações estão agendados nos PE's selecionados para a arquitetura-alvo, e o instante de execução de cada nó é fornecido.

2.4.2 Estrutura de dados para a descrição de aplicações

Um nó-aplicação é denominado de "modo" pelo algoritmo de particionamento. Um nó é chamado de "tarefa" apenas no nível mais alto da aplicação: As tarefas são os macro nós que formam a aplicação, e contêm apenas um grafo de mesmo nome, que por sua vez pode conter vários nós internos, formando assim uma hierarquia. Modos e tarefas são nós hierárquicos (que contém grafos com outros nós). Os nós que constituem as tarefas podem ou não ser hierárquicos. Nós que não contém grafos internos são chamados nós atômicos. A figura 2.2 exemplifica um modo de um sistema, com a sua hierarquia de nós-tarefas e nós funcionais.

Os nós hierárquicos são abstratos e não possuem existência real para o algoritmo de particionamento: eles só existem para agrupar nós logicamente relacionados em grafos de aplicações, tarefas e sub-tarefas internas, com o intuito de facilitar a descrição, e só aparecem nos nomes completos dos nós atômicos, construindo a hierarquia. Os nós hierárquicos não possuem declarações de portas de entrada e saída explícitas, e nem as demais informações de um nó atômico. Os nós atômicos constituem-se nas verdadeiras unidades de particionamento para o algoritmo, aquelas que são consideradas individualmente para implementação em *software* ou *hardware*. São apenas os nós atômicos que devem ser declarados de forma completa, atribuindo-se-lhes os seus nomes de instância, de tipo do

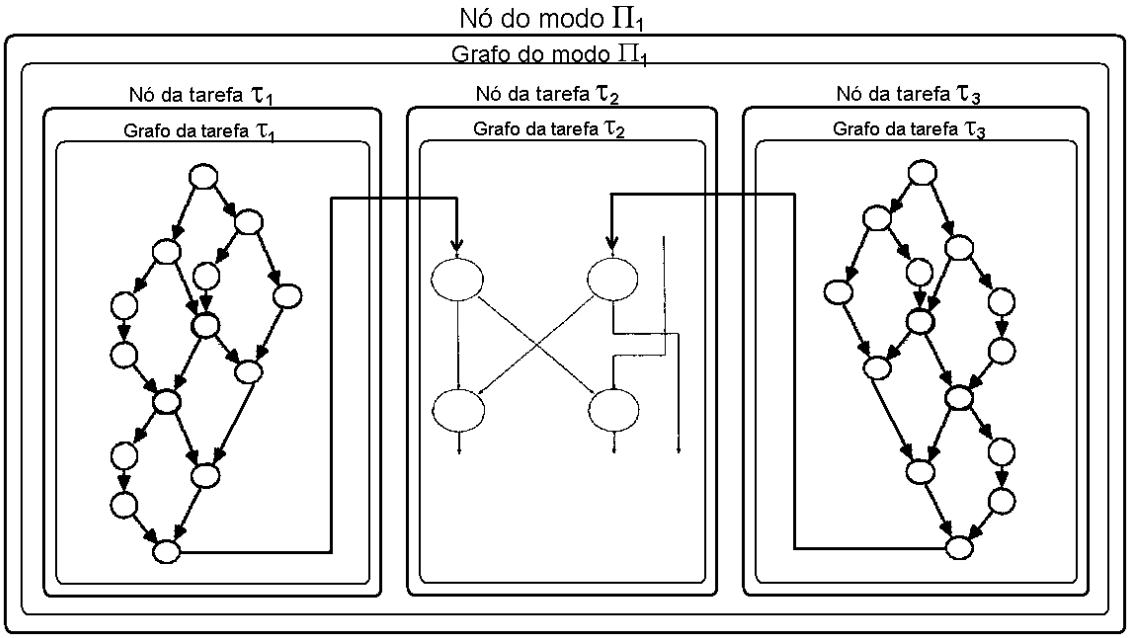


Figura 2.2: Estrutura hierárquica de nós e grafos de um modo qualquer.

nó e o nome hierárquico completo. Uma aplicação pode conter várias instâncias de nós do mesmo tipo, cada uma com um nome hierárquico, único em toda a aplicação, que identifica a instância e a diferencia das outras do mesmo tipo.

Na versão original do algoritmo de particionamento de PEaCE, todo nó possuía, para cada processador '`p`' e implementação '`i`', matrizes de tempo de execução (`mExTime[p][i]`), custo (`mCost[p][i]`), área (`mArea[p][i]`) e consumo de potência (`mPower[p][i]`). Entretanto, a versão original do algoritmo não trabalhava com múltiplas alternativas de implementação para os nós, portanto apesar de o índice '`i`' estar definido para as matrizes de cada nó, ele era fixado em zero durante a execução do algoritmo, o que significa que somente uma implementação era considerada. Os valores de tempo de execução, área e potência eram lidos diretamente do arquivo `TimeCost.xml`, enquanto `mCost[p][0]` deveria ser calculado pelo algoritmo de particionamento para representar o custo total da implementação de um nó em um determinado processador '`p`'.

2.4.3 Descrição do algoritmo de particionamento original

Realiza-se nesta seção uma análise do algoritmo de particionamento original de PEaCE, para sistemas compostos de vários modos, com várias tarefas.

O algoritmo de particionamento global consiste de um laço com duas partes principais, como mostrado na figura 2.3 [14]. O agendador para arquiteturas HMP (no caso, o agendador BIL) tenta agendar os grafos de tarefas nos PE's sempre visando obter o mínimo tempo de execução total (duração da agenda) possível, baseado na tabela de

perfis nó-PE. O controlador de alocação nó-PE modifica a tabela de perfis nó-PE em cada iteração, de acordo com os objetivos de projeto. A execução do algoritmo é dividida em três fases: a primeira procura agendar os nós nos recursos dos PE's. A segunda fase examina a possibilidade de reduzir ainda mais o custo do sistema pelo bloqueio, na ordem inversa, dos PE's já desbloqueados, seguido pelo agendamento do grafo. "Bloquear" um par nó-PE significa estabelecer que um nó não pode ser executado em um dado PE. Nesse processo, todos os PE's são visitados. A terceira fase faz o agendamento final dos nós das tarefas, insere os nós de comunicação e verifica a agendabilidade de todo o sistema.

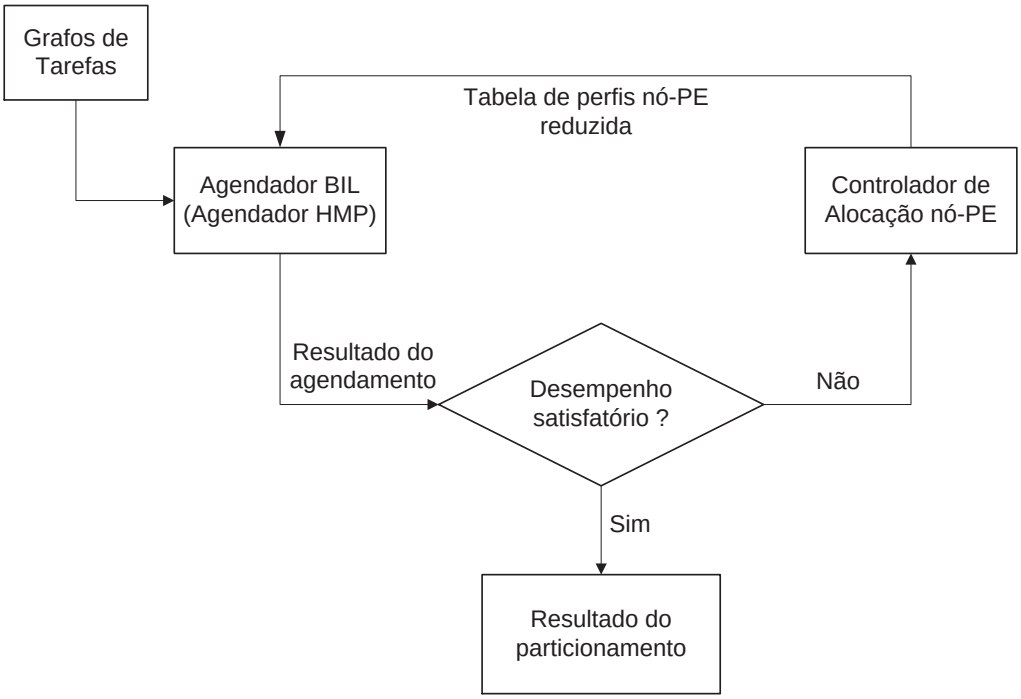


Figura 2.3: Fluxograma do algoritmo de particionamento de Oh e Ha.

As iterações começam com **o controlador de alocação nó-PE** liberando somente o PE de custo mínimo, dentre os PE's candidatos de entrada, para uma tentativa inicial de solução. Os nós de todas as tarefas são então agendados nesse PE, um por um, pelo agendador BIL. O objetivo do agendamento de cada tarefa τ_j é sempre minimizar a duração de sua agenda `sl(`τ_j`,PE)` no PE selecionado. Todos os demais PE's, exceto o PE de mínimo custo, ficam bloqueados para a alocação de nós. Esse passo é aplicado a todos os grafos de tarefa separadamente. Obtém-se o resultado do agendamento e a duração da agenda `sl(`τ_j`,PE)`.

Em seguida, o bloco de avaliação de desempenho verifica se as restrições de projeto foram atendidas. Em [16], é verificado apenas se as tarefas respeitam a restrição de agendabilidade. A utilização é a medida para determinar a agendabilidade. Com base nas durações das agendas das tarefas, `sl(`τ_j`,PE)`, o bloco de avaliação de desempenho

calcula os fatores de utilização do modo Π_i:

$$\mathbf{U}_{\Pi_i}(PE) = \sum_{\tau_j \in \Pi_i} \frac{sl(\tau_j, PE)}{T_{ij}} \tag{2.1}$$

onde T_{ij} é o período da tarefa τ_j no modo Π_i.

Cada PE possui uma utilização para cada tarefa. Se a utilização do PE for menor que 1, o bloco de avaliação de desempenho encerra as iterações e exibe os resultados do particionamento. Mas se a utilização for maior que 1, então a avaliação falha e os resultados do agendamento e a informação de violação são passados para o controlador de alocação, para que este aloque outros PE's de modo a reduzir a `sl(`τ_j`,PE)` das tarefas, até que a restrição de utilização seja satisfeita para todas.

As iterações recomeçam: o controlador de alocação nó-PE libera um nó para mapeamento em outro PE de custo não-mínimo, formando assim outros pares nó-PE de custo total mais alto. Dentre muitos PE's candidatos para implementação, deseja-se selecionar outro PE, que reduza os tempos de execução das tarefas tanto quanto possível, mas minimize o incremento de custo. Define-se portanto o Decremento Esperado na Utilização (EUD) e o Incremento Esperado de Custo (ECI) para cada PE candidato. Define-se ainda a `Folga` como a diferença entre a restrição de utilização `U*` e a utilização atual, de modo a não reduzir demais o fator de utilização, com um PE mais caro (para evitar liberar um PE de custo alto demais, sem necessidade).

$$Folga_{\Pi_i} = \mathbf{U}_{\Pi_i}(PE) - U^* \tag{2.2}$$

ECI(p_n) é simplesmente a diferença entre os custos do PE p_n e do PE anterior, e EUD(p_n) é definido como a diferença entre a utilização antes de alocar-se p_n e a utilização após alocar-se p_n.

$$EUD(p_n) = \sum_{\Pi_i \in \Pi} min(\mathbf{U}_{\Pi_i}(PE) - \mathbf{U}_{\Pi_i}(PE \cup \{p_n\}), Folga_{\Pi_i}) \tag{2.3}$$

Após calcular-se EUDs e ECIs de todos os PE's, escolhe-se uma entrada entre os pares nó-PE utilizados que tenha o maior valor da razão EUD/ECI dentre os PE's não selecionados, uma vez que dessa forma a utilização deverá diminuir significativamente com o mínimo aumento de custo. O algoritmo desbloqueia o PE selecionado, e o controlador de alocação entrega essa tabela de perfis nó-PE modificada para o agendador BIL. Não é garantido, entretanto, que os nós serão mapeados para o PE recém-selecionado, o que só ocorrerá quando a duração total da agenda for realmente reduzida, considerando-se os tempos extras de comunicação.

Repete-se esta iteração básica até que se possa agendar todas as tarefas dentro da restrição de utilização, quando então encerra-se o laço de iterações. Quando isso acontecer, o controlador examina a possibilidade de reduzir ainda mais o custo do sistema, re-bloqueando os PE's alocados na ordem inversa; um PE é travado pela fixação do tempo de execução do nó nesse PE em infinito. Agendando o grafo, se a duração da agenda ficar maior que a restrição de tempo, ou o custo não for reduzido, então a alocação do PE é restaurada. Repete-se esses passos de redução de custo até que todos os PE's alocados tenham sido visitados.

PEaCE usa o método do compartilhamento de tarefas, o que significa que se uma tarefa é comum a vários modos, o resultado do particionamento da tarefa é sempre exatamente o mesmo em cada modo. Aplicar o particionamento a cada modo em separado não leva a um resultado ótimo se uma tarefa é comum a múltiplos modos.

Para um problema de um único modo ou aplicação, não é necessário calcular a utilização. Ao invés dela, a duração da agenda de cada módulo pode ser usada nos cálculos [14]. Os procedimentos são similares aos descritos acima.

Exemplo de particionamento com o algoritmo original

Seja o sistema da figura 2.4a, que tem dois modos de operação e duas tarefas diferentes, que consistem de três nós funcionais, com dois nós compartilhados entre as tarefas. Há dois processadores candidatos e diferentes implementações em *hardware* para os nós funcionais, como mostrado na tabela de perfis nó-PE da figura 2.4b.

O período T_{ij} e o *deadline* D_{ij} de cada tarefa τ_j dependem do modo Π_i em que ela vai ser executada: a mesma tarefa pode ser agendada com freqüências de execução diferentes em modos diferentes, conforme as exigências de desempenho de cada modo. No caso de uma tarefa esporádica, T_{ij} pode ser estipulado como o mínimo tempo entre chegadas de pedidos sucessivos.

Por simplicidade, considera-se que o objetivo deste projeto seja minimizar o custo do sistema. Portanto, o controlador de alocação de PE's aloca inicialmente o PE de menor custo: `P1`. A tabela de perfis nó-PE reduzida é mostrada na figura 2.4c. Conseqüentemente, o agendador BIL mapeia todos os nós em `P1`, como mostrado na figura 2.4d, e obtém os resultados de agendamento para os dois grafos.

O próximo passo é a avaliação de desempenho, que testa se as tarefas são agendáveis. Como discutido anteriormente, a utilização é a medida para determinar a agendabilidade. Do agendamento da figura 2.4d, a utilização do modo Π_1, $\mathbf{U}_{\Pi_1}$, torna-se 1,14 (=25/40 + 31/60), e $\mathbf{U}_{\Pi_2}$ resulta 1,03 (=31/30). Para uma restrição de utilização de 1 (100%), devem ser alocados mais PE's de modo a reduzir a $sl(\tau_j, PE)$ de todas as tarefas, até que

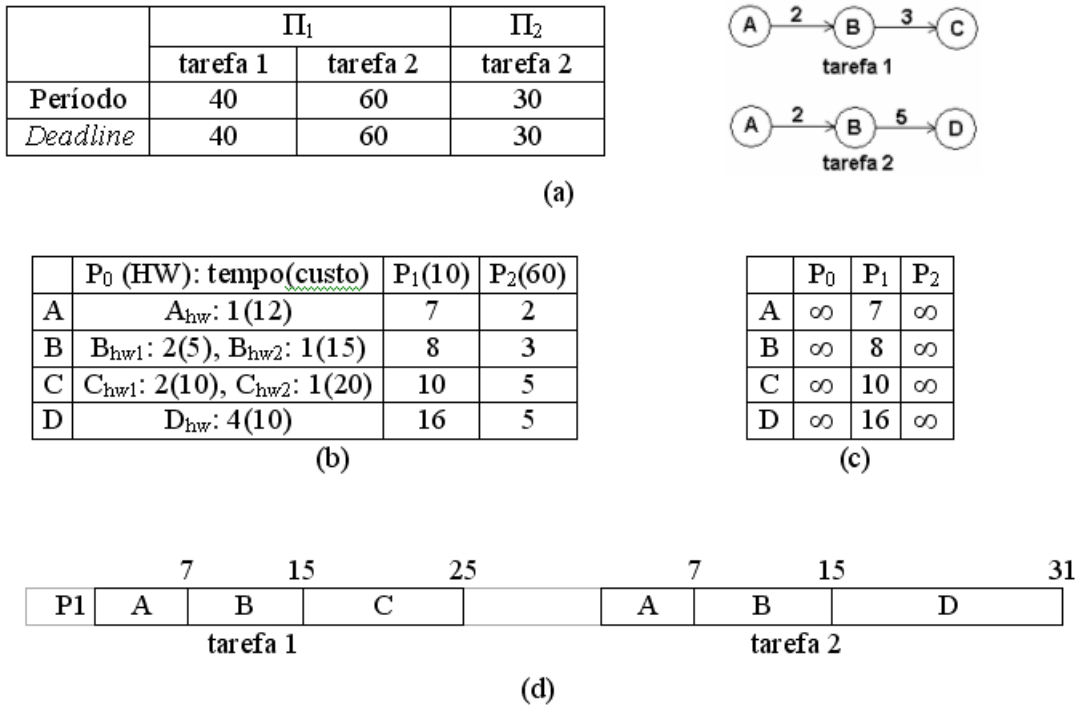


Figura 2.4: Exemplo de sistema multi-modo. (a) Modos e grafos de tarefas; (b) Tabela de perfis nó-PE; (c) Tabela de perfis nó-PE reduzida inicialmente; (d) Resultados do agendamento.

a restrição de utilização seja satisfeita para todas.

A figura 2.5a representa os valores de EUD e ECI de PE's candidatos no início da segunda iteração. Por exemplo, $EUD(A_{hw})$ é a soma de min(1,14 - (21/40 + 27/60), 0,14) do modo Π_1 e min(1,03 - 27/30, 0,03) do modo Π_2. Neste exemplo, A_{hw} é escolhido por apresentar a maior razão de todas. O resultado do agendamento HMP é mostrado na figura 2.5c: agora $\mathbf{U}_{\Pi_1} = 21/40 + 27/60 = 0,975$ e $\mathbf{U}_{\Pi_2} = 27/30 = 0,9$. Uma vez que todas as tarefas já podem ser agendadas dentro da restrição de utilização, o laço de iterações é encerrado.

Na próxima seção, descreve-se o algoritmo BIL original [18], responsável pelo agendamento de uma tarefa individual.

2.4.4 Algoritmo de agendamento BIL

O algoritmo BIL [18] é o algoritmo de agendamento executado para os nós de cada tarefa de cada um dos modos do sistema. Todo nó passa a ter um vetor de valores de BIL indexado pelo processador 'p', `mBIL[p]`. O índice BIL de um nó é o mais longo tempo de execução total, para um caminho partindo desse nó até o fim do grafo (até um dos nós terminais). Os nós hierárquicos não possuem um valor de BIL próprio, somente os

	Duração da agenda		EUD	ECI	EUD/ECI
	tarefa 1	tarefa 2			
A_{hw}	21	27	0,17	12	**0,014**
B_{hw1}	24	31	0,03	5	0,005
B_{hw2}	23	31	0,05	15	0,003
C_{hw1}	20	31	0,125	10	0,013
C_{hw2}	19	31	0,15	20	0,008
D_{hw}	25	24	0,15	15	0,010
P_2	10	10	0,17	60	0,003

(a)

	P_0	P_1	P_2
A	1	7	∞
B	∞	8	∞
C	∞	10	∞
D	∞	16	∞

(b)

(c)

Figura 2.5: Particionamento. (a) Valor EUD/ECI para todos os PE's candidatos; (b) Tabela de perfis nó-PE modificada após a seleção de A_{hw}; (c) Resultados do agendamento.

nós atômicos que os compõem. No início do agendamento, os valores de BIL dos nós do grafo da tarefa corrente são calculados na ordem inversa da ordem topológica dos nós, isto é, primeiramente é calculado o BIL do último nó atômico do grafo (ou de um dos nós terminais, isto é, que não tiver mais nós descendentes) e depois de seus sucessivos antecessores, no sentido inverso ao dos arcos, até finalmente calcular-se o BIL do primeiro nó. Isso porque o BIL de um nó n_i qualquer é função dos valores de BIL de todos os seus nós descendentes no grafo, até o último nó de cada caminho. Assim, para que o BIL de n_i seja calculado, primeiro é preciso calcular os valores de BIL de todos os seus nós descendentes, isto é, do fechamento transitivo de n_i. É por isso que o algoritmo sempre começa seus cálculos por um nó terminal do grafo.

As etapas da execução do algoritmo de agendamento BIL para o grafo de uma tarefa são descritas a seguir:

1. Seja a variável n_i inicializada com o primeiro nó do grafo da tarefa. Sejam `time` e `bil`, respectivamente, o tempo de execução mExTime[p][0] deste nó no processador '`p`', e o valor do índice BIL deste nó em '`p`' devido aos seus descendentes no grafo. Quando n_i for um nó terminal do grafo (sem sucessores), `bil` começa valendo 0.

2. Enquanto (n_i possuir sucessores), $n_i \leftarrow$ (primeiro sucessor da lista de sucessores de n_i).

3. Se n_i for um nó hierárquico, o algoritmo entra no seu grafo interno e reinicia a execução no passo 1 para este grafo. Dessa forma, é garantido o cálculo do valor de BIL de todos os nós atômicos dentro do grafo de uma tarefa, em todos os níveis hierárquicos.

4. Para cada 'p', recalcula-se o valor de BIL[p] de n_i como o valor anterior de BIL (`bil`) mais o produto de `time` pelo número total de repetições da execução do nó: `mBIL[p] = bil + time*mNumGlobalRepetitions`. Para cada n_i, `bil` será a soma acumulada dos valores de BIL de cada sucessor e dos sucessores destes, e assim sucessivamente até o nó terminal de cada caminho.

5. O algoritmo encontra `mProcWithMinBIL`, o processador onde o BIL de n_i é mínimo.

6. Designando-se agora o nó cujo BIL acaba de ser calculado como n_j, ele foi chamado por ser sucessor de outro nó, n_i. O BIL de n_i em cada 'p' agora vai ser acrescido dos efeitos da "execução" (agendamento) de n_j em 'p' e da comunicação de n_i com n_j.

6.1. É calculado o acréscimo de tempo de comunicação (IPC) de n_i para seu nó sucessor corrente n_j, estando n_i em 'p' e n_j no processador onde seu BIL é mínimo (`mProcWithMinBIL`). Este é o atraso no arco ligando os dois nós. Se ambos estiverem no mesmo processador, o atraso de comunicação é zero, caso contrário ele é calculado de uma forma única para qualquer par de PE's, levando em conta o tamanho da palavra comunicada em *bytes* (que depende somente da aplicação) e a taxa de comunicação. A contenção no barramento não é considerada nesta versão.

7. O efeito do nó sucessor n_j sobre o BIL de n_i é a adição do mínimo valor de BIL de n_j (`MinBIL(`n_j`) = BIL(`n_j`,mProcWithMinBIL)`) com o IPC do passo 6.1. Esse novo valor corrigido de BIL para n_i é chamado de "`newBIL`", e assume que n_j está mapeado em `mProcWithMinBIL`.

8. Se `newBIL` ficar maior que `BIL(`n_j`,p)` ('p' é o mesmo processador considerado para o mapeamento de n_i), isso terá sido por causa de `IPC(`n_i`,`n_j`)`. Significa que é melhor mapear n_j junto com n_i em 'p', mesmo que o BIL de n_j não seja mínimo em 'p'. Pois mapear n_j no seu `mProcWithMinBIL`, apesar de minimizar o seu BIL, acrescenta um sobrecusto de tempo de comunicação que elimina essa vantagem. Então, se `newBIL > BIL(`n_j`,p)`, `newBIL` $\leftarrow$ `BIL(`n_j`,p)`. Isso é equivalente a mapear n_j em 'p' ao invés de em `mProcWithMinBIL`. Desse modo:

`newBIL = min { MinBIL(`n_j`) + IPC(p, mProcWithMinBIL(`n_j`)), BIL(`n_j`,p) }`

9. O BIL próprio de n_i em cada processador 'p' ainda não foi calculado, então bil = `BIL(`n_i`,p)` é inicializado em `newBIL`, apenas para o primeiro sucessor n_j.

10. O algoritmo continua percorrendo a lista de nós sucessores n_j do nó n_i, para calcular os valores de BIL de cada um. Se `newBIL` devido ao n_j corrente em 'p' for maior que `BIL(`n_i`,p)`, então `BIL(`n_i`,p)` $\leftarrow$ `newBIL`. Após considerarem-se todos os sucessores de n_i, `BIL(`n_i`,p)` será igual ao máximo `newBIL` que tiver aparecido naquele 'p' devido a algum nó sucessor.

11. Repete-se os passos 3-10 até retornar-se aos nós iniciais do grafo da tarefa. O índice BIL foi calculado para todos os nós atômicos do grafo da tarefa

12. Identificam-se os nós do grafo da tarefa que estão prontos para serem "executados" (agendados) neste passo do agendamento, os chamados "*runnable nodes*", que são os nós cujos antecessores já foram agendados. Esses nós são colocados em uma lista de nós executáveis (agendáveis).

13. Enquanto existir um nó executável no grafo da tarefa, percorre-se a lista corrente desses nós, e para cada nó `node` atômico:

13.1. Calcula-se o seu índice BIM para cada processador 'p' da arquitetura, definido como a melhor duração de agenda possível para `node`: `BIM(node,p) = ECST(node,p) + BIL(node,p)`, onde `ECST(node,p) = max(DRT(node), RRT(node,p))`. `DRT(node)` é o instante quando todos os dados de entrada necessários para a execução de `node` tornam-se disponíveis, e `RRT(node,p)` é o instante quando o recurso de 'p' onde `node` foi mapeado torna-se disponível.

14. Determina-se o maior valor de BIM a ocorrer dentre todos os nós agendáveis, `largestBIM`, e qual o nó que possui esse valor, `nodeWithLargestBIM`, independente do processador. Se houver nós "empatados" nesse critério (mais de um nó com o mesmo valor `largestBIM`), o algoritmo tenta desempatar: encontrar o maior valor de BIM que também seja único, isto é, que nenhum outro nó possua, ou um único nó com o máximo BIM.

15. Uma vez encontrado um nó com o máximo BIM, seu BIM é recalculado para

todo processador 'p' da arquitetura, agora considerando o fator de correção que mede o grau de paralelismo do grafo, que é a diferença entre o número de nós simultaneamente agendáveis (executam em paralelo) e o número de processadores, normalizada por este último: `Paralelismo = numParNodes/numProcs - 1`. Esse fator de correção é multiplicado pelo tempo de execução total de `nodeWithLargestBIM` em cada 'p', no cálculo do novo BIM, indicado como BIM*. Portanto, `BIM*(node,p) = ECST(node,p) + BIL(node,p) + ExTime(node,p) * mNumGlobalRepetitions(node) * Paralelismo`.

16. Determina-se o processador ótimo `optProcId` para mapear o nó de máximo BIM `nodeWithLargestBIM`, que é o processador onde seu BIM* assume seu valor mínimo, dentre todos os processadores. Seleciona-se o processador com o menor valor de BIM* porque ele corresponde à menor combinação de BIL e ECST.

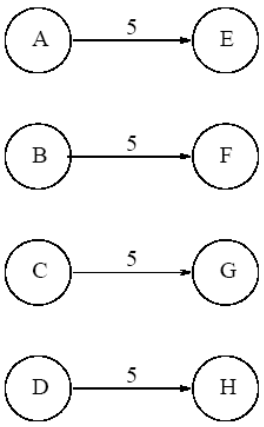
17. Finalmente mapeia-se e agenda-se `nodeWithLargestBIM` no processador ótimo para ele, `optProcId`.

18. Atualiza-se a lista de nós executáveis do grafo da tarefa, removendo o nó que acaba de ser agendado e acrescentando seus sucessores, e retorna-se ao passo 13 para prosseguir no laço de agendamento dos nós executáveis, até o último nó.

Como um exemplo da execução de um agendamento pelo algoritmo BIL, seja o grafo da figura 2.6, cujos nós possuem os tempos de execução listados na tabela, em dois PE's candidatos `P0` e `P1`. Os valores sobre os arcos correspondem aos tempos de comunicação entre os nós quando estes são mapeados em PE's diferentes.

As tabelas (a)-(g) da figura 2.7 mostram os valores de BIL e as etapas de cálculo dos valores de BIM e de mapeamento dos nós desse grafo nos PE's `P0` e `P1`. Todos os valores de BIL foram calculados nos passos 4-11 do algoritmo. Os primeiros nós que estão prontos para serem agendados são `A`, `B`, `C` e `D`. Seus valores de BIM foram calculados no passo 13.1. O nó com o maior valor de BIM é `A`. Para o cálculo de seu BIM*, como há quatro nós simultaneamente agendáveis (`k = 4`) e dois PE's (`N = 2`), então `Paralelismo = 1`. `A` é mapeado em `P1`, onde seu BIM* é menor. O `RRT(P1)` passa a ser igual ao tempo de execução de `A` em `P1`, 5.

Na segunda iteração, `E` torna-se agendável e os valores de BIM dos nós agendáveis são recalculados, por causa do agendamento de `A`. O maior valor de BIM passa a ser o de `B` e `C`, igual a 17. Escolhendo-se `B` e calculando-se seus índices BIM*, determina-se que `B` seja mapeado em `P0`. Então `F` torna-se agendável e o processo é reiniciado. Mais adiante



	P0	P1
A	9	5
B	5	7
C	5	7
D	5	5
E	5	11
F	5	5
G	5	5
H	5	5

Tempos de execução

Figura 2.6: Grafo para agendamento em dois processadores.

no mapeamento, a escolha de mapear `G` em `P0` ou `P1` é indiferente, mas uma vez feita ela determina o mapeamento de `H`. O agendamento final é mostrado na figura 2.7(h).

2.4.5 Complexidade temporal

A complexidade temporal `S` do agendador BIL é `S = O(N`^{2}`.p.log(p))`, onde `N` é o número de nós do grafo de uma tarefa e `p` é o número de PE's considerados para o agendamento [18]. A complexidade temporal do algoritmo de particionamento global de [16] torna-se $O(S.P^2.N_t)$, onde `P` é o número total de PE's candidatos e N_t é o número de tarefas. Uma vez que a complexidade de tempo deste algoritmo é muito baixa e linear com o número de tarefas, ele é aplicável eficientemente a sistemas de grande tamanho.

2.4.6 Depuração do código do algoritmo de particionamento original

Uma análise detalhada do código-fonte original do algoritmo de particionamento de PEaCE, tal como ele é disponibilizado na versão atual (1.0.1) da ferramenta [17], mostrou tratar-se de um código bastante incompleto e com muitos erros de lógica relevantes, não correspondendo à descrição feita em [16]. Esse código precisou ainda de um esforço considerável de programação apenas para passar a fazer um particionamento válido, e então poder ser tomado como base de comparação para avaliar-se o impacto das otimizações propostas sobre a qualidade final de um sistema heterogêneo.

A ferramenta PEaCE possui todo o seu código-fonte escrito na linguagem C++ para o sistema operacional Linux Red Hat 8.0. Somente a interface gráfica com o usuário foi escrita em Java, porém ela ainda encontra-se em processo de desenvolvimento e por

Nós	BIL		BIM		BIM*		optProcId
	P0	P1	P0	P1	P0	P1	
A	14	15	14	15	23	20	P1
B	10	12	10	12	-	-	-
C	10	12	10	12	-	-	-
D	10	10	10	10	-	-	-
E	5	11	-	-	-	-	-
F	5	5	-	-	-	-	-
G	5	5	-	-	-	-	-
H	5	5	-	-	-	-	-

(a)

Nós	BIL		BIM		BIM*		optProcId
	P0	P1	P0	P1	P0	P1	
A	14	15	14	15	23	20	P1
B	10	12	10	17	15	24	P0
C	10	12	10	17	-	-	-
D	10	10	10	15	-	-	-
E	5	11	15	16	-	-	-
F	5	5	-	-	-	-	-
G	5	5	-	-	-	-	-
H	5	5	-	-	-	-	-

(b)

Nós	BIL		BIM		BIM*		optProcId
	P0	P1	P0	P1	P0	P1	
A	14	15	14	15	23	20	P1
B	10	12	10	17	15	24	P0
C	10	12	15	17	20	24	P0
D	10	10	15	15	-	-	-
E	5	11	15	16	-	-	-
F	5	5	10	15	-	-	-
G	5	5	-	-	-	-	-
H	5	5	-	-	-	-	-

(c)

Nós	BIL		BIM		BIM*		optProcId
	P0	P1	P0	P1	P0	P1	
A	14	15	14	15	23	20	P1
B	10	12	10	17	15	24	P0
C	10	12	15	17	20	24	P0
D	10	10	20	15	25	20	P1
E	5	11	15	16	-	-	-
F	5	5	15	15	-	-	-
G	5	5	15	15	-	-	-
H	5	5	-	-	-	-	-

(d)

Nós	BIL		BIM		BIM*		optProcId
	P0	P1	P0	P1	P0	P1	
A	14	15	14	15	23	20	P1
B	10	12	10	17	15	24	P0
C	10	12	15	17	20	24	P0
D	10	10	20	15	25	20	P1
E	5	11	15	21	20	32	P0
F	5	5	15	15	-	-	-
G	5	5	15	15	-	-	-
H	5	5	15	15	-	-	-

(e)

Nós	BIL		BIM		BIM*		optProcId
	P0	P1	P0	P1	P0	P1	
A	14	15	14	15	23	20	P1
B	10	12	10	17	15	24	P0
C	10	12	15	17	20	24	P0
D	10	10	20	15	25	20	P1
E	5	11	15	21	20	32	P0
F	5	5	20	15	25	20	P1
G	5	5	20	15	-	-	-
H	5	5	20	15	-	-	-

(f)

Nós	BIL		BIM		BIM*		optProcId
	P0	P1	P0	P1	P0	P1	
A	14	15	14	15	23	20	P1
B	10	12	10	17	15	24	P0
C	10	12	15	17	20	24	P0
D	10	10	20	15	25	20	P1
E	5	11	15	21	20	32	P0
F	5	5	20	15	25	20	P1
G	5	5	20	20	20	20	P0
H	5	5	20	20	20	20	P1

(g)

	5	10	15	20
P0	B	C	E	G
P1	A	D	F	H

(h)

Figura 2.7: (a)-(g) Etapas do agendamento BIL para o grafo da figura 2.6; (h) Resultado do agendamento.

isso possui muitos erros de codificação, que tornam a sua utilização ineficiente. Sendo assim, optou-se por não utilizar PEaCE como uma ferramenta autônoma, mas somente aproveitar o seu código-fonte em C++ como ponto de partida para a implementação das otimizações do particionamento.

Decidiu-se migrar todo o trabalho de codificação do algoritmo otimizado para o sistema operacional Windows XP^{TM}, por causa da disponibilidade do ambiente de de-

senvolvimento Microsoft Visual Studio^{TM} 2003 com Visual C++ .NET^{TM}, que facilita enormemente o processo de depuração do código original e o desenvolvimento do novo código. Após obter-se PEaCE na internet [17], identificou-se os arquivos-fonte C++ responsáveis exclusivamente pela implementação do algoritmo de particionamento, e a partir daí passou-se a trabalhar somente com esses arquivos, com o intuito de criar-se um programa de particionamento independente do restante do ambiente PEaCE.

No novo programa assim criado, que se constitui na primeira versão para Windows do algoritmo de particionamento da ferramenta PEaCE, o usuário fornece o nome e o tipo de sistema a particionar no início da sua execução, que pode ser multi-modo/multi-tarefa ou mono-modo/mono-tarefa, e ainda pode escolher entre dois métodos para particionamento de tarefas: "*Independent*" (Independente) e "*Overlapped*" (Sobreposto) [54]. Ambos são aplicáveis a sistemas multi-processados, mas "*Independent*" assume que uma tarefa monopoliza todo o sistema heterogêneo (sistema mono-tarefa), sem preempção por tarefas de maior prioridade: uma tarefa roda de cada vez. "*Overlapped*" é para sistemas multi-modo e multi-tarefa. A principal diferença entre os dois métodos é que "*Independent*" não considera a execução concorrente de tarefas. O método "*Overlapped*" faz o mapeamento das tarefas baseando-se na utilização do processador pela tarefa de maior prioridade, de modo a paralelizar as execuções entre tarefas. Para uma arquitetura-alvo monoprocessada, esses dois métodos produzem o mesmo resultado, logo a escolha por um ou outro torna-se indiferente.

Em função das entradas do usuário, o programa seleciona e configura o particionador de sistema correspondente, para então iniciar a sua execução. O particionador abre os três arquivos XML correspondentes ao nome do sistema a particionar e copia o(s) grafo(s) da(s) aplicação(ões) do sistema para a memória, reproduzindo nesta toda a topologia do sistema, com seus nós, arcos e subgrafos, e também os valores das propriedades associadas aos elementos do grafo. A partir daí, o processo de particionamento tem início realmente.

Passou-se então a uma fase de análise e depuração do programa de particionamento no ambiente Visual C++. Vários erros foram encontrados no código do algoritmo de particionamento original de PEaCE, que se constituíam em erros de lógica do próprio algoritmo, sem qualquer relação com as adaptações feitas para a execução deste trabalho. Uma descrição detalhada das limitações e erros encontrados, juntamente com as melhorias e correções realizadas visando a implementação de uma **versão de base** do algoritmo de particionamento, é dada em [23]. A falha mais grave foi a ausência de uma função de custo para nós, recursos internos e PE's: o programa não levava em conta os valores de tamanho de memória ou área de *hardware* ocupados pelos nós nos processadores, mas visava somente minimizar o tempo de execução da solução, não importando o seu custo. Além disso, somente dois tipos de PE's eram aceitos nas tabelas do arquivo `TimeCost.xml`:

"CPU" e "ASIC". Mesmo que na prática o usuário estivesse utilizando um FPGA, deveria declará-lo como do tipo "ASIC" nesse arquivo, porque as restrições e custos próprios de um FPGA não eram consideradas. Isso somente é feito a partir da primeira versão otimizada do algoritmo, apresentada no próximo capítulo.

Para permitir o uso de uma função de custo elementar pelo algoritmo de base, primeiramente acrescentaram-se novos campos ao arquivo `TimeCost.xml` original, informando os custos e as restrições dos PE's de *software* (CPU's) e de *hardware* (ASIC's) a serem usados no particionamento. O custo para uma CPU passa a ser o seu custo monetário por unidade. No início das tabelas de CPU's, as restrições passam a ser as quantidades disponíveis de memória, a qual é dividida em memória de programa e de dados. O custo monetário total dessas memórias passa a ser também fornecido. O valor de custo fornecido para um ASIC candidato passa a ser o seu custo monetário por unidade de área ($/mm^2). O custo final de um ASIC vai depender portanto da sua área total e da tecnologia (= custo/mm^2). No início da tabela de um ASIC, a restrição passa a ser a área máxima permitida para ele.

Os custos dos nós das aplicações em cada processador passam a ser calculados a partir dos valores fornecidos pelo usuário no arquivo `TimeCost.xml`. Um nó mapeado em uma CPU consome o espaço de memória disponível para ela (e essa memória é um recurso elementar finito). Assim, o custo de um nó em *software* passa a ser calculado tomando-se inicialmente a quantidade total de memória (em *bytes*), de programa e de dados, ocupada pelo código executável do nó. Essa quantidade de memória é em seguida normalizada, para evitar-se a soma de grandezas diferentes. Para isso, divide-se o valor da ocupação de memória pela restrição (tamanho máximo) de memória. Em seguida, o tamanho em memória normalizado do nó é multiplicado pelo preço da memória, resultando no valor final de custo já em uma unidade monetária. Como os tamanhos em memória são função da complexidade do nó, então o custo de um nó em *software* passa a ser igual ao seu custo em memória.

Para um nó mapeado em um ASIC, o custo desse nó passa a ser igual à sua área de circuito ocupada, que é a métrica clássica da literatura, porém aplicável apenas a ASIC's, onde a unidade de área possui um custo fixo e conhecido, definido pela tecnologia de fabricação. Essa área de *hardware* ocupada é multiplicada pelo custo do mm^2 de área do ASIC.

A função de custo elementar criada para o algoritmo de particionamento de base calcula da seguinte maneira o custo total de uma solução de particionamento. Todos os nós de um mesmo tipo possuem o mesmo tempo de execução e o mesmo custo, em um dado processador. As equações 2.4 e 2.5 expressam os custos dos nós nos dois tipos de PE's, que são os custos dos recursos que implementam esses nós. O custo de um recurso

em um PE passa a ser o custo de um nó do mesmo tipo nesse PE.

Para um nó n_i em um PE de software 'p':

$$Custo(n_i, p) = \frac{AreaMem(n_i, p)}{TamMemProg(p) + TamMemDados(p)} * CustoMem(p) \qquad (2.4)$$

Para um nó n_i em um PE de hardware 'p':

$$Custo(n_i, p) = Area(n_i, p) * Custo\_mm^2(p) \qquad (2.5)$$

O algoritmo de base percorre a lista de recursos alocados pelo controlador de alocação, identifica o PE ao qual cada recurso alocado pertence, e adiciona (acumula) os custos desses recursos por instância de PE. Assim, para cada PE considerado no arquivo `TimeCost.xml`, a função de custo obtém o custo total dos recursos alocados pertencentes a esse PE. Isso é feito porque cada PE tem um custo diferente de utilização, e portanto não é possível somar custos de recursos implementados em PE's diferentes, mesmo que sejam PE's do mesmo tipo. Por exemplo, dois ASIC's podem ter custos por mm^2 diferentes, se forem de tecnologias diferentes, e portanto os recursos alocados em cada um desses ASIC's têm que ser totalizados em separado. Depois o algoritmo percorre os PE's e adiciona os custos totais dos recursos alocados em cada PE ao custo monetário do PE individual (se houver), e acumula esses resultados em um custo total da arquitetura do sistema.

Para o custo da partição de *software*, o custo monetário de cada processador de *software* é adicionado ao custo total de sua(s) memória(s) associada(s). Na versão de base do algoritmo de particionamento, os consumos de potência dos nós e dos PE's não são considerados.

2.5 Conclusão

Este capítulo descreveu em detalhes o algoritmo de particionamento da ferramenta de *Codesign* acadêmica PEaCE [17], proposto por H. Oh e S. Ha [16], escolhido como o ponto de partida para a realização desta tese. A escolha por este algoritmo foi justificada e explicou-se o seu funcionamento de forma detalhada. Tendo sido corrigidos os principais erros do seu código original, e principalmente com a criação de uma função de custo elementar, pôde-se finalmente executar o particionamento corretamente, antes da implementação das otimizações propostas por esta tese. O próximo capítulo descreve essas otimizações para o algoritmo de particionamento de base, que passa a ser denominado de

"versão 1.0". Justifica-se a necessidade para cada otimização, descrevem-se as alterações que serão necessárias no código do programa, e explica-se como este passa a funcionar depois de cada mudança. Essas otimizações e suas implementações constituem-se na contribuição deste trabalho.

Tabela 2.1: Classificação dos algoritmos de particionamento investigados para sistemas com *hardware* fixo.

Algoritmo de Particionamento	Domínio de Aplicação	Arquitetura-Alvo	Custos de Comunicação HW-SW	Paralelismo de Execução HW/SW	Escolha da Alternativa de Implementação em HW ou SW	Compartilhamento de Recursos de HW entre Nós	Número de Modos do Sistema
Gupta e DeMicheli [1,2]	Multi-domínio	Clássica	Sim	Sim	Não	Não	1
Ernst et al. [4]	Multi-domínio	Clássica	Sim	Não	Manual	Não	1
Barros et al. [5,6]	Multi-domínio	Clássica	Sim	Sim	Automática ou manual, em HW	Sim	1
Kalavade e Lee [7,8]	Multi-domínio	Clássica	Sim	Sim	Automática	Não	1
Madsen et al. [9]	Multi-domínio	Clássica	Sim	Não	Não	Sim	1
Jeon e Choi [10]	Multi-domínio	Clássica	Sim	Sim	Não	Sim	1
Jeon e Choi [11]	Multi-domínio	Clássica ou HMP	Sim	Sim	Automática, em HW	Sim	1
Rousseau et al. [3,12]	Telecom/DSP	Clássica	Não	Sim	Não	Não	1
Hou e Wolf [13]	Multi-domínio	HMP sintetizada	Sim	Sim	Automática	Sim	1
Oh e Ha [14]	Multi-domínio	HMP sintetizada	Sim	Sim	Automática	Sim	1
Kalavade e Subrahmanyam [15]	Multi-domínio	HMP sintetizada	Sim	Sim	Automática	Sim	Múltiplos
Oh e Ha [16]	Multi-domínio	HMP sintetizada	Sim	Sim	Automática	Sim	Múltiplos

Tabela 2.2: Classificação dos algoritmos de particionamento investigados para sistemas com *hardware* dinamicamente reconfigurável.

Algoritmo de Particionamento	Domínio de Aplicação	Arquitetura-Alvo	Reconfiguração Parcial dos FPGAs	Custos de Comunicação HW-SW	Paralelismo de Execução HW/SW	Escolha da Alternativa de Implementação em HW ou SW	Número de Modos do Sistema
Peterson et al. [53]	Multi-domínio	Coprocessador; CCMs multi-FPGAs genéricas	Não	Sim	Não	N/D	1
Vincentelli et al. [47]	Controle	CSoC; processador de SW reconfigurável	N/D	Não há	Não	N/D	1
Harkin et al. [51]	Multi-domínio	Clássica	Não	Sim	Não	N/D	1
Chatha e Vemuri [52]	Multi-domínio	Clássica	Não	Sim	Sim	Não	1
Rakhmatov e Vrudhula [50]	Controle	Clássica	Sim	Sim	Não	Não	1
Noguera e Badia [42–45]	Multi-domínio	Clássica[a]	Não	Sim	Sim	Não	Múltiplos
Dick e Jha (CORDS) [48]	Multi-domínio	DHE sintetizada	Sim	Sim	Sim	Sim	1
Dave (CRUSADE) [49]	Multi-domínio	DHE sintetizada	Não	Não	Sim	Sim	1
Choi et al. [46]	Multi-domínio	Clássica	Sim	Não	Sim	Não	1

[a]O coprocessador dinamicamente reconfigurável possui uma arquitetura específica concebida pelos autores.

Capítulo 3

Proposta de Algoritmo Otimizado de Particionamento

Este capítulo explica as otimizações realizadas no algoritmo de particionamento de base apresentado no Capítulo 2, doravante denominado de "versão 1.0". A necessidade para cada otimização é justificada, explicando-se como o algoritmo 1.0 procedia originalmente, em seguida quais as mudanças feitas no seu código e finalmente como ele passa a funcionar depois da implementação da otimização. As otimizações propostas não foram encontradas reunidas em um mesmo algoritmo de particionamento na literatura.

A Seção 3.1 discute as modificações feitas no arquivo de entrada `TimeCost.xml` e no código do algoritmo de particionamento, com o objetivo de considerar de forma precisa as restrições de PE's de *software* e de *hardware* e os custos de nós implementados nesses processadores, para o particionamento. A versão 1.0 do particionador não considera o consumo dos recursos internos dos processadores pelos nós, e nem o consumo de potência dos nós. Este trabalho propõe-se a levar em conta esses custos de forma mais fiel à realidade, pela utilização da função de custo introduzida em [19, 20]. A subseção 3.1.1 explica as novas restrições criadas para o sistema, com o objetivo de melhor definir o espaço válido de soluções de particionamento, restringindo-o nos eixos de desempenho, custo e potência. A Seção 3.2 explica o impacto da consideração dos tempos de reconfiguração de FPGA's dinamicamente reconfiguráveis sobre a estrutura de dados e o agendamento. Finalmente, a Seção 3.3 descreve como informar, para o algoritmo de particionamento, que um nó poderá ter múltiplas alternativas de implementação.

Essas otimizações são gradativamente codificadas no algoritmo de particionamento, que é testado para cada uma individualmente, para que se possa acompanhar a evolução de sua qualidade e entender porque cada nova codificação é melhor que a anterior. A implementação e avaliação dessas otimizações, que levam a um algoritmo de particionamento otimizado, constituem-se na contribuição desta tese de doutorado.

3.1 Custos, restrições e consumos de potência dos PE's, dos nós e da solução

Esta seção descreve o conjunto de modificações realizadas no código da versão 1.0 do algoritmo de particionamento para capacitá-lo a reconhecer e tratar PE's do tipo "FPGA", em adição aos PE's dos tipos "CPU" e "ASIC" que ele já reconhecia. O programa é estendido para:

1. Ler e trabalhar com as restrições e os custos próprios de um FPGA e dos nós implementados nele, por meio de acréscimos ao arquivo `TimeCost.xml` e ao código-fonte do programa.

2. Considerar o consumo de potência estática dos PE's e de potência dinâmica dos nós em cada PE, visando o cálculo do consumo de potência total de cada partição proposta pelo algoritmo.

3. Ler e trabalhar com as novas restrições definidas para o sistema a ser particionado, incluídas no arquivo `mode.xml`.

Finalmente, acrescenta-se ao algoritmo uma função de custo englobando os custos de implementação dos nós e as restrições dos PE's em uma única expressão, garantindo assim automaticamente a validade das soluções, tal como foi demonstrado em [19, 20]. A versão do algoritmo de particionamento resultante das otimizações descritas nesta seção é doravante denominada "versão 2.0".

A primeira melhoria consiste no acréscimo do tipo de processador "FPGA" no trecho do programa que lê o arquivo `TimeCost.xml`. Ainda na versão 1.0, já eram informados os custos monetários por unidade para os processadores de *software* e *hardware* considerados para a arquitetura, no arquivo `TimeCost.xml`. Na versão 2.0, isto passa a ser feito também para os FPGA's. Entretanto, somente o custo monetário não é suficiente para expressar, de uma forma única e completa, o poder de computação de um PE de *software* ou a quantidade e diversidade de recursos de um PE de *hardware* (no caso de um FPGA). O custo monetário é um valor muito variável em função da quantidade de itens comprados, do tipo de comprador (se é universidade ou empresa), e do tempo. Além disso, nada impede que uma companhia lance no mercado um FPGA rico em recursos (quantidade e diversidade), com um preço realmente mais baixo que o de um FPGA equivalente em tamanho, de outra companhia. O custo de um mesmo recurso de *hardware* fornecido pela empresa 'X' pode ser bem diferente do seu custo pela empresa 'Y'.

Na versão 1.0, já eram especificadas as restrições para PE's de *software* (tamanho de memória disponível) e ASIC's (área máxima). Na versão 2.0, passa-se a especificar

também as restrições de FPGA's no arquivo `TimeCost.xml` e a ler seus valores com o programa. Especificar uma restrição em termos de área só faz sentido para ASIC's, para os quais o custo por mm^2 é conhecido sendo dada a tecnologia, ou quando existir uma restrição de tamanho máximo para o sistema como um todo. Para um FPGA, restrições muito diferentes têm que ser consideradas, porque trata-se de um dispositivo pré-fabricado e "fechado", com um custo unitário conhecido, e que oferece uma diversidade de tipos de recursos elementares. As quantidades totais disponíveis de cada tipo de seus recursos internos constituem as **restrições do FPGA** para a implementação de nós nele. Assim, o cabeçalho de um FPGA no arquivo `TimeCost.xml` vai listar como restrições as quantidades totais de:

- LE's (Elementos Lógicos);
- *bits* de RAM;
- blocos multiplicadores;
- blocos de PLL's;
- pinos de E/S.

que constituem os tipos de recursos elementares consumíveis por uma aplicação tipicamente encontrados em FPGA's. O custo da memória de reconfiguração associada a um FPGA também passa a ser dado na sua tabela no arquivo `TimeCost.xml`.

O algoritmo de particionamento 1.0 já considerava os custos de implementação dos nós de aplicação nos PE's, por meio das quantidades de recursos elementares de um PE consumidas pelo nó mapeado nele. Esses recursos elementares podiam ser memória ou área, conforme o PE fosse uma CPU ou um ASIC. Na versão 2.0, passa-se a informar, para cada nó na tabela de um FPGA, as quantidades de cada tipo de recurso elementar do FPGA que são ocupadas por este nó, quando mapeado nesse FPGA; a partir dessas quantidades, calcula-se o custo da implementação deste nó no FPGA. A grande maioria dos algoritmos de particionamento da literatura considera tão-somente a área de *hardware* absoluta ocupada por um nó como a medida do seu custo de implementação em *hardware*. Entretanto, esta métrica somente é adequada para ASIC's, onde a área do circuito determina diretamente o seu custo monetário: `preço = (custo/mm`^{2}`).área`. Não foi encontrada, na literatura, a consideração da heterogeneidade e das quantidades dos recursos elementares internos de um FPGA para fins de particionamento, o que é um indicativo da originalidade desta tese.

O algoritmo de particionamento 1.0 não verificava se as ocupações dos recursos elementares de um PE, por um nó de aplicação mapeado no mesmo, respeitavam as restrições

correspondentes do PE. Ou seja, não era verificada a validade dos mapeamentos feitos pelo controlador de alocação nó-PE, pois não havia um acompanhamento das quantidades de recursos elementares ainda disponíveis no PE, após cada iteração do algoritmo. Esse problema é resolvido na versão 2.0 com a verificação das restrições antes do mapeamento de cada nó em um PE, e com a introdução de uma função de custo. Se o nó em questão não respeitar as restrições próprias do tipo de PE para o qual foi alocado, o controlador de alocação deve alocá-lo em outro PE.

- Se o PE for uma CPU, verifica-se se a memória ocupada pelo nó é menor ou igual que a memória disponível; caso positivo, atualiza-se a memória disponível pela subtração da memória do novo nó de *software*.
- Se o PE for um ASIC, verifica-se se a área ocupada pelo nó é menor ou igual que a área que resta disponível nesse ASIC; caso positivo, a restrição de área disponível é atualizada pela subtração da área do nó.
- Se o PE for um FPGA, verifica-se se as ocupações de recursos elementares pelo nó não excedem os limites de quantidade disponível de cada tipo de recurso elementar no FPGA. Somente se todas as ocupações de recursos estiverem dentro do permitido, o nó é autorizado a ser mapeado no FPGA. As quantidades ocupadas de cada recurso elementar são subtraídas das quantidades disponíveis, atualizando-as para os próximos mapeamentos.

No arquivo `TimeCost.xml`, cada nó atômico em uma tabela de FPGA está listado com a sua quantidade de pinos de E/S ocupados, que é fornecida pelo ambiente de síntese do fabricante desse FPGA (no contexto desta tese, esse ambiente é o Quartus II^{TM}). Essa informação da quantidade de pinos utilizados por um nó no FPGA é passada para o algoritmo apenas para constar no relatório final de custos emitido pelo programa de particionamento, mas ela **não** deve ser tomada como uma medida de custo da implementação desse nó no FPGA. Muito embora uma exigência de pinos de E/S muito grande possa levar à escolha de um outro FPGA, maior do aquele que bastaria para a lógica do projeto, isso será indicado previamente pelo ambiente de síntese utilizado, quando da síntese de um nó hierárquico ou do projeto completo. Porém, se o particionador mapear vários nós para *hardware*, o algoritmo não tem condições de calcular uma quantidade total de pinos para esse conjunto, pois não existe uma regra para se obter a quantidade de pinos de um bloco hierárquico sabendo-se as quantidades de pinos ocupados por seus nós internos. A prática mostra que a simples adição das quantidades de pinos dos nós individuais resulta muito maior que a quantidade de pinos real do bloco hierárquico que os contém, e pode resultar mesmo maior que a quantidade de pinos disponíveis no FPGA inteiro, falseando a

medida de custo da implementação. Por causa disso, as quantidades de pinos não entram no cálculo do custo de um nó e nem na função de custo.

Para a versão 2.0 do programa de particionamento, acrescenta-se um campo à tabela de cada PE no arquivo `TimeCost.xml` para a sua potência estática. Passa-se também a ler e tratar as potências dinâmicas dos nós em cada PE. O particionador trata as potências dos nós da mesma forma que seus custos de implementação nos PE's, mas independentemente destes, isto é, as potências são totalizadas em separado para cada PE e somente são combinadas aos custos dos recursos na função de custo global.

3.1.1 Restrições dos modos do sistema

No arquivo de entrada `mode.xml`, especificam-se as restrições para a implementação de cada modo. Na sua versão original, esse arquivo continha somente os períodos e *deadlines* das tarefas, e somente o período era realmente utilizado. Na versão 2.0 do algoritmo, o arquivo `mode.xml` passa a especificar o número de modos do sistema e as restrições globais de desempenho (o tempo de execução total), custo monetário e consumo de potência para cada modo. Esses são os valores máximos aceitáveis pela aplicação.Em sistemas mono-tarefa como os utilizados nesta tese, a restrição de tempo de execução do modo não é considerada pelo particionador, que usa o período da tarefa como a restrição de desempenho para o particionamento.

As restrições de projeto servem para delimitar as soluções de particionamento propostas para o sistema que podem ser consideradas válidas, definindo o espaço de busca da solução final. Elas servem também como critérios para a avaliação da qualidade de um sistema particionado e para a comparação entre as qualidades de diferentes sistemas, por meio da comparação de cada parâmetro de qualidade (desempenho, custo ou potência) de uma solução de particionamento com a sua restrição correspondente. A forma como as restrições impostas às aplicações são utilizadas é explicada na próxima seção.

3.1.2 Funções de custo otimizadas

Os objetivos de uma função de custo são: guiar o algoritmo de particionamento para uma meta, dentro do espaço de projeto válido, e medir a qualidade de cada solução proposta, por meio da estimativa de seu custo global. Uma boa função de custo consiste de uma soma de vários custos ou objetivos de projeto, ponderados e normalizados, com os quais se possa experimentar diversos compromissos de particionamento por meio da alteração dos pesos. Neste trabalho, adota-se a formulação unificada para funções de custo proposta em [19, 20]: a expressão da função integra, além dos custos, termos de correção (um para cada restrição) para guiar a busca pela solução de mínimo custo global

sempre dentro do espaço de projeto válido. Essa formulação tem a vantagem de verificar a violação das restrições do sistema diretamente na própria expressão da função de custo. A forma geral de tal função de custo é dada na equação 3.1:

$$F(\wp) = \sum_i \kappa_i \times \frac{C_i(\wp)}{C_i} + \sum_i \kappa_{C_i} \times F_C(C_i(\wp)) \tag{3.1}$$

onde $\wp$ é a partição sob avaliação, $C_i(\wp)$ é o valor de um custo particular (atributo de qualidade), κ_i é o peso atribuído a este custo na avaliação da partição, C_i é a restrição de projeto aplicada ao custo $C_i(\wp)$ na partição $\wp$, e que serve para a normalização, $F_C(C_i(\wp))$ é o termo de correção e κ_{C_i} é o peso para o termo de correção. Os termos de correção $F_C(C_i(\wp))$ para a função de custo neste trabalho são os termos de penalidade introduzidos em [19, 20].

Os termos de correção por penalidade punem fortemente a exploração de soluções que resultariam em violações médias ou grandes das restrições, mas permitem a exploração de soluções próximas ao limite das restrições. Funções com este tipo de termo de correção são adequadas quando as restrições do sistema não são rígidas. O peso κ_{C_i} associado é importante em determinar o quão próxima uma solução precisa estar da fronteira das restrições para ser aceita ou não. Os termos de penalidade procuram minimizar os parâmetros do sistema aos quais são associados, que ficam bem abaixo das suas restrições de projeto correspondentes, atendendo-as com folga. A expressão analítica geral para o termo de correção por penalidade é dada na equação 3.2:

$$F_C(C_i, C_i(\wp)) = \sum_i r^2[C_i(\wp), C_i] \tag{3.2}$$

onde

$$r[C_i(\wp), C_i] = \max\left\{0, \frac{[C_i(\wp) - C_i]}{C_i}\right\} \tag{3.3}$$

Quando se analisa o comportamento de um parâmetro isoladamente no particionamento, sem o uso dos termos de correção na função de custo, pode-se até obter valores (custos) inferiores aos obtidos com o emprego dos termos de correção, porém ao mesmo tempo outras restrições de projeto podem estar deixando de ser atendidas, o que não deve ser permitido em hipótese alguma.

Geralmente, quando um FPGA não é completamente ocupado por uma aplicação, seus recursos que ficaram ociosos são desperdiçados pelo sistema. Entretanto, para as aplicações consideradas neste trabalho, o FPGA candidato no particionamento é grande demais até mesmo para implementá-las por completo, de forma que uma boa parte de

seus recursos internos fica forçosamente desocupada. Por causa disso, o algoritmo de particionamento procura minimizar a ocupação de recursos do FPGA na partição final, o que significa que, para implementá-la, um FPGA muito menor e mais barato pode ser utilizado.

Na versão 2.0 do algoritmo de particionamento, os cálculos dos custos de mapeamento dos nós nos PE's (consumos de recursos elementares), e dos consumos de potência resultantes em cada PE, são feitos juntos. A função de custo engloba os custos e os consumos de potência em um valor único, que representa globalmente a qualidade da solução. Esses cálculos são implementados da seguinte maneira:

1. Os custos dos nós são calculados como os totais de recursos elementares consumidos. Cada instância de nó tem o seu custo calculado em cada processador `p`. Evidentemente, instâncias de nós do mesmo tipo têm o mesmo custo em um mesmo PE. As equações dos custos são as seguintes:

Para um nó n_i em um PE de software `p`*:*

$$Custo(n_i, p) = \frac{TamMemProg(n_i, p) + TamMemDados(n_i, p)}{TamMemProg(p) + TamMemDados(p)} \tag{3.4}$$

Para um nó n_i em um ASIC `p`*:*

$$Custo(n_i, p) = Area(n_i, p) \tag{3.5}$$

Para um nó n_i em um FPGA `p`*:*

$$Custo(n_i, p) = \frac{[\frac{LEs(n_i,p)}{LEs(p)} + \frac{RAMs(n_i,p)}{RAMs(p)} + \frac{Mults(n_i,p)}{Mults(p)} + \frac{PLLs(n_i,p)}{PLLs(p)}]}{4} \tag{3.6}$$

A equação 3.6 totaliza o custo de um nó implementado em um FPGA. A divisão por 4 é feita para manter esse custo na faixa de 0 a 1.

2. O custo e a potência de um recurso alocado pelo controlador de alocação, em cada iteração do particionamento, passam a ser o custo e a potência de um nó do mesmo tipo, no mesmo processador onde esse recurso está alocado.

Se tipo(RecAloc) = tipo(n_i):

$$CustoRecAloc(p) = Custo(n_i, p) \tag{3.7}$$

$$PotenRecAloc(p) = Potencia(n_i, p) \tag{3.8}$$

3. Para cada instância de PE no arquivo `TimeCost.xml`, totalizam-se o custo e a potência dos recursos alocados que pertencem a esse PE. Por exemplo, dois FPGA's diferentes são totalizados em separado. Isso é feito porque cada PE tem um custo monetário e uma potência estática diferentes, e portanto não se pode somar custos e potências de recursos alocados em PE's diferentes, mesmo que sejam PE's do mesmo tipo.

$$CustoRecsAlocs(p) = \sum CustoRecAloc(p), \forall p, \forall RecAloc \tag{3.9}$$

$$PotenRecsAlocs(p) = \sum PotenRecAloc(p), \forall p, \forall RecAloc \tag{3.10}$$

4. O algoritmo de particionamento visita cada PE, verifica sua natureza e converte o custo total dos recursos alocados nesse PE em um custo monetário, por meio das equações:

Para uma CPU `p`:

$$CustoRecsAlocs(p) \leftarrow CustoRecsAlocs(p) \times CustoMem(p) + Custo(p) \tag{3.11}$$

Para um ASIC `p`:

$$CustoRecsAlocs(p) \leftarrow CustoRecsAlocs(p) * Custo\_mm^2(p) \tag{3.12}$$

Para um FPGA `p`:

$$CustoRecsAlocs(p) \leftarrow CustoRecsAlocs(p) \times Custo(p) + CustoMemReconf(p) \tag{3.13}$$

5. O algoritmo de particionamento visita cada PE e acrescenta a potência estática do mesmo ao valor total da sua potência dinâmica (que é devida aos recursos alocados nesse PE):

$$PotenRecsAlocs(p) \leftarrow PotenRecsAlocs(p) + PotEstat(p) \tag{3.14}$$

6. O custo total de todos os recursos alocados para a arquitetura (custo da partição) é a somatória dos custos dos recursos alocados (pares {nó, PE}) em cada PE `p`, totalizada para todos os PE's:

$$CustoPart(\wp) = \sum_{p} CustoRecsAlocs(p) \tag{3.15}$$

7. A potência total de todos os recursos alocados para a arquitetura (potência da partição) é a somatória dos consumos de potência totais de cada PE `p`:

$$PotenPart(\wp) = \sum_{p} PotenRecsAlocs(p) \tag{3.16}$$

8. Aplicam-se os termos de correção ao custo e à potência totais da partição, envolvendo as restrições dadas no arquivo `mode.xml`. A equação final da função de custo é:

$$\begin{aligned} F(\wp) = {} & \kappa_{custo} \times \frac{CustoPart(\wp)}{RestrCustoModo} + \\ & \kappa_{poten} \times \frac{PotenPart(\wp)}{RestrPotenModo} + \\ & \kappa_{C_{custo}} \times F_C(RestrCustoModo, CustoPart(\wp)) + \\ & \kappa_{C_{poten}} \times F_C(RestrPotenModo, PotenPart(\wp)) \end{aligned} \tag{3.17}$$

onde κ_{custo} e κ_{poten} são os pesos do custo monetário total e da potência total, respectivamente. Esses pesos permitem ao projetista enfatizar o atributo de qualidade desejado, dando a ele um peso ligeiramente maior que os pesos dos outros atributos. Os pesos dos termos de correção $\kappa_{C_{custo}}$ e $\kappa_{C_{poten}}$ devem receber valores altos para aumentar consideravelmente a penalidade da função de custo quando houver violações das restrições. É interessante fazer com que $\sum_i \kappa_i = 1$, sendo portanto $F(\wp) = 1$ um valor de referência, uma vez que: violações das restrições resultarão em valores $F(\wp) \gg 1$; e valores otimizados dos atributos resultarão em $F(\wp) < 1$.

Experimentou-se acrescentar um termo para a latência da tarefa na função de custo da equação 3.17, igual ao tempo de execução da tarefa (dado pelo agendamento) dividido pela sua restrição de período. Entretanto, não houve qualquer alteração nos resultados do particionamento, que resultaram idênticos aos obtidos sem a inclusão desse termo de latência. Isso se deve provavelmente ao fato de que o algoritmo de particionamento de base já faz o agendamento visando minimizar a duração da agenda de cada tarefa, e portanto não pode ser obtido um melhoramento adicional por meio da função de custo.

Em sua versão original, o programa particionador de PEaCE produzia como resultado apenas um arquivo de resultado do agendamento `Exemplo_sched.xml`. Nesse arquivo, são mostrados os nós dos grafos das aplicações agendados nos PE's escolhidos para a arquitetura-alvo, e o tempo de execução de cada processador. Acrescentou-se a todas as versões do algoritmo de particionamento, inclusive à versão 1.0, a criação de um arquivo adicional para os custos `Exemplo_costs.xml`, que fornece: as ocupações de memória (CPU), área (ASIC) e recursos elementares (FPGA) das implementações dos nós; os custos e consumos de potência das partições de *software* e de *hardware*; e o custo e o consumo de potência da solução final.

3.2 Reconfiguração dinâmica de *hardware*

A versão 2.0 do algoritmo de particionamento pode trabalhar com CPU's, ASIC's e FPGA's sem reconfiguração dinâmica. A versão 3.0 descrita nesta seção é uma extensão da anterior, feita para trabalhar com esses mesmos tipos de PE's e também com FPGA's dinamicamente reconfiguráveis, considerando seus tempos de reconfiguração durante o agendamento dos nós. Isso pode interferir decisivamente no resultado do particionamento. O algoritmo de agendamento BIL é quem escolhe os PE's a serem utilizados no sistema, visando reduzir o tempo de execução de cada tarefa, conforme explicado na Seção 2.4.4. Portanto, é o algoritmo BIL que escolhe se o *hardware* do sistema particionado será fixo ou reconfigurável, ou se ambos os tipos serão utilizados.

Na versão 3.0, implementa-se o suporte a uma reconfiguração total e disruptiva do FPGA. O tempo de reconfiguração nesse caso é o tempo para reconfigurar toda a matriz, que é quase sempre muito maior que os tempos de processamento dos nós de aplicação. Portanto, esse tipo de reconfiguração não pode ser agendado em paralelo com uma comunicação entre nós da aplicação porque todo o FPGA vai parar de operar enquanto for reconfigurado. Ela pode entretanto ser agendada em paralelo com a execução da partição de *software* do sistema. O tempo de reconfiguração de um FPGA da Altera depende de um conjunto de fatores, principalmente do esquema de configuração escolhido e do tamanho do seu arquivo de configuração, que por sua vez é um valor fixo para um mesmo FPGA e uma mesma versão do ambiente Quartus II utilizada [55].

As várias formas de implementar-se a reconfiguração dinâmica nos FPGA's Stratix II são descritas em [55], Volume 2, Capítulo 8. Os FPGA's dessa família podem manipular até sete páginas de configuração de aplicação, armazenadas em suas memórias de configuração não-voláteis. O usuário, no momento da implementação do sistema, já deve deixar essas páginas de aplicação na memória de configuração do FPGA, e implementar

neste uma lógica adicional para iniciar, no momento certo, o ciclo de reconfiguração de uma nova página no FPGA. Dependendo da relação entre o tamanho da aplicação e o tamanho do FPGA, este vai sendo reconfigurado para cada um dos estágios subseqüentes da computação, que calculam seus resultados intermediários baseados nas saídas dos estágios anteriores, guardadas em memória. Os estágios vão sendo permutados no FPGA à medida que vão sendo necessários para contribuir para um cálculo final.

Como primeira modificação para implementar a versão 3.0 do algoritmo de particionamento, adapta-se o programa para reconhecer o novo tipo de PE "FPGADin" (criado para representar um processador dinamicamente reconfigurável, diferenciado do FPGA sem reconfigurabilidade dinâmica), declarado pelo usuário no arquivo `TimeCost.xml`. Apesar do tempo de reconfiguração ser uma constante para um mesmo FPGA, o campo "tempo de reconfiguração" é criado individualmente por nó, ao invés de globalmente por FPGA, por uma questão de generalidade, prevendo a implementação futura do suporte à reconfiguração parcial no algoritmo de particionamento. Acrescenta-se esse campo a cada nó na tabela de um PE do tipo "FPGADin", no arquivo `TimeCost.xml`. Se o FPGA for reconfigurável dinamicamente mas não o for parcialmente, então os tempos de reconfiguração de todos os nós são iguais ao tempo de reconfiguração do FPGA inteiro.

No código do programa de particionamento, é acrescentado um vetor de tempos de reconfiguração, indexado por PE, ao conjunto de propriedades da classe "nó". Esse vetor é sempre inicializado com o valor '0' para todos os nós e todos os PE's, simplificando o código do particionamento ao permitir que o tempo de reconfiguração possa ser sempre adicionado no agendamento, mas sem ter efeito para processadores não-dinamicamente reconfiguráveis.

O passo seguinte consiste em adaptar-se o algoritmo BIL para tratar o tempo de reconfiguração de um FPGADin. Não se pode saber *a priori* se a reconfiguração dinâmica será feita para um sistema, e além disso é preciso determinar-se quando ela será feita, isto é, entre quais nós da aplicação. Para definir isso, o agendador BIL precisa conhecer os consumos de recursos do FPGA por cada nó, acompanhando as quantidades de cada tipo de recurso ainda disponíveis no FPGA de forma precisa; dessa forma, ele pode optar pela utilização ou não da reconfiguração dinâmica e, em caso positivo, calcular o instante no tempo quando agendá-la. Quando um FPGADin for candidato para ser usado no sistema, então antes de cada mapeamento de um novo nó n_j nele (isto é, um nó de um tipo ainda não presente na configuração corrente do FPGADin), o agendador BIL deve primeiro verificar o respeito às restrições de recursos desse FPGA:

1. Se o tamanho do novo nó n_j, em termos de quantidades de recursos ocupadas, for menor que as quantidades de recursos correntemente disponíveis no FPGA, então o algoritmo mapeia n_j na configuração inicial do FPGA, e portanto não há necessidade

para uma reconfiguração dinâmica. As quantidades de recursos disponíveis no FPGA são atualizadas pela subtração dos recursos ocupados pelo nó n_j.

2. Se o tamanho de n_j demandar mais recursos do que estão disponíveis no FPGA, então é necessário agendar a reconfiguração do FPGA, pois este não tem nenhum nó já configurado que seja do mesmo tipo de n_j, e os nós já alocados nele não deixam recursos livres suficientes para que n_j seja também alocado.

Se a situação "2" ocorrer, o algoritmo encontra o instante quando agendar a reconfiguração do FPGA, que é o instante em que este se torna disponível (seus nós mapeados terminam de executar). A partir desse instante são agendados dois novos tipos de nós criados na versão 3.0 do algoritmo: um "nó de salvamento de contexto" e um "nó de reconfiguração". O primeiro corresponde à tarefa de salvar em memória SRAM os resultados intermediários dos nós de aplicação que serão desconfigurados, para que os nós da próxima página de configuração possam dar continuidade ao processamento da aplicação. O nó de reconfiguração, por sua vez, recebe um valor de tempo de execução igual ao tempo de reconfiguração de n_j (que nesta versão é o próprio tempo de reconfiguração do FPGA, como já explicado). Esses dois nós são agendados em seqüência no FPGA a partir do instante de início da reconfiguração. Para fins de avaliação da técnica, o valor utilizado neste trabalho para a soma dos tempos de salvamento de contexto e de reconfiguração é de 1 s (um segundo). O consumo de potência de uma reconfiguração dinâmica também é considerado, sendo arbitrado em 500 mW. Esses valores são adicionados respectivamente ao tempo de execução e ao consumo de potência totais do sistema, fornecidos pelo programa de particionamento no arquivo de resultados `costs.xml`.

Depois de agendada a reconfiguração, os recursos (tipos de nós) que existiam antes no FPGA são marcados como "não configurados" e as quantidades de recursos disponíveis são reinicializadas a seus valores totais. Então o algoritmo continua a agendar os nós subseqüentes da aplicação, que vão ler da memória os resultados intermediários calculados pelos nós da configuração anterior. O resultado final (aplicação particionada e agendada) contém o nó de reconfiguração, com a duração total do agendamento sendo acrescida do tempo da reconfiguração, e a potência do sistema acrescida do consumo de potência da reconfiguração. O custo monetário do sistema é potencialmente inferior, pois como o *hardware* é multiplexado no tempo, então menos LE's são empregados para implementar a aplicação, apesar de que o FPGA dinamicamente reconfigurável tem um preço maior por unidade.

Múltiplas reconfigurações, na implementação de um algoritmo, **podem** acarretar melhor desempenho que uma versão configurada uma vez somente, porque a quantidade de cálculo feita em *hardware* é muito maior do que a área disponível permitiria. A característica essencial para justificar o uso da reconfigurabilidade dinâmica é que alguma

quantidade de estados mude com uma freqüência alta o suficiente para poder tirar vantagem da reconfiguração do *hardware*, mas baixa o suficiente para mascarar o tempo dessa reconfiguração. A eficiência dos sistemas RTR depende da quantidade de tempo que a plataforma gasta fazendo cálculos úteis; à medida em que a freqüência das reconfigurações aumenta, o tempo de processamento pode acabar sendo sobrepujado pelo tempo de reconfiguração, anulando a vantagem da reconfiguração dinâmica.

3.3 Múltiplas alternativas de implementação

A versão 4.0 do algoritmo de particionamento capacita-o para lidar com múltiplas alternativas de implementação definidas pelo projetista do sistema para um mesmo nó em um mesmo PE, de *hardware* ou de *software*, realizando assim a terceira otimização proposta neste trabalho. Especialmente em *hardware*, observa-se que uma mesma funcionalidade pode ser implementada de diversas maneiras, correspondendo por exemplo a diferentes arquiteturas para um mesmo nó modeladas em HDL, cada uma representando um compromisso diferente entre desempenho, área e potência, mas todas realizando a mesma tarefa. Para o algoritmo de particionamento, isso significa que instâncias diferentes do mesmo tipo de nó (com a mesma funcionalidade) possam ter valores de custo e desempenho diferentes. Elas precisam ser incluídas no arquivo `TimeCost.xml`, na tabela do processador onde as alternativas de implementação são possíveis para o nó em questão.

Demonstra-se a seguir o suporte a múltiplas alternativas de implementação em *hardware*, que constituem-se no caso mais comum. Tome-se como exemplo a operação de multiplicação, feita em *software* sempre com o mesmo operador, mas que em *hardware* pode ser implementada de várias formas diferentes. Todos os nós de multiplicação precisam ter o mesmo nome de tipo, para indicar que trata-se da mesma operação. O arquivo `.xml` do grafo da aplicação não sofre qualquer alteração, com os nomes de instância sendo declarados de forma imparcial, como por exemplo `mulIX`. Nas tabelas de nós para PE's de *software* do arquivo `TimeCost.xml`, todas as instâncias de nós de multiplicação mantém os mesmos nomes de instância do grafo, porque não há outras alternativas de implementação neste caso.

As alternativas de implementação no *hardware* são escolhidas por instância de nó e de forma independente, ou seja, um nó de multiplicação pode ser implementado por uma alternativa, ao mesmo tempo em que outro nó de multiplicação é implementado por outra alternativa. A mudança é feita nas tabelas de nós para os PE's de *hardware*, onde as alternativas de implementação **para cada instância** dos nós de multiplicação `mulIX` devem ser declaradas em seqüência, usando o mesmo nome da instância com o acréscimo

apenas de um segundo índice, como: `mulIO_0`, `mulIO_1`, ..., `mulIO_n`. O algoritmo de particionamento lê os valores de tempo de execução, área, recursos ocupados, potência e tempo de reconfiguração das alternativas de implementação, e preenche as matrizes correspondentes de um mesmo objeto nó, já indexadas por PE '`p`' e por implementação '`i`'. Para cada instância do nó no grafo da aplicação, o algoritmo de particionamento analisa cada uma das alternativas de implementação disponíveis para decidir se essa instância deve ser mapeada em *hardware*, e escolher ao mesmo tempo a implementação mais conveniente. O critério para essa escolha é dado pela função de custo utilizada, conforme explicado na Seção 3.1.2.

Para que a escolha da alternativa de implementação funcione, é preciso que as diferentes alternativas sejam criadas com a mesma interface, para que uma possa ser substituída pela outra no grafo particionado. A diferença entre elas precisa ser limitada somente à arquitetura.

Para implementar-se o suporte a múltiplas alternativas de implementação, é necessário adaptar-se todos os métodos de agendamento e particionamento para trabalharem com matrizes ao invés de vetores, o que ainda não era feito pela versão 1.0. Por exemplo, as declarações `BIL[p]` e `BIM[p]` têm que ser modificadas para `BIL[p][i]` e `BIM[p][i]`, o que acarreta modificações em todos os métodos que calculam ou utilizam os valores de BIL e BIM de um nó. Onde houver laços que percorram os processadores candidatos, é necessário acrescentar-se laços internos para percorrer também as alternativas de implementação possíveis de cada nó em cada processador.

3.4 Complexidade computacional do algoritmo otimizado

Como mostrado na Seção 2.4.5, a complexidade computacional do algoritmo de particionamento original (versão 1.0) é $O(S.P^2.N_t)$ [16], onde S é a complexidade do agendador BIL, P é o número total de PE's candidatos e N_t é o número de tarefas. Por sua vez, $S = O(N^2.p.\log(p))$, onde N é o número de nós do grafo de uma tarefa e p é o número de PE's considerados para o agendamento [18].

As versões 2.0 e 3.0 acrescentam novos membros de dados aos PE's e nós, novos testes de condições e os cálculos de custos e potências de PE's, nós e de soluções heterogêneas, listados nas equações 3.1 a 3.17. Entretanto, elas não acrescentam novas iterações em função das quantidades de elementos a tratar, por isso a complexidade computacional permanece a mesma.

A versão 4.0 acrescenta uma nova dimensão ao particionamento, a alternativa de

implementação, que representa mais uma escolha de particionamento e agendamento; entretanto, múltiplas alternativas somente estão disponíveis para alguns nós e em alguns PE's, e em quantidades variáveis para cada par (nó, PE), dificultando a dedução de uma expressão geral de complexidade. O limite máximo ocorre no caso em que todos os `N` nós do grafo possuem `I` alternativas de implementação cada um em todos os PE's, das quais `i` alternativas nos PE's candidatos, quando então a complexidade `S` do agendador BIL torna-se $S = O(N^2.p.i.\log(p.i))$, e a complexidade do particionamento torna-se $O(S.(P.I)^2.N_t)$.

3.5 Conclusão

Este capítulo descreveu em detalhes as três versões otimizadas do algoritmo de particionamento, realizadas sucessivamente a partir da versão 1.0, onde cada nova versão incorpora os aprimoramentos realizados nas versões anteriores e acrescenta-lhes uma nova otimização. A versão 2.0 do algoritmo passa a trabalhar com FPGA's e a diferenciá-los dos ASIC's, por meio do uso de parâmetros de custo específicos a cada um, o que não era feito no modelamento da arquitetura-alvo pela versão 1.0. A versão 2.0 também considera detalhadamente os custos associados ao consumo de recursos de um FPGA, os consumos de potência por nó das aplicações e as restrições do sistema em termos de desempenho, custo e potência simultaneamente. São incorporadas ao algoritmo funções de custo que integrem custos e restrições na mesma expressão.

Neste trabalho, a função de custo tem como objetivo a minimização do custo do sistema particionado final, e seu valor é empregado pelo controlador de alocação na seleção dos recursos a serem disponibilizados para o agendamento (ver Seção 2.4.3). Por sua vez, o algoritmo de agendamento BIL procura minimizar a agenda (tempo de execução) com a arquitetura de mínimo custo que recebe do algoritmo de particionamento principal. O efeito resultante é que o processo de particionamento como um todo minimiza o custo da solução, para quase todos os conjuntos de valores das restrições, o que pode ser observado para as aplicações no próximo capítulo.

A versão 3.0 introduz o agendamento para sistemas com *hardware* dinamicamente reconfigurável, onde um nó de reconfiguração é agendado no sistema quando os recursos do FPGA esgotam-se, simulando o atraso e a potência de uma verdadeira reconfiguração de toda a matriz do FPGA. Finalmente, a versão 4.0 passa a aceitar múltiplas alternativas de implementação para quaisquer nós de uma aplicação.

O próximo capítulo descreve a execução do particionamento com aplicações reais. Para os testes, são utilizadas duas aplicações de PDS, com o fim de se poder comparar as dife-

rentes versões do algoritmo de particionamento. São explicados todos os aspectos práticos da criação dos arquivos de entrada para o programa, com maior atenção à obtenção dos valores de desempenho, custo e potência de cada nó da aplicação, para o preenchimento da tabela de perfis "nó-PE" no arquivo `TimeCost.xml`. Finalmente, apresentam-se os resultados de particionamento das aplicações.

Capítulo 4

Exemplos de Particionamento de Aplicações

4.1 Introdução

Neste capítulo, utilizam-se as quatro versões do algoritmo de particionamento explicadas nos Capítulos 2 e 3, com as diferentes otimizações propostas, para obter soluções heterogêneas (*HW/SW*) de duas aplicações de processamento de sinais: um receptor de rádio digital *Rake* para sistemas CDMA e um decodificador de canal LDPC.

Para realizar-se o particionamento de uma aplicação, esta tem que ser dividida em nós, com a granularidade desejada, e então escrita como um grafo desses nós na linguagem XML, constituindo assim o primeiro arquivo de entrada para o algoritmo de particionamento. Cada um desses nós tem que ser avaliado separadamente nos processadores das arquiteturas-alvo que forem consideradas para implementar a aplicação particionada final (no caso, apenas a placa Nios II / Stratix II). Isso significa que, para cada nó, precisa-se medir os valores de parâmetros de desempenho (tempos de execução), custo (consumo de recursos de memória e de *hardware*) e consumo de potência em cada processador candidato, para o preenchimento da tabela de perfis "nó-PE" do arquivo de entrada `TimeCost.xml`.

O programa particionador é executado para o sistema, produzindo como resultado um arquivo de agendamento `sched.xml`, onde os nós da(s) tarefa(s) da(s) aplicação(ões) estão particionados e agendados nos PE's escolhidos para compor o sistema, e um arquivo de custos `costs.xml`, contendo os custos dos nós, das partições de *software* e de *hardware* e da solução final. Em seguida, as implementações heterogêneas de cada aplicação são avaliadas globalmente e comparadas com as implementações inteiramente em *software* e em *hardware*, para demonstrar as vantagens da utilização do algoritmo de particionamento.

A Seção 4.2 explica a arquitetura da placa Nios II / Stratix II, utilizada neste trabalho para a caracterização dos nós das aplicações. A Seção 4.3 fornece orientações gerais para a criação do grafo de uma aplicação visando o particionamento, e em seguida descreve como utilizar a placa e as ferramentas de desenvolvimento Nios II IDE e Quartus II da Altera para a caracterização, em *software* e *hardware*, de cada nó/função de uma aplicação. Os métodos adotados são generalizáveis para qualquer outra aplicação, nessa e em outras plataformas similares da Altera. A Seção 4.4 apresenta informações gerais e independentes da aplicação sobre as soluções heterogêneas geradas pelas quatro versões do algoritmo de particionamento. As seções 4.5 e 4.6 explicam a criação dos grafos, a caracterização dos nós e os resultados do particionamento das aplicações *Rake* e LDPC respectivamente, pelas diversas versões do algoritmo de particionamento. As conclusões são dadas na Seção 4.7.

4.2 Plataforma heterogênea para caracterização

A plataforma heterogênea considerada para o particionamento é a placa Nios II / Stratix II^{TM} da Altera, na qual aplicações particionadas podem ser implementadas, executadas e avaliadas. Entretanto, neste trabalho utiliza-se essa plataforma somente para caracterizar os nós das aplicações a serem particionadas, deixando-se as implementações das mesmas para um trabalho posterior. Essa placa é composta principalmente de um FPGA Stratix II EP2S60F672C5 da Altera, no qual pode ser implementado um núcleo processador de *software* Nios II. Nios II é um processador embarcado virtual (configurado no FPGA) com arquitetura de 32 *bits*, que pode ser programado nas linguagens C/C++ por meio de seu ambiente de desenvolvimento Nios II IDE^{TM}. Existem três versões desse processador: rápida (Nios II/f), econômica (Nios II/e), e padrão (Nios II/s). Neste trabalho, utiliza-se somente a versão padrão.

A arquitetura interna dos FPGA's da família Stratix II possui, como elemento lógico básico para a implementação das funções do usuário, o bloco LUT (tabela de consulta adaptativa), equivalente ao LE de outras arquiteturas de FPGA's. Assim, o tamanho de um nó implementado em um FPGA Stratix II é medido em termos do número de LUT's que ele ocupa, pois o bloco LUT é a célula usada pela ferramenta de desenvolvimento Quartus II^{TM} da Altera para a síntese lógica dos projetos do usuário. A organização da arquitetura dos FPGA's Stratix II é explicada detalhadamente em [55].

A tabela 4.1 lista as quantidades totais de recursos internos do FPGA EP2S60F672C5, que estão disponíveis para a implementação de funções.

Tabela 4.1: Recursos internos do FPGA EP2S60F672C5.

LUT's	*Bits* de RAM	Blocos de DSP[1]	PLL's	Pinos de E/S para o usuário
48.352	2.544.192	36	12	493

[1] Correspondem a 288 multiplicadores 9 x 9 *bits*, ou 144 multiplicadores 18 x 18 *bits*, ou 36 multiplicadores 36 x 36 *bits*.

A implementação do processador Nios II/s no FPGA EP2S60, com um conjunto particular de opções de síntese do ambiente Quartus II, ocupa as quantidades de seus recursos internos listadas na tabela 4.2:

Tabela 4.2: Recursos internos do FPGA EP2S60F672C5 ocupados pelo processador Nios II/s.

LUT's	*Bits* de RAM	Multiplicadores 9 x 9 *bits*	PLL's	Pinos de E/S para o usuário
3.399	571.136	8	1	182

Enquanto o algoritmo de particionamento não mapear nenhum nó da aplicação para *software*, as restrições para a partição de *hardware* são as da tabela 4.1. Quando algum nó for mapeado para *software* durante o particionamento, as quantidades de recursos internos do FPGA EP2S60 que restarão disponíveis para a implementação dos nós de *hardware* do sistema passam a ser as diferenças entre os valores das tabelas 4.1 e 4.2. Essas novas restrições são listadas na tabela 4.3:

Tabela 4.3: Recursos internos do FPGA EP2S60F672C5 disponíveis para a partição de *hardware*.

LUT's	*Bits* de RAM	Multiplicadores 9 x 9 *bits*	PLL's	Pinos de E/S para o usuário
44.953	1.973.056	280	11	311

Os valores da tabela 4.1 são as restrições declaradas para o FPGA EP2S60 no arquivo `TimeCost.xml`, mas se for necessário utilizar o Nios II no sistema, as restrições de fato para o *hardware* são corrigidas para os valores da tabela 4.3 pelo algoritmo de particionamento.

Para este trabalho, arbitra-se o custo do FPGA Stratix II em 250.000 unidades monetárias, para ser igual ao custo total do ASIC da versão 1.0, e o custo de sua memória de reconfiguração é arbitrado em zero. Entretanto, apesar de o processador Nios II ser ele próprio um nó configurado no FPGA, que ocupa seus recursos internos, seu custo (do Nios II) para o particionamento não pode ser calculado como uma fração do custo do

FPGA, segundo a equação 3.6. Se assim fosse feito, o custo de implementar os nós das aplicações em *hardware* seria muito menor do que em *software* (o que é irreal), pois o tamanho do "nó" Nios II é muito maior que os dos nós das aplicações consideradas neste trabalho. Assim, todos os particionamentos resultariam inteiramente em *hardware*. A arquitetura da plataforma Nios II / Stratix II utilizada possui a propriedade singular de que o PE de *software* é implementado usando recursos do *hardware* dedicado, uma situação ainda não encontrada na literatura. Diante do exposto, arbitra-se o custo do processador Nios II/s em 100 unidades monetárias.

A placa Nios II / Stratix II dispõe ainda de 1 M × 32 *bits* de SRAM, 16 M × 32 *bits* de SDRAM e 16 *Mbytes* de memória *flash*. Os dois primeiros tipos de memória podem ser usados pelo processador Nios II como memória de uso geral, enquanto a memória *flash* é utilizada tanto para um armazenamento não-volátil pelo Nios II quanto para armazenar as configurações do FPGA. A freqüência de operação padrão da placa e do FPGA é de 50 MHz, valor utilizado para as simulações em *hardware* de todos nós das aplicações deste trabalho, feitas no Quartus II.

Para a partição de *software* no particionamento da aplicação *Rake*, considera-se a existência somente da memória SRAM de 1 M palavras de 32 bits, como a quantidade total de memória disponível para programa e dados. No caso da aplicação LDPC, declara-se a SDRAM de 16 M x 32 bits como a memória total disponível. Os custos de ambas as memórias são arbitrados em 16 unidades monetárias.

4.3 Construção do grafo e perfilamento dos nós de uma aplicação

Os algoritmos das duas aplicações consideradas neste trabalho foram escritos em linguagem C ou C++ de forma bastante modular, e então divididos em funções com uma granularidade fina ou média, a serem avaliadas enquanto nós para o particionamento. Para a construção do grafo representando o algoritmo de uma aplicação, as funções de processamento de dados e os principais blocos lógicos de instruções do seu código têm que ser associados a nós, observando-se seus sinais de entrada e saída. As funções mais elementares devem ser definidas como nós atômicos (indivisíveis), e as funções que as chamam como nós hierárquicos. É necessária ainda a criação de nós atômicos não-funcionais (**nós "teóricos"**) nos grafos das aplicações, que surgem naturalmente por causa da necessidade de compatibilizar os tipos dos dados trocados entre os nós de processamento, ou para modelar aqueles blocos de código de uma função hierárquica que não fazem um processamento de dados verdadeiro.

O primeiro caso acontece, por exemplo, quando uma função recebe o ponteiro de um vetor ou matriz, e passa para outra função um valor desse vetor ou matriz. O valor pode ter um tamanho em *bytes* diferente do tamanho do ponteiro, sendo incorreto simplesmente substituir um pelo outro no grafo. Então insere-se um nó teórico para converter um tipo de dado no outro, mas sem corresponder a um real processamento da aplicação. O segundo caso acontece porque somente os nós atômicos são particionáveis, e os nós hierárquicos (nós que contêm outros nós) existem apenas para organizar a hierarquia da aplicação e discriminar instâncias do mesmo tipo de nó chamadas a partir de funções diferentes[1]. Não se pode declarar os nós hierárquicos no arquivo `TimeCost.xml`, apenas os nós atômicos. Entretanto, os tempos de execução dos nós hierárquicos são geralmente maiores do que soma dos tempos de execução apenas dos seus nós **funcionais** (de processamento) internos. Essa diferença é devida aos nós teóricos existentes dentro da função hierárquica, que também contribuem ao tempo de execução dela, e se comunicam com os nós funcionais. Em qualquer dos casos, os nós teóricos constituem-se em novos nós particionáveis de uma aplicação, que também precisam ser declarados e caracterizados no arquivo `TimeCost.xml`.

Se não for possível medir-se os tempos de execução dos nós teóricos individualmente, pelo fato de eles não serem funcionais, então deve-se calcular esses tempos a partir dos tempos de execução dos nós funcionais, para "completar" o valor que falta para atingir-se o tempo de execução total medido para a função hierárquica à qual pertencem. Para efetuar esse cálculo, é necessário determinar-se primeiramente as equações de números de iteração para os nós internos do nó hierárquico, as quais dependem dos valores das variáveis de controle dos laços. Isso significa que os tempos de execução dos nós podem ser dependentes dos dados de entrada da aplicação.

Tanto em *software* quanto em *hardware* há trechos de código ou funções que executam tarefas específicas de *software* e de *hardware* respectivamente, e que no grafo da aplicação podem ser representados por um mesmo nó teórico. O resultado do particionamento dos nós teóricos deve ser interpretado com cautela, porque eles não têm sentido sozinhos, e geralmente somente podem ser implementados em um dos dois domínios (ou em *software* ou em *hardware*), mas não em ambos, conforme o caso. Os nós teóricos são nós de suporte e geralmente têm que acompanhar o mapeamento dos nós funcionais de processamento.

A avaliação de um nó de aplicação em um processador, seja de *software* ou de *hardware*, pode ser feita por meio de simulação ou de implementação no processador real. A simulação é preferível sempre que estiver disponível para os processadores em uso, por ser um método de avaliação mais rápido do que a implementação. Entretanto, para avaliar-se

[1] Um nó hierárquico corresponde na prática a uma função que chama outras funções.

a execução dos nós em *software*, a simulação só é válida se levar em conta o modelo de memória utilizado na arquitetura real, com os atrasos de acesso à memória envolvidos, pois caso contrário os resultados serão otimistas demais. Para avaliar-se os nós em *hardware*, deve ser feita uma simulação pós-síntese, que leva em conta os atrasos de propagação reais entre os nós, internamente aos PE's em *hardware*. Em ambos os casos, é muito importante estabelecer-se um método sistemático, reutilizável com qualquer aplicação, para fazer-se as medições do tempo de execução, do "custo" (expresso como o consumo de recursos do PE), e da potência consumida pelos nós, tanto em *software* quanto em *hardware*.

4.3.1 Perfilamento dos nós em *software*

Neste trabalho, a avaliação dos nós das aplicações em *software* é feita por meio de suas implementações no processador Nios II da placa Altera. O método de estimação dos tempos de execução dos nós emprega a função `alt_timestamp()` da linguagem C do Nios II [22], que retorna o número de ciclos de relógio transcorridos desde o início da execução do programa. Chamando-se essa função no início e no término do trecho de código (nó) cujo tempo de execução deseja-se conhecer, a diferença entre as leituras fornecerá o número de ciclos transcorridos naquele trecho. Como a freqüência de relógio do FPGA Stratix II (e do Nios II dentro dele) é de 50 MHz, encontram-se os tempos de execução dos nós multiplicando-se suas contagens de ciclos pelo período do relógio.

Quando os nós testados são muito simples, com tempos de execução muito pequenos, os valores retornados por `alt_timestamp()` perdem a precisão. Para resolver o problema, deve-se passar a chamar esses nós de 100 mil a 1 milhão de vezes dentro do programa, por meio de um laço `for`. Assim procedendo, o resultado fornecido estabiliza-se, bastando dividir-se o tempo obtido pelo número de iterações do laço para conhecer o tempo de execução do nó. Entretanto, a estimação dos tempos de execução dos blocos de código de uma aplicação pela função `alt_timestamp()` não tem uma boa repetibilidade, isto é, o tempo de execução obtido para um mesmo bloco de código, com os mesmos dados, varia de uma medição para outra. Nas medições feitas executando-se o decodificador LDPC várias vezes com o mesmo arquivo de entrada, observam-se variações de 0,2% até 6%, ficando geralmente em torno de 2%. Por causa dessa margem de erro, os valores de latência medidos por meio da função `alt_timestamp()` devem ser tomados como estimativas, podendo além disso ser dependentes dos dados.

Para medir-se os tamanhos ocupados na memória pelos nós da aplicação (na forma de seus códigos executáveis), primeiramente deve-se individualizá-los em arquivos-fonte C independentes, com a granularidade desejada. Se o nó for uma função, ela deve ser chamada por uma função `main()` simples que lhe passa seus parâmetros de entrada típicos,

e retorna para `main()` após seu término [2]. Dessa forma, pode-se compilar o arquivo-fonte em C de cada nó para o Nios II, por meio do ambiente Nios II IDE, e o tamanho em *bytes* de seu arquivo executável (com extensão `.elf`) é declarado como a medida da ocupação de memória pelo nó, no arquivo `TimeCost.xml`.

Não é possível medir-se o consumo de potência instantâneo somente do Nios II na placa Nios II / Stratix II, e conseqüentemente não se pode medir o consumo de potência dos nós em *software*. No arquivo `TimeCost.xml` é sempre utilizado o mesmo valor de potência em *software* para todos os nós, igual à potência estimada pelo Quartus II para o processador Nios II/s[3], que é de 62 mW. Este valor é usado com ambos os projetos desta tese.

4.3.2 Perfilamento dos nós em *hardware*

Para conhecer-se os tempos de execução dos nós em *hardware*, é necessário fazer-se a síntese seguida de simulação dos mesmos na ferramenta Quartus II, para o modelo de FPGA (EP2S60F672C5) presente na placa Nios II / Stratix II. O Quartus II fornece também os tipos e quantidades de recursos elementares do FPGA a serem utilizados por cada nó, e o consumo de potência de cada nó no FPGA.

A versão 1.0 do algoritmo de particionamento recebe e processa somente a área de *hardware* ocupada por um nó, tal como em um ASIC. Para a contabilização da "área equivalente" de um nó para o arquivo `TimeCost.xml` da versão 1.0, associa-se livremente um LUT a 1 mm^2: os números de LUT's ocupados pelos nós passam a ser seus valores de área. No caso de funções que também ocupam multiplicadores dedicados do FPGA, a quantidade equivalente de LUT's é calculada utilizando-se a relação de equivalência de 1 multiplicador de 9 x 9 *bits* para 14 LUT's (relação determinada por meio do Quartus II para o FPGA EP2S60F672C5). Nos casos em que a implementação de um nó no FPGA utiliza *bits* de memória, aplica-se a equivalência de 1 *bit* de RAM = 1,73 LUT's, determinada por meio da síntese desse nó no Quartus II com as opções de reconhecimento e utilização de blocos de RAM desativadas. Dessa forma, pode-se expressar os tamanhos dos nós por um valor único de "área".

Quanto aos valores de consumo de potência, eles não são considerados pela versão 1.0 do algoritmo de particionamento. Somente a partir da versão 2.0, proposta neste trabalho de tese, os consumos de recursos internos de um FPGA e os consumos de potência dos nós passam a ser considerados no particionamento.

O consumo de potência de todo projeto implementado em um FPGA Altera pode ser

[2] A existência de `main()` é necessária para a geração de um programa executável.

[3] Maiores detalhes sobre a estimação de potência são dados na Seção 4.3.2.

estimado por meio da ferramenta "*PowerPlay Power Analyzer*", existente no Quartus II [56]. Ela pode estimar os consumos de potência estática e dinâmica após a síntese, de todos os nós atômicos e hierárquicos de uma aplicação, sem a necessidade de simulação. Para a máxima precisão dos valores de potência fornecidos pela ferramenta, o simulador do Quartus II pode gerar um arquivo de "atividades de sinal" (arquivo `.saf`) contendo as taxas de chaveamento do sistema, para o conjunto de sinais de entrada e saída da simulação do projeto no FPGA. Esse arquivo deve ser passado ao *PowerPlay*. No relatório de potências gerado, deve-se observar a distribuição do consumo de potência dinâmica total da aplicação pela sua hierarquia. Não se analisa a potência de cada nó atômico individualmente porque ela depende fortemente dos dados de simulação, portanto de uma simulação a outra os valores não seriam proporcionais, com diferenças importantes entre eles. Simulando-se a aplicação completa com um mesmo conjunto de dados, obtém-se rapidamente o consumo dos blocos hierárquicos e em que proporção ele se distribui entre os seus nós internos.

O consumo de potência estática é um valor constante para um mesmo FPGA, independente da complexidade do projeto que estiver implementado nele. Por essa razão, esse valor é tratado como um custo fixo do FPGA, a ser especificado no seu cabeçalho no arquivo `TimeCost.xml`, a partir da versão 2.0 do particionamento. O consumo de potência estática do FPGA EP2S60F672C5, dado pelo *Power Analyzer*, é de 645 mW. A potência especificada para cada nó no arquivo `TimeCost.xml` é somente a sua potência dinâmica, que depende do circuito de cada nó e varia em função da freqüência de operação, servindo portanto como medida de custo.

Como os blocos hierárquicos em *hardware* não são iguais à simples "adição" dos nós atômicos de processamento, mas incluem também elementos de controle que só existem em *hardware* (e portanto não entram no particionamento), adota-se um cálculo mais realista para a quantidade de recursos do FPGA, o tempo de execução em *hardware* e o consumo de potência de um nó. Por exemplo, seja `A` um nó hierárquico formado por dois nós funcionais atômicos, $\mathtt{n}_1$ e $\mathtt{n}_2$. Se, após serem sintetizados para *hardware*, `A` apresenta 100 LUT's a mais que a soma de LUT's ocupados por $\mathtt{n}_1$ e $\mathtt{n}_2$ juntos, esses 100 LUT's de excesso em `A` correspondem à lógica de controle. Se pelo menos um dos nós atômicos for realmente mapeado para *hardware* pelo algoritmo de particionamento, a lógica de controle terá que acompanhá-lo, como acontece na realidade.

4.4 Soluções heterogêneas para uma aplicação

Quando o particionador propõe uma solução heterogênea, ele insere nós de interfaceamento `Send` e `Receive` extras, para cada sinal a ser comunicado entre os domínios de *hardware* e *software*. A implementação desses nós fica a cargo do usuário: para realmente implementar a solução heterogênea, o usuário tem que modificar as funções em C e os módulos em VHDL originais que precisarem se comunicar, acrescentando-lhes os comandos ou estruturas para escrita e leitura de variáveis entre o processador de *software* e o *hardware* dedicado, que dependem dos PE's em particular, da topologia de interconexão e do protocolo de comunicação escolhidos.

Os tempos de execução e os custos de implementação (complexidades) em *software* e *hardware* dos nós de interfaceamento são considerados no particionamento, aumentando o tempo de execução e o custo totais da solução heterogênea final. O tempo para a comunicação de 1 *byte* entre o *software* (Nios II) e o circuito de *hardware* dedicado no restante do FPGA é arbitrado em 1 ns, nos dois sentidos. Esse tempo é multiplicado, em cada enlace, pelo tamanho da palavra a comunicar e pela taxa de comunicação (número de palavras a comunicar em cada iteração do grafo), informados no grafo da aplicação. Os tamanhos em *software* e *hardware* dos nós de interfaceamento são feitos arbitrariamente iguais aos tamanhos dos nós `Send` e `Receive` (de entrada e saída) do grafo. Todos os valores são declarados no arquivo `TimeCost.xml`.

Na versão 1.0 do algoritmo de particionamento, a única restrição para uma tarefa é o seu período, cujo respeito é garantido por meio do controle da utilização (Seção 2.4.3). Valores de custo não são considerados durante o particionamento, somente valores de desempenho. Como explicado no Capítulo 2, a versão 1.0 somente trabalha com ASIC's e não com FPGA's, por isso tanto a restrição para o PE de *hardware* quanto o tamanho da partição resultante são expressos como áreas em mm^2.

Na versão 2.0 do algoritmo de particionamento, as restrições para um modo são o seu tempo de execução total, seu custo monetário e seu consumo de potência. Porém, em sistemas mono-tarefa, como o receptor *Rake* e o decodificador LDPC, a restrição de desempenho para o particionamento continua sendo a restrição de período da tarefa, que substitui a restrição de tempo de execução do modo.

A versão 3.0 do algoritmo de particionamento presume a situação usual em que o processador de *software* é fisicamente independente do processador em *hardware* dedicado. No caso específico da plataforma Nios II / Stratix II disponível para este trabalho, é praticamente inviável trabalhar-se com um sistema heterogêneo utilizando reconfiguração dinâmica, pois o PE de *software* Nios II é ele próprio uma configuração feita no FPGA, ou seja, até mesmo a execução da partição de *software* pararia durante uma reconfiguração,

até que o Nios II fosse carregado novamente na matriz do FPGA para poder retomá-la, resultando em um custo temporal muito elevado.

A versão 4.0 somente é explorada para o exemplo do *Rake*, pois somente ele dispõe de duas variantes de nós para o mesmo fim, no caso nós de multiplicação. Como a função de custo introduzida na versão 2.0 procura minimizar o custo monetário do sistema heterogêneo final, esse é o critério para a escolha da alternativa de implementação a utilizar.

4.5 Receptor *Rake*

A primeira aplicação usada para testar o algoritmo de particionamento é um receptor de rádio *Rake* de arquitetura iterativa. Ele é baseado em uma implementação em *pipeline* dos diferentes processos da demodulação do canal, de modo a maximizar o número de iterações possíveis [57]. Dispõe-se inicialmente do algoritmo do receptor *Rake*, implementado em linguagem C, e de sua descrição em VHDL para implementação em *hardware*. A Seção 4.5.3 apresenta as soluções heterogêneas do *Rake* propostas pelas diversas versões do algoritmo de particionamento. Os valores de desempenho e custo dessas soluções são comparados com os das implementações em *software* e em *hardware* do receptor.

4.5.1 Grafo de entrada e determinação dos parâmetros de *software*

A figura 4.1 mostra a representação gráfica do grafo da aplicação *Rake*, que possui 21 nós, para um conjunto de valores típicos das variáveis de iteração: `L` = 3, `sf` = 4 e `M` = 5, o que significa um receptor num ambiente com três percursos, com um fator de espalhamento igual a quatro e para cinco símbolos recebidos. Esse conjunto de valores é utilizado nas implementações em *software* e *hardware* do *Rake* neste trabalho. São indicados no grafo os nomes de instância dos nós, os nomes das variáveis comunicadas em cada arco, os números totais de repetições de execução de cada nó[4] (definidos pelos valores de `M`, `sf` e `L`) e a taxa de comunicação em cada arco, conforme o modelo fracionado definido em [54,58]. Os nós dos tipos `Receive` e `Send` são os nós de interface da aplicação com seu exterior. Os nós teóricos inseridos são `rake_initI0`, `temp0I0`, `temp1I0`, `temp2I0` e `temp3I0`. No nó hierárquico `MRC_additionneur`, por exemplo, os nós de processamento são `addI1` e `mul_cI1`, e `temp3I0` é um nó teórico. A função `trivial_multiplier()` corresponde aos nós atômicos `temp1I0`, `temp2I0`, `mul_cI0` e `integ_addI0`.

[4] quando diferente de '1'

Embora o grafo da figura 4.1 tenha sido criado a partir do código de *software* do *Rake*, ele é igualmente válido para a descrição em VHDL da aplicação.

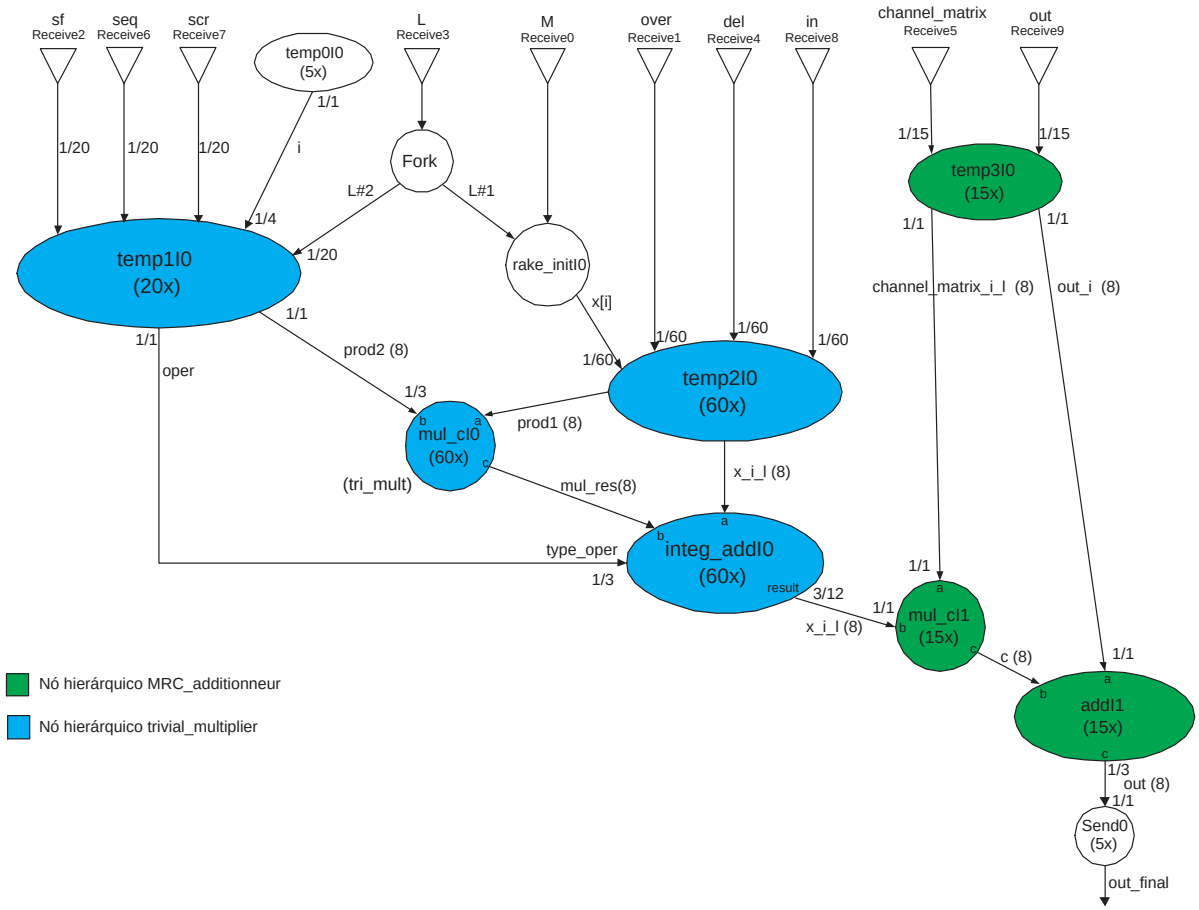


Figura 4.1: Grafo do algoritmo do receptor *Rake*.

Os valores dos tempos de execução aproximados para cada um dos nós do *Rake*, no processador Nios II, bem como seus tamanhos em memória, estão listados na tabela 4.4.

Para estimar-se o tamanho total de uma partição de *software* na memória, **não** se pode simplesmente somar os tamanhos em *bytes* dos arquivos executáveis de cada nó que for mapeado para *software*. A razão para isto é que todo programa executável possui sua própria função `main()`, que representa a principal parcela do seu tamanho[5], enquanto a partição de *software* final possuirá somente uma única função `main()`. Observando-se os tamanhos dos arquivos executáveis de todos os nós do *Rake*, pode-se constatar que eles são muito próximos entre si e dominados pelo tamanho de `main()`, pois são nós de granularidade fina. Na tabela 4.4, a função mais simples de todas (`add`) ocupa 181.889 *bytes*, enquanto a aplicação *Rake* completa ocupa 190.948 *bytes*, ou seja, para um aumento muito grande da complexidade, o aumento do tamanho do arquivo é de apenas 5%. Diante

[5]Independentemente da complexidade desta função `main()`

Tabela 4.4: Valores de desempenho e custo dos nós do receptor *Rake* em *software*.

Nó	Tempo de execução (ns)	Tamanho em memória (*bytes*)
MRC_additionneur: add	4.097	181.889
MRC_additionneur: mul_c, trivial_multiplier: mul_c	5.229	181.927
MRC_additionneur: temp3	18.530	181.320
trivial_multiplier: integ_additionneur	7.528	182.528
trivial_multiplier: temp1	24.774	181.344
trivial_multiplier: temp2	21.240	181.241
Rake: Receive	2.000	181.107
Rake: rake_init	132.521	181.693
Rake: temp0	4.972	180.917
Rake: MRC_additionneur	52.132	182.335
Rake: trivial_multiplier	267.201	183.460
Rake: Send	2.000	181.107
Rake	3.110.729	190.948

disso, modela-se a contribuição da função `main()` ao tamanho de um programa por meio do "código elementar" abaixo, no qual ela apenas chama uma função vazia e depois encerra o programa, sem um real processamento de dados.

```
void test() { }

int main()
{
    test();

    return (0);
}
```

O tamanho do programa executável gerado para o código elementar acima, $\mathtt{as}_{elem}$, depende do processador de *software*, dos arquivos de cabeçalho incluídos e do conjunto de opções do compilador C utilizado. Para o processador Nios II e a aplicação *Rake*, o tamanho do executável elementar é de 180.806 *bytes*, determinado por compilação no ambiente Nios II IDE. Esse tamanho passa a ser lido da tabela do PE de *software* no arquivo

`TimeCost.xml`, em todas as versões do algoritmo de particionamento. Nesse arquivo de entrada, continua-se a escrever os tamanhos reais `as` dos nós atômicos executáveis, em *bytes*. O tamanho incremental de um nó de *software* qualquer, $\texttt{as}_{incr}$, é calculado pelo algoritmo de particionamento como a diferença entre o tamanho real `as` do nó e o tamanho do executável elementar:

$$as_{incr} = as - as_{elem} \tag{4.1}$$

Uma maneira mais correta para o algoritmo calcular o tamanho em memória da partição de *software* é tomar como ponto de partida o tamanho do programa executável elementar e adicionar-lhe apenas os tamanhos incrementais dos nós que compõem a partição de *software*. Dessa forma, a contribuição da função `main()` para o tamanho da partição de *software* é levada em conta somente uma vez.

4.5.2 Determinação dos parâmetros de *hardware*

Dispõe-se também do receptor *Rake* descrito inteiramente como uma conexão de componentes em VHDL, com uma hierarquia similar à utilizada com as funções do código em C. Alguns desses componentes em VHDL correspondem diretamente a uma função em C da implementação em *software*. Os nós de processamento do *Rake* são sintetizados e simulados individualmente no ambiente Quartus II, e a tabela 4.5 mostra seus valores de implementação no FPGA, para $\texttt{L} = 3$, $\texttt{sf} = 4$ e $\texttt{M} = 5$. Os nós `tempX` e `rake_init` representam funções de controle e sincronização próprias da arquitetura em *pipeline* do *Rake*.

Os tempos de execução para o receptor *Rake* em *software* e em *hardware*, nas tabelas 4.4 e 4.5, consideram os tempos para a leitura dos valores de entrada e para a escrita dos valores de saída da aplicação. Esses tempos de comunicação são representados, no grafo da figura 4.1, pelos nós de tipos `Receive` e `Send`, que são os nós de interface da aplicação com seu exterior.

4.5.3 Resultados do particionamento

Esta seção apresenta as soluções heterogêneas fornecidas pelas diferentes versões do algoritmo de particionamento para o receptor *Rake*, com os valores de desempenho e custo das tabelas 4.4 e 4.5 preenchidos no arquivo `TimeCost.xml`, e para diversos conjuntos de valores das restrições da aplicação preenchidos no arquivo `mode.xml`.

Versão 1.0

Para o teste da versão 1.0, a restrição de área máxima estipulada para a aplicação *Rake* é de 500 mm^2 e o custo por unidade de área é arbitrado em 500 $/$mm^2$ (valores típicos), resultando em uma restrição de custo máximo de 250.000 $. A tabela 4.6 (página 106) mostra as propostas de implementações heterogêneas do *Rake* na placa Nios II / Stratix II feitas pela versão 1.0 do algoritmo de particionamento, obtidas para diversos

Tabela 4.5: Valores de desempenho e custo dos nós do receptor *Rake* em *hardware*.

Nó	Latência (ns)	Recursos utilizados	Potência (mW)
`MRC_additionneur: add`	8,8	56 LUT's, 171 pinos	1,59
`MRC_additionneur: mul_c`	20	106 LUT's, 8 multiplicadores 9 x 9 *bits*, 116 pinos	4,13
`MRC_additionneur: temp3`	7	43 LUT's, 76 pinos	1,15
`trivial_multiplier: temp1`	8	25 LUT's, 75 pinos	6,79
`trivial_multiplier: temp2`	4	24 LUT's, 73 pinos	5,1
`trivial_multiplier: tri_mult`	8,6	128 LUT's, 46 pinos	3,98
`trivial_multiplier: integ_additionneur`	8	36 LUT's, 95 pinos	4,34
`Rake: Receive`	7	10 LUT's, 2 pinos	0,55
`Rake: rake_init`	120	202 LUT's, 12 pinos	10,6
`Rake: temp0`	20	94 LUT's, 95 pinos	4,00
`Rake: MRC_additionneur`	20	205 LUT's, 8 multiplicadores 9 x 9 *bits*, 139 pinos	6,87
`Rake: trivial_multiplier`	60	213 LUT's, 65 pinos	20,21
`Rake: Send`	7	10 LUT's, 20 pinos	0,55
`Rake`	620	824 LUT's, 8 multiplicadores 9 x 9 *bits*, 134 pinos	47,73

valores da restrição de período do *Rake* no arquivo `mode.xml`. São mostrados os valores de desempenho e custo que caracterizam as várias soluções: o tempo de execução total, a quantidade de memória ocupada pela partição de *software* e o número de recursos internos do FPGA utilizados pela partição de *hardware*, expressos como uma "área equivalente". O custo monetário total inclui os preços do processador (100 $), da fração de memória utilizada e da área do FPGA, conforme as equações 3.11 e 3.12.

A tabela 4.6 mostra que uma restrição de período muito pequena pode não ser satisfeita mesmo com uma implementação inteiramente em *hardware* da aplicação. No exemplo do *Rake*, observa-se da tabela 4.5 que seu tempo de execução em *hardware* é de 620 ns, logo uma restrição de 500 ns não pode ser atendida. Nesse caso, o algoritmo fornece uma mensagem indicando a impossibilidade de agendar uma solução que obedeça a essa restrição de desempenho.

Para restrições de tempo até 5,531 μs, o algoritmo 1.0 fornece uma solução inteiramente em *hardware* com um tempo de execução de 639 ns. A versão 1.0 não verifica o respeito às restrições de custo, atendendo somente à restrição de tempo (o desempenho exigido). As soluções propostas violam as restrições de área de HW máxima (500 mm^2) e de custo máximo (250.000 $) para períodos menores que 190 μs aproximadamente. Essa falha é corrigida a partir da versão 2.0 do particionamento, com a incorporação de uma função de custo, o que constitui-se em uma das contribuições desta tese.

O gráfico da figura 4.2 mostra os valores de período de execução do receptor *Rake* heterogêneo, em função de suas restrições. Para todas as soluções propostas, os valores de período obtidos acompanham os valores de suas restrições correspondentes e são sempre inferiores a estes. Para efeito de comparação, o tempo de execução do *Rake* somente no *hardware* é de 639 ns, e o seu tempo de execução em *software* é de 3.121 μs, quase 5 mil vezes maior. Esse é o máximo ganho de desempenho (ou a máxima redução no tempo de execução) que pode ser obtido para o *Rake* com a migração de suas funções do *software* para o *hardware* dedicado pela versão 1.0.

O gráfico da figura 4.3 mostra o comportamento dos valores da área de *hardware* e do custo monetário das diversas soluções, em função dos valores de período das mesmas, onde observa-se que a área e o custo diminuem de forma aproximadamente exponencial com o aumento do período do *Rake*. As quantidades de recursos do FPGA ocupados por cada nó, determinadas por meio do ambiente Quartus II, são convertidas em valores de "área equivalente" para cada nó, conforme explicado na Seção 4.3.2, e são então fornecidas para o particionamento no arquivo `TimeCost.xml`. Conseqüentemente, os tamanhos das partições de *hardware* geradas pela versão 1.0 são expressos por valores de "área", resultantes da adição das áreas individuais dos nós que compõem essas partições. Esses valores de área são convertidos em custos monetários pela multiplicação com o custo por

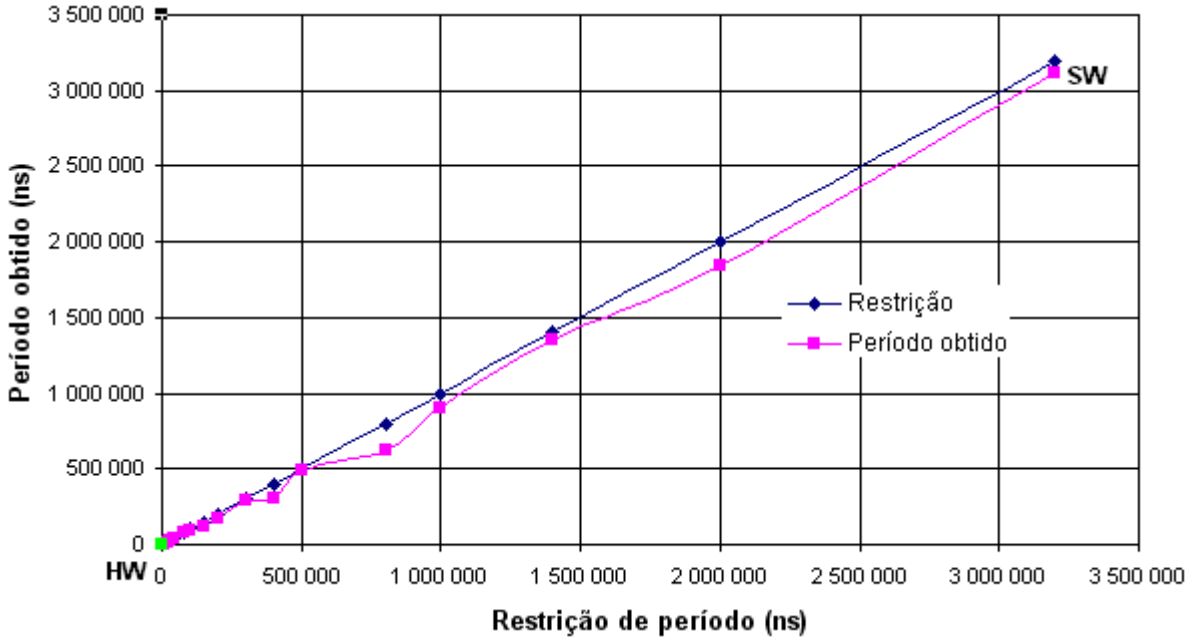


Figura 4.2: Valores de período do receptor *Rake* particionado com a versão 1.0.

unidade de área arbitrado para o *hardware* (500 \$/mm^2). Acrescentam-se a eles os custos das partições de *software*, obtendo-se assim os custos totais das soluções. O custo de implementação do *Rake* inteiramente em *hardware* é de 468.000 \$, e seu custo inteiramente em *software* é de 103 \$, uma diferença de 4.540 vezes.

Como mostrado na tabela 4.4, os tamanhos em *software* dos nós são muito próximos uns dos outros, pois são dominados pelo tamanho do código elementar, o que faz com que todas as partições de *software* também tenham tamanhos muito próximos: o tamanho estimado pelo particionador para a solução inteiramente em *software* (191.649 *bytes*) é apenas 6 % maior que o tamanho do código elementar (180.806 *bytes*). Essa diferença na realidade é ainda menor porque o tamanho estimado pelo particionador está em excesso em relação ao valor real (tabela 4.4) por 0,37 %, devido às otimizações feitas pelo compilador, que o particionador não leva em conta. A combinação de baixo custo (16 \$) e alta capacidade (1 MB) das memórias permite que as partições de *software* tenham custos extremamente baixos em comparação com qualquer partição de *hardware*: o custo da fração de memória utilizada fica entre 2 e 3 \$, que adicionado ao custo do processador (100 \$), resulta em custos das partições de *software* sempre próximos de 103 \$.

Observa-se ainda, da tabela 4.6, que algumas soluções heterogêneas consomem mais memória que a solução inteiramente em *software*; isso se deve à consideração do tamanho em memória dos nós de interfaceamento *SW-HW*.

Comparando-se a solução em *software* com a solução heterogênea de menor tempo de execução (aquela somente com o nó `tempOIO` em *SW*), a redução do tempo de execução

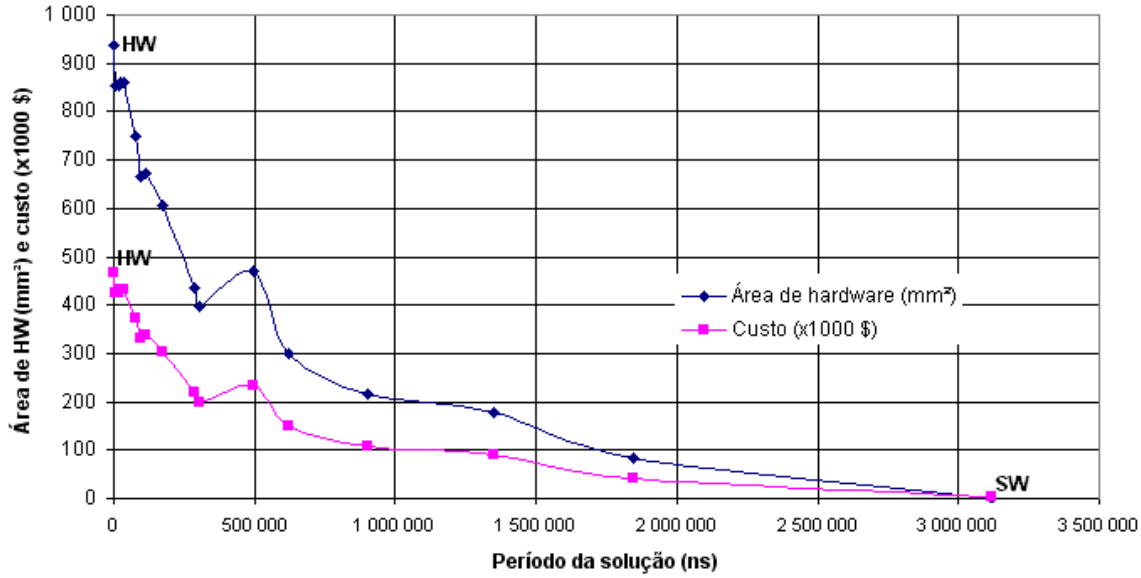


Figura 4.3: Valores de custo do receptor *Rake* particionado com a versão 1.0.

é de 564 vezes e o aumento de custo é de quase 4.140 vezes; a redução da utilização de memória é de apenas 1,06. Percebe-se que a influência da quantidade de memória utilizada sobre o custo da solução é desprezível, pois o custo é determinado principalmente pela área de *hardware*. Por outro lado, comparando-se a solução inteiramente em *hardware* do *Rake* com a solução heterogênea de menor custo (aquela somente com o nó `temp2I0` em *HW*), o aumento do tempo de execução é de 2.890 vezes para uma redução de custo de 11 vezes, a mesma redução da utilização de *hardware*.

Versão 2.0

A tabela 4.7 (página 107) mostra as propostas de implementações heterogêneas para o *Rake* na placa Nios II / Stratix II, feitas pela versão 2.0 do algoritmo de particionamento, para diversos valores das restrições de período de execução, custo e potência do *Rake* no arquivo `mode.xml`. Percebe-se que a versão 2.0 respeita simultaneamente todas as restrições impostas pelo usuário ao sistema particionado, em conseqüência do uso de funções de custo englobando custos e restrições em uma mesma expressão.

O tempo de execução do grafo sempre satisfaz a sua restrição, mas não é o mínimo valor possível. Isso pode ser observado quando relaxa-se a restrição de período, aceitando-se valores maiores: o algoritmo aproveita a "folga" no tempo de execução permitido para tentar alocar o mínimo possível de *hardware* dedicado, porque os pesos da função de custo o direcionam para a minimização do custo total, o qual é amplamente dominado pela quantidade de *hardware* utilizada. Assim, o algoritmo "descobre" que o custo pode ser minimizado mantendo mais funções em memória e ocupando menos recursos do FPGA,

o que significa **potencialmente** que um FPGA menor e mais barato do que o disponível poderia ser empregado, diminuindo assim o custo do sistema como um todo.

A comparação entre a terceira linha (`Tr` = 6 μs, `Cr` = 7.000 $ e `Pr` = 700 mW) e a quarta linha (`Tr` = 6 μs, `Cr` = 7.000 $ e `Pr` = 800 mW) de valores da tabela 4.7 mostra que a menor restrição de potência no primeiro caso foi decisiva para definir os resultados do particionamento. A permissão de uma maior potência na quarta linha dá margem para o algoritmo utilizar o PE de *software*, que acrescenta 62 mW ao orçamento de potência, para substituir um nó que estava em *hardware* e assim reduzir o custo. Isso confirma que a prioridade do algoritmo 2.0 é realmente a minimização do custo, pois os resultados da terceira linha também satisfariam as restrições da quarta linha.

O gráfico da figura 4.4 mostra o comportamento dos períodos obtidos para as várias soluções, juntamente com suas restrições correspondentes. Percebe-se que não há um esforço do algoritmo 2.0 em minimizar os períodos, mas apenas em mantê-los abaixo das restrições, tal como faz a versão 1.0.

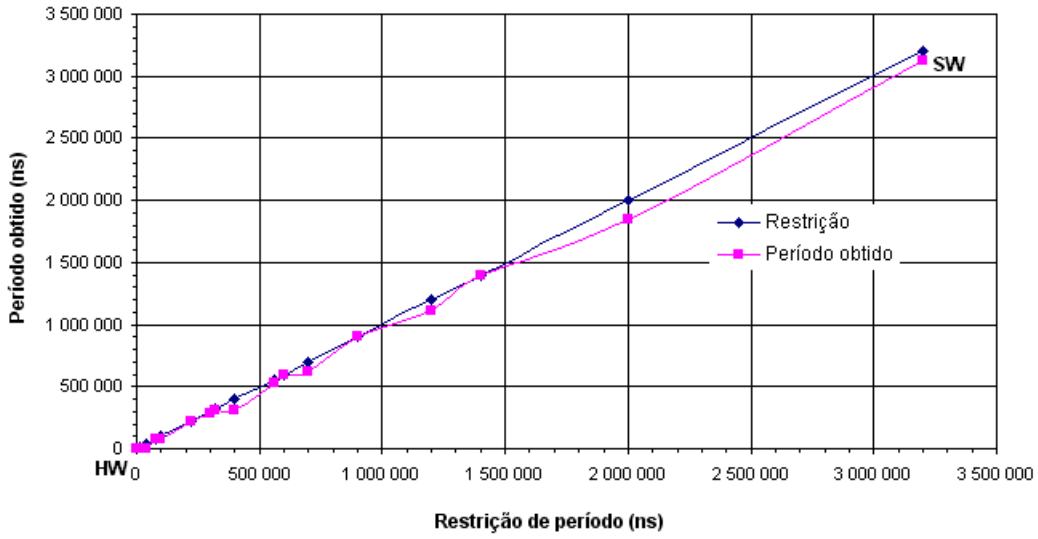


Figura 4.4: Valores de período do receptor *Rake* particionado com a versão 2.0.

O gráfico da figura 4.5 mostra os custos monetários das soluções, juntamente com suas restrições correspondentes, em função dos períodos dessas soluções. A redução de custo observada da solução inteiramente em *hardware* (2.931,4 $) para a solução inteiramente em *software* (103 $) é de 28,5 vezes. Na tabela 4.7, o uso de restrições de custo muito maiores que os custos esperados para o sistema mostra que o algoritmo de particionamento não explora a folga permitida ao custo para minimizar o tempo de execução.

Comparando-se a solução em *software* com a solução heterogênea de menor tempo de execução (aquela somente com o nó `tempOI0` em *SW*), a redução do tempo de execução é de 565 vezes e o aumento de custo é de 28,3 vezes. A redução de tempo de execução é a

mesma da versão 1.0, mas o aumento de custo é muito menor. Os valores de custo obtidos pela versão 2.0 são bastante inferiores aos fornecidos pela versão 1.0, em conseqüência da utilização de um FPGA no particionamento, e da consideração precisa da ocupação de seus recursos internos, enquanto a versão 1.0 utiliza um ASIC. Como foram declarados valores típicos de custo para ASIC's, os resultados das duas versões refletem a realidade de que o custo de implementação em ASIC é muito superior ao custo de implementação em FPGA[6]. Mas o mais importante no gráfico da figura 4.5 não são os valores numéricos e sim o comportamento dos custos das soluções em função dos valores das restrições.

Por outro lado, comparando-se a implementação inteiramente em *hardware* do *Rake* com a solução heterogênea de menor custo (aquela somente com o nó `temp2I0` em *HW*), o aumento do tempo de execução é de 2.894 vezes para uma redução de custo de 13 vezes, valores próximos aos encontrados para a versão 1.0, pois nos dois casos existe a mesma proporção de *hardware* e é o custo do *hardware* que domina.

O gráfico da figura 4.6 mostra o comportamento dos valores de potência das soluções geradas, que se concentram um pouco acima da soma das potências estáticas do FPGA (645 mW) e do processador (62 mW), sempre presentes em todas as soluções heterogêneas. Somente na solução inteiramente em *software* o valor de potência é muito diferente dos demais, e igual a 62 mW.

O gráfico da figura 4.7 mostra simultaneamente os valores de tempo de execução, custo e potência das soluções, normalizados por suas restrições correspondentes em cada solução. Neste gráfico, observa-se grosso modo que a razão C/C_r é a mais minimizada das

[6]Para pequenos volumes de produção.

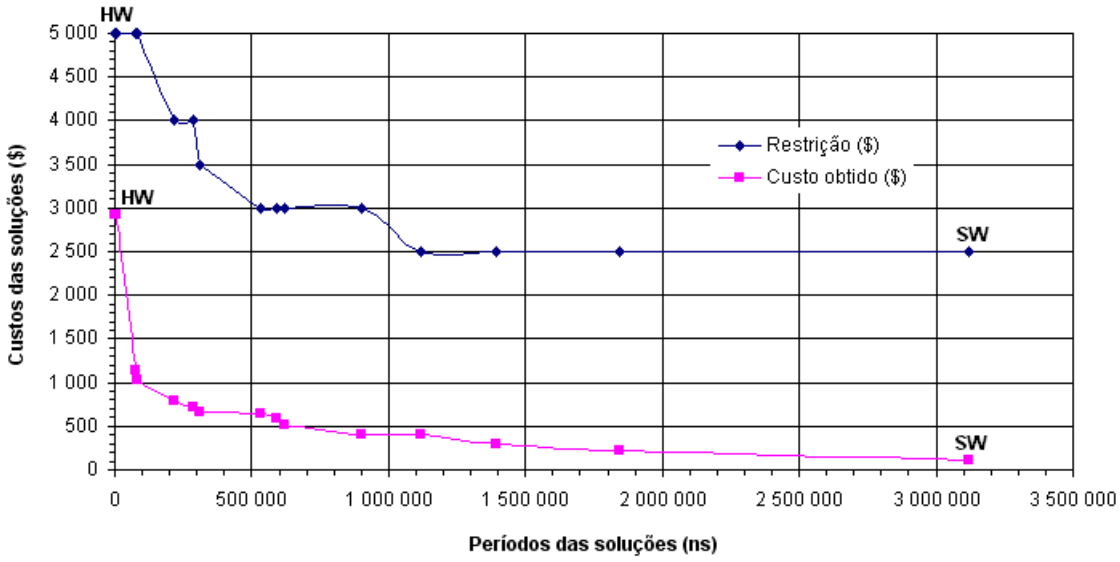


Figura 4.5: Valores de custo do receptor *Rake* particionado com a versão 2.0.

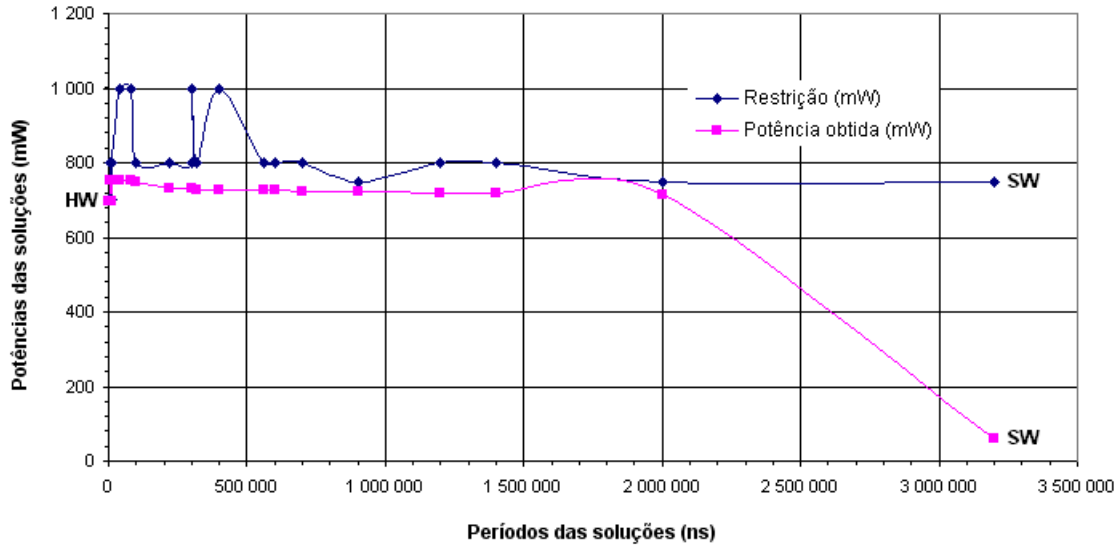


Figura 4.6: Valores de potência do receptor *Rake* particionado com a versão 2.0.

três, apesar de ainda apresentar oscilações, enquanto as curvas T/T_r e P/P_r permanecem geralmente próximas à linha limite custo/restrição = 1.

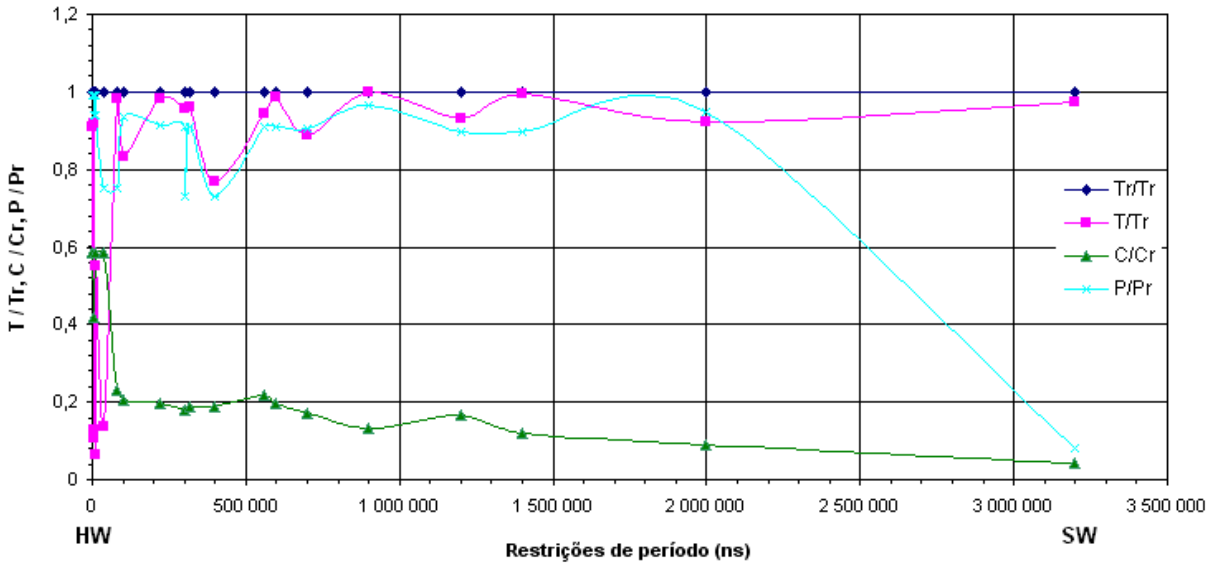


Figura 4.7: Valores normalizados de período, custo e potência para o *Rake* com a versão 2.0.

Comparando-se os resultados das versões 1.0 e 2.0 do algoritmo de particionamento, observa-se que os valores de período do *Rake* particionado são praticamente iguais para ambas, como era de se esperar, pois a diferença entre elas está no tratamento dos custos e restrições, e não no agendamento. Isso é mostrado na figura 4.8. Os valores dos custos, apesar da grande diferença numérica explicada anteriormente, possuem aproximadamente

o mesmo comportamento de decaimento exponencial; entretanto, a curva de custos da versão 2.0 é mais estável que a da versão 1.0, o que pode ser visto na figura 4.9.

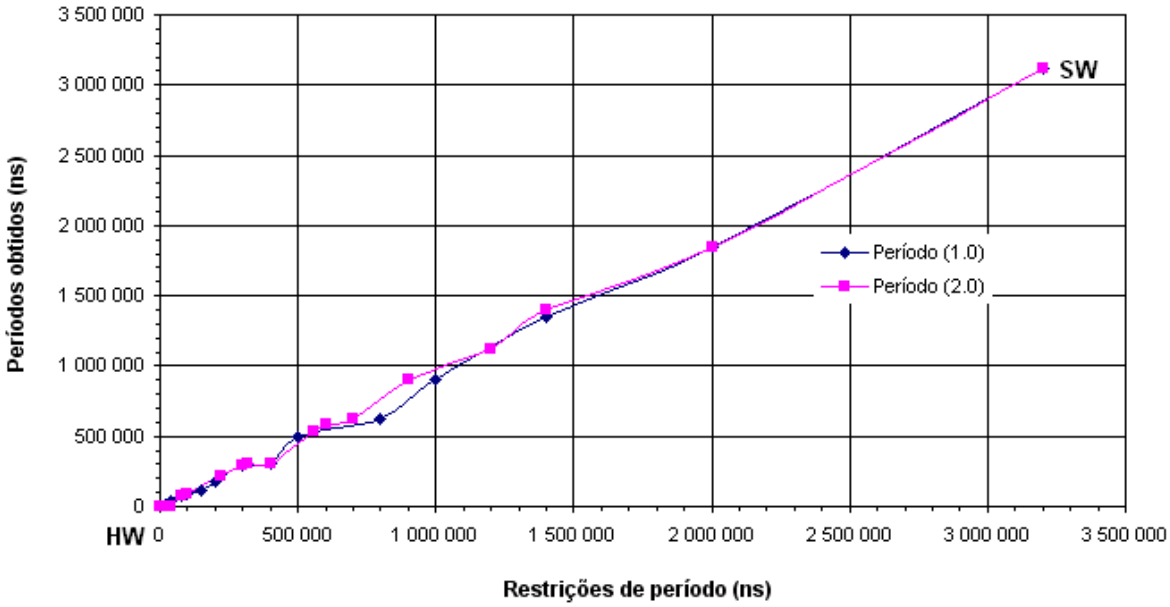


Figura 4.8: Valores de período do receptor *Rake* particionado com as versões 1.0 e 2.0.

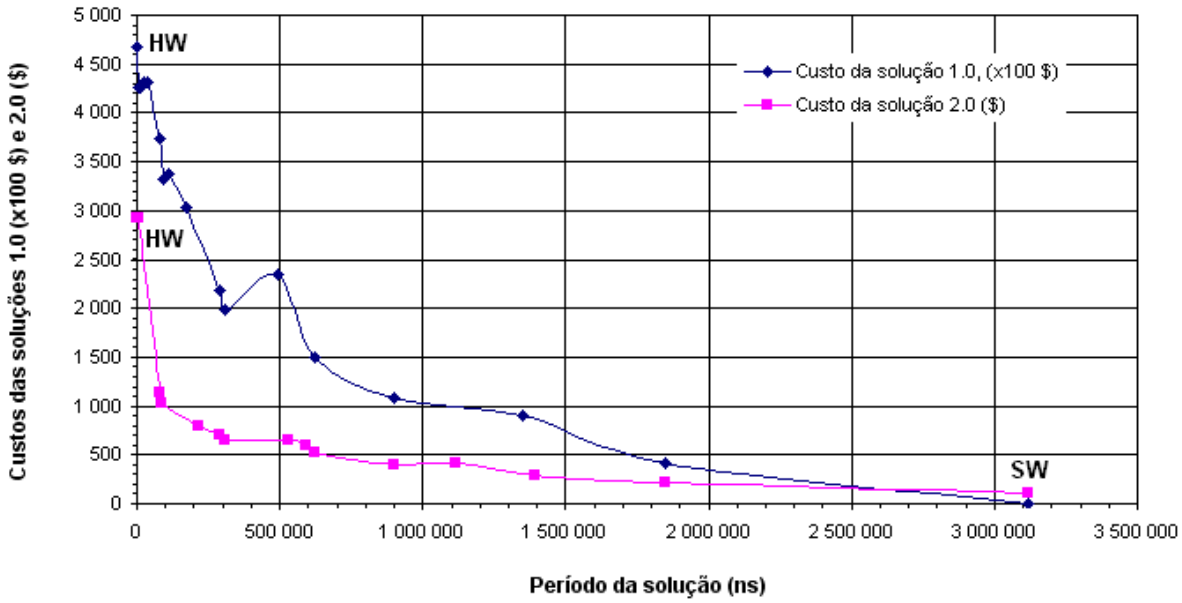


Figura 4.9: Valores de custo do receptor *Rake* particionado com as versões 1.0 e 2.0.

Versão 3.0

Para o teste da versão 3.0 do particionamento com a aplicação *Rake*, altera-se o tamanho declarado para o FPGA Stratix II no arquivo `TimeCost.xml`: como o *Rake*

é pequeno demais em comparação com o tamanho real desse FPGA, então declara-se um PE "FPGADin" com uma quantidade bem menor de LUT's do que a real (apenas 500), de modo que apenas alguns nós do *Rake* (que ocupa 824 LUT's) bastam para preenchê-lo por completo. Dessa forma, durante o agendamento atinge-se a condição em que um nó a ser mapeado no FPGA demanda mais recursos do que estão disponíveis nele, o que força a necessidade de agendamento de uma reconfiguração dinâmica. Como explicado na Seção 3.2, o tempo de reconfiguração total é arbitrado em 1 s e o consumo de potência é arbitrado em 500 mW. Executando-se o particionamento para o *Rake* com restrições de período, custo e potência de 700 ns, 5.000 $ e 700 mW respectivamente (para forçar o mapeamento da aplicação em *hardware*), observa-se no resultado final o agendamento do nó de reconfiguração `ReconfI0` entre os nós `temp1I0` e `tri_multI0`, por causa da limitação de LUT's do FPGA. Entretanto, devido à duração e ao consumo de potência da reconfiguração, as restrições do arquivo `mode.xml` não são respeitadas: o agendamento total calculado para o *Rake* fica em 1.000.000.638 ns, e o consumo de potência resulta 1.194,73 mW. Entretanto, somente 500 LUT's são empregados de forma multiplexada no tempo, ao invés de 824.

Versão 4.0

No exemplo do receptor *Rake*, o nó `tri_mult` representa um nó multiplicador específico, para um caso particular de multiplicação de um número complexo por uma constante complexa. Na versão em *hardware* do *Rake*, esse nó é implementado de forma mais simples que uma multiplicação genérica; esta é realizada pelo nó `mul_cI1`, do tipo `mul_c`. Mas em *software*, ambas as multiplicações são realizadas por meio do operador de multiplicação padrão, e portanto são nós do mesmo tipo genérico `mul_c`; em *software* não há outras alternativas de implementação para `mul_c`.

Assim, conforme explicado na Seção 3.3, para agendar-se o nó de multiplicação em *hardware*, pode-se dar ao algoritmo a opção de escolher entre a forma genérica `mul_c` ou o caso particular `tri_mult`. Para isso basta declarar, na tabela do FPGA no arquivo `TimeCost.xml`, duas instâncias consecutivas de um nó de tipo `mul_c`: uma chamada `mul_cI0_0` com os valores característicos do nó `mul_c`, e outra chamada `mul_cI0_1` com os valores de um nó `tri_mult`. Essas duas alternativas são listadas abaixo com os valores usados no *Rake*:

```
<node name='mul_cI0_0'>
    <fullName>RAKE_TEST.rakeI0.trivial_multiplierI0.mul_cI0_0</fullName>
    <className>mul_c</className>
    <time>20</time>
    <LEs>106</LEs>
```

```
        <RAMs>0</RAMs>
        <Mults>8</Mults>
        <PLLs>0</PLLs>
        <IOs>116</IOs>
        <power>4.13</power>
        <ReconfigTime>1000000000</ReconfigTime>
    </node>
    <node name='mul_cI0_1'>
        <fullName>RAKE_TEST.rakeI0.trivial_multiplierI0.mul_cI0_1</fullName>
        <className>mul_c</className>
        <time>9</time>
        <LEs>128</LEs>
        <RAMs>0</RAMs>
        <Mults>0</Mults>
        <PLLs>0</PLLs>
        <IOs>46</IOs>
        <power>3.98</power>
        <ReconfigTime>1000000000</ReconfigTime>
    </node>
```

Ao encontrar duas instâncias consecutivas de um mesmo tipo de nó, cujos nomes sejam diferentes apenas pelo algarismo depois do segundo caractere de sublinha ('_'), o algoritmo sabe que tratam-se de alternativas de implementação mutuamente exclusivas, isto é, das quais somente uma pode ser agendada. Executando-se o particionamento do *Rake* com a mesma função de custo da versão 2.0, a análise do sistema particionado mostra que o algoritmo escolhe a implementação `mul_cI0_1` acima, devido ao fato de esta apresentar menor custo que a alternativa `mul_cI0_0`.

4.6 Decodificador LDPC

Os códigos corretores de erro do tipo LDPC (Verificação de paridade de baixa densidade) foram inventados na década de 1960 por Robert Gallager e atualmente são largamente utilizados nos diferentes sistemas de comunicação digital (dos telefones móveis às comunicações espaciais). Esses códigos pertencem ao grupo dos códigos em bloco, permitindo uma aproximação do limite de Shannon por algumas frações de dB. Esse ótimo desempenho, associado a sua relativa simplicidade de decodificação, tornam os códigos LDPC muito atraentes para os sistemas de transmissão digital, como a norma de teledifusão por satélite DVB-S2, que utiliza um código LDPC irregular para a proteção dos dados no enlace de descida.

A implementação em *software* (C++) e a descrição em *hardware* (VHDL) forneci-

das para o algoritmo do decodificador de códigos LDPC foram desenvolvidas em [59] e utilizadas para a realização desta tese.

4.6.1 Grafo de entrada e determinação dos parâmetros de *software*

A figura 4.10 mostra o grafo da aplicação LDPC representado graficamente, para um conjunto de valores típicos das variáveis de iteração. As taxas de repetição dos nós do grafo provêm de um sinal de exemplo, fornecido em um arquivo de entrada, com uma SNR variando de 3 dB a 4 dB em passos de 0,25 dB, e com os demais parâmetros assumindo os seguintes valores: `NbIterations` = 3, `NbMotMax` = 10 e `NbLigne` = 4. A aplicação LDPC possui ao todo 32 nós.

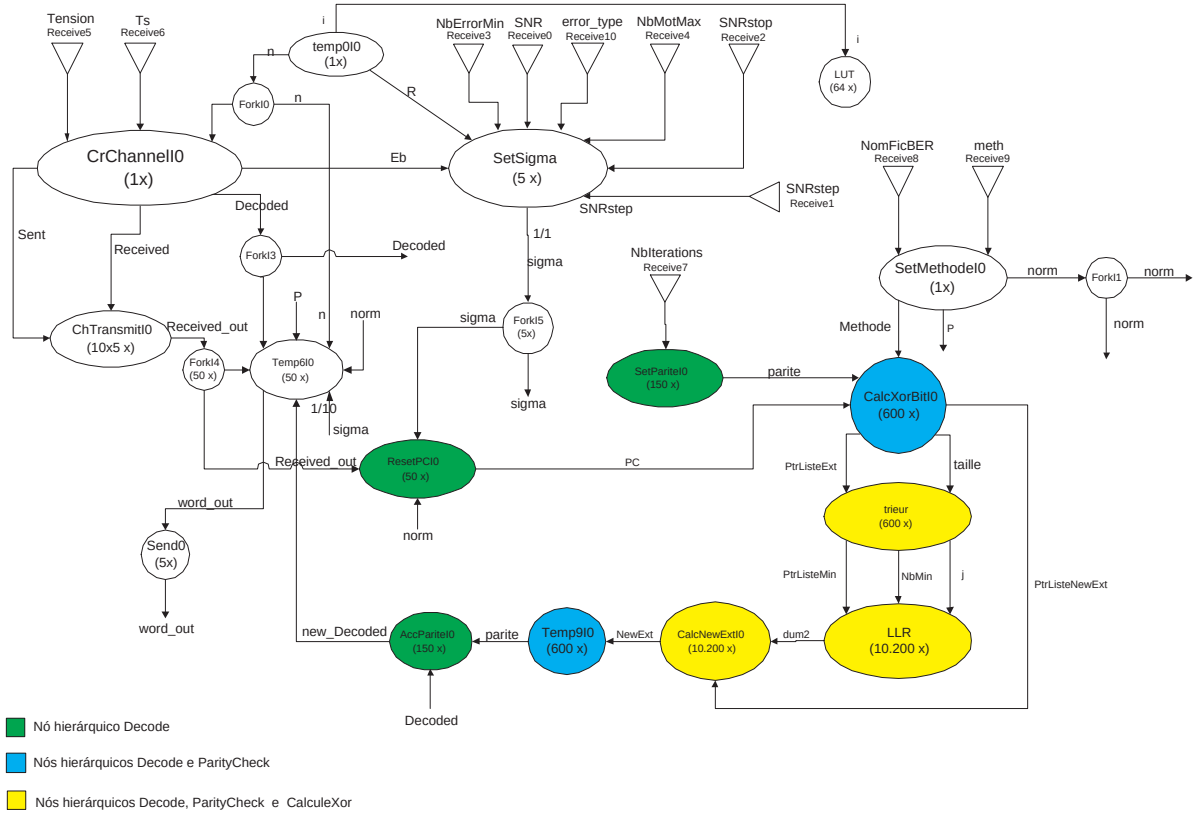


Figura 4.10: Grafo do algoritmo do decodificador LDPC.

Os valores dos tempos de execução aproximados para cada um dos nós do decodificador LDPC, no processador Nios II, bem como seus tamanhos em memória, estão listados na tabela 4.8 (página 108). Os tempos foram medidos durante a execução do código do LDPC no Nios II e correspondem a uma iteração de cada nó. A única exceção é o tempo de execução total do LDPC, que foi calculado a partir do grafo da figura 4.10

pela multiplicação dos tempos de cada nó atômico pelos respectivos números de iterações, escritos entre parênteses na figura 4.10.

Percebe-se da tabela 4.8 que o tamanho do código elementar para a aplicação LDPC não é o mesmo que no caso do *Rake*. Isto se deve ao grande número de arquivos de cabeçalho incluídos no programa em C++ do decodificador LDPC. Assim, adota-se como programa elementar o código listado abaixo que, após compilação no ambiente Nios II IDE, atinge o tamanho $\texttt{as}_{elem}$ de 2.321.576 *bytes*. Esse valor é incluído na tabela do PE Nios II no arquivo `TimeCost.xml` para o LDPC. Neste arquivo, continua-se sempre a escrever os tamanhos reais `as` dos nós da aplicação, listados na tabela 4.8. O tamanho incremental de um nó, $\texttt{as}_{incr}$, é calculado pelo algoritmo de particionamento a partir de `as` por meio da equação 4.1 da Seção 4.5.1.

```
#include <iostream>
#include <fstream>
#include <math.h>
#include <stdlib.h>
#include <cstdio>

void test() { }

int main()
{
  test();

  return 0;
}
```

4.6.2 Determinação dos parâmetros de *hardware*

A tabela 4.9 (página 109) lista os tempos de execução aproximados dos nós do LDPC, as quantidades de recursos ocupadas por eles e seus consumos de potência, para o FPGA EP2S60. Os tempos de execução foram determinados por meio de simulação pós-síntese no ambiente Quartus II e são os tempos de uma iteração de cada nó. O tempo de execução total do decodificador LDPC também foi determinado pelo Quartus II para o sinal de exemplo utilizado, levando em consideração os números de iterações de cada nó atômico do grafo, escritos entre parênteses na figura 4.10.

4.6.3 Resultados do particionamento

Esta seção apresenta as soluções heterogêneas fornecidas pelas diferentes versões do algoritmo de particionamento para o decodificador LDPC, com os valores de desempenho e custo das tabelas 4.8 e 4.9 (páginas 108 e 109) preenchidos no arquivo `TimeCost.xml`, e para diversos conjuntos de valores das restrições da aplicação preenchidos no arquivo `mode.xml`.

Versão 1.0

A restrição de área máxima estipulada para o LDPC é de 1.000 mm^2 e o custo por unidade de área é arbitrado em 500 $/$mm^2$ (valores típicos), resultando em uma restrição de custo máximo de 500.000 $. As tabelas 4.10 a 4.13 (páginas 110 a 113) mostram as propostas de implementações heterogêneas do LDPC na placa Nios II / Stratix II feitas pela versão 1.0 do algoritmo de particionamento, obtidas para diversos valores da restrição de período do LDPC no arquivo `mode.xml`. São mostrados os valores de desempenho e custo que caracterizam as várias soluções: o tempo de execução total, a quantidade de memória ocupada pela partição de *software* e o número de recursos internos do FPGA utilizados pela partição de *hardware*, expressos como uma "área equivalente". O custo monetário total inclui os preços do processador (100 $), da fração de memória utilizada e da área do FPGA, conforme as equações 3.11 e 3.12.

As tabelas 4.10 a 4.13 mostram que, tal como no exemplo do *Rake*, uma restrição de período para o LDPC (3 ms) menor que o seu tempo de execução em *hardware* (que pela tabela 4.9 é de 3,260 ms) não é satisfeita. Para restrições de período até 24,450 ms, o algoritmo 1.0 fornece uma solução inteiramente em *hardware* com um tempo de execução de 3,267 ms. Como a versão 1.0 não verifica o respeito às restrições de custo, mas somente a satisfação do desempenho exigido, essa solução viola as restrições de área e custo máximos em quase 6.000 %, atendendo somente à restrição de tempo. A área de *HW* e o custo da solução são maiores que suas restrições para períodos menores que 139 s aproximadamente. Essa falha é corrigida a partir da versão 2.0 do particionamento, como demonstrado para o *Rake*.

O gráfico da figura 4.11 mostra que os valores de período do LDPC heterogêneo fornecidos pela versão 1.0 acompanham os valores de suas restrições, e são sempre inferiores a estes. Para comparação, o tempo de execução do LDPC em *hardware* é de 3,267 ms, e seu tempo de execução em *software* é de 226 s, quase 70 mil vezes maior. Esse é o máximo ganho de desempenho que pode ser obtido para o LDPC com a migração de funções da solução em *software* para o *hardware* dedicado.

O gráfico da figura 4.12 mostra o comportamento dos valores da área de *hardware*

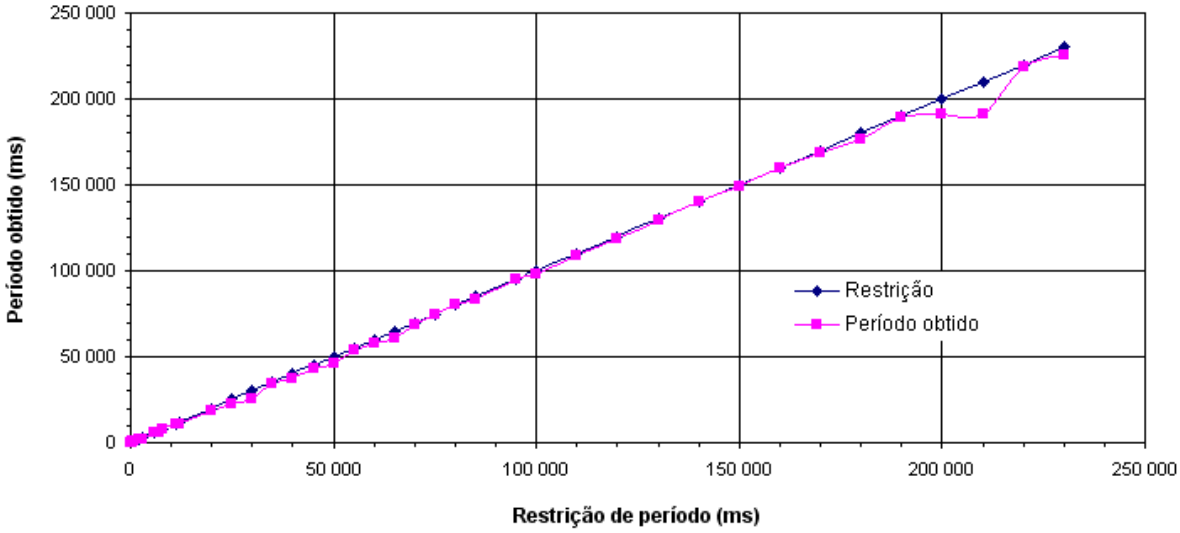


Figura 4.11: Valores de período do LDPC particionado com a versão 1.0.

e do custo monetário das diversas soluções, em função dos valores de período destas. Observa-se que a área e o custo formam três patamares de valores. As transições entre esses patamares são provocadas pelas transferências, de *hardware* para *software*, de nós que ocupam uma grande área em *hardware*: a primeira é devida ao mapeamento do nó `temp6I0` e a segunda, do nó `ResetPCI0`, provocando uma queda considerável no custo. O custo de implementação do LDPC inteiramente em *hardware* é de 30.224 k$, e seu custo inteiramente em *software* é de 103 $, uma diferença de custo de 293 mil vezes.

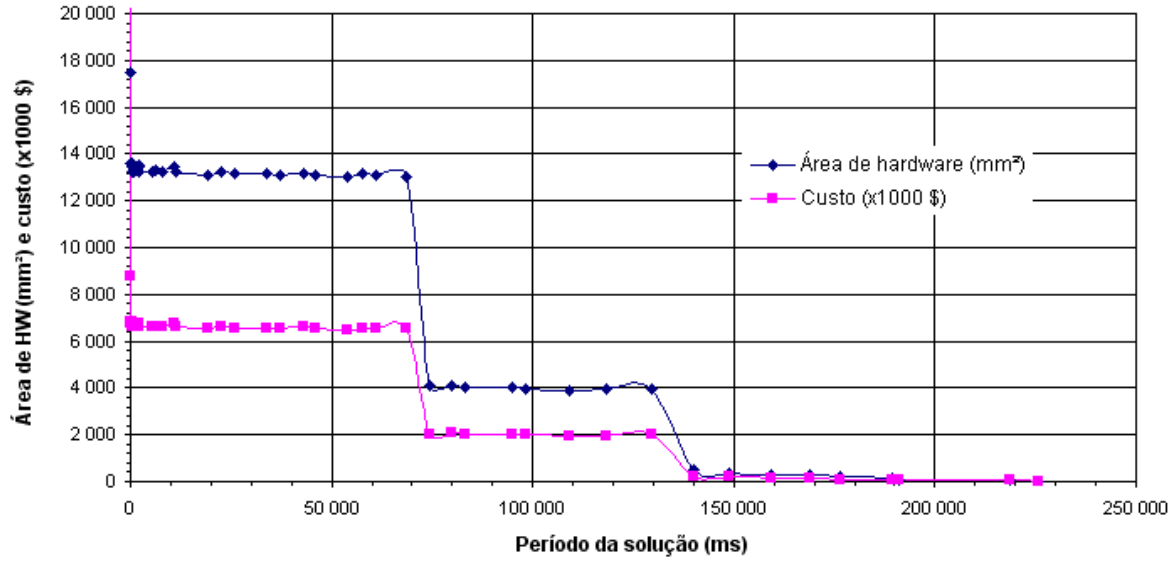


Figura 4.12: Valores de custo do LDPC particionado com a versão 1.0.

Tal como no exemplo do *Rake*, todas as partições de *software* possuem tamanhos muito próximos, e suas contribuições para os custos das soluções heterogêneas variam

entre 102 e 103 $. O tamanho estimado pelo particionador para a solução inteiramente em *software* (2.918.674 *bytes*) é apenas 25,7 % maior que o tamanho do código elementar (2.321.576 *bytes*). Essa diferença é menor ainda na realidade, porque o tamanho estimado pelo particionador está em excesso em relação ao valor real da tabela 4.8 por 7 %, por causa das otimizações feitas pelo compilador e não consideradas no particionamento.

Observa-se ainda, das tabelas 4.10 a 4.13, que algumas soluções heterogêneas consomem mais memória que a solução inteiramente em *software*; isso se deve à consideração do tamanho em memória dos nós de interfaceamento *SW-HW*.

As tabelas 4.10 a 4.13 mostram como a versão 1.0 do particionador é ineficiente em reduzir o custo do sistema heterogêneo: a solução com o nó `SetMethodeI0` em *software*, para uma restrição de período de 25 ms, possui um custo maior que o da solução inteiramente em *hardware* encontrada para a restrição de 10 ms. O mapeamento de `SetMethodeI0` para *software* acarreta um aumento de custo porque este nó ocupa uma área de *hardware* pequena (poucos LUT's do FPGA) e os cinco nós de interfaceamento inseridos em seu lugar anulam a vantagem de seu mapeamento em *software*, com o efeito resultante de um aumento do custo. O mesmo fenômeno ocorre nas soluções para as restrições de 180 ms e 230 ms, que possuem tanto o tempo de execução quanto o custo maiores que os da solução anterior para 150 ms. Isso mostra que nem sempre o mapeamento de mais nós de *hardware* para *software* reduz necessariamente o custo do sistema, o que dependerá das quantidades e dos custos dos nós de interfaceamento inseridos, em relação aos custos dos nós deslocados para *software*. Conclui-se, tal como no exemplo do *Rake*, que o objetivo do particionamento na versão 1.0 é apenas encontrar uma solução cujo tempo de execução respeite à restrição correspondente, ou seja, ajustar o tempo de execução da solução à sua restrição. Não há a tentativa de minimização do custo da solução heterogênea.

Comparando-se a solução em *software* com a solução heterogênea de menor tempo de execução (aquela somente com o nó `SetMethodeI0` em *SW*), a redução do tempo de execução é de 9.240 vezes, mas o aumento de custo é de 293 mil vezes; ao mesmo tempo, a redução da utilização de memória é de apenas 1,23. Percebe-se uma vez mais que a influência da quantidade de memória utilizada sobre o custo da solução é desprezível, pois o custo é determinado pela área de *hardware*. Por outro lado, comparando-se a solução inteiramente em *hardware* do LDPC com a solução heterogênea de menor custo (aquela somente com o nó `ChTransmitI0` em *HW*), o aumento do tempo de execução é de 58.480 vezes para uma redução de custo de 649 vezes, praticamente a mesma redução da utilização de *hardware*.

Versão 2.0

As tabelas 4.14 a 4.16 (páginas 114 a 116) mostram as propostas de implementações heterogêneas para o LDPC na placa Nios II / Stratix II, feitas pela versão 2.0 do algoritmo de particionamento, obtidas para diversos valores das restrições de período de execução, custo e potência da tarefa LDPC no arquivo `mode.xml`. Analisam-se os efeitos das variações paramétricas das restrições sobre os resultados do particionamento e mostram-se os valores de desempenho (tempo de execução), custo e potência que caracterizam as soluções.

As tabelas 4.14 a 4.16 mostram novamente que a versão 2.0 do algoritmo de particionamento respeita todas as restrições para o sistema particionado. Tal como no exemplo do *Rake*, o tempo de execução do grafo sempre satisfaz a sua restrição, mas não é o mínimo valor possível. Isso pode ser observado quando relaxa-se a restrição de período, aceitando-se valores maiores: o algoritmo aproveita a "folga" permitida no tempo de execução para tentar minimizar o custo do sistema, alocando menos nós no *hardware* dedicado.

A comparação entre a quinta linha (`Tr` = 100 ms, `Cr` = 5.000 $ e `Pr` = 900 mW) e a sexta linha (`Tr` = 100 ms, `Cr` = 5.000 $ e `Pr` = 800 mW) de valores da tabela 4.14 mostra que a menor restrição de potência no segundo caso é decisiva para definir os resultados do particionamento. Observando-se primeiro a sexta linha e depois a quinta, pode-se confirmar que a prioridade do algoritmo é realmente a minimização do custo: a permissão de uma maior potência na quinta linha dá margem para o algoritmo utilizar o PE de *software*, que acrescenta 62 mW ao orçamento de potência, para substituir um nó que estava em *hardware* e assim reduzir o custo. Toda redução no consumo de LUT's significa **potencialmente** que um FPGA menor e mais barato do que o disponível poderia ser empregado, diminuindo assim o custo da implementação de *hardware*, e do sistema como um todo.

O gráfico da figura 4.13 mostra o comportamento dos períodos obtidos para as soluções junto a suas restrições correspondentes, onde percebe-se que não há um esforço do algoritmo em minimizá-los, mas apenas em mantê-los abaixo das restrições, tal como faz a versão 1.0.

O gráfico da figura 4.14 mostra os custos monetários das soluções juntamente com suas restrições, em função dos períodos das soluções, demonstrando a eficácia do algoritmo em minimizar esses custos. A redução de custo observada da solução inteiramente em *hardware* (4.545 $) para a solução inteiramente em *software* (103 $) é de 44 vezes. Nas tabelas 4.14 a 4.16, o uso de restrições de custo muito maiores que os custos reais do sistema mostra que o algoritmo de particionamento não explora a folga permitida ao valor de custo para minimizar o tempo de execução, e sim minimiza o custo em todos os

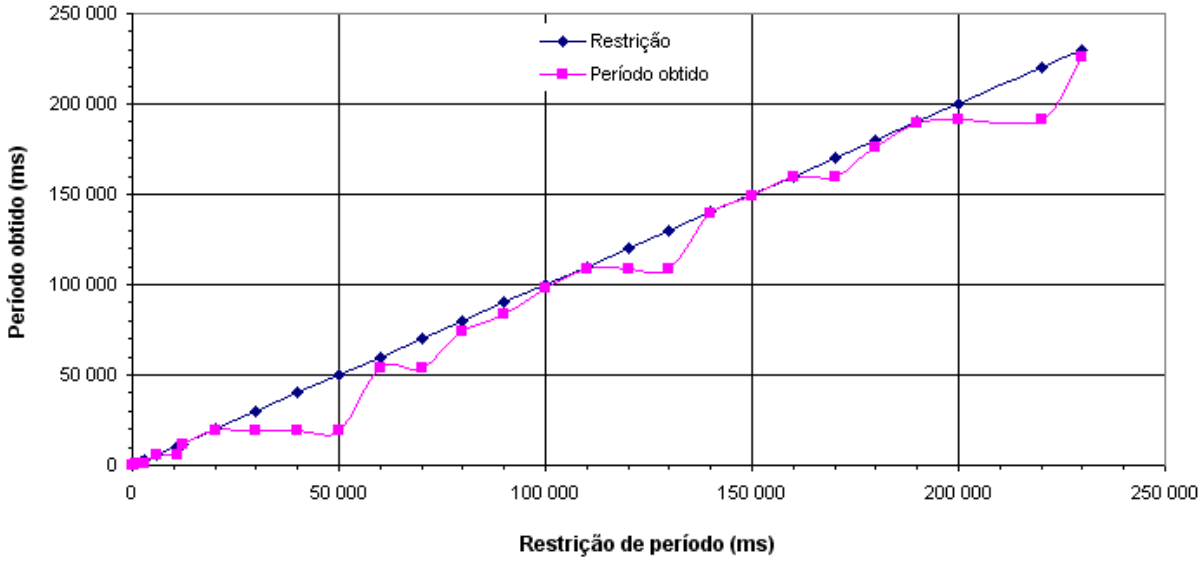


Figura 4.13: Valores de período do LDPC particionado com a versão 2.0.

casos.

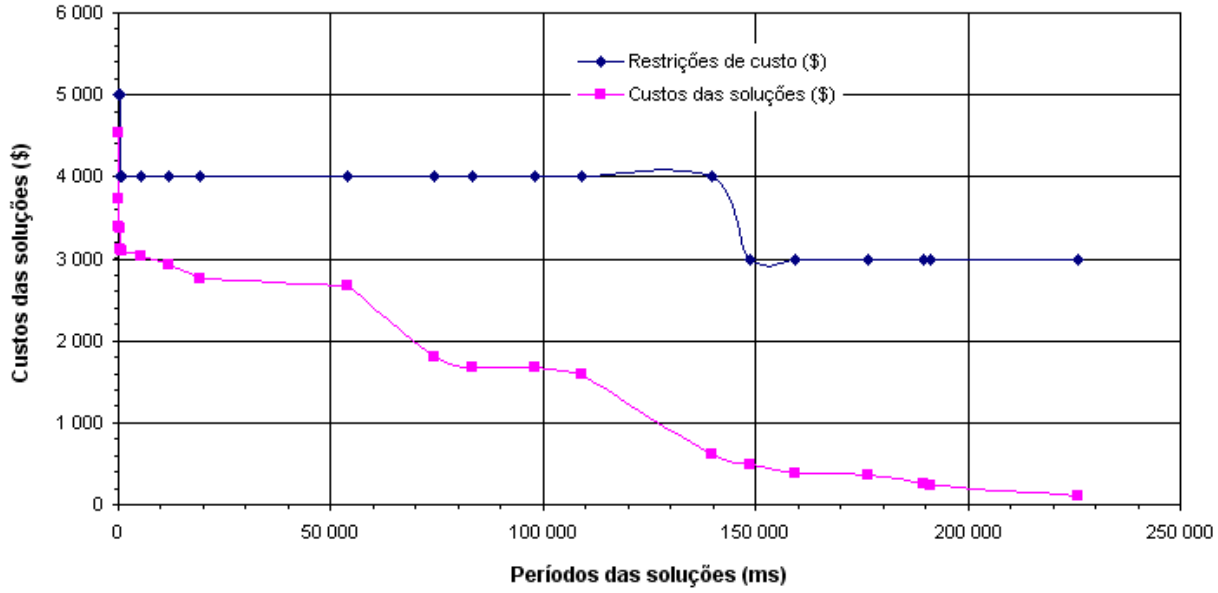


Figura 4.14: Valores de custo do LDPC particionado com a versão 2.0.

Comparando-se a solução em *software* com a solução heterogênea de menor tempo de execução (aquela somente com o nó `SetSigmaI0` em SW), a redução do tempo de execução é de 8.270 vezes e o aumento de custo é de 36,2 vezes. A redução no tempo de execução não é a mesma da versão 1.0: a solução heterogênea de menor tempo de execução fornecida pela versão 2.0 é diferente daquela obtida com a versão 1.0, porque a função de custo introduzida na versão 2.0 direciona o particionamento a minimizar o custo. Já os valores de custo e o aumento de custo observado são muito menores do que os da

versão 1.0 do algoritmo: isso decorre da utilização de um FPGA no particionamento e da consideração precisa da ocupação de seus recursos internos, enquanto a versão 1.0 utiliza um ASIC. Como foram declarados valores típicos de custo para ASIC's, os resultados das duas versões refletem a realidade de que o custo de implementação em ASIC é muito superior ao custo de implementação em FPGA. Mas o mais importante no gráfico da figura 4.14 não são os valores numéricos e sim o comportamento dos custos das soluções em função de suas restrições.

Por outro lado, comparando-se a solução inteiramente em *hardware* do LDPC com a solução heterogênea de menor custo (aquela somente com o nó `ChTransmitIO` em *HW*), o aumento do tempo de execução é de 58.480 vezes (o mesmo da versão 1.0) para uma redução de custo de 19,6 vezes (bem menor que a da versão 1.0).

O gráfico da figura 4.15 mostra o comportamento dos valores de potência das soluções geradas, que se concentram um pouco acima da soma das potências estáticas do FPGA (645 mW) e do processador (62 mW), sempre presentes em todas as soluções heterogêneas. Somente na solução inteiramente em *software* o valor de potência é muito diferente dos demais, e igual a 62 mW.

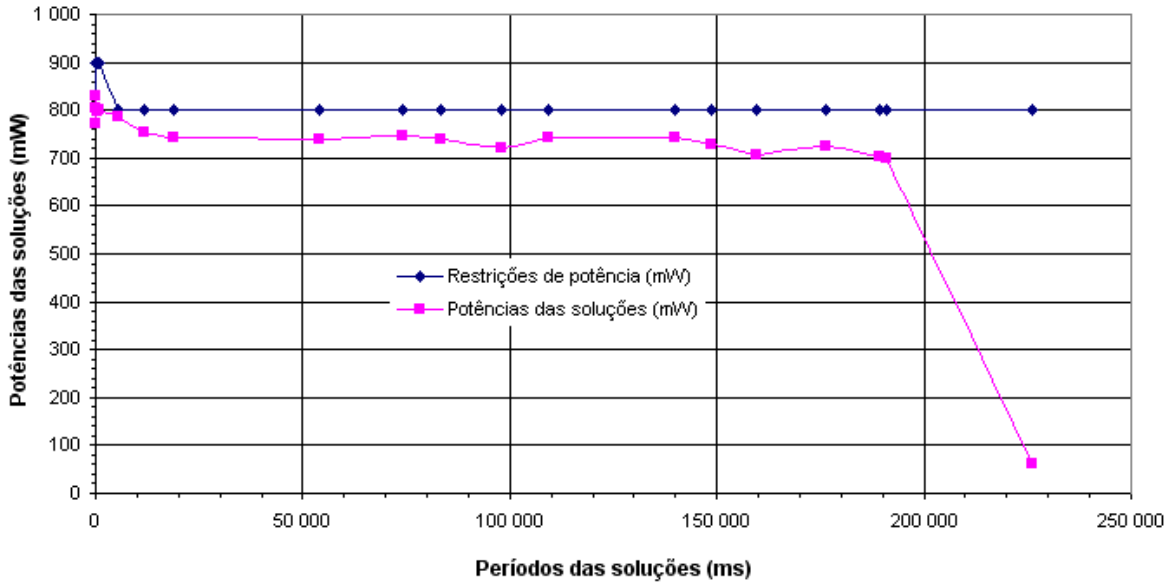


Figura 4.15: Valores de potência do LDPC particionado com a versão 2.0.

O gráfico da figura 4.16 mostra simultaneamente os valores de tempo de execução, custo e potência das soluções, normalizados por suas restrições correspondentes em cada solução. Neste gráfico, pode-se observar que a razão C/C_r é a mais minimizada das três, enquanto as curvas T/T_r e P/P_r permanecem geralmente próximas à linha limite custo/restrição = 1.

Comparando-se as curvas de período do LDPC ao ser particionado pelas versões 1.0

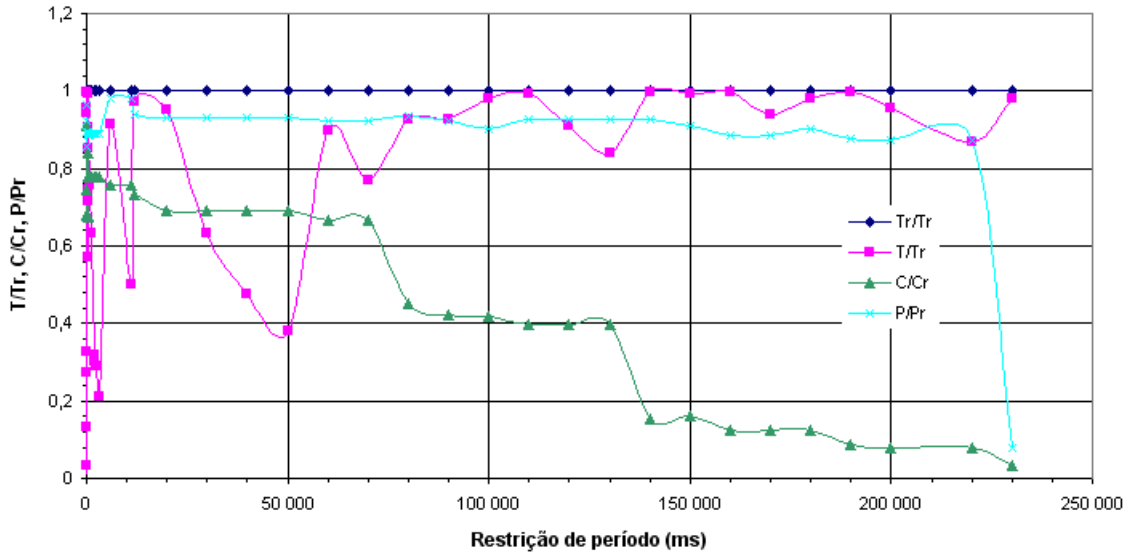


Figura 4.16: Valores normalizados de período, custo e potência para o LDPC, com a versão 2.0.

e 2.0 do particionamento, mostradas na figura 4.17, observa-se que elas diferem entre as duas versões, o que é devido ao fato de que a versão 2.0 busca minimizar o custo da aplicação e por isso algumas soluções heterogêneas encontradas pela versão 1.0 não são fornecidas pela versão 2.0. Lendo-se as tabelas 4.10 a 4.13 da versão 1.0, notam-se várias ocasiões em que o aumento da restrição de período leva a uma solução com período próximo à nova restrição, mas com um custo **maior**. Este tipo de situação é filtrado pela versão 2.0, a qual fornece sempre a solução de menor custo existente, para cada restrição de período. Esta é a causa das divergências entre as duas curvas da figura 4.17.

Os custos do LDPC particionado pelas duas versões, além da grande diferença numérica explicada anteriormente, não possuem o mesmo comportamento: a curva de custos da versão 2.0 é bem mais suave que a da versão 1.0, resultado da consideração precisa dos custos dos nós mapeados no FPGA. Isso pode ser visto na figura 4.18 (página 103).

Versão 3.0

Para o teste da versão 3.0 do particionamento com a aplicação LDPC, é necessário alterar-se novamente o tamanho declarado para o FPGA Stratix II no arquivo `TimeCost.xml`, pois o LDPC é pequeno demais em comparação com o tamanho real desse FPGA. Declara-se então um PE do tipo "FPGADin" com uma quantidade bem menor de LUT's do que a real (apenas 1.500), de modo que alguns nós do LDPC (que ocupa 2.503 LUT's) bastam para preenchê-lo por completo. Dessa forma, durante o agendamento é atingida a condição em que um nó a ser mapeado demanda mais recursos

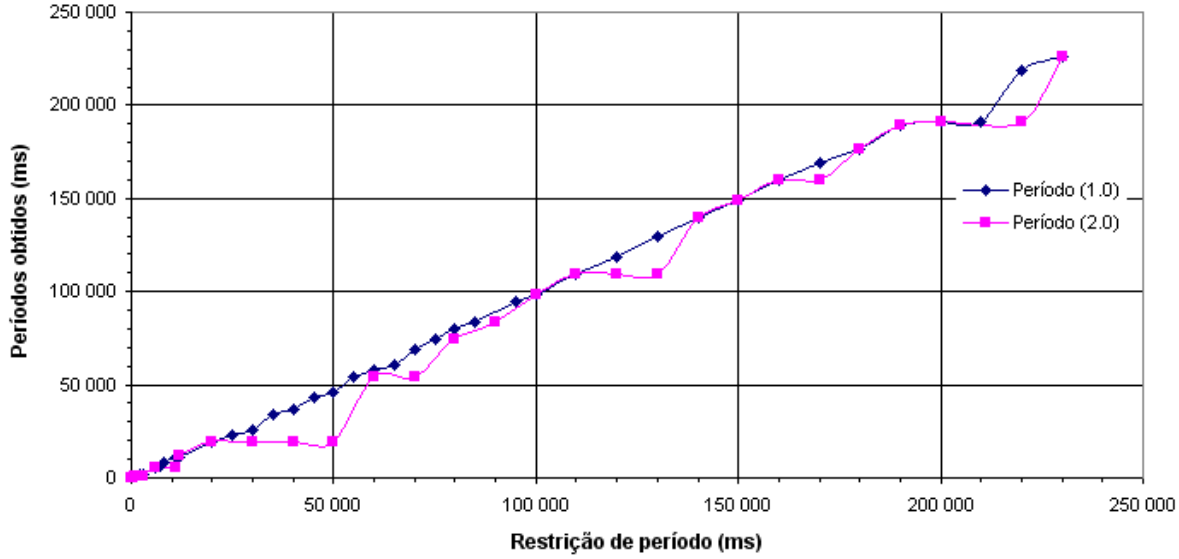


Figura 4.17: Valores de período do LDPC particionado com as versões 1.0 e 2.0.

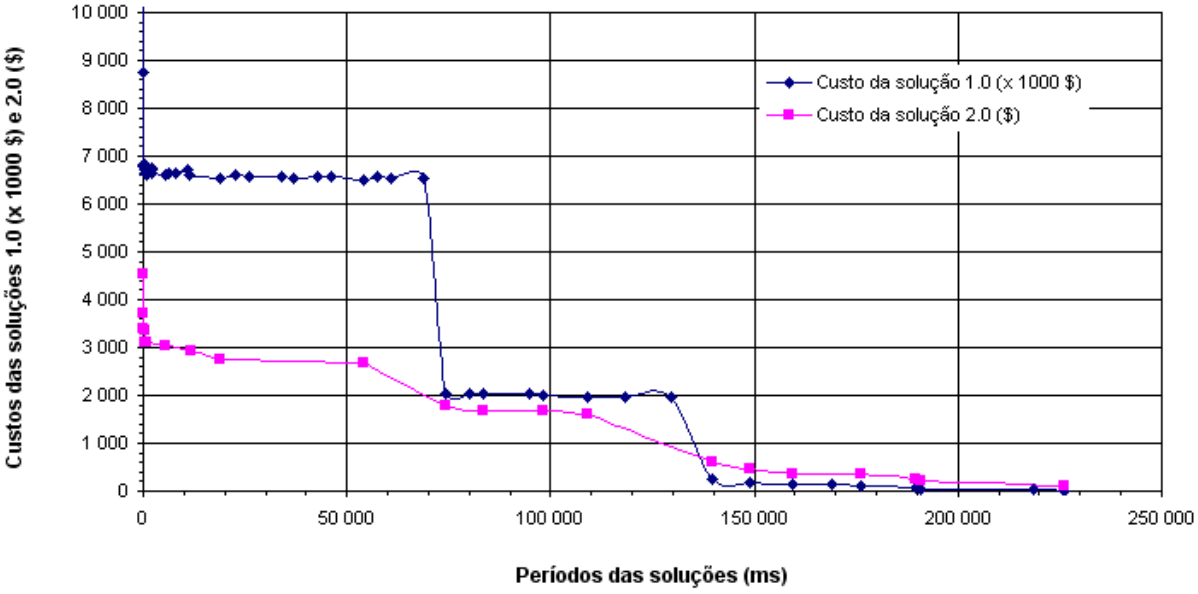


Figura 4.18: Valores de custo do LDPC particionado com as versões 1.0 e 2.0.

do que a quantidade que ainda resta disponível no FPGA, o que força a necessidade de agendamento de uma reconfiguração dinâmica. Como explicado na Seção 3.2, o tempo de reconfiguração total é arbitrado em 1 s e o consumo de potência é arbitrado em 500 mW. Executando-se o particionamento do LDPC com restrições de período, custo e potência de 3,270 ms, 5.000 $ e 800 mW respectivamente (para forçar a implementação em *hardware*), observa-se na solução heterogênea o agendamento do nó de reconfiguração dinâmica `ReconfIO` entre os nós `CalcNewExtIO` e `temp9IO`, por causa do esgotamento dos LUT's do FPGA. Entretanto, devido à duração e ao consumo de potência da reconfiguração, as

restrições do arquivo `mode.xml` não são respeitadas: o agendamento total do LDPC fica em 1.003,267 ms, e o consumo de potência em 1.272,5 mW. Entretanto, somente 1.500 LUT's são empregados de forma multiplexada no tempo, ao invés de 2.503.

Versão 4.0

A versão 4.0 do particionador não foi testada com a aplicação LDPC, porque esta não possui qualquer nó com mais de uma alternativa de implementação, em *hardware* ou em *software*.

4.7 Conclusão

Os testes das três versões otimizadas do algoritmo de particionamento, propostas no Capítulo 3, demonstram os benefícios obtidos com a integração de características de qualidade de outros algoritmos de particionamento propostos na literatura. A versão 2.0 mostra-se eficiente em minimizar os custos das soluções heterogêneas, por meio do acréscimo de uma função de custo englobando custos e restrições e avaliada em cada iteração do particionamento. Essa versão pode ser direcionada também para minimizar o tempo de execução ou o consumo de potência das soluções, conforme os valores atribuídos aos seus pesos. As versões 3.0 e 4.0 acrescentam novas funcionalidades ao algoritmo, habilitando-o a prever a reconfiguração dinâmica do FPGA no agendamento e a escolher a alternativa de implementação de um nó que melhor atenda ao objetivo da função de custo.

Os testes foram realizados com aplicações de PDS de pequena complexidade, um receptor *Rake* e um decodificador LDPC, respectivamente com 21 e 32 nós de processamento. Os tempos de execução dos particionamentos variaram entre 1 s e 8 s. Eles dependem da complexidade da aplicação, em termos de quantidade de nós e densidade do grafo, e das restrições: como o algoritmo sempre parte de uma solução inicial inteiramente em *software*, decorre que restrições mais agressivas (que exijam que a maioria dos nós seja mapeada em *hardware*) demandam maior tempo de particionamento que restrições mais conservadoras (que sejam satisfeitas com poucos nós em *hardware*). Uma solução inteiramente em *software* é obtida em apenas uma iteração do particionamento. Somente a versão 4.0 do algoritmo acrescenta à complexidade do algoritmo inicial.

A principal dificuldade para a utilização das versões atuais do algoritmo de particionamento com aplicações reais reside na criação do grafo para uma aplicação, a partir de seu código-fonte em C, de tal forma que esse grafo seja também adequado para o código em VHDL da mesma aplicação. Quando as aplicações são desenvolvidas por terceiros,

seus códigos em C e em VHDL são otimizados para *software* e para *hardware* de forma independente, considerando particularidades de cada domínio de implementação. As aplicações nesse caso não são criadas visando o particionamento ou uma futura possibilidade de implementação heterogênea, e portanto seus códigos não refletem uma preocupação com a divisão modular das funcionalidades e a correspondência das mesmas entre os blocos de *software* e *hardware*. Não é necessariamente trivial dividir as funcionalidades em blocos para o particionamento, pois elas não estão claramente identificadas e isoladas entre si, tornando difícil e impreciso escolher os nós para caracterizar, posicionar os comandos de medição de tempo no Nios II e interpretar seus resultados posteriormente. Essa é outra grande dificuldade no uso deste algoritmo de particionamento para projetos práticos: a obtenção dos valores para o preenchimento da tabela de perfis "Nó-PE", para cada aplicação que se desejar particionar e sobre uma plataforma específica.

Diante dessas dificuldades e da constatação de suas causas, propõe-se que as aplicações a particionar sejam descritas em uma linguagem de sistema mais abstrata que C ou VHDL, como por exemplo a linguagem visual do ambiente Ptolemy, que facilite a geração do seu grafo e as estimativas dos tempos de execução e dos custos. A descrição deve ser feita de forma modular, com a separação clara das tarefas de processamento em blocos, para que a posterior criação das funções em C e VHDL reflita essa modularidade.

Tabela 4.6: Resultados do particionamento do receptor *Rake* pela versão 1.0.

Restrição de período (μs)	Solução	Período obtido (μs)	Memória (*bytes*)	Área equivalente (mm^2)	Custos ($)
0,5	Não há	-	-	-	-
0,7	Inteiramente em *HW*	0,639	-	936 (87,2 %)	468.000
5	Inteiramente em *HW*	0,639	-	936 (87,2 %)	468.000
6	`temp0I0` em *SW*	5,532	181.218	852 (70,4 %)	426.103
20	`temp0I0` e `Send0` em *SW*	15,046	181.820	852 (70,4 %)	426.103
30	`ReceiveX` e `temp0I0` em *SW*	25,28	187.539	862 (72,4 %)	431.103
40	`ReceiveX`, `temp0I0` e `Send0` em *SW*	35	188.141	862 (72,4 %)	431.103
80	`mul_cI1` em *SW*	78,56	182.830	748 (49,6 %)	374.103
100	`temp0I0`, `mul_cI1` e `Send0` em *SW*	93,45	183.844	664 (32,8 %)	332.103
150	`ReceiveX`, `temp0I0`, `mul_cI1` e `Send0` em *SW*	113	190.165	674 (34,8 %)	337.103
200	`ReceiveX`, `temp0I0`, `mul_cI1`, `addI1` e `Send0` em *SW*	175	190.947	608 (21,6 %)	304.103
300	`rake_initI0`, `temp0I0`, `mul_cI1`, `addI1` e `Send0` em *SW*	287	186.115	436 (-12,8 %)	218.103
400	`ReceiveX`, `rake_initI0`, `temp0I0`, `mul_cI1`, `addI1` e `Send0` em *SW*	307	191.533	396 (-20,8 %)	198.103
500	`rake_initI0`, `temp0I0`, `temp3I0` e `mul_cI1` em *SW*	494	186.148	469 (-6,2 %)	234.603
800	`ReceiveX`, `rake_initI0`, `temp0I0`, `tri_multI0`, `mul_cI1`, `addI1` e `Send0` em *SW*	621	193.557	298 (-40,4 %)	149.103
1.000	`temp1I0`, `temp2I0` e `integ_addI0` em *HW*	899	192.867	215 (-57 %)	107.603
1.400	`temp1I0` e `temp2I0` em *HW*	1.351	194.589	179 (-64,2 %)	89.603
2.000	`temp2I0` em *HW*	1.846	193.020	84 (-83,2 %)	42.103
3.200	Inteiramente em *SW*	3.121	191.649	-	103

Tabela 4.7: Resultados do particionamento do receptor *Rake* pela versão 2.0.

Restrições			Solução	Atributos		
Período (μs)	Custo ($)	Potência (mW)		Período (μs)	Custo ($)	Potência (mW)
0,7	5.000	700	Inteiramente em *HW*	0,638	2.931,4	694,73
5	7.000	700	Inteiramente em *HW*	0,638	2.931,4	694,73
6	7.000	700	Inteiramente em *HW*	0,638	2.931,4	694,73
6	7.000	800	`temp0I0` em *SW*	5,521	2.917,3	752,73
10	5.000	700	Inteiramente em *HW*	0,638	2.931,4	694,73
10	5.000	800	`temp0I0` em *SW*	5,521	2.917,3	752,73
40	5.000	1.000	`temp0I0` em *SW*	5,521	2.917,3	752,73
80	5.000	1.000	`mul_cI1` em *SW*	78,56	1.142,8	752,6
100	5.000	800	`temp0I0` e `mul_cI1` em *SW*	83,44	1.026	748,6
220	4.000	800	`rake_initI0`, `temp0I0` e `mul_cI1` em *SW*	216	786,87	731,95
300	4.000	800	`rake_initI0`, `temp0I0`, `mul_cI1`, `addI1` e `Send0` em *SW*	287,4	709,04	729,36
300	4.000	1.000	`rake_initI0`, `temp0I0`, `mul_cI1`, `addI1` e `Send0` em *SW*	287,4	709,04	729,36
320	3.500	800	`ReceiveX`, `rake_initI0`, `temp0I0`, `mul_cI1`, `addI1` e `Send0` em *SW*	307,4	653,5	728,36
400	3.500	1.000	`ReceiveX`, `rake_initI0`, `temp0I0`, `mul_cI1`, `addI1` e `Send0` em *SW*	307,4	653,5	728,36
560	3.000	800	`rake_initI0`, `temp0I0`, `tri_multI0` e `mul_cI1` em *SW*	529,7	650,65	727,97
600	3.000	800	`rake_initI0`, `temp0I0`, `tri_multI0`, `mul_cI1` e `addI1` em *SW*	591	586,71	726,38
700	3.000	800	`temp1`, `temp2`, `temp3` e `integ_additionneur` em *HW*	621	517,28	724,38
900	3.000	750	`temp1`, `temp2` e `integ_additionneur` em *HW*	899	401,87	723,23
1.200	2.500	800	`temp2`, `temp3` e `integ_additionneur` em *HW*	1.117	412,98	717,59
1.400	2.500	800	`temp2` e `integ_additionneur` em *HW*	1.395	297,58	716,44
2.000	2.500	750	`temp2` em *HW*	1.846	219,73	712,1
3.200	2.500	70	Inteiramente em *SW*	3.121	103	62

Tabela 4.8: Valores de desempenho e custo dos nós do decodificador LDPC em *software*.

Nó	Tempo de execução (ms)	Tamanho em memória (*bytes*)
LUT	1,566	2.343.284
Receive	7,178	2.328.558
SetMethode	21,184	2.342.572
CrChannel	154,955	2.357.066
ChTransmit	697,715	2.357.135
ResetPC	1.187,5	2.330.874
SetParite	0,541	2.321.639
CalcXorBit	9,557	2.329.953
trieur	12,998	2.350.644
LLR	1,446	2.347.314
CalcNewExt	0,318	2.329.449
CalculeXor	30,585	2.385.410
Temp9	0,193	2.325.275
ParityCheck	42,189	2.389.307
AccParite	172,289	2.329.190
Decode	2.212	2.501.397
Temp6	1.441	2.366.236
SetSigma	5,464	2.345.971
Send	11,01	2.324.930
Temp0	1.618	2.563.980
LDPC	219.611	2.728.985

Tabela 4.9: Valores de desempenho e custo dos nós do decodificador LDPC em *hardware*.

Nó	Latência (ns)	Recursos utilizados	Potência (mW)
`LUT`	14	4 LUT's, 12 pinos	0,19
`Receive`	7	10 LUT's, 2 pinos	0,55
`SetMethode`	150	43 LUT's, 48 pinos	0,64
`CrChannel`	273	78 LUT's, 84 *bits* de memória, 130 pinos	4,69
`ChTransmit`	676	63 LUT's, 48 pinos	5,25
`ResetPC`	990	674 LUT's, 1640 *bits* de memória, 87 pinos	6,65
`SetParite`	102	8 LUT's, 22 pinos	0,76
`CalcXorBit`	190	10 LUT's, 11 pinos	2,45
`trieur`	450	144 LUT's, 55 pinos	35,34
`LLR`	320	78 LUT's, 70 pinos	20,12
`CalcNewExt`	117	38 LUT's, 67 pinos	10,35
`CalculeXor`	754	260 LUT's, 137 pinos	65,8
`Temp9`	630	217 LUT's, 2144 *bits* de memória, 45 pinos	25,61
`ParityCheck`	1.170	487 LUT's, 2144 *bits* de memória, 78 pinos	93,86
`AccParite`	85	6 LUT's, 64 *bits* de memória, 27 pinos	0,73
`Decode`	2.388	1175 LUT's, 3848 *bits* de memória, 108 pinos	102
`Temp6`	1.450	826 LUT's, 4792 *bits* de memória, 112 pinos	8,74
`SetSigma`	827	195 LUT's, 24768 *bits* de memória, 104 pinos	4,21
`Send`	7	10 LUT's, 20 pinos	0,55
`Temp0`	9	3 LUT's, 15 pinos	0,68
`LDPC`	3.260.000	2503 LUT's, 17688 *bits* de memória, 75 pinos	133

Tabela 4.10: Resultados do particionamento do decodificador LDPC pela versão 1.0.

Restrição de período (ms)	Solução	Período obtido (ms)	Memória (*bytes*)	Área equivalente (mm^2)	Custos ($)
3	Não há	-	-	-	-
3,270	Inteiramente em HW	3,267	-	60.448 (5.944,8 %)	30.224.000
10	Inteiramente em HW	3,267	-	60.448 (5.944,8 %)	30.224.000
25	`SetMethodeI0` em SW	24,45	2.369.952	60.465 (5.946,5 %)	30.233.102
50	`SetSigmaI0` em SW	27,32	2.405.181	17.505 (1.650,5 %)	8.752.402
150	`SetSigmaI0` e `temp9I0` em SW	143,12	2.419.216	13.588 (1.258,8 %)	6.794.302
180	`SetSigmaI0`, `SetMethodeI0` e `temp9I0` em SW	164,38	2.470.946	13.615 (1.261,5 %)	6.807.802
230	`SetSigmaI0`, `SetPariteI0` e `temp9I0` em SW	224,35	2.432.969	13.610 (1.261 %)	6.805.302
300	`CrChannelI0`, `SetSigmaI0` e `temp9I0` em SW	298,08	2.485.440	13.435 (1.243,5 %)	6.717.602
350	`CrChannelI0`, `SetSigmaI0`, `SetMethodeI0` e `temp9I0` em SW	319,26	2.533.816	13.452 (1.245,2 %)	6.726.102
400	`CrChannelI0`, `SetSigmaI0`, `SetMethodeI0`, `temp9I0` e `Send0` em SW	374,31	2.544.152	13.452 (1.245,2 %)	6.726.102
500	`ReceiveIX`, `CrChannelI0`, `SetSigmaI0`, `SetMethodeI0`, `temp9I0` e `Send0` em SW	453,27	2.554.488	13.252 (1.225,2 %)	6.626.102
600	`CrChannelI0`, `SetPariteI0`, `LUTI0`, `SetSigmaI0`, `temp9I0`, `SetMethodeI0` e `Send0` em SW	555,68	2.583.241	13.470 (1.247 %)	6.735.102
1.000	`ReceiveIX`, `CrChannelI0`, `SetPariteI0`, `LUTI0`, `SetSigmaI0`, `temp9I0`, `SetMethodeI0` e `Send0` em SW	634,64	2.583.241	13.250 (1.225 %)	6.625.102
2.000	`temp0I0`, `CrChannelI0`, `SetSigmaI0`, `SetMethodeI0`, `temp9I0` e `Send0` em SW	1.992	2.779.300	13.449 (1.244,9 %)	6.724.602
2.090	`temp0I0`, `CrChannelI0`, `SetPariteI0`, `SetSigmaI0`, `SetMethodeI0`, `temp9I0` e `Send0` em SW	2.073	2.803.389	13.491 (1.249,1 %)	6.745.603
2.100	`temp0I0`, `CrChannelI0`, `LUTI0`, `SetSigmaI0`, `temp9I0`, `SetMethodeI0` e `Send0` em SW	2.092	2.811.344	13.465 (1.246,5 %)	6.732.603
2.200	`temp0I0`, `CrChannelI0`, `LUTI0`, `SetPariteI0`, `SetSigmaI0`, `temp9I0`, `SetMethodeI0` e `Send0` em SW	2.173	2.821.743	13.477 (1.247,7 %)	6.738.603

Tabela 4.11: Resultados do particionamento do decodificador LDPC pela versão 1.0. (cont.)

Restrição de período (ms)	Solução	Período obtido (ms)	Memória (*bytes*)	Área equivalente (mm^2)	Custos ($)
3.000	`ChTransmitI0`, `ResetPCI0`, `CalcXorBitI0`, `trieurI0`, `LLRI0`, `CalcNewExtI0`, `AccPariteI0` e `temp6I0` em *HW*	2.252	2.821.743	13.257 (1.225,7 %)	6.628.603
6.000	`ChTransmitI0`, `ResetPCI0`, `CalcXorBitI0`, `trieurI0`, `LLRI0`, `AccPariteI0` e `temp6I0` em *HW*	5.496	2.837	13.229 (1.222,9 %)	6.614.603
7.000	`temp0I0`, `ChTransmitI0`, `ResetPCI0`, `trieurI0`, `LLRI0`, `CalcNewExtI0`, `AccPariteI0` e `temp6I0` em *HW*	6.369	2.618.998	13.300 (1.230 %)	6.650.102
8.000	`ChTransmitI0`, `ResetPCI0`, `trieurI0`, `LLRI0`, `CalcNewExtI0`, `AccPariteI0` e `temp6I0` em *HW*	7.987	2.840.456	13.267 (1.226,7 %)	6.633.603
11.000	`ReceiveIX`, `ChTransmitI0`, `SetPariteI0`, `ResetPCI0`, `trieurI0`, `LLRI0`, `LUTI0`, `AccPariteI0` e `temp6I0` em *HW*	10.970	2.826.558	13.461 (1.246,1 %)	6.730.603
12.000	`ChTransmitI0`, `ResetPCI0`, `trieurI0`, `LLRI0`, `AccPariteI0` e `temp6I0` em *HW*	11.230	2.844.975	13.219 (1.221,9 %)	6.609.603
20.000	`ChTransmitI0`, `ResetPCI0`, `LLRI0`, `AccPariteI0` e `temp6I0` em *HW*	19.029	2.877.397	13.085 (1.208,5 %)	6.542.603
25.000	`ChTransmitI0`, `ResetPCI0`, `trieurI0`, `CalcNewExtI0`, `AccPariteI0` e `temp6I0` em *HW*	22.736	2.890.494	13.229 (1.222,9 %)	6.614.603
30.000	`ChTransmitI0`, `ResetPCI0`, `trieurI0`, `AccPariteI0` e `temp6I0` em *HW*	25.979	2.884.677	13.161 (1.216,1 %)	6.580.603
35.000	`ChTransmitI0`, `ResetPCI0`, `trieurI0`, `CalcNewExtI0`, `LLRI0` e `temp6I0` em *HW*	33.830	2.844.716	13.140 (1.214 %)	6.570.303
40.000	`ChTransmitI0`, `ResetPCI0`, `trieurI0`, `LLRI0` e `temp6I0` em *HW*	37.073	2.849.235	13.092 (1.209,2 %)	6.546.303
45.000	`ResetPCI0`, `trieurI0`, `CalcNewExtI0`, `LLRI0`, `AccPariteI0` e `temp6I0` em *HW*	42.872	2.862.325	13.174 (1.217,4 %)	6.587.103
50.000	`ResetPCI0`, `trieurI0`, `LLRI0`, `AccPariteI0` e `temp6I0` em *HW*	46.116	2.866.844	13.126 (1.212,6 %)	6.563.103

Tabela 4.12: Resultados do particionamento do decodificador LDPC pela versão 1.0. (cont.)

Restrição de período (ms)	Solução	Período obtido (ms)	Memória (*bytes*)	Área equivalente (mm^2)	Custos ($)
55.000	`ResetPCI0`, `LLRI0`, `AccPariteI0` e `temp6I0` em *HW*	53.915	2.899.266	12.992 (1.199,2 %)	6.496.103
60.000	`ResetPCI0`, `trieurI0`, `CalcNewExtI0`, `AccPariteI0` e `temp6I0` em *HW*	57.621	2.912.363	13.136 (1.213,6 %)	6.568.103
65.000	`ResetPCI0`, `trieurI0`, `AccPariteI0` e `temp6I0` em *HW*	60.865	2.906.546	13.068 (1.206,8 %)	6.534.103
70.000	`ResetPCI0`, `trieurI0`, `CalcNewExtI0`, `LLRI0` e `temp6I0` em *HW*	68.716	2.866.585	13.047 (1.204,7 %)	6.523.803
75.000	`ChTransmitI0`, `ResetPCI0`, `CalcXorBitI0`, `trieurI0`, `CalcNewExtI0`, `LLRI0` e `AccPariteI0` em *HW*	74.309	2.846.279	4.080,9 (308,09 %)	2.040.603
80.000	`ChTransmitI0`, `ResetPCI0`, `LUTI0`, `trieurI0`, `CalcNewExtI0`, `LLRI0` e `AccPariteI0` em *HW*	79.943	2.846.638	4.104,9 (310,49 %)	2.052.603
85.000	`ChTransmitI0`, `ResetPCI0`, `trieurI0`, `LLRI0` e `AccPariteI0` em *HW*	83.286	2.869.511	4.042,9 (304,29 %)	2.021.603
95.000	`ChTransmitI0`, `ResetPCI0`, `trieurI0`, `CalcNewExtI0` e `AccPariteI0` em *HW*	94.792	2.915.030	4.052,9 (305,29 %)	2.026.603
100.000	`ChTransmitI0`, `ResetPCI0`, `trieurI0` e `AccPariteI0` em *HW*	98.036	2.909.213	3.984,9 (298,49 %)	1.992.603
110.000	`ChTransmitI0`, `ResetPCI0`, `trieurI0` e `LLRI0` em *HW*	109.130	2.863.435	3.896,2 (289,62 %)	1.948.203
120.000	`AccPariteI0`, `ResetPCI0`, `trieurI0` e `LLRI0` em *HW*	118.172	2.891.380	3.949,9 (294,99 %)	1.975.103
130.000	`AccPariteI0`, `ResetPCI0`, `trieurI0` e `CalcNewExtI0` em *HW*	129.678	2.936.899	3.959,9 (295,99 %)	1.980.103
140.000	`temp0I0`, `ChTransmitI0`, `LLRI0`, `CalcXorBitI0`, `CalcNewExtI0` e `AccPariteI0` em *HW*	139.866	2.680.505	488,72 (-51,128 %)	244.463
150.000	`temp0I0`, `ChTransmitI0`, `LLRI0` e `AccPariteI0` em *HW*	148.843	2.672.729	390,72 (-60,928 %)	195.463
160.000	`ChTransmitI0`, `CalcXorBitI0` e `AccPariteI0` em *HW*	159.476	2.925.512	309,72 (-69,028 %)	154.963
170.000	`temp0I0`, `ChTransmitI0`, `CalcXorBitI0` e `LLRI0` em *HW*	168.953	2.689.284	314 (-68,6 %)	157.103

Tabela 4.13: Resultados do particionamento do decodificador LDPC pela versão 1.0. (cont.)

Restrição de período (ms)	Solução	Período obtido (ms)	Memória (*bytes*)	Área equivalente (mm^2)	Custos ($)
180.000	`ChTransmitIO` e `LLRIO` em *HW*	176.305	2.888.111	211 (-78,9 %)	105.603
190.000	`ChTransmitIO` e `tempOIO` em *HW*	189.436	2.675.347	126 (-87,4 %)	63.103
200.000	`ChTransmitIO` em *HW*	191.054	2.896.805	93 (-90,7 %)	46.603
210.000	`ChTransmitIO` em *HW*	191.054	2.896.805	93 (-90,7 %)	46.603
220.000	`tempOIO` e `CalcXorBitIO` em *HW*	218.588	2.719.847	103 (-89,7 %)	51.603
230.000	Inteiramente em *SW*	225.939	2.918.674	-	103

Tabela 4.14: Resultados do particionamento do decodificador LDPC pela versão 2.0.

Restrições			Solução	Atributos		
Período (ms)	Custo ($)	Potência (mW)		Período (ms)	Custo ($)	Potência (mW)
3,270	5.000	800	Inteiramente em *HW*	3,267	4.545	772,5
10	5.000	800	Inteiramente em *HW*	3,267	4.545	772,5
25	5.000	900	Inteiramente em *HW*	3,267	4.545	772,5
29	5.000	900	`SetSigmaI0` em *SW*	27,320	3.732,1	830,3
100	5.000	900	`SetSigmaI0` em *SW*	27,320	3.732,1	830,3
100	5.000	800	Inteiramente em *HW*	3,267	4.545	772,5
150	5.000	900	`SetSigmaI0` e `temp9I0` em *SW*	143,12	3.390,3	804,69
200	5.000	900	`SetSigmaI0` e `temp9I0` em *SW*	143,12	3.390,3	804,69
250	5.000	900	`SetSigmaI0` e `temp9I0` em *SW*	143,12	3.390,3	804,69
300	5.000	900	`CrChannelI0`, `SetSigmaI0` e `temp9I0` em *SW*	298,075	3.362,7	800
350	5.000	900	`CrChannelI0`, `SetSigmaI0` e `temp9I0` em *SW*	298,075	3.362,7	800
400	4.000	900	`CrChannelI0`, `SetSigmaI0` e `temp9I0` em *SW*	298,075	3.362,7	800
500	4.000	900	`ReceiveIX`, `CrChannelI0`, `SetSigmaI0`, `SetMethodeI0`, `temp9I0` e `Send0` em *SW*	453,27	3.108,3	798,5
600	4.000	900	`ReceiveIX`, `CrChannelI0`, `SetSigmaI0`, `SetMethodeI0`, `temp9I0` e `Send0` em *SW*	453,27	3.108,3	798,5
1.000	4.000	900	`ReceiveIX`, `CrChannelI0`, `SetPariteI0`, `LUTI0`, `SetSigmaI0`, `SetMethodeI0`, `temp9I0` e `Send0` em *SW*	634,645	3.105,54	798,85
2.000	4.000	900	`ReceiveIX`, `CrChannelI0`, `SetPariteI0`, `LUTI0`, `SetSigmaI0`, `SetMethodeI0`, `temp9I0` e `Send0` em *SW*	634,645	3.105,54	798,85
2.100	4.000	900	`ReceiveIX`, `CrChannelI0`, `SetPariteI0`, `LUTI0`, `SetSigmaI0`, `SetMethodeI0`, `temp9I0` e `Send0` em *SW*	634,645	3.105,54	798,85
2.200	4.000	900	`ReceiveIX`, `CrChannelI0`, `SetPariteI0`, `LUTI0`, `SetSigmaI0`, `SetMethodeI0`, `temp9I0` e `Send0` em *SW*	634,645	3.105,54	798,85
3.000	4.000	900	`ReceiveIX`, `CrChannelI0`, `SetPariteI0`, `LUTI0`, `SetSigmaI0`, `SetMethodeI0`, `temp9I0` e `Send0` em *SW*	634,645	3.105,54	798,85
6.000	4.000	800	`ChTransmitI0`, `ResetPCI0`, `CalcXorBitI0`, `trieurI0`, `LLRI0`, `AccPariteI0` e `temp6I0` em *HW*	5.496	3.034,7	785,6
11.000	4.000	800	`ChTransmitI0`, `ResetPCI0`, `CalcXorBitI0`, `trieurI0`, `LLRI0`, `AccPariteI0` e `temp6I0` em *HW*	5.496	3.034,7	785,6

Tabela 4.15: Resultados do particionamento do decodificador LDPC pela versão 2.0 (cont.).

Restrições			Solução	Atributos		
Período (ms)	Custo ($)	Potência (mW)		Período (ms)	Custo ($)	Potência (mW)
12.000	4.000	800	`temp0I0`, `ChTransmitI0`, `ResetPCI0`, `LLRI0`, `CalcXorBitI0`, `AccPariteI0` e `temp6I0` em *HW*	11.677	2.936	751,62
20.000	4.000	800	`ChTransmitI0`, `ResetPCI0`, `LLRI0`, `AccPariteI0` e `temp6I0` em *HW*	19.029	2.765	744,5
30.000	4.000	800	`ChTransmitI0`, `ResetPCI0`, `LLRI0`, `AccPariteI0` e `temp6I0` em *HW*	19.029	2.765	744,5
40.000	4.000	800	`ChTransmitI0`, `ResetPCI0`, `LLRI0`, `AccPariteI0` e `temp6I0` em *HW*	19.029	2.765	744,5
50.000	4.000	800	`ChTransmitI0`, `ResetPCI0`, `LLRI0`, `AccPariteI0` e `temp6I0` em *HW*	19.029	2.765	744,5
60.000	4.000	800	`ResetPCI0`, `LLRI0`, `AccPariteI0` e `temp6I0` em *HW*	53.915	2.664	739,25
70.000	4.000	800	`ResetPCI0`, `LLRI0`, `AccPariteI0` e `temp6I0` em *HW*	53.915	2.664	739,25
80.000	4.000	800	`ChTransmitI0`, `ResetPCI0`, `CalcXorBitI0`, `trieurI0`, `CalcNewExtI0`, `LLRI0` e `AccPariteI0` em *HW*	74.309	1.801,6	746,8
90.000	4.000	800	`ChTransmitI0`, `ResetPCI0`, `trieurI0`, `LLRI0` e `AccPariteI0` em *HW*	83.286	1.682,7	739,2
100.000	4.000	800	`ChTransmitI0`, `ResetPCI0`, `trieurI0` e `AccPariteI0` em *HW*	98.036	1.679	722,4
110.000	4.000	800	`ChTransmitI0`, `ResetPCI0`, `trieurI0` e `LLRI0` em *HW*	109.130	1.595	741,9
120.000	4.000	800	`ChTransmitI0`, `ResetPCI0`, `trieurI0` e `LLRI0` em *HW*	109.130	1.595	741,9
130.000	4.000	800	`ChTransmitI0`, `ResetPCI0`, `trieurI0` e `LLRI0` em *HW*	109.130	1.595	741,9
140.000	4.000	800	`temp0I0`, `ChTransmitI0`, `LLRI0`, `AccPariteI0`, `CalcXorBitI0` e `CalcNewExtI0` em *HW*	139.866	616,2	742,73
150.000	3.000	800	`temp0I0`, `ChTransmitI0`, `LLRI0` e `AccPariteI0` em *HW*	148.843	480	727,7
160.000	3.000	800	`temp0I0`, `ChTransmitI0`, `CalcXorBitI0` e `AccPariteI0` em *HW*	159.476	372,2	707,2
170.000	3.000	800	`temp0I0`, `ChTransmitI0`, `CalcXorBitI0` e `AccPariteI0` em *HW*	159.476	372,2	707,2
180.000	3.000	800	`ChTransmitI0` e `LLRI0` em *HW*	176.305	368,55	723

Tabela 4.16: Resultados do particionamento do decodificador LDPC pela versão 2.0 (cont.).

Restrições			Solução	Atributos		
Período (ms)	Custo ($)	Potência (mW)		Período (ms)	Custo ($)	Potência (mW)
190.000	3.000	800	`ChTransmitI0` e `temp0I0` em *HW*	189.436	264,3	701,7
200.000	3.000	800	`ChTransmitI0` em *HW*	191.054	232,3	699,4
220.000	3.000	800	`ChTransmitI0` em *HW*	191.054	232,3	699,4
230.000	3.000	100	Inteiramente em *SW*	225.939	103	62

Capítulo 5

Conclusões

O objetivo desta tese de doutorado foi desenvolver um algoritmo de particionamento *HW/SW* aprimorado, por meio da integração de conhecimentos produzidos em pesquisas anteriores sobre o tema, o que se constitui em uma estratégia de pesquisa ainda não explorada na literatura, conforme revelado pelo levantamento bibliográfico realizado. A análise de diversos algoritmos de particionamento já publicados [23] levou à determinação de critérios de qualidade para essa classe de programas, e de três otimizações relevantes a serem implantadas e avaliadas em um algoritmo de particionamento escolhido como referência. Esse algoritmo de base, proposto por H. Oh e S. Ha [16] e integrado à ferramenta de *Codesign* PEaCE [17], foi explicado em detalhes na Seção 2.4.3. Para que fosse possível desenvolver um trabalho autônomo e concentrado exclusivamente no algoritmo, seu código-fonte foi desmembrado da ferramenta PEaCE e migrado para o ambiente Microsoft Visual C++. Vários erros de lógica foram encontrados nesse código, os quais precisaram ser primeiramente corrigidos para atingir-se um nível mínimo de funcionalidade, que permitisse a implementação e avaliação das otimizações propostas.

Tendo sido corrigidos os erros e principais deficiências encontrados no programa de particionamento de base, pôde-se finalmente executá-lo corretamente, e ele recebeu a denominação de "versão 1.0". Passou-se então às sucessivas implementações das otimizações propostas nesta tese para a versão 1.0, as quais são descritas detalhadamente no Capítulo 3 e consistem em:

1. Considerar de forma precisa as restrições de processadores de *software* e de *hardware* e os custos de nós implementados nesses processadores. Para um FPGA, as quantidades totais disponíveis de cada tipo de seus recursos internos (por exemplo, LE's, *bits* de RAM, blocos de DSP, PLL's) são especificadas como as restrições para implementações de nós nesse FPGA. Os custos dos nós em *hardware* podem ser calculados em função da área de circuito ocupada pelo nó, no caso de um ASIC, ou

das quantidades de recursos internos de um FPGA ocupadas pelo nó. Os consumos de potência de nós, em *software* e *hardware*, também passam a ser especificados e considerados nas decisões de particionamento.

2. Definir novas restrições para modos e tarefas, visando melhor definir o espaço de soluções válidas do particionamento nos eixos de desempenho, custo e potência. Essas novas restrições consistem dos valores máximos permitidos pela aplicação para o tempo de execução total, o custo monetário e o consumo de potência do sistema. Incorporar então ao algoritmo de particionamento uma função de custo que integre os novos custos e restrições em uma mesma expressão, e cujo objetivo possa ser regulado por meio de valores de pesos. Neste trabalho, o objetivo do particionamento foi sempre a minimização do custo do sistema final.

3. Considerar os tempos de reconfiguração de FPGA's dinamicamente reconfiguráveis no agendamento, capacitando o algoritmo para o particionamento de sistemas com *hardware* fixo e com *hardware* dinamicamente reconfigurável.

4. Adaptar o algoritmo para escolher dentre múltiplas alternativas de implementação para um mesmo nó em um mesmo PE.

Para os testes das quatro versões do algoritmo de particionamento, foram utilizadas duas aplicações de processamento de sinais: um receptor de rádio digital *Rake* e um decodificador de canal LDPC. Foram abordados os aspectos da criação dos arquivos de entrada para o particionamento, com ênfase na obtenção dos valores de desempenho, ocupação de recursos e consumo de potência de cada nó de aplicação sobre uma plataforma heterogênea, no caso a placa Nios II / Stratix II da Altera. A caracterização dos nós das aplicações que se deseja particionar, por meio da determinação desses valores essenciais para o particionamento, constitui-se atualmente na principal dificuldade para a utilização do algoritmo proposto. Entretanto, os métodos adotados para caracterizar o *Rake* e o LDPC são generalizáveis para quaisquer outras aplicações, e para outras plataformas da Altera.

Os resultados do particionamento dessas aplicações foram apresentados no Capítulo 4, comparando-se seus valores de desempenho e custo com os de suas implementações inteiramente em *software* e em *hardware*. A versão 2.0 do particionamento é eficaz em minimizar o custo das soluções heterogêneas propostas, atendendo simultaneamente às restrições de tempo de execução e potência impostas à aplicação pelo usuário. Além disso, a consideração das quantidades de recursos internos do FPGA ocupadas por cada nó permite ter-se uma idéia precisa da ocupação final do FPGA por uma partição qualquer. A versão 3.0 acompanha a ocupação do FPGA durante o particionamento e, caso seja

necessário, agenda uma reconfiguração dinâmica do FPGA a ser feita durante a operação do sistema. A versão 4.0 flexibiliza a escolha do nó a ser mapeado em um PE, selecionando a alternativa de implementação mais conveniente para os objetivos da função de custo.

A hipótese principal deste trabalho foi assim demonstrada, segundo a qual seria possível integrar características vantajosas específicas de algoritmos preexistentes a um algoritmo de base, reutilizando-as de uma forma cumulativa e compatível entre si, para o desenvolvimento de um algoritmo de particionamento único e de melhor qualidade. Essa prática levou a melhoramentos de funcionalidade no algoritmo, fornecendo claramente um melhor controle sobre os sistemas heterogêneos resultantes do particionamento, que potencialmente terão melhor desempenho e menor custo do que aqueles resultantes do algoritmo de base. A outra hipótese, que também foi comprovada, é a de que seria possível criar um único algoritmo aplicável ao particionamento de sistemas com *hardware* fixo e com *hardware* reconfigurável, que decidisse pela exploração ou não da reconfigurabilidade de *hardware*, conforme fosse mais vantajoso.

Os resultados obtidos no Capítulo 4, para as duas aplicações de teste utilizadas, sugerem a compatibilidade entre o algoritmo escolhido como ponto de partida para o trabalho e as três técnicas de otimização do particionamento nele realizadas. A proposta, a implementação e a avaliação do efeito das três otimizações propostas, realizadas no algoritmo de particionamento original, constituem-se na contribuição desta tese de doutorado.

Este trabalho deixa várias perspectivas promissoras de continuação, em diversas direções. Algumas dessas possibilidades são:

1. Implementar as aplicações particionadas obtidas neste trabalho na placa Nios II / Stratix II da Altera, para realimentar possíveis correções e aprimoramentos ao algoritmo de particionamento em termos do modelamento da sincronização entre as partições de *hardware* e *software*.

2. Implementar uma interface gráfica com o usuário que facilite a especificação do grafo da aplicação, dos valores dos nós e das restrições, de modo a agilizar a aplicação do algoritmo a problemas práticos.

3. Implementar uma saída gráfica, que mostre o resultado de um agendamento por meio de um diagrama temporal da execução dos nós, e que também forneça diretamente os gráficos do tempo de execução, custo e potência em função das restrições para cada particionamento.

4. Implementar a possibilidade de exploração do espaço de projeto de uma aplicação, por meio da varredura automática dos valores das restrições de uma situação inicial a uma situação final, com a realização de um particionamento para cada situação

(conjunto de valores). Isso permitirá a obtenção de vários particionamentos, de forma bastante rápida, cujos parâmetros poderão então ser exibidos graficamente por meio da saída gráfica do item anterior.

5. Acrescentar mais informações quantitativas ao relatório de custo fornecido, que informem as diferenças percentuais entre os resultados e suas restrições correspondentes, o que seria útil para a comparação entre as qualidades de soluções obtidas para diferentes restrições.

6. Integrar outros algoritmos de particionamento, em especial algoritmos para grafos cíclicos, dando ao usuário a possibilidade de escolher qual algoritmo utilizar para cada problema.

7. Determinar outras características de qualidade que se revelem adequadas a serem integradas ao algoritmo desenvolvido neste trabalho, além das três que foram implementadas.

8. Estender o algoritmo de particionamento proposto para poder trabalhar com FPGA's dotados de reconfiguração dinâmica parcial, ou seja, a reconfiguração de nós individuais e não necessariamente do FPGA inteiro. O algoritmo poderá então explorar os pontos em comum entre nós candidatos ao FPGA para reduzir a quantidade de dados de reconfiguração necessários, e portanto reduzir o tempo necessário para cada reconfiguração, bem como a quantidade de reconfigurações [46, 51].

Referências Bibliográficas

[1] R. K. Gupta, G. DeMicheli. "System-level synthesis using re-programmable components". In *Proceedings of the 3rd. European Conference on Design Automation (EuroDAC'92)*, pp. 2–7, Março 1992.

[2] R. K. Gupta, G. DeMicheli. "Hardware-software cosynthesis for digital systems". *IEEE Design and Test of Computers*, vol. 10, n° 3, pp. 29–41, Setembro 1993.

[3] F. Rousseau, J-M. Berge, M. Israel. "Synthèse des Méthodes et Algorithmes de Partitionnement Logiciel/Matériel". In *Actes du 3ème Symposium sur les Architectures Nouvelles de Machines*, Rennes, França, 1995.

[4] R. Ernst, J. Henkel, T. Benner. "Hardware-software cosynthesis for microcontrollers". *IEEE Design and Test of Computers*, vol. 10, n° 4, pp. 64–75, Dezembro 1993.

[5] E. N. S. Barros, W. Rosenstiel. "A Method for Hardware Software Partitioning". In *Proceedings of COMPEURO'92*, pp. 580–585, The Hague, Holanda, Maio 1992.

[6] E. N. S. Barros, W. Rosenstiel, X. Xiong. "A Method for Partitioning UNITY Language in Hardware and Software". In *Proceedings of the European Design Automation Conference*, pp. 220–225, Grenoble, França, Setembro 1994.

[7] A. Kalavade, E. A. Lee. "A Global Criticality / Local Phase Driven Algorithm for the Constrained Hardware/Software Partitioning Problem". In *Proceedings of the 3rd International Workshop on Hardware/Software Codesign (CODES'94)*, pp. 42–48, Roma, Itália, Setembro 1994.

[8] A. Kalavade, E. A. Lee. "The Extended Partitioning Problem: Hardware/Software Mapping and Implementation-bin Selection". In *Proceedings of the 6th IEEE International Workshop on Rapid System Prototyping*, pp. 12–18, Junho 1995.

[9] J. Madsen et al. "LYCOS: the Lyngby Co-Synthesis System". *Design Automation for Embedded Systems*, vol. 2, n° 2, pp. 195–236, 1997.

[10] J. Jeon, K. Choi. "An Effective Force-Directed Partitioning Algorithm for Hardware/Software Codesign". Relatório técnico, Seoul National University, Seoul, Coréia, Maio 1997.

[11] J. Jeon, K. Choi. "Loop Pipelining in Hardware-Software Partitioning". In *Proceedings of the Asia South Pacific Design Automation Conference (ASPDAC'98)*, pp. 361–366, Fevereiro 1998.

[12] F. Rousseau, J-M. Berge, M. Israel. "Hardware/Software Partitioning for Telecommunications Systems". In *Proceedings of the 20th International Computer Software and Applications Conference (COMPSAC '96)*, pp. 483–488, Agosto 1996.

[13] J. Hou, W. Wolf. "Process Partitioning for Distributed Embedded Systems". In *Proceedings of the 4th International Workshop on Hardware-Software Codesign (CODES/CASHE'96)*, pp. 70–76, Março 1996.

[14] H. Oh, S. Ha. "A Hardware-Software Cosynthesis Technique Based on Heterogeneous Multiprocessor Scheduling". In *Proceedings of the 7th International Workshop on Hardware/Software Codesign (CODES'99)*, pp. 183–187, Roma, Itália, Março 1999.

[15] A. Kalavade, P. A. Subrahmanyam. "Hardware/Software Partitioning for Multi-Function Systems". In *Proceedings of the 1997 IEEE/ACM International Conference on Computer-Aided Design*, pp. 516–521, Novembro 1997.

[16] H. Oh, S. Ha. "Hardware-Software Cosynthesis of Multi-Mode Multi-Task Embedded Systems with Real-Time Constraints". In *Proceedings of the 10th International Symposium on Hardware-Software Codesign (CODES'02)*, pp. 133–138, Colorado, EUA, Maio 2002.

[17] H. Oh, S. Ha, et al. "PEaCE: Ptolemy Extension as a Codesign Environment", The CAP Laboratory of Seoul National University, Seoul, Coréia, Novembro 2004. http://peace.snu.ac.kr/research/peace/getpeace.php. Válido em 10/08/2003.

[18] H. Oh, S. Ha. "A Static Scheduling Heuristic for Heterogeneous Processors". In *Proceedings of the 2nd International Euro-Par Conference on Parallel Processing*, vol. II, pp. 573–577, Agosto 1996.

[19] M. L. López-Vallejo, J. Grajal, J. C. López. "Constraint-driven System Partitioning". In *Proceedings of Design, Automation And Test In Europe (DATE) Conference and Exhibition*, pp. 411–416, Paris, França, Março 2000.

[20] M. López-Vallejo, J. C. López. "On the hardware-software partitioning problem: System modeling and partitioning techniques". *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 8, n° 3, pp. 269–297, Julho 2003.

[21] D. C. de Souza. "Pesquisa em Projeto de Sistemas Reconfiguráveis Baseados em FPGA, para a Área de Codesign". Relatório técnico, Coordenação de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande (UFCG), Campina Grande (PB), Brasil, Fevereiro 2004. Projeto de Pesquisa orientado por B. G. Aguiar Neto e M. A. de Barros.

[22] Altera Corporation. *"Nios II Software Developer's Handbook, Seção II, Capítulo 4"*, Maio 2006.

[23] D. C. de Souza. "Algoritmo de Particionamento Otimizado para Sistemas Dinamicamente Reconfiguráveis". Relatório técnico, Programa de Pós-Graduação em Engenharia Elétrica, Centro de Engenharia Elétrica e Informática, Universidade Federal de Campina Grande (UFCG), Campina Grande (PB), Brasil, Maio 2006. Exame de Qualificação orientado por B. G. Aguiar Neto e L. A. B. Naviner.

[24] G. DeMicheli, R. K. Gupta. "Hardware/Software Co-Design". *Proceedings of the IEEE, Special Issue on Hardware/Software Co-Design*, vol. 85, n° 3, pp. 349–365, Março 1997.

[25] D. C. de Souza. "Pesquisa sobre Metodologias de Particionamento de Hardware e Software para Codesign no Domínio de Processamento Digital de Sinais". Relatório técnico, Coordenação de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande (UFCG), Campina Grande (PB), Brasil, Outubro 2003. Projeto de Pesquisa orientado por B. G. Aguiar Neto e M. A. de Barros.

[26] D. C. de Souza, M. A. de Barros, L. A. B. Naviner, B. G. Aguiar Neto. "Pesquisa sobre Metodologias de Particionamento de Hardware e Software para Codesign no Domínio de Processamento Digital de Sinais". In *Anais do 1° Encontro Regional em Instrumentação e Metrologia Científica (I ERIMC)*, Campina Grande (PB), Brasil, Dezembro 2003. (em CD-ROM).

[27] D. C. de Souza, M. A. de Barros, L. A. B. Naviner, B. G. Aguiar Neto. "Representação do Conhecimento e Critérios de Qualidade para o Particionamento Hardware/Software". In *Anais do Congresso Brasileiro de Gestão do Conhecimento (KM Brasil 2003)*, São Paulo (SP), Brasil, Novembro 2003. (em CD-ROM).

[28] D. C. de Souza, M. A. de Barros, L. A. B. Naviner, B. G. Aguiar Neto. "Représentation de la Connaissance et Critères de Qualité pour des Méthodes de Partitionnement Optimisées". In *Actes des 7èmes Journées Nationales du Réseau Doctoral de Microélectronique (JNRDM 2004)*, Marseille, França, Maio 2004. (sur CD-ROM).

[29] D. C. de Souza, L. A. B. Naviner, M. A. de Barros, B. G. Aguiar Neto. "Proposition d'un algorithme de partitionnement matériel/logiciel, optimisé pour des systèmes multi-modes reconfigurables". In *Actes des 8èmes Journées Nationales du Réseau Doctoral de Microélectronique (JNRDM 2005)*, Paris, França, Maio 2005.

[30] D. C. de Souza, M. A. de Barros, L. A. B. Naviner, B. G. Aguiar Neto. "On Algorithms and Methodologies for Optimized Partitioning". In *Actes des Journées Scientifiques Francophones (JSF'03)*, Tozeur, Tunísia, Dezembro 2003. (sur CD-ROM).

[31] D. C. de Souza, M. A. de Barros, L. A. B. Naviner, B. G. Aguiar Neto. "On Relevant Quality Criteria for Optimized Partitioning Methods". In *Proceedings of the 46th IEEE International Midwest Symposium on Circuits and Systems (MWSCAS '03)*, Cairo, Egito, Dezembro 2003. (on CD-ROM).

[32] D. C. de Souza, M. A. de Barros, L. A. B. Naviner, B. G. Aguiar Neto. "Knowledge Representation and Quality Criteria for Optimized Partitioning Methods". In *Proceedings of the 11th International Conference on Mixed Design of Integrated Circuits and Systems (MIXDES '04)*, Szczecin, Polônia, Junho 2004.

[33] D. C. de Souza, I. Krikidis, L. Naviner, J.-L. Danger, M. A. de Barros, B. G. Aguiar Neto. "Implementation of a Digital Receiver for DS-CDMA Communication Systems Using HW/SW Codesign". In *Proceedings of the 48th IEEE International Midwest Symposium on Circuits and Systems (MWSCAS '05)*, Cincinatti, EUA, Agosto 2005. (on CD-ROM).

[34] D. C. de Souza, I. Krikidis, L. Naviner, J.-L. Danger, M. A. de Barros, B. G. Aguiar Neto. "Heterogeneous Implementation of a Rake Receiver for DS-CDMA Communication Systems". In *Proceedings of the 12th IEEE International Conference on Electronics, Circuits and Systems (ICECS 2005)*, vol. 2, pp. 450–453, Gammarth, Tunísia, Dezembro 2005.

[35] H. Oh, S. Ha. "Partitioning Framework for Less Restricted Partitioning Problems". In *Proceedings of the 6th International Conference on VLSI and CAD*, Seoul, Coréia, Outubro 1999.

[36] E. Barros, S. Cavalcante, M. E. de Lima, C. A. Valderrama. *"Hardware/Software Co-design: Projetando Hardware e Software Concorrentemente"*. Escola de Computação de 2000, IME/USP, São Paulo (SP), Brasil, Julho 2000.

[37] P. Lacomme, C. Prins, M. Sevaux. *"Algorithmes de Graphes"*. Editions Eyrolles, Paris, França, 2ème édition, 2003.

[38] A. Azzedine, J.-P. Diguet, J.-L. Pillippe. "Large Exploration for HW/SW Partitioning of Multirate and Aperiodic Real-Time Systems". In *Proceedings of the 10th International Symposium on Hardware/Software Codesign (CODES'02)*, pp. 85–90, Colorado, EUA, Maio 2002.

[39] F. Vahid, J. Gong, D.D. Gajski. "A Binary-Constraint Search Algorithm for Minimizing Hardware During Hardware/Software Partitioning". In *Proceedings of the European Design Automation Conference (EuroDAC)*, pp. 214–219, Setembro 1994.

[40] S. Prakash, A. C. Parker. "Synthesis of Application-Specific Multiprocessor Architectures". In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pp. 8–13, Junho 1991.

[41] S. Prakash, A. C. Parker. "A Design Method for Optimal Synthesis of Application-Specific Heterogeneous Multiprocessor Systems". In *Proceedings of the Workshop on Heterogeneous Processing*, pp. 75–80, Março 1992.

[42] J. Noguera, R. M. Badia. "Run-time HW/SW Codesign for Discrete Event Systems Using Dynamically Reconfigurable Architectures". In *Proceedings of the 13th International Symposium on System Synthesis*, pp. 100–106, Setembro 2000.

[43] J. Noguera, R. M. Badia. "A HW/SW partitioning algorithm for dynamically reconfigurable architectures". In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, pp. 729–734, Março 2001.

[44] J. Noguera, R. M. Badia. "Dynamic Run-Time HW/SW Scheduling Techniques for Reconfigurable Architectures". In *Proceedings of the 10th International Symposium on Hardware-Software Codesign (CODES'02)*, pp. 205–210, Maio 2002.

[45] J. Noguera, R. M. Badia. "HW/SW Codesign Techniques for Dynamically Reconfigurable Architectures". *IEEE Transactions on Very Large Scale Integration Systems*, vol. 10, n° 4, pp. 399–415, Agosto 2002.

[46] K. Choi et al. "Hardware-Software Cosynthesis for Run-time Incrementally Reconfigurable FPGAs". In *Proc. of the Asia South Pacific Design Automation Conference (ASPDAC'00)*, pp. 169–174, Janeiro 2000.

[47] A. S.-Vincentelli et al. "HW/SW Partitioning and Code Generation of Embedded Control Applications on a Reconfigurable Architecture Platform". In *Proceedings of the 10th International Symposium on Hardware-Software Codesign (CODES'02)*, pp. 151–156, Maio 2002.

[48] R. P. Dick, N. K. Jha. "CORDS: Hardware-Software Co-Synthesis of Reconfigurable Real-Time Distributed Embedded Systems". In *Digest of Technical Papers of the 1998 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'98)*, pp. 62–68, Novembro 1998.

[49] B. P. Dave. "CRUSADE: Hardware/Software Co-Synthesis of Dynamically Reconfigurable Heterogeneous Real-Time Distributed Embedded Systems". In *Proc. of Design, Automation and Test in Europe, Conference and Exhibition*, pp. 97–104, Março 1999.

[50] D. N. Rakhmatov, S. B. K. Vrudhula. "Hardware-Software Bipartitioning for Dynamically Reconfigurable Systems". In *Proceedings of the 10th International Symposium on Hardware-Software Codesign (CODES'02)*, pp. 145–150, Maio 2002.

[51] J. Harkin, T. M. McGinnity, L. P. Maguire. "Partitioning methodology for dynamically reconfigurable embedded systems". *IEE Computers and Digital Techniques*, vol. 147, n° 6, pp. 391–396, Novembro 2000.

[52] K. S. Chatha, R. Vemuri. "Hardware-Software Codesign for Dynamically Reconfigurable Architectures". In *Proceedings of the 9th International Workshop on Field Programmable Logic and Applications (FPL'99)*, pp. 175–184, Setembro 1999.

[53] J. B. Peterson, R. B. O'Connor, P. M. Athanas. "Scheduling and Partitioning ANSI-C Programs onto Multi-FPGA CCM Architectures". In *Proceedings of the IEEE Symposium on FPGAs for Custom Configurable Computing Machines*, pp. 178–187, Abril 1996.

[54] The CAP Laboratory of Seoul National University. *"PEaCE User's Manual"*. Seoul, Coréia, Novembro 2004. http://peace.snu.ac.kr.

[55] Altera Corporation. *"Stratix II Device Handbook, Volumes 1 e 2"*, Abril 2006.

[56] Altera Corporation. http://www.altera.com/products/, Design Software.

[57] I. Krikidis, J.-L. Danger, L. A. B. Naviner. "Flexible and reconfigurable receiver architecture for WCDMA systems with low spreading factors". *IEE Electronics Letters*, vol. 41, n° 1, pp. 22–24, Janeiro 2005.

[58] H. Oh, S. Ha. "Fractional Rate Dataflow Model and Efficient Code Synthesis for Multimedia Applications". In *Proceedings of the (LCTES'02 - SCOPES'02)*, Junho 2002.

[59] F. Guilloud. "Architecture générique de décodeurs de codes LDPC", Paris, França, 2004. Tese de doutorado, disponível em http://pastel.paristech.org/bib/archive/00000806.