Evaluating a Dynamic and a Modulo Scheduling-based Static Approach for Configuration Generation in CGRA Accelerators

Lucas Fernandes Ribeiro, Francisco Carlos Silva Júnior, Ivan Saraiva Silva

***Abstract*—With the increasing complexity of the applications, there is an urgent need for solutions to improve the performance of these applications. It is noted that the loops present in some of them are responsible for up to 71% of the code execution time. So, optimizing the loop's execution it is possible to accelerate the execution of the whole application. This performance gain can be obtained with the use of software pipelining. This work proposes RENOIR, a tool that uses software pipelining technique through the implementation of the modulo scheduling algorithm, developed in software, that acts in the generation of configurations for a coarse-grained reconfigurable architecture (CGRA). RENOIR is implemented in a compiler, which receives an application and generates a code that can run on a gem5 implementation of a CGRA that includes a RISC-V microprocessor and a thin reconfigurable array. The results show that all the applications tested achieved an improvement in performance, reaching a gain of 2,32x in certain applications.**

***Index Terms*—Software pipelining, Modulo scheduling, Loops, Reconfigurable Architectures.**

I. Introdução

Devido ao aumento na complexidade das aplicações nos últimos anos, cada vez mais são necessárias soluções que atuem na melhora de desempenho dessas aplicações. Uma forma que tem sido considerada para a obtenção de mais desempenho é a paralelização de laços das aplicações. No trabalho proposto por Park [1] foram analisadas três aplicações multimídia 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 paralelizáveis. A análise dessas aplicações mostrou que, em média, 35% dos laços são paralelizáveis.

A partir da identificação de laços paralelizáveis é possível obter maior desempenho quando da execução das aplicações com o uso de *software pipelining* (SWP) [2]. A técnica de SWP [3] propicia a exploração do paralelismo presente nos laços internos de computação intensiva das aplicações. Essa técnica consiste em sobrepor a execução de instruções de múltiplas iterações de um laço. Ao aplicar a técnica de SWP, o objetivo é alcançar um intervalo mínimo entre o início das diferentes iterações. Nesse caso, uma iteração pode iniciar antes mesmo do fim da iteração anterior. Ainda em [1], mediu-se o tempo de execução gasto nos laços paralelizados utilizando SWP e no restante do código, comparando a execução em uma arquitetura VLIW (*Very Long Instruction Word*) e em uma arquitetura reconfigurável de granularidade grossa (CGRA - *Coarse-Grained Reconfigurable Architecture*). Em todas as aplicações, o tempo total na execução dos laços foi menor na CGRA, mostando a efetividade da utilização de CGRAs na execução de laços e que eles são responsáveis por até 71% do tempo de execução do código. Uma forma de aplicar a técnica de SWP é chamada de *modulo scheduling* (MS) [4], que consiste, basicamente, em um algoritmo que tenta sucessivamente mapear o laço na arquitetura alvo, levando em consideração o intervalo mínimo entre o início das diferentes iterações do laço.

Trabalhos recentes com arquiteturas não convencionais buscam também geração otimizada de código, baseada em compiladores específicos. Em [5], um último nível de compilação realiza a exploração do paralelismo inerente a arquitetura proposta. No trabalho aqui apresentado, optou-se por otimizar a execução de laços de repetição objetivando ganho de desempenho com a CGRA. Mais precisamente em relação à CGRAs, os trabalhos [6], [7], [8], [9], [10], [11], [12], [13] têm apostado em dotar os *arrays* reconfiguráveis com recursos de *hardware* capazes de, em tempo de execução, gerar configurações, principalmente, de trechos de código que representam laços. Esses trechos são então executados nas unidades do *array* reconfigurável. O principal argumento é a eliminação da necessidade de recompilação de aplicações, aproveitando o código objeto já existente.

Nos dias atuais, entretanto, a atualização remota e dinâmica de aplicações para sistemas embarcados já é largamente discutida [14], [15]. Nesse contexto, uma das contribuições deste trabalho é, utilizando um *benchmark* que inclui muitas aplicações largamente utilizadas em sistema embarcados, a Mibench *suite* [16], [17], [18], é colocar em discussão a importância de, tanto dispor de CGRAs autônomas e dinâmicas, tal como em [19] e muitas outras que podem ser encontradas em [20], quanto de compiladores capazes de explorar de forma eficientes os recursos do *hardware*. Assim, uma comparação é feita entre o desempenho obtido pela geração dinâmica, em *hardware*, de configurações e pela geração de configurações com o auxílio de um compilador baseado em *modulo scheduling*.

Por fim, muitos artigos já discutem a significativa contribuição que representaria a utilização de CGRAs em sistemas embarcados, quer seja pelo desempenho oferecido, quer seja pela economia de energia. Neste trabalho, ao mostrarmos que o compilador proporciona melhor exploração dos recursos de *hardware* que os geradores em *hardware* de configurações,

Lucas Fernandes Ribeiro, UnB, lucasrib.ufpi@gmail.com
Francisco Carlos Silva Júnior, UnB, juninho.ufpi@hotmail.com
Ivan Saraiva Silva, UFPI, ivan@ufpi.edu.br

ressaltamos a contribuição de CGRAs/Compiladores para sistemas embarcados.

Neste artigo, apresenta-se a proposta e implementação de RENOIR (A Tool for Configu**R**ation G**E**neration Usi**N**g M**O**dulo Schedul**I**ng Algo**R**ithm), uma ferramenta para detecção, geração e mapeamento de laços em uma CGRA, que utiliza *modulo scheduling* com o objetivo de acelerar as aplicações utilizadas. RENOIR foi implementada em COGNITE, um *framework* para construção de geradores de código para arquiteturas multi-core [21]. RENOIR usa como ponto de partida o grafo de dependências entre as instruções (DFG - *Data-flow Graph)* do laço mais interno da aplicação, logo, um mecanismo de detecção de laços e geração do DFG foi desenvolvido. A CGRA alvo da implementação de RENOIR foi ATHENA [19]. Neste trabalho, a geração de código feita por RENOIR é comparada com a geração realizada por um bloco de *hardware* para Geração de Configuração Dinâmica (GCD); essa comparação é realizada levando em consideração o desempenho e o paralelismo, obtidos a partir desses trechos de códigos executados por ATHENA.

O restante deste trabalho está organizado da seguinte forma: a Seção II apresenta os principais conceitos relacionados ao tema proposto pelo trabalho; na Seção III, o estado da arte é analisado; na Seção IV, o sistema proposto é apresentado; na Seção V, os experimentos e resultados são apresentados e analisados, juntamente com a metodologia utilizada e, por último, na Seção VI, a conclusão e trabalhos futuros.

II. CGRAs e Modulo Scheduling

Esta seção relata um resumo do referencial teórico deste trabalho, apresentando conceitos introdutórios sobre CGRAs e *modulo scheduling*.

A. Arquiteturas Reconfiguráveis de Granularidade-Grossa

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. ARs são sistemas integrados que permitem customização da unidade de *hardware* para satisfazer os requisitos computacionais de diferentes aplicações [22]. As ARs podem variar: i) com relação aos tipos de elementos de processamento (EPs), onde as operações são realizadas, ii) tipos de rede de interconexão, responsável pela comunicação entre os EPs e iii) granularidade, determinada pelo tamanho do dado que pode ser operado pelos EPs; tal granularidade influencia diretamente a memória que armazena as configurações geradas para a AR. CGRAs surgiram com o principal objetivo de reduzir a complexidade e o tempo de configuração, posicionamento (responsável por alocar operações aos EPs) e roteamento (responsável pela conexão entre os EPs) das arquiteturas de granularidade fina.

B. Modulo Scheduling

A abordagem *modulo scheduling* para geração de configurações para ARs surge da técnica de SWP. A técnica SWP consiste em paralelizar a execução de laços, sobrepondo iterações sem ter que esperar as iterações anteriores terminarem, executando várias iterações simultaneamente. O MS é um dos recursos utilizados para aumentar o ILP (*Instruction Level Parallelism*) em laços de computação intensiva e é caracterizado como a combinação de três subproblemas: escalonamento, posicionamento e roteamento. O problema de escalonamento caracteriza-se pela identificação das dependências entre as operações e pela atribuição do tempo em que as operações são executadas; o posicionamento é a etapa que determina em qual elemento de processamento serão alocadas as operações e, por último, o roteamento é responsável pela conexão entre os elementos de processamento. Durante essas etapas, 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.

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. Já 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 é violada. Entre o início de duas iterações sucessivas há um intervalo mínimo, chamado de intervalo de iniciação (II). Portanto, uma nova iteração do laço deve ser iniciada a cada II ciclos. 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 Fig. 1, a arquitetura demonstrada na Fig. 1a possui quatro elementos de processamento e deve executar o grafo representado na Fig. 1b utilizando *modulo scheduling*. O primeiro passo é calcular o mínimo intervalo de iniciação (MII) (Esse cálculo será apresentado em seções posteriores). Nesse caso, o MII é representado por (II) na Fig. 1c. O objetivo aqui é demonstrar a execução somente do corpo do laço, também denominado de *Kernel*, que se caracteriza como uma condição estacionária de estado, alcançada quando todas as instruções do laço são executadas simultaneamente por conta das sucessivas iterações. Nota-se que, do ponto de vista das dependências, a segunda iteração poderia ser inicializada junto com o nó D da primeira, entretanto essa iniciação acarretaria uma execução incorreta, visto que D estaria sendo executado duas vezes dentro do *Kernel*. 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. Cada nível do grafo deve ser executada em uma partição da arquitetura e cada partição representa uma execução da arquitetura no tempo; diante disso, duas partições devem ser utilizadas para executar as instruções do *kernel*, como mostra a Fig. 1d. Na primeira partição, o nível do grafo com a instrução D é executado e, na partição seguinte, o nível

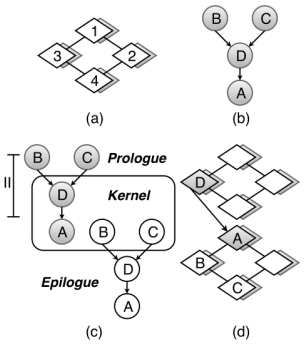

Fig. 1. Exemplo da execução do MS. (a) Arquitetura alvo (b) DFG inicial (c) Execução (d) Mapeamento do DFG na arquitetura.

com as instruções A, B e C é executado.

III. Trabalhos Relacionados

O primeiro trabalho publicado referente ao uso de *modulo scheduling* em CGRAs foi em [23], que aplicou o algoritmo para uma AR denominada Garp [24], obtendo bons resultados. Mas foi em [6] onde a utilização do MS resultou em um impacto significativo na área de pesquisa de geração de código para CGRA, com o desenvolvimento do compilador DRESC (*Dynamically Reconfigurable Embedded System Compiler*). Esse compilador é capaz de analisar, transformar e organizar código fonte da linguagem C para uma família de ARs compatíveis com o compilador. O algoritmo *modulo scheduling* foi implementado com o uso da meta-heurística *simulated annealing*, que, a cada iteração, melhora o código gerado, aprimorando a utilização de recursos da CGRA, entretanto, apesar de gerar um bom escalonamento de operações nas unidades do *array* reconfigurável, obtendo frequentemente o mínimo 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.

Em [7], a heurística de busca PSO (*Particle Swarm Optimization*) foi utilizada para atribuir iterativamente operações a recursos e associar um custo a esse mapeamento. O problema com essa abordagem é que o processo de busca possui um longo tempo de execução. Já no trabalho apresentado em [8] propõe-se EMS (*Edge-Centric Modulo Scheduling*), que reduziu o tempo de compilação em comparação a outros trabalhos da área. Enquanto DRESC e outros algoritmos de MS 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, EMS apresenta intervalos de iniciação maiores, reduzindo a qualidade do escalonamento. Ainda utilizando a abordagem de EMS (orientar o posicionamento através do roteamento), o trabalho proposto em [9] ganha em tempo de compilação, comparado com DRESC. Além disso, atinge escalonamentos melhores que EMS, mas, por outro lado, é duas vezes mais lento que ele.

EPIMap [10] propôs o que os autores chamaram de recomputação para resolver os problemas de posicionamento. Na recomputação, um nó do grafo de operações 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 obtido por EMS. Na proposta apresentada em [11], foi 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 recursos do *array* reconfigurável, alcançando em média 62% de ocupação de UFs (unidades funionais) contra 54% de DRESC. Em [11], nenhuma comparação é feita com EMS de modo que seja possível estabelecer uma comparação a partir de leitura e análise. Em [12], é proposta uma solução que fornece aceleração na fase de mapeamento baseada em três mecanismos: i) uma heurística de *modulo scheduling*, ii) uma rede *crossbar* e iii) uma CGRA virtual, mapeada em uma FPGA (*Field-Programmable Gate Array*), obtendo uma qualidade de escalonamento similar a EPIMap, alcançando a solução ótima em alguns casos. Já em [13], é proposta a aplicação de tradução binária com a utilização de *modulo scheduling*. Nessa abordagem, o ponto de partida é o grafo de fluxo de dados (DFG) do laço; assim, é necessário um mecanismo especial de detecção de laços.

De acordo com os autores em [8], o problema de roteamento é uma etapa que consome um alto tempo no mapeamento em CGRAs. Neste trabalho, é utilizado um banco de registradores como recursos de roteamento, eliminando os problemas referentes ao roteamento entre os recursos, diminuindo o tempo de execução do algoritmo e melhorando a qualidade do mapeamento. Conforme o trabalho proposto em [12], o uso de uma rede *crossbar* em vez de uma malha reduz a complexidade do mapeamento das aplicações e, consequentemente, propicia uma rápida fase de mapeamento. Seguindo esse raciocínio, a AR alvo deste trabalho também possui rede *crossbar*, mas, diferentemente de [12], o *crossbar* é utilizado na conexão de colunas do *array* reconfigurável, e não entre EPs. Assim como em [13], o grafo utilizado como ponto de partida é um DFG; diante disso, um mecanismo de detecção de laços foi desenvolvido para essa geração. Com a utilização do DFG, o problema escalonamento fica mais simples de se resolver, visto que o DFG mantém a ordem das operações seguindo a dependência de dados entre elas.

IV. A Arquitetura Alvo e a Proposta de MS para Geração de Configurações

A proposta deste trabalho é o desenvolvimento de uma ferramenta que utiliza o algoritmo *modulo scheduling* implementado, em *software*, a partir de um compilador disponível. O principal objetivo do mecanismo de MS desenvolvido é geração de configurações para uma CGRA alvo e estudar o desempenho resultante da configurações geradas, juntamente com o ILP das aplicações que serão executadas.

A. Arquitetura Alvo

A arquitetura reconfigurável alvo do trabalho [19] é composta por três tipos de unidades funcionais: i) elementos de

processamento (EP), divididos em duas colunas interconectadas por um *crossbar*, cada coluna possuindo três elementos de processamento; ii) um multiplicador inteiro e iii) uma unidade de *Load/Store*. Como a maioria das operações na AR são baseadas em registrador, um banco de registradores global foi adicionado, com a finalidade de armazenar os resultados das operações. Cada EP possui uma ULA (Unidade Lógica e Aritmética) e suas entradas veem do banco de registradores global ou de outro EP, através do *crossbar*. A ULA utilizada no EP é mais simples que a ULA utilizada pelo processador, pois só implementa as operações inteiras mais comuns, como *add, addi, sub, sll*, etc. Devido à essa simplicidade dos EPs, a ATHENA consegue executar até duas instruções dependentes em um único ciclo do processador. A unidade de *load/store* tem dois ciclos de latência, considerando um *cache hit*. Caso ocorra um *cache miss*, a latência vai ser a latência para resolução do *cache miss*. Por fim, o multiplicador possui uma latência de três ciclos. ALLEGRI [25] (***A*** ***L****ightweight* *Pipe****L****ined r****E****confi****G****urable a****R****ch****I****tecture*) é uma variação desta CGRA, composta por uma única coluna de EPs que são explorados através da alocação das operações em diferentes ciclos.

O *array* reconfigurável é integrado a um processador superescalar RISC-V (*Reduced Instruction Set Computer*-V) ao qual foi acoplado um módulo de *hardware* para Geração de Configuração Dinâmica (GCD), que mapeia dinamicamente trechos de código para serem executados no *array* reconfigurável. Uma cache de configuração é usada para guardar as configurações geradas pelo GCD. O sistema geral e o *array* reconfigurável podem ser vistos na Fig. 2. No trabalho aqui proposto, a execução da geração de configurações realizadas por RENOIR será comparada com a execução da geração feita pelo GCD, a fim de avaliar o desempenho de ambas as gerações executadas por ATHENA. No modelo de funcionamento da arquitetura, para cada endereço de acesso à memória de instruções (endereço armazenado no registrador PC (*Program Counter*) do processador RISC-V), verifica-se se a cache de configurações possui uma configuração para aquele trecho de código. Se a resposta for sim, a execução no processador é interrompida, o contexto de execução e a configuração são carregadas no *array* reconfigurável e a execução da configuração é iniciada. Ao final da execução da configuração, o *array* é interrompido, o contexto de execução é restaurado para o processador e o endereço armazenado no registrador PC é atualizado, retomando-se à execução no processador.

B. Proposta de MS para Geração de Configurações

Como RENOIR usa como ponto de partida o DFG, um mecanismo de detecção da laços e geração do DFG foi desenvolvido.

1) Detecção dos laços: O primeiro passo no processo de MS é a detecção dos laços presentes na aplicação. Os laços extraídos pelo compilador são caracterizados por possuírem, como endereço alvo de uma instrução de salto, um endereço já visitado, ou seja, um endereço que seja menor que ao da própria instrução de salto. O algoritmo MS atua em cima dos

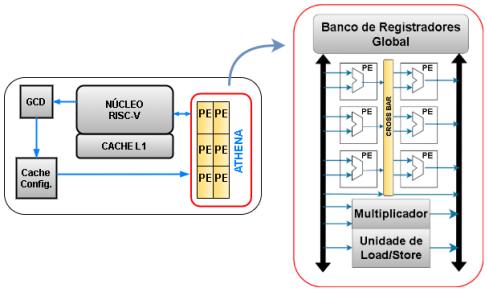

Fig. 2. Arquitetura alvo, ATHENA.

laços mais internos das aplicações, pois são tipicamente os mais executados. Portanto, é crítico tentar expor e explorar o máximo de paralelismo possível a partir desses laços.

2) Geração do grafo de dependências: Após detectados os laços presentes na aplicação, é preciso gerar o DFG. Com as instruções do laço mais interno em mãos, uma busca *bottom-up* é feita a partir da instrução com maior endereço do laço. Nessa busca, ocorre o processo de análise das dependências no qual, a cada nova instrução, se verifica se algum operando utilizado na instrução vigente está presente e/ou foi modificado em alguma das instruções anteriores, determinando as dependências de dados existentes entre essas instruções. Essas dependências podem ser de dois tipos: verdadeiras ou falsas. As dependências verdadeiras, também chamadas como dependências RAW (*Read After Write*), ocorrem quando uma instrução utiliza um operando que é produzido por uma instrução anterior. As dependências falsas classificam-se em WAR (*Write After Read*) e WAW (*Write After Write*): a primeira ocorre quando uma instrução lê em um operando que é escrito por uma instrução sucessora; já a segunda, quando uma instrução escreve em um operando que é também escrito por uma instrução sucessora.

C. Exemplo do Funcionamento do Algoritmo

Considerando o exemplo de DFG na Fig. 3, o grafo possui seis nós, equivalentes a seis instruções. Primeiramente, na Fig. 3, executado sem MS, em seguida, nas Fig. 4, Fig. 5 e Fig. 6 executado com MS. Esse processo, detalhadamente, consiste em, na Fig. 3, o nó A é executado, seguido pelo nó B, que depende do resultado de A. Então, C é executado e os nós D e E são executados em paralelo após o nó C e, por último, F é executado. Essa é a descrição do processo de execução do grafo sem a aplicação do MS. Na arquitetura alvo, cada nó do DFG (que corresponde a uma instrução) é executado em um EP de uma das colunas do *array*. Assim, os níveis 0 e 1 (compostos pelo nós A e B) são executados na primeira e segunda coluna da primeira partição (Pt0), respectivamente. Os níveis 2 e 3 (nós C, D e E) são executados na terceira e quarta coluna da segunda partição (Pt1), e o nível 4 (nó F), na quinta coluna da terceira partição (Pt2) e a última coluna da última partição é formada por EPs ociosos. Como já comentado, essas partições se referem a execuções da AR ao longo do tempo. 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

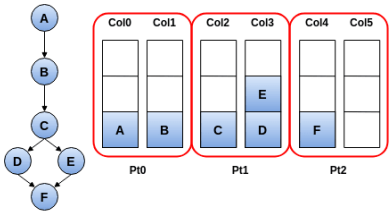

Fig. 3. Exemplo de mapeamento do DFG na arquitetura sem MS.

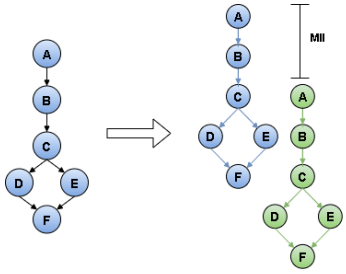

Fig. 4. Execução do grafo com o *modulo scheduling*.

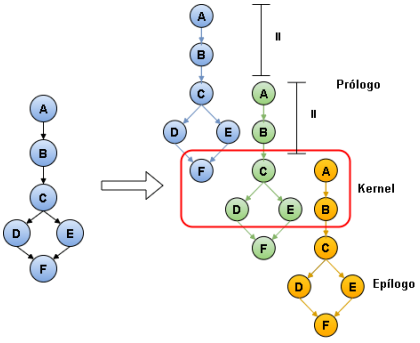

Fig. 5. Demonstração do prólogo, *kernel* e epílogo.

de operações a serem mapeadas dividido pelo número de EPs presentes por coluna; o resultado seria 6/3 = 2. Logo, o grafo será replicado 2 níveis abaixo com relação ao grafo inicial, conforme mostra a Fig. 4.

Percebe-se que, com a execução do grafo com *modulo scheduling*, ainda não é possível paralelizar todas as instruções do laço original com apenas duas sobreposições do laço, ou seja, ainda não foi criado o *kernel* desse laço. Então, uma nova iteração do grafo original deve ser criada, respeitando o MII. Na Fig. 5, uma terceira iteração do grafo original foi criada. Observa-se que, nos níveis quatro e cinco, é possível executar os seis nós do grafo original. Agora, todos os nós estão paralelizados, criando-se o *kernel*, juntamente com seu prólogo e epílogo. Esse processo envolvendo o DFG, a começar pela identificação das dependências e chegando na ordem final de como as instruções de diferentes iterações devem ser executadas, caracteriza o subproblema de escalonamento do algoritmo.

Uma configuração pode ser vista como um DFG, no qual os níveis do grafo são palavras de configuração. Uma palavra de configuração define as operações realizadas em um ciclo de execução no *array* reconfigurável. A profundidade da configuração remete à quantidade de palavras que uma configuração possui, ou seja, a quantidade de níveis define

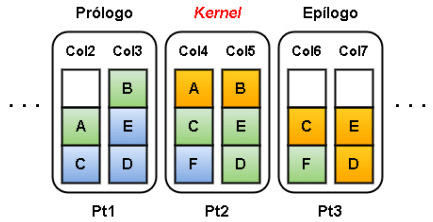

Fig. 6. Demonstração do posicionamento dos nós do grafo.

a quantidade de colunas do *array* que serão alocadas para executar tal trecho. Como já dito, os nós do grafo devem ser vinculados aos EPs do *array* reconfigurável. O processo de posicionamento dos nós em relação aos EPs que serão executados é demonstrado na Fig. 6. A primeira e a última partição (Pt0 e Pt4) foram ocultadas a fim de uma melhor visualização. Na Fig. 6, é visto, primeiramente, uma parte do prólogo, com o posicionamento dos nós dos níveis 2 e 3, executados na segunda partição (Pt1) da arquitetura. O *kernel* é então posicionado, com as seis instruções que serão executadas em paralelo na terceira partição (Pt2), seguido pelo posicionamento da primeira parte do epílogo, executado na quarta partição (Pt3), com os níveis 6 e 7. Ao final da vinculação dos nós do DFG aos EPs, o subproblema de posicionamento é concluído. Este trabalho não entra no subproblema de roteamento, que faria a ligação entre os EPs na arquitetura. Essa vantagem é implicada pela CGRA em si e pelo uso do *crossbar*. O roteamento entre os EPs dentro da mesma partição é realizado através do *crossbar* e o realizado entre diferentes partições é determinado pelo banco de registradores.

Como já falado na seção II-B, 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 mapeamento 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 elas 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), o mecanismo de *register renaming* (renomeação de registradores) [26] foi implementado no algoritmo de MS. Essa técnica consiste em renomear os registradores, aumentando o paralelismo disponível, atuando na solução das falsas dependências, seja *write after read* (WAR) ou *write after write* (WAW). No que se refere ao tratamento das restrições de recursos, em especial, os recursos referente à operações na memória, o algoritmo proposto neste trabalho segue uma abordagem conservadora, pois a ATHENA, AR alvo, não possui unidade de desambiguação de endereços de memória e, portanto, não consegue, em tempo de execução, detectar dependências de memória. Por esse motivo, para assegurar a corretude da execução na CGRA, a alocação das operações de memória

são feitas de forma conservativa. A abordagem conservativa consiste em executar os *stores* na ordem do programa. As operações de *load* podem ser reordenadas entre si, pois não há problemas de dependência de memória, mas não podem ser movidas para antes de um *store*. Isso é uma limitação da arquitetura alvo e não do algoritmo proposto. Em relação às operações lógicas e aritméticas, no caso de falta de recurso, as operações são movidas para execução no primeiro EP ou multiplicador disponível nas próximas colunas.

V. Resultados

Nesta seção são apresentados os experimentos e resultados referentes ao desempenho e o paralelismo obtido a partir da execução das aplicações. Foram selecionadas nove aplicações do *benchmark* Mibench [16], que consiste em uma série de aplicações disponíveis com o código fonte em C, divididas em seis categorias, cada categoria possuindo um alvo específico na área de sistemas embarcados, que são: automação e controle industrial, dispositivos de consumo, automação de escritório, redes, segurança e telecomunicações.

A fim de analisar a relevância dos laços para as aplicações selecionadas, na Fig. 7, os laços de cada aplicação foram organizados pela sua contribuição ao tempo de execução, mostrando quantos laços são necessários para atingir uma determinada taxa do tempo de execução. Os laços enumerados de 1 a 10 são os 10 mais importantes em termos de tempo de execução. Na Fig. 7, é possível perceber que, na aplicação Susan S, a execução de apenas 1 laço corresponde a quase 90% do tempo de execução total da aplicação, enquanto os demais quase não afetam o tempo total da execução, visto que a linha do gráfico se mantém quase constante, ou seja, Susan S passa quase 90% do tempo na execução de laços, em que, nessa aplicação, somente 1 laço executa, de fato, todo esse tempo.

Outra aplicação que passa a maior parte do tempo executando laços é SHA (*Secure Hash Algorithm*), mas, diferentemente de Susan S, em que 1 laço utiliza quase todo o tempo de execução, em SHA, esse tempo gasto acontece gradualmente à medida que a quantidade de laços aumenta, chegando a 80% do tempo total de execução da aplicação, quando se acumula o tempo executando dos 8 laços mais importantes. Por outro lado, aplicações como FFT (*Fast Fourier Transform*) e Jpeg D quase não gastam o tempo de execução em laços, visto que a execução dos laços mais relevantes representa apenas 2,6% do tempo total de execução em FFT e 1,1% em Jpeg D. Essa diferença acontece devido à natureza da aplicação em si; em algumas aplicações, todo o processo computacional acontece dentro de laços e em outras não. Isso implica que algumas aplicações vão se beneficiar mais do uso de um *array* reconfigurável para aceleração de laços.

No estudo de aceleração de laços em ARs, aplicações que concentram a maior parte do tempo de execução em poucos laços são mais adequadas, pois todo o esforço do sistema reconfigurável é concentrado em um grupo específico. Essa avaliação mostra que os laços dessas aplicações consomem uma boa parte do tempo de execução; logo, utilizando o algoritmo de *modulo scheduling*, podemos obter um ganho em performance geral na aplicação.

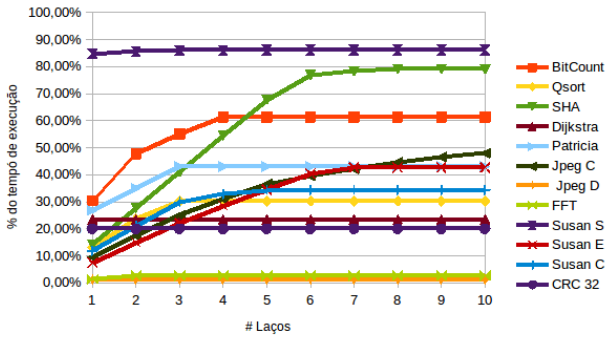

Fig. 7. Relevância do tempo de execução nos laços de cada aplicação.

A. Avaliação de Desempenho

Com o objetivo de avaliar o desempenho das aplicações, foi utilizado o simulador gem5 [27], visto que a AR alvo do trabalho foi implementada nesse simulador. O processador usado é um superescalar fora de ordem 8-*issue* e a ISA RISC-V é utilizada como conjunto de instruções, dada a sua alta aceitação tanto na academia, como na indústria.

A comparação de desempenho foi realizada levando em consideração três tipos de execuções: i) somente pelo processador (*standalone*), ii) com a AR utilizando a geração dinâmica de configuração (ATHENA + GCD) e iii) com a AR utilizando a geração de RENOIR (ATHENA + RENOIR). A primeira comparação é normalizada pela execução do processador (*standalone*), apresentada na Fig. 8; assim, é possível visualizar o desempenho utilizando as gerações dos dois aceleradores, em relação a uma execução sem qualquer acelerador (*standalone*). Percebe-se que a execução ATHENA + RENOIR foi a que obteve um maior desempenho, em especial, as aplicações bitcount, SHA e Susan S. Outra comparação importante se refere à execução utilizando a geração de RENOIR normalizada pela execução ATHENA + GCD, na Fig. 9. Assim, é obtida a relação de ganho de desempenho da abordagem aqui proposta em relação a algo já acelerado.

Percebe-se que bitcount foi a aplicação que mais obteve ganho em desempenho, obtendo 2,32x de *speedup* em relação à versão *standalone* e 1,81x em relação à ATHENA + GCD. Isso acontece devido à quantidade de instruções que realizam operações lógicas, pois bitcount possui quase 71% dessas instruções, segundo a Tabela I. De forma semelhante, Susan S e SHA obtiveram ganhos de 1,93x e 1,95x, respectivamente, em relação à execução *standalone* e 1,48x e 1,17x, em relação à ATHENA + GCD, e ambas as aplicações também possuem uma alta taxa de operações lógicas e aritméticas realizadas, Susan S com quase 70% e SHA com quase 80%, além de possuírem uma quantidade maior de instruções por bloco básico, propiciando uma melhor exploração do paralelismo. Por outro lado, as aplicações dijkstra, fft_inverse e Jpeg C, apesar de alcançarem um ganho de *speedup* nas execuções realizadas, não foi um ganho tão significativo como as outras aplicações. Isso acontece por possuírem uma maior quantidade de instruções de acesso à memória, afetando o desempenho da execução da aplicação.

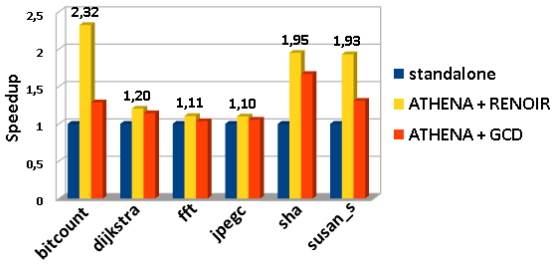

Fig. 8. Desempenho obtido normalizado pela execução *standalone*.

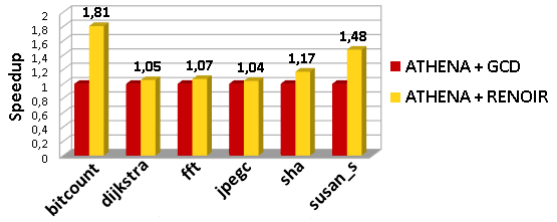

Fig. 9. Desempenho obtido normalizado pela execução ATHENA + GCD.

TABELA I
CARACTERÍSTICAS DAS APLICAÇÕES

Aplicação	% op. ULA	% Load/Store	% Saltos	ints. p/ BBs
bitcount	70,81	5,34	23,83	4,19
dijkstra	35,55	36,99	27,42	3,65
fft_inverse	44,86	29,93	18,28	5,47
jpegc	52,08	33,89	14,02	7,13
sha	78,52	15,99	5,47	18,27
susan s	67,15	24,03	8,65	11,55

B. Avaliação do Paralelismo e Intervalo de Iniciação

Outro resultado importante na avaliação das aplicações é o IPC (*Instructions Per Cycle*), que remete à quantidade de instruções que podem ser executadas simultaneamente, exibido como ILP. Para avaliar o paralelismo obtido, novamente, foram utilizados os três tipos de execuções: *standalone*, ATHENA + GCD e ATHENA + RENOIR. Na Tabela II, é possível ver os resultados referentes ao ILP das aplicações segundo essas execuções. Na quinta coluna, é apresentada a relação de ganho entre as execuções ATHENA + RENOIR e ATHENA + GCD. Com relação ao ILP, houve um aumento do paralelismo em todas as aplicações executadas. Com a utilização do algoritmo proposto de *modulo scheduling*, todas as aplicações obtiveram um ganho relevante no ILP, principalmente, as aplicações Susan S, bitcount e dijkstra, que obtiveram um maior ganho no ILP utilizando RENOIR com relação à execução utilizando GCD, com um ganho de 2,25x, 2,23x e 2,05x, respectivamente, o que também justifica o porquê das aplicações bitcount e Susan S obtivessem os maiores ganhos em desempenho na Fig. 9. Já com relação à djkstra, o que fez essa aplicação mais do que dobrar seu ILP foi a utilização da geração de RENOIR, visto que seu ILP quase não obteve aumento quando executado seguindo a geração de GCD, mesmo sendo a aplicação com a maior taxa de instruções de acesso à memória, o que mostra que houve uma boa realocação de instruções entre os EPs.

A aplicação que obteve um maior ILP quando executada seguindo a geração GCD foi SHA, mas não conseguiu aumentar esse ILP na execução ATHENA + RENOIR; isso acontece porque a distribuição das instruções na arquitetura já obteve um resultado excelente, e a geração com RENOIR não conseguiu melhorar esse resultado. Mesmo assim, o conjunto ATHENA + RENOIR conseguiu obter 5,05 de ILP, um aumento de 3,77x com relação à execução *standalone*. Comparando os resultados de qualidade de mapeamento, com aqueles obtidos por EPIMap [10], o trabalho aqui proposto, alcançou o mínimo intervalo de iniciação (MII) em 90,3% dos 52 DFGs testados. Já com EPIMap, que utiliza a técnica de recompilação, mais lenta, portanto, que a técnica utilizada neste trabalho, alcançou o MII em 93% de 14 DFGs testados.

TABELA II
ILP OBTIDO DAS APLICAÇÕES

Aplicação	cpu ILP	ATHENA + GCD	ATHENA + RENOIR	Rel. RENOIR / GCD
bitcount	0,92	2,5	5,58	2,23x
dijkstra	1,19	1,4	2,87	2,05x
fft_inverse	0,95	1,5	2,37	1,58x
jpegc	1,11	2,3	2,72	1,18x
sha	1,34	5,19	5,05	0,97x
susan s	1,56	1,57	3,53	2,25x

VI. Conclusão e Trabalhos Futuros

Este trabalho propõe uma ferramenta chamada RENOIR, que utiliza o algoritmo *modulo scheduling* nos laços presentes nas aplicações, a fim de acelerar o desempenho das aplicações utilizadas. Para análise de resultados, a execução da geração de código feita por RENOIR foi comparada com a execução da geração feita por um módulo de *hardware* que mapeia dinamicamente trechos de código a serem executados no *array* reconfigurável, chamado GCD. Essa comparação teve como base a execução das aplicações pelo processador RISC-V. Os resultados de desempenho mostraram que a execução ATHENA + RENOIR obteve um ganho de desempenho médio de 1,27x em relação à execução ATHENA + GCD e 1,6x em relação à execução somente pelo processador. Outra comparação realizada foi a avaliação do paralelismo da geração utilizada. A geração ATHENA + RENOIR conseguiu chegar a um paralelismo médio de 1,71x em relação à ATHENA + GCD, alcançando mais que o dobro de paralelismo em três das seis aplicações avaliadas. Como trabalhos futuros, estudos estão sendo realizados buscando a implementação do algoritmo de MS proposto em RENOIR diretamente no simulador gem5.

Referências

[1] H. Park, Y. Park, and S. Mahlke, "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*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 370–380. [Online]. Available: http://doi.acm.org/10.1145/1669112.1669160

[2] A. Hatanaka and N. Bagherzadeh, "A modulo scheduling algorithm for a coarse-grain reconfigurable array template," in *2007 IEEE International Parallel and Distributed Processing Symposium*, March 2007, pp. 1–8.

[3] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan, "Software pipelining," *ACM Comput. Surv.*, vol. 27, no. 3, pp. 367–432, Sep. 1995. [Online]. Available: http://doi.acm.org/10.1145/212094.212131

[4] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, ser. MICRO 27. New York, NY, USA: ACM, 1994, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/192724.192731

[5] J. V. Couto and S. R. Fernandes, "Generating optimized code for parallelism exploitation to an unconventional architecture," *IEEE Latin America Transactions*, vol. 15, no. 10, pp. 1967–1976, 2017.

[6] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "Dresc: a retargetable compiler for coarse-grained reconfigurable architectures," in *2002 IEEE International Conference on Field-Programmable Technology, 2002. (FPT). Proceedings.*, Dec 2002, pp. 166–173.

[7] R. Gnanaolivu, T. S. Norvell, and R. Venkatesan, "Mapping loops onto coarse-grained reconfigurable architectures using particle swarm optimization," in *2010 International Conference of Soft Computing and Pattern Recognition*, Dec 2010, pp. 145–151.

[8] H. Park, K. Fan, S. Mahlke, T. Oh, H. Kim, and H. s. Kim, "Edge-centric modulo scheduling for coarse-grained reconfigurable architectures," in *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Oct 2008, pp. 166–176.

[9] T. Oh, B. Egger, H. Park, and S. Mahlke, "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*, ser. LCTES '09. New York, NY, USA: ACM, 2009, pp. 21–30. [Online]. Available: http://doi.acm.org/10.1145/1542452.1542456

[10] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "Epimap: Using epimorphism to map applications on cgras," in *DAC Design Automation Conference 2012*, June 2012, pp. 1280–1287.

[11] L. Chen and T. Mitra, "Graph minor approach for application mapping on cgras," in *2012 International Conference on Field-Programmable Technology*, Dec 2012, pp. 285–292.

[12] R. Ferreira, V. Duarte, W. Meireles, M. Pereira, L. Carro, and S. Wong, "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)*, July 2013, pp. 188–195.

[13] R. Ferreira, W. Denver, M. Pereira, J. Quadros, L. Carro, and S. Wong, "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)*, July 2014, pp. 75–82.

[14] K. Telschig and A. Knapp, "Towards safe dynamic updates of distributed embedded applications in factory automation," in *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2017, pp. 1–4.

[15] ——, "Synchronous reconfiguration of distributed embedded applications during operation," in *2019 IEEE International Conference on Software Architecture (ICSA)*, 2019, pp. 121–130.

[16] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, 2001, pp. 3–14.

[17] W. R. Azevedo Dias and E. D. Moreno, "Code compression using huffman and dictionary-based pattern blocks," *IEEE Latin America Transactions*, vol. 13, no. 7, pp. 2314–2321, 2015.

[18] W. R. Azevedo Dias, E. D. Moreno, and R. da Silva Barreto, "Architectural characterization and code compression in embedded processors," *IEEE Latin America Transactions*, vol. 10, no. 4, pp. 1865–1873, 2012.

[19] F. Silva Junior, J. P. S. Patrocinio, I. S. Silva, and R. P. Jacobi, "Design space exploration of a reconfigurable accelerator in a heterogeneous multicore," in *33rd Symposium on Integrated Circuits and Systems Design*, 2020.

[20] A. Podobas, K. Sano, and S. Matsuoka, "A survey on coarse-grained reconfigurable architectures from a performance perspective," *IEEE Access*, vol. 8, pp. 146 719–146 743, 2020.

[21] E. S. CARVALHO, "Cognite - um framework para construção de geradores de código para arquiteturas multi-core," Master's thesis, Universidade Federal do Piauí, Piauí, 2018.

[22] H. Singh, Ming-Hau Lee, Guangming Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho, "Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Transactions on Computers*, vol. 49, no. 5, pp. 465–481, May 2000.

[23] T. J. Callahan and J. Wawrzynek, "Adapting software pipelining for reconfigurable computing," in *Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, ser. CASES '00. New York, NY, USA: ACM, 2000, pp. 57–64. [Online]. Available: http://doi.acm.org/10.1145/354880.354889

[24] J. R. Hauser and J. Wawrzynek, "Garp: a mips processor with a reconfigurable coprocessor," in *Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No.97TB100186)*, 1997, pp. 12–21.

[25] F. Silva Junior, I. S. Silva, and R. P. Jacobi, "Evaluation and proposal of a lightweight reconfigurable accelerator for heterogeneous multicore," *IEEE Latin America Transactions*, Early Access. [Online]. Available: https://latamt.ieeer9.org/index.php/transactions/article/view/3717

[26] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990.

[27] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

Lucas Fernandes Ribeiro É graduado e mestre em Ciência da Computação pela Universidade Federal do Piauí (UFPI). Desde 2019 é aluno de Doutorado do Programa de Pós-Graduação em Informática da Universidade de Brasília (UnB). Possui experiência na área de Ciência da Computação, com ênfase em Arquitetura de Sistemas de Computação.

Francisco Carlos Silva Junior É graduado em Ciência da Computação pela Universidade Federal do Piauí (UFPI), em 2015; Mestre em Ciência da Computação pela UFPI, em 2018. Atualmente é doutorando na Universidade de Brasília (UnB), desde 2018. Atua e tem interesse na área de aceleradores e arquiteturas reconfiguráveis, processadores multicore e sistemas heterogêneos.

Ivan Saraiva Silva Possui graduação em Engenharia Elétrica e Mestrado em Engenharia Elétrica pela Universidade Federal da Paraíba. Possui Diplôme d'Études Approfondies (Mestrado) em Microélectronique et Microinformatique pela Universidade Pierre et Marie Curie (Paris VI) e Doutorado em Informática também pela Universidade Pierre et Marie Curie (Paris VI). Foi de 1996 a 2009 professor da Universidade Federal do Rio Grande do Norte (Departamento de Informática e Matemática Aplicada), sendo atualmente professor Associado IV da Universidade Federal do Piauí (Departamento de Computação). Atua nas áreas de Arquitetura de Computadores e Concepção de Sistemas integrados e Sistemas Embarcados.