Universidade Federal do Piauí
Centro de Ciências da Natureza
Programa de Pós-Graduação em Ciência da Computação

RENOIR - Uma Ferramenta para Geração de Configurações utilizando o algoritmo de *Modulo Scheduling*

Lucas Fernandes Ribeiro

Número de Ordem PPGCC: M001

Teresina-PI, Agosto de 2019

Lucas Fernandes Ribeiro

RENOIR - Uma Ferramenta para Geração de Configurações utilizando o algoritmo de *Modulo Scheduling*

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da UFPI (área de concentração: Sistemas de Computação), como parte dos requisitos necessários para a obtenção do Título de Mestre em Ciência da Computação.

Universidade Federal do Piauí – UFPI

Centro de Ciências da Natureza

Programa de Pós-Graduação em Ciência da Computação

Orientador: Ivan Saraiva Silva

Teresina-PI

Agosto de 2019

Lucas Fernandes Ribeiro

RENOIR - Uma Ferramenta para Geração de Configurações utilizando o algoritmo de *Modulo Scheduling*/ Lucas Fernandes Ribeiro. – Teresina-PI, Agosto de 2019-

49 p. : il. (algumas color.) ; 30 cm.

Orientador: Ivan Saraiva Silva

Dissertação (Mestrado) – Universidade Federal do Piauí – UFPI
Centro de Ciências da Natureza
Programa de Pós-Graduação em Ciência da Computação, Agosto de 2019.

1. Software pipelining. 2. Modulo scheduling. I. Ivan Saraiva Silva. II. Universidade Federal do Piauí. III. Centro de Ciências da Natureza. IV. RENOIR - Uma Ferramenta para Geração de Configurações utilizando o algoritmo de *Modulo Scheduling*

CDU 02:141:005.7

"RENOIR - Uma Ferramenta para Geração de Configurações Utilizando o Algoritmo de Módulo Scheduling"

LUCAS FERNANDES RIBEIRO

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências da Natureza da Universidade Federal do Piauí, como parte integrante dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Aprovada por:

Prof. Ivan Saraiva Silva
(Presidente da Banca Examinadora)

Prof. Sílvio Roberto Fernandes de Araújo
(Examinador Externo à Instituição)

Prof. Kelson Rômulo Teixeira Soares
(Examinador Interno)

Prof. Raimundo Santos Moura
(Examinador Interno)

Teresina, 05 de agosto de 2019

Aos meus pais João Pedro e Zilmar Fernandes,
por sempre estarem comigo em todos os momentos.

Agradecimentos

Agradeço aos meus pais, João Pedro e Zilmar, que com muito carinho e apoio, não mediram esforços para que eu chegasse até esta etapa de minha vida. E ainda, ao meu irmão, Mateus.

À minha namorada, Juliana, por toda a paciência e carinho comigo, por estar ao meu lado durante toda essa jornada.

Agradeço ao meu orientador, Ivan Saraiva, por todos os conselhos, pela paciência e ajuda nesse período.

Aos meus amigos desde a graduação, que fizeram esse processo muito mais divertido: Valeska, Renato, Felipe, Luis Henrique, Rodolfo, Natanael e Bruno.

Aos meus amigos de laboratório: Laysson, Ramon, Jônatas, Jônatas (calouro), Thiago, Eugênio e João Pedro. E principalmente ao Juninho, um grande amigo que me apoiou e ajudou durante todo esse processo, compartilhando conhecimento e amizade.

Finalmente, aos professores por todo o conhecimento compartilhado.

"O que não sabe é um ignorante, mas o que sabe e não diz nada é um criminoso.

Resumo

No momento atual, com o aumento na complexidade das aplicações, a quantidade de dados gerados vem crescendo rapidamente, exigindo-se um desempenho cada vez maior dos processadores. Pra lidar com essa crescente, arquiteturas reconfiguráveis (AR) surgem como uma atrativa solução. Ainda em meio ao aumento na quantidade de dados gerados e na complexidade das aplicações, nota-se que os laços de repetição presentes em algumas dessas aplicações são responsáveis por até 71% do tempo de execução do código. Otimizando-se esse tempo, é possível obter um ganho no tempo total de execução da aplicação. Esse ganho de desempenho pode ser obtido com uso de *software pipelining*. Este trabalho propõe o uso da técnica de *software pipelining* utilizando *modulo scheduling* em *software* para uma arquitetura reconfigurável de grão grosso. O algoritmo proposto foi implementado em um compilador, que recebe como entrada uma aplicação desenvolvida na linguagem de programação *Go* e gera como saída o conjunto de instruções *assembly* MIPS. Para análise de resultados, cinco aplicações foram desenvolvidas. Os resultados obtidos mostram um bom mapeamento alcançado e que o paralelismo dos laços mais internos em algumas aplicações superam os 70% de ganho obtido.

Palavras-chaves: Software pipelining. Modulo scheduling. Arquitetura reconfigurável. Laço.

Abstract

Nowadays, with the increase in the complexity of the applications, the amount of data generated is growing fastly, requiring an ever-increasing performance of the processors. To deal with that, reconfigurable (AR) architectures arise as an attractive solution. Still in the midst of the increase in the amount of data generated complexity of the applications, it is noted that the loops present in some of these applications are responsible for up to 71% of the code execution time. Optimizing this execution time, it is possible to obtain a gain in the total execution time of the application, this gain performance can be obtained with the use of pipelining software. This work proposes the use of the pipelining software technique using modulo scheduling in software for a coarse grained reconfigurable architecture. The proposed algorithm was implemented in a compiler, which receives an application developed in the programming language Go and generates as output the MIPS assembly instruction set. For analysis of results, five applications were developed.The obtained results show a good mapping achieved and that the ILP of the inner most loops in some applications exceed in 70% the gain obtained.

Keywords: Software pipelining. Modulo scheduling. Reconfigurable architectures. Loop.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ASIC	*Application Specific Integrated Circuits*
AR	Arquitetura Reconfigurável
BR	Banco de Registradores
CGRA	*Coarse-Grained Reconfigurable Architecture*
DFG	*Data-flow Graph*
EPR	Escalonamento, posicionamento, roteamento
FPGA	*Field Programmable Gate Array*
FSM	*Finite State Machine*
GPP	*Genral-Purpose Processors*
II	Intervalo de Iniciação
ILP	*Instruction Level Parallelism*
LLVM	*Low Level Virtual Machine*
LW	*Load Word*
MII	Mínimo Intervalo de Iniciação
PE	Elemento de Processamento
RENOIR	*Configuration Generation Tool Using Modulo Scheduling Algorithm*
SW	*Store Word*
VLIW	*Very Long Instruction Word*

Sumário

1 Introdução

Devido ao aumento na complexidade das aplicações, a quantidade de dados gerados vem crescendo, exigindo-se um desempenho cada vez maior dos processadores. Algumas abordagens existentes tentam solucionar esse problema, cada uma com suas vantagens. Na primeira abordagem, os processadores de propósito geral (GPP - *General-Purpose Processors*) são utilizados para a execução de qualquer aplicação, por serem flexíveis o suficiente. Na segunda abordagem, os circuitos integrados de aplicação específica (ASIC - *Application Specific Integrated Circuits*) são utilizados na execução de aplicações específicas, pois possuem estruturas de *hardware* próprias para a aplicação desenvolvida, obtendo assim um alto desempenho. No entanto, demandam um longo ciclo de tempo para desenvolvimento, gerando um alto custo, além de não poderem ser reprogramados, o que torna incompatível com o mercado dinâmico, no qual as aplicações estão em constante adaptação.

Para lidar com essa crescente complexidade das aplicações, arquiteturas reconfiguráveis (AR) surgem como uma atrativa solução, unindo a flexibilidade dos GPPs com o alto desempenho dos ASICs. A computação reconfigurável ao mesmo tempo em que pode explorar o nível de paralelismo (ILP - *Instruction Level Parallelism*) (WALL, 1991) (XU; ALBONESI, 1999) disponível nas aplicações, também pode acelerar sequências de instruções dependentes de dados, que é a principal vantagem quando comparada com arquiteturas tradicionais (VENKATARAMANI et al., 2001a; STITT; VAHID, 2002).

Ainda em meio a esse aumento na quantidade de dados gerados e na complexidade das aplicações, no trabalho proposto por (PARK; PARK; MAHLKE, 2009) foram analisadas três aplicações multimídias importantes (*3D graphics rendering, AOC decoder* e *H.264 decoder*), a partir dessa análise foi extraída a quantidade total de laços e a quantidade de laços que podem ser paralelizáveis. Esse resultado pode ser visto na Tabela 1.

Tabela 1 – Total de laços e laços paralelizáveis. Extraídos de (PARK; PARK; MAHLKE, 2009)

Aplicação	**Total de laços**	**Laços paralelizáveis**
3D	102	36
AAC	260	83
H.264	269	81

A partir desses laços paralelizáveis, é possível obter um ganho de desempenho com o uso de *software pipelining* (CALLAHAN; WAWRZYNEK, 2000; HATANAKA; BAGHERZADEH, 2007). A técnica de *software pipelining* (ALLAN et al., 1995) proporciona a oportunidade de explorar o paralelismo de instruções presentes nos laços internos

de computação intensiva das aplicações. Essa técnica consiste em sobrepor a execução de instruções de múltiplas iterações de um laço, em que o objetivo é alcançar uma taxa mínima para o intervalo entre o início das diferentes iterações. Nesse caso, uma iteração pode iniciar antes mesmo do fim da iteração anterior. Uma das abordagens na utilização do *software pipelining* é chamada de *modulo scheduling* (RAU, 1994), que consiste, basicamente, em sucessivas tentativas de escalonar o laço na arquitetura alvo.

Ainda em (PARK; PARK; MAHLKE, 2009), foi medido o tempo de execução gasto nos laços que podem ser paralelizáveis utilizando *software pipelining* (SWP) e no restante do código. A Figura 1 mostra a distribuição de tempo de execução gasto nas regiões paralelizáveis e no resto da aplicação.

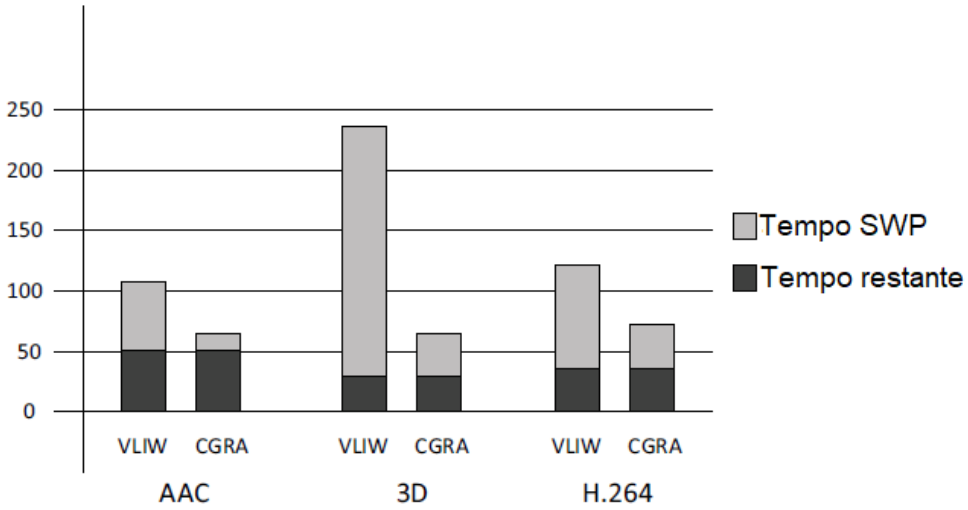

Figura 1 – Tempo de execução gasto nos laços com *software pipelining* e no restante da aplicação. A cor cinza é o tempo gasto em laços paralelizáveis e a cor preta é o tempo gasto na execução do restante da aplicação. Extraído de (PARK; PARK; MAHLKE, 2009).

Observado a imagem, a barra da esquerda de cada aplicação representa o tempo de execução quando apenas um processador VLIW (*Very Long Instruction Word*) é utilizado para toda a aplicação, sem a utilização de *software pipeling* (Tempo SWP). A barra da direita se refere ao tempo de execução considerando que as regiões paralelizáveis foram executadas com *software pipelining* em CGRA (*Coarse-Grained Reconfigurable Architecture*).

Portanto, analisando essas aplicações, em média 35% dos laços são paralelizáveis e que os laços presentes são responsáveis por até 71% do tempo de execução do código. Assim, otimizando-se o tempo de execução do laço, é possível obter um ganho no tempo total de execução da aplicação.

Neste trabalho é proposto a implementação e funcionamento da técnica de *software pipepiling* utilizando *modulo scheduling* implementado em *software* para geração de

configuração para uma arquitetura reconfigurável de grão grosso (CGRA - *Coarse-Grained Reconfigurable Architecture*), juntamente com a estrutura do sistema reconfigurável utilizado. O objetivo principal é mapear os laços presentes nas aplicações, gerando uma melhor configuração das instruções a serem executadas na arquitetura alvo, com os objetivos específicos de melhorar a qualidade dessa configuração e avaliar o paralelismo obtido a partir da utilização do algoritmo.

Este trabalho está organizado da seguinte forma :

- **Fundamentação Teórica:** neste capítulo serão apresentados os principais conceitos relacionados ao tema proposto pelo trabalho, para um melhor entendimento do escopo do trabalho;
- **Estado da arte**: neste capítulo, são apresentados alguns dos principais trabalhos na literatura sobre os assuntos deste trabalho: arquiteturas reconfiguráveis, *software pipelininge* e *modulo scheduling*;
- **Sistema proposto:** aqui será apresentado a proposta do trabalho, demonstrando sua implementação e funcionamento;
- **Resultados:** neste capítulo, os resultados encontrados serão demonstrados e analisados;
- **Conclusão e Trabalhos Futuros:** por último, a conclusão e os trabalhos futuros serão apresentados.

2 Fundamentação Teórica

Este capítulo apresenta o referencial teórico deste trabalho, apresentando conceitos introdutórios sobre arquiteturas reconfiguráveis, configuração em arquiteturas reconfiguráveis e *modulo scheduling*.

2.1 Arquiteturas Reconfiguráveis

O objetivo deste trabalho não é propor uma arquitetura reconfigurável (AR), porém é necessário fazer uma breve introdução, tendo em vista que será utilizada uma AR como arquitetura alvo do trabalho.

Arquiteturas reconfiguráveis de grão-grosso, (do inglês: *Coarse-Grained Reconfigurable Architectures* - CGRAs), surgiram com o principal objetivo de reduzir a complexidade e o tempo de configuração, posicionamento e roteamento das arquiteturas de grão-fino (do inglês: *Field Programmable Gate Array* - FPGA). As CGRAs são sistemas integrados que permitem customização da unidade de *hardware* para satisfazer os requisitos computacionais de diferentes aplicações (SINGH et al., 2000). Todas as CGRAs possuem elementos de processamentos (do inglês: *Processing Elements* - PEs), onde as operações são realizadas; uma rede de interconexão, que é responsável pela comunicação entre os elementos de processamento dentro da arquitetura; e, por último, uma memória para a configuração da arquitetura. As ARs podem variar com relação aos tipos de PEs, tipos de rede e granularidade.

- **Elementos de processamento**: Se forem homogêneos, todos os PEs podem realizar qualquer tipo de operação de um determinado grupo. Se forem heterogêneos, os PEs são divididos em grupos, onde cada grupo realiza um conjunto de operações diferentes.
- **Rede de interconexão**: Existem várias formas de interconexão entre os PEs. As mais comuns podem ser vistas na Figura 2.
  - Rede *Crossbar*: permitem qualquer permutação na conexão das entradas e nas saídas, contudo possui o custo $O(n^2)$, para n entradas/saídas.
  - Malha Simples: normalmente a escolha mais popular entre as CGRAs, a comunicação entre as unidades funcionais é feita vizinho-a-vizinho e possui uma alta escalabilidade.
  - Malha *Plus*: possui uma interconexão em malha melhorada, com novas conexões entre as unidades funcionais, melhorando a capacidade de roteamento.

– Agrupamento: também chamada de arquitetura em *clusters*, são arquiteturas híbridas de malhas divididas em *clusters*, e cada unidade funcional de fronteira conecta-se ao seu vizinho de agrupamento.

– Linha: a computação flui linha por linha, tendo as conexões de unidades funcionais feitas nas redes entre as linhas.

– Multiestágio: possui o custo $O(nlog(n))$, assim como a rede *crossbar*, possibilita a conexão entre quaisquer unidades funcionais, com a possibilidade de acontecer conflitos na rede quando há a necessidade de se usar um caminho que já está sendo utilizado na rede.

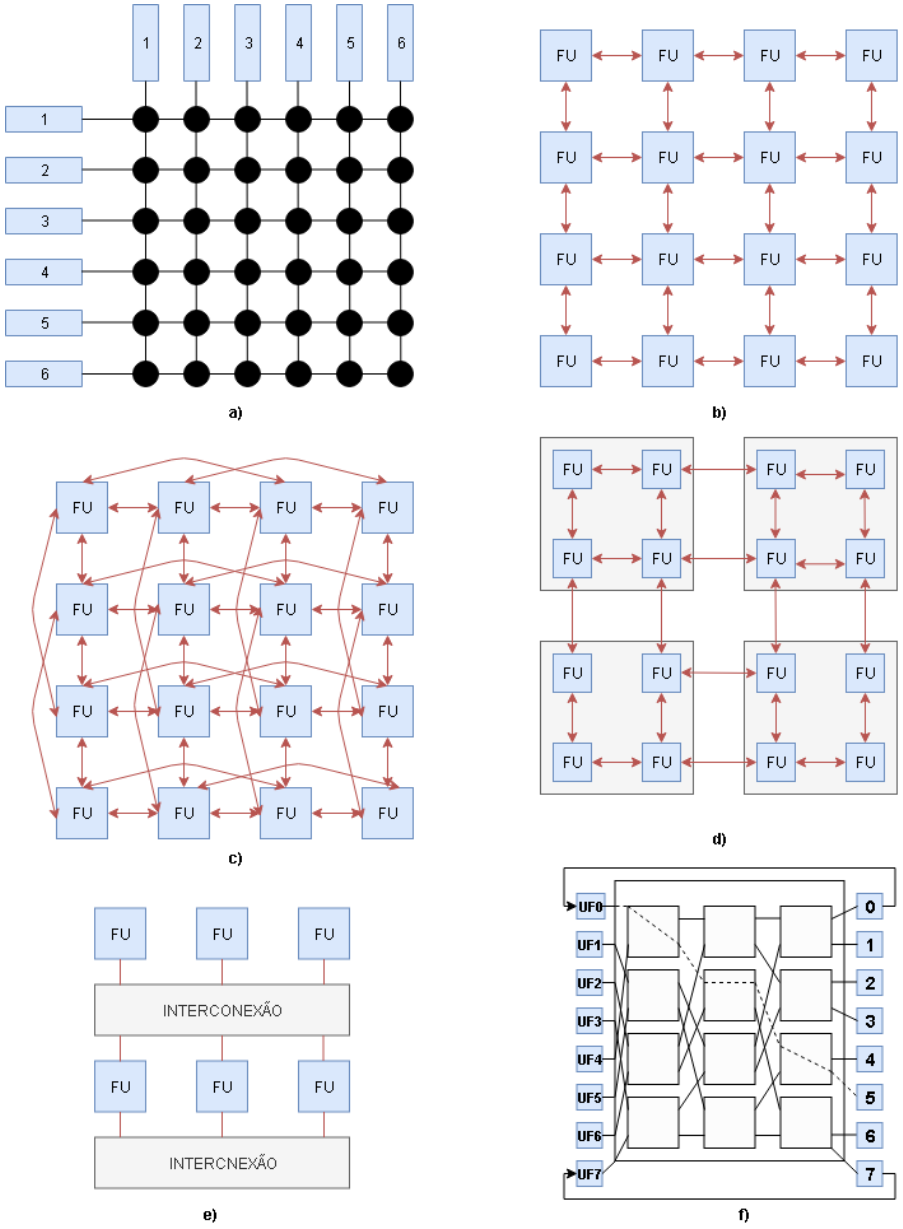

Figura 2 – Exemplos de redes de interconexão. a) Rede *Crossbar*, b) Malha Simples, c) Malha *Plus*, d)Agrupamento, e)Linha, f)Multiestágio

- **Granularidade**: A granularidade é determinada pelo tamanho do dado que pode ser operado pelos PEs. Essa granularidade influencia diretamente no tamanho da memória de configuração. Em FPGAs, as operações são feitas em níveis de *bits*, como em (HAUSER; WAWRZYNEK, 1997; YE et al., 2000). Nessas arquiteturas, as unidades funcionais são constituídas por componentes de *hardware* cuja configuração geram operadores a nível do *bit*. Já em CGRAs, as operações são feitas no nível de palavras, possuindo componentes de *hardware* mais complexos em suas unidades funcionais.

2.2 Configuração de Arquiteturas Reconfiguráveis

De forma simples, computação reconfigurável é o processo de melhor explorar o potencial do *hardware* reconfigurável (HARTENSTEIN, 2001). Pode ser dividida em dois tipos de configuração: estática e dinâmica (SANCHEZ et al., 1999). Na configuração estática, a configuração é carregada apenas uma vez e depois não sofre mudanças durante sua execução. Possui dois objetivos principais: 1) aumento de performance, como a velocidade de execução e 2) otimização da utilização de recursos, como o consumo de energia.

Por outro lado, na configuração dinâmica, a configuração pode ser alterada durante a execução da tarefa, o que permite que vários segmentos da aplicação sejam mapeados na arquitetura. Assim, mais recursos poderão ser utilizados, resultando em uma maior flexibilidade.

O mecanismo de reconfiguração define como diversas configurações podem ser carregadas na AR. Na reconfiguração estática, as configurações são geradas normalmente por um compilador, que identifica as partes da aplicação que podem ser aceleradas pela AR. Além disso, possui a característica de só poder ser carregada no sistema quando a arquitetura estiver inativa. Assim, todo o mecanismo que compõe a AR é reconfigurado antes de passar a executar uma nova tarefa. Já na reconfiguração dinâmica, a reconfiguração pode ser feita ainda com o sistema em operação, assim é possível gerar configurações em tempo de execução e carregar uma nova configuração na AR.

2.3 Modulo Scheduling

A abordagem do *modulo scheduling* (MS) surge da técnica de *software pipelining*. Essa técnica de *software pipelining* consiste em paralelizar a execução de laços, sobrepondo iterações do laço sem ter que esperar as iterações anteriores terminarem, executando várias iterações simultaneamente. Suponhamos que a Figura 3a) é um laço composto por 3 instruções, que será executado de forma sequencial. Utilizando a técnica de *software*

pipelining neste laço, as iterações serão executadas em paralelo, como mostra a Figura 3b). O *modulo scheduling* é um dos recursos utilizados para aumentar o paralelismo a nível de instrução (ILP) em laços de computação intensiva. A sobreposição de diferentes iterações do laço deve garantir a não violação das dependências internas e externas do grafo, além de evitar restrições de recursos.

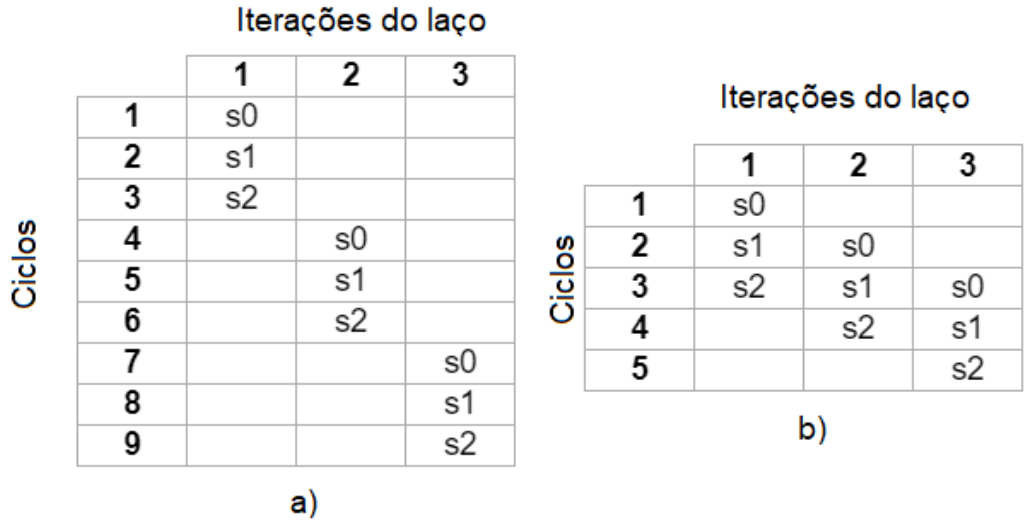

Figura 3 – a) Execução sequencial do algoritmo, b) Execução do algoritmo com *software pipelining*.

As dependências internas são aquelas em que os nós dentro da mesma iteração são dependentes entre si e a ordem deve ser respeitada. Já as dependências externas são aquelas em que um ou mais nós de uma determinada iteração dependem do resultado da interação anterior, assim, as duas não podem ser sobrepostas e executadas ao mesmo tempo.

As restrições de recursos caracterizam-se pelas restrições da arquitetura em si, como a quantidade de elementos de processamento ou a disponibilidade da interconexão. Um escalonamento válido é aquele em que não há nenhum conflito no uso de recursos entre as operações e nenhuma dependência interna ou externa da iteração é violada.

Entre o início de uma iteração e outra há um intervalo, que é chamado de intervalo de iniciação (II). Um algoritmo de MS sempre inicia seu escalonamento calculando o II inicial, chamado de mínimo II (MII), que é calculado com base nas restrições de recursos da arquitetura. Caso um escalonamento válido não seja obtido com o MII, esse intervalo é incrementado e um novo escalonamento é realizado, com um novo II.

Na Figura 4, a arquitetura demonstrada na Figura 4a) executará o grafo representado na Figura 4b) utilizando *modulo scheduling*. O primeiro passo é calcular o mínimo intervalo de iniciação (MII) (esse cálculo será apresentado em seções posteriores), representado por (II) na Figura 4c). O objetivo aqui é demonstrar a execução somente do corpo do

laço, também denominado de *Kernel*, que é onde as operações serão executadas em paralelo ocupando os PEs disponíveis. As operações do laço que iniciam o processo de paralelismo antes da formação do *kernel* são denominadas prólogo e as operações que sucedem o *kernel* formam o epílogo. Na Figura 4d), é representado quantas partições da arquitetura são necessárias para executar tais instruções, duas partições devem ser utilizadas. Na primeira partição, a instrução D é executada e na partição seguinte, as intruções A, B e C são executadas, assim como mostra a Figura 4e).

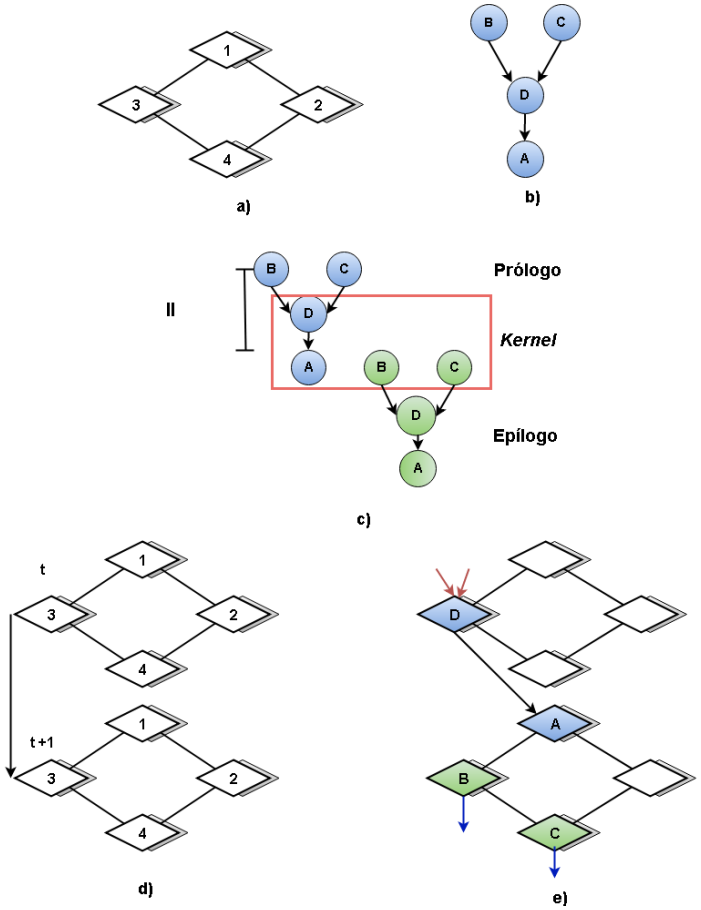

Figura 4 – Exemplos de redes de um mapeamento na arquitetura. a) Arquitetura alvo, b) Grafo inicial, c) Grafo com *modulo scheduling*, d) Partições necessárias, e) Mapeamento do grafo na arquitetura

O *modulo scheduling* em CGRAs é caracterizado como a combinação de três subproblemas: escalonamento, posicionamento e roteamento (EPR).

- **Escalonamento**: etapa onde são identificadas as dependências entre as operações e pela atribuição do tempo no qual as operações são executadas;

- **Posicionamento**: etapa em que é determinado em qual elemento de processamento serão alocadas as operações;

- **Roteamento**: etapa responsável pela conexão das operações mapeadas, gerando uma cadeia de operações conforme a sequência de dependências definida pela aresta do grafo de fluxo de dados.

Para exemplificar esses três subproblemas, considere o grafo presente na Figura 5a). O objetivo é mapeá-lo em uma CGRA composta por 4 PEs, como mostra a Figura 5b). Para melhor visualização, os PEs da CGRA serão mostrados em uma forma linear, como na Figura 5c).

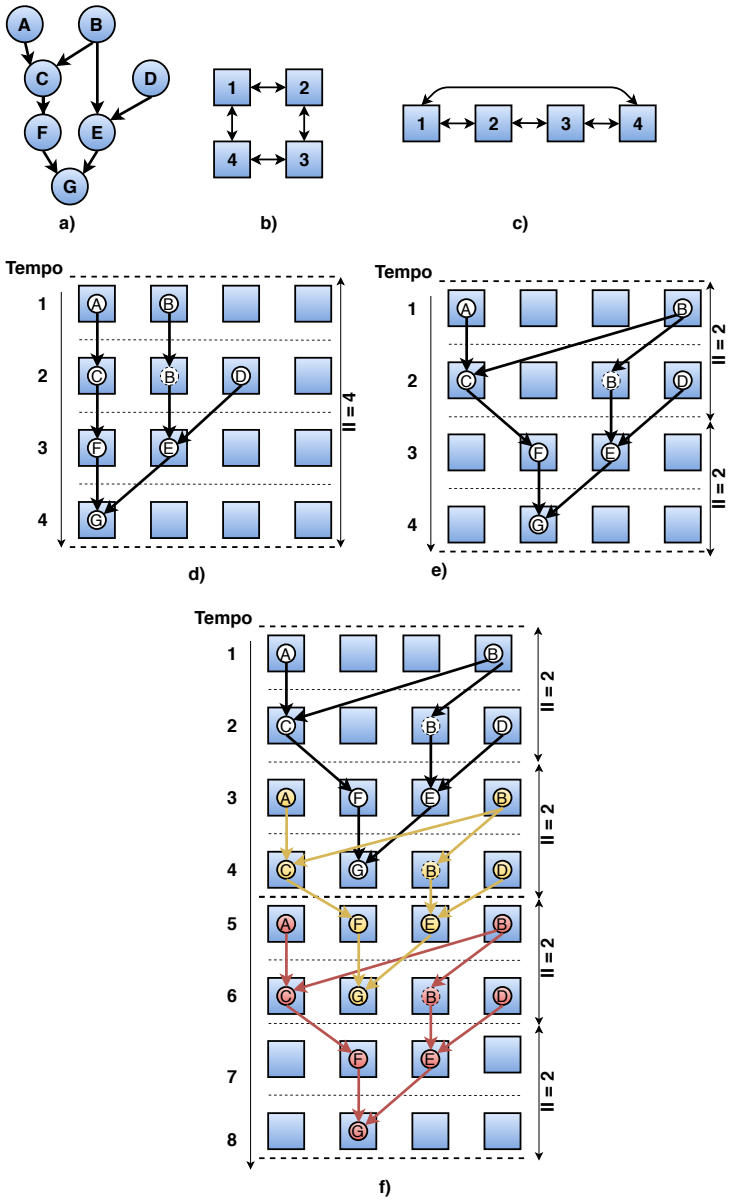

Figura 5 – Exemplos de redes de um mapeamento na arquitetura. a) Grafo de entrada, b) 2x2 CGRA, c) CGRA na forma linear, d) Mapeamento válido do grafo na CGRA com $II = 4$, e) Outro mapeamento com $II = 2$, f) Execução de três iterações sucessivas do laço

Com o grafo em mãos e conhecendo a arquitetura, já se inicia o processo de **escalonamento**. Na Figura 5d) é apresentado um primeiro mapeamento possível, nesse

mapeamento, são necessários 4 ciclos para ser executada uma iteração do laço. O mapeamento começa no ciclo 1, quando os nós A e B são mapeados nos PE_1 e PE_2. No próximo ciclo, os nós C e D são executados nos PE_1 e PE_3. Como a saída da instrução B só será usada no ciclo 3, seu resultado então é mantido no PE_2 no ciclo 2, para então se conectar com a instrução no ciclo 3, aqui já se caracteriza o **roteamento**.

No ciclo 3, os nós F e E são executados nos PE_1 e PE_2. Então, PE_1 executa o nó G no ciclo 4 e a execução de uma iteração do laço é terminada. A próxima iteração do laço então pode ser inicializada no ciclo 5, implicando em um $II = 4$. Com o fim do escalonamento e feita as conexões entre os nós seguindo a cadeia de operações, a etapa de **roteamento** é caracterizada.

O II (intervalo de iniciação) é proporcional ao tempo de execução e inversamente proporcional à performance. Como desejamos minimizar o tempo de execução, consequentemente, o II deve ser o mínimo possível. É possível aumentar a performance em duas vezes apenas realocando os nós em diferentes PEs. Esse processo de alocação e realocação caracteriza a etapa de **posicionamento**. Na Figura 5e), outro mapeamento do grafo é apresentado. No ciclo 1, os nós A e B são executados nos PE_1 e PE_4. No próximo ciclo, C e D são mapeados nos PE_1 e PE_4, novamente PE_3 guarda o nó B. No terceiro ciclo, os nós F e E são executados nos PE_2 e PE_3. E finalmente, PE_2 executa o nó G, completando uma iteração do laço.

Semelhante ao primeiro mapeamento, uma iteração do laço levou 4 ciclos para ser executada, entretanto, o II foi reduzido para 2. Para uma melhor visualização desse novo II, a execução de três iterações consecutivas do laço é feita na Figura 5f). A primeira iteração é caracterizada pelos nós na cor branca, a segunda iteração os nós possuem a cor amarela e na terceira iteração os nós possuem a cor vermelha. Como pode ser visto, no ciclo 3, PE_1 e PE_4 não estão sendo utilizados por nenhum nó da primeira iteração. Logo, é possível utilizá-los para executar os nós A e B da segunda iteração, asim como acontece com C, D e B no ciclo 4. Portanto, a execução da segunda iteração do laço pode ser feita sem conflito de recursos com a execução da primeira iteração do laço, assim como a execução da terceira iteração, que segue o mesmo padrão de mapeamento das demais.

Esse exemplo demonstra a execução do *modulo scheduling* de forma geral, mostrando a execução de nós de diferentes iterações sem a ocorrência de conflitos, além de exemplificar os problemas de escalonamento, posicionamento e roteamento.

3 Estado da Arte

Neste capítulo, alguns trabalhos relevantes da literatura que envolvem o escopo deste trabalho serão apresentados.

3.1 Trabalhos Relacionados

Esta seção foi subdividida em dois tópicos de estudo, o primeiro referente aos trabalhos relacionados à aceleração de laços e o segundo referente ao *modulo scheduling*.

3.1.1 Aceleração de Laços

Aceleração de laços vem sendo consistentemente um atrativo campo de pesquisa quando se fala de compilação. Vários problemas devem ser superados para que se alcance uma taxa de aceleração relevante em laços (MUCHNICK, 1997). O objetivo principal em todos esses problemas é minimizar os tempos de execução e maximizar a utilização dos recursos computacionais. Esses problemas incluem otimização de memória, desenrolamento de laços, vetorização, *software pipelining*, e uma eficiente representação dos recursos, como pode ser visto na Figura 6.

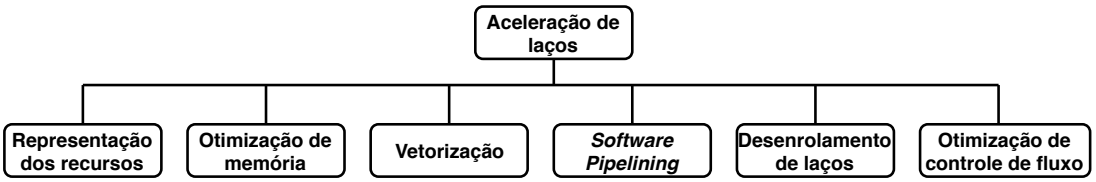

Figura 6 – Problemas envolvendo aceleração de laços.

O objetivo em otimização de memória é dispor as variáveis do laço de certa forma que elas possam ser melhor acessadas em relação ao padrão de acesso do laço(MEHTA; LIN; YEW, 2014; RAVISHANKAR et al., 2012; STOCK et al., 2014). Existem diversas técnicas que capturam o padrão de acesso do laço e o transformam de forma a maximizar a distância entre as dependências de memória.

Desenrolamento de laços é outra importante otimização, essa técnica desenrola o laço uma quantidade de vezes com o objetivo de aumentar a utilização de recursos, atuando na remoção e redução das iterações (CARDOSO; DINIZ, 2004; CARR; GUAN, 1997; GUPTA et al., 2004; SARKAR, 2000). Para uma quantidade limitada de laços, a vetorização é uma otimização para o desenrolamento de laços. Para esses laços, operações em várias iterações consecutivas são transformadas em um único vetor de operações. Essa auto vetorização em laços é um ativo campo de pesquisa (BANDISHTI; PANANILATH;

BONDHUGULA, 2012; KONG et al., 2013; VASILACHE et al., 2012) e que possui várias técnicas implementadas em compiladores comerciais (MALEKI et al., 2011).

Controle de dependências, seja dentro de uma mesma iteração do laço ou dentro de consecutivas iterações, limita a performance, principalmente no paralelismo em nível de instrução. Existem diversas técnicas que minimizam esse efeito, incluindo a técnica de otimização de controle de fluxo através de predição (MAHLKE, 1997; MAHLKE et al., 1995).

Quando uma arquitetura é exposta ao compilador, a representação de recursos é uma importante função no mapeamento. Diversas representações para os recursos já foram propostas, tais como representação em mesa (RAU, 1994), e representação em grafos (HATANAKA; BAGHERZADEH, 2007; MEI et al., 2003). Todas essas representações precisam ser endereçadas em um compilador, para que o mapeamento seja possível efetivamente. O objetivo do mapeamento é ordenar as instruções de modo que o tempo de execução do laço seja minimizado.

Outra técnica que atua no processo de aceleramento de laços, é a técnica de *software pipelinig* juntamente com o *modulo sheduling*, que será mais aprofundada na seção seguinte.

3.1.2 Modulo Scheduling

O primeiro trabalho publicado referente ao uso do *modulo scheduling* em CGRAs foi em (CALLAHAN; WAWRZYNEK, 2000), que aplicou o algoritmo para a arquitetura reconfigurável 1D, denominada *Garp*, além de atuar no desenvolvimento do compilador para a arquitetura. Mas foi em (MEI et al., 2002) que a utilização do *modulo scheduling* obteve um impacto significativo na área, com o desenvolvimento do sistema *DRESC*. Esse sistema é capaz de analisar, transformar e organizar código fonte da linguagem *C* para uma família de arquiteturas reconfiguráveis compatíveis com o compilador. Além disso, é utilizada a técnica de *simulated annealing*, na qual, a cada intervalo de iniciação, é gerado um escalonamento e posicionamento inicial das operações, satisfazendo as restrições de dependência, mas ainda pode haver excesso de uso dos recursos. Por exemplo, mais de uma operação é posicionada no mesmo elemento de processamento no mesmo ciclo. O algoritmo interativamente reduz esse excesso de uso e tenta fazer uma nova organização, posicionando randomicamente uma operação em um novo elemento de processamento. Feito isso, uma função que calcula o custo dessa nova organização é chamada e o compara com o custo encontrado da iteração anterior. Se o custo for menor do que o antigo, o novo posicionamento e roteamento é aceito. Até mesmo em casos em que o novo custo é maior, ainda há a chance de ocorrer a mudança, dependendo do que ele chama de "temperatura", que evita mínimos locais. A temperatura é gradualmente diminuída com o tempo, o que torna cada vez mais difícil mover uma operação. No fim, se o tempo atingir o limite, o algoritmo reinicia com um novo intervalo de iniciação. Apesar de possuir um

bom escalonamento, obtendo frequentemente o mínimo de intervalo de iniciação, o grande problema em *DRESC* é o seu alto tempo de execução, podendo chegar a 15 minutos em grafos com 80 operações. Detalhes sobre a função de custos e sobre a redução da "temperatura" podem ser vistos em (VASSILIADIS; SOUDRIS, 2007).

Particle Swarm Optimization (PSO) (KENNEDY; EBERHART, 1995) é outra busca heurística que imita o comportamento social dos bandos de pássaros. Essa estratégia de busca é usada por (GNANAOLIVU; NORVELL; VENKATESAN, 2010) para atribuir iterativamente operações a recursos e associar um custo a esse mapeamento. O grande problema nessa estratégia de busca é o longo tempo de execução.

EMS (PARK et al., 2008), reduzia o tempo de compilação em comparação a outros trabalhos da área. Enquanto *DRESC* e outros algoritmos de *modulo scheduling* utilizavam a abordagem de primeiro posicionar a operação e depois rotear, em *EMS*, o posicionamento era feito através do roteamento. Assim, o posicionamento somente seria realizado quando as informações sobre o roteamento estivessem disponíveis. Apesar de se mostrar mais rápido que *DRESC*, pois evitava muitas tentativas de posicionamento inviáveis de serem roteadas, *EMS* apresentava intervalos de iniciação maiores, caindo em até 85% na qualidade do escalonamento. Ainda utilizando a abordagem de *EMS* (orientar o posicionamento através do roteamento), (OH et al., 2009) ganha em tempo de compilação comparado com *DRESC*, e ainda atinge escalonamentos melhores que em *EMS*, mas possui a deficiência de ser duas vezes mais lento.

No trabalho proposto por (VENKATARAMANI et al., 2001b) foi apresentado uma técnica de mapeamento para a arquitetura *MorphoSys* (LEE et al., 2000). Uma vez que um tempo é atribuído às operações, é assumido que uma alocação de recursos válida pode ser feita, devido à rica quantidade de interconexões entre os PEs em *MorphoSys*. Depois, o grafo que será executado é particionado em grupos de 16 operações. O objetivo desse particionamento é minimizar o tempo de execução.

EPIMap (HAMZEH; SHRIVASTAVA; VRUDHULA, 2012) propõe o uso de recomputação para resolver os problemas de posicionamento na arquitetura, em que um nó pode ser calculado mais de uma vez. Apesar de obter resultados melhores em termos de intervalo de iniciação em comparação ao *EMS*, seu tempo de mapeamento é pior, chegando a ser 6 vezes maior que o presente em *EMS*. Na proposta de (CHEN; MITRA, 2014) é utilizado um banco de registradores e registradores locais como recursos de roteamento. Esse trabalho obteve uma melhoria com relação a *DRESC* em tempo de compilação e alocação dos elementos de processamento, alcançando em média 62% contra 54% de *DRESC*.

Em (FERREIRA et al., 2011) é proposto uma CGRA virtual, implementada sob uma FPGA e seu algoritmo de mapeamento. Possui uma rede global multiestágio, permitindo que um elemento de processamento possa ser conectado a qualquer outro, simplificando o processo de posicionamento e roteamento, com a possibilidade de conflitos

na rede. No trabalho de (FERREIRA et al., 2013) é proposta uma solução que fornece uma aceleração na fase de mapeamento baseada em três mecanismos: 1) uma simples e eficiente heurística de *modulo scheduling*; 2) uma rede *crossbar*; e 3) uma CGRA virtual, que é utilizada em cima de uma FPGA. A principal vantagem dessa estratégia é a baixa sobrecarga de configuração, já que a CGRA trabalha em nível de palavra em vez do nível de *bits* e a possibilidade de usar uma FPGA. A troca no uso de uma rede *crossbar* em vez de uma rede de malha reduz a complexidade no mapeamento das aplicações e consequentemente propicia uma simples e rápida fase de mapeamento. Em (FERREIRA et al., 2014) é proposto a aplicação de tradução binária em abordagens de *modulo scheduling*. Nessa abordagem, o ponto de partida é o grafo de fluxo de dados do laço (DFG - *Dataflow graph*), assim, é necessário um mecanismo especial de detecção de laços. Nesse trabalho, é proposto um algoritmo de detecção, geração e organização do laço a partir do código binário.

4 Sistema Proposto

A proposta deste trabalho é desenvolver um mecanismo de *modulo scheduling*, em *software*, a partir de um compilador disponível já implementado. O principal objetivo é gerar configuração para uma CGRA alvo e estudar a performance e desempenho dessa geração, analisando principalmente o ILP das aplicações que serão executadas.

4.1 Arquitetura Alvo

A arquitetura alvo (SILVA, 2017) é composta de uma coluna dotada de 5 elementos de processamento e uma unidade *Load/Store* de dois estágios. A arquitetura é controlada por uma máquina de estados (FSM - *Finite State Machine*) e possui um banco de registradores (BR), como pode ser visto na Figura 7.

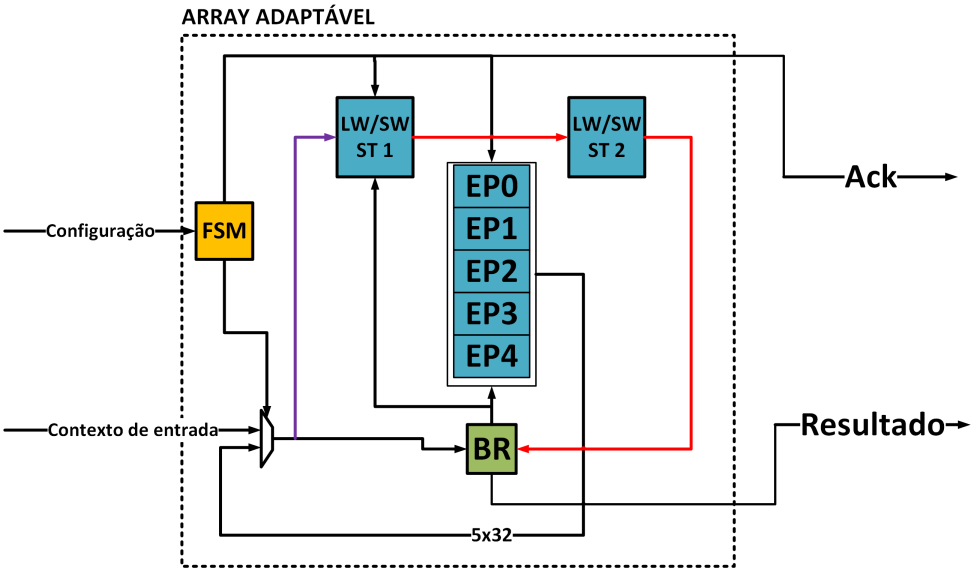

Figura 7 – Arquitetura alvo, por (SILVA, 2017).

O *array* reconfigurável é integrado a um processador *multicore* com 4 núcleos. Cada um desses núcleos é um MIPS *pipeline* de 5 estágios como descrito por (SILVA et al., 2014). Cada núcleo tem acoplado a si um *array* reconfigurável, juntamente com uma cache de configuração. O *array* reconfigurável é fortemente acoplado ao núcleo MIPS *pipeline*, uma vez que precisa ter acesso ao banco de registradores do processador. A integração do *array* reconfigurável com os 4 núcleos do processador *multicore* pode ser vista na Figura 8. Assim, em um processador *multicore* de 4 núcleos será necessária a utilização de um *array* de 4 colunas.

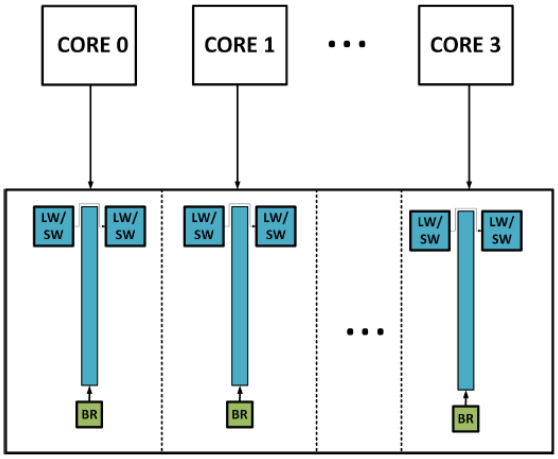

Figura 8 – Integração processador *multicore* com *array* adaptável, por (SILVA, 2017).

4.2 Compilador

O compilador utilizado foi implementado na linguagem de programação *Java*. Possui como entrada a aplicação escrita na linguagem de programação *Go* e possui como saída o conjunto de instruções *assembly* MIPS referentes àquela aplicação de entrada. O algoritmo de*modulo scheduling* foi implementado dentro do compilador para a geração das configurações. Esse processo pode ser visto na Figura 9.

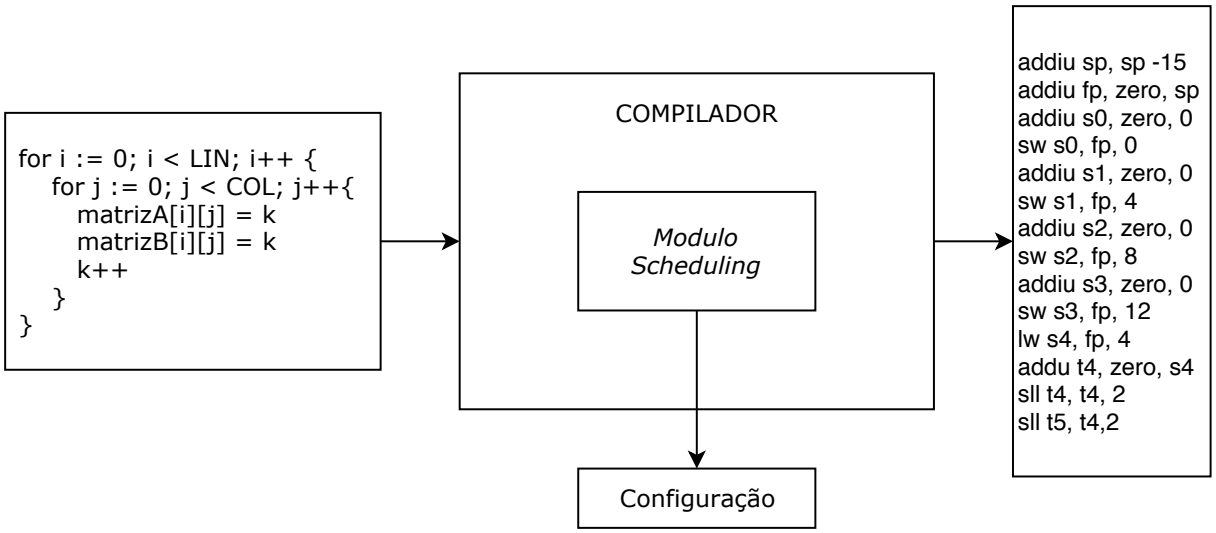

Figura 9 – Exemplo de programa em *Go* utilizado no compilador. E o mesmo programa no conjunto de instruções MIPS.

4.3 Modulo Scheduling

Nesta seção será apresentado o algoritmo de *modulo scheduling* proposto aqui no trabalho.

4.3.1 Detecção dos laços da aplicação

O primeiro passo no processo de *modulo scheduling* é a detecção dos laços presentes na aplicação. Os laços extraídos pelo compilador são caracterizados por possuírem como retorno da instrução de salto uma instrução que possui um endereço anterior, ou seja, um laço é caracterizado quando a instrução referente ao salto, executa esse salto para trás. Um exemplo de salto produzido pelo compilador é mostrado na Figura 10. Ainda pode ocorrer laços dentro de outros laços, conhecidos como laços aninhados (*nested loops*), como pode ser visto na Figura 11.

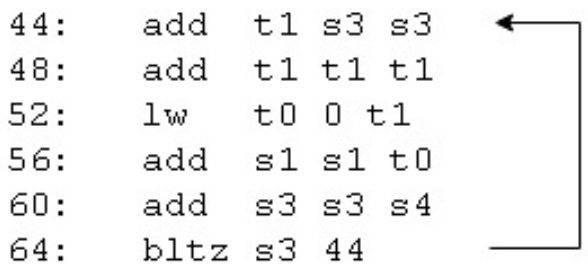

Figura 10 – Exemplo de laço: a instrução com endereço 64 executa um salto para a instrução com endereço 44, se o valor de s3 for menor que 0.

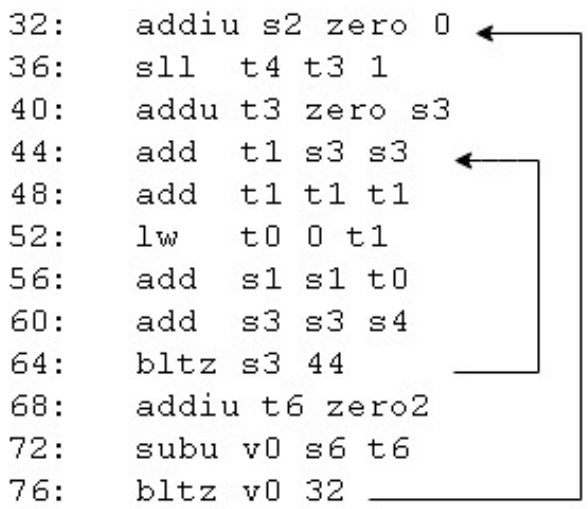

Figura 11 – Laços aninhados: a instrução com endereço 76 executa um salto para a instrução 32, se o valor de v0 for menor que 0. E a instrução com endereço 64 executa um salto para a instrução 44, se o valor de s3 for menor que 0. O segundo laço é executado dentro do primeiro laço.

Neste trabalho, nos casos em que os laços detectados são caracterizados como laços aninhados, apenas o laço mais interno é escolhido para o processamento do *modulo scheduling*. Isso acontece devido ao laço mais interno ser executado mais vezes, assim, otimizações nos laços mais internos obterão um ganho de desempenho maior na execução da aplicação. Essa ideia pode ser justificada pela Lei de Amdahl (GUSTAFSON, 1988),

que é usada para encontrar a máxima melhora esperada para um sistema em geral quando apenas uma única parte do mesmo é melhorada.

4.3.2 Geração do Grafo de Dependências

Após detectados os laços presentes na aplicação, é preciso gerar o grafo de dependências entre as instruções, chamado de *Data-flow Graph* (DFG). Com as instruções do laço mais interno em mãos, uma busca *bottom - up* é feita a partir da última instrução do laço. No caso do exemplo da Figura 11, da instrução com endereço 64 (`bltz s3 44`) até a instrução com endereço 44 (`add t1 s3 s3`).

Nessa busca de instruções é onde ocorre o processo de análise das dependências. Cada nova instrução é marcada como visitada e verificada se algum operando utilizado na instrução vigente estava presente e/ou foi modificado em alguma das instruções anteriores. Dessa forma, existem três possibilidades, mostradas a seguir:

1. Se o operando estiver presente, mas não foi modificado, não existe dependência entre as instruções, pois o valor do registrador comparado ainda será o mesmo da instrução anterior;

2. Se o operando estiver presente e modificado, uma aresta é criada entre essas instruções (nós do grafo), caracterizando uma dependência. A cada nova instrução dependente, uma nova aresta é criada entre as instruções, ligando assim os nós do grafo;

3. Se o operando não estiver presente, a instrução é caracterizada como um nó folha, ou seja, sem dependência com outras instruções.

Essas possibilidades podem ser melhor visualizadas na Figura 12. Um exemplo de grafo gerado a partir de um trecho de código pode ser visto na Figura 13.

Na Figura 13, a primeira instrução a ser executada possui endereço 120, logo após, a instrução 128 é executada. Perceba que existe a dependência entre as instruções 120 e 128 no registrador `s3`. Ou seja, para a execução da instrução 128, é necessário o valor do registrador `s3` presente a instrução 120, o que caracteriza a ligação (120, 128) no *DFG*. Da mesma forma existe a dependência entre 128 e 132 no registrador `t7`. Já na instrução 136 não há dependência entre nenhuma das instruções anteriores, podendo então ser executada em paralelo com a instrução 128. Assim acontece, sucessivamente com todas as demais instruções. De forma geral, instruções que não possuam dependência entre si podem ser executadas em paralelo.

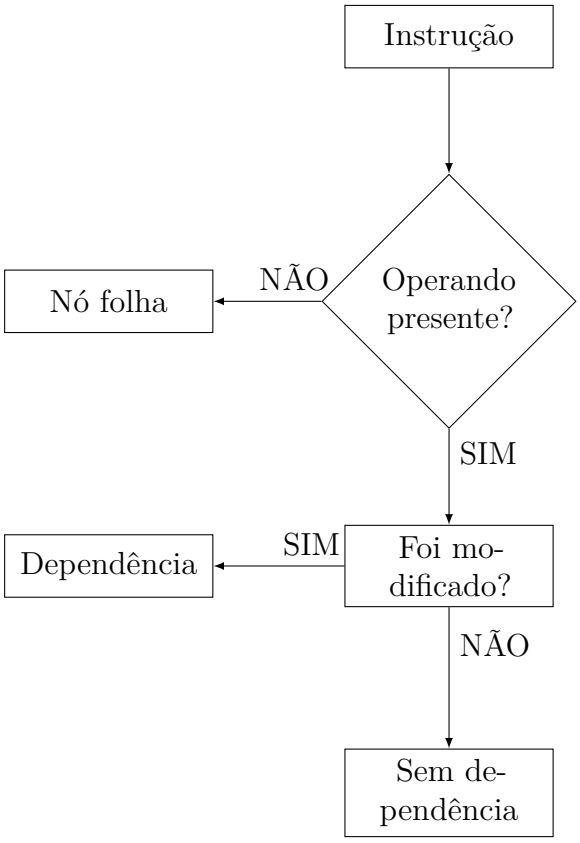

Figura 12 – Esquema do processo de detecção das dependências.

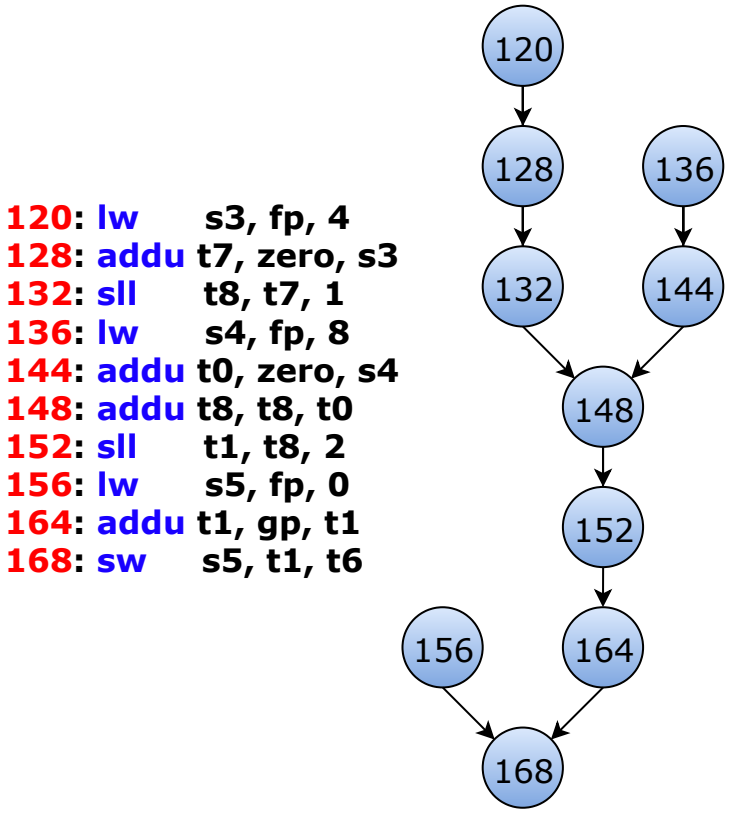

Figura 13 – Exemplo de grafo gerado pelo compilador.

4.3.3 Exemplo do Funcionamento do Algoritmo

Considere o exemplo de *DFG* na Figura 14, o grafo possui 4 nós. Os nós D e E podem ser executados em paralelo, o nó F só pode ser executado após a execução do nó D e por último o nó G só pode ser executado após a execução dos nós F e E.

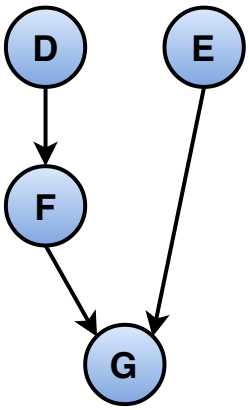

Figura 14 – Exemplo de grafo.

Para a arquitetura alvo do trabalho, a execução desse grafo sem a aplicação do *modulo scheduling* é apresentada na Figura 15:

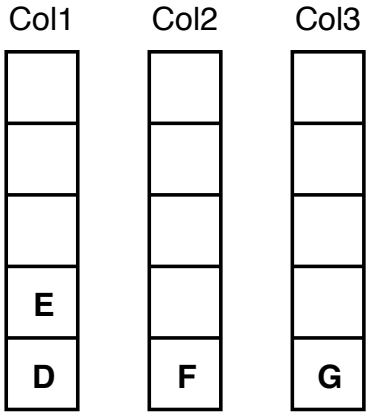

Figura 15 – Execução do grafo sem *modulo scheduling*.

Na arquitetura alvo, cada nível do *DFG* é executado em uma coluna do *array* reconfigurável da *CGRA*. Assim, o nível 0 (composto pelos nós D e E) é executado pela primeira coluna, o nível 1 (nó F) é executado na segunda coluna e o nível 2 (nó G) na terceira coluna.

Já usando a técnica de *modulo scheduling* no grafo anteriormente citado, o primeiro passo é calcular o mínimo intervalo de iniciação (MII). O MII será calculado considerando o número de operações a serem mapeadas dividido pelo número de PEs disponíveis, o resultado seria $4/5 = 0{,}8$, ou seja, no mínimo 1, logo o grafo será replicado 1 nível abaixo com relação ao grafo inicial, conforme mostra a Figura 16.

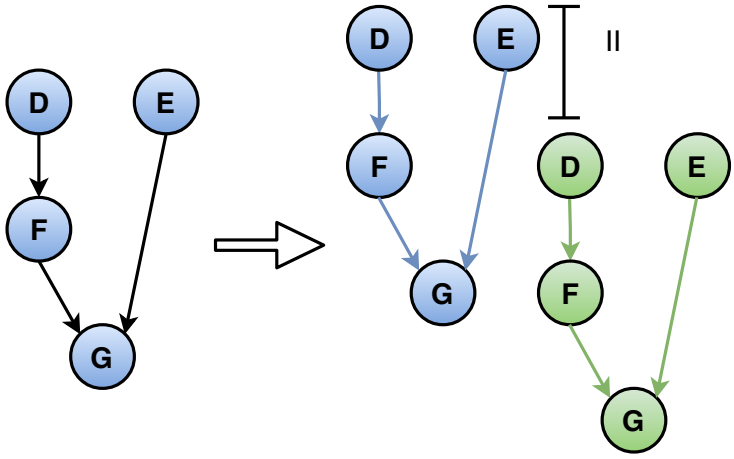

Figura 16 – Execução do grafo com o *modulo scheduling*.

Na Figura 16, o *DFG* à esquerda é o grafo original, sem *modulo scheduling*. Já no *DFG* à direita, com o uso do *modulo shcheduling*, já foi criada uma nova iteração ou partição do grafo original, baseado no intervalo de iniciação anteriormente calculado, igual a 1.

Perceba, que com a execução do grafo com *modulo scheduling*, ainda não é possível paralelizar todas as instruções do laço original com apenas duas partições, ou seja, ainda não foi criado o *kernel* desse laço. Então, uma nova partição do grafo original deve ser criada.

Na Figura 17, uma terceira partição do grafo original foi criada. Veja que no nível 2 é possível executar os quatro nós do grafo original, agora sim todos os nós estão paralelizáveis, criando então o *kernel* do novo laço, juntamente com seu prólogo e epílogo, como ilustrado na Figura 18.

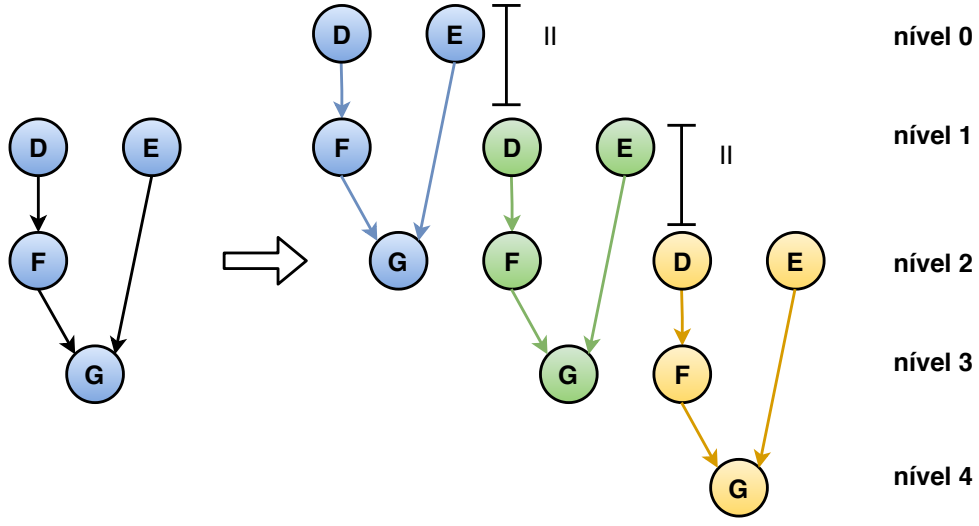

Figura 17 – Execução do grafo com o *modulo scheduling* utilizando três partições.

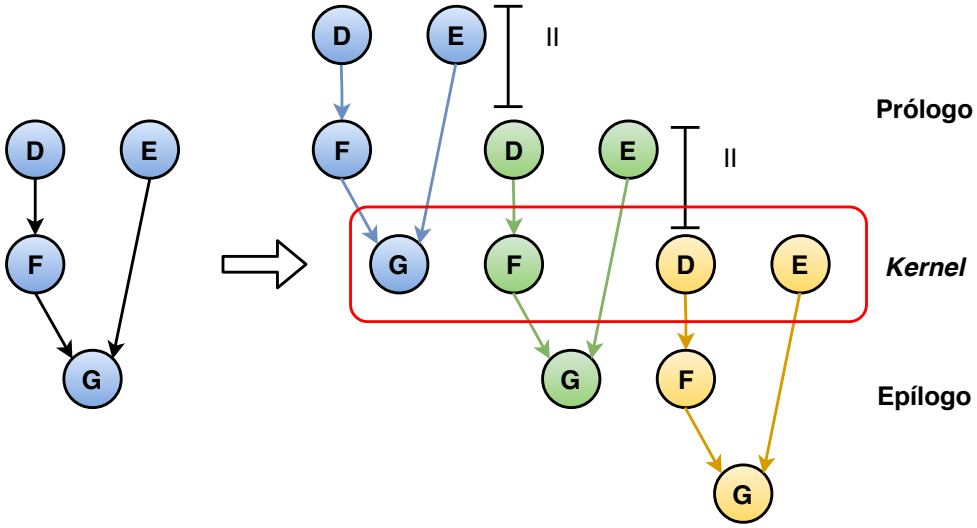

Figura 18 – Demonstração do prólogo, *kernel* e epílogo

Feita a demonstração da execução do *modulo scheduling* do *DFG* inicial da Figura 14, o novo mapeamento desse grafo na *CGRA* após o uso do *modulo shceduling* é demonstrado na Figura 19.

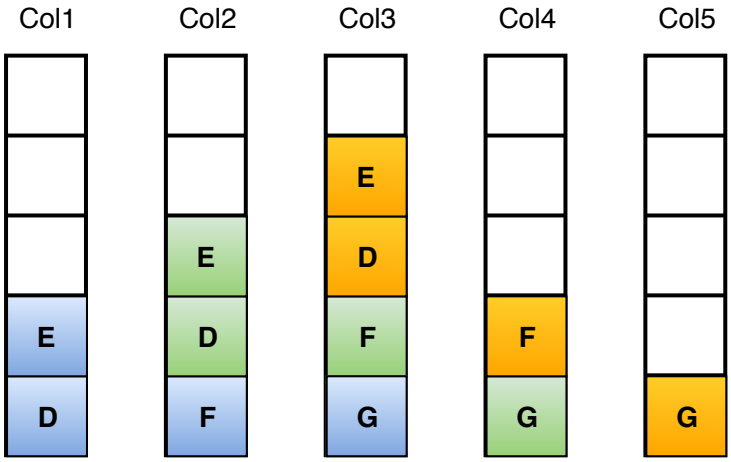

Figura 19 – Execução do grafo com o *modulo scheduling* utilizando três partições.

Como já dito, cada nível do grafo é mapeado para uma coluna do *array* reconfigurável, os PEs representados pela cor azul são referentes à primeira partição do grafo, na cor verde à segunda partição e na cor amarela à terceira partição.

Uma configuração pode ser vista como um *DFG*, onde cada nível do grafo é uma palavra de configuração. A profundidade da configuração remete à quantidade de palavras que uma configuração possui. Ou seja, cada nível é uma coluna do *array* reconfigurável e a quantidade de níveis é a quantidade de colunas alocadas para executar tal trecho.

A grande vantagem do trabalho proposto é que não é preciso se preocupar com a etapa de roteamento, que faria a ligação entre os elementos de processamento na arquitetura, pois na arquitetura alvo do trabalho, os PEs se comunicam através do banco

de registradores. Veja o exemplo do *modulo scheduling* em uma arquitetura alvo onde a etapa do roteamento é fundamental para o bom funcionamento do mecanismo, como mostra a Figura 20.

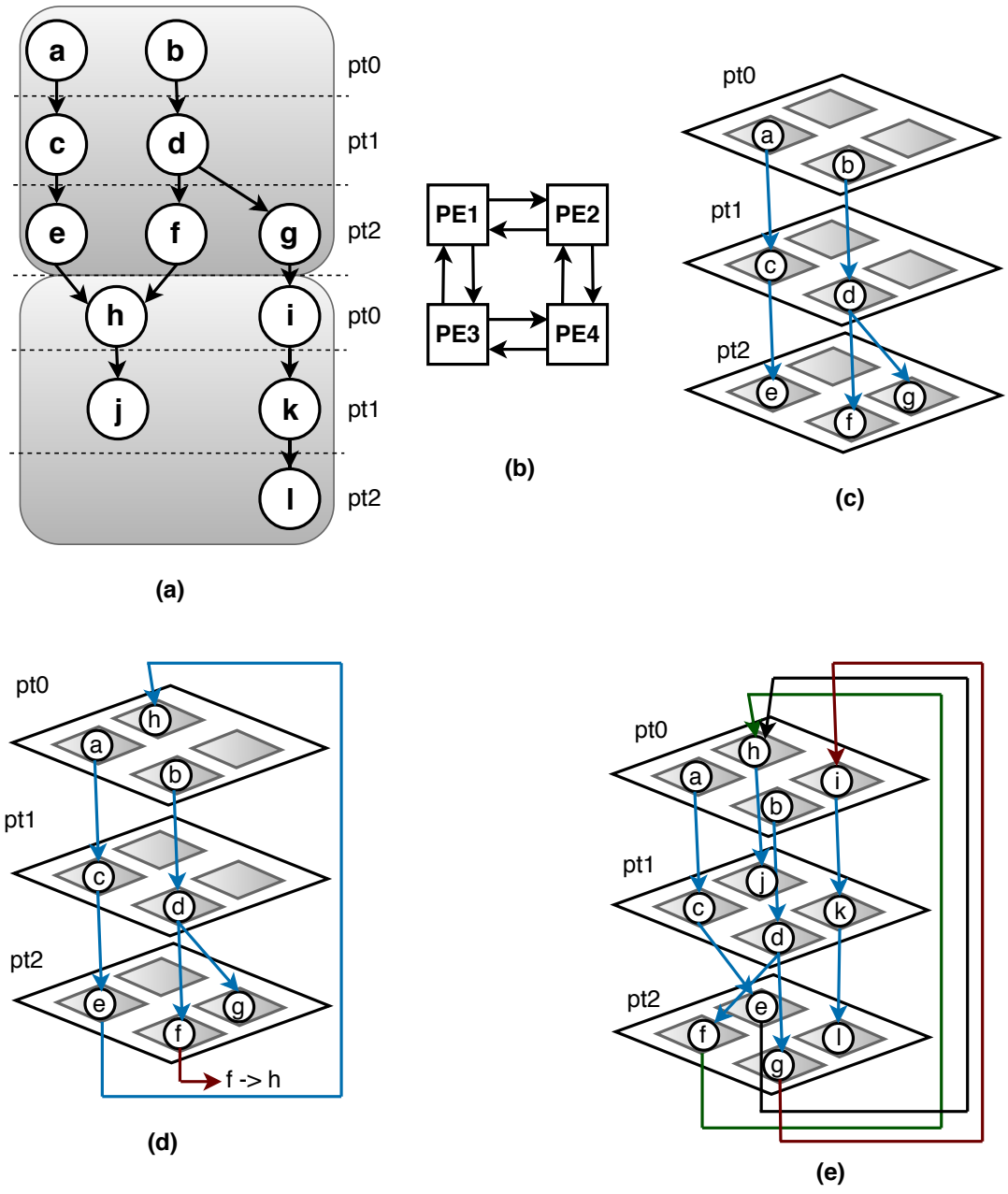

Figura 20 – Mapeamento do grafo em (LOPES, 2013).

Nesse exemplo, os nós a e b são posicionados nos PE1 e PE3 na partição pt0. Os nós c e d devem ser posicionados em pt1 devido às suas dependências de a e b. A Figura 20(c) mostra que c e d foram posicionados nos mesmos PEs que a e b e o roteamento é possível, pois a arquitetura suporta o roteamento entre os PEs 1 e 3. Para e, f e g, um posicionamento e roteamento possível na partição pt2 é mostrado na Figura 20(c). Entretanto, quando o nó h é posicionado na PE2 na pt0, não é possível rotear de f na PE3 para h no PE2, pois não há conexões diagonais na arquitetura da Figura 20(b). O

roteamento f –> h falha, como mostrado em vermelho na Figura 20(d).

A solução proposta por (LOPES, 2013) foi mudar o posicionamento de e, f e g na pt2. Assim, e e f da pt2 podem rotear para h na pt0, assim como o grafo todo: g –> i, h –> j, i –> k, e k –> l, como mostra a Figura 20(e). Portanto, nos trabalhos onde o roteamento é fundamental, várias alternativas para o roteamento podem ser avaliadas durante o processo de escalonamento e a rede de interconexão influencia diretamente. Isso afeta no tempo total de execução do processo, problema que não existe na abordagem presente neste trabalho. No trabalho aqui proposto, o mesmo exemplo seria representado pela Figura 21.

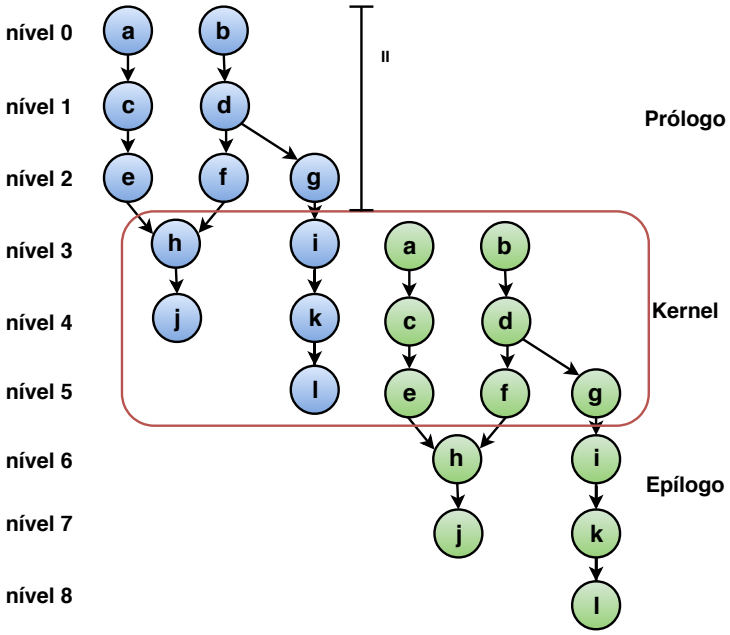

Figura 21 – Execução do grafo da Figura 20.

No cálculo do MII, a quantidade de 12 nós é dividida pela quantidade de 5 PEs, criando um MII = 3. Por isso, a segunda partição do grafo inicial foi replicada no quarto nível. Criando um *kernel* composto pelas 12 operações que serão executadas em 3 colunas do *array* reconfigurável.

4.3.4 Falhas no Processo

Como já falado, o intervalo de iniciação é incrementado quando a execução do *modulo scheduling* não encontra uma configuração considerável estável. Considere a Figura 22.

Possuindo o MII = 2, é possível observar que essa configuração não é satisfatória, apesar de no nível 2 e nível 3 todos os nós estarem sendo executados. No nível 2 a arquitetura não suporta essa configuração, pois o número de PEs por coluna é extrapolado, pois, como já dito, a arquitetura suporta apenas 5 PEs por coluna, e no nível 2, 6 instruções devem sem executadas, extrapolando assim a quantidade de PEs. A solução seria incrementar o

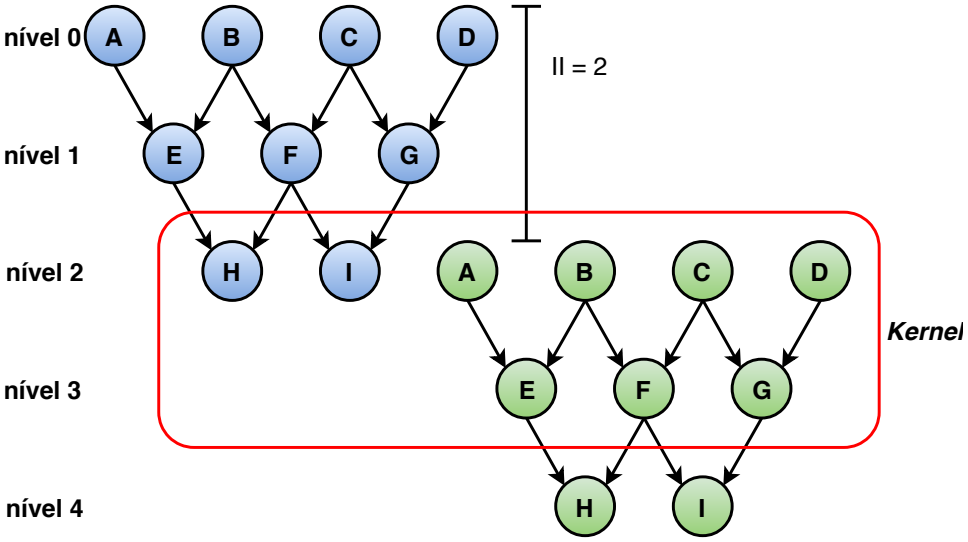

Figura 22 – Intervalo de iniciação = 2. Apenas duas partições.

intervalo de iniciação. Com o II = 3, a nova configuração fica como mostra a Figura 23. Nessa nova configuração, apesar de não extrapolar nenhum limite da arquitetura, não caracteriza o *modulo scheduling*, pois não há a sobreposição de iterações do grafo, e sim, apenas uma execução do mesmo grafo duas vezes, uma execução começando no nível 0 e outra execução começando no nível 3.

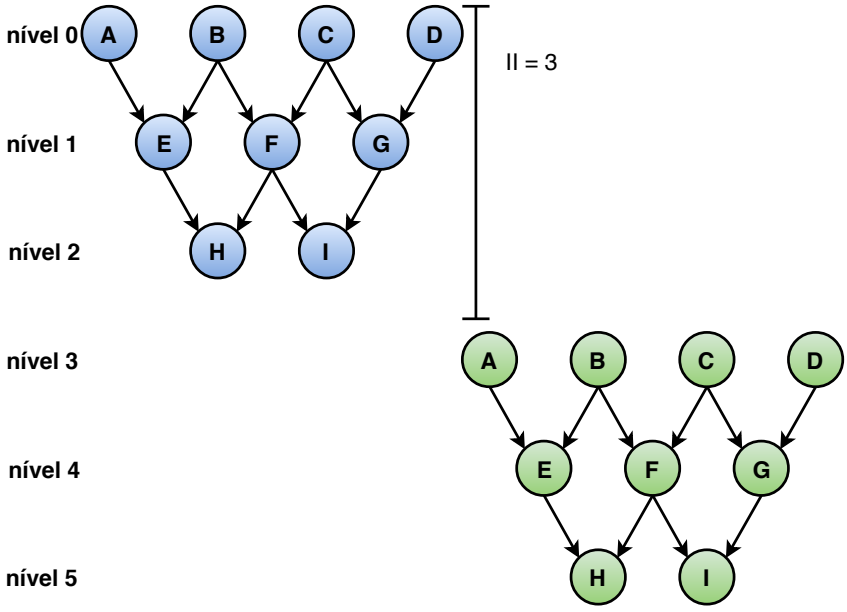

Figura 23 – Intervalo de iniciação = 3. Apenas duas partições.

Uma solução para esse tipo de problema é a realocação de nós para colunas onde a quantidade de PEs não é extrapolada. Na Figura 22, um nó presente no nível 2 deve ser realocado para outra coluna, é então escolhido o nó D para ser realocado. No nível 0, apesar de apenas 4 nós estarem sendo executados, já existe o nó D da primeira iteração. Saltando para o nível 1, apenas 3 operações estão sendo realizadas, havendo então espaço

para a alocação do novo nó, como ilustrado na Figura 24. Essa realocação do nó D do nível 2 para o nível 1 não afeta a execução dos nós não realocados, pois como o nó D já foi executado na primeira iteração, essa operação já foi feita e somente terá seu valor copiado no nó D presente no nível 1.

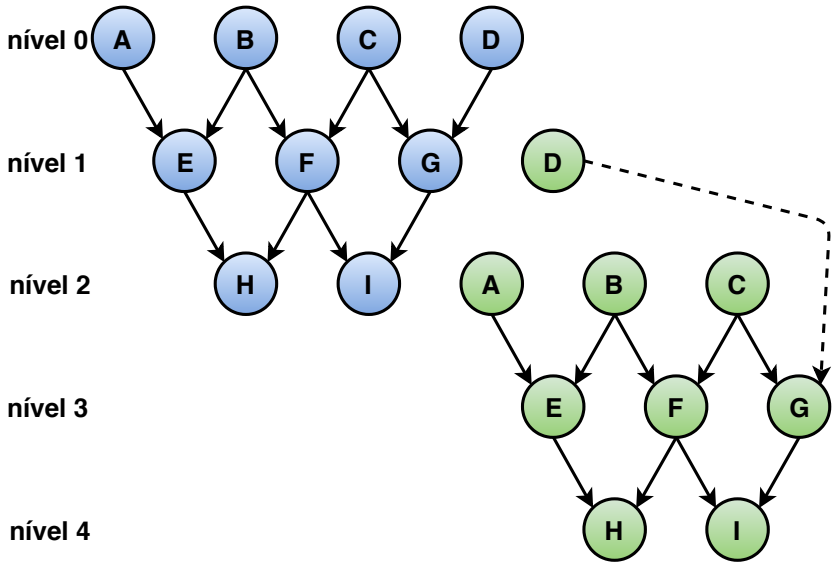

Figura 24 – Exemplo de mapeamento do grafo na arquitetura.

Por último, em um caso extremo onde há a extrapolação dos limites da arquitetura e que não possa ser feita a relocação desses nós para outras colunas, chega-se à conclusão de que a melhor configuração para o grafo já é o seu estado inicial.

4.4 Restrições de recursos da arquitetura

Como já falado na seção 2.3, a sobreposição de diferentes iterações do laço deve garantir a não violação das dependências internas e externas do grafo, além de evitar restrições de recursos. Um escalonamento válido é aquele em que não há nenhum conflito no uso de recursos entre as operações e nenhuma dependência interna ou externa é violada. Neste trabalho, a não violação das dependências internas (quando os nós dentro da mesma iteração são dependentes entre si) é garantida a partir da criação do *DFG*, que analisa instrução por instrução para garantir que a dependência existente entre as mesmas seja mantida, permitindo a correta execução das operações pela arquitetura. Já para lidar com a violação das dependências externas (quando nós de uma determinada iteração dependem do resultado da iteração anterior) um mecanismo de *Register Renaming* (Renomeação de registradores) (PATTERSON; HENNESSY, 1990) foi implementado.

- Mecanismo para lidar em casos de violação das dependências externas:

  Como dito, a opção utilizada para lidar com esse tipo de violação foi o *register renaming*. Essa técnica consiste em renomear os registradores, aumentando o paralelismo

disponível, ou seja, permitindo que duas instruções que antes deveriam ser executadas serialmente agora possam ser executadas em paralelo. Considere o exemplo, onde as instruções `add r7 r5 r6` e `sub r5 r9 r6` pertencem a diferentes iterações e ambas devem ser executadas em paralelo. É possível perceber que o registrador `r5` é lido na primeira instrução e escrito na segunda instrução, caracterizando uma falsa dependência denominada *write after read*, pois não é possível ler e escrever em um registrador paralelamente, logo uma alternativa nesse caso é renomear o segundo registrador `r5` para um nome qualquer, aqui chamado de `r45`. Agora, `add r7 r5 r6` e `sub r45 r9 r6` podem ser executadas em paralelo, pois não há mais nenhum outro conflito.

Outra falsa dependência existente, é a chamada *write after write*. Quando uma instrução escreve no mesmo registrador que foi escrito na instrução anterior, como em `add r7 r5 r6` e `add r7 r1 r2`, novamente o mecanismo de *register renaming* pode ser utilizado, renomeando o segundo resgistrador `r7` e permitindo então a execução em paralelo.

Com relação a restrições de recursos, a arquitetura utilizada no trabalho não suporta a execução de mais de uma instrução de acesso à memória por coluna do *array* reconfigurável, como *load words* (lw) e *store words* (sw). Além de possuir a limitação de 5 PEs por coluna. Logo, é necessário um mecanismo especial para lidar em casos onde esses limites da arquitetura sejam extrapolados.

- Mecanismo para lidar em casos onde existe mais de uma instrução de acesso à memória por coluna:

  Após realizado o algoritmo de *modulo scheduling*, percebe-se que existem colunas que possuem mais de uma instrução de acesso à memória para serem executadas. Logo, uma dessas instruções deve ser realocada para outra coluna.

  Considere o exemplo hipotético de MS com duas partições presente na Figura 25(a), no nível 3, duas instruções de acesso à memória devem ser executadas na mesma coluna, mas sabemos que essa execução não é possível, uma solução é descer uma dessas instruções para um nível abaixo, logo a instrução *LW* da segunda partição (representada na cor verde) desce para ser executada no nível 4, consequentemente a instrução *D* da segunda partição também deve ser executada um nível abaixo (por se tratarem de instruções dependentes), e assim sucessivamente com as outras instruções, como é demonstrado na Figura 25(b). O resultado dessa configuração na arquitetura é representado na Figura 25(c), onde cada nível do grafo é mapeado em uma coluna do *array*. Tratando o problema dessa forma é obtido um *ILP* igual a 1.75, onde quatorze instruções são executadas em oito colunas, a partir do cálculo

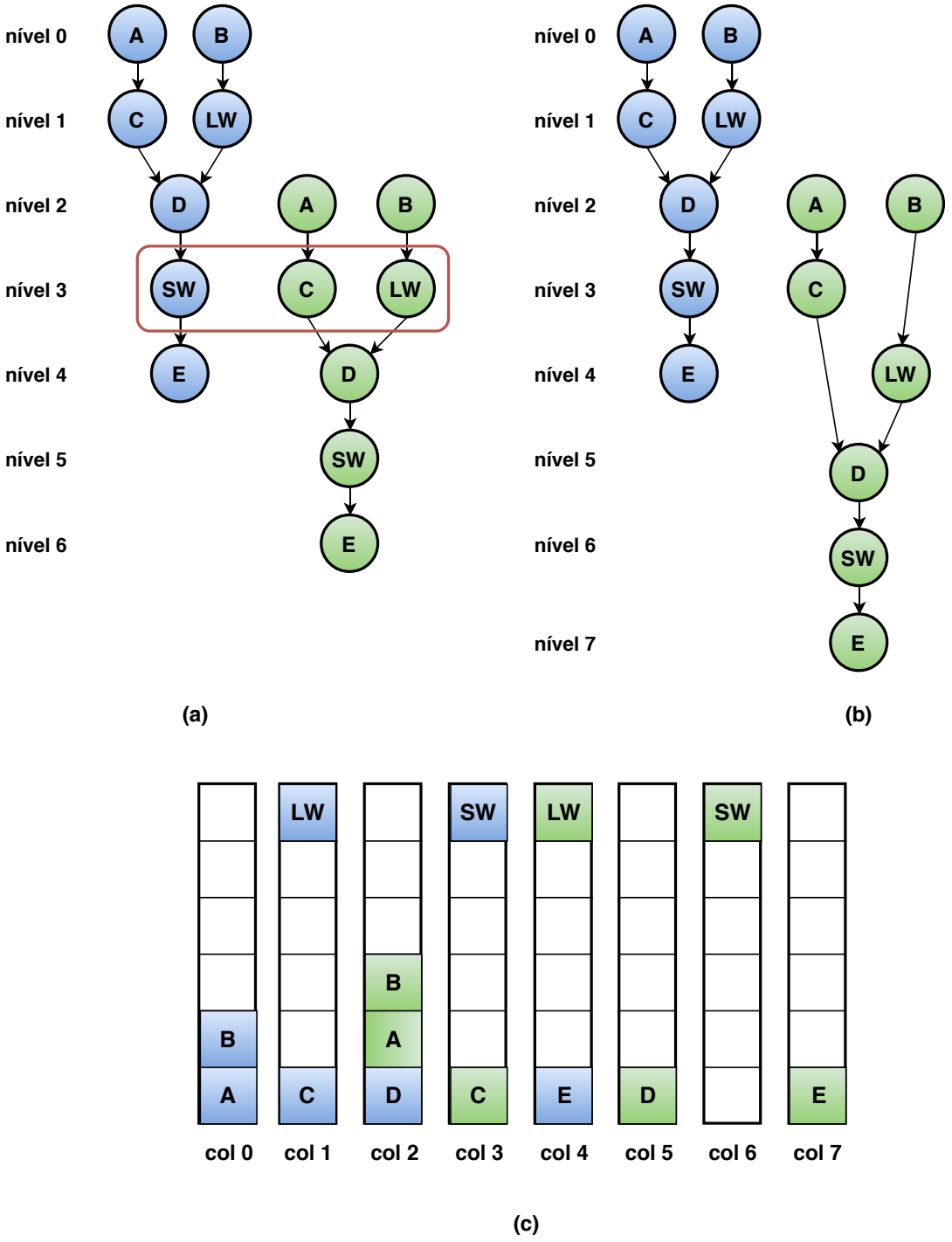

Figura 25 – Execução do grafo na arquitetura.

do ILP onde a quantidade de instruções será dividida pela quantidade de colunas utilizadas.

Uma outra solução para esse problema pode ser vista na Figura 26.

Como já falado anteriormente, o algoritmo de *MS* é dividido em problema de escalonamento, posicionamento e roteamento. No trabalho proposto não precisamos nos preocupar com o roteamento das instruções do grafo, pois a arquitetura alvo salva o resultado em um banco de registradores. Isso traz uma grande vantagem

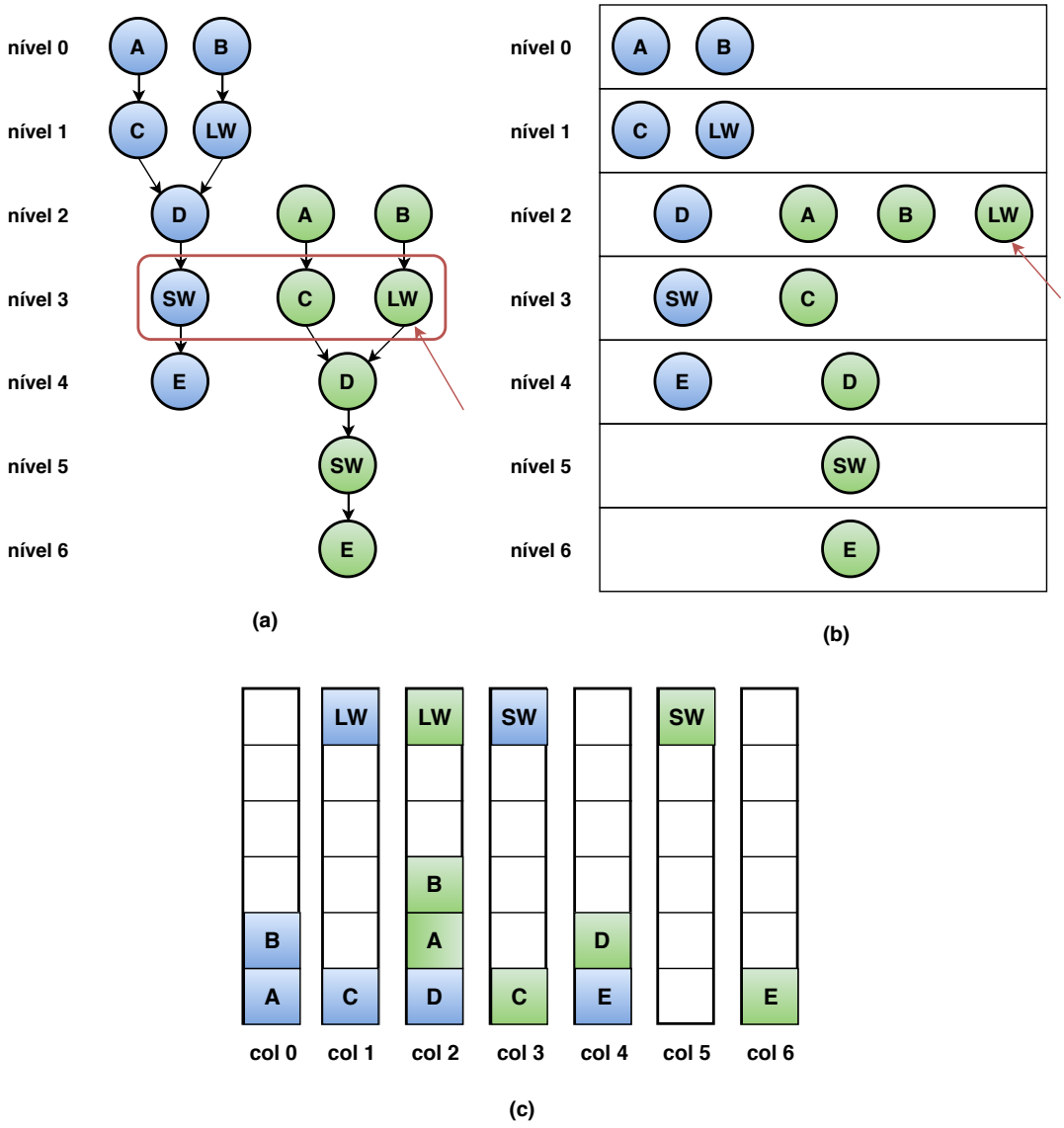

Figura 26 – Nova configuração, utilizando uma nova abordagem para realocação dos nós.

para o trabalho. Veja que nessa abordagem mudamos a representação do grafo da Figura 25(b) para uma nova representação em linhas na Figura 26(b), pois estamos descartando o roteamento entre as instruções. Nessa nova abordagem, podemos realocar a instrução *LW* do nível 3 para o nível 2, pois o que nos impedia de fazer isso, era o roteamento da instrução *B* no nível 2 com *LW* no nível 3 na Figura 26(a). Na abordagem em linhas, é possível realocar a instrução *LW* no nível 2, pois já temos o seu resultado quando a instrução foi executada no nível 1, então somente passamos o resultado da instrução do nível 1 para o nível 2 na Figura 26(b).

1. Mas por que mover a instrução *LW* e não a instrução *SW* presentes no nível 3 na Figura 26(a)?

   Movendo-se a instrução *SW*, perdemos a integridade dos valores das instruções, o resultado da instrução *E* no nível 4 estaria errado, pois a mesma depende de

SW no nível 3.

2. E pra onde mover a instrução *LW* escolhida?

   A instrução deve ser movida para um nível onde não haja nenhuma outra instrução de acesso a memória e que a mesma já tenha sido executada. Por exemplo, na Figura 26(b), a instrução *LW* foi realocada no nível 2, pois é um nível que não apresenta nenhuma outra instrução de acesso à memória e é um nível em que a intrução a ser movida já foi executada anteriormente, no caso, no nível 1. Por isso, a escolha do nível 2 para a realocação.

Então, a regra geral aqui é sempre mover a instrução que não afeta a integridade dos dados das outras instruções no *DFG*, e movê-la para uma linha onde não há outra instrução de acesso a memória e onde o resultado dessa instrução já esteja disponível na iteração anterior. Nessa abordagem, o *ILP* obtido é igual a 2, que quatorze instruções são executadas em sete colunas. Obtendo um ganho de 0.25 no *ILP* em relação à abordagem da Figura 25. No trabalho proposto, em casos onde acontece esse tipo de restrição, é utilizado a segunda abordagem apresentada, visto que é obtido um maior *ILP* na execução das instruções.

- Mecanismo para lidar em casos onde a quantidade de instruções excede a quantidade de PEs:

  Outra restrição de recurso da arquitetura é a limitação na quantidade de PEs por coluna do *array* reconfigurável, podendo executar apenas 5 instruções em 5 PEs. Portanto, em alguns casos na execução do algoritmo de *modulo scheduling*, essa restrição acaba sendo extrapolada, logo instruções que excedem esse limite devem ser realocadas para outras colunas. Nesse trabalho, foi proposta a seguinte abordagem nesses casos. Considere o exemplo de MS da Figura 27.

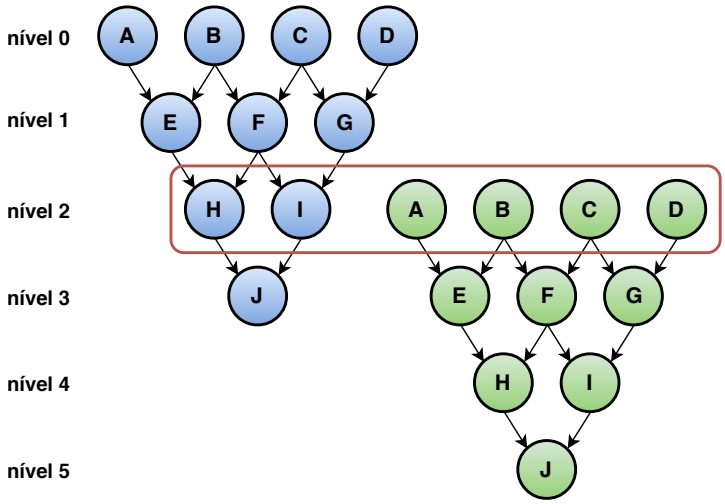

Figura 27 – Grafo com restrição de recurso no nível 2.

No nível 2, 6 instruções devem ser executadas, logo uma dessas instruções deve ser realocada em outra coluna. Uma solução possível é representada na Figura 28. Onde a instrução *D* desce do nível 2 para o nível 3 do grafo para então ser executada, consequentemente todas as outras instruções também descerão um nível no grafo, pois todas são dependentes entre si. Nessa abordagem, vinte instruções serão executadas em sete colunas, totalizando um *ILP* de 2.85.

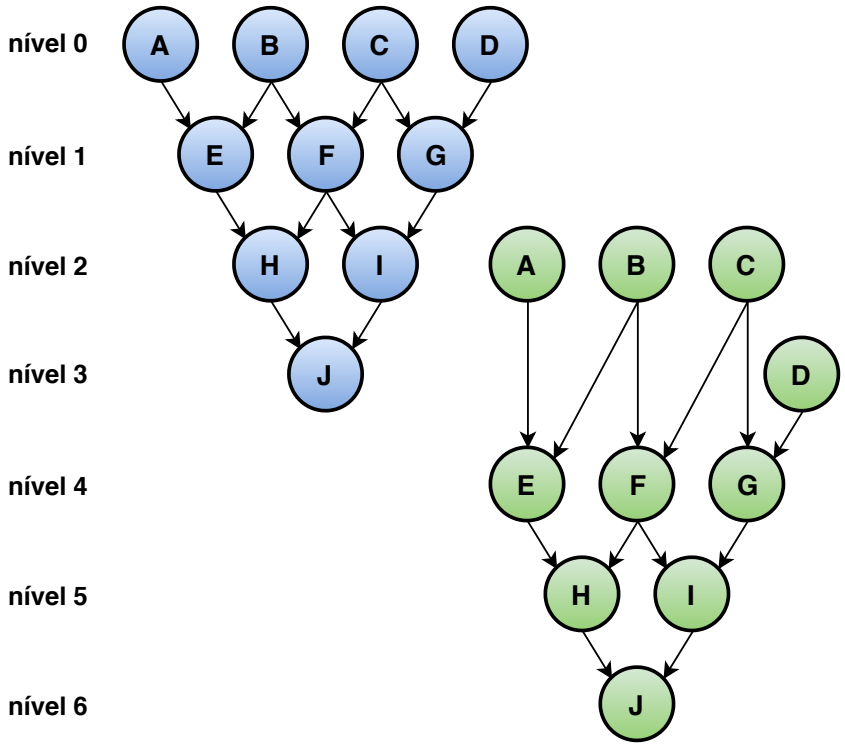

Figura 28 – Exemplo de grafo a ser executado na arquitetura.

Uma outra solução para esse problema pode ser vista na Figura 29.

Nessa solução, também será utilizada a abordagem em linhas, já comentada no trabalho. A instrução *D* presente na Figura 27 no nível 2 será realocada para o nível 3, como pode ser visto na Figura 29.

1. Mas por que mover a instrução *D* e não qualquer outra instrução presente no nível 2 na Figura 27?

   Movendo-se as instruções *H* ou *I*, perdemos a integridade do valor da instrução seguinte, pois o resultado da instrução *J* estaria errado, pois a mesma depende de *H* e *I* no nível 2. Então, pode-se mover as instruções *A*, *B*, *C* ou *D*, pois já temos os seus resultados no nível 0. A instrução D foi escolhida apenas como comodidade, o algoritmo sempre pegará as últimas instruções do nível para serem movidas, garantindo que sua execução já foi feita em níveis anteriores.

2. E pra onde mover a instrução escolhida?

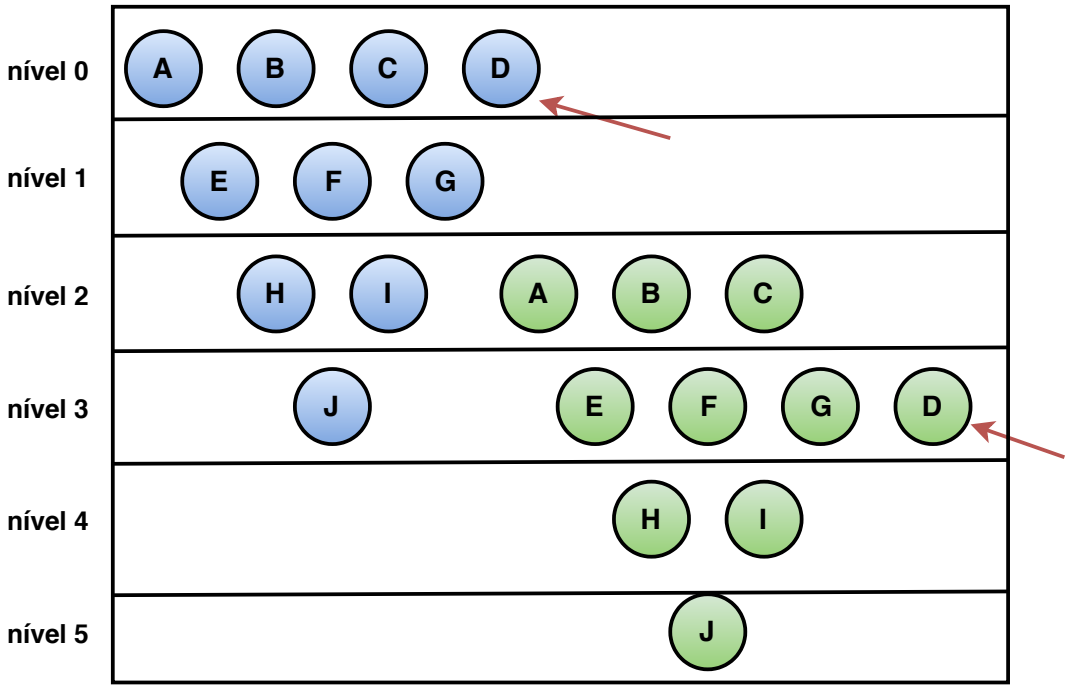

Figura 29 – Exemplo da nova realocação da instrução.

A instrução deve ser movida para um nível onde a quantidade de PEs disponíveis seja suficiente para receber as novas instruções e que as novas instruções já tenham sido executadas. Por exemplo, na Figura 29, como queremos realocar somente a instrução *D*, o melhor local possível é onde há somente 1 PE disponível, pois assim estamos buscando uma maximização do *ILP* por coluna do *array*. Perceba, que no nível 1, a instrução *D* poderia ser realocada, mas como temos 2 PEs vazios e no nível 3 temos 1 PE vazio, a melhor escolha do algoritmo é realocar para o nível 3, completando assim os 5 PEs em utilização, consequentemente, aumentando o *ILP* daquela coluna.

Então, a regra geral em casos onde a quantidade de PEs é extrapolada, é sempre mover as instruções mais a direita da coluna com instruções excedentes, buscando ocupar colunas onde a execução dessas instruções já estejam sido realizadas e que maximizem a quantidade de PEs utilizados naquela coluna que receberá as novas instruções. Utilizando essa abordagem no exemplo demonstrado, vinte instruções são executadas em seis colunas, totalizando um *ILP* igual à 3.33. Obtendo um ganho de 0.48 em relação à primeira solução. No trabalho proposto, em casos onde acontece essa extrapolação na quantidade PEs por coluna, é utilizado a segunda abordagem apresentada (Figura 29), visto que é obtido um maior *ILP* utilizando essa segunda solução.

Neste capítulo, foi demonstrado todo o sistema proposto nesse trabalho, desde a arquitetura alvo que será utilizada até a metodologia do algoritmo utilizado. O primeiro

passo do algoritmo é detectar os laços presentes na aplicação, esse processo é feito primeiramente partir da implementação da aplicação em *Go* no compilador utilizado, que gera as instruções *assembly* MIPS, desse código *assembly* os laços são detectados. O próximo passo é gerar o DFG desse laço, onde cada instrução é analisada e o DFG é gerado seguindo um determinado algoritmo. O passo seguinte é a execução do algoritmo em si, onde as partições são criadas e sobrepostas e por último alguns mecanismos especiais foram implementados para lidar com as restrições de recursos da arquitetura.

5 Resultados e Discussão

Neste capítulo serão mostrados os resultados da proposta, utilizando aplicações implementadas de diferentes *benchmarks*. As aplicações utilizadas são:

- Multiplicação de matrizes;
- *Bitcount*: Consiste em um algoritmo que conta o número de *bits* em um *array* de inteiros. Pertence ao *benchmark* ParMiBench (Iqbal; Liang; Grahn, 2010), utilizado na avaliação de desempenho de sistemas embarcados;
- LU: É uma aplicação utilizada em álgebra linear, que fatora uma matriz quadrada em duas matrizes triangulares, uma matriz triangular superior e uma inferior, de modo que o produto dessas duas matrizes forneça a matriz original. Essa aplicação pertence ao *benchmark* OpenMP (Dorta; Rodriguez; de Sande, 2005), composto por aplicações escritas em *C*, *C++* e *Fortran* que inclui um amplo espectro de situações paralelizáveis;
- *FIR* e *FIR2*: São filtros cuja respostas ao impulso é de duração finita. São aplicações que compõem o *benchmark* (EXPRESS, 2013), que consiste em um repositório selecionado entre mais de 1400 grafos de fluxo de dados obtidos de aplicações de *MediaBench Benchmark Suite*. As aplicações exploram o paralelismo de operações entre matrizes, processamento de sinais digitais e compressões de imagens;

Todas as aplicações foram implementadas na linguagem de programação *Go* e serviram como entrada para o compilador implementado em *Java* utilizado nesse trabalho, que possui como saída a aplicação no conjunto de instruções *assembly MIPS*. Na Tabela 2 é mostrada a quantidade de nós presentes no *DFG* utilizado em cada aplicação como entrada do algoritmo de MS.

Tabela 2 – Quantidade de operações utilizadas como entrada do algoritmo por aplicação.

Aplicação	**Operações**
Mult. de Matrizes	35
Bitcount	12
LU	67
FIR	61
FIR2	28

5.1 Intervalo de Iniciação

Como já falado em seções anteriores, o intervalo de iniciação (II) é o intervalo que existe entre o início de uma iteração e outra, caso o algoritmo não alcance um escalonamento válido com o mínimo intervalo de iniciação (MII), esse intervalo é incrementado e um novo escalonamento é realizado, com um novo II. Portanto, quanto menor o II, melhor a performance. Os resultados referentes ao II estão demonstrados na Tabela 3.

Tabela 3 – Intervalo de iniciação obtido pelo algoritmo.

Aplicação	**MII**	**Intervalo de Iniciação Obtido**
Mult. de Matrizes	7	7
Bitcount	3	3
LU	3	3
LU2	11	11
FIR	13	13
FIR2	6	6

Observando a Tabela 3, nota-se que o algoritmo proposto conseguiu escalonar todas as cinco aplicações utilizando o MII, caracterizando o resultado ótimo levando em consideração a arquitetura alvo do trabalho. O motivo desse ótimo escalonamento se dá a partir do processo de realocação dos nós que é feita na etapa de escalonamento das operações nos PEs da arquitetura. Essa realocação é demonstrada na Seção 4.3.4. O algoritmo proposto sempre tenta escalonar os nós com o MII, quando detecta uma falha e que deve recomeçar com um novo II, ao invés de recomeçar, ele entra na fase de realocação dos nós, que busca a melhor localização possível para o nó que acarretaria alguma falha no processo. Ainda é possível perceber que a linha com a aplicação LU2 foi adicionada, isso aconteceu por que na aplicação LU foram detectados dois laços mais internos diferentes, o primeiro laço, LU, foi mapeado com II = 3 e o segundo laço, LU2, foi mapeado com II = 11, diferentemente de FIR e FIR2 que são duas aplicações distintas.

5.2 Paralelismo

Nessa seção, os resultados referentes ao paralelismo serão apresentados. O ILP é calculado a partir da divisão entre o número de operações a serem realizadas pela quantidade de colunas do *array* reconfigurável necessárias para executar tal aplicação. Esses resultados podem ser vistos na Tabela 4.

Na Tabela 4, a coluna Operações se refere a quantidade de operações que foram geradas naquela aplicação, contando somente a partição original, a segunda coluna apre-

Tabela 4 – ILP obtido com a execução da aplicação sem o algoritmo de MS.

Aplicação	**Operações**	**Colunas**	**ILP sem MS**
Mult. de Matrizes	35	20	1.75
Bitcount	12	9	1.33
LU	12	12	1
LU2	55	22	2.5
FIR	61	56	1.09
FIR2	28	22	1.27

senta quantas colunas do *array* reconfigurável são necessárias para executar as operações e a terceira coluna é o cálculo do ILP. Nessa tabela, os resultados do ILP apresentados se referem à execução sem a utilização do *modulo scheduling*, e sim à execução da aplicação a partir do DFG, onde cada nível do DFG representa uma coluna do *array reconfigurável*, esse processo foi melhor explicado na seção 4.3.3. Esse resultado de ILP sem a execução do *modulo scheduling* foi obtido para ser utilizado como efeito de comparação com o ILP obtido a partir da execução das aplicações utilizando o algoritmo MS.

Na Tabela 5, foi adicionado o resultado do ILP obtido utilizando o algoritmo de *modulo scheduling* e a relação de ganho de paralelismo entre a execução da aplicação sem o algoritmo e com o algoritmo.

Tabela 5 – Relação de ganho em paralelismo quando se utiliza o algoritmo de MS.

Aplicação	**Operações**	**Colunas**	**ILP sem MS**	**ILP com MS**	**Ganho**
Mult. de Matrizes	35	20	1.75	2.5	42.86%
Bitcount	12	9	1.33	1.71	28.57%
LU	12	12	1	1.33	33%
LU2	55	22	2.5	2.82	12.80%
FIR	61	56	1.09	1.66	52.34%
FIR2	28	22	1.27	1.86	46.46%

É possível perceber que existe um ganho significativo em todas as aplicações utilizando o algoritmo. A aplicação que obteve um menor ganho foi o segundo laço mais interno da aplicação LU, chamada de LU2. Isso aconteceu por que a aplicação executada sem MS deve ser mapeada em 22 colunas, e quando se utiliza o MS é obtido um II = 11 e o mapeamento deve ser feito com duas partições, ou seja, a nova iteração da aplicação só será iniciada na 12ª coluna, logo as 11 primeiras colunas com instruções não serão executadas com as novas instruções pertencentes à segunda iteração, diminuindo assim o ILP médio da aplicação. Ao contrário do segundo laço mais interno da aplicação LU,

a aplicação que obteve um maior ILP foi FIR, mesmo com II = 13. Isso acontece por que diferentemente de LU, FIR foi mapeada com 4 iterações, ou seja, as instruções de quatro diferentes iterações serão executadas em paralelo em algum determinado ponto da execução da aplicação como um todo. Aumentando assim, o paralelismo médio obtido pela aplicação.

Outro resultado importante com relação ao ILP é o paralelismo obtido após a etapa de realocação dos nós. Os resultados referentes a essa etapa podem ser vistos na Tabela 6.

Tabela 6 – Relação de ganho em paralelismo quando se utiliza a realocação dos nós.

Aplicação	**ILP sem MS**	**ILP com MS**	**ILP após realoc.**	**Ganho CReal/SReal**	**Ganho CReal/SMS**
Mult. de Matrizes	1.75	2.5	3.09	23.52%	76.46%
Bitcount	1.33	1.71	1.8	5.26%	35.34%
LU	1	1.33	1.54	15.79%	54%
LU2	2.5	2.82	3.14	11.35%	25.60%
FIR	1.09	1.66	1.88	13.08%	72.27%
FIR2	1.27	1.86	1.96	5.38%	54.33%

Na tabela acima, a coluna **ILP após reloc.** se refere ao ILP obtido após a etapa de realocação dos nós, a coluna **Ganho CRel/SReal** calcula o ganho do ILP utilizando realocação de nós (coluna ILP após realoc.) em comparação ao ILP utilizando MS sem realocação de nós (coluna ILP com MS), e na última coluna os resultados de ganho das aplicações utilizando o cálculo entre o ILP executando MS com realocação (coluna ILP após realoc.) em relação ao ILP sem MS, essa última comparação é a relação de ganho final existente entre o uso ou não do algoritmo no processo de execução da aplicação.

Observando a Tabela 6, vemos que a aplicação que obteve um maior ganho na coluna **Ganho CReal/SReal** foi a multiplicação de matrizes. Isso acontece porque foi a aplicação que mais teve nós realocados, essa aplicação consiste em um DFG com 35 nós e para seu mapeamento foram necessárias 3 partições do DFG para o algoritmo de MS, totalizando 105 nós. Dos 105 nós utilizados, 22 foram realocados da sua posição inicial para diferentes posições, ou seja, quase 21% dos nós foram realocados, o que acarretou em uma melhor distribuição das operações, alcançando assim um melhor ganho no paralelismo. Diferentemente da multiplicação de matrizes, as aplicações que menos obtiveram ganho na etapa de realocação foi *bitcount* e FIR2. *Bitcount* possui 12 nós no seu DFG e três partições, totalizando 36 nós, onde somente 4 desses nós foram realocados para diferentes posições, e FIR2 possui 28 nós em seu DFG e foi mapeado em 4 partições, um total de 112 nós, desse total, somente 7 foram realocados.

Com relação ao ganho total obtido, as aplicações que mais se beneficiaram com a

utilização do algoritmo de MS foi primeiramente a multiplicação de matrizes que antes da execução do algoritmo possuia um ILP igual a 1.75 e ao término da execução possui um ILP igual a 3.09, totalizando um ganho de quase 77%. Logo atrás a aplicação FIR, que saiu de um ILP equivalente a 1.09, o que equivale há quase uma instrução por coluna, para um ILP igual a 1.88, obtendo um ganho de 72.27%.

A Figura 30 mostra um gráfico de evolução do ILP para cada aplicação, onde é possível perceber que o algoritmo atua diretamente no paralelismo existente nas aplicações, sendo a barra azul o ILP obtido sem MS, a barra vermelha o ILP obtido com MS e por último a barra amarela com o ILP obtido após a relocação dos nós, e possível perceber que a cada nova etapa o ILP acaba obtendo um ganho.

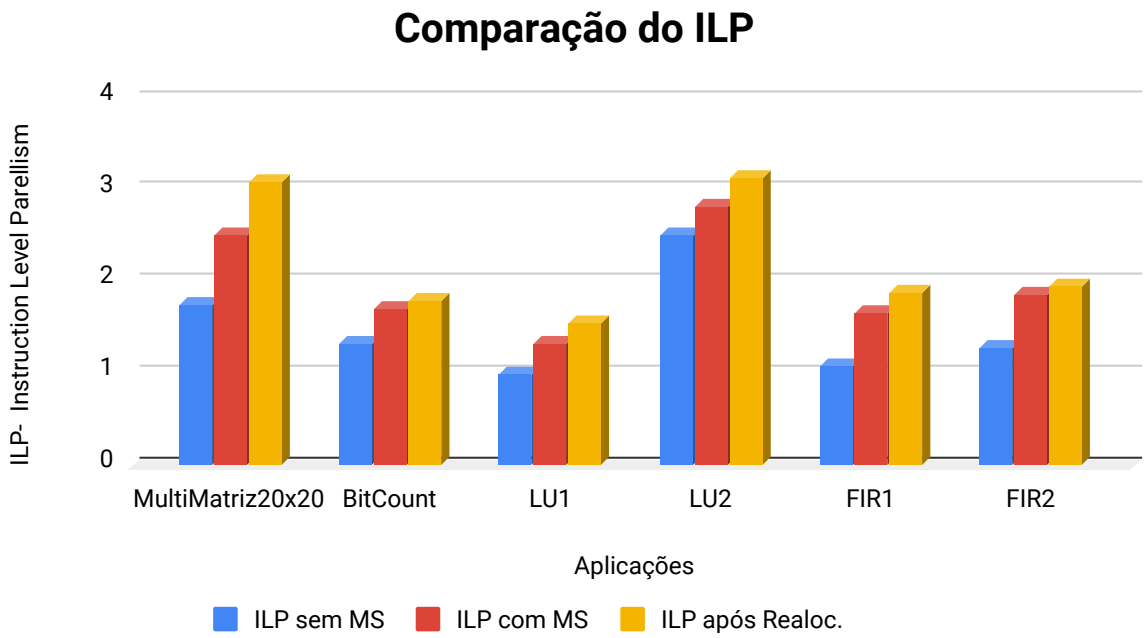

Figura 30 – Gráfico demonstrando o crescimento do ILP ao longo das etapas do algoritmo de MS.

Esses foram os resultados referentes à evolução do ILP utilizando a abordagem proposta. Para comparar esse valor com outros trabalhos da área, o cálculo que determina o ILP deve ser alterado.

Nos trabalhos que iremos comparar, o cálculo do ILP é determinado pela divisão do número de operações do grafo pelo II. Os trabalhos selecionados utilizam aplicações do *benchmark* (EXPRESS, 2013), por isso, as aplicações aqui comparadas foram o FIR1 e FIR2, que fazem parte desse mesmo *benchmark*.

O trabalho de (LOPES, 2013) e (COSTA, 2013) utilizam essa abordagem para o cálculo do ILP e serão os trabalhos comparados. Ambos trabalhos obtiveram os mesmos resultados referentes ao ILP das aplicações FIR1 e FIR2, um ILP equivalente a 7.33 para

o FIR1 e 10 para o FIR2, no trabalho aqui proposto foi obtido um ILP igual a 18.76 para o FIR1 e 19.33 para FIR2. Essa diferença acontece porque os trabalhos utilizam uma arquitetura alvo com tamanho fixo de 16 PEs, na arquitetura alvo do trabalho aqui proposto não há esse limite na quantidade de PEs total utilizados, o que acontece é um limite na quantidade de PEs por coluna do *array*, mas podendo ser utilizadas inúmeras colunas, o que é uma vantagem do trabalho aqui proposto. Essa quantidade fixa de 16 PEs mostra um bom resultado para uma arquitetura pequena, entretanto, para grafos grandes, o escalonamento de 16 PEs pode falhar, e por isso, arquiteturas maiores são necessárias.

Baseado nos resultados obtidos, é possível perceber que o algoritmo realizou um ótimo mapeamento dos nós na arquitetura, alcançando em todas as aplicações o mapeamento com o MII. Além dos resultados de mapeamento, os resultados referentes à análise do ILP também foram satisfatórios, visto que o ILP foi subindo à medida que novas etapas do algoritmos foram sendo realizadas, obtendo um bom resultado final em relação à execução das aplicações sem a utilização do *modulo scheduling*.

6 Conclusões e Trabalhos Futuros

O objetivo deste trabalho foi gerar configuração para um arquitetura reconfigurável de grão grosso utilizando o algoritmo de *modulo scheduling* (MS) atuando nos laços de computação intensiva presentes nas aplicações, com o intuito de obter um melhor paralelismo entre essas instruções a partir do algoritmo proposto. Um importante passo do trabalho é assumir o DFG gerado por um compilador como entrada no algoritmo, portanto um mecanismo especial de detecção de laços e geração do DFG foi criado a partir de um compilador, assim, o algoritmo proposto detecta, gera e organiza o laço mais interno a partir de um código binário. Outra vantagem presente no trabalho proposto está na arquitetura alvo utilizada, o problema de MS é subdividido em três outros problemas: escalonamento, posicionamento e roteamento, devido à arquitetura alvo ser conectada com um banco de registradores. No trabalho aqui proposto não é necessário a resolução do problema de roteamento, diminuindo o esforço computacional utilizado e facilitando o mapeamento das instruções nos PEs da arquitetura.

Em relação aos resultados do algoritmo. O primeiro resultado se refere à qualidade do mapeamento, que foi o ideal, em que todas as aplicações conseguiram ser mapeadas na arquitetura a partir do MII, não promovendo falhas nessa etapa do processo. Esse resultado foi obtido com a utilização da etapa de realocação de nós, em outros trabalhos da área, essa etapa muitas vezes não é implementada e o algoritmo fica iterativamente repetindo o processo de mapeamento diferentes vezes com II diferentes. Esse tipo de computação não ocorre no trabalho aqui proposto.

Olhando os resultados referentes ao ILP, fica claro que a utilização do algoritmo atua diretamente no paralelismo existente entre as instruções, e que a cada nova etapa de execução do algoritmo um novo ILP melhor foi obtido. Isso remete a um dos principais objetivos do trabalho, que é melhorar a execução de aplicações que utilizam processamento de laços de forma intensiva, em que uma melhoria nessa execução do laço melhora a execução da aplicação como um todo.

O principal problema enfrentado neste trabalho, foi conseguir fazer a comparação do trabalho com outros trabalhos que utilizam abordagens diferentes do algoritmo. No trabalho aqui proposto, todas as aplicações utilizadas tiveram que ser implementadas na linguagem de programação *Go*, portanto, enquanto os outros trabalhos utilizam quase todas as aplicações do *benchmark* (EXPRESS, 2013), que são implementadas em *C* e que possuem o código fonte disponível no ponto para compilação, aqui foram utilizadas somente FIR e FIR2 porque o esforço necessário para implementar todas as aplicações do *benchmark* de *C* para *Go* tomaria todo o tempo necessário para a realização desse trabalho

de mestrado. Além das diferentes abordagens utilizadas nos trabalhos, como arquiteturas diferentes, cálculo de ILP diferentes e etapas de mapeamento diferentes. Por isso, um foco maior foi dado na apresentação dos resultados referentes à qualidade do mapeamento realizado utilizando a arquitetura alvo e na evolução do paralelismo obtido à medida que novas soluções eram propostas, como a realocação dos nós e o *register renaming*.

Devido a isso, como trabalho futuro, pretende-se desenvolver a implementação do algorimo de *modulo scheduling* em um compilador já estabelecido, como é o caso do LLVM. Assim, todos os problemas citados no parágrafo anterior seriam superados, visto que o LLVM é um compilador utilizado mundialmente, com padrões de resultados já bem definidos. O objetivo aqui é a implementação do algoritmo na forma de representação intermediária no LLVM, onde várias otimizações podem ser feitas para melhorar essa representação.

Referências

ALLAN, V. H. et al. Software pipelining. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 27, n. 3, p. 367–432, set. 1995. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/212094.212131>. Citado na página 1.

BANDISHTI, V.; PANANILATH, I.; BONDHUGULA, U. Tiling stencil computations to maximize parallelism. In: *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. [S.l.: s.n.], 2012. p. 1–11. ISSN 2167-4337. Citado na página 14.

CALLAHAN, T. J.; WAWRZYNEK, J. Adapting software pipelining for reconfigurable computing. In: *Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. New York, NY, USA: ACM, 2000. (CASES '00), p. 57–64. ISBN 1-58113-338-3. Disponível em: <http://doi.acm.org/10.1145/354880.354889>. Citado 2 vezes nas páginas 1 e 14.

CARDOSO, J. M. P.; DINIZ, P. C. Modeling loop unrolling: Approaches and open issues. In: PIMENTEL, A. D.; VASSILIADIS, S. (Ed.). *Computer Systems: Architectures, Modeling, and Simulation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. p. 224–233. ISBN 978-3-540-27776-7. Citado na página 13.

CARR, S.; GUAN, Y. Unroll-and-jam using uniformly generated sets. In: *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 1997. (MICRO 30), p. 349–357. ISBN 0-8186-7977-8. Disponível em: <http://dl.acm.org/citation.cfm?id=266800.266835>. Citado na página 13.

CHEN, L.; MITRA, T. Graph minor approach for application mapping on cgras. *ACM Trans. Reconfigurable Technol. Syst.*, ACM, New York, NY, USA, v. 7, n. 3, p. 21:1–21:25, set. 2014. ISSN 1936-7406. Disponível em: <http://doi.acm.org/10.1145/2655242>. Citado na página 15.

COSTA, L. M. *Uma heurística gulosa para Modulo Scheduling em arquiteturas reconfiguráveis em tempo de execução*. Dissertação (Mestrado) — Universidade Federal de Viçosa, Viçosa, 2013. Citado na página 41.

Dorta, A. J.; Rodriguez, C.; de Sande, F. The openmp source code repository. In: *13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*. [S.l.: s.n.], 2005. p. 244–250. ISSN 1066-6192. Citado na página 37.

EXPRESS. Extensible, programmable and reconfig'rable embedded systems group. In: . [S.l.: s.n.], 2013. Citado 3 vezes nas páginas 37, 41 e 43.

FERREIRA, R. et al. A run-time modulo scheduling by using a binary translation mechanism. In: *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*. [S.l.: s.n.], 2014. p. 75–82. Citado na página 16.

FERREIRA, R. et al. A just-in-time modulo scheduling for virtual coarse-grained reconfigurable architectures. In: *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. [S.l.: s.n.], 2013. p. 188–195. Citado na página 16.

FERREIRA, R. et al. An fpga-based heterogeneous coarse-grained dynamically reconfigurable architecture. In: *2011 Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*. [S.l.: s.n.], 2011. p. 195–204. Citado na página 15.

GNANAOLIVU, R.; NORVELL, T. S.; VENKATESAN, R. Mapping loops onto coarse-grained reconfigurable architectures using particle swarm optimization. In: *2010 International Conference of Soft Computing and Pattern Recognition*. [S.l.: s.n.], 2010. p. 145–151. Citado na página 15.

GUPTA, S. et al. Loop shifting and compaction for the high-level synthesis of designs with complex control flow. In: *Proceedings Design, Automation and Test in Europe Conference and Exhibition*. [S.l.: s.n.], 2004. v. 1, p. 114–119 Vol.1. ISSN 1530-1591. Citado na página 13.

GUSTAFSON, J. L. Reevaluating amdahl's law. *Commun. ACM*, ACM, New York, NY, USA, v. 31, n. 5, p. 532–533, maio 1988. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/42411.42415>. Citado na página 19.

HAMZEH, M.; SHRIVASTAVA, A.; VRUDHULA, S. Epimap: Using epimorphism to map applications on cgras. In: *DAC Design Automation Conference 2012*. [S.l.: s.n.], 2012. p. 1280–1287. ISSN 0738-100X. Citado na página 15.

HARTENSTEIN, R. A decade of reconfigurable computing: a visionary retrospective. In: *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*. [S.l.: s.n.], 2001. p. 642–649. ISSN 1530-1591. Citado na página 7.

HATANAKA, A.; BAGHERZADEH, N. A modulo scheduling algorithm for a coarse-grain reconfigurable array template. In: *2007 IEEE International Parallel and Distributed Processing Symposium*. [S.l.: s.n.], 2007. p. 1–8. ISSN 1530-2075. Citado 2 vezes nas páginas 1 e 14.

HAUSER, J. R.; WAWRZYNEK, J. Garp: a mips processor with a reconfigurable coprocessor. In: *Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No.97TB100186)*. [S.l.: s.n.], 1997. p. 12–21. Citado na página 7.

Iqbal, S. M. Z.; Liang, Y.; Grahn, H. Parmibench - an open-source benchmark for embedded multiprocessor systems. *IEEE Computer Architecture Letters*, v. 9, n. 2, p. 45–48, Feb 2010. ISSN 1556-6056. Citado na página 37.

KENNEDY, J.; EBERHART, R. Particle swarm optimization. In: *Proceedings of ICNN'95 - International Conference on Neural Networks*. [S.l.: s.n.], 1995. v. 4, p. 1942–1948 vol.4. Citado na página 15.

KONG, M. et al. When polyhedral transformations meet simd code generation. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2013. (PLDI '13), p. 127–138. ISBN

978-1-4503-2014-6. Disponível em: <http://doi.acm.org/10.1145/2491956.2462187>. Citado na página 14.

LEE, M.-H. et al. Design and implementation of the morphosys reconfigurable computing processor. *Journal of VLSI signal processing systems for signal, image and video technology*, v. 24, n. 2, p. 147–164, Mar 2000. ISSN 0922-5773. Disponível em: <https://doi.org/10.1023/A:1008189221436>. Citado na página 15.

LOPES, V. D. *Uma heurística polinomial para escalonamento de loops em arquiteturas reconfiguráveis de grão grosso*. Dissertação (Mestrado) — Universidade Federal de Viçosa, Viçosa, 2013. Citado 4 vezes nas páginas 15, 25, 26 e 41.

MAHLKE, S. A. *Exploiting Instruction Level Parallelism in the Presence of Conditional Branches*. Tese (Doutorado), Champaign, IL, USA, 1997. UMI Order No. GAX97-17305. Citado na página 14.

MAHLKE, S. A. et al. A comparison of full and partial predicated execution support for ilp processors. In: *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*. New York, NY, USA: ACM, 1995. (ISCA '95), p. 138–150. ISBN 0-89791-698-0. Disponível em: <http://doi.acm.org/10.1145/223982.225965>. Citado na página 14.

MALEKI, S. et al. An evaluation of vectorizing compilers. In: *2011 International Conference on Parallel Architectures and Compilation Techniques*. [S.l.: s.n.], 2011. p. 372–382. ISSN 1089-795X. Citado na página 14.

MEHTA, S.; LIN, P.-H.; YEW, P.-C. Revisiting loop fusion in the polyhedral framework. In: *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM, 2014. (PPoPP '14), p. 233–246. ISBN 978-1-4503-2656-8. Disponível em: <http://doi.acm.org/10.1145/2555243.2555250>. Citado na página 13.

MEI, B. et al. Dresc: a retargetable compiler for coarse-grained reconfigurable architectures. In: *2002 IEEE International Conference on Field-Programmable Technology, 2002. (FPT). Proceedings.* [S.l.: s.n.], 2002. p. 166–173. Citado na página 14.

MEI, B. et al. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. *IEE Proceedings - Computers and Digital Techniques*, v. 150, n. 5, p. 255–, Sep. 2003. ISSN 1350-2387. Citado na página 14.

MUCHNICK, S. S. *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. ISBN 1-55860-320-4. Citado na página 13.

OH, T. et al. Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures. In: *Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. New York, NY, USA: ACM, 2009. (LCTES '09), p. 21–30. ISBN 978-1-60558-356-3. Disponível em: <http://doi.acm.org/10.1145/1542452.1542456>. Citado na página 15.

PARK, H. et al. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In: *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. [S.l.: s.n.], 2008. p. 166–176. Citado na página 15.

PARK, H.; PARK, Y.; MAHLKE, S. Polymorphic pipeline array: A flexible multicore accelerator with virtualized execution for mobile multimedia applications. In: *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*. New York, NY, USA: ACM, 2009. (MICRO 42), p. 370–380. ISBN 978-1-60558-798-1. Disponível em: <http://doi.acm.org/10.1145/1669112.1669160>. Citado 4 vezes nas páginas 15, 17, 1 e 2.

PATTERSON, D. A.; HENNESSY, J. L. *Computer Architecture: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990. ISBN 1-55880-069-8. Citado na página 28.

RAU, B. R. Iterative modulo scheduling: An algorithm for software pipelining loops. In: *Proceedings of the 27th Annual International Symposium on Microarchitecture*. New York, NY, USA: ACM, 1994. (MICRO 27), p. 63–74. ISBN 0-89791-707-3. Disponível em: <http://doi.acm.org/10.1145/192724.192731>. Citado 2 vezes nas páginas 2 e 14.

RAVISHANKAR, M. et al. Code generation for parallel execution of a class of irregular loops on distributed memory systems. In: *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. [S.l.: s.n.], 2012. p. 1–11. ISSN 2167-4337. Citado na página 13.

SANCHEZ, E. et al. Static and dynamic configurable systems. *IEEE Transactions on Computers*, v. 48, n. 6, p. 556–564, Jun 1999. ISSN 0018-9340. Citado na página 7.

SARKAR, V. Optimized unrolling of nested loops. In: *Proceedings of the 14th International Conference on Supercomputing*. New York, NY, USA: ACM, 2000. (ICS '00), p. 153–166. ISBN 1-58113-270-0. Disponível em: <http://doi.acm.org/10.1145/335231.335246>. Citado na página 13.

SILVA, F. C. et al. Designing a complete pipelined datapath to mips isa: Learning in pratice. In: *Microelectronics Students Forum*. Aracaju: [s.n.], 2014. Citado na página 17.

SILVA, F. C. J. *Arquitetura Adaptável Column-Based Para Processadores Multicore*. 17–26 p. Dissertação (Mestrado) — Universidade Federal do Piauí, Teresina, PI, 2017. Citado 3 vezes nas páginas 15, 17 e 18.

SINGH, H. et al. Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Transactions on Computers*, v. 49, n. 5, p. 465–481, May 2000. ISSN 0018-9340. Citado na página 5.

STITT, G.; VAHID, F. Hardware/software partitioning of software binaries. In: *Proceedings of the 2002 IEEE/ACM International Conference on Computer-aided Design*. New York, NY, USA: ACM, 2002. (ICCAD '02), p. 164–170. ISBN 0-7803-7607-2. Disponível em: <http://doi.acm.org/10.1145/774572.774596>. Citado na página 1.

STOCK, K. et al. A framework for enhancing data reuse via associative reordering. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2014. (PLDI '14), p. 65–76. ISBN 978-1-4503-2784-8. Disponível em: <http://doi.acm.org/10.1145/2594291.2594342>. Citado na página 13.

VASILACHE, N. et al. Joint scheduling and layout optimization to enable multi-level vectorization. In: *2nd International Workshop on Polyhedral Compilation Techniques*. Paris, France: [s.n.], 2012. Citado na página 14.

VASSILIADIS, S.; SOUDRIS, D. *Fine- and Coarse-Grain Reconfigurable Computing*. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2007. ISBN 1402065043, 9781402065040. Citado na página 15.

VENKATARAMANI, G. et al. A compiler framework for mapping applications to a coarse-grained reconfigurable computer architecture. In: *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. New York, NY, USA: ACM, 2001. (CASES '01), p. 116–125. ISBN 1-58113-399-5. Disponível em: <http://doi.acm.org/10.1145/502217.502235>. Citado na página 1.

VENKATARAMANI, G. et al. A compiler framework for mapping applications to a coarse-grained reconfigurable computer architecture. In: *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. New York, NY, USA: ACM, 2001. (CASES '01), p. 116–125. ISBN 1-58113-399-5. Disponível em: <http://doi.acm.org/10.1145/502217.502235>. Citado na página 15.

WALL, D. W. Limits of instruction-level parallelism. In: *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 1991. (ASPLOS IV), p. 176–188. ISBN 0-89791-380-9. Disponível em: <http://doi.acm.org/10.1145/106972.106991>. Citado na página 1.

XU, B.; ALBONESI, D. H. *Methodology for the analysis of dynamic application parallelism and its application to reconfigurable computing*. 1999. 78-86 p. Disponível em: <http://dx.doi.org/10.1117/12.359526>. Citado na página 1.

YE, Z. A. et al. Chimaera: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In: *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*. [S.l.: s.n.], 2000. p. 225–235. ISSN 1063-6897. Citado na página 7.