LUCAS MUCIDA COSTA

UMA HEURÍSTICA GULOSA PARA *MODULO SCHEDULING* EM ARQUITETURAS RECONFIGURÁVEIS EM TEMPO DE EXECUÇÃO

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

VIÇOSA
MINAS GERAIS - BRASIL
2013

Ficha catalográfica preparada pela Seção de Catalogação e Classificação da Biblioteca Central da UFV

T
C837h
2013

Costa, Lucas Mucida, 1987-
Uma heurística gulosa para *Modulo Scheduling* em arquiteturas reconfiguráveis em tempo de execução / Lucas Mucida Costa. – Viçosa, MG, 2013.
ix, 52 f. : il. (algumas color.) ; 29 cm.

Orientador: Ricardo dos Santos Ferreira.
Dissertação (mestrado) - Universidade Federal de Viçosa.
Referências bibliográficas: f. 48-52.

1. Arquitetura de computador. 2. Arquitetura de rede de computador. 3. Computadores - Projetos e construção. 4. Pipeline (Software). I. Universidade Federal de Viçosa. Departamento de Informática. Programa de Pós-Graduação em Ciência da Computação. II. Título.

CDD 22. ed. 004.22

LUCAS MUCIDA COSTA

UMA HEURÍSTICA GULOSA PARA MODULO SCHEDULING EM ARQUITETURAS RECONFIGURÁVEIS EM TEMPO DE EXECUÇÃO

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae.*

APROVADA: 23 de julho de 2013.

Carlos de Castro Goulart

José Augusto Miranda Nacif

Ricardo Dos Santos Ferreira
(Orientador)

Agradecimentos

Agradeço primeiramente a Deus por ter me dado esta oportunidade de fazer um mestrado. Gostaria de agradecer também aos meus pais e irmão, que sempre foram minha raiz, que não me deixam cair em momento algum e me dão forças para continuar de pé e enfrentar qualquer desafio. Em especial à minha mãe que com sua sabedoria de vida me guiava para os melhores caminhos, e ao meu pai que me ensinou a ser uma boa pessoa, que respeita ao próximo e assim conquista muitas amizades, que foram de extrema importância ao longo desse caminho.

Aos amigos da graduação, que tornaram esses anos na melhor fase da minha vida, me ajudaram em todos os momentos difíceis, em todas as provas e trabalhos, e me fizeram chegar até onde cheguei, num mestrado nesta grande instituição de ensino que é a UFV.

À toda a minha família que têm me dado forças para conquistar meus objetivos, especialmente à pessoa que acabou de entrar para ela.

Aos docentes do Departamento de Informática da Universidade Federal de Viçosa que compartilharam seus conhecimentos, num programa de graduação de altíssima qualidade, abrindo-me portas para um futuro profissional brilhante. Agradeço ao meu orientador Ricardo pela oportunidade de fazer parte deste grande trabalho de mestrado, ampliando meus conhecimentos na área e também ampliando as minhas oportunidades no futuro.

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

COSTA, Lucas Mucida, M.Sc., Universidade Federal de Viçosa, julho de 2013. **Uma heurística gulosa para *Modulo Scheduling* em arquiteturas reconfiguráveis em tempo de execução** Orientador: Ricardo Santos Ferreira

Atualmente, a quantidade de dados gerados pelas aplicações vem crescendo mais rapidamente do que a lei de Moore. Arquiteturas reconfiguráveis de grão grosso também chamadas de *CGRA* (*Coarse Grained Reconfigurable Architecture*), tem sido propostas para tentar tratar esse grande fluxo de dados, levando em consideração também a necessidade de eficiência em termos de potência, energia e desempenho. Um dos desafios é o mapeamento dos laços de computação intensiva nas CGRAs. Este trabalho propõem um algoritmo guloso, simples e rápido, que pode ser implementado em hardware e ser usado em tempo de execução. Enquanto soluções anteriores obtinham respostas em segundos, a abordagem proposta reduz o tempo de mapeamento para ordem de microssegundos. Portanto, a solução proposta é viável para mapear laços em tempo de execução.

Abstract

COSTA, Lucas Mucida, M.Sc., Universidade Federal de Viçosa, July 2013. **A runtime Modulo Scheduling greedy heuristic in reconfigurable architectures** Adviser: Ricardo Santos Ferreira.

Nowadays, the amount of data generated by applications has been growing faster than Moore's law. Coarse grain reconfigurable architectures also called CGRA has been proposed to handle this large data streams, considering power, energy and performance is needed. Mapping innerlopps in CGRAs is a challenge. This work proposes a greedy algorithm (simple and fast), which can be implemented in hardware and to be used at runtime. Comparing to previous solutions by three to five orders of magnitude, our approach reduces the mapping time. Thus, the proposed solution is viable to handle with the demand for data flow at runtime.

Capítulo 1

Introdução

Nos dias de hoje, a quantidade de dados gerados pelas aplicações vem crescendo mais rapidamente do que a lei de Moore [33]. Cerca de 90% dos dados existentes foram gerados nos últimos 2 anos. Todos os dias, aproximadamente 2.273.736,75 terabytes são gerados em todo o mundo [41]. Este grande volume de dados e as exigências eficiência e baixo consumo de energia são alguns dos grandes desafios da atualidade.

As tecnologias existentes ainda precisam evoluir para processar o grande volume de dados com uma a eficiência em termos de desempenho, potência e uso de memória [22, 25] e ter escalabilidade.

Buscando soluções para tratar o grande fluxo de dados [26] somados à necessidade de eficiência em termos de potência, energia, desempenho e a demanda por sistemas em tempo de execução, vários esforços tem sido tomados, entre eles podemos destacar as arquiteturas reconfiguráveis. Em especial, as arquiteturas reconfiguráveis de grão grosso, também chamadas de *CGRA* [21, 4, 40] (*Coarse Grained Reconfigurable Architecture*, tem sido propostas para alcançar a eficiência e aceleração de hardware necessárias [7, 27].

Grande parte CGRAs encontrados na literatura requerem ferramentas especiais ou compiladores para gerar a configuração necessária para executar os cálculos [6, 1], e gastam um tempo de mapeamento na ordem de segundos [29], o que inviabiliza a implementação em tempo de execução.

O objetivo deste trabalho é encontrar um algoritmo guloso em tempo de execução que seja simples e rápido, e que possa ser executado em uma arquitetura alvo também simples.

1.1 Motivação

Em [38], pode-se ver que os laços de algumas aplicações são responsáveis por até 71% do tempo gasto para executar o código. Se o tempo de execução do laço for otimizado, pode-se obter um ganho de mais de 3x no tempo total de execução da aplicação.

O ganho de desempenho pode ser obtido com uso de *software pipeline* em laços [2, 23]. Essa técnica permite a sobreposição de instruções de diferentes iterações.

Uma das formas de utilizar o *software pipeline* é através do *Modulo Scheduling* visto em [39]. Em [15], o uso da CGRA juntamente com uma heurística de mapeamento permitiu alcançar um tempo de compilação viável na ordem de milissegundos. Porém, a arquitetura usada em [15] é grande (256 UFs), possui uma ocupação baixa, pontos que podem ser melhorados. Outro fator a ser melhorado é o intervalo de iniciação obtido que não é mínimo em [15].

1.2 Objetivos

O objetivo deste trabalho é o desenvolvimento de um algoritmo guloso que seja viável para ser usado em tempo de execução. O algoritmo otimiza a execução de laços fazendo o mapeamento e escalonamento em uma arquitetura CGRA.

1.3 Organização do texto

Primeiramente, no capítulo 2, uma revisão bibliográfica é apresentada, abordando os CGRA's, a técnica de *Modulo Scheduling* utilizada e as partições temporais, juntamente com a apresentação dos trabalhos relacionados.

Em seguida, o capítulo 3 descreve o algoritmo proposto por esta dissertação, para dois tipos de redes de interconexão, multiestágio e *Crossbar*. Será mostrado como o algoritmo guloso percorre uma única vez o grafo executando o mapeamento em tempo de execução. A implementação foi realizada com uma máquina de estados que executa passo a passo a leitura, escalonamento, posicionamento e roteamento de cada aresta.

Em seguida, o capítulo 4 apresenta os resultados obtidos pelo algoritmo proposto, mostrando o tempo de execução, grau de paralelismo extraído do grafo, a ocupação e o intervalo de iniciação obtido para um conjunto de *benchmarks* de aplicações multimídia.

Finalmente, o capítulo 5 apresenta as conclusões e aborda sugestões de melhorias que podem ser feitas em trabalhos futuros.

Capítulo 2

Revisão Bibliográfica

Este capítulo apresenta os trabalhos relacionados ao tema de *Modulo Scheduling* em arquiteturas reconfiguráveis.

Primeiramente será explicado o conceito *Software Pipeline*. Em seguida, será dada uma explicação sobre a heurística de *Modulo Scheduling* para depois introduzirmos o conceito de *Modulo Scheduling* aplicado ao CGRA. Mais à frente, será abordada a heurística criada em [15], que é a base deste trabalho. Em seguida serão revistas as redes de interconexões abordadas em [15] e que serão utilizadas no algoritmo proposto nesta dissertação.

2.1 Software Pipeline

O *Software Pipeline* [28] é uma técnica que visa paralelizar a execução de laços. Esta técnica sobrepõe iterações de um laço, sem ter que esperar que as iterações anteriores terminem, executando várias iterações simultaneamente.

A Figura 2.1 (a) ilustra um exemplo retirado de [28] onde uma sequência de instruções é executada para adicionar uma constante a cada elemento de um vetor. Suponha que a leitura e escrita na memória gastam apenas 1 ciclo, já para adicionar a constante são necessários 2 ciclos. A Figura 2.1 (b) mostra diferentes iterações sendo executadas em paralelo com sobreposição, que é uma forma de *Software Pipeline*.

Otimizar um laço com *Software Pipeline* é um problema NP-Completo [28]. Uma alternativa para diminuir a complexidade do *Software Pipeline* é o uso de heurísticas, como o *Modulo Scheduling*.

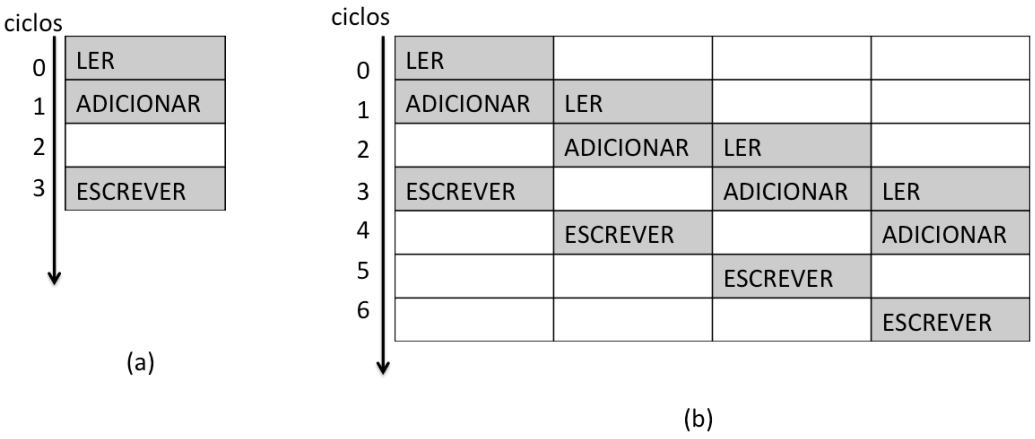

Figura 2.1. (a) Sequência de Instruções (b) Iterações Sobrepostas

2.2 Modulo Scheduling

A técnica de *Modulo Scheduling* (MS) [39, 32, 35, 23, 32] é um dos recursos utilizados para aumentar o paralelismo a nível de instrução (ILP) em laços de computação intensiva. O MS realiza a sobreposição de diferentes iterações do laço, de forma que as dependências internas e externas do grafo não sejam violadas, podendo assim começar uma nova iteração antes que a outra termine, extraindo o paralelismo presente no grafo. Além disso, o MS busca extrair o escalonamento ótimo para uma iteração do laço evitando violações nas restrições de dependência e de recursos.

Estas dependências são caracterizadas da seguinte forma: dependências externas são aquelas em que um ou mais nós de uma determinada iteração dependem do resultado da iteração anterior, não permitindo que estas duas sejam sobrepostas e executadas ao mesmo tempo. As dependências internas são aquelas em que os nós dentro da mesma iteração são dependentes entre si e a ordem deve ser respeitada.

As restrições de recursos que dizem respeito aos recursos da arquitetura; a quantidade de unidades funcionais e seu tipo; e a disponibilidade da rede de interconexão.

A Figura 2.2 ilustra o processamento dos dados de um laço com 4 operações. O mesmo laço pode ser executado em um tempo muito menor quando utilizada o MS onde as iterações são sobrepostas (2.2 (b) e (c)).

Na figura 2.3 (b) três operações são executadas em t2: O3 na iteração I1, O2 na iteração I2 e O3 na iteração. Para a figura 2.3 (c), em t2 são executadas duas operações.

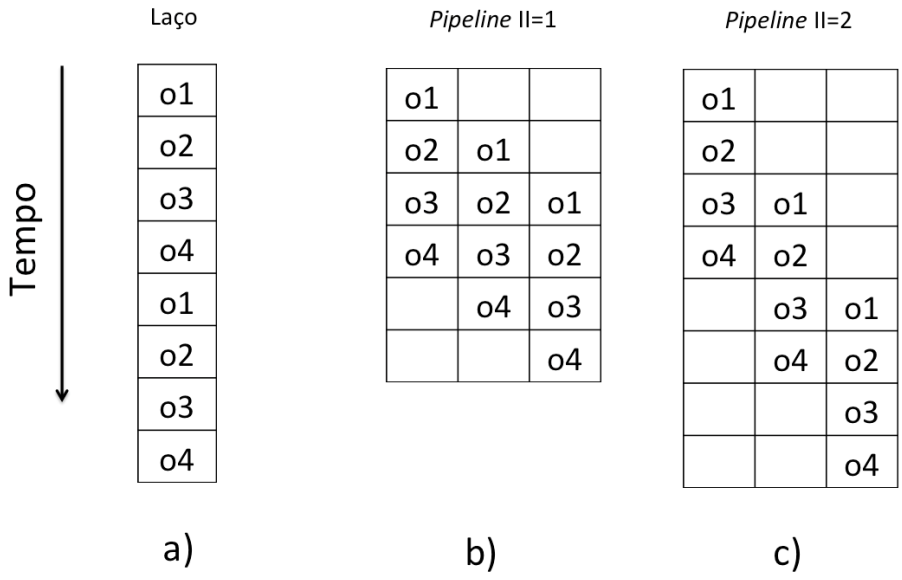

Figura 2.2. (a) processamento normal de dados, (b) processamento com *Modulo Scheduling* com II = 1, (c) processamento com *Modulo Scheduling* com II=2

O *Modulo Scheduling*, após preencher o pipeline, alcança o estado de equilíbrio com sobreposição de duas ou mais iterações gerando uma vazão constante.

O intervalo entre as iterações é chamado de intervalo de iniciação (II) e é determinado pelo algoritmo de acordo com as restrições de recursos da arquitetura e as dependências do grafo. O intervalo de iniciação mínimo inicial é chamado de MII. Contudo, não se pode garantir que o algoritmo irá conseguir sempre um II igual ao MII devido a restrições de recursos da arquitetura como unidades funcionais e recursos de interconexões.

Nos últimos anos, várias abordagens [38, 35, 29, 15, 3, 20], foram propostas para reduzir o tempo de compilação de algoritmos *Modulo Scheduling*, que é um dos grande desafios na sua implementação.

2.2.1 Configurações

O caso ideal do *Modulo Scheduling* é um intervalo de iniciação (II) igual a 1. Porém, devido a falta de recursos na arquitetura ou dependências entre as iterações, o II pode ser maior que 1.

Suponha um caso com restrições de recursos na arquitetura. A Figura 2.3 mostra um exemplo onde um grafo com sete nós é mapeado em uma arquitetura com apenas quatro unidades de execução, utilizando duas configurações. Em azul

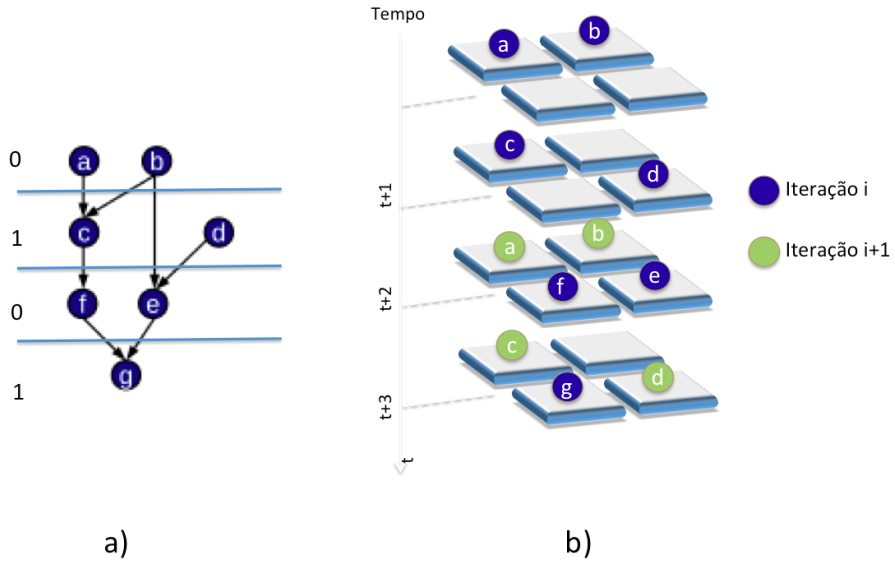

Figura 2.3. (a) grafo a ser mapeado, (b) mapeamento na arquitetura utilizando II=2

estão as operações da primeira iteração ocupando a arquitetura. Em verde estão as operações da segunda iteração que são parcialmente sobrepostas com o *Modulo Scheduling*.

A Figura 2.4 mostra o mesmo grafo da Figura 2.3 mapeado com II=3. Este cenário pode ocorrer caso não haja recursos de roteamento, por exemplo.

Neste caso, apenas em uma configuração, como ilustrado no tempo t+3, ocorre sobreposição das iterações

2.2.2 *Modulo Scheduling* para CGRA

Para utilizar o *Modulo Scheduling* em arquiteturas CGRA, é preciso contornar alguns novos desafios introduzidos em comparação ao uso do MS em arquiteturas VLIW (Very Long Instruction Word) [5] como pode ser visto em [38].

Enquanto em arquiteturas VLIW o roteamento é feito de maneira implícita pelo banco de registradores com acesso global, nos CGRAs é preciso ter recursos de roteamento entre as operações. Neste trabalho foram utilizados as redes de interconexão que serão apresentadas na seção 2.4.1. Além disso, UFs homogêneas foram utilizadas, onde todas elas podem realizar qualquer operações.

O MS em CGRAs utiliza os seguintes passos: Escalonamento, Posicionamento e Roteamento (EPR).

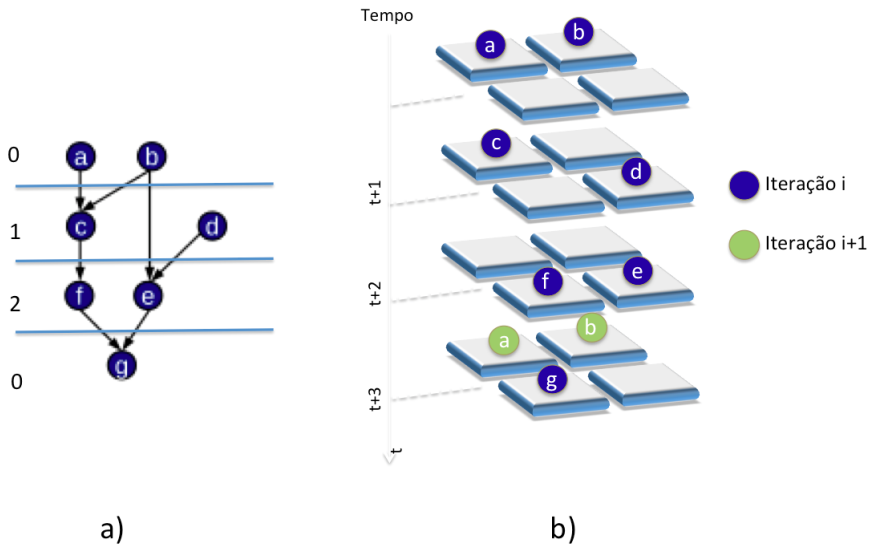

Figura 2.4. (a)grafo a ser mapeado, (b) mapeamento na arquitetura utilizando II=3

- Escalonamento: é responsável pela atribuição do tempo no qual as operações são executadas;
- Posicionamento: é responsável pela alocação das operações nas UFs disponíveis no tempo escalonado;
- Roteamento: é responsável por efetuar a conexão entre as operações que possuam dependência e que foram posicionadas nas UFs;

A Figura 2.5 mostra como fica o exemplo da Figura 2.3 utilizando a sobreposição de iterações. Note que há uma diminuição dos recursos ociosos quando o MS entra no estágio de equilíbrio. A ocupação é calculada pela divisão (operações/II*arquitetura) . Neste caso a ocupação é de 87,5% considerando 7 operações, II igual a 2 e uma arquitetura de tamanho 4.

2.3 Outros trabalhos

Um dos trabalhos pioneiros da área de *Modulo Scheduling* para CGRA é o compilador DRESC, elaborado por [31], que utiliza *Simulated Annealing* e reposiciona as operações nas UFs para encontrar um escalonamento e roteamento válido. Este

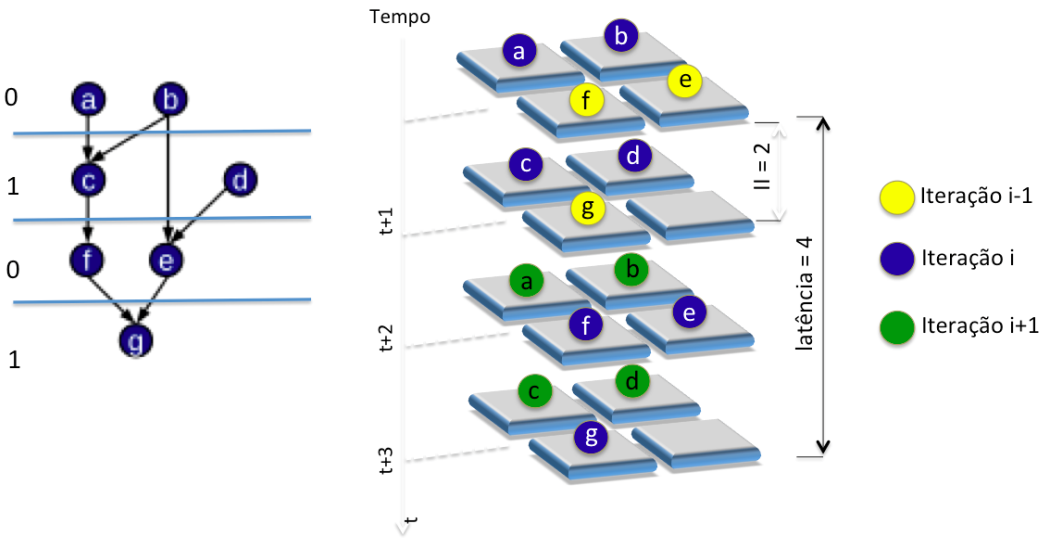

Figura 2.5. Sobreposição das Iterações

algoritmo depende de algumas funções que analisam se vale a pena ou não fazer esta troca, o que gera um tempo de compilação elevado [38].

Posteriormente, o *Edge Centric Modulo Scheduling* ou EMS [38] foi proposto. O EMS reduz o tempo de compilação, priorizando o roteamento das arestas, e a partir daí, posiciona os nós nas UFs. O EMS reduz tempo de mapeamento, porém apresenta intervalos de iniciação maiores que o DRESC.

Recentemente, um outro algoritmo foi proposto em [20], o EPIMAP. Este algoritmo propõem o uso da recomputação para resolver os problemas de posicionamento na arquitetura, onde um nó pode ser calculado mais de uma vez. Esta heurística apresenta resultados melhores, ótimos em muitos casos, em termos de II em relação ao EMS, porém o tempo de mapeamento piorou, chegando a ser 6 vezes maior do que o EMS.

Outra abordagem recente é o algoritmo *Graph Minor* [3] que minimiza o uso de registradores necessários para repassar o mesmo valor para diferentes nós. Consequentemente isto contribui para a melhoria do escalonamento e posicionamento do grafo. Este trabalho teve uma melhoria significativa em relação ao DRESC em termos de tempo de compilação e alocação de UFs, obtendo 62% de ocupação média contra 54% do DRESC.

Em [15], o uso da CGRA com uma rede global de interconexão multiestágio, permitiu alcançar um tempo de compilação viável na ordem de milissegundos. Porém, o escalonamento é feito separadamente, utilizando ASAP(*As Soon As Possible*)

e ALAP (*As Late As Possible*) ao longo do grafo para balanceá-lo antes de do algoritmo executar o mapeamento. Além disto, nessa abordagem, o posicionamento não é guloso, pois se ocorrer falha no roteamento, o algoritmo tenta reposicionar o nó em outra unidade e posteriormente rotear, aumentando o tempo de mapeamento.

O grupo de pesquisa do Departamento de Informática da UFV vem desenvolvendo trabalhos com arquiteturas reconfiguráveis envolvendo os problemas de: posicionamento [10, 11, 16, 18], roteamento de interconexões [42, 14, 13, 12], tolerância a falhas [9], escalonamento [15], tradução binária [13, 12, 9]. Este trabalho faz parte da linha de pesquisa em escalonamento, posicionamento e roteamento de laços nas arquiteturas em tempo de execução [19, 34, 17].

Esta dissertação busca reduzir o tempo de mapeamento sem perder qualidade no escalonamento. Este trabalho faz parte de um ambiente de aceleração de laços com computação intensiva implementados em FPGA. A Figura 2.6 mostra um diagrama do ambiente de mapeamento em desenvolvimento no Departamento de Informática/UFV, um trabalho conjunto de três dissertações de mestrado. O custo e as possibilidades de implementação dos CGRAs em FPGAs foram avaliados em [19].

Duas abordagens para o algoritmo de mapeamento foram desenvolvidas. Ambas abordagens visam o mapeamento dinâmico dos laços em tempo de execução. Com o objetivo de reduzir o tempo de mapeamento, ambas as abordagens são baseadas em heurísticas gulosas que percorrem o grafo de fluxo de dados do laço apenas uma vez. O grafo é percorrido em largura.

A abordagem proposta nesta dissertação, visa uma implementação em hardware do mapeamento composto pelas três operações: escalonamento, posicionamento e roteamento (EPR). O grafo é descrito como uma lista de arestas, o que simplifica a implementação em hardware com uma máquina de estados. O algoritmo simulado e validado nesta dissertação foi implementado em VHDL [19].

A segunda abordagem [30] faz o processamento do grafo por vértice e foi implementada para compilacao JIT (*Just-in-Time*). O algoritmo também foi modelado com o modelo de máquina de estados [30], e uma versão em VHDL pode ser desenvolvida futuramente. Além disso, a segunda abordagem [30] realizou uma redução no número de registradores.

No trabalho base [15] para esta dissertação, o posicionamento sequencial das UFs gera conflitos de roteamento na rede multiestágio. Mostramos nesta dissertação que a utilização de uma permutação no último estágio da rede, diminui os conflitos de roteamento entre as UFs mesmo com um posicionamento sequencial.

Este trabalho se diferencia de [15] em alguns aspectos. O escalonamento é

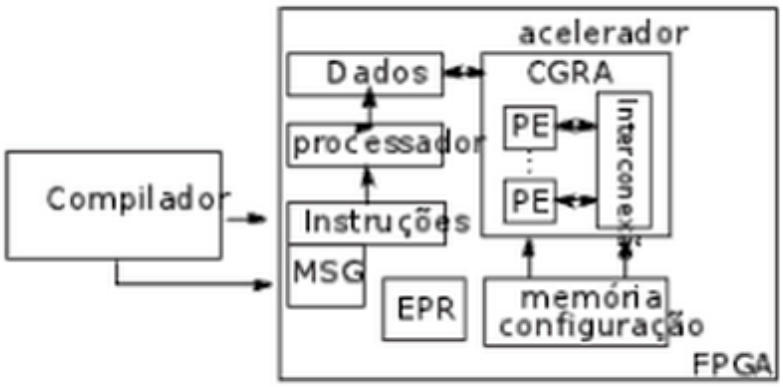

Figura 2.6. Diagrama do ambiente de mapeamento

feito em conjunto com o posicionamento e o roteamento da aresta, enquanto em [15] o escalonamento é realizado separadamente do posicionamento e roteamento, percorrendo o grafo usando ASAP e ALAP para determinar o balanceamento de cada aresta e inserindo o número de registros necessários para balancear o grafo. Só então, o algoritmo executa o posicionamento e roteamento para cada aresta (já balanceada) e verifica se é possível rotear.

2.4 Arquiteturas Reconfiguráveis de Grão-Grosso (CGRAs)

Propor um novo CGRA não é tema principal deste trabalho. Porém o CGRA proposto em [15] foi melhorado através da modificação da rede de interconexão. Nesta seção iremos rever os conceitos e partes do CGRA com destaque para interconexões do modelo proposto em [15]. Para descrever melhor as CGRAs, pode-se dividi-la em três partes: unidades funcionais (UF), granularidade e redes de Interconexão.

As unidades funcionais são responsáveis pela operação que é executada pelo nó, que pode ser uma soma, uma multiplicação, uma operação lógica dentre outras. Se a arquitetura for homogênea, uma UF pode realizar qualquer operação. Se for heterogênea, as UFs são divididas em grupos, onde as UFs de um determinado grupo realizam em geral um conjunto de operações diferente das operações realizadas pelos

outros grupos.

A granularidade diz respeito ao número de bits que será calculado nas unidades. AS FPGAs operam em nível de bit, enquanto as CGRAS operam em nível de palavras [21]. Quanto menor a granularidade, mais flexível é a arquitetura, porém maior é o tamanho da memória de configuração. Uma granularidade maior reduz a complexidade.

As redes de interconexão são responsáveis pela comunicação entre as UFs dentro da arquitetura. Elas tem impacto no custo da implementação física e na complexidade dos algoritmos de mapeamento.

As redes de interconexão utilizadas neste trabalho serão detalhadas a seguir.

2.4.1 Redes de Interconexão

As redes de interconexão podem ser estáticas ou dinâmicas [14]. As redes estáticas usam topologia fixa, com conexão direta entre os componentes. Nas redes dinâmicas, as ligações são estabelecidas conforme a demanda, através de comutadores. A seguir veremos *Crossbar* e multiestágio.

2.4.1.1 Rede *Crossbar*

A rede *Crossbar* permite qualquer permutação na conexão das entradas e nas saídas. Ela tem um custo O(n^2) e sua utilização se torna inviável em FPGA para n>=32 e caminhos de dados de 32 bits [24]. Sua vantagem é ser uma rede não bloqueante, pois como mencionado, todas as ligações entre as entradas e saídas podem ser realizadas concorrentemente.

A Figura 2.7 mostra um exemplo de uma rede de interconexão *Crossbar* onde podemos ver os n^2 comutadores para n=7.

2.4.1.2 Rede *Multiestágio*

Com custo O(nlog(n)) e construção viável, a rede multiestágio é a principal rede de interconexões utilizada nesta dissertação. Usamos a rede Ômega bloqueante e como o próprio nome indica, não permite que qualquer conexão entre entrada e saída seja estabelecida cncorrentemente. Portanto, podem ocorrer conflitos de roteamento para um subconjunto de permutações. Porém mostraremos como reduzir os conflitos.

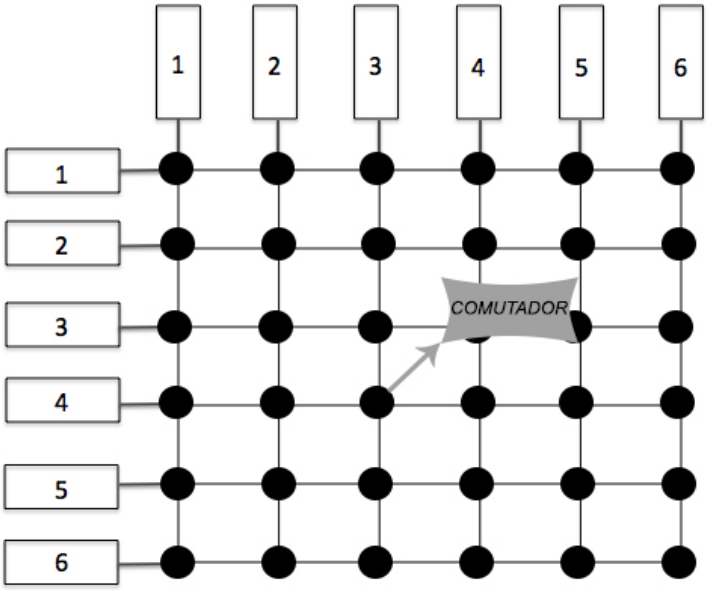

Figura 2.7. Rede *Crossbar* de custo O(n^2)

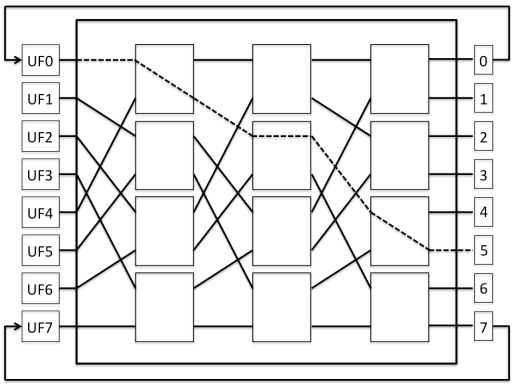

Figura 2.8. Rede multiestágio ômega 8x8 de custo O(nlog(n))

A Figura 2.8 mostra um exemplo de uma rede multiestágio de tamanho 8, sendo usada para conectar oito unidades funcionais (UF's), onde vemos o roteamento da ligação $UF0 \rightarrow UF5$.

A Figura 2.9 mostra um exemplo de conflito de roteamento, onde podemos observar a saída do comutador que está circulada em destaque. Suponha que a ligação $UF0 \rightarrow UF5$ já esteja feita. Ao tentar rotear a ligação $UF2 \rightarrow UF4$, um conflito é gerado na primeira saída do segundo comutador da segunda coluna. Esta saída já está sendo utilizada pela ligação $UF0 \rightarrow UF5$, portanto a ligação $UF2 \rightarrow UF4$ não pode ser realizada e o roteamento falha.

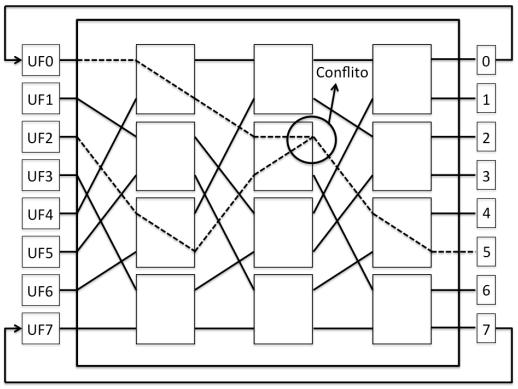

Figura 2.9. Conflito na rede de roteamento multiestágio

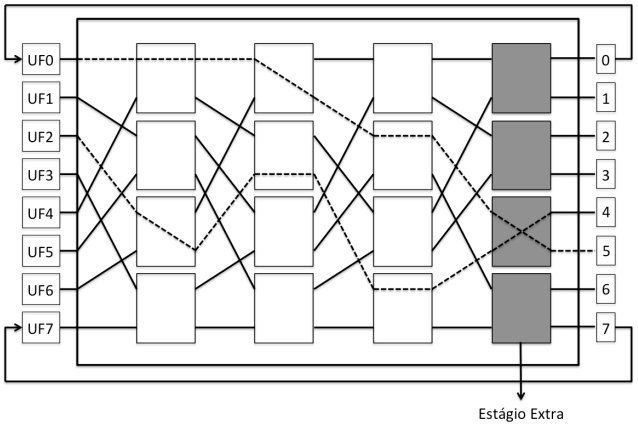

Figura 2.10. Utilização do estágio extra para resolver o conflito

O uso de estágios extras na rede é uma alternativa. Um estágio extra insere uma coluna de comutadores da rede, aumentado as possibilidades de roteamento. Para cada par de entrada e saída, o número de caminhos é dado por r^x onde **r** é o RADIX do comutador utilizado e **x** é o número de estágios extras. Um estágio extra aumenta em 2 vezes as possibilidades de roteamento entre os endereços de entrada e saída em uma rede radix 2 [15]. A Figura 2.10 mostra como isso resolveria o problema do conflito anterior.

Neste trabalho foram utilizados comutadores RADIX 4 como mostra a Figura 2.11. Note que, para 16 UFs, são necessários apenas 2 colunas com 4 comutadores em cada uma delas poder para construir a rede, ao contrário da rede RADIX 2 que, com 16 UF, necessita de 4 colunas com 8 comutadores em cada, como mostra a Figura 2.12. Apesar de reduzir o número de comutadores, aumenta o custo individual dos

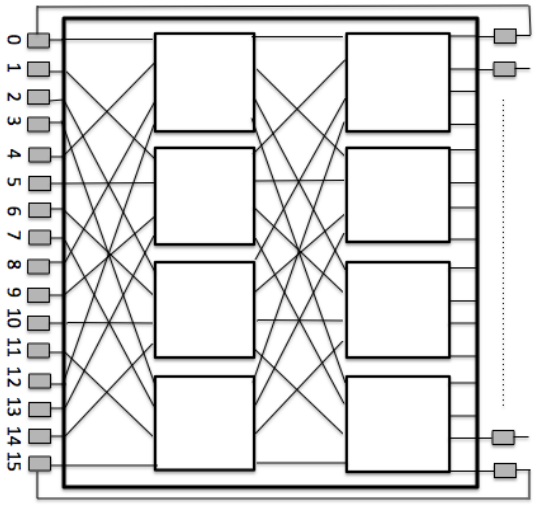

Figura 2.11. Rede multiestágio utilizando comutadores radix 4

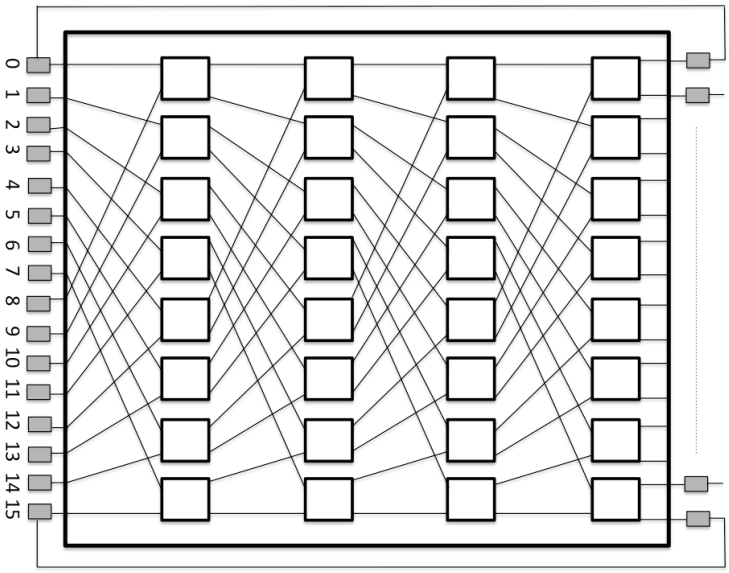

Figura 2.12. Rede multiestágio utilizando comutadores radix 2

comutadores. Aumentar o radix reduz os conflitos, mas em geral aumenta o custo, já que o custo do comutador e quadrátcico. Porém o radix4 é mapeado de forma eficiente em FPGAs com LUT de 6 entradas.

As redes de interconexão não são o tema principal deste trabalho, são apenas utilizadas como base para o algoritmo. Maiores detalhes em [15]

O capítulo a seguir mostrará como as soluções anteriores foram melhoradas pela heurística proposta nesta dissertação.

Capítulo 3

Heurística de Modulo Scheduling

3.1 Introdução

Este capítulo apresenta a heurística para o algoritmo de *Modulo Scheduling* proposta nesta dissertação, que executa o escalonamento, posicionamento e roteamento (EPR) de um grafo de operações de um laço em uma arquitetura alvo reconfigurável. O algoritmo é viável para realizar o mapeamento em tempo de execução.

O algoritmo foi implementado com o modelo de máquina de estados de Moore na linguagem C. A máquina de estados proposta neste trabalho foi validada em [19] com uma implementação em VHDL em FPGA.

Esta heurística de *Modulo Scheduling* é gulosa e utiliza o intervalo de iniciação mínimo MII como ponto de partida para realizar o EPR. Caso haja falha, o algoritmo incrementa o intervalo de iniciação e começa tudo novamente. O algoritmo percorre o grafo em largura processando cada aresta uma única vez.

Existem quatro tipos de arestas. Suponha a aresta $x \rightarrow y$:

1. O nó **x** é uma entrada e o nó **y** é interno. Ambos ainda não foram visitados. O algoritmo aloca nó interno na primeira UF livre.

2. O nó **x** é uma entrada e o nó interno **y** já foi visitado anteriormente. Faz-se o posicionamento somente da entrada na primeira UF alocada pra **y**.

3. Os dois nós **x** e **y** são internos, onde o primeiro nó **x** já foi necessariamente visitado e alocado, restando apenas alocar o nó destino **y**. Posteriormente é feito o roteamento da aresta entre a unidade de **x** e uma nova unidade alocada para **y**.

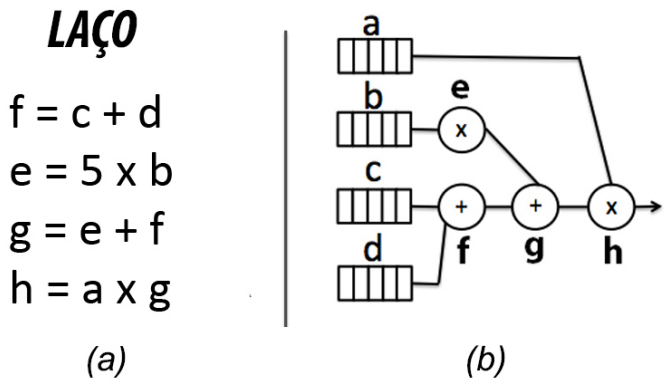

Figura 3.1. (a) Laço relacionado ao grafo, (b) Grafo de Operações [34]

4. **x** e **y** são nós internos e já foram visitados e alocados anteriormente. Resta saber se a aresta $x \rightarrow y$ deve ser balanceada. Em caso de desbalanceamento, é necessário inserir registros e fazer o roteamento da unidade **y** passando por todos os registros inseridos até chegar à nova unidade de destino **y**.

Neste trabalho foram considerado operadores homogêneos, que podem executar qualquer operação lógica aritmética.

A seguir será explicado em maiores detalhes como funciona o algoritmo guloso de *Modulo Scheduling* com o modelo de máquina de estados, onde primeiramente testaremos os grafos em arquiteturas com recursos suficientes para mapeá-lo. Posteriormente, as arquiteturas utilizadas passam a ter uma quantidade de UFs menor do que o número de operações do grafo a ser mapeado.

3.1.1 EPR Com Rede *Crossbar*

Essa seção descreve a implementação do algoritmo EPR onde a arquitetura possui duas redes *Crossbar* e recursos suficientes para mapear todo o grafo com II=1. A rede *Crossbar* permite um roteamento ideal, ou seja, não haverá conflitos, como já mencionado na Seção 2.4.1.

A Figura 3.1 (b) mostra o grafo que será usado de exemplo para explicar o funcionamento do algoritmo proposto. Este grafo é simples mas visita todos os estados da máquina, cobrindo todos os casos. A Figura 3.1 (a) mostra como seria o laço de onde este grafo foi extraído.

O algoritmo é descrito por uma máquina de estados como ilustrado na Figura 3.2. A Figura 3.3 ilustra o modelo de código da implementação da máquina na

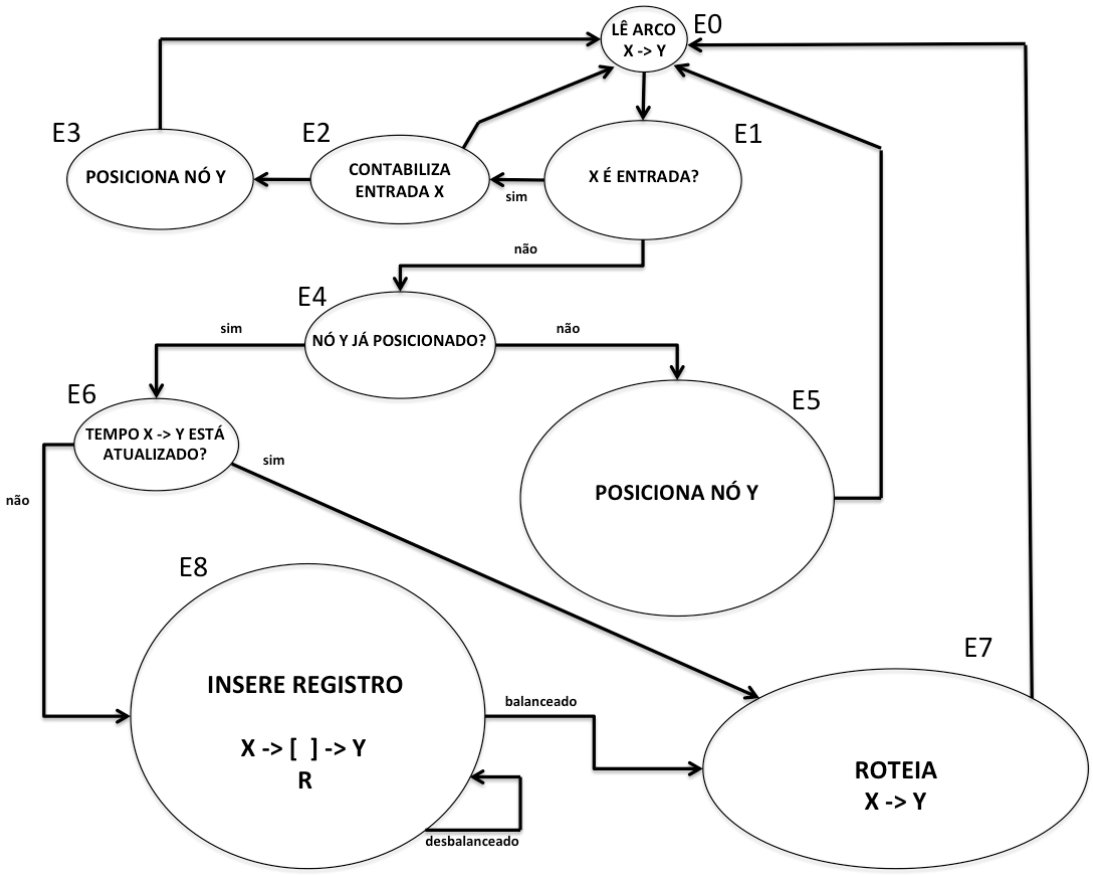

Figura 3.2. Máquina de estados do algoritmo EPR *Crossbar*

```
while(true) {
    switch (estado) {
        case le: estado = X;
                 ação_le();
                 break;
        case ....
    }
}
```

Figura 3.3. implementação em C da máquina de estados

linguagem C.

Para melhor entendimento, a Figura 3.4 mostra a tabela de ID's utilizada ao longo deste capítulo. Cada vértice é identificado por um inteiro, que será seu ID. Através dos IDs, os vetores que controlam o escalonamento, posicionamento e roteamento são indexados.

O grafo da Figura 3.1 (b) é percorrido em largura, passando em uma aresta de

TABELA DE ID's

NÓ	ID
a	1
b	2
c	3
d	4
e	5
f	6
g	7
h	8
R1	9
R2	10

Figura 3.4. Id's dos nós do grafo

cada vez. Cada aresta é representada por um nó fonte **x** e um de destino **y**. O arco $x \rightarrow y$ será processado de acordo com a máquina de estados da Figura 3.2, onde os estados analisam **x** e **y** e determinam o caminho a ser seguido para processar o arco no diagrama de Máquina de Estados.

Na implementação na linguagem C, cada estado é implementado por uma cláusula *case* e executa uma função no modelo de Moore para máquina de estado. Duas restrições foram impostas para as estrutura de dados: variáveis e vetores. As variáveis podem ser diretamente implementadas com registros e os vetores por memória distribuída. Se uma variável recebe uma atribuição, por exemplo $\mathbf{a} = \mathbf{b}$ em um estado, então o valor de **a** só poderá ser lido no próximo estado.

A máquina de estados foi elaborada com 9 estados. Esta máquina de estados funciona da seguinte forma (utilize a figura 3.2 para melhor entendimento):

- O estado **E0** é responsável pela leitura dos arcos. A cada iteração um novo par de arcos é lido.
- O estado **E1** é responsável pela verificação se o nó **x** é uma entrada ou não. O fluxo pode seguir para o estado **E2** se **x** for entrada, e para o estado **E4** se **x** é interno.
- O estado **E2** é responsável pela contagem de nós de entrada.

- O estado **E3** é responsável pelo posicionamento do nó **y**. Feito isso, retorna ao estado inicial (**E0**) para processar a próxima aresta.
- O estado **E4** é responsável por verificar se o nó **y** já foi visitado anteriormente ou não. Se sim, ele passa o fluxo para o estado **E6**. Caso o contrário o fluxo passa para o estado **E5** para alocar uma nova unidade.
- O estado **E5** é responsável pelo posicionamento do nó **y** na primeira UF livre e também pelo roteamento da aresta. Ao final da execução ele retorna ao estado inicial.
- O estado **E6** é responsável pela verificação do balanceamento dos caminhos caso o nó **y** já esteja posicionado. Se o arco $x \rightarrow y$ estiver desbalanceado, é necessário inserir registros entre esta ligação até que o tempo de **y** seja uma unidade maior do que o tempo de **x**, podendo assim ser feito o roteamento. Se caso o grafo já estiver balanceado, apenas o roteamento $x \rightarrow y$ é realizado.
- O estado **E7** é responsável pelo roteamento da aresta. Retorna ao estado inicial.
- O estado **E8** é responsável pela inserção de registro, colocando o registro na UF de **y**, e **y** numa próxima UF livre, a fim de balancear o grafo. Esta inserção é repetida até que o caminho esteja balanceado. Este estado também é responsável pelo roteamento do nó **x** e o primeiro registro inserido e do roteamento entre os registros inseridos. O último roteamento é feito em **E7**.

A seguir será explicado passo a passo o funcionamento do algoritmo utilizando o grafo proposto na Figura 3.1 (a).

3.1.1.1 Algoritmo

Na Figura 3.5(a) vemos o exemplo do primeiro arco lido, onde **x**=**c** é uma entrada de dados e **y**=**f** é um operador interno, mais especificamente um operador de soma. Neste exemplo, o arco $c \rightarrow f$ será $x \rightarrow y$. Os vetores ilustrados nas figuras utilizam os IDs relacionados na figura 3.4 vista anteriormente. A Figura 3.6 mostra a máquina de estados para esta situação.

O vetor posicionamento indica qual é a UF livre em que o nó (ID) está alocado. O vetor escalonamento indica em que tempo o nó (ID) foi visitado.

No estado **E1** é verificado no vetor *Posicionamento*, se a posição **x** já foi alocada ou está vazia. Se estiver vazia, significa que não existe nenhuma *UF* alocada

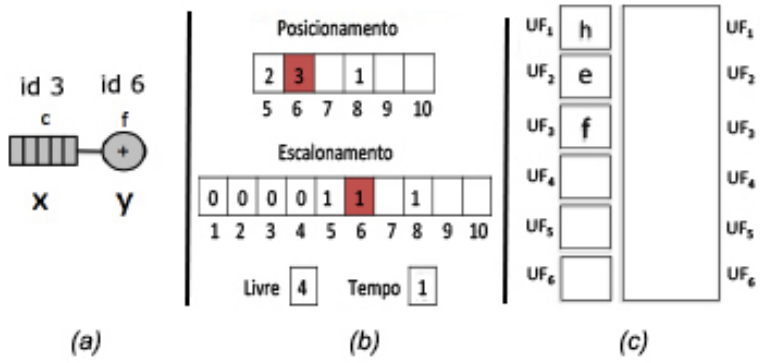

Figura 3.5. (a) Arco Atual, (b) Comportamento das Estruturas *Posicionamento*, *Escalonamento*, *Livre* e *Tempo*, (c) Posicionamento da Arquitetura

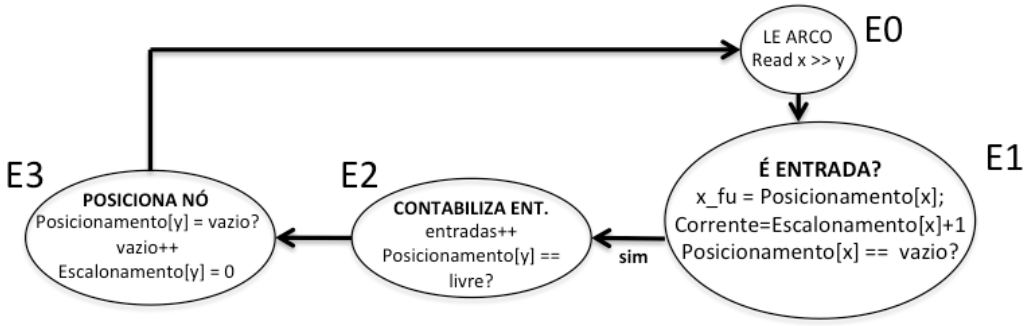

Figura 3.6. máquina de estado parcial

para **x**, e além disso, sendo ele nó fonte da aresta, podemos caracteriza-lo como um nó de entrada. Tendo isso, a máquina passa para o estado **E2** onde um contador de entradas é atualizado e é verificado se o vetor *Posicionamento* na posição **y** está vazio. Esta verificação é feita para saber se o nó **y** já foi visitado anteriormente e já possui uma UF alocada. Neste caso, foi a primeira vez que este nó foi visitado, logo a máquina passará para o estado **E3**. No estado, **E3** uma variável chamada *Unidade Livre* informa qual é a próxima *UF* livre como pode ser visto na Figura 3.5. Como **y** tem como predecessor uma entrada, será escalonada no tempo igual a 1, esta informação é armazenada no vetor chamado *Escalonamento*. Após feito isso, a máquina de estado volta ao estado **E0** para ler a próxima aresta. Como veremos para outros casos, as ações são simples em cada estado e viáveis de implementar em hardware.

A Figura 3.5 ilustra o estado inicial dos vetores Posicionamento e Escalonamento e também os valores iniciais das variáveis Livre e Tempo. Como as arestas

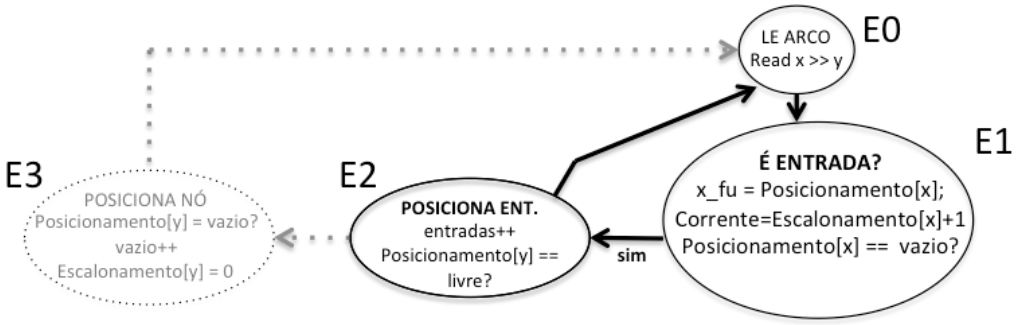

Figura 3.7. Máquina de estado parcial

$a \rightarrow h$, $b \rightarrow e$ e $c \rightarrow f$ foram lidas nessa ordem, na Figura 3.5, em (a) vemos este último arco sendo processado, em (b) vemos os vetores e as variáveis sendo atualizadas para este arco e em (c) vemos quais UFs estão sendo utilizadas na arquitetura de tamanho 8. O tempo dos nós de entrada é igual a zero, e já está ilustrados na figura 3.5.

No próximo passo será avaliada uma aresta onde **y** já foi posicionado em uma UF, a partir do estado **E1**. Para continuar o algoritmo observe a Figura 3.7.

Neste caso, o arco $x \rightarrow y$ agora corresponde aos nós d e f. No estado **E1** novamente se verifica que **x** é uma entrada, caso nenhuma *UF* tenha sido alocada. A máquina então passa para o estado **E2**, onde encontramos uma situação diferente para o primeiro caso. Nessa etapa é verificado que no vetor *Posicionamento* o ID correspondente a **y** não aponta para uma unidade vazia, pois o nó **y** já foi visitado na primeira etapa, a unidade alocada está armazenada no vetor *Posicionamento*. Para este segundo caso, o arco $x \rightarrow y$ está concluído, o algoritmo retorna ao estado **E0**.

A Figura 3.8 ilustra máquina de estados para esta situação. A Figura 3.9(a) ilustra o próximo tipo de aresta a ser visitada, onde o nó **y** ainda não foi posicionado.

Nesta etapa o estado **E1** começa diferente das etapas anteriores pois, ao se verificar se o nó **x** está alocado em alguma *UF*, a resposta será positiva, indicando que este nó não é uma entrada. Aqui a máquina deverá seguir outro caminho, indo para o estado **E4**. Antes de passar para este estado, uma variável chamada *Tempo Atual* armazenará o valor do tempo corrente, que é a soma do tempo de **x** armazenado em *Escalonamento* mais 1(uma) unidade, ou seja, o tempo do sucessor é igual ao tempo do predecessor acrescentado de uma unidade. No estado **E4**, é verificado se o nó **y** já foi visitado anteriormente, neste caso ele ainda não foi. O algoritmo

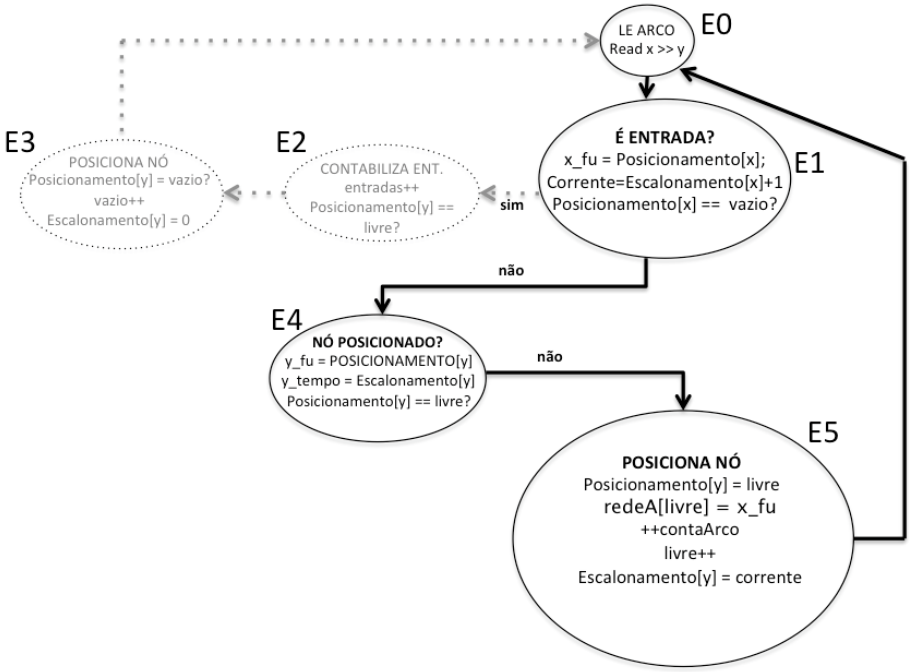

Figura 3.8. Máquina de estado parcial

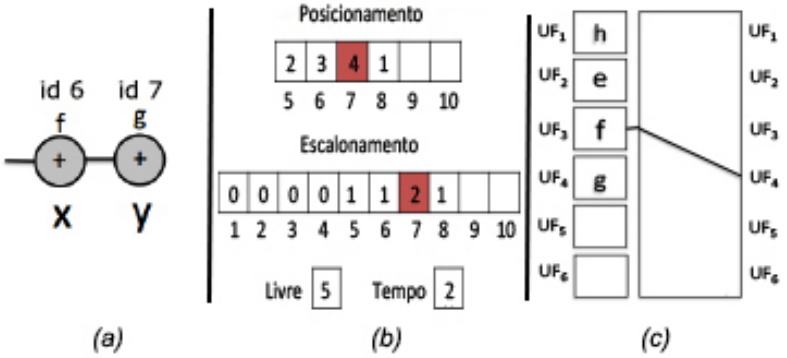

Figura 3.9. (a) Arco Atual, (b) Comportamento das Estruturas *Posicionamento*, *Escalonamento*, *Livre* e *Tempo*, (c) Posicionamento da Arquitetura

primeiramente atualiza o tempo de **y** no vetor *Escalonamento* e passa para o estado **E5**. Nesse estado o vetor *Posicionamento* é atualizado alocando-se a próxima *UF* livre para *y*, usando a variável *Unidade Livre* e esta variável é incrementada. Em seguida é feito o roteamento de **x** para **y**. Como na rede *Crossbar* não há conflitos, esse roteamento será efetuado com sucesso. O tempo de **y** será o tempo corrente armazenado na variável *Tempo Atual*. Finalizando, o algoritmo processa a próxima aresta. A Figura 3.9 mostra como ficaram as estruturas após o processamento desse último arco.

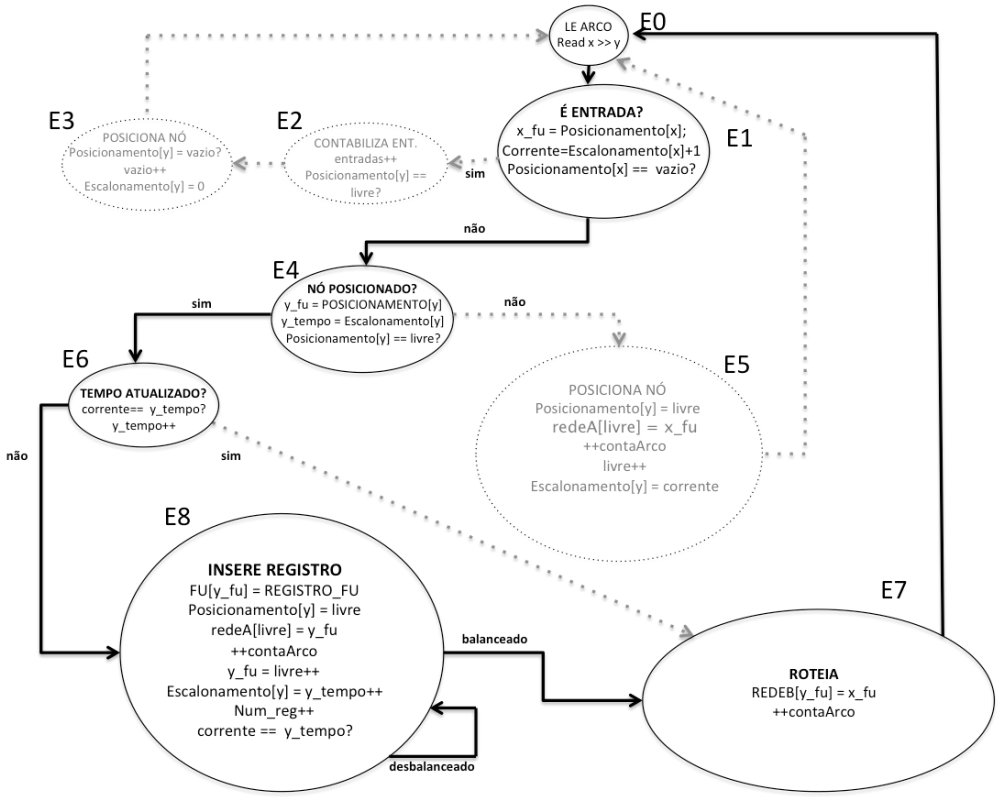

Figura 3.10. Máquina de estado parcial

O próximo caso, refere-se ao caso 4 da seção 3.1 e está ilustrado na Figura 3.11 onde em (a) estão destacados os nós que serão lidos neste caso. A Figura 3.10 destaca os estados que a máquina passará para realizar a computação desta aresta.

No estado **E0**, que é também o ponto de partida nessa etapa, verifica-se que o nodo **x** já foi visitado e não é uma entrada, passando então para o estado **E4**. Desta vez, em **E4**, é verificado que o vetor Posicionamento já possui uma UF alocada para **y**, o que significa que **y** já foi visitado anteriormente. O próximo passo é então verificar se o tempo de **y** está sendo alterado, ou seja, se o tempo de **y** armazenado no vetor *Escalonamento* é igual ao tempo da variável *Tempo Atual*, o que será realizado no estado **E6**. Neste caso os tempos são iguais, o nó **y** portanto está balanceado e pode ser roteado ao seu par. A máquina então passa para o estado **E7**, onde faz a ligação $x \rightarrow y$ na rede *Crossbar* e, em seguida, o processamento do próximo arco, retornando para o estado **E0**.

A Figura 3.11 mostra como ficaram os vetores, as variáveis e a ocupação da arquitetura de controle após a execução desse passo.

Agora veremos como se comporta a máquina de estados quando a aresta está desbalanceada.

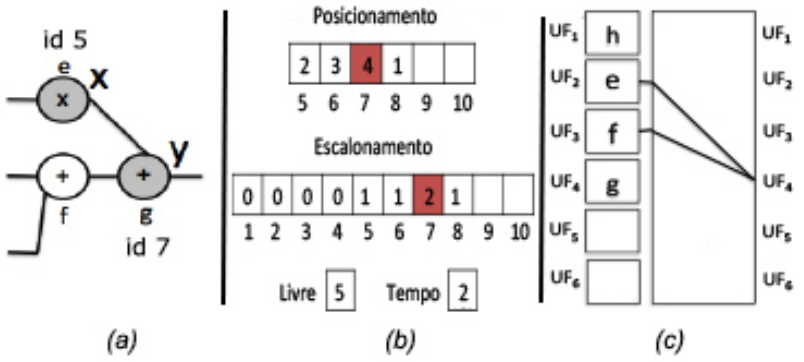

Figura 3.11. (a) Arco Atual, (b) Comportamento das Estruturas *Posicionamento*, *Escalonamento*, *Livre* e *Tempo*, (c) Posicionamento da Arquitetura

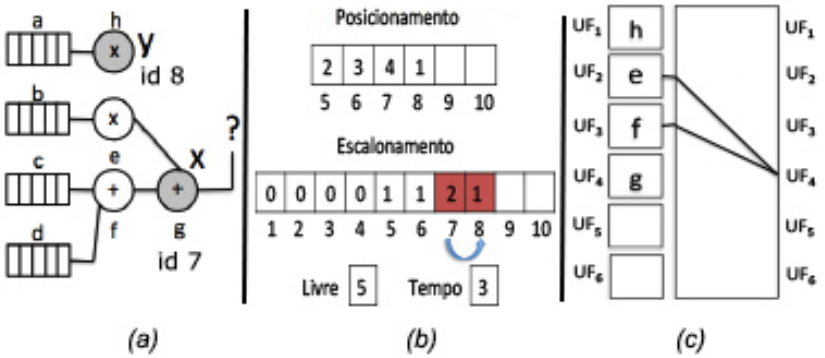

Figura 3.12. (a) Arco Atual, (b) Comportamento das Estruturas *Posicionamento*, *Escalonamento*, *Livre* e *Tempo*, (c) Posicionamento da Arquitetura

Novamente, a máquina começará o estado **E0**, passando para o estado **E4** após verificar que o nó **x** não se trata de uma entrada. Ainda em **E4** verifica-se que o nó **y** já foi visitado anteriormente, passando assim para o estado **E6** onde é verificado se o tempo corrente e o tempo de **y** são iguais.

Continuando o algoritmo, no estado **E6**, ao se comparar o tempo de **y** com o tempo em *Tempo Atual* é verificado que há um desbalanceamento do grafo como se pode ver na 3.12, pois o nó **y** tem tempo 1 gravado em *Escalonamento*, já a variável *Tempo Atual* tem tempo 3 (pois *Tempo Corrente* = *Escalonamento[x]+1*, e **x** tem tempo 2) resultando em uma diferença de 2 unidades de tempo (delta = 1-3). O próximo passo realizado pela máquina de estados é passar para o estado **E8** que insere registros a fim de balancear esse grafo. Em **E8**, a *UF* que armazenava **y** passará agora a armazenar um registrador **R1**, enquanto **y** será alocada na próxima *UF* livre. Em seguida, esse novo arco $R1 \rightarrow b$ deve ser roteado na rede *Crossbar*. O registrador guardará o tempo de **y** e este terá o tempo acrescentado em uma

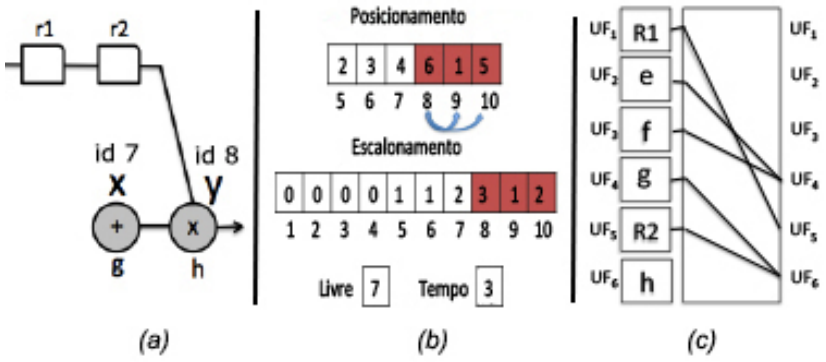

Figura 3.13. (a) Arco Atual, (b) Comportamento das Estruturas *Posicionamento*, *Escalonamento*, *Livre* e *Tempo*, (c) Posicionamento da Arquitetura

unidade, um contador de registro também é aumentado.

Novamente é preciso verificar se o tempo de **y** ainda está desbalanceado, como a diferença era de 2, ainda falta colocar mais um registro para que este arco esteja balanceado. O nó **y** dará lugar a um novo registro **R2** e ocupa o lugar de uma próxima *UF* livre, e o roteamento desse arco $R2 \rightarrow b$ é realizado. O tempo de **y** é acrescido de uma unidade e verifica-se novamente se os tempos estão corretos. Dessa vez está tudo correto e a máquina passa para o estado **E7**. A Figura 3.13 mostra como ficaram as estruturas de armazenamento após o balanceamento do grafo.

Para finalizar, no estado E7, o arco $x \rightarrow y$ será finalmente roteado na rede *Crossbar*.

A Figura 3.14 ilustra um exemplo do último item listado, onde há desbalanceamento do arco. A Figura 3.14 (a) mostra o arco $x \rightarrow y$ que representa respectivamente g e h, onde está se tentando posicionar h. Porém a Figura 3.14 (b) mostra que o tempo de g (nó fonte) é maior do que o tempo de h (nó destino). Para resolver o desbalanceamento, é necessário inserir registros entre o arco, até que o tempo de h seja maior do que o tempo de g. Neste caso, serão necessários 2 registros para que o arco seja balanceado. O primeiro registro passa a ocupar a UF que armazenava a operação de h, e h é alocado numa próxima unidade livre. Posteriormente, como o arco ainda continua desbalanceado, outro registro é inserido, ocupando novamente a UF que pertencia a h, e h é alocado na próxima UF livre. A Figura 3.14 (d) mostra como as UFs ficaram alocadas para cada operação e o tempo final de cada nó.

3.1.2 EPR com Rede Multiestágio

Esta seção apresenta o algoritmo EPR utilizando redes multiestágios Omega como interconexão. A vantagem da rede Omega em comparação com a rede *Crossbar* é

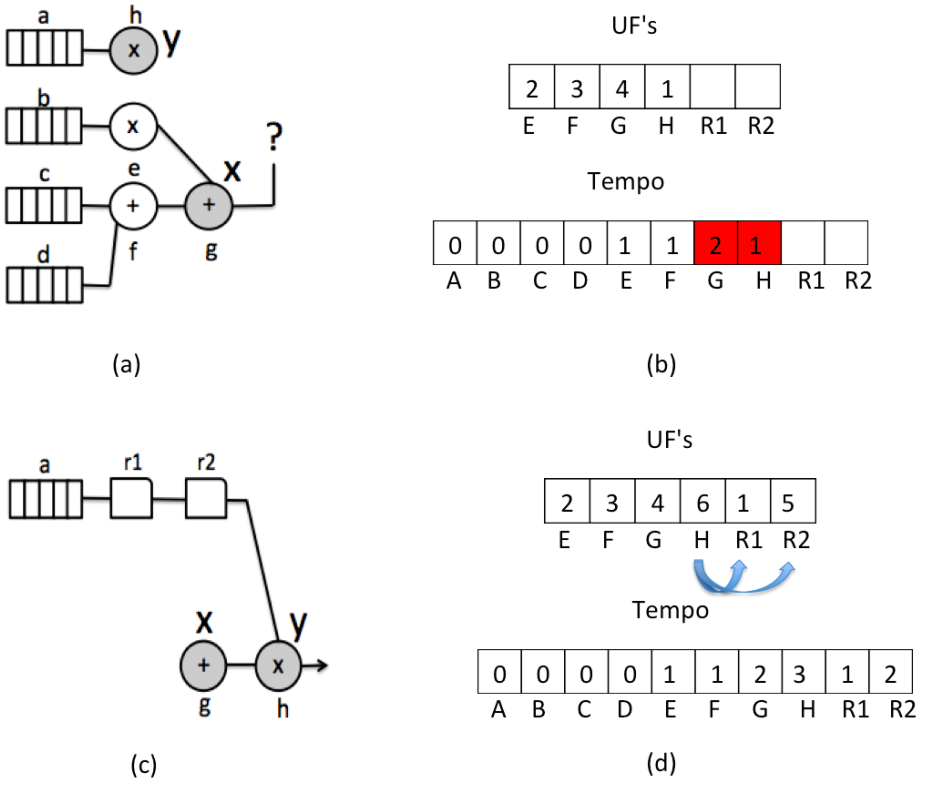

Figura 3.14. (a) Arco $g \rightarrow h$ desbalanceado, (b) Tempo de g é maior do que h, (c) Inserção de registros entre o arco $g \rightarrow g$, (d) Tempo de h se torna 1 unidade maior do que g após balanceamento

a redução do custo de implementação de O(n^2) para O(nlog(n)). Similar à rede *Crossbar*, a arquitetura utiliza duas redes multiestágio como mostra a Figura 3.15. Essa arquitetura foi proposta em [15]. Entretanto, uma modificação foi necessário no último estágio para reduzir os conflitos, alternando randomicamente a posição das UFs nas saídas da rede global.

A substituição da rede *Crossbar* pela multiestágio, reduz o custo total da arquitetura, podendo ser possível o uso de 64 unidades funcionais. Mesmo a construção de uma arquitetura de 256 unidades funcionais utilizando a rede multiestágio[15][14], o que não é possível com a rede *Crossbar*.

A Figura 3.16 mostra a máquina de estados para o EPR com multiestágio. Ela difere da máquina da Figura 3.2 somente na parte de roteamento, onde o vetor *RedeA[UF livre]* é substituído pela função *Rotear* quem implementa o algoritmo de roteamento proposto em [14]. A função *Rotear* verifica a disponibilidade de comutadores na rede de interconexão (Figura 3.15). Dada uma aresta, a função verifica se é possível conectar **x** em **y**.

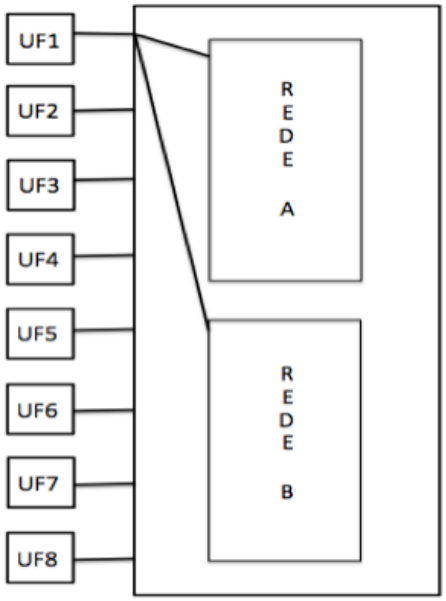

Figura 3.15. Rede multiestágio com 2 redes de interconexão

Como estão sendo utilizadas arquiteturas menores ou iguais ao tamanho do grafo e a heurística é gulosa, se houver falha de roteamento ao longo da execução o algoritmo aborta a execução. Aumentar o intervalo de iniciação em uma unidade ou mais é uma das alternativas para contornar este problema, porém, para isto, são necessárias estruturas adicionais que serão mostradas na seção 3.2

3.1.2.1 Algoritmo

O algoritmo EPR utilizando a rede multiestágio é bem semelhante ao anterior, que usa a rede *Crossbar*. A máquina de estado possui o mesmo número de estados e o fluxo entre eles é idêntico. Mas como já foi dito anteriormente, a função *Rotear* é o que diferencia os algoritmos.

Os únicos estados que são diferentes são os estados **E5**, **E7** e **E8**. Com a função *Rotear*, o algoritmo de escalonamento pode não encontrar um roteamento válido, devido a conflitos.

A sequência de estados para o primeiro tipo de aresta é exibido na Figura 3.17.

Neste caso, após ser verificado que **x** não é um nó de entrada no estado **E1** e, em **E4** é verificado que **y** ainda não foi visitado, a máquina de estado chega ao estado **E5**, faz o posicionamento de **y** e em seguida tenta rotear o arco $x \rightarrow y$ na rede multiestágio. Se houver algum conflito na rede, a função *Rotear* retorna uma falha e o algoritmo irá abortar a execução. No caso de sucesso, o arco $x \rightarrow y$ foi roteado

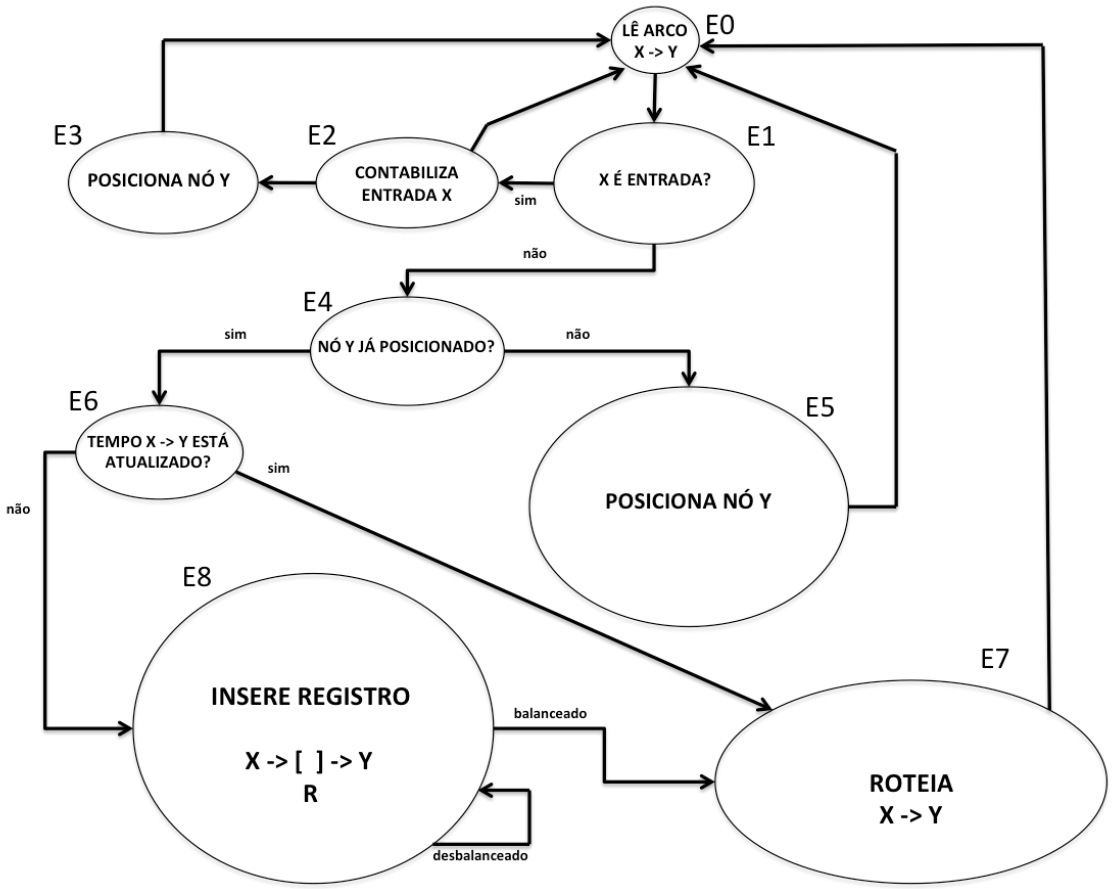

Figura 3.16. Máquina de estados do algoritmo com rede *Multiestágio*

na rede e a configuração da rede é atualizada nos vetores. Também é atualizado o vetor *Escalonamento[b]* que armazena o tempo de **y**.

A Figura 3.18 mostra um exemplo onde o arco $x \rightarrow y$ não possui nós de entrada e já foi percorrido anteriormente. Neste caso, o algoritmo apenas irá tentar rotear a ligação entre **x** e **y**.

Partindo do estado **E1**, verifica-se que o nodo **x** é um nó de entrada, passando então para o estado **E4**. Em **E4**, é verificado **y** já foi posicionado, e a unidade está armazenada no vetor *Posicionamento*. Semelhante à implementação com *Crossbar*, o próximo passo será a verificação do balanceamento do grafo em relação ao tempo de **y** e o tempo em *Tempo Atual*, que é verificado no estado **E6**. Neste caso os tempos são iguais, o nó **y** está balanceado e pode ser roteado. A máquina então passa para o estado **E7**, onde tenta fazer a ligação $x \rightarrow y$ na rede *Multiestágio*.

Em seguida, a pré-leitura do próximo arco, retornando para o estado **E1**.

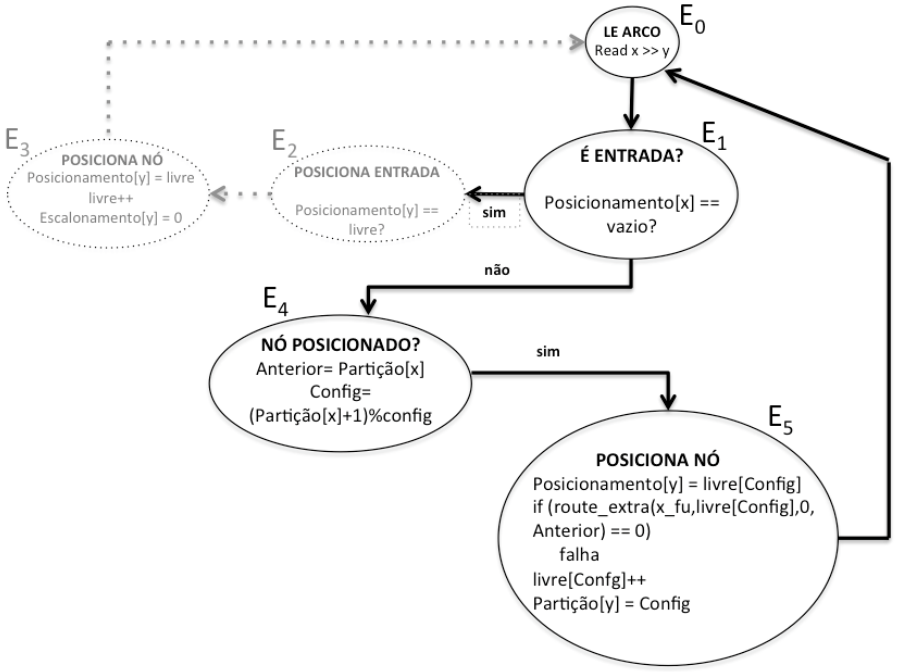

Figura 3.17. Máquina de estado parcial modificada

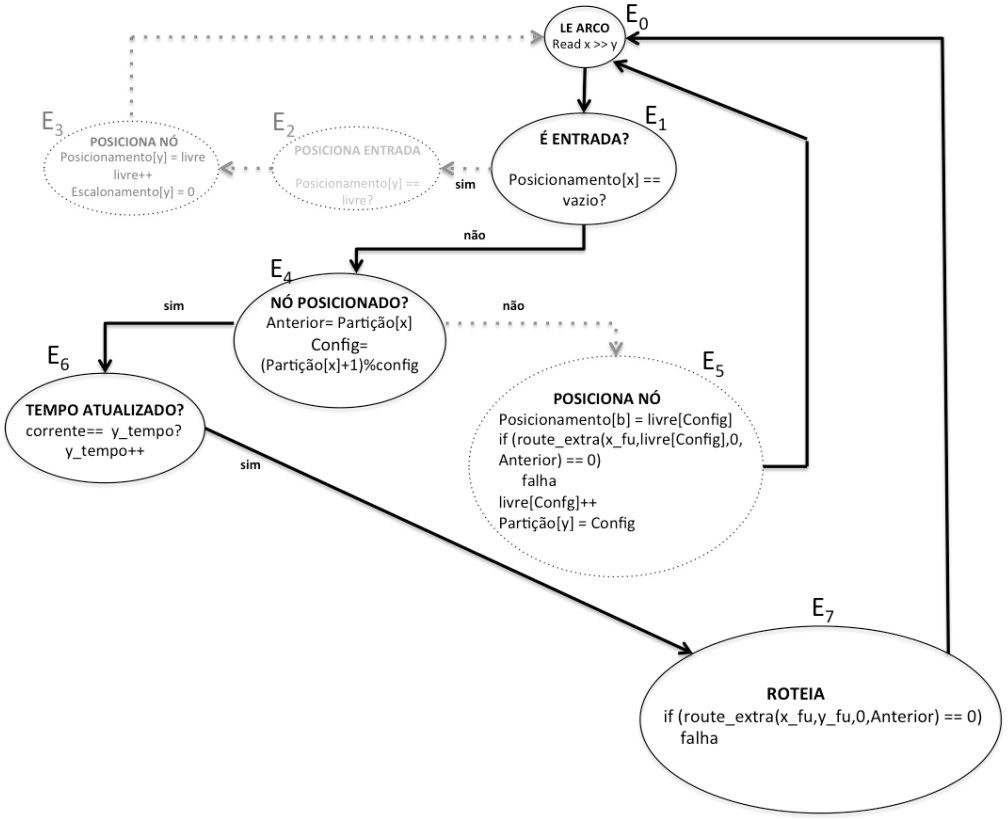

Figura 3.18. Máquina de estado parcial modificada

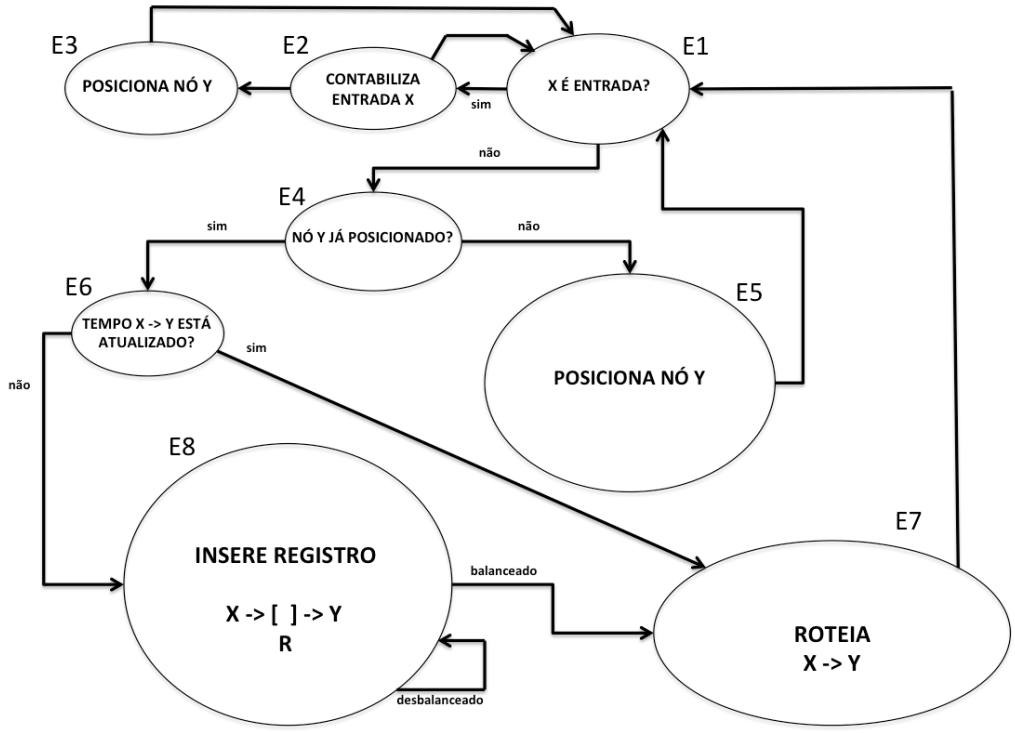

Figura 3.19. Máquina de estado com pré-leitura

3.1.3 Pré-leitura

As máquinas de estado, tanto *Crossbar* como multiestágio, podem ser otimizadas, como por exemplo, onde ao invés de, após o fim de um ciclo, a máquina retornar ao estado **E0** e ler os próximos valores de **x** e **y**, o último estado implementa também a leitura de **x** e **y**, podendo assim ir direto ao estado **E1**, economizando um ciclo de máquina para cada aresta do grafo de operações.

Apesar de simples, esta otimização gera um ganho de 20% no tempo de execução do algoritmo, como será apresentado no capítulo 4.

A Figura 3.19 mostra a máquina de estados utilizando a pré-leitura

3.2 EPR Mutiestágio com Partição Temporal

3.2.1 Algoritmo

Nas seções anteriores, o mapeamento foi realizado em uma arquitetura com recursos suficientes para receber todos os operadores e registros. Considere o exemplo da figura 3.20 onde a arquitetura tem apenas 4 unidades e o grafo tem 6 operações (fi-

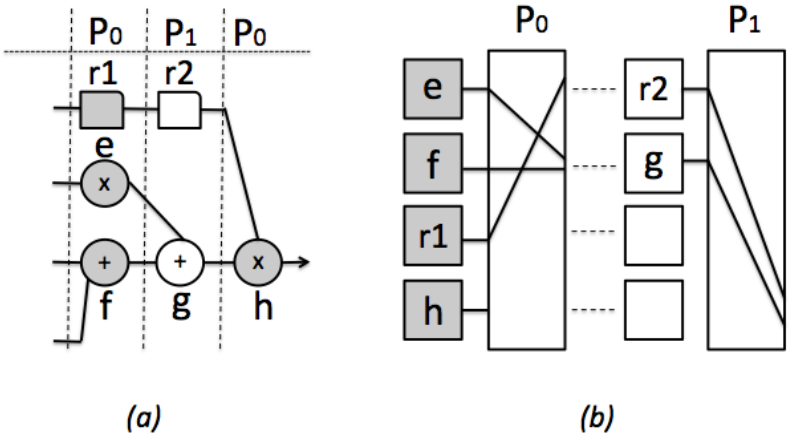

Figura 3.20. (a) Escalonamento e Partições (b) Posicionamento e Roteamento

gura 3.20 (b)). Serão necessárias pelo menos duas partições temporais para mapear o grafo da figura 3.20 (a), pois pode-se mapear 4 nós em *P0* e 2 em *P1*.

A modificação no algoritmo é simples. A Figura 3.21 apresenta outro exemplo, o MII inicial é 2, pois a arquitetura possui 2 UFs e o grafo possui 4 operações. Começando com II igual a 2, o algoritmo posiciona o nó A em uma UF na configuração 0, usando $(tempo_A) mod II$. Em seguida, os nós B e C serão posicionados nas UFs para a configuração 1, pois ambos possuem o mesmo tempo. Posteriormente usando $(IIatual)\ mod\ IIMAX$ nota-se que D será posicionado em uma UF livre da configuração 0.

Outro ponto importante é o roteamento que será realizado entre as partições temporais, ou seja, de A para B por exemplo. Outro exemplo está ilustrado na figura 3.20(b), onde o arco $UF_2(P_0) \rightarrow UF_2(P_1)$ no processamento do arco $f \rightarrow g$ está sendo roteado entre as partições.

A Figura 3.21 mostra mais um exemplo onde o grafo possui mais nós do que o número de UF's da arquitetura alvo. Para contornar o problema, o grafo é dividido em duas partições. Agora, cada partição possui um número de nós que cabe na arquitetura alvo, no caso, 2 nós para 2 UF.

Porém esta abordagem possui desvantagens. O aumento do número de partições reduz a vazão das respostas do algoritmo. Se para um grafo utilizarmos 2 partições, a arquitetura pode ter um tamanho 2 vezes menor para mapear o grafo, porém, após entrar no estágio de equilíbrio do pipeline, as respostas passam a ser calculadas de 2 em 2 unidades de tempo, pois com o II=2.

Como já mencionado, o algoritmo é guloso passando apenas uma vez em cada

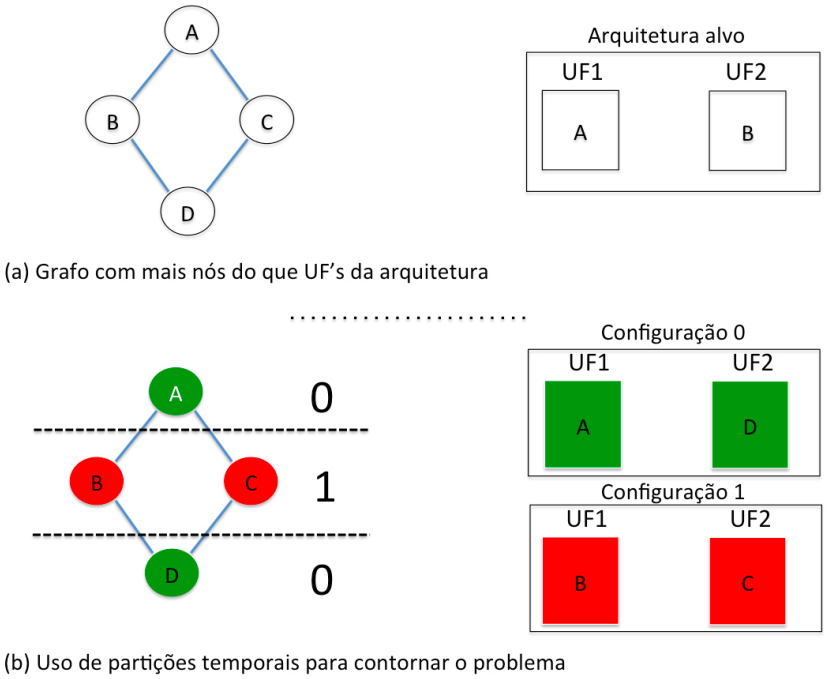

Figura 3.21. (a) Problema de posicionamento do grafo na arquitetura (b) Solução do problema com o uso de partições temporais

arco, isto simplifica o algoritmo e viabiliza a implementação em hardware. Além disso, ele executa poucas operações por arco. Porém, o posicionamento pode falhar por falta de unidade e/ou o roteamento não ser possível (conflitos na rede). Neste cenário, o número de partições será incrementado, e o algoritmo é novamente executado para buscar um novo escalonamento com mais partições.

Note que para adaptar a heurística ao uso de partições, duas variáveis foram criadas para armazenar a configuração atual e outra para armazenar a configuração máxima. A variável indica as UFs livres agora é um vetor, guardando por configuração qual UF estará ocupada e em qual está livre.

Os resultados dos testes experimentais mostram que apesar de guloso, em 10 de 16 casos o algoritmo não necessitou incrementar o número de partições. Dos outros 6 casos, um número maior de partições foi necessário em 4 casos, mas a solução foi encontrada e somente em 2 o roteamento falhou.

A Figura 3.22 mostra um trecho do pseudocódigo do *Modulo Scheduling*, onde a configuração do nó **y** é sempre acrescida de uma unidade em relação à configuraçao do nó x, mas respeitando o limite da configuração máxima para aquele algoritmo. Caso não seja possível rotear com determinado número de configurações, o número máximo de configurações é incrementado, e o algoritmo roda novamente desde o

```
1.  Se Y não alocado {
2.      uf_y = posicionamento [y]
3.      tempo[y] = escalonamento[y]
4.      configAnterior = configuracao[x]
5.      configAtual = (configuracao[x]+1) mod configMaxima
6.  }
```

Figura 3.22. Algoritmo de atualização das configurações

início.

3.3 EPR Mutiestágio com Restrição de Entradas

3.3.1 Algoritmo

Nas abordagens anteriores, através dos resultados obtidos, foi observado que o algoritmo muitas vezes abortava a execução por estar com alguma configuração cheia, e não poderiam ser efetuados mais nenhum posicionamento e roteamento naquela configuração. Na maioria das vezes, as entradas são responsáveis pelo acúmulo de nós na primeira configuração. Para resolver este problema, a ideia de restringir o número de entradas por configuração foi introduzida, espalhando a alocação nas configurações.

A implementação de restrição de entradas é simples. Foi necessário apenas estabelecer o número máximo de entradas por configuração. Feito isto, o estado da máquina responsável pela alocação de entradas passa a verificar se o número máximo já foi atingido. Quando chega ao limite, o algoritmo muda a configuração atual, e as próximas entradas são alocadas na próxima configuração.

A Figura 3.23 mostra o trecho do pseudocódigo que implementa estas funcionalidades.

Ao passar as próximas entradas para a próxima configuração, os nós ligados às entradas também são deslocados para a próxima partição temporal, como se pode ver na Figura 3.24.

A latência pode aumentar se o nó estiver no caminho crítico. Com relação do intervalo de inicialização e balanceamento, isto dependerá do grafo.

Os resultados dos nossos experimentos mostram que, com esta abordagem, o número de grafos que foram roteados com sucesso aumentou, e gerou uma redução do II, ou seja, melhorou o desempenho (reduzindo a vazão).

```
1.   ESTADO POSICIONA ENTRADA
2.   se (não é a primeira aresta)
3.        --máximoDeEntradas
4.
5.   ESTADO POSICIONA NÓ
6.   se (máximoDeEntradas == 0){
7.        máximoDeEntradas = MAX_ENT
8.        ++configuracaoInicial
9.        se (configuracaoInicial > confiMaxima){
10.           falha
11.       }
12.  }
```

Figura 3.23. Algoritmo de restrição de entradas

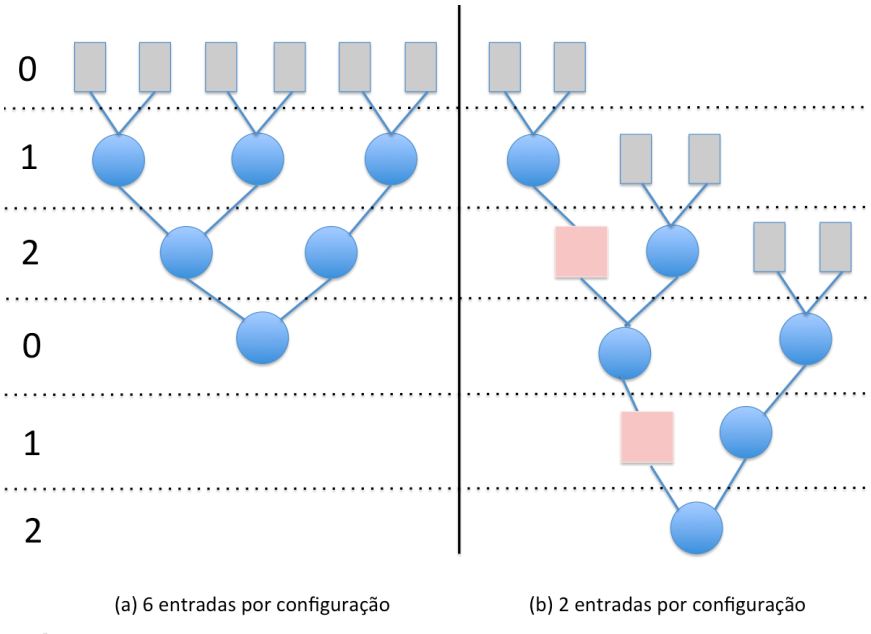

Figura 3.24. (a) Grafo onde as 6 entradas são permitidas na configuração atual. (b) Grafo onde apenas 2 entradas são permitidas por configuração

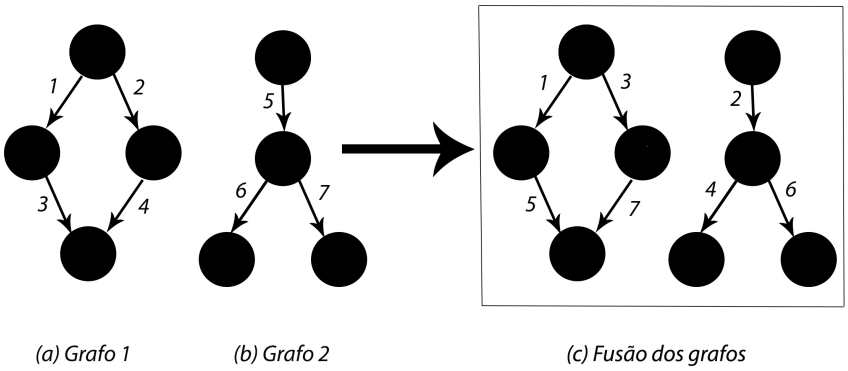

Figura 3.25. Fusão de dois grafos e a ordem em que as arestas são visitadas pelo algoritmo

3.4 Processamento Multitarefa

Muitos sistemas multimídias processam mais de uma aplicação simultaneamente, e isto requer mais poder de processamento. Tablets, celulares e *smart* TVs são exemplos atuais de dispositivos que executam várias funções simultaneamente, exigindo um processamento eficiente, juntamente com economia de energia e potência. No caso dos celulares, ainda podemos enfatizar a exigência cada vez maior de mais poder de processamento num dispositivo cada vez menor, o que limita ainda mais os projetistas de hardware.

Alguns trabalhos vem sido realizados a fim de suprir essa demanda como em [36].

Nesta dissertação, avaliamos o potencial de mapeamento de duas aplicações simultaneamente, fazendo uma mesclagem de duas aplicações diferentes, e processando as duas ao mesmo tempo numa arquitetura alvo, como mostra a figura 3.25.

Na Figura 3.25 (c), pode-se notar que os grafos das aplicações são disjuntos, podendo ser processados alternadamente ou um após o outro, desde que a ordem em largura dentro de cada componente conexo seja respeitada. Neste exemplo, as arestas são alternadas entre os grafos 1 e 2, gerando o mapeamento dos dois grafos ao mesmo tempo.

Com esta abordagem, obtivemos ganhos em termos de ocupação e ILP , chegando a um ILP de 38.2 e uma ocupação de 85% de arquitetura de 64 UF, onde três *benchmarks* foram mapeados ao mesmo tempo, com 218 operações no total. O

```
01. ENQUANTO HOUVE LIGAÇÕES
02.       lê x -> y
03.       se x_É_ENTRADA( )
04.           POSICIONA_ENTRADA( )
05.               se (UFs desta configuração estão cheias)
06.                   falha
07.               fim se
08.       POSICIONA_NÓ_Y( )
09.       senão
10.           se NÓ_Y_NÃO_POSICIONADO( )
11.           POSCIONA_Y( )
12.               se (UFs desta configuração estão cheias)
13.                       falha
14.               senão // y já está posicionado
15.                   se está balanceado
16.                           ROTEIA_X->Y( )
17.                   senão
18.                       enquanto estiver desbalanceado
19.                           INSERE_REGISTRO( )
20.                       fim enquanto
21.                       ROTEIA_X->Y( )
22.                   fim se
23.               fim se
24.           fim se
25.   fim enquanto
```

Figura 3.26. Pseudo-código da modelagem em software

intervalo de iniciação foi mínimo.

O algoritmo não muda, apenas deve ser observado quais entradas e saídas foram alocadas para cada aplicação.

3.5 Modelagem em Software

Esta seção descreve uma implementação em software da heurística. Essa implementação desconsidera as restrições de hardware levadas em conta nas implementações feitas com máquina de estado. A implementação permite comparar os tempos de execução entre as versões em hardware e a versão em software que pode ser executada em um processador *softcore* dentro do circuito reconfigurável.

O algoritmo também percorre todas as arestas em largura, como o EPR.

A figura 3.26 ilustra o pseudocódigo da modelagem em software usando a linguagem C.

Capítulo 4

Resultados

As heurísticas utilizadas neste trabalho foram validadas utilizando exemplos do conjunto do *benchmarks* para processamento multimídia disponíveis em [8].

4.1 Sem Partições Temporais

Para avaliar o algoritmo em sua primeira versão, cada *benchmark* foi escalonado e mapeado em uma arquitetura maior do que o número de operações totais do grafo. Isto foi feito com o intuito de tentar escalonar o grafo inteiro num II = 1, sem partições temporais.

Alguns casos como os *benchmarks matrix* e *smooth*, a arquitetura precisa ter 1024 UFs para poder mapear todo o grafo. Este tamanho é inviável devido ao tamanho e atraso na rede na implementação FPGA.

A tabela 4.1 exibe os resultados obtidos para arquiteturas maiores que o grafo. As colunas R e OP mostram o número de registros e de operadores de cada grafo. A coluna UF mostra o número de unidades de execução. A coluna Soft (seção 3.5) mostra a execução da versão em *software* para um Intel i7 1.8 Ghz com 4Gb de RAM. O tempo foi medido em microssegundos (μs) e o algoritmo foi executado 100.000 vezes para cada *benchmark*.

Esta tabela já mostra resultados positivos alcançados pelo algoritmo. A coluna *hard* mostra o tempo aproximado para a execução do algoritmo em *hardware*, que pode ser comparado com o tempo da versão em *software*. Este tempo foi medido em funcão do número de ciclos da máquina de estado. Foi considerado um clock de 100Mhz. Nota-se um ganho de até 10 vezes no tempo de execução do *Modulo Scheduling* em *hardware* em relação ao *software*.

Tabela 4.1. Arquitetura maior que Grafo

Nome	R	Op	UF	Soft μs	Hard μs	Ganho
arf	5	36	64	23,54	2,18	10,8
collapse	38	65	256	30,84	5,33	5,7
cosine1	8	82	256	26,82	5,19	5,1
cosine2	62	112	256	36,61	8,56	4,2
ewf	51	36	256	30,54	3,93	7,7
feedback	22	74	256	24,50	4,17	5,8
fir	26	66	256	24,71	3,94	6,2
fir2	21	56	256	23,93	3,25	7,3
grafo24	23	79	256	25,41	5,02	5,0
h2v2	56	67	256	30,43	5,18	5,8
interpolat	40	156	256	31,30	8,66	3,6
matrix	240	370	1024	233,24	31,40	7,4
matmul	48	132	256	43,04	8,31	5,1
motion	15	46	256	7,44	2,49	2.9
smooth	128	260	1024	172,60	17,11	10,1
write bmp	54	154	256	32,83	8,10	4,0

A máquina de estado da figura 3.16 foi utilizada como base para a medida de tempo. Isto é feito contando os ciclos gastos para processar as arestas do grafo que representa o laço. Cada ciclo corresponde a um estado da máquina. O total de ciclos foi multiplicado por 10 ns para obter o tempo apresentado na tabela 4.1 na coluna hard.

4.2 Pré-Leitura

Algumas otimizações foram implementadas, como a pré-leitura. Esta estratégia não passa pelo estado E0 da máquina de estado a cada novo par de nós, mas coloca a leitura das arestas nos estados finais da máquina, e então retorna ao estado inicial que passa a ser E1, como visto na seção 3.1.3.

Para avaliar os resultados dessa etapa, foram comparados os resultados de alguns *benchmarks* executados primeiramente sem o uso da pré-leitura, ou seja, tinham um estado a mais na máquina, com os resultados dos que foram executados utilizando a pré-leitura.

A tabela 4.2 mostra o resultado da otimização de pré-leitura. A alteração gera um ganho de 16 a 21% no tempo de execução da versão com pré-leitura, pois processamento de cada aresta agora consome 1 ciclo ou estado a menos.

Tabela 4.2. Pré-Leitura

Nome	Op	UE	Com Pré-Busca μs	Sem Pré-Busca μs	G
arf	36	64	1,8	2,18	1,21
feedback	74	256	3,46	4,17	1,21
fir2	56	256	2,7	3,25	1,20
h2v2	67	256	4,48	5,18	1,16
matrix	370	1024	27,09	31,4	1,16
write bmp	154	256	6,84	8,1	1,18

Na versão sem a otimização para gerar a configuração do grafo *arf*, são necessários 218 ciclos, ou seja 5,7 ciclos por operador. Com otimização da pré-leitura, esta média cai para 4 a 5 ciclos pro operador, gastando 180 ciclos como vemos na tabela 4.2.

4.3 Partições Temporais

Utilizamos partições temporais para avaliar esta versão do algoritmo (seção 3.2). Com esta abordagem, é possível testar grafos que possuem um número de operações maior do que o número de UFs da arquitetura alvo.

Foram utilizadas duas arquiteturas diferentes, uma com 64 UFs e outra com 256. A maioria dos grafos foi mapeadas na arquitetura de 64 unidades, porém, alguns deles necessitaram de um intervalo de iniciação maior do que o mínimo.

Na arquitetura com 256 unidades foram roteados os *benchmarks* restantes, e também mapeados aqueles que não geraram um II igual ao MII.

A tabela 4.3 supõem uma arquitetura com 64 UF. A rede de conexão é uma multiestágio com 1 estágio extra. A coluna OP+R mostra o total de operadores e registros para cada grafo. Quando este valor é maior que 64 é necessário mais de uma partição temporal. A coluna MIN mostra o mínimo de partições para cada grafo e a coluna II mostra o valor encontrado pelo algoritmo. Finalmente, o tempo de execução da versão em software e da versão em *hardware* são apresentados. A coluna G mostra o ganho de desempenho, que varia de 4,5 a 16,9 vezes. Para dois casos (*matrix* e *smooth*) devido a conflitos de roteamento, não foi possível escalonar o grafo.

A tabela 4.4 usa uma arquitetura maior com 256 UF e uma rede com 1 estágio extra. A maioria dos *benchmarks* é mapeado em uma partição e não são mostrados na tabela. Apenas são apresentados os exemplos maiores que ocupam mais de 1

Tabela 4.3. Com Partições Temporais, UF=64

Nome	Op+R	Min	II	Soft μs	Hard μs	G
arf	38	1	2	58,1	3,45	16,9
collapse	97	2	2	51,59	5,71	9,0
cosine1	74	2	2	41,42	5,33	7.7
cosine2	143	3	3	54,68	8,54	6.4
ewf	85	2	2	46,69	4,26	10,9
feedback	75	2	2	38,38	4,35	8,8
fir	70	2	2	38,81	4,05	9,5
fir2	61	1	1	43,02	3,66	11,7
grafo24	78	2	2	43,31	5,31	8,1
h2v2	107	2	2	47,77	5,56	8,5
interpolat	148	3	4	98,51	17,70	5,5
matrix	533	9	FAIL	0	0	0
matmul	156	3	4	77,24	17,15	4,5
motion	47	1	1	35,70	2,71	13,1
smooth	324	6	FAIL	0	0	0
write bmp	170	3	4	97,59	15,99	6,1

partição e os que não foram roteados com a arquitetura de 64 unidades. Os ganhos variam de 6 a 22 vezes da versão em *hardware* com relação a versão em software. Em alguns casos, como o grafo *matrix* que possui 533 operações, o algoritmo de roteamento encontra alguns conflitos de posicionamento e roteamento. Maiores detalhes sobre o roteamento estão descritos em [14]. Os ciclos gastos pela máquina de estado mesmo quando esta encontra um conflito também são somados ao número de ciclos total, o que reduz um pouco o desempenho. Em certos casos, como o exemplo da matriz, o algoritmo falha para II=3, então um novo escalonamento é realizado com II=4 que também falha. Finalmente para II=5, o roteamento é bem sucedido para todas as ligações e o escalonamento é realizado. Se a rede de interconexão for modificada para incluir mais uma camada de estágio extra, o algoritmo executa em um passo e gasta o número mínimo de partições temporais.

4.4 Restrição de Entradas

Esta versão do algoritmo mostra os resultados obtidos utilizando a técnica de restrição de entradas. Espalhando as operações pelas partições, foi possível aliviar a concentração de UFs utilizadas na partição inicial, evitando falha por falta de UFs disponíveis numa partição.

Tabela 4.4. Com Partições Temporais, UF=256

Nome	Op+R	Min	II	Soft μs	Hard μs	G
interpolate	148	1	1	183,41	9,34	19,6
matrix	533	3	5	540,95	86,77	6,2
matmul	156	1	1	204,01	9,41	21,6
smooth	324	2	2	293,87	31,44	9,3
write bmp	170	1	1	177,24	8,10	21,8

Foram considerados dois tamanhos CGRA a fim de analisar o algoritmo em diferentes condições. A primeira arquitetura alvo é composta por uma CGRA de 16 UFs, utilizando a rede *Crossbar* e tendo no máximo 4 entradas por configuração. A segunda tem 64 UFs e até 12 entradas por configuração. Em ambos os casos, foram consideradas unidades funcionais homogêneas, como a maioria das abordagens de *Modulo Scheduling* [39, 23, 37, 35].

A tabela 4.5 mostra os resultados obtidos para os *benchmarks* analizados nesta etapa. Grafos maiores como *matrix* e *smooth* não foram analisados, pois mesmo com estas otimizações, eles não são mapeados em arquiteturas de 16 e 64 UFs.

Na tabela 4.5, podemos ver o número de entradas e de operações do grafo, que estão representados nas colunas Ent e Op, respectivamente. Após o grafo ser balanceado e registros serem inseridos, o número total de operações passa a ser o da coluna OPR. O intervalo mínimo inicial (MII) do grafo e o II que foi alcançado pelo algoritmo são apresentadas na coluna Min e II, respectivamente. A coluna ILP mostra o nível de paralelismo de instruções obtido, e a coluna Occ descreve a ocupação média da CGRA.

Podemos ver nesta tabela o nível de paralelismo de instrução (ILP) atingido pelo algoritmo, que é calculada pelo número de operações do grafo divididas pelo II alcançado. O ILP médio é de 8,11. O pior caso é a referência h2v2 (parte central da compressão jpeg [8]). Este resultado é devido ao grande número de registros necessários para balancear os caminhos internos e também devido ao baixo grau de paralelismo (nós/caminho crítico) [8]

Também podemos ver a ocupação média da CGRA (OCC). A ocupação representa a porcentagem dos recursos utilizados durante a execução ($100 - OCC$ indica o percentual de recursos ociosos). Para os parâmetros avaliados, a ocupação média é de 76%, que é próximo da ocupação média nas arquiteturas DRESC e G-Minor[3]. O resultado desta ocupação não foi muito bom devido ao grande número de entradas. Quando o grafo apresenta uma grande quantidade de entradas (por exemplo,

Tabela 4.5. Resultados para Restrição de Entradas

16 UF, 4 Ent								
Bench	Ent	Op	OpR	Min	II	ILP	Occ	Tempo
arf	8	28	41	3	3	9.33	85%	2.47
cosine1	16	66	82	6	7	9.43	73%	14.5
ewf	2	34	85	6	6	5.67	89%	4.27
fir1	22	44	50	6	6	7.33	52%	3.18
fir2	16	40	49	4	4	10.0	77%	4.36
grafo24	24	55	77	6	6	9.17	80%	6.25
h2v2	16	51	103	7	10	5.1	64%	18.8
inter	48	108	172	11	12	9.00	90%	13.4
motion	14	32	49	4	4	8.00	77%	3.36
média						8.11	76%	7.8
64 UF, 12 Ent								
Bench	Ent	Op	OpR	Min	II	ILP	Occ	Tempo
cosine2	31	81	120	3	3	27.0	67%	7.84
matmul	24	108	144	3	3	36.0	75%	9.15
interpolate	48	108	148	4	4	27.0	58%	10.66
writebmp	38	116	167	4	4	29.0	65%	9.28
smooth	64	196	320	6	6	32.67	83%	18.17
média						30.3	69%	11.02

com 22 entradas *fir1*), é necessário dividir estas entradas em muitas configurações, o que resulta numa baixa ocupação.

Além disso, podemos observar que o intervalo de iniciação mínimo (MII) foi alcançado para todos os *benchmarks*. O ILP médio é de 30,3, indicando uma extração significativamente alta de paralelismo. Outro resultado importante foi a ocupação média de 69%. Este OCC também se deve ao aumento do número de unidades funcionais. O fato do grafo geralmente não possuir um número de nós múltiplo do tamanho da arquitetura pode acabar contribuindo para um ILP baixo, por exemplo, se a o grafo tem 129 nós e a arquitetura tem 64 recursos disponíveis, por 1 nó apenas o algoritmo terá que rodar com 3 configurações para suportar todo o grafo, acarretando assim numa baixa ocupação.

Analisando os resultados obtidos vemos que o algoritmo foi eficiente na utilização do recurso disponível, pois nota-se uma alta ocupação média do CGRA, chegando a ocupar 90% da arquitetura em alguns casos. Também foi minimizado o número de configurações necessárias para escalonar, mapear e rotear todo o grafo em tempo de execução. O nível de paralelismo alcançado também foi alto, demonstrando que o algoritmo é capaz de escalonar muitas operações em paralelo obtendo um bom desempenho.

Tabela 4.6. Resultados para Multitarefa

multithread, 64UF, 8 Ent								
Bench	Ent	Op	OpR	Min	II	ILP	Occ	Tempo
fir2+arf	24	68	92	3	3	22.7	48%	6.35
ewf+arf	10	62	122	2	3	20.7	64%	12.33
feed+elf	23	87	159	3	4	22.1	60%	14.7
arf+collap+cos1	31	153	218	4	4	38.2	85%	14.2
média						26.0	64%	11.9

4.5 Multitarefa

Nesta abordagem, o resultado da mesclagem de dois *benchmarks* foi medido, obedecendo a ordem em largura de cada um, gerando assim um novo grafo com os dois núcleos integrados.

A arquitetura utilizada tem 64 UFs e uma restrição de 8 entradas por configuração. Para os grafos multi-tarefas compostos por dois núcleos individuais: *fir2* & *arf* e *EWF* & *arf*, a execução concorrente atinge o mesmo tempo total de execução de série. No entanto, para dois casos de combinação multi-tarefas: *feedback* & *ewf* e *arf* & *collap* & *cos1*, a execução concorrente reduz o tempo de execução em 25 %. O ILP médio foi de 26, e a ocupação foi de 64,5%.

Os resultados apresentados indicam que há espaço para investir em novas soluções para acelerar a computação multi-tarefa a fim de melhorar a ocupação da arquitetura.

Capítulo 5

Conclusão

Este trabalho tem como objetivo propor uma heurística gulosa que é viável de implementar em tempo de execução para realizar o modulo scheduling de laço em arquiteturas reconfiguraveis de grao grosso (CGRA).

As principais características do algoritmo são:

- Algoritmo que respeita as restrições para modelagem com uma máquina de estados simples.
- Redução da complexidade do algoritmo e da implementação em comparação com trabalhos anteriores [15], onde o escalonamento era feito a parte utilizando as técnicas ASAP e ALAP para balancear o grafo. Esta dissertação mostra uma nova implementação em hardware em tempo de execução.
- Algoritmo melhorou a taxa de ocupação da arquitetura em relação a [15]. Foi possível mapear grafos de até 172 nós em arquiteturas pequenas de 16 UFs, gerando uma ocupação de 90% da CGRA. A ocupação média alcançada foi de cerca de 76%.
- Mapeamento possibilitou a execução de mais de uma aplicação ao mesmo tempo numa arquitetura CGRA.
- Este algoritmo apresentou um ganho de 2 a 3 ordens de grandeza em relação à proposta JIT apresentada em [15] e de 4 a 7 ordens de grandeza em relação algoritmos da literatura ([31, 38, 20, 3]) que são implementados para executar em tempo de compilação.

Uma consideração importante em relação a este algoritmo é que ele demonstra apenas a simulação do que seria o resultado em hardware, pois foi modelado em

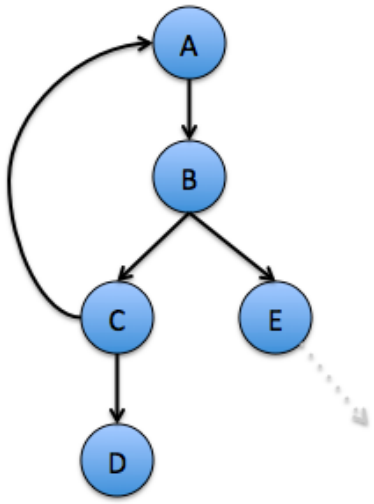

Figura 5.1. Loop Interno. Nó **C** se liga ao nó **A**

software utilizando a linguagem C. O trabalho de [19] apresenta a implementação do algoritmo em hardware. Já em [30], podemos ver os resultados obtidos para uma derivação deste trabalho, utilizando a heurística MSG.

5.1 Dependência entre iterações do laço

Dependência entre iterações, como ilustra a Figura 5.1, são casos onde um caminho fechado é encontrado no grafo de fluxo de dados. O algoritmo proposto neste trabalho não trata este tipo de situação, mas pode ser alterado como trabalho futuro..

No exemplo da Figura 5.1, o II mínimo seria 3, em função da dependência. A adaptação do algoritmo deve incluir mais um caso para tratar a aresta de realimentação entre os vertces C e A do exemplo.

5.2 Unidades Funcionais Heterogêneas

Como foi dito anteriormente, este trabalho faz uso das unidades funcionais homogêneas, ou seja, unidades que não se diferem entre si e podem realizar tanto multiplicações quanto somas, operações lógicas e armazenar registros. A utilização de unidades funcionais heterogêneas pode reduzir tanto a área da arquitetura quanto a potência necessária [38].

Para implementar as unidades heterogêneas nesta arquitetura, basta modificar o vetor *livre*, adicionando uma dimensão extra onde poderemos armazenar também

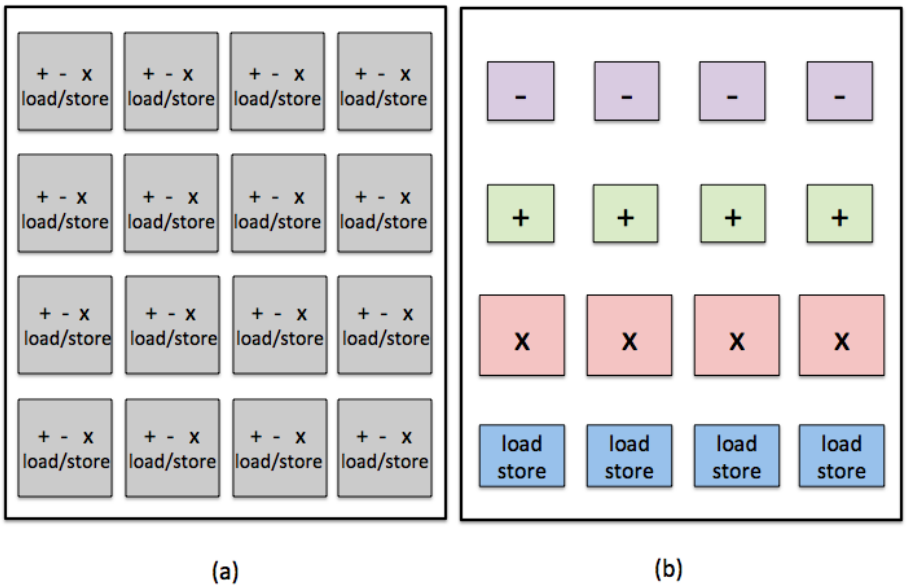

Figura 5.2. (a) Aruitetura CGRA com UFs homogêneas. (b) Arquitetura CGRA com UFs heterogêneas

o tipo de unidade funcional é aquela (subtrator, multiplicador, somador, etc.). Caso as unidades específicas para uma operação estejam ocupadas na configuração atual, o algoritmo irá buscar uma unidade da próxima configuração. O resto do algoritmo permanece sem modificações.

A Figura 5.2 mostra um exemplo de como seria uma arquitetura heterogênea em comparação com uma homogênea. Note que em (a) todas as UFs podem fazer as 4 operações em questão (adição, subtração, multiplicação e load/store) e, por isso, são mais complexas e ocupam mais espaço na arquitetura CGRA. Em (b) as UFs são heterogêneas, logo, as operações de adição e subtração que são mais simples necessitam de UFs menos complexas, sobrando mais espaço na arquitetura.

Referências Bibliográficas

[1] Brant, A. & Lemieux, G. (2012). Zuma: An open fpga overlay architecture. In *Proc. Field-Programmable Custom Computing Machines (FCCM)*, pp. 93 –96.

[2] Callahan, T. J. & Wawrzynek, J. (2000). Adapting software pipelining for reconfigurable computing. In *Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '00, pp. 57--64, New York, NY, USA. ACM.

[3] Chen, L. & Mitra, T. (2012). Graph minor approach for application mapping on cgras. In *Field Programmable Technology (FPT)*, pp. 285–292.

[4] Choi, K. (2011). Coarse-grained reconfigurable array: Architecture and application mapping. *IPSJ Transactions on System LSI Design Methodology*, 4:31–46.

[5] Colwell, R. P.; Nix, R. P.; O'Donnell, J. J.; Papworth, D. B. & Rodman, P. K. (1988). A vliw architecture for a trace scheduling compiler. In *Computers, IEEE Transactions, v. 37, n. 8, p. 967-979.*

[6] Coole, J. & Stitt, G. (2010). Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing. In *Proc. International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pp. 13 –22.

[7] Devi, V. V. & Kumar, B. (2012). Optimal performance mapping on reconfigurable architecture for multimedia applications. *European Journal of Scientific Research*, 80(2):69–175.

[8] ExPRESS (2013). Electrical computer engineering dep., ucsb, usa. http://express.ece.ucsb.edu/benchmark/.

[9] Ferreira, R.; Bueno, C.; Laure, M.; Pereira, M. & Carro, L. (2011a). A dynamic reconfigurable super vliw architecture for a fault tolerant nanoscale design. In *In 4TH HiPEAC Workshop on Reconfigurable Computing.*

[10] Ferreira, R.; Damiany, A.; Vendramini, J.; Teixeira, T. & Cardoso, J. (2009a). On simplifying placement and routing by extending coarse-grained reconfigurable arrays with omega networks. In Becker, J.; Woods, R.; Athanas, P. & Morgan, F., editores, *Reconfigurable Computing - Architectures, Tools and Applications (ARC09)*, volume 5453 of *Lecture Notes in Computer Science*, pp. 145–156. Springer Berlin Heidelberg.

[11] Ferreira, R.; Garcia, A.; Teixeira, T. & Cardoso, J. (2007). A polynomial placement algorithm for data driven coarse-grained reconfigurable architectures. In *VLSI, 2007. ISVLSI '07. IEEE Computer Society Annual Symposium on*, pp. 61–66.

[12] Ferreira, R.; Laure, M.; Beck, A.; Lo, T.; Rutzig, M. & Carro, L. (2009b). A low cost and adaptable routing network for reconfigurable systems. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on Reconfigurable Architectures Workshop (RAW)*, pp. 1–8.

[13] Ferreira, R.; Laure, M.; Rutzig, M.; Beck, A. & Carro, L. (2008a). Reducing interconnection cost in coarse-grained dynamic computing through multistage network. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pp. 47–52.

[14] Ferreira, R.; Vendramini, J. & Nacif, M. (2011b). Dynamic reconfigurable multicast interconnections by using radix-4 multistage networks in fpga. In *Industrial Informatics (INDIN), 2011 9th IEEE International Conference on*, pp. 810 –815.

[15] Ferreira, R.; Vendramini, J. G.; Mucida, L.; Pereira, M. M. & Carro, L. (2011c). An fpga-based heterogeneous coarse-grained dynamically reconfigurable architecture. In *Proc. Compilers, architectures and synthesis for embedded systems (CASES)*.

[16] Ferreira, R. S.; Cardoso, J. M.; Damiany, A.; Vendramini, J. & Teixeira, T. (2008b). Fast placement and routing by extending coarse-grained reconfigurable arrays with omega networks. In *Journal of Systems Architecture, 57(8), 761-777.*

[17] Ferreira, R. S.; Duarte, V.; Meirelles, W.; Pereira, M.; Carro, L. & WONG, S. (2013a). A just-in-time modulo scheduling for virtual coarse-grained reconfigurable architectures. In *SAMOS XIII, 2013, Samos. International Conference on Embedded Computer Systems: Architectures, Modelling and Simulations.*

[18] Ferreira, R. S.; Rocha, L.; Santos, A. G.; Nacif, J. A.; Carro, L. & Wong, S. (2013b). A run-time graph-based polynomial placement and routing algorithm for virtual fpgas. In *23rd International Conference on Field Programmable Logic and Applications, 2013, Porto. International Conference on Field Programmable Logic and Applications (FPL)*.

[19] Filho, M.; Denver, W. & Ferreira, R. (2012). A hardware-assisted modulo scheduling, placement and routing algorithm for stream computing in coarse-grained reconfigurable architectures. In *Student Forum / SBCCI*.

[20] Hamzeh, M.; Shrivastava, A. & Vrudhula, S. (2012). Epimap: using epimorphism to map applications on cgras. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pp. 1284--1291, New York, NY, USA. ACM.

[21] Hartenstein, R. (2001a). Coarse grain reconfigurable architectures. In *Asia and South Pacific Design Automation Conference, 2001. Proceedings of the ASP DAC 2001.*

[22] Hartenstein, R. (2001b). A decade of reconfigurable computing: a visionary retrospective. In *DATE Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings*, pp. 642 –649.

[23] Hatanaka, A. & Bagherzadeh, N. (2007). A modulo scheduling algorithm for a coarse-grain reconfigurable array template. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pp. 1 –8.

[24] Hoo, C. H. & Kumar, A. (2012). An area-efficient partially reconfigurable crossbar switch with low reconfiguration delay. In *Proc. Field Programmable Logic and Applications (FPL)*.

[25] Kim, Y.; Lee, J.; Shrivastava, A. & Paek, Y. (2011a). Memory access optimization in compilation for coarse-grained reconfigurable architectures. *ACM Trans. Des. Autom. Electron. Syst.*, 16(4):42:1--42:27.

[26] Kim, Y.; Lee, J.; Shrivastava, A.; Yoon, J.; Cho, D. & Paek, Y. (2011b). High throughput data mapping for coarse-grained reconfigurable architectures. *CAD of Int. Circuits and Systems, IEEE Trans. on*, 30(11):1599 –1609.

[27] Kim, Y.; Park, I.; Choi, K. & Paek, Y. (2006). Power-conscious configuration cache structure and code mapping for coarse-grained reconfigurable architecture. In *Proceedings of the 2006 international symposium on Low power electronics and design*, ISLPED '06, pp. 310--315, New York, NY, USA. ACM.

[28] Lam, M. (1988). Software pipelining: An effective scheduling technique for vliw machines. In *ACM Sigplan Notices. Vol. 23. No. 7. ACM.*

[29] Lee, G.; Choi, K. & Dutt, N. (2011). Mapping multi-domain applications onto coarse-grained reconfigurable architectures. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(5):637 –650.

[30] Lopes, V. (2013). Heurística polinomial para escalonamento de loops em arquiteturas reconfiguráveis de grão grosso. In *Dissertação de mestrado, Departamento de Informatica, UFV.*

[31] Mei, B.; Vernalde, S.; Verkest, D.; De Man, H. & Lauwereins, R. (2002). Dresc: a retargetable compiler for coarse-grained reconfigurable architectures. In *Proc. Field Programmable Technology (FPT)*, pp. 166 – 173.

[32] Mei, B.; Vernalde, S.; Verkest, D.; Man, H. D. & Lauwereins, R. (2003). Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, p. 10296, Washington, DC, USA. IEEE Computer Society.

[33] Moore, G. E. (1965). Cramming more components onto integrated circuits. In *Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp. 114 ff. Solid-State Circuits Newsletter, IEEE, 11(5), 33-35.*

[34] Mucida, L.; Lopes, V.; Meireles, W. & Ferreira, R. (2012). Problem oriented approach to hardware-assisted algorithm design in c: A case study for scheduling, placement and routing. In *Simpósio em Sistemas Computacionais (WSCAD-SSC).*

[35] Oh, T.; Egger, B.; Park, H. & Mahlke, S. (2009). Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures. In *Proc. Languages, Compilers and Tools for Embedded Systems (LCTES)*, pp. 21--30.

[36] Panda, R.; Wood, A.; McVicar, N.; Ebeling, C. & Hauck, S. (2012). Extending course-grained reconfigurable arrays with multi-kernel dataflow. In *The Second Workshop on the Intersections of Computer Architecture and Reconfigurable Logic.*

[37] Park, H.; Fan, K.; Kudlur, M. & Mahlke, S. (2006). Modulo graph embedding: mapping applications onto coarse-grained reconfigurable architectures. In *Proc. CASES*, pp. 136--146.

[38] Park, H.; Fan, K.; Mahlke, S. A.; Oh, T.; Kim, H. & Kim, H.-s. (2008). Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pp. 166--176, New York, NY, USA. ACM.

[39] Rau, B. R. (1994). Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proc. Proceedings of the 27th annual international symposium on Microarchitecture (MICRO)*, pp. 63--74.

[40] Theodoridis, G.; Soudris, D. & Vassiliadis, S. (2007). A survey of coarse-grain reconfigurable architectures and cad tools. In *Proc. Field-Programmable Custom Computing Machines (FCCM)*, pp. 89--149.

[41] Trends, T. D. (2012). Big data to grow 500 percent by 2015 and 2 companies are uniquely positioned to benefit. http://seekingalpha.com/article/458361-big-data-to-grow-500-by-2015-and-2-companies-are-uniquely-positioned-to-benefit.

[42] Vendramini, J. & Ferreira, R. (2010). Parallel routing algorithm for extra level omega networks on reconfigurable systems. In *Simpósio em Sistemas Computacionais (WSCAD-SCC)*.