MAPEAMENTO DE INSTRUÇÕES *DATAFLOW*

Alexandre Ferreira Sardinha de Mattos

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientadores: Felipe Maia Galvão França
Valmir Carneiro Barbosa

Rio de Janeiro
Junho de 2012

MAPEAMENTO DE INSTRUÇÕES *DATAFLOW*

Alexandre Ferreira Sardinha de Mattos

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof. Felipe Maia Galvão França, Ph.D.

Prof. Valmir Carneiro Barbosa, Ph.D.

Prof. Ricardo Cordeiro de Farias, Ph.D.

Prof. Fabio Protti, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
JUNHO DE 2012

Mattos, Alexandre Ferreira Sardinha de

Mapeamento de instruções *Dataflow*/Alexandre Ferreira Sardinha de Mattos. – Rio de Janeiro: UFRJ/COPPE, 2012.

XIV, 83 p.: il.; 29, 7cm.

Orientadores: Felipe Maia Galvão França
Valmir Carneiro Barbosa

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2012.

Referências Bibliográficas: p. 74 – 76.

1. *Dataflow*. 2. Mapeamento. 3. Simulador. I. França, Felipe Maia Galvão *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

"Se o malandro soubesse o
quanto vale a honestidade, ele
seria honesto por malandragem."

Agradecimentos

Agradeço a minha mãe Josefina e meu pai Manoel José, pelo grande exemplo que sempre foram e por tudo que fizeram por mim. Também gostaria de agradecer a Lúcia e família por todo o carinho que tem tido comigo.

Agradeço aos meus amigos que me apoiaram e souberam compreender quando não pude estar presente: Alexandre Nascimento, Alexandre Faria, Clarice Sá, Eduardo Ambrósio, Gabriel Dottori, Leandro Pose, Márcio Taddei, Nanna Mabelle, Raphael Martins, Renato Aranha, Tatiana Hessab, Thaian Moraes, Thiago Silva, Wagner Fortes, Wando Fortes e Warny Marçano. Um agradecimento especial a minha namorada Thatiana Pinto que me incentivou bastante e sempre foi compreensiva.

Ao professor Ageu Cavalcanti que me apresentou à Arquitetura de Computadores e ao professor Gabriel Silva, que foi decisivo para que eu seguisse esta linha no mestrado. Também agradeço a Sérgio Guedes e Juliana Correa pelo que aprendi no cluster Netuno.

Ao pessoal da Petrobras: Fernando, Luís Felipe, Sandoval e Verônica, que me auxiliaram a conciliar o mestrado com os meus compromissos profissionais.

Ao Programa de Engenharia de Sistemas e Computação (PESC/COPPE/UFRJ), pela excelente estrutura que proporciona a seus alunos. Ao CNPq e a FAPERJ pelas bolsas de estudo concedidas. E ao povo brasileiro que financia estas instituições.

Aos amigos que conheci na graduação: Alex Sanches, Anselmo Luiz, Bruno "Codorna", Douglas Cardoso, Eliza França, Renan Baqui, Thiago "Escazi" e Wendel Melo. Especialmente ao companheiro de todas as horas, Rafael Espirito Santo.

Ao pessoal do mestrado Hugo Nobrega, Rodrigo Nürmberg e Larissa Spinelli. Ao pessoal do LAM: Saulo Tavares, Fábio Freitas, Flávio Faria, Bruno França, Fábio Flesch, Alexandre Nery, Lawrence Bandeira. Especialmente aos "pastores da Igreja Dataflow", Tiago Alves e Leandro Marzulo. Tiago foi fundamental nesta dissertação, desde a sugestão de literatura, auxiliando no uso do compilador, até a revisão deste trabalho. Um agradecimento especial também a Lúcio Paiva, pois o que seria de Jay sem Silent Bob?

Ao professor Felipe França, pelo apoio incondicional e pela compreensão mesmo

quando me tornei aluno de tempo parcial. Ao professor Valmir que me “adotou” durante a dissertação e cuja contribuição foi imprescindível neste trabalho. Ambos foram extremamente atenciosos e solícitos comigo. Sinto-me privilegiado por terem sido meus orientadores e sou eternamente grato aos dois. Aproveito para desejar pronta recuperação ao professor Felipe.

Agradeço também aos professores Fabio Protti e Ricardo Farias pela gentileza de participar da banca de defesa de mestrado.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

MAPEAMENTO DE INSTRUÇÕES *DATAFLOW*

Alexandre Ferreira Sardinha de Mattos

Junho/2012

Orientadores: Felipe Maia Galvão França
Valmir Carneiro Barbosa

Programa: Engenharia de Sistemas e Computação

O limite de dissipação de energia térmica de um *chip* fez com que as frequências dos processadores não pudessem aumentar como o esperado. Então, a indústria teve de buscar alternativas para garantir o aumento de performance computacional e passou a investir em processadores com múltiplos núcleos em um mesmo *chip*.

Para aproveitar todo o potencial dos processadores *multicores*, surge a necessidade da adoção de modelos alternativos de programação paralela, como é o caso do modelo *Dataflow*. A característica principal nesta arquitetura é que as instruções podem ser executadas assim que seus operandos estão disponíveis. Portanto, o programa segue o fluxo de dados ao invés de seguir a ordem sequencial do código (modelo Von Neumann).

No contexto das arquiteturas *Dataflow*, existe o problema de como mapear instruções para os Elementos de Processamento. Isto é, dado um conjunto de instruções I, um grafo direcionado $G(I, E)$, que descreve as dependências entre instruções, e um conjunto de Elementos de Processamento P, qual deve ser a maneira de distribuir (mapear) as instruções nos Elementos de Processamento, de modo a tentar minimizar o tempo total de execução do programa *Dataflow*. É este o problema que nos dedicamos a estudar nesta dissertação.

Neste trabalho, implementamos um Simulador Arquitetural *Dataflow*, a nível de ciclo, para servir de ambiente de testes e auxiliar no estudo dos efeitos do mapeamento na execução de um programa. Além disso, propomos um Algoritmo de Mapeamento a partir de uma técnica de programação dinâmica com algumas melhorias. Por fim, implementamos um conjunto de programas de teste para servir de *benchmarking* e apresentamos um estudo comparativo entre o algoritmo proposto e outros algoritmos de mapeamento.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

DATAFLOW INSTRUCTIONS PLACEMENT

Alexandre Ferreira Sardinha de Mattos

June/2012

Advisors: Felipe Maia Galvão França
Valmir Carneiro Barbosa

Department: Systems Engineering and Computer Science

The limit of thermal energy dissipation on a chip meant that the processors frequencies could not increase as expected. Thus, the industry had to seek alternatives to ensure the increase in computational performance and spent investing in multicore processors on a single chip.

To harness the full potential of multicore processors, it becomes necessary to adopt alternative models of parallel programming, such as Dataflow model. The main feature of this architecture is that the instructions can be performed as soon as their operands are available. Therefore, the program follows the data flow rather than following the code sequentially (Von Neumann model).

In Dataflow architectures context, there is the problem of how to place instructions to the Processing Elements. That is, given an instruction set I, a directed graph $G(I, E)$, which describes the dependencies between instructions, and a set of Processing Elements P, which should be a way to distribute (place) instructions in the Processing Elements, to try to minimize the total execution time of the Dataflow program. This is the problem we dedicated to study in this master thesis.

In this work, we implemented an Architectural Dataflow Simulator, at cycle level, to serve as test environment and assist to study the placement effects in program execution. Furthermore, we propose a placement algorithm from a dynamic programming technique with some improvements. Finally, we implemented a test-suite programs to serve as a benchmarking and present a comparative study between the proposed algorithm and other placement algorithms.

Sumário

Lista de Figuras **xi**

Lista de Tabelas **xiv**

1 Introdução **1**

2 Motivação e Trabalhos Relacionados **8**

2.1 Motivação 8

2.2 Trabalhos Relacionados 9

2.2.1 Arquitetura WaveScalar 9

2.2.2 Máquina Virtual *Trebuchet* 10

2.2.3 Compilador *Couillard* 11

2.2.4 Mapeamento na WaveScalar 11

2.2.5 Algoritmo de escalonamento de tarefas 12

3 Simulador Arquitetural *Dataflow* **13**

3.1 Visão Geral 14

3.2 Instruções 14

3.3 *Loader* 17

3.4 *Instruction List* 19

3.5 Rede de Interconexão 20

3.6 *Buffer* de Entrada 20

3.7 *Matching Table* 21

3.8 Execução 23

3.9 Utilitários 23

4 Algoritmo de Mapeamento **24**

4.1 Definição do Problema 24

4.2 Primeiras Tentativas 26

4.2.1 Escalonamento por Reversão de Arestas 26

4.2.2 *List Scheduling* 26

4.3 Programação Dinâmica 27

4.4 Adaptações 33
4.4.1 Heurísticas de Ordenação 33
4.4.2 Comunicação 34
4.4.3 Componentes Fortemente Conexas 34
4.4.4 Tempo de Execução Personalizado 44
4.5 O Algoritmo 46

5 Experimentos e Resultados **50**
5.1 Metodologia 50
5.2 Algoritmos 52
5.2.1 Programação Dinâmica 52
5.2.2 Componentes Fortemente Conexas 52
5.2.3 Tempos de Execução Personalizados 52
5.2.4 *Static-snake* 53
5.2.5 *Depth-first-snake* 53
5.2.6 *Breadth-first-snake* 53
5.2.7 Sem comunicação inter-EP 54
5.3 *Benchmarking* 54
5.4 Resultados 61
5.5 Efeitos da Fila de Operandos 65

6 Conclusões e Trabalhos Futuros **71**

Referências Bibliográficas **74**

A Aplicações **77**

Lista de Figuras

1.1 O "primeiro" programa *Dataflow* descrito no artigo de Dennis e Misunas [1]. 2
1.2 Exemplo de mapeamento em 6 Elementos de Processamento. Cada conjunto de instruções dentro dos retângulos é alocado a um EP distinto. 4
1.3 Os dois extremos do espectro no mapeamento: todas instruções em um mesmo EP (a); uma instrução em cada EP (b). 6

2.1 A hierarquia de Elementos de Processamento da WaveScalar [2]. . . . 10

3.1 O pipeline de um Elemento de Processamento no Simulador Arquitetural. 15
3.2 Representação da instrução condicional *Steer*. 16
3.3 Exemplo de um arquivo de entrada do Simulador. 18
3.4 Representação do Grafo *Dataflow* descrito no arquivo de entrada da Figura 3.3. 19
3.5 Exemplo de comunicação inter-EP, utilizando a Rede de Interconexão. 21
3.6 Trace gerado pelo Simulador ao executar o programa da Figura 3.5. . 22

4.1 Exemplo de Grafo *Dataflow* a ser mapeado utilzando o Algoritmo 4.1. 29
4.2 Iteração 1 da execução do Algoritmo 4.1. 29
4.3 Iteração 2 da execução do Algoritmo 4.1. 30
4.4 Iteração 3 da execução do Algoritmo 4.1. 31
4.5 Iteração 4 da execução do Algoritmo 4.1. 32
4.6 Estado final da execução do Algoritmo 4.1. 32
4.7 Exemplo de Grafo *Dataflow* a ser mapeado utilzando o algoritmo de mapeamento considerando a latência de comunicação L.. 35
4.8 Iteração 1 da execução do algoritmo de mapeamento considerando a latência de comunicação L. 35
4.9 Iteração 2 da execução do algoritmo de mapeamento considerando a latência de comunicação L. 36
4.10 Iteração 3 da execução do algoritmo de mapeamento considerando a latência de comunicação L. 37

4.11 Iteração 4 da execução do algoritmo de mapeamento considerando a latência de comunicação L. 38
4.12 Iteração 5 da execução do algoritmo de mapeamento considerando a latência de comunicação L. 39
4.13 Estado final da execução do algoritmo de mapeamento considerando a latência de comunicação L. 40
4.14 Exemplo de transformação do grafo *Dataflow* original em grafo acíclico através do agrupamento de CFCs. 43
4.15 Exemplo onde o somatório do tempos execução de cada instrução não é uma boa medida para o cálculo do *makespan* da CFC sucessora. . . 44
4.16 Exemplo de grafo onde os Tempos de Execução Personalizados serão calculados. 46
4.17 Cálculo dos Tempos de Execução Personalizados do grafo da Figura 4.16. 47

5.1 Fluxo executado para avaliação de um programa em nossa plataforma de testes. 51
5.2 Exemplo de mapeamentos gerados através do algoritmos. 54
5.3 Bloco básico acíclico. 56
5.4 Bloco básico com ciclo. 57
5.5 Bloco básico com ciclo aninhado. 57
5.6 Programa com 4 blocos básicos em paralelo. 58
5.7 Programa com 4 blocos básicos de maneira serial. 58
5.8 Programas com blocos de maneira serial e em paralelo. 59
5.9 Programa misto, com a presença de cada um dos blocos básicos. . . . 59
5.10 Tempo de execução dos programas acíclicos com latência de comunicação de 5 ciclos. 65
5.11 Tempo de execução dos programas com ciclo com latência de comunicação de 5 ciclos. 65
5.12 Tempo de execução dos programas com ciclo aninhado e o programa misto, com latência de comunicação de 5 ciclos. 66
5.13 Tempo de execução dos programas acíclicos com latência de comunicação de 10 ciclos. 66
5.14 Tempo de execução dos programas com ciclo com latência de comunicação de 10 ciclos. 67
5.15 Tempo de execução dos programas com ciclo aninhado e o programa misto, com latência de comunicação de 10 ciclos. 67
5.16 Tempo de execução dos programas acíclicos com latência de comunicação de 15 ciclos. 68

5.17 Tempo de execução dos programas com ciclo com latência de comunicação de 15 ciclos. 68
5.18 Tempo de execução dos programas com ciclo aninhado e o programa misto, com latência de comunicação de 15 ciclos. 69
5.19 Exemplo de trace de mapeamento do programa `acíclico`. 69
5.20 Exemplo de trace execução do programa `acíclico`. 70
5.21 Trace do EP0 duranto o ciclo 7 do programa `acíclico`. 70

Lista de Tabelas

3.1 Instruções implementadas no Simulador. 16

5.1 Sumário dos programas utilizados no *benchmarking*. 60
5.2 Tempos de execução dos programas (em ciclos), utilizando a latência de comunicação $L = 5$ ciclos. 62
5.3 Tempos de execução dos programas (em ciclos), utilizando a latência de comunicação $L = 10$ ciclos. 63
5.4 Tempos de execução dos programas (em ciclos), utilizando a latência de comunicação $L = 15$ ciclos. 64

Capítulo 1

Introdução

A partir da década de 1990, o aumento de performance computacional foi alcançado, em grande parte, devido a avanços na tecnologia do silício. Estes avanços permitiam que a performance dos processadores dobrasse a cada 18 meses [3]. Em 2005, as previsões eram de que as frequências dos processadores chegassem a 10 GHz em 2008 e 15 GHz em 2010. No entanto, o limite de dissipação de energia térmica de um *chip* foi alcançado e as frequências não chegaram a 5 GHz. Foi então que a indústria decidiu investir em processador com múltiplos núcleos em um mesmo *chip* (*multicores*).

Processadores *multicore* representam a possibilidade de executar mais instruções por ciclo de máquina e por isso podem oferecer mais desempenho. Além disso, também são mais eficientes do ponto de vista do consumo de energia [4]. Algumas aplicações se beneficiam automaticamente deste desempenho, visto que são aplicações trivialmente paralelizáveis. Por outro lado, é necessário grande esforço para paralelizar outras classes de aplicações, sendo necessário reescrever praticamente todo o código da aplicação.

Essa dificuldade de programação se deve ao modelo de execução utilizado tradicionalmente (Von Neumann) ser inerentemente sequencial. Por isso, surge a necessidade da adoção de alternativas a esse modelo para a programação paralela, como é o caso do modelo *Dataflow* que abordaremos nesse trabalho.

As arquiteturas *Dataflow* tiveram sua inspiração no Algoritmo de Tomasulo [1], baseado em fluxo de dados e implementado em hardware. Foi utilizado inicialmente no *mainframe* IBM 360/91 e mais tarde na linha de processadores Pentium™ da Intel®. O Algoritmo de Tomasulo é uma técnica para se obter ILP (*Instruction Level Parallelism*) em máquinas superescalares através da execução fora de ordem. Estações de reserva (registradores) são utilizadas para armazenar as instruções despachadas e seus respectivos operandos. Assim que seus operandos estão prontos, uma instrução pode ser executada na unidade funcional apropriada. Com isso, instruções podem "passar na frente" de outras no estágio de execução do *pipeline*,

dependendo da ordem que recebem os operandos.

O conceito da arquitetura *Dataflow* foi concebido em 1973 por Jack B. Dennis e David P. Misunas que publicaram o artigo seminal [1] descrevendo a arquitetura. A característica principal nesta arquitetura é que as instruções podem ser executadas assim que seus operandos estão disponíveis. Portanto, o programa segue o fluxo de dados ao invés de seguir a ordem sequencial do código (modelo Von Neumann). Além disso, as dependências de dados e controle são resolvidas naturalmente e não há a ocorrência de bolhas no *pipeline* de execução.

Para expressar o fluxo de dados entre instruções surge a necessidade de utilizar um grafo direcionado. Os nós do grafo representam instruções e as arestas direcionadas representam o sentido em que os operandos são enviados. A Figura 1.1 exemplifica um programa e seu respectivo grafo *Dataflow* associado.

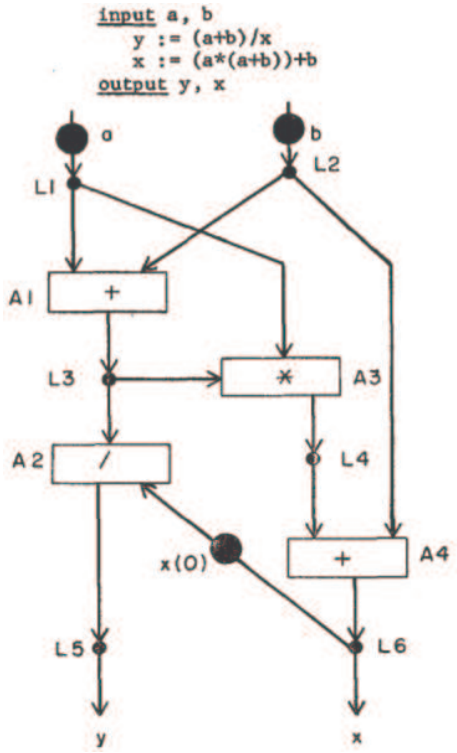


Figura 1.1: O "primeiro" programa *Dataflow* descrito no artigo de Dennis e Misunas [1].

As instruções *Dataflow* são executadas em Elementos de Processamento (EPs). Eles devem ser tão simples quanto possível, de modo que possam ser replicados massivamente para oferecer o máximo de paralelismo. Os EPs são responsáveis por receber operandos, executar instruções e enviar os resultados (operandos de saída) para as instruções de destino, que podem estar no mesmo EP ou não. Ao receber

um operando, o Elemento de Processamento verifica se possui alguma instrução que está a espera deste para prosseguir. Este processo de verificação tem o nome de *matching*. Caso todos os operandos de entrada de uma instrução estejam disponíveis, ela é despachada para execução. Em caso negativo, o operando é armazenado e fica a espera dos demais operandos que tornem a instrução pronta para executar.

Uma diferença fundamental entre o Algoritmo de Tomasulo e a arquitetura *Dataflow* é que no primeiro as estações de reserva são limitadas. Isto implica em limitar o número de instruções despachadas por ciclo. Além disso, as instruções são despachadas na ordem original do programa. Já na arquitetura *Dataflow*, virtualmente todas as instruções estão em memória e podem ser executadas tão logo que seus operandos estejam prontos.

Simultaneamente ao advento da arquitetura *Dataflow*, surge o problema de como mapear instruções para os EPs (Figura 1.2). Isto é, dado um conjunto de instruções I, um grafo direcionado $G(I, E)$, que descreve as dependências entre instruções, e um conjunto de Elementos de Processamento P, qual deve ser a maneira de distribuir (mapear) as instruções nos Elementos de Processamento, de modo a tentar minimizar o tempo total de execução do programa *Dataflow*? É este o problema que nos dedicamos a estudar nesta dissertação.

Vale a pena mencionar que o mapeamento em uma máquina *Dataflow* pode ser estático ou dinâmico. Entende-se por mapeamento estático quando uma instrução é alocada a um EP em tempo de compilação e permanece no mesmo Elemento de Processamento até o fim da execução do programa. Já no mapeamento dinâmico, a instrução pode ser alocada a um EP no momento de sua execução, de modo a tentar melhorar a performance e diminuir o tempo total de execução do programa. Outra possibilidade é, a partir de um mapeamento estático, migrar instruções durante a execução do programa. Além disso, um EP pode requisitar instruções de outros EPs de acordo com sua ocupação. Esta última técnica de mapeamento é conhecida como roubo de tarefas.

Neste trabalho, trataremos apenas do mapeamento estático. O roubo de tarefas é um dos temas abordados em tese de doutorado em elaboração no PESC/COPPE/UFRJ. Note que é aconselhável que o algoritmo de mapeamento dinâmico comece a ser executado a partir de um bom escalonamento estático, devido ao alto custo computacional de se reescalonar uma instrução para um outro Elemento de Processamento. Portanto o roubo de tarefas não invalida este estudo. Na verdade, são trabalhos complementares.

Curiosamente, em 1987, pesquisadores do *NASA Ames Research Center* conduziram um estudo de viabilidade de arquiteturas *Dataflow* para aplicações de Dinâmica de Fluídos (CFD). Neste artigo [5], os autores comentam sobre a dificuldade de se obter bons mapeamentos (chamados por eles de *partitioning*) e a necessidade de

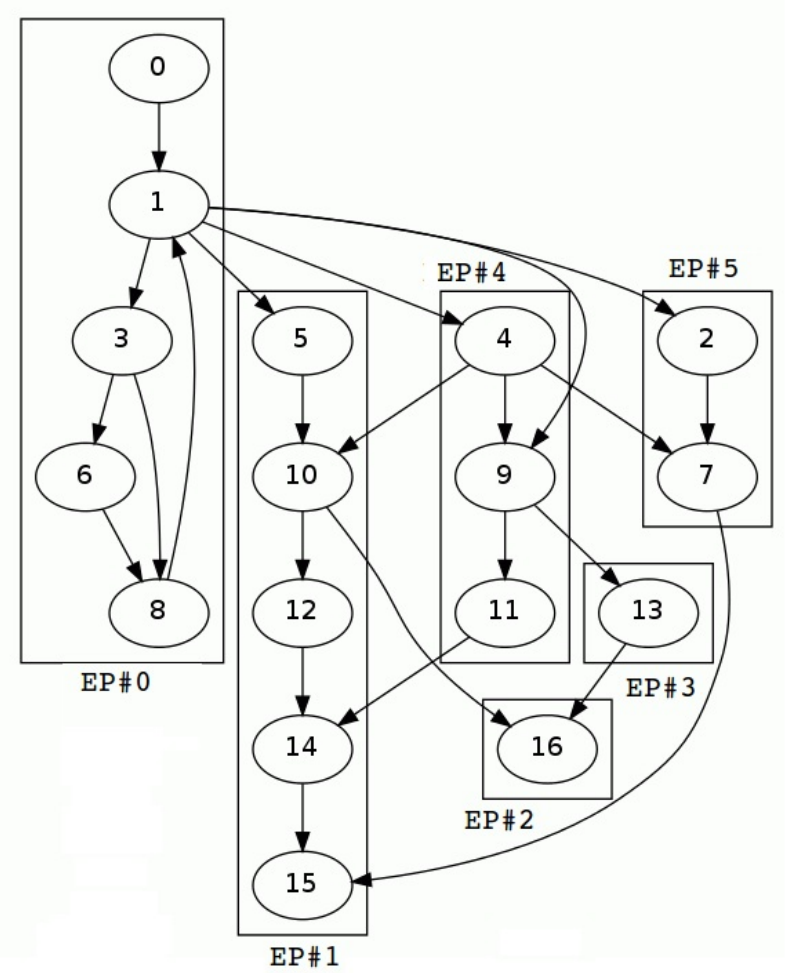


Figura 1.2: Exemplo de mapeamento em 6 Elementos de Processamento. Cada conjunto de instruções dentro dos retângulos é alocado a um EP distinto.

algoritmos que resolvam este problema (*automatic partitioner*):

> "The partitioning of instruction nodes among PE is an important and difficult problem. Utility of the machine is determined by the node assignments. Since optimal partitioning is difficult, an acceptable solution is an automatic partitioner that runs fast and produces execution times close to, or better than, a hand partitioning for the same data flow graph."

O mapeamento estático, mais conhecido como *Placement* no contexto do *Dataflow* é um problema NP-difícil [6]. Trata-se de como distribuir as instruções nos Elementos de Processamento com o objetivo de minimizar o tempo total de execução de um programa. Para alcançar este objetivo, temos basicamente duas estratégias, a saber:

1. Maximizar o paralelismo da instruções executadas nos EPs.
2. Minimizar a comunicação (troca de operandos) entre EPs distintos.

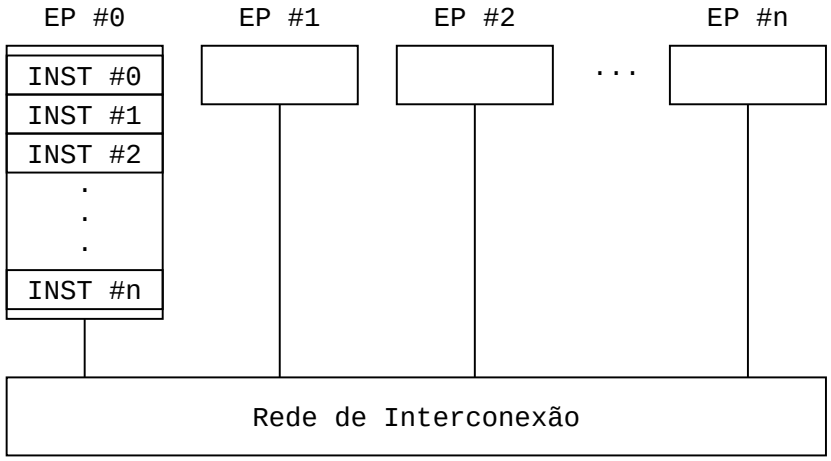
Como podemos ver nas Figuras 1.3(a) e 1.3(b), estas estratégias são conflitantes. Atacar o problema do *mapeamento* consiste em lidar com este conflito para obter o melhor desempenho de um programa.

É intuitivo perceber por que maximizar o paralelismo dos EPs minimiza o tempo total de execução de um programa, visto que paralelizar é uma oportunidade para executar mais de uma instrução por ciclo de máquina. No entanto, minimizar a troca de operandos entre EPs distintos também melhora o desempenho de um programa.

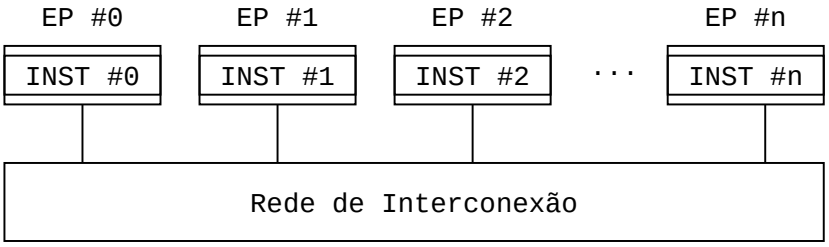
Quando um EP necessita enviar um dado (operando) para outro, eles se comunicam através de uma rede de interconexão. Consequentemente, existe um tempo de propagação (latência) desse operando na rede. Já quando a comunicação ocorre dentro do mesmo EP, o operando não necessita trafegar na rede e está disponível para o EP já no próximo ciclo. Portanto a maneira com que as instruções estão mapeadas nos EPs afeta o tempo utilizado com comunicação e isto impacta fortemente no tempo total de execução do programa.

Esta dissertação de mestrado dedica-se a estudar o problema do mapeamento das instruções *Dataflow* e faz as seguintes contribuições:

- A implementação de um Simulador Arquitetural *Dataflow*.
- Um Algoritmo de Mapeamento.
- Estudo comparativo entre algoritmos de Mapeamento.



(a) *Overhead* de comunicação mínimo porém paralelismo mínimo.



(b) Paralelismo máximo porém *overhead* de comunicação máximo.

Figura 1.3: Os dois extremos do espectro no mapeamento: todas instruções em um mesmo EP (a); uma instrução em cada EP (b).

Este trabalho está dividido da maneira que segue. No Capítulo 2, iremos explicar as motivações que nos levaram a escolha deste tema de dissertação e mencionar alguns estudos que o contextualizam. No Capítulo 3, apresentaremos o Simulador Arquitetural *Dataflow*, que será nossa plataforma para realização de experimentos e validação do algoritmo. Em seguida, percorreremos o caminho que levamos para conceber o algoritmo e apresentaremos sua forma final. No Capítulo 5, descreveremos o experimentos de validação do Algoritmo e um estudo comparativo entre outras abordagens de escalonamento propostas em [6] e [7]. Por fim, apresentaremos as conclusões deste trabalho e sugestões de trabalhos futuros.

Capítulo 2

Motivação e Trabalhos Relacionados

Neste capítulo iremos apresentar os fatores que motivaram a escolha do tema desta dissertação. Além disso, comentaremos sobre cada um dos trabalhos relacionados e a forma como contribuíram para este trabalho. Assim, esperamos contextualizar a arquitetura *Dataflow* e o problema do mapeamento de instruções.

2.1 Motivação

A proposta inicial desta dissertação de mestrado era dar continuidade a implementação da máquina virtual *Dataflow Trebuchet* [8] e estendê-la para utilização em *clusters* de computadores. O intuito era demonstrar a escalabilidade do paradigma *Dataflow* e da máquina virtual em si. No entanto, durante a etapa de revisão bibliográfica, nos convencemos que para obter sucesso nesta iniciativa, deveríamos primeiramente tratar do problema de como mapear instruções *Dataflow* entre os diferentes nós computacionais e suas respectivas unidades de processamento (*cores*).

Apesar de termos a máquina virtual *Trebuchet* implementada, ela não é a mais indicada para estudar como o mapeamento influencia no tempo de execução de um programa Dataflow. Primeiramente, o *overhead* da virtualização poderia esconder informações sobre o impacto do mapeamento no real tempo de execução [9]. Além disso, outros fatores poderiam influenciar na medição do tempo, como por exemplo, o escalonamento de processos do sistema operacional (a máquina virtual é vista com um processo pelo sistema operacional). Por isso, um dos produtos desta dissertação foi a implementação de Simulador Arquitetural *Dataflow*, em nível de ciclo de máquina, que será descrito minuciosamente no capítulo 3.

Por fim, com este trabalho esperamos contribuir para consolidar as iniciativas do grupo LAM/PESC/COPPE/UFRJ ([8, 10–16]) no estudo do paradigma *Dataflow*.

Vale ressaltar que um estudo sobre o mapeamento dinâmico de instruções (roubo de tarefas) está em desenvolvimento em uma tese de doutorado e poderá utilizar os resultados e conclusões obtidos aqui.

2.2 Trabalhos Relacionados

Primeiramente, apresentaremos a arquitetura WaveScalar que contribuiu para o ressurgimento da arquitetura *Dataflow* no atual cenário da Arquitetura de Computadores. Depois discutiremos a *Trebuchet*, máquina virtual *Dataflow* que se aproveitou do ressurgimento das ideias *Dataflow* e da popularização dos *multicores*. Também apresentaremos o compilador *Couillard* que foi utilizado neste trabalho.

Em seguida, abordaremos trabalhos relacionados ao problema do mapeamento de instruções propriamente dito. Como referência, temos um trabalho dos próprios desenvolvedores da arquitetura WaveScalar. Além dele, também nos baseamos em um trabalho para escalonamento de tarefas.

2.2.1 Arquitetura WaveScalar

Um grande entrave para a popularização do paradigma *Dataflow* era o fato de suportar apenas linguagens de programação funcionais (ex: Lisp, ML, Haskell). Esta decisão se deve ao fato das linguagens funcionais não serem suscetíveis a *efeitos colaterais*, isto é, algo que altere o estado global da computação, por exemplo, acionar um *flag* de exceção ou até mesmo a escrita em memória global. Por isso, não haveria a necessidade de lidar com alguns problemas, tais como a ordem em que os dados seriam escritos na memória e consequentemente a implementação em hardware era facilitada.

Porém, por não estar disponível para ser utilizada com linguagens imperativas (ex: C, Fortran, Pascal) que eram mais populares e produtivas, a arquitetura *Dataflow* praticamente caiu no esquecimento dos arquitetos de computadores. Mas em 2006, Steven Swanson, da Universidade de Washington, propôs em sua tese de doutorado a arquitetura WaveScalar [2], com a ideia de tirar proveito do paradigma *Dataflow* e ao mesmo tempo utilizar linguagens imperativas,

Uma das premissas da arquitetura WaveScalar é que em tempo de compilação são adicionadas às instruções *Wave-ordering annotations*, informações com a ordem de acesso a memória do programa na linguagem imperativa original. Então a arquitetura se encarrega de garantir que as operações de memória sejam realizadas na ordem das *Wave-ordering annotations*. Ou seja, a execução obedece a execução *Dataflow*, mas o acesso a memória continua obedecendo a sequência do programa original. Com isso a semântica do programa original é preservada.

Além de resolver o problema da linguagens imperativas, a arquitetura WaveScalar também propõe conjunto de instruções, modelo de execução e uma hierarquia escalável de Elementos de Processamento (WaveCache) - ver Figura 2.1. Na WaveCache, a latência de rede depende da distância entre os EPs na hierarquia.

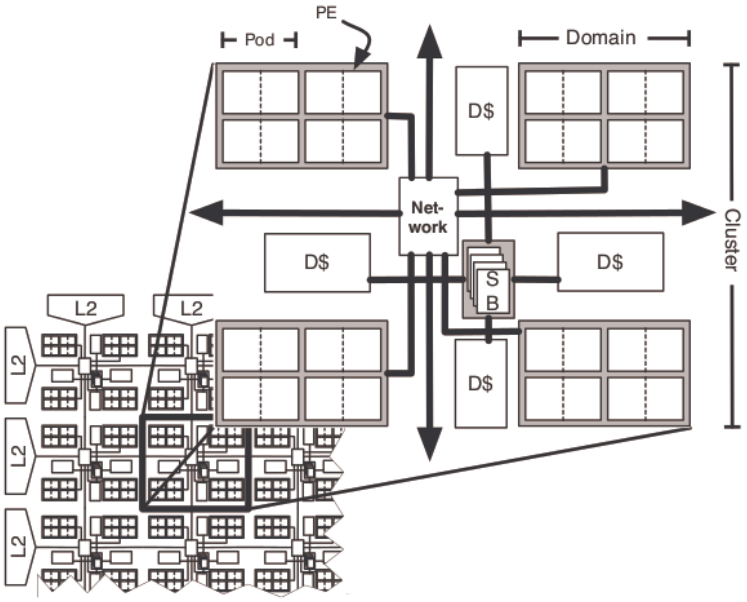


Figura 2.1: A hierarquia de Elementos de Processamento da WaveScalar [2].

2.2.2 Máquina Virtual *Trebuchet*

Com o intuito de aproveitar a disseminação do uso dos processadores *multicores* e explorar o paralelismo oferecido pelos mesmos e aproveitando as ideias de Swanson, surgiu a ideia de implementar uma máquina virtual Dataflow, a *Trebuchet*. Apesar de ser uma opção em software, esta máquina foi concebida de forma a tentar se mitigar ao máximo o *overhead* inerente da virtualização e tentar obter o melhor desempenho possível.

Nesta máquina virtual, as instruções são mapeadas em Elementos de Processamento *Dataflow*, que por sua vez são mapeados nos próprios processadores. Por isto a Trebuchet é *multi-threaded* e cada *thread* representa o fluxo de execução de um dos elementos de processamento *Dataflow*.

A máquina virtual Trebuchet também oferece a possibilidade de paralelização de programas em diferentes níveis de granularidade. Isto é feito em nível de instruções simples (grão fino) até funções, também chamadas de superinstruções (grão grosso).

Atualmente a Trebuchet continua em desenvolvimento e foi tema de dissertação

de Tiago Alves [8] e da tese doutorado de Leandro Marzulo [11].

2.2.3 Compilador *Couillard*

O compilador Dataflow *Couillard* [11] foi criado para transformar código de alto nível (linguagem `C`) em *assembly* da máquina virtual *Trebuchet*, descrevendo um grafo *Dataflow*. Ele também gera o código de superinstruções (instruções de granularidade grossa) que podem ser definidas pelo programador. Essas superinstruções são posteriormente compiladas por um compilador `C` tradicional (ex: gcc) e podem ser carregadas dinamicamente pela *Trebuchet*. Por isso, o *Couillard* não necessita suportar toda a gramática ANSI-C, já que algumas instruções ficam dentro da própria superinstrução.

Nesta dissertação, como o foco não era a utilização da máquina virtual e sim do Simulador Arquitetural (com a finalidade do estudo dos efeitos do mapeamento) utilizamos apenas a parte da transformação do código de alto nível em grafo *Dataflow*. Além da linguagem de montagem da máquina virtual, o compilador *Couillard* gera o grafo *Dataflow* que descreve a troca de operandos entre as instruções no formato `dot` e pode ser plotado através do Graphviz [17].

Para utilizar o compilador neste trabalho, decidimos utilizar este formato do Graphviz como linguagem intermediária. Assim, compilamos um programa em `C` e geramos o arquivo `dot`. Implementamos um programa que transforma o arquivo `dot` no formato de entrada do Simulador Arquitetural Dataflow, descrito na seção 3.3. Com isto, conseguimos transformar um programa em `C` em um programa executável no Simulador. É claro que temos algumas limitações, pois, como dito anteriormente, o compilador não suporta toda a gramática ANSI-C. No entanto, o compilador *Couillard* nos foi bastante útil para transformar os códigos dos programas de teste (ver Apêndice) que foram utilizados no estudo comparativo.

2.2.4 Mapeamento na WaveScalar

Neste trabalho [7], os autores estudam os efeitos do mapeamento (*Placement*) na arquitetura WaveScalar. Um modelo de performance é proposto e algoritmos de mapeamento são sugeridos. Também é realizado um estudo de avaliação dos algoritmos. Deste estudo, utilizamos os seguinte algoritmos de mapeamento de instruções *Dataflow*: `static-snake` e `depth-first-snake`.

Vale lembrar que a arquitetura WaveScalar é espacial, isto é, existe uma hierarquia de comunicação entre os EPs que varia de acordo com a distância espacial entre EPs. No nosso Simulador, por questão de simplicidade, decidimos considerar que todos os EPs tem a mesma distância entre si, e consequentemente a latência é constante para a comunicação inter-EP. Se quiséssemos implementar uma hierar-

quia, teríamos que inserir no Simulador uma matriz de latências, onde cada entrada $Latencia_{i,j}$ seria o custo de comunicação (em ciclos de máquina) do EP i ao EP j.

Por não possuirmos um ambiente de testes compatível ao do WaveScalar e ser de difícil adaptação não conseguiríamos reproduzir os testes no ambiente deles. No entanto, através da descrição dos algoritmos do artigo, foi possível implementá-los (com algumas adaptações) para serem utilizados no nosso estudo comparativo de algoritmos de mapeamento. Os algoritmos serão descritos em detalhes na seção 5.3.

2.2.5 Algoritmo de escalonamento de tarefas

Na etapa de revisão bibliográfica, na busca de outros algoritmos de mapeamento/escalonamento de instruções, nos deparamos com o artigo "Non-evolutionary algorithm for scheduling dependent tasks in distributed heterogeneous computing environments" [6]. Este artigo aborda um problema semelhante ao nosso, que é o escalonamento de tarefas dependentes.

Uma diferença para o nosso trabalho, é que em [6] o escalonamento é em um ambiente de computação heterogênea (DHC - *Distributed Heterogeneous Computing*), onde uma tarefa t executa em uma máquina m em um tempo de execução $E_{m,t}$. No nosso caso, uma instrução leva o mesmo tempo para ser executada em qualquer Elemento de Processamento, embora as diferentes instruções possam ter tempos de execução diferentes.

Outra importante diferença, é que no contexto deste trabalho as dependências entre tarefas estão descritas em DAGs (Directed Acyclic Graph). Já em um programa *Dataflow* é comum a presença de ciclos, devido a loops, por exemplo.

Neste artigo, utiliza-se uma técnica de programação dinâmica para o cálculo do *makespan* de um conjunto de tarefas, isto é, o tempo esperado para execução deste conjunto de tarefas respeitando as restrições de dependências. No algoritmo proposto, utilizamos esta técnica e a estendemos para utilização em grafos com ciclos.

Capítulo 3

Simulador Arquitetural *Dataflow*

Neste capítulo, iremos descrever o Simulador Arquitetural *Dataflow* que foi implementado durante esta dissertação. Apresentaremos aspectos da arquitetura *Dataflow*, como foram implementadas no Simulador e eventuais simplificações realizadas. O Simulador foi desenvolvido na linguagem Java por uma questão de praticidade, mas isto não influencia a performance dos programas já que o tempo de execução é medido em ciclos de máquina.

Apesar do Simulador não ser o foco principal deste trabalho, necessitávamos de uma plataforma de testes onde pudéssemos medir os efeitos do mapeamento das instruções no desempenho de um programa. Ou seja, era preciso um ambiente controlado onde pudéssemos testar nossas hipóteses. Além disso, munidos de um Simulador Arquitetural, podemos ter um melhor controle sobre a execução do programa e gerar *traces* para serem estudados a *posteriori*.

Conforme dito no capítulo anterior, tínhamos a opção de utilizar a máquina virtual *Dataflow Trebuchet* como plataforma de testes. No entanto, ela não seria a mais apropriada para este tipo de estudo, devido aos efeitos da virtualização. Vale lembrar, que a máquina virtual é vista como um processo pelo sistema operacional e seu tempo de execução não é determinístico. Seria impossível isolar precisamente as variáveis envolvidas na execução de um programa a partir de seu mapeamento.

Por isso, desenvolvemos o Simulador em nível de ciclo de máquina, isto é, ao final de uma execução, sabemos quantos ciclos foram necessários para executar um programa e o que cada Elemento de Processamento realizou a cada ciclo. Também temos informações acerca de como a Rede de Interconexão foi utilizada. Assim, podemos calcular tempos de comunicação e computação, quantos ciclos um determinado EP ficou ocioso, etc. Este tipo de informação nos auxiliou na concepção do Algoritmo proposto e no estudo comparativo entre algoritmos de mapeamento.

3.1 Visão Geral

Em alto nível, a arquitetura *Dataflow* é um ambiente de computação distribuída, caracterizada por um conjunto de Elementos de Processamento (processadores) interligados por uma Rede de Interconexão. Diferentemente da arquitetura Von Neumann, não existe a figura do PC (*Program Counter*). A execução é guiada pelo fluxo de dados, isto é, assim que uma instrução recebe todos os operandos que necessita, ela é despachada e pode executar.

Por exemplo, uma instrução de adição, que necessita de 2 operandos de entrada, ao receber o primeiro, ainda não pode executar e portanto apenas armazena-se este operando em uma estrutura especial (*Matching Table*) e fica a espera do próximo. Ao receber o segundo operando, o EP detecta que todos os operandos necessários para aquela adição estão disponíveis, então pode executar. Após a execução, esta adição gera um operando de saída (resultado da adição) que será enviado para outras instruções que dependem deste resultado. Consequentemente, outras execuções poderão ser disparadas.

Neste Simulador, decidimos implementá-lo de maneira síncrona, ou seja, a execução de cada EP é acionada por um único relógio global. Isto facilita a implementação e a reprodução das execuções dos programas. No entanto, nada impediria que cada Elemento de Processamento tivesse seu próprio relógio e sua própria frequência.

Cada Elemento de Processamento possui um pipeline interno (ver Figura 3.1). Apesar de ter pipeline definido, o EP se comporta como um monociclo, no sentido que todas as etapas do pipeline são realizadas sequencialmente em um mesmo ciclo. O pipeline se divide basicamente em 3 estágios: recebimento de dados no *Buffer de Entrada*, comparação de operandos na *Matching Table* e execução de fato. Estas etapas serão descritas mais adiante neste capítulo.

3.2 Instruções

Uma instrução *Dataflow* é representada por um nó no grafo *Dataflow*. Os operandos da instrução são recebidos por portas de entrada e enviados por portas de saída. As portas de uma instrução são ordenadas, visto que existe uma ordem entre os operandos. Por exemplo, a instrução `COMPMEN` ("Operador Menor que", $<$) tem duas portas de entrada. Como executar $A < B$ é diferente de $B < A$, a instrução sempre executa $PE_1 < PE_2$, sendo PE_j a j-ésima porta de entrada da instrução.

É importante ressaltar que o número de portas de uma instrução não é necessariamente igual ao número arestas que incidem nela. Isto é, dependendo do grafo *Dataflow* pode haver mais de uma aresta incidindo em uma mesma porta. De acordo com caminho que o programa tomar, uma instrução pode ser executada ou não e

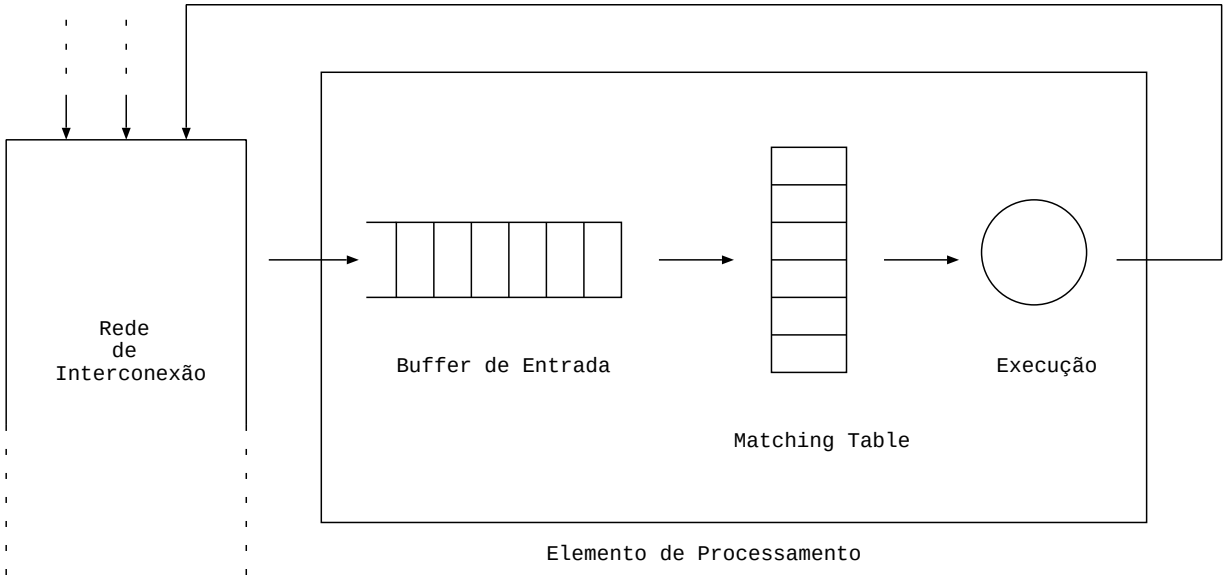


Figura 3.1: O pipeline de um Elemento de Processamento no Simulador Arquitetural.

consequentemente um operando ser enviado ou não. Isto ocorre porque o caminho de execução de um grafo *Dataflow* depende da entrada do programa, já que existem instruções condicionais (*Steers*) e um grafo pode ter um trecho executado ou não.

A Tabela 3.1 lista as instruções implementadas atualmente no Simulador. Note que foi implementado apenas um conjunto reduzido de instruções. Cada instrução foi adicionada conforme a necessidade já que o propósito não era criar um Simulador com um conjunto de instruções completo e sim estudar os efeitos do mapeamento. Estas instruções foram suficientes para rodar o nosso conjunto de programas de teste.

Vale a pena comentar sobre algumas instruções especiais *Dataflow* que não são encontradas no processadores convencionais. *Steer* (`ST`) é uma instrução condicional e guia o fluxo de dados em um grafo *Dataflow*. É uma instrução que tem 2 portas de entrada. Pela porta de entrada PE_1 recebe um valor booleano (0 ou 1) e pela porta PE_2 recebe um valor. Se booleano é 1, ela envia o operando recebido na PE_2 pela porta de saída PS_1 (também representada por `T` de *true*); se o booleano é 0, ele envia o operando recebido pela PE_2 pela porta de saída PS_2 (também representada por `F` de *false*). Costuma-se representar o *Steer* conforme a Figura 3.2.

Para explicar as instruções `WA` (*Wave Advance*) e `ZW` (Zera *Wave*) é necessário mencionar o conceito de Onda. Em uma arquitetura *Dataflow* dinâmica, como é a implementada no Simulador, é possível executar múltiplas instâncias da mesma instrução. Consequentemente, podem haver ciclos no grafo *Dataflow* para expressar

Mnemônico	Descrição
ADD	Adição
ADDI	Adição com imediato
MUL	Multiplicação
COMPMEN	Operador Menor que
COMPMENI	Operador Menor ou Igual
COMPIGUI	Operador Igual
OUT	Saída de dados (impressão)
CONST	Produz constante
LOAD	Leitura em memória
STORE	Escrita em memória
WA	Wave Advance
ZW	Zerar Wave
ST	Steer

Tabela 3.1: Instruções implementadas no Simulador.

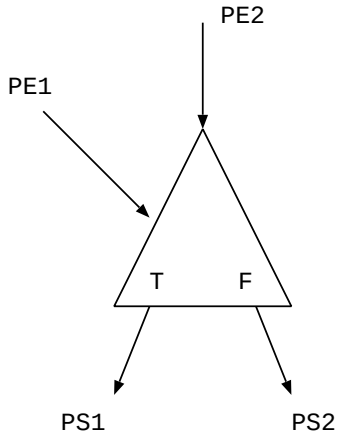


Figura 3.2: Representação da instrução condicional *Steer*.

repetições. Por isso, uma instrução pode receber operandos de diferentes iterações da repetição. No entanto, só faz sentido executar uma instrução ao receber operandos que sejam da mesma iteração. Surge um problema: como o EP identificaria de qual iteração um operando pertence? Então surge o conceito de Onda, um identificador de iteração do operando que é incrementado (a cada iteração) pela instrução `WA` (*Wave Advance*) — também denotada por *Inc Tag* (`IT`). Pela regra *Dataflow*, uma instrução só é executada se todos os operandos que necessita possuírem o mesmo número de Onda, assim assegurando a correta computação. Se houver a necessidade de zerar este identificador, para fazer operações com operandos oriundos de diferentes ciclos, por exemplo, utiliza-se a instrução `ZW` (Zera *Wave*).

3.3 *Loader*

Para carregar um programa *Dataflow* no Simulador é necessário um arquivo de entrada que descreva o programa a ser executado. Este arquivo é processado por um *loader*. O *loader* é responsável por realizar o *parsing* do grafo *Dataflow*, carregar as instruções para *Instruction List* de cada EP e carregar os mensagens iniciais (operandos) nos *Buffers* de Entrada de cada EP.

O *parser* foi desenvolvido com o Javacc (Java Compiler Compiler) [18], um gerador automático de *parsers*, de modo a possibilitar uma maior flexibilidade em seu desenvolvimento. O *parser* é descrito através de uma gramática. Com isto, é muito simples adicionar uma nova instrução no Simulador por exemplo.

O arquivo de entrada do Simulador está divido em 4 partes, são elas:

1. A lista de instruções.
2. O grafo de dependências entre instruções, descritos através de arestas.
3. O mapeamento, ou seja, em qual EP cada instrução ficará alocada.
4. As mensagens inicias, isto é, os operandos responsáveis por iniciar a computação Dataflow.

Na lista de instruções, cada linha descreve uma instrução (nó no grafo) e segue o seguinte formato: `Id da Instrução: Tempo de Execução : Código da Instrução : Imediato` . O id é o número identificador único da instrução. Já o tempo de execução é o número de ciclos de máquina que a instrução deverá passar no estágio de execução do EP, realizando a computação de fato. Note que isto abre a possibilidade de realizar testes com instruções de diferentes níveis de granularidade. Em seguida, temos o mnemônico da instrução (conforme a tabela 3.1) que identifica qual operação será realizada, quais são seus operandos de entrada e de saída. Por fim, temos o operando imediato, que é opcional, pois só faz sentido nas instruções que utilizam operandos imediatos, como é o caso do `ADDI`. O bloco `NODES` da Figura 3.3 mostra um exemplo de lista de instruções.

Para representar as dependências entre instruções, é necessário descrever suas arestas. A descrição de uma aresta tem o seguinte formato: `Id da Instrução Predecessora ( Porta de Saída da Instrução Predecessora ) -> Id da Instrução Sucessora ( Porta de Entrada da Instrução Sucessora )` . Os ids das instruções sucessora e predecessora são definidos previamente na lista de instruções. Quando não especificado, a porta de saída *default* é a 0, mas é possível utilizar uma outra porta de saída. No bloco `EDGES` da Figura 3.3 as portas de saída são explícitas na instrução 6 (*Steer*), para especificar as 2 possibilidades de

```
NODES
0:1:WA
1:1:WA
2:1:WA
3:1:WA
4:1:COMPMEN
5:1:ST
6:1:ST
7:1:ST
8:1:ST
9:1:ADDI:1
10:1:ADD
11:1:OUT

EDGES
0 -> 5(1)
1 -> 6(1)
2 -> 4(0),7(1)
3 -> 4(1),8(1)
4 -> 5(0),6(0),7(0),8(0)
5 -> 0(0),10(0)
6(0) -> 10(1)
6(1) -> 11(0)
7 -> 9(0)
8 -> 3(0)
9 -> 2(0)
10 -> 1(0)

PLACEMENT
[[2, 3, 4, 7, 8, 9], [0, 5], [1, 6, 10, 11]]

MESSAGES
0(0)=5, 1(0)=0, 2(0)=0, 3(0)=6
```

Figura 3.3: Exemplo de um arquivo de entrada do Simulador.

caminhos no grafo de acordo com o valor booleano. Por último, temos a lista de instruções sucessoras e suas respectivas portas de entrada.

Uma parte fundamental do arquivo de entrada do Simulador é a descrição do mapeamento, que é tema desta dissertação. Como os Elementos de Processamento são idênticos entre si, a comunicação inter EP tem a mesma latência e não há número máximo de EPs, não é necessário identificar explicitamente os EPs no arquivo. Para definir o mapeamento, basta descrever uma partição das instruções e cada subconjunto da partição será alocado a um Elemento de Processamento distinto.

No bloco `PLACEMENT` da Figura 3.3, temos uma partição com 3 subconjuntos de instruções. As instruções `2, 3, 4, 7, 8, 9` serão mapeadas no EP_0, as instruções de `0, 5` serão mapeadas no EP_1 e as instruções `1, 6, 10, 11` no EP_2 , sendo EP_i , o i-ésimo Elemento de Processamento. O mapeamento é representado através do grafo da Figura 3.4.

Finalmente, os operandos iniciais são descritos através do seguinte

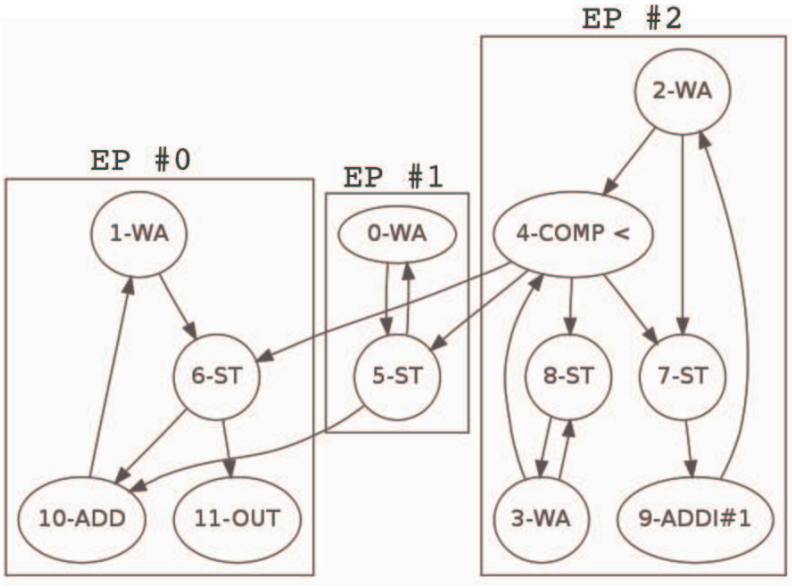


Figura 3.4: Representação do Grafo *Dataflow* descrito no arquivo de entrada da Figura 3.3.

formato: `Id da Instrução destino ( Porta de Entrada da Instrução ) = Imediato`. Como não existe um PC (*Program Counter*), para iniciar a computação é necessário incluir estes operandos para que as primeiras instruções executem. No bloco `MESSAGES` da Figura 3.3, as instruções `0, 1, 2 e 3` receberam pela porta `0` os operandos `5, 0, 0 e 6` respectivamente.

3.4 *Instruction List*

A *Instruction List* é uma região de memória que receberá as instruções mapeadas em um EP. Além das instruções em si, a *Instruction List* também armazena a lista de instruções destino de cada instrução. O *Loader* é o responsável por carregar as instruções na *Instruction List* de cada EP, realizando o mapeamento entre instruções e EPs.

Note que cada EP possui uma ULA (Unidade Lógica e Aritmética) preparada para executar qualquer instrução do conjunto de instruções. O fato da instrução estar alocada em sua *Instruction List* apenas significa que aquele EP será responsável por executá-la. Se duas instruções estão alocadas em um mesmo EP significa que não poderão executar em paralelo. No entanto, a latência de comunicação entre elas será menor, visto que o operando não necessitará trafegar pela Rede de Interconexão.

3.5 Rede de Interconexão

A Rede de Interconexão tem a função de enviar os operandos produzidos por uma instrução a suas instruções destino. Para realizar esta tarefa, a Rede de Interconexão deve primeiramente localizar em qual EP está mapeada a instrução de destino. Por isso, quando as instruções são carregadas pelo *Loader* nos EPs, é construída uma tabela de roteamento na Rede de Interconexão. Então, quando deseja-se enviar um operando para uma instrução I, localiza-se o EP em que I foi mapeada através da tabela de roteamento e a mensagem (operando) é direcionada ao EP correto.

A instrução de destino pode estar no mesmo EP ou em um EP diferente da instrução de origem. Se a instrução estiver no mesmo EP (comunicação intra- EP), a mensagem faz o *bypass* da Rede de Interconexão e estará no *Buffer* de Entrada do EP no ciclo seguinte. Caso contrário, a mensagem trafegará no barramento da Rede de Interconexão até chegar ao outro EP, o que chamamos de comunicação inter-EP.

No Simulador, a latência de comunicação é constante para quaisquer dois EPs que realizem troca de operandos. Ou seja, não existem EPs que sejam mais próximos, o que importa é se a comunicação é inter ou intra EP. Além disso, a latência de comunicação inter-EP (em ciclos de máquina) é um parâmetro do Simulador. Ou seja, podemos variar a latência para simular ambientes onde é mais "caro" ou mais "barato" realizar comunicação. Isto influencia diretamente no mapeamento, já que o custo da comunicação pode se tornar proibitivo, incentivando o algoritmo de mapeamento a alocar as instruções em um mesmo EP para evitar o custo da comunicação inter-EP.

Para ilustrar a comunicação inter-EP e como ela ocorre no Simulador, vejamos o exemplo da Figura 3.5 e seu *trace* de execução na Figura 3.6. Neste exemplo, o parâmetro de latência de comunicação utilizado no Simulador foi de 3 ciclos de máquina. Durante a execução, no primeiro ciclo o `EP 0` recebe a mensagem inicial (`M0`) e a instrução `I0` é executada (`Última Instrução: I0`). A instrução `I0` produz a mensagem (operando) para instrução 1 que trafegará no barramento por 3 ciclos de máquina (`M1:3`). Ao observar o barramento da Rede de Interconexão no ciclo 3, temos `[M1:3]`, no ciclo 2 temos `[M1:2]` e no ciclo 3 `[M1:1]`, indicando que o tempo restante para a mensagem chegar ao EP destino (`EP 1`) é decrementado a cada ciclo. No ciclo 4, a mensagem é recebida pelo EP 1 que finalmente pode executar.

3.6 *Buffer* de Entrada

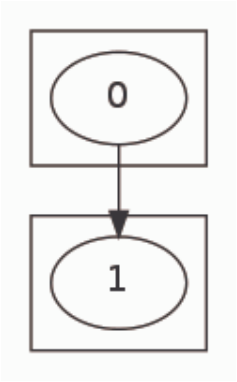
O Buffer de entrada é área de memória que recebe os operandos enviados por outras instruções. Ele atua como uma fila recebendo os operandos que chegaram através da Rede de Interconexão.

```
NODES
0:1:ADDI:1
1:1:OUT

EDGES
0 -> 1(0)

PLACEMENT
[[0],[1]]

MESSAGES
0(0)=1
```



(a) Arquivo de Entrada. (b) Grafo *Dataflow* mapeado em 2 EPs.

Figura 3.5: Exemplo de comunicação inter-EP, utilizando a Rede de Interconexão.

Se houver operandos no buffer de entrada, o primeiro operando da fila é removido para ser processado na *Matching Table*. A cada ciclo é removido apenas um operando do Buffer de Entrada. Este operando será utilizado na Matching Table.

Note que podem chegar operandos em rajada no *Buffer* de Entrada, isto é, mais de um operando pode ser recebido em um mesmo ciclo. Isto ocorre porque são produzidos operandos por diversos EPs.

Como apenas um operando é processado por ciclo, pode ocorrer de o EP já dispor de todas os operandos para executar uma instrução e não executá-la. O EP terá que esperar cada operando ser processado para que possa executar. Além disso, a ordem que os operandos chegam no Buffer pode alterar a computação.

3.7 *Matching Table*

Um dos principais pilares da filosofia das Arquiteturas *Dataflow* é que assim que os operandos de uma instrução estão prontos, ela pode ser executada. Essa é a chamada regra *Dataflow*. E é na *Matching Table* que esta regra é implementada.

A *Matching Table* é uma memória de operandos. Ao receber um operando que foi removido do Buffer de Entrada, ela primeiramente verifica (através de um campo do operando) para qual instrução este operando foi destinado. A partir da instrução (que está na Instruction List), é possível verificar quais são as portas de entrada da instrução e consequentemente quais são os operandos necessários para sua execução.

A Matching Table é percorrida para verificar se os demais operandos (além do que está sendo processado) estão presentes na tabela. Todos os operandos devem ser da mesma instrução, e ter o mesmo número de Onda (ver Seção 3.2). Se todos os operandos necessários já chegaram, dizemos que a instrução foi casada e os operandos são removidos da Matching Table e a instrução é despachada para fila de

```
1
EP #0
Buffer de Entrada:[M0(0)]
Matching Table: []
Última Instrução: I0
EP #1
Buffer de Entrada:[]
Matching Table: []
Última Instrução:
Rede de Interconexão
Barramento: [M1:3]
--------------------------------------------------
2
EP #0
Buffer de Entrada:[]
Matching Table: []
Última Instrução:
EP #1
Buffer de Entrada:[]
Matching Table: []
Última Instrução:
Rede de Interconexão
Barramento: [M1:2]
--------------------------------------------------
3
EP #0
Buffer de Entrada:[]
Matching Table: []
Última Instrução:
EP #1
Buffer de Entrada:[]
Matching Table: []
Última Instrução:
Rede de Interconexão
Barramento: [M1:1]
--------------------------------------------------
4
EP #0
Buffer de Entrada:[]
Matching Table: []
Última Instrução:
EP #1
Buffer de Entrada:[M1(0)]
Matching Table: []
Última Instrução: I1
Rede de Interconexão
Barramento: []
```

Figura 3.6: Trace gerado pelo Simulador ao executar o programa da Figura 3.5.

instruções prontas. Em caso negativo, o operando é simplesmente armazenado na Matching Table e fica-se a espera dos demais operandos que disparem a execução.

3.8 Execução

Se houver alguma instrução na fila de instruções prontas, ela será removida e enfim executada. A execução de fato ocorre na ULA (Unidade Lógica e Aritmética). O tempo de execução de uma instrução (em ciclos de máquina) é variável e é determinado no arquivo de entrada do Simulador (ver Seção 3.3). Enquanto uma instrução executa, outros operandos podem chegar no *Buffer* de Entrada e ser processados pela *Matching Table*.

Existe uma diferença fundamental entre uma ULA *Dataflow* e ULA de um processador convencional. No caso da ULA *Dataflow*, é necessário produzir tantos operandos de saída quanto a lista de dependências da instrução. Ou seja, se a instrução I tem n operandos de destino, a ULA produzirá um resultado que será replicado em n mensagens (operandos) para que possa ser enviado as suas n instruções de destino.

Após a execução, os operandos gerados são encaminhados para a Rede de Interconexão, que por sua vez se encarregará de encaminhar para as instruções de destino. Os operandos chegam no *Buffer* de Entrada dos EPs e todo o ciclo se reinicia.

3.9 Utilitários

Além do Simulador, alguns utilitários foram implementados para auxiliar neste trabalho. Para facilitar a visualização dos grafos e suas alocações, foi implementado de um programa gerador de imagens com o auxílio do pacote Graphviz. Diversas figuras desta dissertação foram gerados através deste programa.

Outro produto desta dissertação foi a integração parcial do Simulador Arquitetural com o compilador *Couillard*. Com isto, pudemos compilar programas em C para executar no Simulador. Esta integração será explicada em mais detalhes no Capítulo 5.

Capítulo 4

Algoritmo de Mapeamento

Depois de ter um Simulador pronto para execução de programas *Dataflow*, pudemos finalmente nos concentrar no problema de como mapear instruções para os Elementos de Processamento. Neste capítulo, iremos primeiramente definir o problema do mapeamento e descrever as variáveis envolvidas. Depois comentaremos sobre algumas tentativas preliminares de conceber um algoritmo que não se mostraram adequadas. Em seguida, começaremos descrevendo as ideias que inspiraram o Algoritmo proposto e sua evolução. Por fim, apresentaremos o Algoritmo em sua forma final.

4.1 Definição do Problema

Um programa *Dataflow* é um grafo direcionado $G(I, E)$, onde I é o conjunto das instruções (nós) e E das arestas que expressam as trocas de operandos entre instruções.

Temos um conjunto O de operandos iniciais, que são responsáveis por disparar a computação. Este conjunto de operandos pode ser considerado a entrada no programa pois guiam o fluxo de dados, impactando assim na execução do programa. Estes operandos podem ser utilizados, por exemplo, como contador de iterações de um loop ou como entrada de instruções condicionais.

Cada instrução executa em TE_i unidades de tempo, sendo i uma instrução. Portanto, as instruções podem ter tempos de execução distintos entre si. Esta possibilidade foi considerada para testarmos o mapeamento de instruções de diferentes granularidades (ex: superinstruções).

As instruções são executadas em um conjunto de Elementos de Processamento, P, e para determinar em qual EP cada instrução será executada, é necessário um mapeamento $M(I \rightarrow P)$. Além disso, existe uma rede de interconexão que conecta os EPs e possibilita troca de operandos entre eles. Esta rede de interconexão poderia ter uma topologia qualquer, mas por simplificação, trataremos de uma rede homogênea.

Por isso, a latência da comunicação inter-EP é constante. Ou seja, um operando leva L unidades de tempo para trafegar do EP p_i ao p_j, quando $i \neq j$. Já quando $i = j$, a comunicação é intra-EP e o operando está disponível para o EP na unidade de tempo seguinte a sua produção.

Como estamos considerando a crescente disponibilidade de processadores no contexto atual da Computação, não fixamos um número máximo de Elementos de Processamento. No entanto, não faria sentido ter mais Elementos de Processamento que de instruções, logo $|P| \leq |I|$. Além disso, não há quantidade máxima de instruções suportadas por um EP, pois supomos que a memória é grande o suficiente para armazenar as instruções.

Finalmente, considerando uma execução do programa *Dataflow* $G(I, E)$, os operandos iniciais O, os tempos de execução das instruções TE_i, o mapeamento $M(I \rightarrow P)$, a latência inter-EP L, o programa todo executa em T unidades de tempo. O problema do mapeamento consiste em encontrar um mapeamento M minimize o tempo total de execução T de um programa *Dataflow*. Neste Capítulo discutiremos um algoritmo que obtenha M que procure minimizar T.

Como consideramos os Elementos de Processamento idênticos e o custo de comunicação inter-EP constante L, o problema se resume a encontrar uma partição do conjunto de instruções I que minimize o tempo de execução T. Cada subconjunto de instruções da partição é mapeado a um EP distinto. Seja uma partição de I nos subconjuntos $I_1, I_2, \ldots, I_X$; as propriedades da partição evidenciam sua equivalência a um mapeamento:

- $I_x \neq \emptyset \ \forall x \in 1, 2, \ldots, X$, pois um EP sem instruções não participaria da computação.
- $\bigcup_{x=1}^{X} I_x = I$, pois todas as instruções devem ser mapeadas em algum EP.
- $I_i \cap I_j = \emptyset$ se $i \neq j$, pois uma instrução só pode estar mapeada em apenas um EP.

Gerar todas as partições (mapeamentos) de I é inviável computacionalmente para $|I|$ grande. Por exemplo, um programa com apenas 20 instruções, permite 51.724.158.235.372 possibilidades de mapeamentos diferentes. O número total de partições de um conjunto de n elementos cresce exponencialmente e é conhecido como número de Bell. Este número pode ser expresso pela recursão:

$$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k$$
$$B_0 = 1$$

Por fim, ainda temos que lidar com a restrição de tempo para a execução de nosso algoritmo de mapeamento. Este algoritmo deve ser executado em tempo de compilação. Portanto, não é aceitável esperar mais que um tempo "normal" de compilação. Isso nos fez caminhar em direção a um algoritmo aproximativo (baseado em heurística).

4.2 Primeiras Tentativas

4.2.1 Escalonamento por Reversão de Arestas

Uma estratégia para gerar uma alocação que maximize o paralelismo na execução *Dataflow* foi estudada em [19]. A ideia consiste primeiramente em gerar um grafo complementar a partir do grafo *Dataflow*. Munido do grafo complementar, gerar uma orientação acíclica para o mesmo. Algoritmos para geração de orientação acíclica foram estudados em [20] e implementados. Por fim, executar a dinâmica de Escalonamento por Reversão de Arestas (SER - *Scheduling by Edge Reversal*) no grafo gerado e alocar os *sinks* em um mesmo EP. Por construção, haverá arestas entre duas instruções independentes no grafo complementar e consequentemente não poderão ser *sinks* ao mesmo tempo. Por isso, serão alocadas em EPs diferentes. Esta abordagem de fato maximiza o paralelismo da execução. No entanto, falha ao minimizar a comunicação entre EPs. Esta foi uma das dificuldades enfrentadas nesta dissertação e fizemos alguns esforços com o intuito de contornar este problema. Por ora, decidimos não continuar com essa abordagem porém não a descartamos totalmente.

4.2.2 *List Scheduling*

Outra estratégia utilizada para mapear instruções foi a de *List Scheduling*. No entanto, os grafos *Dataflow* podem possuir ciclos, que dificultam a previsão do tempo de execução do programa e não se adequam a técnica de *List Scheduling*. Uma heurística considerada foi a remoção dos ciclos e aplicação do algoritmo de *List Scheduling* para se obter o *makespan* (tempo esperado de execução). No entanto, essa heurística não se mostrou razoável, visto que o valor do *makespan* e os tempos de execução de fato dos programa *Dataflow* apresentaram uma baixa correlação. Por isso, também não demos prosseguimento nesta ideia.

4.3 Programação Dinâmica

Durante a revisão bibliográfica, ao pesquisar algoritmos de mapeamento / escalonamento em grafos chegamos ao conhecimento de [6]. O problema tratado neste artigo é semelhante ao nosso. Existe um conjunto de tarefas dependentes e um conjunto de máquinas onde as tarefas são executadas e deseja-se obter um mapeamento de tarefas em máquinas que minimize o tempo total de execução das tarefas. A técnica utilizada neste algoritmo de mapeamento é a programação dinâmica.

No entanto, há algumas diferenças. Primeiramente, em [6] trata-se de um ambiente de computação heterogênea (DHC - *Distributed Heterogeneous Computing*), onde uma tarefa t executa em uma máquina m em um tempo de execução $TE_{t,m}$. Apesar de ser uma possibilidade, esta variação não entrou no nosso modelo. Além disso, em [6] a dependência de tarefas é expressa em um DAG (*Directed Acyclic Graph*) e o grafo *Dataflow* pode ter ciclos, devido a loops por exemplo.

Outra diferença sutil, é que no caso do *Dataflow* existe uma fila de operandos em cada Elemento de Processamento e os operandos podem chegar em rajadas. Isto pode afetar o tempo total de computação como veremos mais a frente.

Por uma questão de simplificação, vamos utilizar a mesma notação que vínhamos utilizando para o problema de mapeamento de instruções em Elementos de Processamento. Portanto as dependências de tarefas são descritas por um DAG $G(I, E)$, onde I é o conjunto das tarefas e E das arestas que expressam dependências entre tarefas. As tarefas são executadas no conjunto de máquinas P e também queremos obter um mapeamento $M(I \rightarrow P)$. E por simplificação, utilizaremos os tempos de execução de tarefas homogêneos TE_i.

No algoritmo de mapeamento de [6], o conceito chave utilizado é o de *makespan*. Tipicamente, *makespan* é o tempo total de execução de um conjunto de tarefas. No algoritmo de mapeamento proposto em [6], o conceito de *makespan* é estendido para o *makespan* de uma tarefa e de uma máquina. O *makespan* de uma tarefa i, MSI_i, é o tempo total de execução do programa até o fim da execução da tarefa i. Analogamente, o *makespan* de um processador, MSP_p, é o tempo total de execução naquela máquina p. Note que as máquinas podem ter makenspans diferentes de acordo com quais tarefas são mapeadas nelas.

Além do conceito de *makespan*, é utilizada a técnica de programação dinâmica neste algoritmo. O cálculo do MSI_i é em função do MSI_j , $\forall$ tarefa j imediatamente antecessora de i, que são previamente calculados. Além disso também são considerados os MSP_p que determinam até quando uma determinada máquina p está sendo utilizada.

Seja uma tarefa i que desejamos alocar, i só pode executar após todas as tarefas antecessoras de i executarem. Em particular, i só pode executar após a tarefa ante-

cessora de maior *makespan*, pois como i precisa esperar *todas* as tarefas antecessoras, consequentemente esperará pela mais demorada (de maior *makespan*). Portanto o $MSI_i > MSI_{j_{max}}$, sendo j_{max} a tarefa antecessora de i de maior *makespan*.

Considerando que uma tarefa i foi alocada a uma máquina p, o MSI_i será o $\max\{MSI_j, MSP_p\} + TE_i$, $\forall$ tarefas j imediatamente antecessoras a i. Além de respeitar a restrição de dependência entre tarefas, é necessário respeitar a restrição da máquina que já poderia estar alocada a uma outra instrução. Por fim, para calcular seu *makespan* é acrescido o tempo de execução da tarefa, TE_i, pois o *makespan* expressa o tempo de conclusão da mesma.

Quando queremos mapear uma tarefa i, devemos considerar a possibilidade de mapear i em todas as máquinas disponíveis e escolher a que minimize o MSI_i. Minimizando o MSI_i estaremos minimizando o *makespan* do conjunto de tarefas I como um todo. Segue um pseudo-algoritmo que ilustra todas as ideias envolvidas neste algoritmo.

Algoritmo 4.1 Gerar mapeamento de tarefas em Máquinas.

Entrada: As tarefas dependentes descritas pelo DAG $G(I, E)$, as máquinas P, os tempo de execução das tarefas TE_i
Saída: Um mapeamento $M(I \rightarrow P)$

```
1: Iniciar lista de tarefas prontas
2: para todo tarefa i ∈ prontas faça
3:    MSI_i ⇐ TE_i
4: enquanto Existirem tarefas prontas a serem escalonadas faça
5:    Remover uma tarefa i ∈ prontas
6:    mapear(i)
7:    Adicionar a prontas tarefas que foram liberadas após mapeamento de i
8: fim enquanto
9: procedimento mapear(i)
10:   para todo Máquina p ∈ P faça
11:      para todo Tarefas j imediatamente antecessoras a i faça
12:         MaxMSI_p ⇐ max{MSI_j, MSP_p}
13:      MinMaxMSI ⇐ min{MaxMSI_p}
14:   MSI_i ⇐ MinMaxMSI + TE_i
15:   MSP_p ⇐ MSI_i
16:   Mapear i na máquina p onde MinMaxMSI é mínimo
17: fim procedimento
```

Para demonstrar uma execução do Algoritmo 4.1, vamos utilizar o grafo da Figura 4.1. As Figuras 4.2 a 4.6 ilustram cada iteração do algoritmo. Note que todas as tarefas tem $TE_i = 1$.

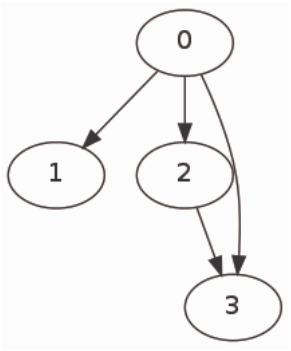


Figura 4.1: Exemplo de Grafo *Dataflow* a ser mapeado utilzando o Algoritmo 4.1.

```
TE[0] = 1
TE[1] = 1
TE[2] = 1
TE[3] = 1

Iteração 1

Prontas: 0

Mapeando 0

MSI[0] = ?
MSI[1] = ?
MSI[2] = ?
MSI[3] = ?

MSP[0] = 0
MSP[1] = 0
MSP[2] = 0
MSP[3] = 0

Tarefa 0 alocada a Máquina 0

MSI[0] = TE[0] = 1
```

Figura 4.2: Iteração 1 da execução do Algoritmo 4.1.

```
Iteração 2

Prontas: 1, 2

Mapeando 1

MSI[0] = 1
MSI[1] = ?
MSI[2] = ?
MSI[3] = ?

MSP[0] = 1
MSP[1] = 0
MSP[2] = 0
MSP[3] = 0

MinMaxMSI = Min {

Máquina 0 : Max{ MSI[0], MSP[0] } = 1
Máquina 1 : Max{ MSI[0], MSP[1] } = 1
Máquina 2 : Max{ MSI[0], MSP[2] } = 1
Máquina 3 : Max{ MSI[0], MSP[3] } = 1

}

Tarefa 1 alocada a Máquina 0

MSI[1] = Max{ MSI[0], MSP[0] } + TE[1] =  1  + 1 = 2
MSP[0] = MSI[1] = 2
```

Figura 4.3: Iteração 2 da execução do Algoritmo 4.1.

```
Iteração 3

Prontas: 2

Mapeando 2

MSI[0] = 1
MSI[1] = 2
MSI[2] = ?
MSI[3] = ?

MSP[0] = 2
MSP[1] = 0
MSP[2] = 0
MSP[3] = 0

MinMaxMSI = Min {

Máquina 0 : Max{ MSI[0], MSP[0] } = 2
Máquina 1 : Max{ MSI[0], MSP[1] } = 1
Máquina 2 : Max{ MSI[0], MSP[2] } = 1
Máquina 3 : Max{ MSI[0], MSP[3] } = 1

}

Tarefa 2 alocada a Máquina 1

MSI[2] = Max{ MSI[0], MSP[1] } + TE[2] =  1  + 1 = 2
MSP[0] = MSI[2] = 2
```

Figura 4.4: Iteração 3 da execução do Algoritmo 4.1.

```
Iteração 4

Prontas: 3

Mapeando 3

MSI[0] = 1
MSI[1] = 2
MSI[2] = 2
MSI[3] = ?

MSP[0] = 2
MSP[1] = 2
MSP[2] = 0
MSP[3] = 0

MinMaxMSI = Min {

Máquina 0 : Max{ MSI[0], MSI[2], MSP[0] } = 2
Máquina 1 : Max{ MSI[0], MSI[2], MSP[1] } = 2
Máquina 2 : Max{ MSI[0], MSI[2], MSP[2] } = 2
Máquina 3 : Max{ MSI[0], MSI[2], MSP[3] } = 2

}

Tarefa 3 alocada a Máquina 0

MSI[3] = Max{ MSI[0], MSI[2], MSP[0] } + TE[3] =  2  + 1 = 3
MSP[0] = MSI[3] = 3
```

Figura 4.5: Iteração 4 da execução do Algoritmo 4.1.

```
Final

MSI[0] = 1
MSI[1] = 2
MSI[2] = 2
MSI[3] = 3

MSP[0] = 3
MSP[1] = 2
MSP[2] = 0

Mapeamento:

[[0, 2, 3], [1]]

Trace de Mapeamento:

     M1  M2
001:000|___|
002:002|001|
003:003|___|
```

Figura 4.6: Estado final da execução do Algoritmo 4.1.

4.4 Adaptações

A partir da ideia básica do Algoritmo 4.1 começamos a desenvolver nossas adaptações para atender o problema do mapeamento de instruções *Dataflow* em Elementos de Processamento. Por isso, a partir de agora voltaremos a nos referir a instruções (conjunto I) e Elemento de Processamento (conjunto P) ao invés de tarefas e máquinas (problema de [6]).

4.4.1 Heurísticas de Ordenação

No passo 5 do Algoritmo 4.1, é necessário remover uma instrução do conjunto de instruções *prontas* para serem mapeadas. No entanto, pode haver mais de uma instrução pronta e a ordem em que as instruções são mapeadas impacta no mapeamento final. Isto é, se a instrução i for mapeada antes da j o mapeamento final pode ser diferente.

Então, estabelecemos alguns critérios de desempate na fila de instruções prontas baseadas em heurísticas. São elas:

- *Fan-in* da instrução i, ou seja, quantas arestas chegam em i.
- *Fan-out* da instrução i, ou seja, quantas arestas saem de i.
- Altura da instrução i.

Note que utilizar a altura só faz sentido nos casos de grafos acíclicos e o grafo *Dataflow* pode conter ciclos. No entanto, isto será resolvido na Subseção 4.4.3. A ideia de utilizar a heurística de altura como critério de desempate se deve ao fato de quanto maior a altura de uma instrução, mais instruções dependem dela indiretamente e por isso deveria ser escalonada com maior prioridade. Já o *fan-in* e *fan-out* seriam medidas de quanto a instrução depende ou é necessária diretamente por outras instruções.

Mesmo com esses critérios, poderiam haver empates durante a etapa da escolha da próxima instrução a ser mapeada. Por isso, decidimos permutar os critérios para obter políticas de ordenação mais refinadas e aproveitar melhor as informações que temos sobre o grafo (*Fan-in*, *Fan-out* e Altura). Note que sempre utilizamos como último critério de desempate o Id da instrução (identificador único) e por isso temos uma ordem total entre as instruções. Cada política representa a ordem em que os critérios de desempate foram aplicados. As políticas utilizadas foram:

- C1: *Fan-in*, *Fan-out*, Altura, Id
- C2: *Fan-in*, Altura, *Fan-out*, Id

- C3: *Fan-out*, *Fan-in*, Altura, Id
- C4: *Fan-out*, Altura, *Fan-in*, Id
- C5: Altura, *Fan-in*, *Fan-out* Id
- C6: Altura, *Fan-out*, *Fan-in*, Id

Após alguns testes, observamos que o melhor critério de desempate na fila de instruções prontas foi a altura, sobretudo na política de ordenação C6. No entanto, esta adaptação acabou não apresentando impacto significativo e por isso não julgamos necessário nos aprofundar nestes resultados.

4.4.2 Comunicação

No mapeamento de instruções *Dataflow*, a comunicação é parte fundamental do problema, impactando fortemente no tempo total de execução do programa. Então, foi necessário incluir esta variável no algoritmo.

Vale lembrar que como estamos considerando uma rede de interconexão homogênea, onde a latência de comunicação entre EPs distintos é constante, L. Os passos 4 a 7 do Algoritmo 4.2 introduzem a variável de comunicação. Se a comunicação é intra-EP (ou seja, j, instrução antecessora a i, já está mapeada em p), o operando fica disponível na unidade de tempo seguinte a sua produção e portanto o cálculo de $MaxMSI_p$ continua o mesmo do algoritmo original (ver passo 12 do Algoritmo 4.1). Porém, se instrução antecessora j está em outro EP, deverá ser acrescido de $(L-1)$, como vemos no passo 7 do Algoritmo 4.2. Note que a latência L é decrementada de 1 unidade de tempo, visto que no após a "L"-ésima unidade de tempo, o operando já estará disponível para o EP e pode executar. Analogamente, não é necessário acrescentar 1 unidade de tempo no caso da comunicação intra-EP.

Para demonstrar uma execução desta versão do algoritmo, vamos utilizar o grafo da Figura 4.7, comum em aplicações do tipo *fork/join*. As Figuras 4.8 a 4.13 ilustram cada iteração do algoritmo. Note que as instruções 1,2 e 3 tem $TE_i = 5$ e latência de comunicação $L = 3$, ou seja, é interessante que as instruções 1, 2 e 3 sejam executadas em paralelo.

4.4.3 Componentes Fortemente Conexas

Como já mencionado algumas vezes, o grafo direcionado *Dataflow* pode conter ciclos, por exemplo, devido a loops do programa. Esta é outra diferença para o problema encontrado em [6], que trata de tarefas dependentes mas descritas por um DAG (*Directed Acyclic Graph*). Portanto, esta foi outra questão que tivemos que tratar.

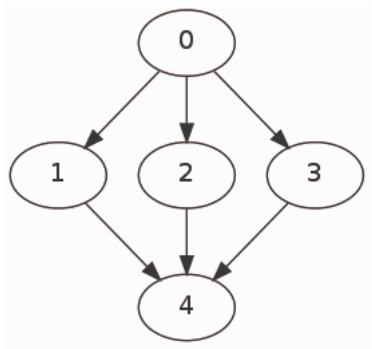


Figura 4.7: Exemplo de Grafo *Dataflow* a ser mapeado utilzando o algoritmo de mapeamento considerando a latência de comunicação L..

```
TE[0] = 1
TE[1] = 5
TE[2] = 5
TE[3] = 5
TE[4] = 1

L=3

Iteração 1

Prontas: 0

Mapeando 0

MSI[0] = ?
MSI[1] = ?
MSI[2] = ?
MSI[3] = ?
MSI[4] = ?

MSP[0] = 0
MSP[1] = 0
MSP[2] = 0
MSP[3] = 0
MSP[4] = 0

Instrução 0 alocada ao EP 0

MSI[0] = TE[0] = 1
```

Figura 4.8: Iteração 1 da execução do algoritmo de mapeamento considerando a latência de comunicação L.

```
Iteração 2

Prontas: 1, 2, 3

Mapeando 1

MSI[0] = 1
MSI[1] = ?
MSI[2] = ?
MSI[3] = ?
MSI[4] = ?

MSP[0] = 1
MSP[1] = 0
MSP[2] = 0
MSP[3] = 0
MSP[4] = 0

MinMaxMSI = Min {

EP 0 : Max{ MSI[0], MSP[0] } = 1
EP 1 : Max{ MSI[0] + (L-1), MSP[1] } = 3
EP 2 : Max{ MSI[0] + (L-1), MSP[2] } = 3
EP 3 : Max{ MSI[0] + (L-1), MSP[3] } = 3
EP 4 : Max{ MSI[0] + (L-1), MSP[4] } = 3

}

Instrução 1 alocada ao EP 0

MSI[1] = Max{ MSI[0], MSP[0] } + TE[1] =  1  + 5 = 6
MSP[0] = MSI[1] = 6
```

Figura 4.9: Iteração 2 da execução do algoritmo de mapeamento considerando a latência de comunicação L.

```
Iteração 3

Prontas: 2, 3

Mapeando 2

MSI[0] = 1
MSI[1] = 6
MSI[2] = ?
MSI[3] = ?
MSI[4] = ?

MSP[0] = 6
MSP[1] = 0
MSP[2] = 0
MSP[3] = 0
MSP[4] = 0

MinMaxMSI = Min {

EP 0 : Max{ MSI[0], MSP[0] } = 6
EP 1 : Max{ MSI[0] + (L-1), MSP[1] } = 3
EP 2 : Max{ MSI[0] + (L-1), MSP[2] } = 3
EP 3 : Max{ MSI[0] + (L-1), MSP[3] } = 3
EP 4 : Max{ MSI[0] + (L-1), MSP[4] } = 3

}

Instrução 2 alocada ao EP 1

MSI[2] = Max{ MSI[0] + (L-1), MSP[1] }  + TE[2] =  3  + 5 = 8
MSP[1] = MSI[2] = 8
```

Figura 4.10: Iteração 3 da execução do algoritmo de mapeamento considerando a latência de comunicação L.

```
Iteração 4

Prontas: 3

Mapeando 3

MSI[0] = 1
MSI[1] = 6
MSI[2] = 8
MSI[3] = ?
MSI[4] = ?

MSP[0] = 6
MSP[1] = 8
MSP[2] = 0
MSP[3] = 0
MSP[4] = 0

MinMaxMSI = Min {

EP 0 : Max{ MSI[0] , MSP[0] } = 6
EP 1 : Max{ MSI[0] + (L-1), MSP[1] } = 8
EP 2 : Max{ MSI[0] + (L-1), MSP[2] } = 3
EP 3 : Max{ MSI[0] + (L-1), MSP[3] } = 3
EP 4 : Max{ MSI[0] + (L-1), MSP[4] } = 3

}

Instrução 3 alocada ao EP 0

MSI[3] = Max{ MSI[0] + (L-1), MSP[2] } + TE[1] =  3  + 5 = 8
MSP[0] = MSI[3] = 8
```

Figura 4.11: Iteração 4 da execução do algoritmo de mapeamento considerando a latência de comunicação L.

```
Iteração 5

Prontas: 4

Mapeando 4

MSI[0] = 1
MSI[1] = 6
MSI[2] = 8
MSI[3] = 8
MSI[4] = ?

MSP[0] = 6
MSP[1] = 8
MSP[2] = 8
MSP[3] = 0
MSP[4] = 0

MinMaxMSI = Min {

EP 0 : Max{ MSI[1], MSI[2] + (L-1), MSI[3] + (L-1), MSP[0] } = 10
EP 1 : Max{ MSI[1] + (L-1), MSI[2], MSI[3] + (L-1), MSP[1] } = 10
EP 2 : Max{ MSI[1] + (L-1), MSI[2] + (L-1), MSI[3], MSP[2] } = 10
EP 3 : Max{ MSI[1] + (L-1), MSI[2] + (L-1), MSI[3] + (L-1), MSP[3] } = 10
EP 4 : Max{ MSI[1] + (L-1), MSI[2] + (L-1), MSI[3] + (L-1), MSP[4] } = 10

}

Instrução 4 alocada ao EP 0

MSI[4] = Max{ MSI[1], MSI[2] + (L-1), MSI[3] + (L-1), MSP[0] } + TE[4] =  10  + 1 = 11
MSP[0] = MSI[4] = 11
```

Figura 4.12: Iteração 5 da execução do algoritmo de mapeamento considerando a latência de comunicação L.

```
Final

MSI[0] = 1
MSI[1] = 6
MSI[2] = 8
MSI[3] = 8
MSI[4] = 11

MSP[0] = 11
MSP[1] = 8
MSP[2] = 8
MSP[3] = 0
MSP[4] = 0

Mapeamento:

[[0, 1, 4], [2], [3]]

Trace de Mapeamento:

    EP0 EP1 EP2
001:000|___|___|
002:001|___|___|
003:001|___|___|
004:001|002|003|
005:001|002|003|
006:001|002|003|
007:___|002|003|
008:___|002|003|
009:___|___|___|
010:___|___|___|
011:004|___|___|
```

Figura 4.13: Estado final da execução do algoritmo de mapeamento considerando a latência de comunicação L.

Esta diferença se refere a como novas tarefas/instruções se tornam prontas para ser mapeadas, como é explicitado no passo 7 do Algoritmo 4.1, "Adicionar a *prontas* tarefas que foram liberadas após mapeamento de i". No contexto das tarefas dependentes representadas por um grafo acíclico, verificar se uma tarefa i foi liberada é simples. Basta verificar através das arestas que incidem em i, se todas as tarefas antecessoras de i já foram mapeadas. Se as antecessoras já foram mapeadas, seus respectivos *makespans* já foram calculados e portanto já é possível mapear i e calcular MSI_i.

No caso das instruções *Dataflow*, existe uma diferença sutil para determinar se uma instrução está liberada para ser mapeada. Como ciclos são permitidos, verificar apenas as dependências (arestas) implicaria em que as instruções do ciclo nunca se tornariam liberadas para ser mapeadas, pois ficaram em dependência circular.

Este problema pode ser contornado. Como explicado na Seção 3.2, no grafo direcionado *Dataflow*, o número de arestas (dependências) que incide em um nó (instrução) é diferente do número de portas (operandos) da instrução. Vale lembrar que para executar a instrução necessita receber operandos por todas suas portas de entrada e não por todas as suas arestas. Então, ao invés de verificar se todas as instruções antecessoras a i já foram mapeadas através do número de arestas que incidem em i, bastaria verificar através do número de portas de entrada i. Com isso, estaríamos adaptando o algoritmo original para atuar também em grafos com ciclos. Inclusive, nos experimentos do Capítulo 5 esta abordagem será testada. No entanto, quisemos testar outra abordagem para tratar o mapeamento de grafos *Dataflow* com ciclos.

Na etapa inicial da investigação desta dissertação, quando já possuíamos o Simulador, experimentamos gerar todas os mapeamentos possíveis para alguns programas

Algoritmo 4.2 Alocação de uma instrução i considerando a comunicação

Entrada: Uma instrução i

Saída: Mapeamento de i em um EP P e seu *makespan* MSI_i

1: **procedimento** mapear(i)
2: **para todo** EPs $p \in P$ **faça**
3: **para todo** Instruções j imediatamente antecessoras a i **faça**
4: **Se** j está mapeada em p **então**
5: $MaxMSI_p \Leftarrow \max\{MSI_j, MSP_p\}$
6: **senão**
7: $MaxMSI_p \Leftarrow \max\{MSI_j + (L-1), MSP_p\}$
8: $MinMaxMSI \Leftarrow \min\{MaxMSI_p\}$
9: $MSI_i \Leftarrow MinMaxMSI + TE_i$
10: $MSP_p \Leftarrow MSI_i$
11: Mapear i no EP p onde $MinMaxMSI$ é mínimo
12: **fim procedimento**

e com isto obter o mapeamento ótimo, isto é, o mapeamento que possibilitasse a execução do programa em menos ciclos de máquina. Isto só pode ser feito para programas pequenos, por exemplo, um programa de apenas 12 instruções permite 4.213.597 mapeamentos possíveis. No entanto, isto nos foi útil para fornecer pistas sobre bons mapeamentos. Nestes experimentos, observamos que nos mapeamentos ótimos as instruções pertencentes a uma Componente Fortemente Conexa (CFC) do grafo *Dataflow* estavam no mesmo Elemento de Processamento.

Relembrando a definição de Componente Fortemente Conexa, de acordo com [21]:

> "Uma componente fortemente conexa de um digrafo $G = (V, E)$ é um conjunto maximal de vértices $C \subseteq V$ tal que para cada par de vértices u e $v \in C$, temos $u \rightsquigarrow v$ e $v \rightsquigarrow u$, isto é, os vértices u e v se alcançam mutuamente."

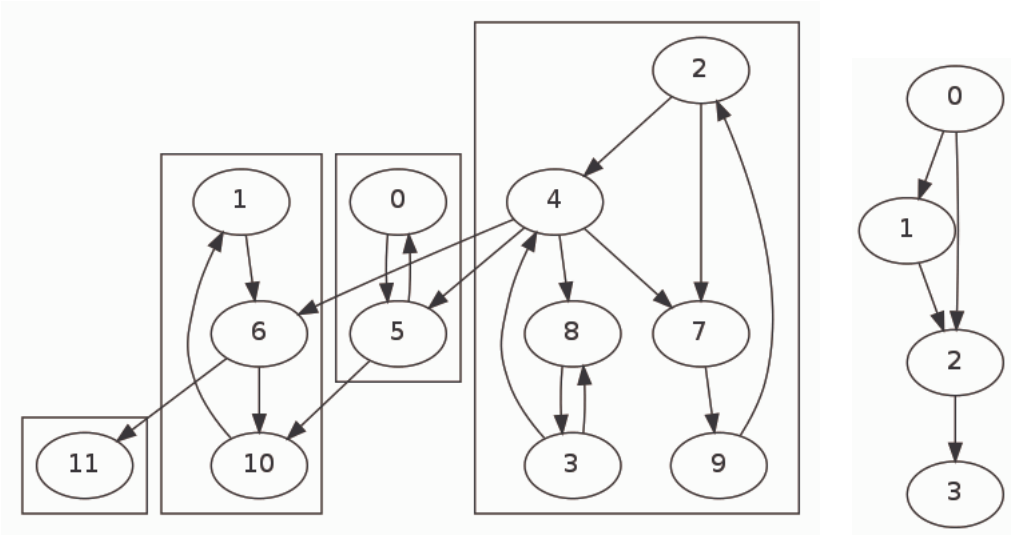
Outro experimento preliminar, foi a implementação de um algoritmo genético que gera aleatoriamente mapeamentos e os executa no Simulador. A partir dos melhores mapeamentos (elitismo), se realizava o *crossing-over* para obter outros mapeamentos e o procedimento era repetido N vezes. No fim, obtinha-se um mapeamento pseudo-ótimo, pois era o melhor mapeamento obtido entre os gerado mas não havia garantia que era ótimo de fato. Cabe ressaltar que este algoritmo genético não se tratava de um algoritmo de geração de mapeamento e sim de um algoritmo de avaliação de mapeamentos, pois era necessário executar o programa *Dataflow* no Simulador para sabermos em quantos ciclos de máquina o programa executou. Nestes testes, observando as mapeamentos pseudo-ótimos, as instruções pertencentes a uma CFC do grafo *Dataflow* também estavam no mesmo Elemento de Processamento.

Além disso, intuitivamente há um acoplamento entre instruções de uma CFC já que todos os vértices são mutuamente alcançáveis e consequentemente todas as instruções que são dependentes entre si trocam operandos. Mapear todas as instruções de uma CFC em um mesmo EP garante que não se pagará a latência de comunicação, pois toda comunicação entre elas será intra-EP. Em último análise, esta seria uma hipótese a ser testada e foi isso que fizemos nesta dissertação.

Para encontrar as componentes fortemente conexas foi utilizado o algoritmo de Kosaraju-Sharir. Sua descrição segue no Algoritmo 4.3.

Ao mapear todas as instruções de uma CFC em um mesmo EP, na verdade estamos tratando toda a CFC com uma instrução. Isto é, estamos agrupando todas as instruções pertencentes a CFC em um nó no grafo. Com isso, transformamos o grafo original *Dataflow*, que poderia ter ciclos, em um grafo acíclico, visto que todos os ciclos ficariam dentro das CFCs. A Figura 4.4.3 ilustra um exemplo com esta mudança. Portanto, ao considerar o agrupamento em CFCs, o problema

do mapeamento de instruções Dataflow também se transforma em como mapear um grafo acíclico. No entanto, surge o problema de como considerar o tempo de execução de uma CFC. Isto será discutido na Seção 4.4.4.



(a) Exemplo de grafo divido em CFCs.

(b) Grafo acíclico resultante do agrupamento das CFCs em instruções.

Figura 4.14: Exemplo de transformação do grafo *Dataflow* original em grafo acíclico através do agrupamento de CFCs.

Algoritmo 4.3 Algoritmo Kosaraju-Sharir, para encontrar Componentes Fortemente Conexas em um Grafo Direcionado.

Entrada: Um grafo direcionado G(I,E)
Saída: Lista de Componentes Fortemente Conexas de G

1: Iniciar uma *pilha* vazia
2: **enquanto** A *pilha* não conter todos os vértices de I **faça**
3: Escolher um vértice i tal que $i \notin pilha$
4: Executar busca em profundidade a partir de i e adicionar cada vértice expandido a *pilha*
5: **fim enquanto**
6: Reverter as arestas do grafo direcionado G
7: **enquanto** A *pilha* não estiver vazia **faça**
8: Desempilhar i do topo da *pilha*
9: Executar busca em profundidade a partir de i e adicionar cada vértice expandido uma CFC
10: Adicionar a CFC a lista de CFCs e remover os vértices da CFC da *pilha*
11: **fim enquanto**

4.4.4 Tempo de Execução Personalizado

Cada instrução tem um tempo de execução, TE_i. No entanto, ao agruparmos instruções em CFCs, deixamos de tratar as instruções individualmente. Então, surge o problema de como determinar o tempo de execução de uma CFC. Na prática, não sabemos o tempo real de execução de uma CFC, já que sua execução depende da entrada do programa. Como estimativa do tempo de execução de uma CFC, consideramos $TE_{cfc} = \sum TE_i$, $\forall i \in$ CFC, ou seja o somatório dos tempo de execução de cada instrução que compõe a CFC. No entanto, isto pode não expressar o tempo de espera das instruções sucessoras. Vejamos o exemplo da Figura 4.15.

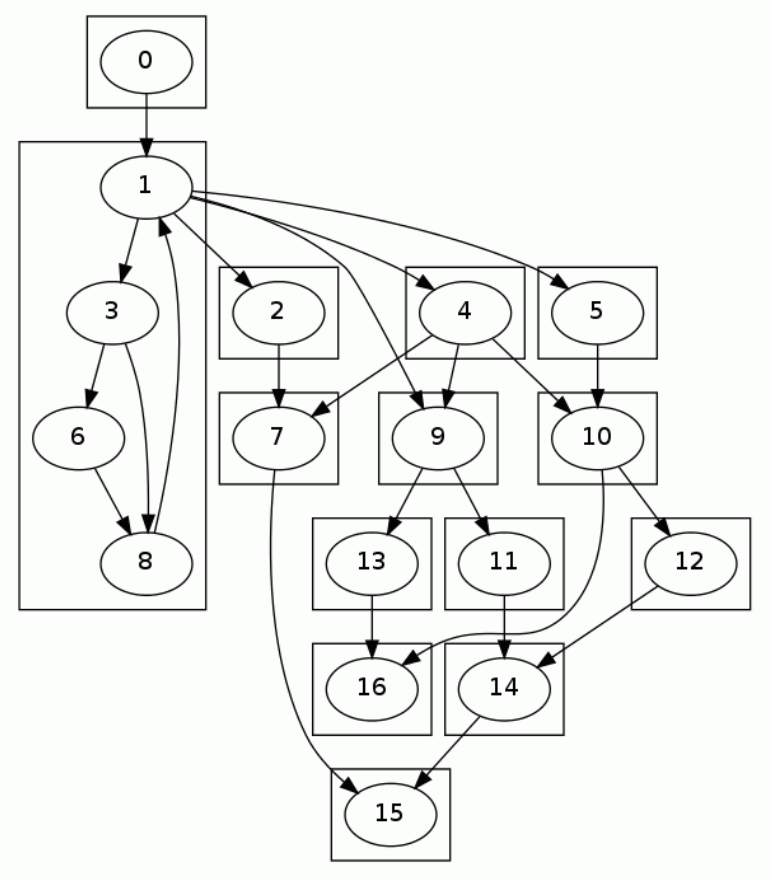


Figura 4.15: Exemplo onde o somatório do tempos execução de cada instrução não é uma boa medida para o cálculo do *makespan* da CFC sucessora.

No exemplo da Figura 4.15, vamos supor que cada instrução originalmente tem $TE_i = 1$. Temos a CFC formada por [1,3,6,8] e as demais CFCs são formadas apenas por uma instrução. Considerando o somatório de TE_i, a CFC [1,3,6,8] tem $TE_i = 4$. No entanto, observe as instruções 2, 4, 9 e 5. Elas dependem da CFC [1,3,6,8]. Em particular, note que elas dependem apenas das instrução 1, que faz parte do caminho

crítico da instrução anterior, no caso a instrução 0, até as instruções sucessoras 2, 4, 9 e 5. Portanto, para as instruções 2, 4, 9 e 5, a CFC [1,3,6,8] deveria ter tempo de execução de $TE_1 = 1$. Note que o TE_i poderia variar de acordo com a instrução sucessora. Por isso, para o cálculo do *makespan* da CFC sucessora j, ao invés de considerar o TE_i, consideramos o $TEP_{i,j}$, Tempo de Execução Personalizado da CFC i perante a CFC sucessora j.

Para calcular $TEP_{i,j}$ devemos considerar o caminho crítico entre as CFCs z (antecessoras de i) e a CFC sucessora j, passando pela CFC i. Por isso definimos um conjunto de instruções de entrada a CFC i, *inputs*. Este conjunto é composto pelas instruções imediatamente antecessoras a i (pertencentes a z) e da própria CFC i (no caso em que recebem operandos iniciais). Também definimos um conjunto de instruções de saída, *outputs* $\in$ CFC j, com as instruções imediatamente sucessoras a CFC i. Então, calculamos o maior caminho entre a instruções *inputs* e *outpus*, $\forall a \in inputs$ e $\forall b \in outputs$. Finalmente, definimos $TEP_{i,j}$ como a maior distância dos *outputs* $\in$ CFC j.

O cálculo do maior caminho entre a instrução de entrada a e a instrução de saída b é a realizado através de uma busca em profundidade a partir da instrução a. Esta busca está limitada às instruções que pertencem a CFC i e a instrução b. Cada instrução tem um valor de distância associado (inicialmente 0). Cada vez que uma instrução é explorada, ela recebe o valor da distância da instrução antecessora + 1. Para garantir o maior caminho, uma instrução só é explorada se seu valor de distância for menor que distância da instrução antecessora + 1. O tamanho do maior caminho é o valor final da distância da instrução de saída b.

Vejamos o exemplo da Figura 4.16. As instruções que estão marcadas com um ponto preto, recebem operandos iniciais e por isso também devem ser consideradas instruções de entrada. Na Figura 4.17 temos o cálculo do $TEP_{i,j}$ para todas as componentes que possuem sucessoras.

Um detalhe a ser ressaltado fica por conta do cálculo dos *makespans* das instruções e dos processadores. Considerando a CFC cfc_i e cfc_j sua CFC sucessora, para o cálculo do MSI_{cfc_i}, ainda deve ser considerado o somatório de todos os TE_i, pois para uma CFC ser considerada terminada todas as suas instruções devem terminar. Analogamente, isto também se aplica ao MSP_p, sendo p o EP que cfc_i foi alocado, pois o processador ficará alocado a todas instruções da CFC. O cálculo do Tempo de Execução Personalizado só influencia no *makespan* de j, que terá seu início adiantado de acordo com o caminho crítico de i.

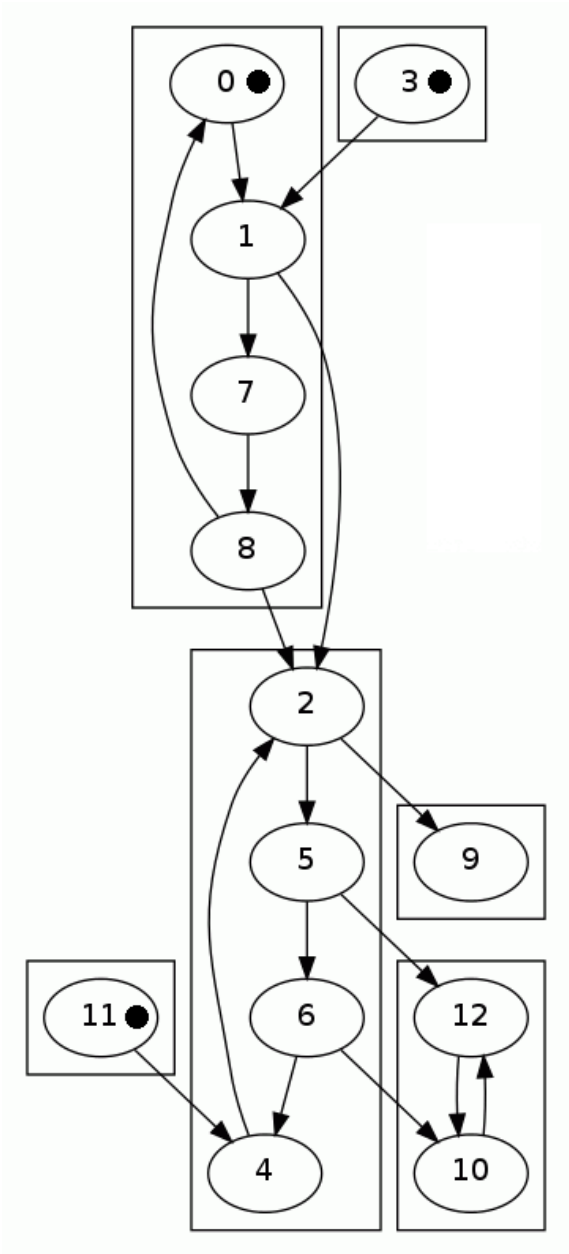


Figura 4.16: Exemplo de grafo onde os Tempos de Execução Personalizados serão calculados.

4.5 O Algoritmo

Nesta Seção apresentaremos a forma final do algoritmo de mapeamento proposto (ver Algoritmo 4.4). Relembrando o caminho percorrido, partimos de um algoritmo de programação dinâmica sugerido por [6], onde o cálculo do *makespan* é realizado de acordo com as dependências das tarefas anteriores. Busca-se minimizar o *makespan* da instrução, considerando todos EPs possíveis e respeitando a restrição do maior

```
CFC [1, 7, 8, 0] - inputs[0, 3] -outputs[2]
dist(0,2)=4
dist(3,2)=3
TEP[1, 7, 8, 0][2]=4
TEP[1, 7, 8, 0][5, 6, 4, 2]=4

CFC [11] - inputs[11] -outputs[4]
dist(11,4)=1
TEP[[11]][4]=1
TEP[[11]][5, 6, 4, 2]=1

CFC [3] - inputs[3] -outputs[1]
dist(3,1)=1
TEP[[3]][1]=1
TEP[[3]][1, 7, 8, 0]=1

CFC [5, 6, 4, 2] - inputs[1, 8, 11] -outputs[9, 10, 12]
dist(1,9)=1
dist(8,9)=1
dist(11,9)=2
TEP[5, 6, 4, 2][9]=2

dist(1,10)=3
dist(8,10)=3
dist(11,10)=4
TEP[5, 6, 4, 2][10]=4
dist(1,12)=2
dist(8,12)=2
dist(11,12)=3
TEP[5, 6, 4, 2][12]=3
TEP[5, 6, 4, 2][10,12]= max {TEP[5, 6, 4, 2][10] , TEP[5, 6, 4, 2][12]} = 4
```

Figura 4.17: Cálculo dos Tempos de Execução Personalizados do grafo da Figura 4.16.

makespan da instrução antecessora.

Além disso, introduzimos prioridades na fila de instruções *prontas*, de acordo com as políticas descritas na Seção 4.4.1. Escolhemos utilizar a política C6. O cálculo das prioridade e a priorização são realizados nos passos 3 e 9 do Algoritmo 4.4, respectivamente

Em seguida, adaptamos o algoritmo inicial para considerar a comunicação inter-EP. Por isso verificamos se a instrução será alocada ao mesmo EP da instrução antecessora e, em caso positivo, acrescentamos a latência de comunicação L. Este procedimento é expresso nos passos 17 a 20 do Algoritmo 4.4.

Depois, por ter de lidar com grafos que possuem ciclos, propusemos o agrupamento de instruções que pertencem a uma mesma CFC. Com isso transformamos o grafo com ciclos em grafo acíclico. Para encontrar as Componentes Fortemente Conexas do grafo utilizamos o Algoritmo de Kosaraju-Sharir (passo 1). Em seguida, construímos o grafo acíclico G' (passo 2).

E por fim, realizamos o cálculo Tempos de Execução Personalizados (passo 4), para obter uma melhor estimativa do tempo de execução de uma CFC, em relação às suas sucessoras. Note que no passo 16, "$MSI'_j \Leftarrow MSI_j - TEj + TEPj, i$", é descontado o tempo de execução da instrução (somatório dos tempos de execução de cada instrução da CFC de j) e é acrescido o Tempo de Execução Personalizado, justamente para personalizar o *makespan* para a CFC j.

Algoritmo 4.4 Forma final do algoritmo de geração de mapeamento com as adaptações propostas.

Entrada: Programa *Dataflow* descrito por $G(I, E)$, os Elementos de Processamento P, os tempo de execução das tarefas TE_i, a latência de comunicação inter-EP L e os operandos iniciais O

Saída: Um mapeamento $M(I \rightarrow P)$

1: Encontrar CFCs em G (Algoritmo de Kosaraju-Sharir)
2: Gerar Grafo Acíclico G' a partir das CFCs
3: Calcular Prioridades das CFCs (Fan-in, Fan-out, Altura)
4: Calcular Tempos de Execução Personalizados, TEP_i
5: Iniciar lista de CFCs *prontas*
6: **para todo** CFC $i \in prontas$ **faça**
7: $MSI_i \Leftarrow TE_i$
8: **enquanto** Existirem CFCs *prontas* a serem escalonadas **faça**
9: Remover a CFC $i \in prontas$ de maior Prioridade
10: mapear(i)
11: Adicionar a *prontas* CFCs que foram liberadas após mapeamento de i
12: **fim enquanto**
13: **procedimento** mapear(i)
14: **para todo** EP $p \in P$ **faça**
15: **para todo** CFC j imediatamente antecessoras a i **faça**
16: $MSI'_j \Leftarrow MSI_j - TEj + TEPj, i$
17: **Se** j está mapeada em p **então**
18: $MaxMSI_p \Leftarrow \max\{MSI'_j, MSP_p\}$
19: **senão**
20: $MaxMSI_p \Leftarrow \max\{MSI'_j + (L-1), MSP_p\}$
21: $MinMaxMSI \Leftarrow \min\{MaxMSI_p\}$
22: $MSI_i \Leftarrow MinMaxMSI + TE_i$
23: $MSP_p \Leftarrow MSI_i$
24: Mapear i no EP p onde $MinMaxMSI$ é mínimo
25: **fim procedimento**

Capítulo 5

Experimentos e Resultados

Neste Capítulo apresentaremos um estudo comparativo entre algoritmos de mapeamento e avaliaremos o Algoritmo proposto. Começaremos descrevendo a metodologia utilizada nos experimentos. Em seguida, apresentaremos cada um dos algoritmos de mapeamento que comparamos no estudo. Também descreveremos o conjunto de programas *Dataflow* implementados que serviu de *benchmark*. Por fim, apresentaremos os resultados obtidos.

5.1 Metodologia

Para testar o algoritmos de mapeamento precisávamos de um conjunto de programas *Dataflow* representativo. Buscamos na literatura e em [6] é utilizado um conjunto de grafos acíclicos gerados aleatoriamente. Não seria adequado utilizar esta abordagem aqui, justamente porque o algoritmo que projetamos também é destinado para lidar com ciclos. Já em [7], os autores utilizaram trechos de programas do SpecInt e do SpecFP, mas por não possuirmos um ambiente de testes compatível com o do WaveScalar, seria difícil adaptar esses programas. Por isso, optamos por implementar o nosso próprio conjunto de *benchmarks*. Foram considerados 13 programas de testes e serão descritos na Seção 5.3.

Para montar conjunto de *benchmarks*, primeiramente implementamos os programas na linguagem C. Depois utilizamos o compilador *Couillard* para compilar os programas. Ele recebe como entrada um programa em C e gera o *assembly* da máquina virtual *Trebuchet* e a descrição do grafo no formato Graphviz (arquivo `.dot`). Implementamos um conversor que, dado um grafo no formato Graphviz, faz a conversão para o arquivo de entrada do Simulador (arquivo `.sim`). Com isso, conseguimos rodar os programas em C em nosso Simulador. Vale lembrar que compilador *Couillard* é voltado originalmente para utilização na máquina virtual *Trebuchet* e por isso possui algumas limitações para o nosso ambiente (ver Seção 2.2.3).

Através do conversor, também foi possível fazer adaptações e inserir algumas

instruções que não estavam presentes no programa original, como a instrução `OUT` (saída de dados). Através dela foi possível comparar os resultados da saída do Simulador, com a saída do programa em C e garantir a correta implementação do programa.

O nosso algoritmo foi implementado, bem como suas variações. Então, buscamos outros algoritmos de mapeamento para servir como base de comparação. Através da descrição em [7], foi possível implementá-los. Ao todo foram considerados 7 algoritmos (ver Seção 5.2). Assim, pudemos aplicar os algoritmos de mapeamento nos programas de *benchmarks* e avaliar os resultados.

Também precisávamos testar como os algoritmos reagiriam ao aumento de latência da comunicação inter-EP, L. Isto é, aumentando o custo de comunicação ente EPs, como os algoritmos de mapeamento iriam se comportar. Lembrando que L é um parâmetro do Simulador. Por isso, fizemos 15 execuções, variando a latência de 1 a 15 ciclos de máquina. Como os resultados se mostraram lineares, vamos discutir 3 amostras (5, 10, 15 ciclos), para indicar como a tendência do aumento da latência impacta nos resultados. Maiores detalhes sobre a latência serão discutidos na Seção 5.4.

Executando um programa no Simulador, podemos gerar *traces* de execução e algumas medidas de interesse como a utilização dos EPs, quantidade de ciclos ociosos, média de mensagens (operandos) enviados por ciclo, etc. No entanto, o objetivo do algoritmo de mapeamento é minimizar o tempo de execução total do programa, em ciclos de máquina. Por isso, esta foi a medida comparada entre cada execução. Assim, podemos comparar se um mapeamento gerado foi melhor que outro comparando o número de ciclos de máquina que um programa executou, utilizando um determinado mapeamento e sob uma latência L.

A Figura 5.1 ilustra o fluxo utilizado para a realização dos experimentos.

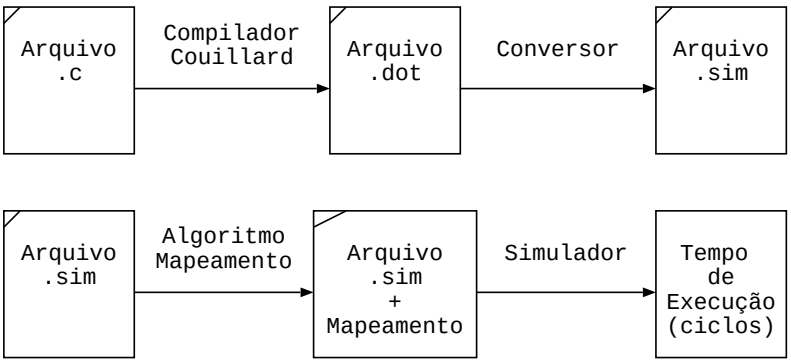


Figura 5.1: Fluxo executado para avaliação de um programa em nossa plataforma de testes.

5.2 Algoritmos

Nesta Seção iremos descrever os algoritmos de mapeamento utilizados neste estudo comparativo. Comparamos o Algoritmo proposto e algumas de suas variações. Além deles, também incluímos outros algoritmos encontrados na literatura. Segue a descrição de cada um dos algoritmos e um exemplo de mapeamento gerado para um grafo (Figura 5.2).

5.2.1 Programação Dinâmica

Este algoritmo é baseado em [6] e já foi descrito na Seção 4.3. Como dito anteriormente, foi este algoritmo serviu de base para o algoritmo final proposto. Nesta versão, para uma comparação mais realista, também acrescentamos a componente da comunicação, que é explicada na Seção 4.4.2.

Além disso, tivemos que fazer uma pequena alteração para que o algoritmo pudesse lidar com grafos com ciclos. Para uma instrução ser considerada pronta para ser executada (ou liberada para ser mapeada), o número de dependências a serem satisfeitas deve ser o mesmo número de portas (operandos), ao invés do número de arestas. Com isso, evitamos que o algoritmo de mapeamento fique em dependência circular em grafos com ciclos.

Esta versão do algoritmo não possui heurísticas de ordenação, agrupamento em Componentes Fortemente Conexas e Tempos Personalizados de Execução. Por uma questão de notação, vamos nos referir a este algoritmo como `progdin`.

5.2.2 Componentes Fortemente Conexas

Este é uma versão do algoritmo proposto, utilizando o agrupamento de instruções que pertençam a uma mesma Componentes Fortemente Conexas. No entanto, não é utilizada a adaptação dos Tempos de Execução Personalizados. Por questão de notação, vamos nos referir a esta versão como `cfc`.

5.2.3 Tempos de Execução Personalizados

Este é a versão final do algoritmo proposto apresentada na Seção 4.5. Decidimos testar esta versão separadamente justamente para medir a contribuição de se utilizar os Tempos de Execução Personalizados no tempo total de execução do programa. Vamos nos referir a esta versão como `cfc + tep`.

5.2.4 *Static-snake*

Este foi um dos algoritmos que encontramos em [7], onde o mapeamento é estudado na arquitetura WaveScalar. É importante lembrar que a arquitetura WaveScalar é espacial, ou seja, existe uma matriz de EPs e a latência de comunicação é heterogênea. O algoritmo mapeia as instruções preenchendo os EPs como se fosse uma "cobra" (*snake*), da esquerda para a direita na primeira linha, da direita para a esquerda na segunda linha e assim em diante.

Outra diferença é que em [7], o número de EPs é fixo. No nosso estudo, decidimos não fixar o número de EPs. Então, para termos uma comparação justa, decidimos utilizar o mesmo número de EPs que o mapeamento gerado pelo algoritmo `cfc + tep`. Por isso, antes executamos o `cfc + tep` e de posse do número de EPs utilizados, executamos o *Static-snake*.

Como não existe a matriz de EPs, a instruções são dividas de maneira uniforme entre os EPs. Note que a ordem de compilação das instruções é mantida para tentar preservar a localidade. Portanto, divide-se as N instruções na ordem do programa compilado em X EPs. Como a divisão entre EPs pode não ser exata, as instruções restantes são distribuídas igualmente entre os EPs. Isto é, N instruções divididas por X EPs, temos quociente Q e resto R. Os $X - R$ EPs terão Q instruções e os R EPs restantes terão $Q + 1$ instruções. Por exemplo, um programa com 152 instruções, ao ser dividido entre 13 EPs (quociente=11 e resto=9): 4 EPs ficarão com 11 instruções e 9 EPs ficarão com 12 instruções. Por questão de notação, vamos nos referir a este algoritmo simplesmente como `snake`.

5.2.5 *Depth-first-snake*

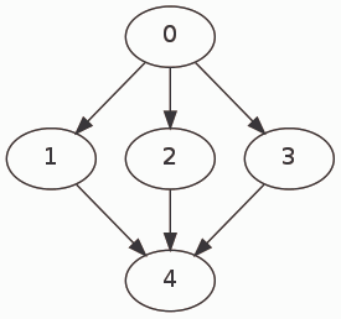
Este algoritmo também foi descrito em [7]. É análogo ao *Static-snake*, só que ao invés de utilizar a ordem de compilação das instruções, é executada uma busca em profundidade no grafo *Dataflow* em pré-ordem. A divisão de instruções ocorre sobre a pré-ordem da mesma maneira que no *Static-snake*. Vamos nos referir a este algoritmo como `profundidade`.

5.2.6 *Breadth-first-snake*

É a versão análoga ao *Depth-first-snake*, só que utilizando a busca em largura. Esta versão não consta em [7], mas decidimos implementá-la mesmo assim para termos uma espécie de grupo de controle e comparar se realmente faz mais sentido utilizar a busca em profundidade que a busca em largura para gerar um mapeamento. Por questão de notação, vamos nos referir a este algoritmo simplesmente como `largura`.

5.2.7 Sem comunicação inter-EP

Decidimos também testar um mapeamento em que todas as instruções fosse alocadas a um mesmo EP. Desta maneira, não há paralelismo mas também não há *overhead* de comunicação. Esta situação é ilustrada na Figura 1.3(a). Assim, buscamos obter um limite superior para a latência de comunicação L. Quando a melhor solução é alocar todas as instruções a um mesmo EP significa que, para a latência utilizada, não há paralelismo suficiente no programa para que valha a pena com comunicação. Vamos nos referir a este algoritmo como `1ep`.



(a) Grafo dataflow de exemplo, $L = 3$, $TE_0 = TE_4 = 1$ e $TE_1 = TE_2 = TE_3 = 5$.

Algoritmo	Mapeamento	Tempo de Execução
progdin	[[0, 3, 4], [2], [1]]	12
cfc	[[0, 1, 4], [2], [3]]	12
cfc + tep	[[0, 1, 4], [2], [3]]	12
snake	[[0, 1], [2, 3], [4]]	16
profundidade	[[0, 1], [4, 2], [3]]	11
largura	[[0, 1], [2, 3], [4]]	16
1ep	[[0, 1, 2, 3, 4]]	17

(b) Mapeamentos gerados.

Figura 5.2: Exemplo de mapeamentos gerados através do algoritmos.

5.3 *Benchmarking*

Nesta Seção descreveremos como foi montado o conjunto de programas utilizados nos testes dos algoritmos de mapeamento. Como em todo *benchmarking*, nosso desafio era montar um conjunto de programas que fosse representativo, isto é, que representasse o comportamento da maioria dos programas *Dataflow*.

Como o nosso foco é o mapeamento, ao invés de buscarmos implementar aplicações específicas (ex: *equake*, *gzip*, *fft*), o que seria muito mais custoso em termos de implementação, decidimos focar no comportamento do grafo em si. Isto é, se o grafo contém ciclos ou não, o tamanho dos ciclos, o quanto paralelismo existe entre instruções.

Então foram implementados programas simples, que chamamos de blocos básicos. E a partir destes blocos básicos, montamos programas mais complexos, através de variações dos blocos básicos. Estas variações serão explicadas adiante.

Foram implementados 3 blocos básicos. Um bloco com um grafo acíclico, um bloco com um loop simples (e consequentemente contendo ciclos no grafo), e um bloco contendo um loop aninhado. No Apêndice A apresentamos o código dos pro-

gramas em C. Estes programas e as variações do blocos básicos foram compilados com o compilador *Couillard* para possibilitar a execução no Simulador.

O bloco acíclico é uma grande expressão de somas e multiplicações gerada aleatoriamente. Dependendo da ordem das operações, elas podem ser executadas em paralelo ou não. Apesar de não ser o foco do nosso algoritmo proposto, os programas acíclicos foram importantes para validar a ideia inicial da programação dinâmica e da comunicação. O grafo *Dataflow* do bloco acíclico pode ser visto na Figura 5.3.

No caso do bloco do loop simples, trata-se de uma multiplicação por sucessivas somas. O laço é repetido 10 vezes. Observe na Figura 5.4 que temos a presença de ciclos no grafo e consequentemente estes ciclos forma componentes fortemente conexas. Lembrando que as instruções que compõe uma CFC serão mapeadas em um mesmo EP pelos algoritmos `cfc` e `cfc + tep`.

O bloco do loop aninhado também é uma repetição de sucessivas somas. O laço externo faz 2 repetições enquanto o interno faz 5. No total também temos 10 iterações assim como no programa do loop simples. O grafo *Dataflow* do bloco ciclo aninhado pode ser visto na Figura 5.5.

Depois de termos os blocos básicos, pudemos compor novos programas fazendo combinações entre eles. Assim, pudemos gerar programas com maior número de instruções e testar versões com maior ou menor paralelismo. Para cada um dos 3 blocos básicos, fizemos 4 variações: apenas o bloco básico, serial, paralela e serial/paralela. Além destes, fizemos um programa misto com a presença de cada bloco básico, totalizando os 13 programas do *benchmarking*.

Nos programas compostos, replicamos o bloco básico 4 vezes. Para montar os programas do tipo serial, utilizamos um operando de saída de um bloco básico, como operando de entrada de outro bloco básico, garantindo assim que haja uma dependência entre os blocos. Para montar as versões do tipo paralelo, adicionamos dependência através dos operandos de saída de cada bloco. Isto pode ser feito somando os resultados de cada bloco básico. As Figuras 5.6, 5.7, 5.8 e 5.9 ilustram as variações dos programas.

A Tabela 5.1 sumariza algumas informações sobre o conjunto de programas. Note que os programas acíclicos tem o mesmo número de instruções e componentes fortemente conexas, já que as componente fortemente conexas são compostas por apenas uma instrução. Os programas com um ciclo simples possuem CFCs pequenas e que não variam muito de tamanho. Já os programas com ciclo aninhado possuem uma grande CFC, composta por 15 instruções e suas CFCs variam muito de tamanho. Estas informações serão importantes para interpretarmos os resultados.

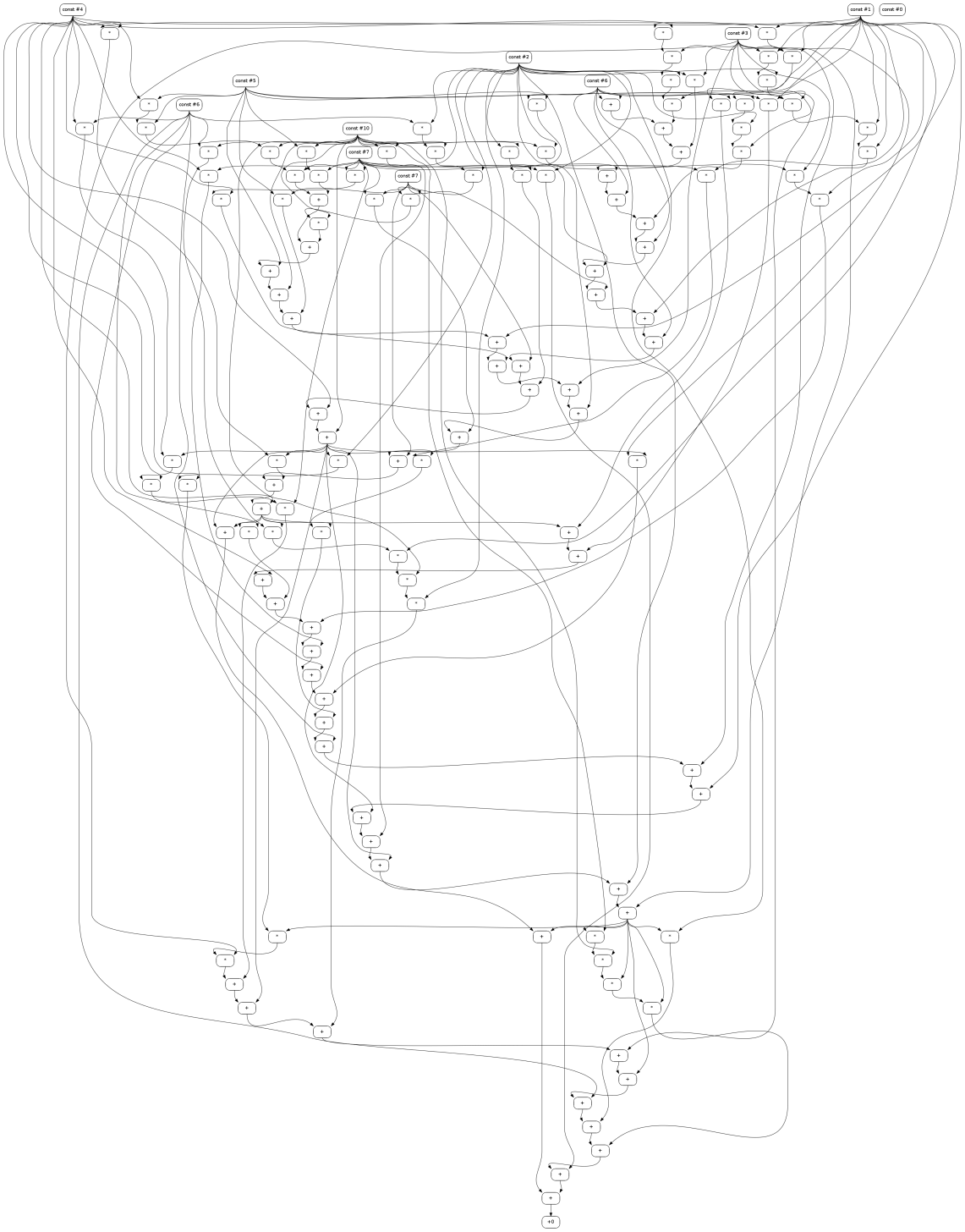


Figura 5.3: Bloco básico acíclico.

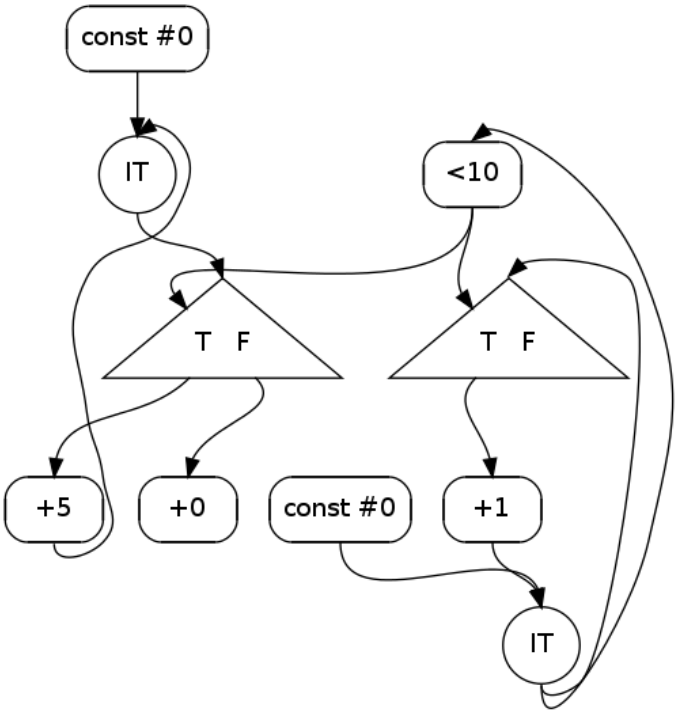


Figura 5.4: Bloco básico com ciclo.

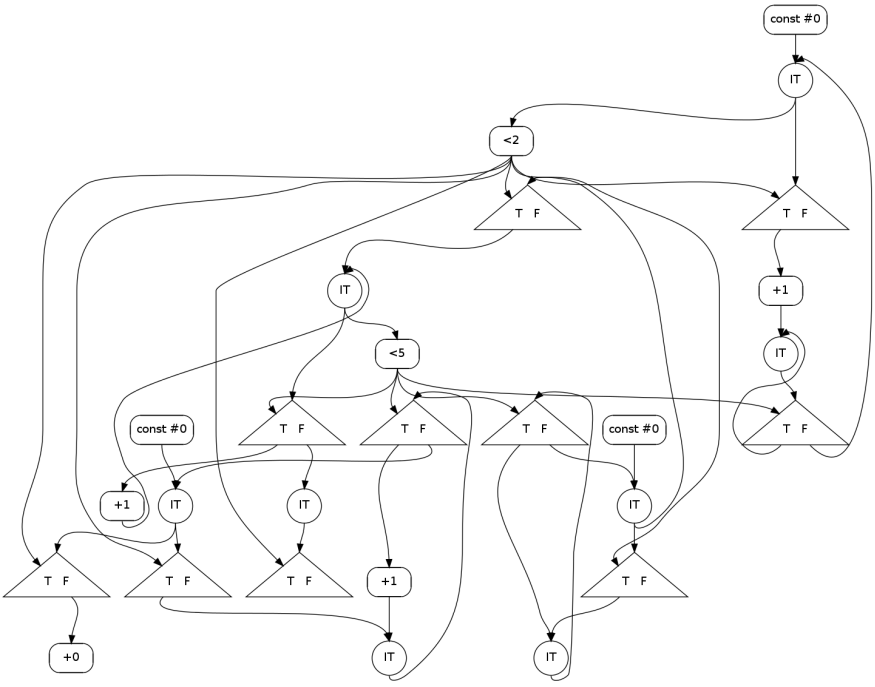


Figura 5.5: Bloco básico com ciclo aninhado.

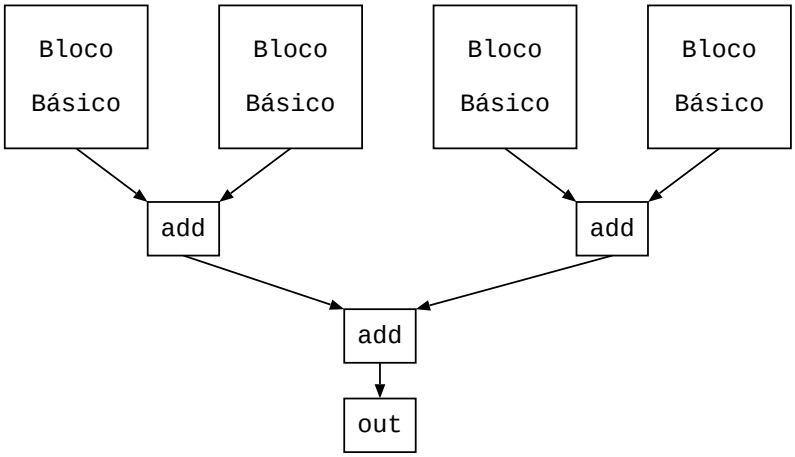


Figura 5.6: Programa com 4 blocos básicos em paralelo.

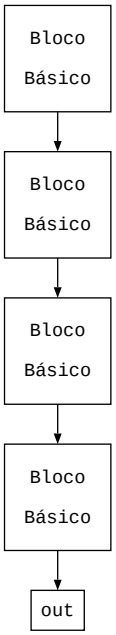


Figura 5.7: Programa com 4 blocos básicos de maneira serial.

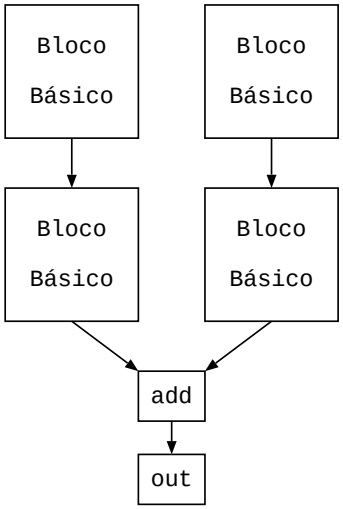


Figura 5.8: Programas com blocos de maneira serial e em paralelo.

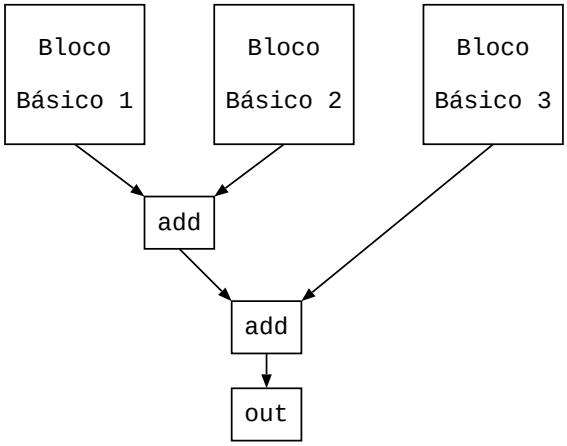


Figura 5.9: Programa misto, com a presença de cada um dos blocos básicos.

	Nº de Instruções	Nº de CFCs	Tam. Maior CFC	Média Tam. CFC	Variância Tam. CFC
acíclico	135	135	1	1	0
acíclico_paralelo	540	540	1	1	0
acíclico_serial	537	537	1	1	0
acíclico_serpar	538	538	1	1	0
ciclo	10	5	4	2	2
ciclo_paralelo	40	20	4	2	1,68
ciclo_serial	61	29	4	2,1	1,02
ciclo_serpar	46	22	4	2,09	1,41
ciclo_aninhado	28	10	15	2,8	19,95
ciclo_aninhado_paralelo	112	40	15	2,8	18,42
ciclo_aninhado_serial	228	85	15	2,69	9,76
ciclo_aninhado_serpar	150	54	15	2,77	14,13
misto	175	152	15	1,15	1,46

Tabela 5.1: Sumário dos programas utilizados no *benchmarking*.

5.4 Resultados

Depois de ter os programas do *benchmarking* e os algoritmos de mapeamento prontos, pudemos finalmente executar os experimentos. Para cada programa do conjunto, foi gerado um mapeamento através dos algoritmos `progdin`, `cfc`, `cfc + tep`, `snake`, `profundidade`, `largura` e `1ep`. Em seguida, o programa foi executado no Simulador utilizando o mapeamento gerado. Então, obtínhamos através do Simulador o tempo total de execução do programa (em ciclos de máquina) utilizando um determinado algoritmo de mapeamento.

Durante a execução do programa, a latência de comunicação L da rede de interconexão é um parâmetro importante, já é o que determina o quão cara é a comunicação entre EPs. Por isso, variamos L para observar como os algoritmos de mapeamento se comportariam. Note que os algoritmos propostos `progdin`, `cfc` e `cfc + tep` são sensíveis ao latência de comunicação, já que utilizam o valor L para estimar os *makespans*. No experimentos, variamos o parâmetro de latência no Simulador de 1 a 15 ciclos. Os resultados apresentam uma tendência de crescimento bem definida e por isso optamos por apresentar os resultados para L para os valores de 5, 10 e 15 ciclos, sem perda de generalidade.

Os resultados são apresentados através das Tabelas 5.2, 5.3 e 5.4. Nos gráficos (Figuras 5.10 a 5.18) , separamos os resultados por tipo de bloco básico para verificar o comportamento dos algoritmos em um cada nicho de atuação.

programa	progdin	cfc	cfc + tep	snake	profundidade	largura	1ep
acíclico	103	106	106	169	100	250	258
acíclico_paralelo	117	129	129	181	116	258	1035
acíclico_serial	355	397	397	573	364	787	1029
acíclico_serpar	180	208	208	310	201	445	1031
ciclo	133	99	99	152	59	59	100
ciclo_paralelo	162	107	107	168	162	162	403
ciclo_serial	475	403	403	590	381	510	793
ciclo_serpar	297	233	233	295	299	299	531
ciclo_aninhado	166	222	161	159	91	227	232
ciclo_aninhado_paralelo	178	235	174	212	193	263	931
ciclo_aninhado_serial	718	1005	650	763	581	1214	2149
ciclo_aninhado_serpar	355	492	333	410	329	532	1335
misto	209	228	167	192	194	265	594

Tabela 5.2: Tempos de execução dos programas (em ciclos), utilizando a latência de comunicação $L = 5$ ciclos.

programa	progdin	cfc	cfc + tep	snake	profundidade	largura	1ep
acíclico	151	124	124	314	164	494	258
acíclico_paralelo	166	140	140	337	193	506	1035
acíclico_serial	553	457	457	1038	589	1515	1029
acíclico_serpar	296	247	247	560	335	870	1031
ciclo	229	99	99	257	64	64	100
ciclo_paralelo	256	112	112	278	269	269	403
ciclo_serial	828	403	403	925	598	872	793
ciclo_serpar	552	238	238	510	519	519	531
ciclo_aninhado	300	229	223	304	116	405	232
ciclo_aninhado_paralelo	313	250	245	370	330	470	931
ciclo_aninhado_serial	1291	1012	871	1146	847	1869	2149
ciclo_aninhado_serpar	619	509	455	707	495	956	1335
misto	306	240	233	365	324	523	594

Tabela 5.3: Tempos de execução dos programas (em ciclos), utilizando a latência de comunicação $L = 10$ ciclos.

programa	progdin	cfc	cfc + tep	snake	profundidade	largura	1ep
acíclico	155	144	144	430	216	690	258
acíclico_paralelo	174	164	164	460	256	706	1035
acíclico_serial	557	501	501	1410	773	2099	1029
acíclico_serpar	304	278	278	760	443	1210	1031
ciclo	313	99	99	341	68	68	100
ciclo_paralelo	338	116	116	366	359	359	403
ciclo_serial	1120	403	403	1191	777	1167	793
ciclo_serpar	758	242	242	682	695	695	531
ciclo_aninhado	408	230	230	420	136	563	232
ciclo_aninhado_paralelo	424	258	258	498	442	641	931
ciclo_aninhado_serial	1763	1012	1012	1484	1081	2514	2149
ciclo_aninhado_serpar	839	517	517	950	634	1296	1335
misto	417	245	239	505	430	730	594

Tabela 5.4: Tempos de execução dos programas (em ciclos), utilizando a latência de comunicação $L = 15$ ciclos.

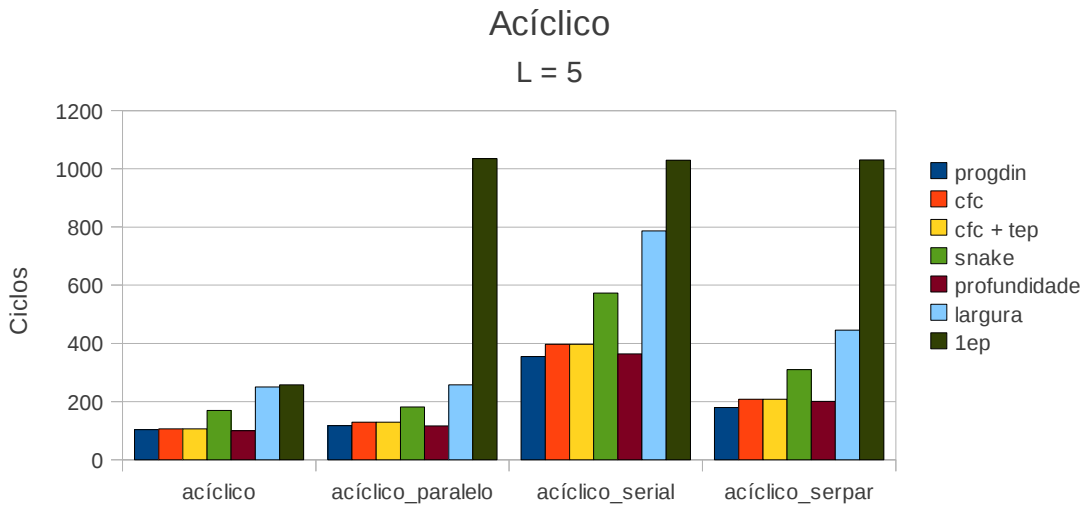


Figura 5.10: Tempo de execução dos programas acíclicos com latência de comunicação de 5 ciclos.

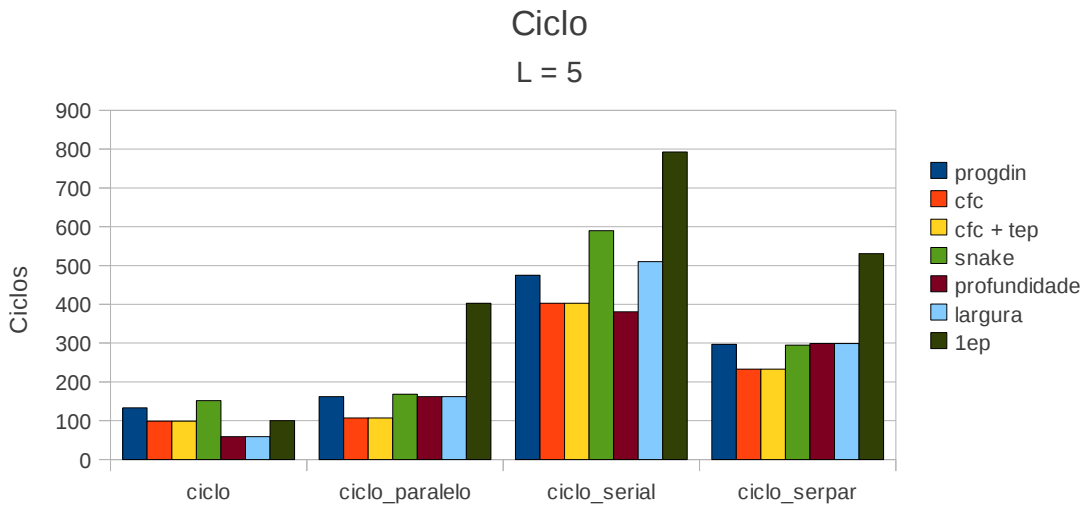


Figura 5.11: Tempo de execução dos programas com ciclo com latência de comunicação de 5 ciclos.

5.5 Efeitos da Fila de Operandos

Um problema encontrado durante os experimentos, foi a descoberta de uma variável que não foi considerada em nosso modelo e consequentemente não foi prevista no Algoritmo proposto. Isto impactou negativamente em alguns resultados e explica por que algumas vezes o algoritmo proposto não é bem sucedido.

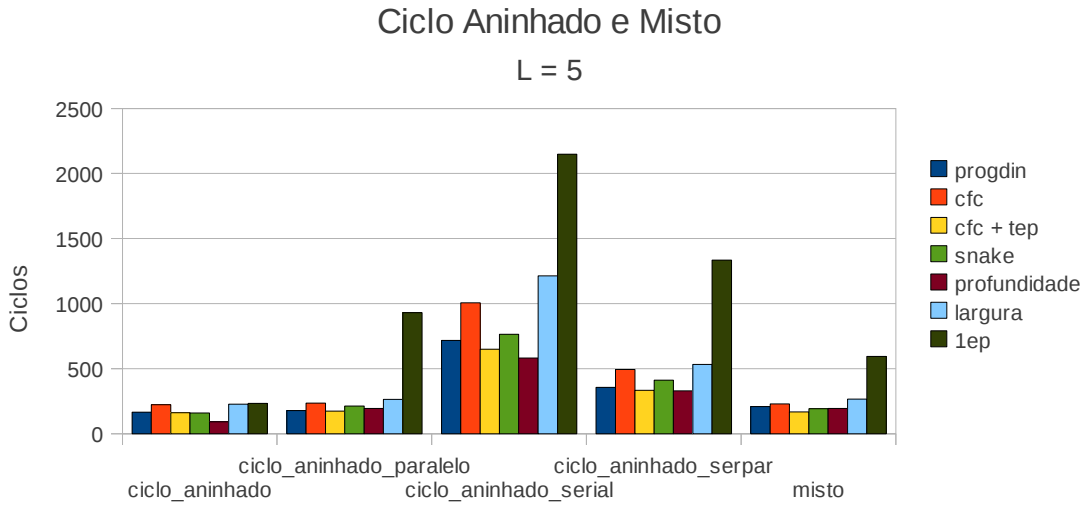


Figura 5.12: Tempo de execução dos programas com ciclo aninhado e o programa misto, com latência de comunicação de 5 ciclos.

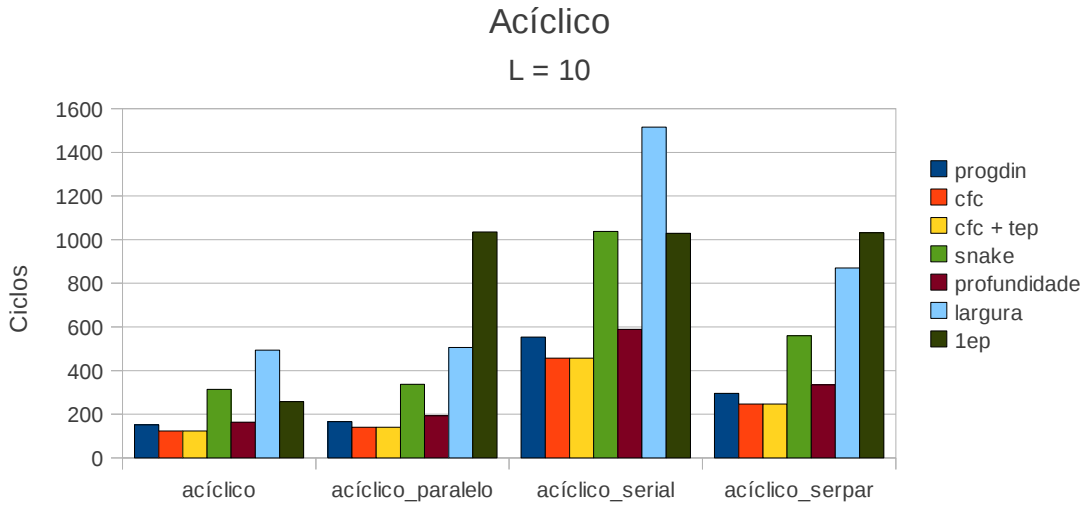


Figura 5.13: Tempo de execução dos programas acíclicos com latência de comunicação de 10 ciclos.

Durante os experimentos, pudemos gerar o trace de mapeamento do algoritmo proposto, isto é, a previsão de onde cada instrução mapeada seria executada e em que tempo seria executada. E através do Simulador, também pudemos gerar o trace de execução, ou seja, onde e quando cada instrução foi executada de fato. Ao comparar os traces de mapeamento e de execução percebemos algumas diferenças.

A Figura 5.19 ilustra o trace de mapeamento e a Figura 5.20 ilustra o trace de execução. Este teste foi realizado com o programa `acíclico` e latência L=5. No

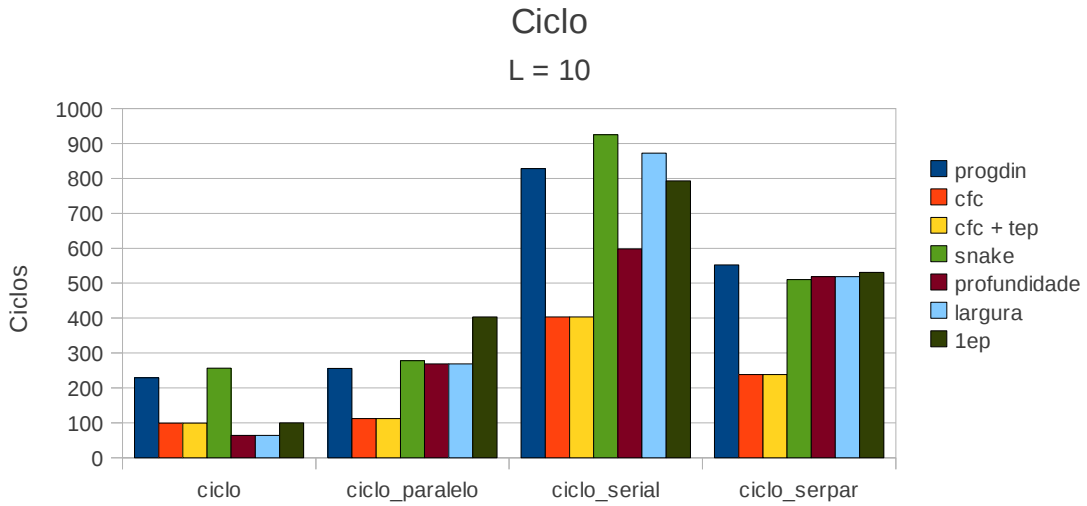


Figura 5.14: Tempo de execução dos programas com ciclo com latência de comunicação de 10 ciclos.

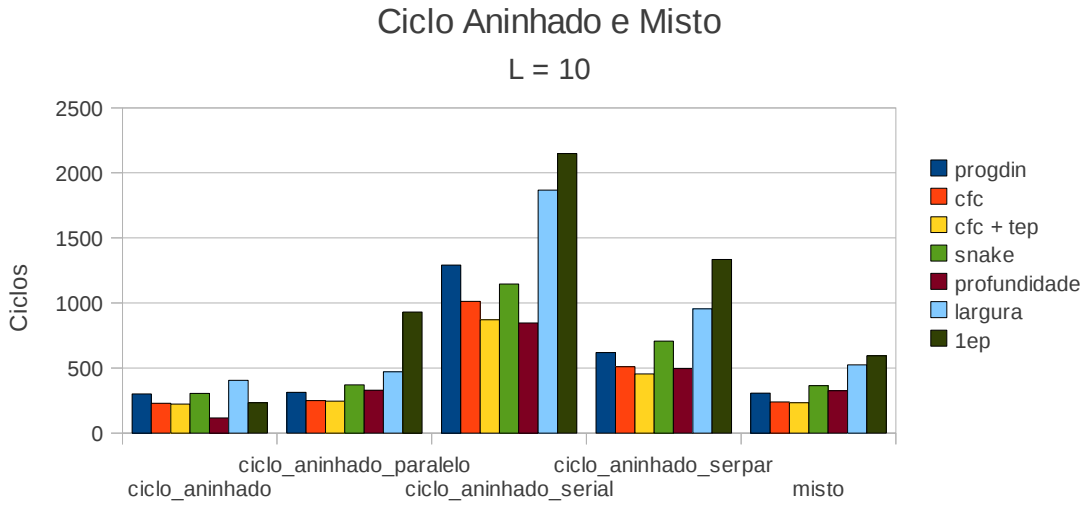


Figura 5.15: Tempo de execução dos programas com ciclo aninhado e o programa misto, com latência de comunicação de 10 ciclos.

trace, cada coluna é referente a um EP e cada linha ao ciclo de máquina em que a instrução deveria executar (no caso do trace de mapeamento) ou executou (no caso do trace de execução). Observe as instruções 12 e 13. Durante o mapeamento foi considerado que elas deveriam executar consecutivamente, nos ciclos 6 e 7 respectivamente. Mas durante a execução a instrução 12 executa no ciclo 6 e a instrução só executa no ciclo 27. Por que isso ocorreu?

Para descobrir, tivemos que recorrer ao trace mais detalhado do Simulador. Neste

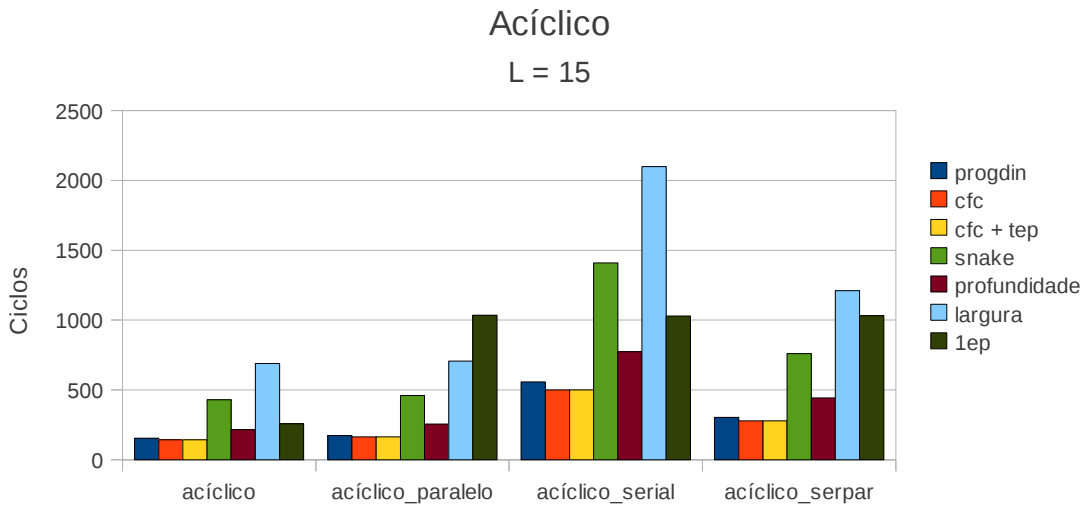


Figura 5.16: Tempo de execução dos programas acíclicos com latência de comunicação de 15 ciclos.

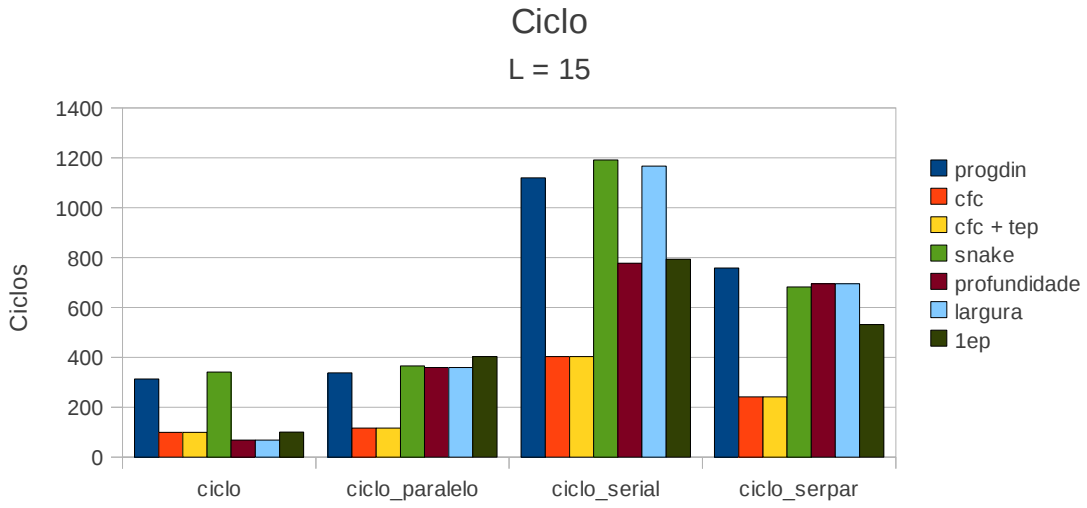


Figura 5.17: Tempo de execução dos programas com ciclo com latência de comunicação de 15 ciclos.

trace, é possível inspecionar cada EP em cada ciclo de execução. No ciclo 7, em que a instrução 13 deveria executar no EP#0 (ver Figura 5.21), descobrimos que o *Buffer* de Entrada esta repleto de mensagens (operandos). Entre elas, os dois operandos que a instrução 13 necessita para executar (marcados em cinza). Ou seja, a instrução 13 já poderia executar, pois o EP já possui os operandos necessários, mas ainda não executou pois cada operando do *Buffer* de Entrada precisa ser processado. Lembrando que no Simulador, um operando do *Buffer* de Entrada é removido a cada

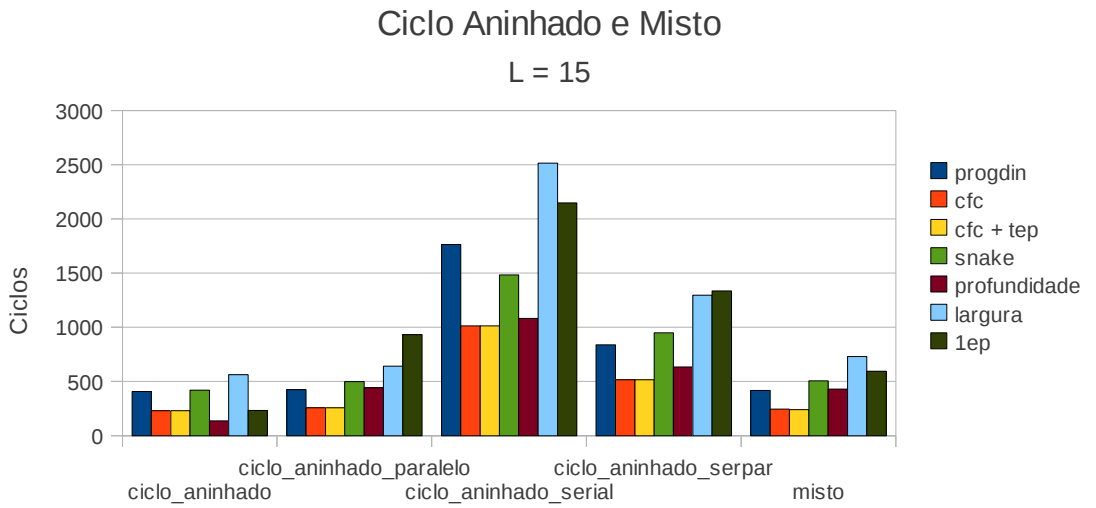


Figura 5.18: Tempo de execução dos programas com ciclo aninhado e o programa misto, com latência de comunicação de 15 ciclos.

```
001:000|003|002|001|004|005|011|009|007|006|008|059|___|___|___|___|___|___|096|___|___|___|010|
002:___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
003:___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
004:___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
005:___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
006:012|021|017|048|041|045|027|030|056|060|036|064|067|071|073|078|082|085|088|105|111|117|129|___|
007:013|022|___|049|042|046|___|031|___|___|037|___|___|___|___|079|___|___|___|___|___|118|___|___|
008:014|023|___|050|043|___|___|032|___|___|___|___|___|___|___|080|___|___|___|___|___|___|___|___|
009:015|024|___|051|044|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
010:016|___|___|052|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
011:018|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
012:019|___|___|___|047|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
013:020|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
014:025|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
015:026|___|___|___|053|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
016:028|___|___|___|054|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
017:029|___|___|___|055|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
018:033|___|___|___|057|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
019:034|___|___|___|058|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
020:035|___|___|___|061|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
021:038|___|___|___|062|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
022:039|___|___|___|063|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
023:040|___|___|___|065|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
024:068|___|___|___|066|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
025:___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
026:___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
027:___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
028:___|086|089|101|106|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
029:069|___|___|102|107|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|
030:070|   |   |   |108|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
```

Figura 5.19: Exemplo de trace de mapeamento do programa `acíclico`.

ciclo. Até o segundo operando de 13 ser processado (M13(0)), serão necessários 21 ciclos, pois existem 21 operandos no Buffer. São esses exatos 21 ciclos que separam o ciclo 6 do ciclo 27 que quando a instrução 13 é executada de fato (ver 5.20).

Portanto, a variável oculta no modelo e não prevista no Algoritmo era o número

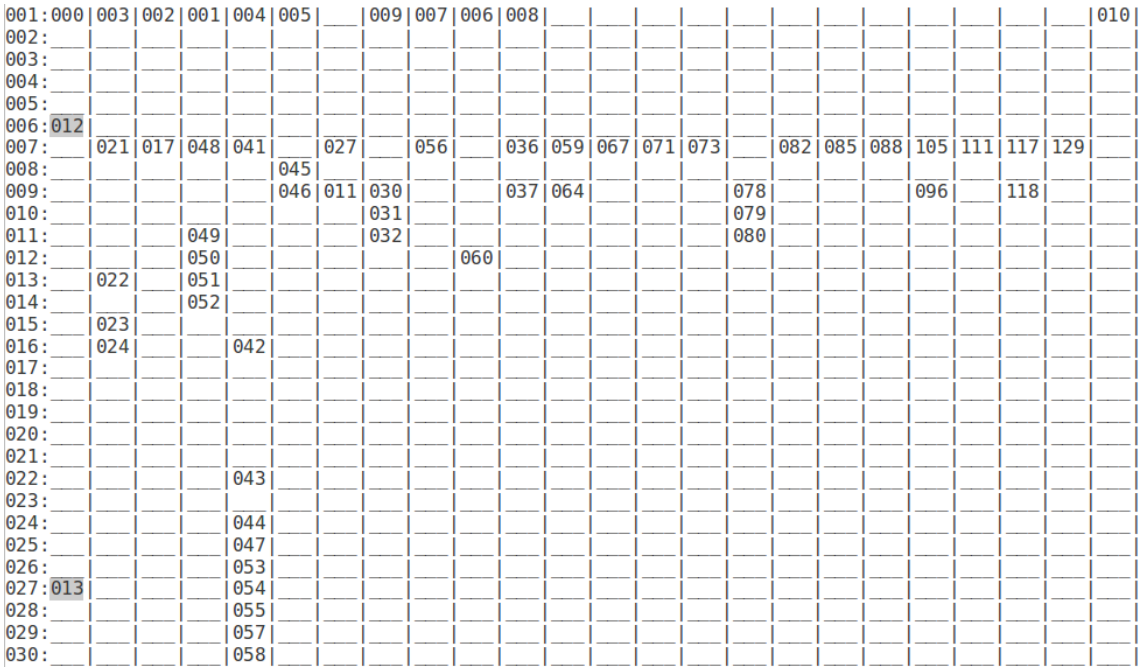


Figura 5.20: Exemplo de trace execução do programa `acíclico`.

```
7
EP #0
Buffer de Entrada:[M39(1), M70(1), M75(1), M13(1), M93(1), M100(1), M104(1),
M14(1), M99(1), M76(1), M84(1), M92(1), M121(1), M40(1), M20(1), M26(1),
M34(1), M19(1), M29(1), M35(1), M13(0)]
Matching Table: [M15(1), M94(1)]
Última Instrução:
```

Figura 5.21: Trace do EP0 duranto o ciclo 7 do programa `acíclico`.

de mensagens no *Buffer* de Entrada. Calcular este número de mensagens e como esta variável interage com o algoritmo não é trivial. Chegamos a fazer uma tentativa para incluir uma variável que desestimulasse a alocação de muitas instruções a um mesmo processador, o que diminuiria a fila de operandos em um EP. No entanto, os resultados foram inconclusivos e preferimos deixá-la de fora deste estudo.

Capítulo 6

Conclusões e Trabalhos Futuros

Neste trabalho implementamos o Simulador Arquitetural *Dataflow*, propomos um algoritmo de mapeamento e apresentamos um estudo comparativo entre algoritmos de mapeamento, utilizando um conjunto de programas de *benchmarking* que também implementamos. Neste capítulo, avaliaremos estas contribuições e faremos sugestões de trabalhos futuros.

O Simulador Arquitetural em nível de ciclo se mostrou adequado ao seu propósito, que era servir como plataforma de experimentos para o estudo do mapeamentos. Através do simulador, conseguimos extrair as medidas de interesse que guiaram o estudo do mapeamento, em especial o tempo de execução dos programas. Os traces de execução foram bastante úteis e através deles que inclusive, foi possível perceber uma nova variável para o problema, o número de operandos no *Buffer de Entrada* de um EP.

O algoritmo proposto se mostrou flexível o suficiente para lidar com grafos acíclicos e com ciclos. No caso dos grafos acíclicos (ver Figuras 5.10, 5.13 e 5.16) o algoritmo se mostrou equivalente aos outros com os quais foi comparado. Note que os algoritmos `progdin`, `cfc` e `cfc + tep` deveriam apresentar os mesmos resultados nestes grafos, visto que as componentes fortemente conexas sãos as próprias instruções. A diferença no resultado é explicada pela ordem da geração das componentes fortemente conexas ser diferente da ordem das instruções do grafo original. Isto altera a ordem em que as instruções são mapeadas pelo algoritmo e consequentemente altera o mapeamento e o tempo de execução final.

Já no caso dos grafos com ciclos simples (ver Figuras 5.11, 5.14 e 5.17), os algoritmos propostos (`cfc` e `cfc + tep`) obtiveram os melhores resultados. Isto fortaleceu a hipótese de lidar com os ciclos através do agrupamento das instruções de uma mesma CFC. A comparação com o algoritmo `progdin`, que também é baseado na programação dinâmica para o cálculo dos makespans mas não agrupa as instruções das CFCs, evidencia isto. A única exceção, na qual os algoritmos `largura` e `profundidade` obtiveram melhores resultados, foi no programa `ciclo`, em que o bloco

básico é executado sozinho mas é muito pequeno (apenas 10 instruções). Contudo, levando em consideração que um programa típico contém ciclos, é interessante que o algoritmo proposto tenha os melhores resultados nesses casos.

No caso dos ciclos aninhados (ver Figuras 5.12, 5.15 e 5.18), obtivemos resultados equivalentes ou piores que os outros algoritmos. Creditamos isso à presença de CFCs grandes (15 instruções) comparadas às outras CFCs do programa (ver Tabela 5.1), visto que o ciclo aninhado se torna uma CFC grande. Atribuir muitas instruções a um mesmo EP pode limitar o paralelismo.

Ainda analisando os resultados dos programas com ciclos aninhados, pudemos notar que para L=5 e L=10, houve uma diferença significativa entre `cfc` e `cfc + tep`. Como no caso, temos a presença da CFC grande (15 instruções), o algoritmo pode se beneficiar da utilização dos Tempos de Execução Personalizados. Assim as instruções sucessoras não necessitaram considerar o tempo total de execução (15 ciclos, considerando o $TE_i = 1$ de cada instrução), podendo considerar o Tempo de Execução Personalizado calculado através do caminho crítico entre a CFC antecessora e sua sucessora.

Sobre os algoritmos encontrados em [7] obtiveram resultados melhores que o esperado, considerando sua baixa complexidade. Especialmente o algoritmo `profundidade`. Aliás, utilizar a busca em profundidade para gerar uma ordem se mostrou mais efetiva que a busca em largura, já que a profundidade tende a agrupar em um EP instruções que são imediatamente dependentes. Por outro lado, o conjunto de programas do *benchmarking* foi bastante regular e bem comportado, isto pode ter beneficiado os algoritmos mais simples durante a divisão de instruções nos EPs.

De uma forma geral, o algoritmo proposto obteve melhores resultados com o aumento de L. Isto é interessante, já que algoritmo seria adequado em em situações onde o custo da comunicação é alto, como em ambientes de computação de memória distribuída . Note que variamos L até 15 ciclos, pois com esta latência o algoritmo `1ep` já começava a se tornar uma boa alternativa, o que indicava que o custo de comunicação era proibitivo em relação ao paralelismo do programa.

Como trabalhos futuros, indicamos o estudo e a possível inclusão da variável do número de operandos no *Buffer* de entrada do EP, no algoritmo de mapeamento. Isto tornaria o algoritmo de mapeamento mais próximo do real, e poderíamos comparar novamente os traces de mapeamento e execução para verificar se houve um ajuste entre o previsto e o executado. Isto também ajudaria na descoberta de outras variáveis que possam estar ocultas no modelo.

Também sugerimos a implementação do Algoritmo de mapeamento proposto na máquina virtual *Dataflow* Trebuchet. Seria interessante estudar os efeitos do mapeamento em um ambiente com mais variáveis envolvidas (cache, sistema operacional, etc) e com o objetivo de extrair performance em aplicações mais complexas. Além

disso, o algoritmo de mapeamento completa o processo de automatização da compilação *Dataflow* e auxiliaria a obter melhores resultados frente a outras ferramentas de paralelização como OpenMP [22], MPI [23], Cilk [24] e Intel Building Blocks [25].

Um estudo para lidar com Componentes Fortemente Conexas grandes, como no exemplo do ciclo aninhado, também poderia ser realizado. A ideia seria descobrir em quais casos é interessante particionar uma CFC para obter melhor desempenho. Uma heurística possível é a divisão da CFC em suas articulações.

A implementação da Trebuchet em *clusters* de computadores também se torna mais atraente, uma vez que os efeitos do mapeamento de instruções foram investigados. Algumas modificações no Algoritmo devem ser feitas para comportar a hierarquia de comunicação em um ambiente de memória distribuída, mas sem perda de generalidade.

Além disso, outro trabalho interessante seria implementar instruções em Arquiteturas heterogêneas (ex: CUDA, FPGA) na Trebuchet , pois o Algoritmo proposto foi concebido justamente para comportar instruções de diferentes tempos de execução.

Outra possibilidade de estudo que não foi explorada aqui, é a inclusão de probabilidades de desvio no problema do mapeamento. As instruções condicionais determinam o fluxo de execução no grafo *Dataflow*. E o fluxo de execução impacta diretamente no mapeamento, já que determinados caminhos serão mais executados que outros. Então pode-se adotar uma estratégia probabilística para condicionais, baseada na frequência em que os desvios são tomados e utilizar a esperança para o cálculo de *makespans*.

Por fim, sugerimos um estudo de performance ao utilizar o escalonamento estático simultaneamente com o escalonamento dinâmico (roubo de tarefas). Com isso poderíamos analisar quais combinações de variáveis possuem maior sinergia e qual seria a melhor escolha para uma melhor performance global.

Referências Bibliográficas

[1] DENNIS, J. B., MISUNAS, D. P. "A preliminary architecture for a basic data-flow processor", *SIGARCH Comput. Archit. News*, v. 3, n. 4, pp. 126–132, 1974. ISSN: 0163-5964. doi: http://doi.acm.org/10.1145/641675.642111.

[2] SWANSON, S. *The WaveScalar Architecture*. Tese de Doutorado, University of Washington, 2006.

[3] ASANOVIC, K., BODIK, R., DEMMEL, J., et al. "A view of the parallel computing landscape", *Commun. ACM*, v. 52, n. 10, pp. 56–67, 2009. ISSN: 0001-0782. doi: http://doi.acm.org/10.1145/1562764.1562783.

[4] JIAN, L., MARTINEZ, J. F. "Power-Performance Implications of Thread-level Parallelism on Chip Multiprocessors". In: *ISPASS '05: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005*, pp. 124–134, Washington, DC, USA, 2005. IEEE Computer Society. ISBN: 0-7803-8965-4. doi: http://dx.doi.org/10.1109/ISPASS.2005.1430567.

[5] BARSZCZ, E., HOWARD, L. S., CENTER., A. R. *A static data flow simulation study at Ames Research Center [microform] / Eric Barszcz, Lauri S. Howard*. National Aeronautics and Space Administration, Ames Research Center ; For sale by the National Technical Information Service, Moffett Field, Calif. : [Springfield, Va. :, 1987.

[6] BOYER, W. F., HURA, G. S. "Non-evolutionary algorithm for scheduling dependent tasks in distributed heterogeneous computing environments", *Journal of Parallel and Distributed Computing*, v. 65, n. 9, pp. 1035–1046, set. 2005. ISSN: 07437315. doi: 10.1016/j.jpdc.2005.04.017. Disponível em: <`http://linkinghub.elsevier.com/retrieve/pii/S0743731505000900`>.

[7] MERCALDI, M., SWANSON, S., PETERSEN, A., et al. "Modeling instruction placement on a spatial architecture". In: *SPAA '06: Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and*

architectures, pp. 158–169, New York, NY, USA, 2006. ACM. ISBN: 1-59593-452-9. doi: http://doi.acm.org/10.1145/1148109.1148137.

[8] ALVES, T. A. *Execução Especulativa em uma Máquina Virtual Dataflow*. Tese de Mestrado, COPPE - UFRJ, may 2010.

[9] SMITH, J., NAIR, R. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 2005. ISBN: 1558609105.

[10] MARZULO, L. A. J. *Transactional WaveCache - Execução Especulativa Fora-de-Ordem de Operações de Memória em uma Máquina Dataflow*. Tese de Mestrado, COPPE - UFRJ, dez. 2007.

[11] MARZULO, L. A. J. *Explorando Linhas de Execução Paralelas com Programação Orientada por Fluxo de Dados*. Tese de Doutorado, COPPE - UFRJ, out. 2011.

[12] ALVES, T. A., MARZULO, L. A., FRANCA, F. M., et al. "Trebuchet: exploring TLP with dataflow virtualisation", *International Journal of High Performance Systems Architecture*, v. 3, n. 2/3, pp. 137, 2011. ISSN: 1751-6528. doi: 10.1504/IJHPSA.2011.040466.

[13] ALVES, T. A., MARZULO, L. A. J., FRANÇA, F. M. G., et al. "Trebuchet: Explorando TLP com Virtualização DataFlow". In: *WSCAD-SSC'09*, pp. 60–67, São Paulo, out. 2009. SBC.

[14] MARZULO, L. A., FRANCA, F. M., COSTA, V. S. "Transactional WaveCache: Towards Speculative and Out-of-Order DataFlow Execution of Memory Operations", *Computer Architecture and High Performance Computing, Symposium on*, v. 0, pp. 183–190, 2008. ISSN: 1550-6533. doi: http://doi.ieeecomputersociety.org/10.1109/SBAC-PAD.2008.29.

[15] MARZULO, L. A. J., ALVES, T. A., FRANCA, F. M. G., et al. "TALM: A Hybrid Execution Model with Distributed Speculation Support", *Computer Architecture and High Performance Computing Workshops, International Symposium on*, v. 0, pp. 31–36, 2010. doi: http://doi.ieeecomputersociety.org/10.1109/SBAC-PADW.2010.8.

[16] MARZULO, L. A. J., FLESCH, F., NERY, A., et al. "FlowPGA: DataFlow de Aplicações em FPGA", *IX Simpósio em Sistemas Computacionais WSCAD - SSC*, 2008.

[17] "Graphviz web-site. http://www.graphviz.org". .

[18] "Javacc^{TM} – The Java Parser Generator http://javacc.java.net". .

[19] FRANÇA, F. M. G., ASSUMP, J., FILHO, M., et al. "Uma Proposta de um Escalonador para Gamma", 2001. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.10.7627>.

[20] PEREIRA, M. R., VARGAS, P. K., FRANÇA, F. M. G., et al. "Applying Scheduling by Edge Reversal to Constraint Partitioning". In: *Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing*, p. 134. IEEE Computer Society, 2003. ISBN: 0-7695-2046-4. Disponível em: <http://portal.acm.org/citation.cfm?id=952456>.

[21] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., et al. *Introduction to Algorithms, Third Edition*. The MIT Press, 2009. ISBN: 0262033844, 9780262033848.

[22] DAGUM, L., MENON, R. "OpenMP: an industry standard API for shared-memory programming", *IEEE Computational Science and Engineering*, v. 5, n. 1, pp. 46–55, 1998. ISSN: 10709924. doi: 10.1109/99.660313. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=660313>.

[23] THE MPI FORUM. "MPI: A message passing interface". In: *Proc. Supercomputing '93*, pp. 878–883, nov. 15–19, 1993. doi: 10.1109/SUPERC.1993.1263546.

[24] JOERG, C. F. *The Cilk System for Parallel Multithreaded Computing*. Tese de Doutorado, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, jan. 1996. Available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-701.

[25] REINDERS, J. *Intel threading building blocks : outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007. ISBN: 0596514808.

Apêndice A

Aplicações

Neste apêndice é exibido o código C de programas que foram utilizadas nos experimentos descritos no Capítulo 5 .

```
//aciclico.c

int main () {

int v1a;
int v2a;
int v3a;
int v4a;
int v5a;
int v6a;
int v7a;
int v8a;
int v9a;
int v10a;
int v11a;

int saida;

v1a= 1;
v2a= 2;
v3a= 3;
v4a= 4;
v5a= 5;
v6a= 6;
v7a= 7;
```

```
v8a= 6;
v9a= 7;
v10a= 10;
v11a = 0;

v8a = v8a + v8a + v1a * v4a * v3a * v2a * v1a + v2a * v3a + v7a + v8a + v3a
 * v1a * v8a * v2a * v3a + v8a + v2a * v10a + v9a + v8a * v3a * v1a * v1a
 + v8a + v9a + v8a * v5a * v7a + v4a + v10a;

v9a = v4a * v5a * v5a * v10a * v7a + v5a * v10a * v2a + v6a * v2a * v10a
 * v2a * v9a * v10a + v5a + v10a + v5a * v9a + v1a + v7a * v7a * v4a + v3a
 + v2a + v7a * v9a + v9a + v1a * v4a * v8a + v4a;

v5a = v9a + v1a * v5a + v5a * v1a + v4a + v9a * v6a + v10a * v2a * v3a * v1a
 + v4a * v7a + v6a + v4a * v1a * v8a + v7a * v1a * v8a * v9a + v6a + v3a
 + v1a + v8a + v10a * v10a + v8a + v2a + v3a;

v4a = v8a * v2a * v4a * v5a * v3a + v4a * v6a * v8a * v4a * v7a + v8a + v6a
 * v10a * v9a * v1a * v10a * v2a + v3a * v1a * v2a + v5a + v6a + v5a * v2a
 + v5a * v10a * v10a * v5a * v5a + v7a * v2a;

v11a = v8a + v9a + v5a + v4a;

saida = v11a + 0;

return(0);
}
```

```
//ciclo.c

int main (void) {

int a, i, saida;

a = 0;
i = 0;

while(i < 10) {

a = a + 5;
i = i + 1;

}

saida = a + 0;

return (0);
}
```

```
//cicloaninhado.c

int main () {

int v1a, v2a, v3a, zeroa, saida;
v1a = 0;
v2a = 0;
v3a = 0;
zeroa = 0;

while(v1a < 2)
{
v1a = v1a + 1;
v2a = zeroa;

while (v2a < 5) {
v2a = v2a + 1;
v3a = v3a + 1;
v1a = v1a;
zeroa = zeroa;
}
}

saida = v3a + 0;

return(0);

}
```

```
//misto.c
int main () {

int v1a, v2a, v3a, v4a, v5a, v6a, v7a, v8a, v9a, v10a, v11a;

int a, i, saida, total;

int v1b, v2b, v3b, zerob;

v1a= 1;
v2a= 2;
v3a= 3;
v4a= 4;
v5a= 5;
v6a= 6;
v7a= 7;
v8a= 6;
v9a= 7;
v10a= 10;
v11a = 0;

a = 0;
i = 0;

v1b = 0;
v2b = 0;
v3b = 0;
zerob = 0;


v8a = v8a + v8a + v1a * v4a * v3a * v2a * v1a + v2a * v3a + v7a + v8a
 + v3a * v1a * v8a * v2a * v3a + v8a + v2a * v10a + v9a + v8a * v3a * v1a
* v1a + v8a + v9a + v8a * v5a * v7a + v4a + v10a;

v9a = v4a * v5a * v5a * v10a * v7a + v5a * v10a * v2a + v6a * v2a
 * v10a * v2a * v9a * v10a + v5a + v10a + v5a * v9a + v1a + v7a * v7a * v4a
 + v3a + v2a + v7a * v9a + v9a + v1a * v4a * v8a + v4a;

v5a = v9a + v1a * v5a + v5a * v1a + v4a + v9a * v6a + v10a * v2a
```

```
 * v3a * v1a + v4a * v7a + v6a + v4a * v1a * v8a + v7a * v1a * v8a * v9a
 + v6a + v3a + v1a + v8a + v10a * v10a + v8a + v2a + v3a;

v4a = v8a * v2a * v4a * v5a * v3a + v4a * v6a * v8a * v4a * v7a +
 v8a + v6a * v10a * v9a * v1a * v10a * v2a + v3a * v1a * v2a + v5a + v6a
 + v5a * v2a + v5a * v10a * v10a * v5a * v5a + v7a * v2a;

v11a = v8a + v9a + v5a + v4a;


while(i < 10) {

a = a + 5;
i = i + 1;

}


while(v1b < 2)
{
v1b = v1b + 1;
v2b = zerob;

while (v2b < 5) {
v2b = v2b + 1;
v3b = v3b + 1;
v1b = v1b;
zerob = zerob;
}
}

a = a + 255;// Zera Wave

v3b = v3b + 255;// Zera Wave

total = v11a + a + v3b;

total = total + 0; //Out
```

```
return(0);
}
```