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

DREAMS - Um Array Reconfigurável Dinâmico para Sistemas Multiprocessados

Francisco Carlos Silva Junior

Teresina-PI, Agosto de 2018

Francisco Carlos Silva Junior

DREAMS - Um Array Reconfigurável Dinâmico para Sistemas Multiprocessados

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 2018

Francisco Carlos Silva Junior
DREAMS - Um Array Reconfigurável Dinâmico para Sistemas Multiprocessados/ Francisco Carlos Silva Junior. – Teresina-PI, Agosto de 2018-
62 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 2018.

1. Arquitetura Reconfigurável. 2. Multicore. I. Ivan Saraiva Silva. II. Universidade Federal do Piauí. III. Centro de Ciências da Natureza. IV. Título

CDU 02:141:005.7

"Um Array Reconfigurável Dinâmico para Sistemas Multiprocessadores"

FRANCISCO CARLOS SILVA JÚNIOR

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)

Profa. Alba Cristina Magalhães Alves de Melo

(Examinadora Externa)

Prof. Kelson Rômulo Teixeira Aires

(Examinador Interno)

Prof. Raimundo Santos Moura

(Examinador Interno)

Teresina, 14 de agosto de 2018

Aos meus pais Francisco e Eliane,
por sempre estarem comigo em todos os momentos.

Agradecimentos

Agradeço primeiramente a Deus.

Agradeço aos meus pais, Francisco e Eliane, por todo o apoio, sem eles a realização deste mestrado não seria possível.

À minha irmã, Enaile, mesmo que distante, sempre me apoia e está ao meu lado.

À minha namorada, Valeska, por todo o apoio e carinho.

Agradeço ao meu orientador e amigo, Ivan Saraiva, por todos os conselhos, pela paciência, conhecimento compartilhado e companheirismo nesse período.

Aos meus amigos que vêm desde a graduação: Felipe, Renato, Rodolfo e Luís.

Aos meus amigos que estão juntos comigo desde de meu ingresso na graduação: Neto, Hugo, Luís Guilherme (GT) e Zé Neto.

Aos companheiros de laboratório: Laysson, Ramon e Joninha.

À meu amigo Lucas Fefe, parceiro de pesquisa e companheiro.

Aos professores por todos os ensinamentos, em especial, os parceiros, e fregueses, de tênis Kelson e Vinícius.

Finalmente, agradeço a todos que diretamente ou indiretamente contribuíram para o sucesso deste trabalho.

"Anything software can do, hardware can do better."

Resumo

Observa-se nos dias atuais que os sistemas embarcados estão cada vez mais heterogêneos. Diferentes funcionalidades são integradas em um mesmo dispositivo. Juntamente com essa heterogeneidade, as aplicações executadas nesses dispositivos estão cada vez mais complexas. Processadores convencionais (Processadores de Propósito Geral ou Processadores de Aplicação específica) são capazes de fornecer desempenho ou flexibilidade, mas não ambos. Nesse cenário, há uma busca por soluções arquiteturais que possam fornecer desempenho e maior flexibilidade aos dispositivos processantes. Arquiteturas reconfiguráveis já se mostraram como uma solução arquitetural com maior flexibilidade que os processadores convencionais e capazes de aumentar desempenho em ambientes *monocore*. Contudo, nos dias atuais, as arquiteturas *multicore* são dominantes no mercado de processadores. Com isso, é necessário rever o modo como as arquiteturas reconfiguráveis são concebidas e utilizadas. Tradicionalmente, em ambientes *monocore*, uma arquitetura reconfigurável (geralmente formada por um *array* de unidades funcionais) é acoplada ao processador, gerando-se considerável custo em área adicional. Portanto, migrar as arquiteturas reconfiguráveis tradicionais para um ambiente *multicore* geraria grande *overhead* de área. Dentro deste contexto, esta dissertação de mestrado propõe e avalia uma arquitetura reconfigurável para processadores *multicore* denominada DREAMS. A arquitetura proposta possui um *array* reconfigurável que é compartilhado entre os quatro núcleos de processamento. O *array* reconfigurado é organizado em 4 colunas reconfiguráveis que são compostas por três elementos de processamento e uma unidade de acesso à memória. A arquitetura proposta oferece recursos computacionais para acelerar múltiplas *threads* (ou processos) executando simultaneamente em diferentes núcleos de um processador *multicore*. Essa arquitetura inclui um tradutor binário que converte, em tempo de execução, sequências de instruções executadas em um núcleo do processador, para serem executadas na arquitetura reconfigurável. O tradutor binário provê compatibilidade de software e faz o mecanismo de reconfiguração da arquitetura ser totalmente transparente. A arquitetura proposta foi implementada utilizando a biblioteca *systemC* e a linguagem de programação C++. Sua validação foi realizada através de simulações utilizando quatro aplicações selecionadas. Para análise de desempenho, foi feita uma comparação do tempo de execução das aplicações em um *multicore* com e sem a arquitetura DREAMS. Também foi feita uma análise do código gerado por dois compiladores: *cross compiler* GNU GCC e um compilador desenvolvido no laboratório CESLa (*Circuits and Embedded System Lab*) do departamento de computação da Universidade Federal do Piauí. Os resultados obtidos mostram que a arquitetura reconfigurável acelera, em média, 39% utilizando o código gerado pelo *cross compiler* GNU GCC e 28% utilizando o código gerado pelo compilador desenvolvido no laboratório CESLa.

Palavras-chaves: Arquitetura reconfigurável. Processador multicore. Tradutor binário. Sistemas embarcados. Arquitetura de Computadores. MIPS.

Abstract

Nowadays, embedded systems are becoming more heterogenous. Different functionalities are integrated in the same device. Along with this heterogeneity, the applications run on these devices are getting more complex. Conventional processors (General Purpose Processor and Application Specific Integrated Circuit) are able to provide performance or flexibility, but not both. In this scenario, there is a quest for architectural solutions that can provide higher performance and flexibility. Reconfigurable architectures have already shown to be an architectural solution able to provide higher performance and flexibility than conventional processors in single core environment. Nevertheless, multicore processors are dominant in the market of processors nowadays. With this, it is necessary to review the way reconfigurable architectures are designed and used. Tradicionally, in a single core environment, the reconfigurable architecture (usually composed by an array of funcional units) is attached to the processor, which implies in an area overhead. Therefore, integrating the tradicional reconfigurable architectures to multicore processors would be prohibitive in terms of area overhead. In this context, this Msc dissertation proposes and evaluate a reconfigurable architecture for multicore processors named DREAMS. The proposed architecture has one reconfigurable array, which is shared among the processors' cores. The reconfigurable array is organized in four reconfigurable columns that are composed of 3 processing elements and 1 load and store unit. The proposed architecture offers computacional resources for speeding up threads running simultaneously on different cores of the processor. This architecture includes a binary translator that translates, at run time, a sequence of instructions run on a core processor to be run on our reconfigurable architecture. The binary translator provides software compatibility and allows the reconfiguration mechanism to be transparent. The proposed architecture was implemented using the systemC library and C++ programming language. The validation was performed through simulations, using four selected applications. In order to evaluate the performance, a comparison between the execution in a multicore processor with and without the DREAMS architecture was performed. The applications were compiled with two compilers: cross compiler GNU GCC and a compiler developed in the CESLa laboratory at UFPI. The results show that the DREAMS architecture speeds up the application, in average, 39% using the cross compiler GNU GCC and 28% using the compiler developed in the laboratory.

Keywords: Reconfigurable architecture. Multicore processor. Binary translator. Embedded system. Computer architecture. MIPS.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

AR	Arquitetura Reconfigurável
ASIC	*Application Specific Integrated Circuits*
CAD	*Computer Aided Desing*
CESLA	*Circuits and Embedded System Lab*
CGRA	*Coarse-Grained Reconfigurable Architectures*
CMP	Chip Multiprocessado
DAP	*Dynamic Adaptative Processor*
DFG	*Data-Flow Graph*
DIM	*Dynamic Instruction Merging*
DREAMS	*Dynamic Reconfigurable Architecture for Multicore Systems*
DSP	*Digital Signal Processor*
EDP	*Energy Delay Product*
EP	Elemento de Processamento
FGRA	*Fine-Grained Reconfigurable Architecture*
FPGA	*Field Programmable Gate Array*
FSM	*Finite State Machine*
GPP	*General Purpose Processor*
HDL	*Hardware Description Language*
ILP	*Instruction-Level Parallelism*
ISA	*Instruction Set Architecture*
LUT	*Look-Up Table*
PC	*Program Counter*
RAW	*Read After Write*

RFUOP	*Reconfigurable Functional Unit Operation*
RPU	*Reconfigurable Processing Unit*
RTL	*Register-Transfer Level*
SPL	*Specialized Programmable Logic*
TB	Tradutor Binário
TLP	*Thread Level Parallelism*
TLM	*Transaction-Level Modeling*
UFR	Unidade Funcional Reconfigurável
ULA	Unidade Lógica e Aritmética
WAR	*Write After Read*
WAW	*Write After Write*

Sumário

1 Introdução

Historicamente, o aumento na complexidade das aplicações tem exigido cada vez mais desempenho dos dispositivos processantes. Para lidar com esta crescente complexidade, duas abordagens têm sido utilizadas desde o surgimento da tecnologia de desenvolvimento de circuitos integrados. Na primeira abordagem, os processadores de propósito geral (GPP - *General-Purpose Processors*), baseados na arquitetura de Von-Neumann, são utilizados para a execução de qualquer aplicação. Ganhos de desempenho advêm de melhorias na organização (pipeline, superescalar, etc) introduzidas na micro-arquitetura dos processadores, bem como do desenvolvimento da tecnologia. Na segunda abordagem, hardware dedicado, ou circuitos integrados de aplicação específica (ASIC – *Application Specific Integrated Circuits*), são utilizados para a execução de aplicações também específicas. Nesta abordagem, o ganho de desempenho é obtido por intermédio da especialização das estruturas de hardware. Os GPPs são flexíveis o suficiente para executar qualquer aplicação, entretanto, essa flexibilidade é alcançada consumindo-se mais energia e provendo desempenho inferior à abordagem ASIC. Por outro lado, os ASICs oferecem alto desempenho e flexibilidade muito baixa, pois possuem estruturas de hardware específicas para a aplicação para a qual foram desenvolvidas.

As arquiteturas reconfiguráveis (AR), viabilizadas a partir da popularização dos dispositivos programáveis (FPGA – *Field Programmable Gate Array*), emergiram como uma solução arquitetural que visa unir a flexibilidade dos GPPs com o desempenho dos ASICs. A Figura 1 mostra a relação de flexibilidade/desempenho entre essas três abordagens. Arquiteturas reconfiguráveis são normalmente constituídas de um *array* de unidades funcionais reconfiguráveis (UFR) ou elementos de processamento (EP), interconectados por intermédio de um subsistema de interconexão. Tanto o desempenho quanto a flexibilidade das ARs são obtidos por intermédio da exploração do paralelismo intrínseco, resultante da disponibilidade de múltiplas unidades funcionais reconfiguráveis.

O aumento exponencial da densidade de transistores, previsto por Gordon Moore (MOORE, 1965), a complexidade e as limitações de desempenho dos monoprocessadores tornaram os chips multiprocessados (CMP), ou *multicore*, dominantes no projeto de processadores (CHIU; CHOU; CHEN, 2010). De modo geral, os CMPs são compostos por núcleos que são replicados várias vezes e conectados por um sistema de interconexão (barramento, rede em chip, etc). Contudo, o desempenho dos CMPs não irá melhorar conforme aumenta-se o número de núcleos, pois o grau de paralelismo da maioria das aplicações é limitado (AMDAHL, 1967). Apesar da adoção de múltiplos núcleos, as arquiteturas fixas dos chips multiprocesssados frequentemente não lidam bem com a crescente complexidade e heterogeneidade das aplicações (PAL; PAUL; PRASAD, 2014).

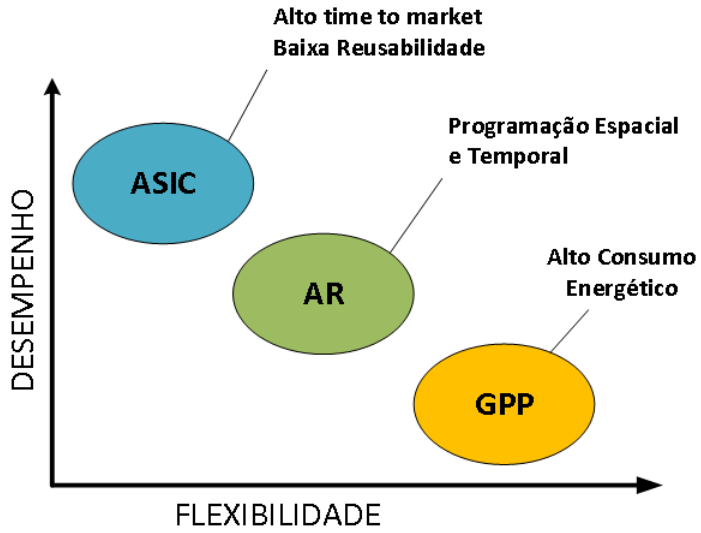


Figura 1 – Comparação da flexibilidade e desempenho fornecido pelas abordagens GPP, AR e ASIC

Utilizar arquitetura reconfigurável juntamente com um processador *multicore* permite a aceleração de aplicações explorando ILP (*Instruction-Level Parallelism*) e TLP(*Thread-Level Parallelism*). Contudo, os trabalhos encontrados na literatura (GOTTLIEB et al., 2002; WATKINS; ALBONESI, 2010; RUTZIG; BECK; CARRO, 2013; SOUZA et al., 2016) encontram dificuldades em usar eficientemente as unidades reconfiguráveis quando a carga de trabalho é desbalanceada entre os núcleos de processamento. Por este motivo, este trabalho propõe e avalia uma arquitetura reconfigurável para processadores *multicore*. A arquitetura proposta permite o compartilhamento de recursos da lógica reconfigurável entre os núcleos de modo que as unidade reconfiguráveis tenham pouca ociosidade e, consequentemente, um uso mais eficiente. A arquitetura proposta tem como propósito oferecer recursos computacionais para acelerar múltiplas *threads* (ou processos) executando simultaneamente em diferentes núcleos de um processador *multicore*. Como objetivo secundário, a arquitetura proposta deve apresentar-se como uma solução que requer um número significativamente menor de elementos de processamentos (EP) que outras arquiteturas reconfiguráveis encontradas atualmente na literatura.

Este trabalho propõe uma arquitetura reconfigurável que usa o mecanismo de tradução binária proposta em (BECK et al., 2008). Esse mecanismo permite que a geração de configuração seja feita em tempo de execução e sem modificações no código binário da aplicação ou necessidade de recompilação. O compartilhamento dos recursos reconfiguráveis também é feito em tempo de execução e sem necessidade de modificação no código binário. À luz do nosso conhecimento, este trabalho é o primeiro a permitir o compartilhamento de recursos reconfiguráveis em tempo de execução em um processador *multicore*. Sendo assim, as principais contribuições da presente dissertação de mestrado são:

- Proposta e implementação de uma arquitetura reconfigurável de granularidade grossa para processadores *multicore* que permite o compartilhamento de recursos reconfiguráveis entre os núcleos de processamento. Geralmente, as arquiteturas reconfiguráveis são integradas ao processador *multicore* de forma que cada núcleo possui acesso exclusivo aos seus recursos reconfiguráveis. No entanto, essa abordagem falha em lidar com aplicações onde há cargas de trabalho desbalanceada entre os núcleos do processador e, consequentemente, no uso eficiente dos seus recursos reconfiguráveis. A presente dissertação de mestrado propõe e implementa uma arquitetura reconfigurável que é compartilhada entre os núcleos de processamento.

- A arquitetura reconfigurável apresentada utiliza consideravelmente menos elementos de processamento que os trabalhos encontrados na literatura. Sabe-se que a arquitetura reconfigurável ocupa parte considerável da área do sistema. Como os processadores *multicores* se tornaram dominantes no mercado de processadores, a integração dessas arquiteturas nesse ambiente pode elevar consideravelmente o *overhead* em área ocasionada pela arquitetura reconfigurável. Por este motivo, é importante, no ambiente *multicore*, a arquitetura reconfigurável ser minimalista para que sua integração nesse ambiente seja viável.

- Proposta e implementação do escalonador de configurações que, em tempo de execução, gerencia os recursos reconfiguráveis da arquitetura e permite o compartilhamento dos mesmos entre os núcleos de processamento. Compartilhar os recursos reconfiguráveis entre os núcleos de processamento oferecendo lógica reconfigurável sob demanda para os núcleos melhora a eficiência do uso da arquitetura reconfigurável em um ambiente *multicore*. Para permitir o compartilhamento dos recursos da arquitetura reconfigurável entre os núcleos de processamento, foi proposto e implementado um escalonador de configuração em hardware.

- Adaptação do algoritmo do tradutor binário proposto em (BECK, 2009) para fazer renomeação de registrador e gerar configuração com execução especulativa. As falsas dependências, que podem ser resolvidas com renomeação de registrador, e as dependências de controle são dois fatores que podem limitar a exploração do ILP entre os os blocos básicos da aplicação. Para contornar esse problema na arquitetura aqui proposta, foi feita uma adaptação do algoritmo do tradutor binário para atender às necessidades da arquitetura reconfigurável proposta nesta dissertação de mestrado.

Esta dissertação está organizada da seguinte forma:

- **Fundamentação Teórica:** Neste capítulo são introduzidos alguns conceitos de arquiteturas reconfiguráveis necessários para o melhor entendimento do trabalho.

- **Estado da Arte:** No terceiro capítulo, são apresentados alguns dos principais trabalhos encontrados na literatura sobre arquitetura reconfiguráveis tanto para o ambiente monoprocessado quanto para o ambiente multiprocessado. Ao final deste capítulo, uma discussão é feita fazendo-se uma comparação dos trabalhos da literatura com arquitetura proposta nesta dissertação e apresentando suas contribuições.

- **A Proposta:** O quarto capítulo apresenta, em detalhes, a arquitetura reconfigurável proposta e seus componentes, bem como seu funcionamento.

- **Resultados:** O quinto capítulo apresenta os resultados de desempenho da arquitetura. Uma discussão sobre os resultados obtidos é feita. Uma avaliação do simulador desenvolvido em *systemC* também é realizada.

- **Conclusão e Trabalhos Futuros:** Por fim, a conclusão e trabalhos futuros são apresentados.

2 Fundamentação Teórica

Neste capítulo é apresentada a definição de arquiteturas reconfiguráveis, bem como o conjunto de critérios que são utilizados na literatura para classificar essas arquiteturas. Em cada seção, uma discussão de como cada um desses critérios impacta o sistema reconfigurável é feita.

2.1 Arquiteturas Reconfiguráveis

Arquiteturas Reconfiguráveis são sistemas integrados que permitem customização da unidade de hardware para satisfazer os requisitos computacionais de diferentes aplicações (SINGH et al., 2000). Geralmente, tais sistemas estão acoplados a um processador de propósito geral, onde parte da execução será acelerada e executada na arquitetura reconfigurável, e o restante da execução será feita no processador de propósito geral. Um exemplo desta execução pode ser vista na Figura 2. FPGA (*Field Programmable Gate Array*) é um exemplo de arquitetura reconfigurável e, geralmente, é composta por uma matriz de blocos lógicos configuráveis que são conectados por um sistema de interconexão programável (AZARIAN; AHMADI, 2009a).

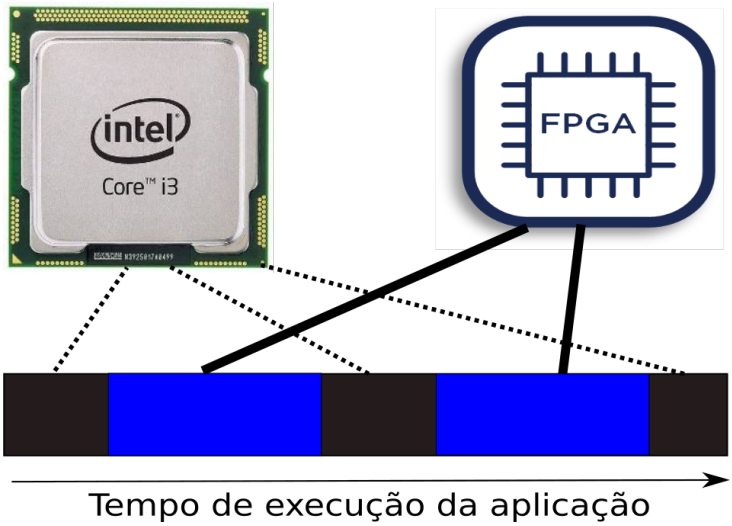


Figura 2 – Exemplo de um sistema com uma arquitetura reconfigurável (FPGA).

Trabalhos propondo uma classificação para as arquiteturas reconfiguráveis podem ser vistos em (HARTENSTEIN, 2001; TEHRE; KSHIRSAGAR, 2012; WIJTVLIET; WAEIJEN; CORPORAAL, 2016). Os critérios que são comuns a todas as propostas são: i) granularidade, ii) acoplamento da arquitetura reconfigurável com o processador, iii) mecanismo de reconfiguração e iv) subsistema de interconexão. Por este motivo, estes serão

os critérios utilizados para classificação das arquiteturas neste trabalho. Nas subseções seguintes, cada um desses critérios serão explanados.

2.1.1 Granularidade

A Granularidade de uma arquitetura reconfigurável é determinada pelo tamanho do dado que pode ser operado pelas unidades funcionais reconfiguráveis (UFR). Unidade funcional reconfigurável é um bloco lógico cuja funcionalidade pode ser reconfigurada (redefinida). Em arquiteturas de granularidade fina (FGRA - *Fine-Grained Reconfigurable Architectures*), como nas arquiteturas Garp (HAUSER; WAWRZYNEK, 1997) e Chimaera (HAUCK et al., 1997), as operações feitas na UFR são a nível de bits, como: armazenamento, adição, subtração, operações lógicas, entre outras. Nessas arquiteturas, as UFRs são constituídas por componentes de hardware cuja configuração geram operadores a nível do bit, tais como *flip-flops*, somadores/subtratores e multiplexadores de 1 bit. Por outro lado, nas arquiteturas reconfiguráveis de granularidade grossa (CGRA - *Coarse-Grained Reconfigurable Architectures*), as operações são feitas no nível de palavras, e as UFRs são constituídas de componentes de hardware mais complexos, tais como blocos de memória, ULAs (Unidades de Lógica e Aritmética), multiplicadores, deslocadores, entre outros.

As FGRAs geralmente são implementadas utilizando FPGAs e possuem alta flexibilidade, já que sua configuração está a nível de bit. Em teoria, qualquer circuito digital pode ser implementado em uma FGRA utilizando-se dos recursos da FPGA (LUTs (*Look-Up Tables*), CLBs (*Configurable Logic Block*), rede de interconexão e demais recursos). Contudo, a implementação de operações de granularidade mais grossa, como uma soma de 32 bits por exemplo, gerará uma configuração maior que uma CGRA geraria, pois esse somador seria constituído pela configuração de pelo menos 32 UFRs. Além disto, seria necessário configurar todo o subsistema de interconexão, de modo a, convenientemente, conectar entradas e saídas de todas as UFR utilizadas. Por outro lado, em uma CGRA, cuja a UFR opere com palavras de 32 bits, por exemplo, a construção do somador iria requerer apenas a configuração da operação de soma em uma das UFR da CGRA, provavelmente em uma ULA fisicamente disponível. Sendo assim, de modo geral, as FGRAs conseguem prover melhor benefício quando executam algoritmos orientados a bit, enquanto que as CGRAs quando executam aplicações de computação intensiva (BECK, 2009).

Percebe-se hoje uma tendência à utilização de CGRAs ao invés de FGRAs para alcançar alta performance e eficiência energética. Isso pode ser observado pela inserção de blocos IPs (*Intellectual property*) de granularidade grossa nas FPGAs, como DSP (*Digital Signal Processor*) de ponto flutuante e interfaces de memória externa (WIJTVLIET; WAEIJEN; CORPORAAL, 2016). Outro ponto que corrobora para essa tendência é o grande *overhead* no custo do roteamento, no tamanho da configuração e no tempo de reconfiguração do sistema causado pelos FGRAs mencionados no parágrafo anterior.

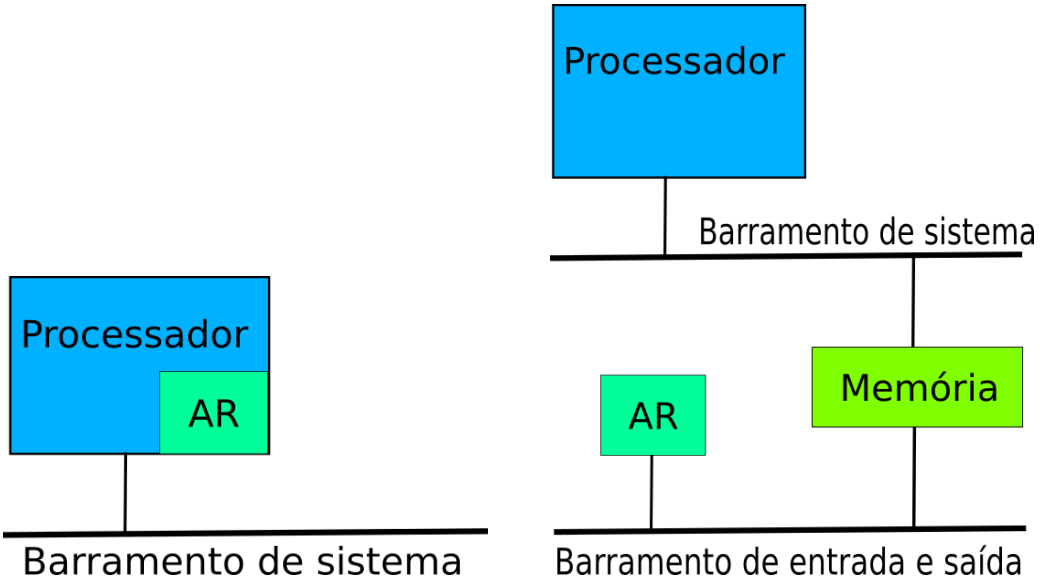
2.1.2 Acoplamento

O acoplamento define como a AR se conectará com o processador. Essa conexão irá influenciar diretamente como a comunicação e sincronização entre a AR e o processador será realizada, bem como a transferência de dados entre eles. De forma geral, quanto mais próximo do processador a AR estiver, mais eficiente será a comunicação entre eles.

As arquiteturas reconfiguráveis, quanto ao acoplamento, podem ser classificadas em: fortemente acoplada, fracamente acoplada e *stand-alone*. Quando a arquitetura não está acoplada a um processador, ela é classificada como *stand-alone* e possui a capacidade de executar uma aplicação inteira na arquitetura sem o auxílio de um processador. Normalmente, essas arquiteturas necessitam de ferramentas, ou de compilador, que possibilitem o mapeamento de uma aplicação completa na arquitetura. Um exemplo de arquitetura *stand-alone* é a TRIPS (SANKARALINGAM et al., 2004). Nas outras duas abordagens, fracamente e fortemente acoplada, o sistema computacional, além da arquitetura reconfigurável, possui um processador para executar trechos não mapeados na arquitetura reconfigurável.

As arquiteturas fracamente acopladas podem estar conectadas ao processador de duas formas: através do barramento de sistema, como um co-processador, ou através de um barramento de entrada e saída. Na primeira abordagem, a AR está mais próxima do processador e a comunicação é mais eficiente que a segunda. Contudo, a velocidade do barramento, seja de sistema ou de entrada e saída, pode ser um gargalo significante no desempenho da AR (HAUCK, 2004). Vale ressaltar que o ganho provido pela AR ao executar um determinado trecho de código será dado pela soma do tempo de comunicação ao tempo da execução do código na AR. Observe que o resultado dessa soma deve ser menor que o tempo de execução no processador para que a AR consiga prover uma melhoria no desempenho do sistema.

Por último, as arquiteturas fortemente acopladas visam melhorar a eficiência da comunicação entre o processador e a AR. Nesse tipo de acoplamento, a AR é implementada como uma unidade funcional dentro do processador, fazendo com que a comunicação entre a AR e o processador seja muito eficiente e rápida. Adicionalmente, essa abordagem permite que o processador e a AR compartilhem alguns recursos, como registradores. Um exemplo de arquitetura fortemente acoplada que compartilha o banco de registradores do processador com a arquitetura reconfigurável é a arquitetura Chimaera (HAUCK et al., 1997). Contudo, essa abordagem aumenta a área do processador, pois toda a arquitetura reconfigurável é implementada no mesmo chip do processador. A Figura 3 mostra os tipos de acoplamentos mencionados.



(a) Fortemente acoplado

(b) Fracamente acoplado no barramento de entrada e saída

(c) Fracamente acoplado como um co-processador

Figura 3 – Tipos de acoplamento da AR com o processador.

2.1.3 Mecanismo de Reconfiguração

A forma como múltiplas configurações podem ser geradas e carregadas na AR é determinada pelo mecanismo de reconfiguração utilizado. Dois mecanismos de reconfiguração são normalmente utilizados: reconfiguração estática e a reconfiguração dinâmica. Na primeira abordagem, um compilador, ou uma ferramenta de apoio, será responsável por identificar as partes da aplicação que podem ser aceleradas pela AR e gerar as configurações para esses trechos (AZARIAN; AHMADI, 2009b). Um dos pontos positivos dessa abordagem é que toda a complexidade da geração da configuração está no compilador, permitindo que várias técnicas de otimização possam ser implementadas para aumentar o ILP (*Instruction Level-Parallelism*), como otimização de *loops*. O DRESC (MEI et al., 2002) é um exemplo de compilador para CGRA que faz otimização de *loops* utilizando *modulo scheduling* e a heurística *simulated annealing*. Apesar dessa vantagem, na reconfiguração estática, a configuração só pode ser carregada no sistema quando a AR estiver inativa. Deste modo, todo o *array* que constitui a AR é reconfigurado antes

de passar a executar outro trecho de uma aplicação ou uma nova aplicação, o que pode implicar em aumento do tempo de execução, quando comparado com o uso do mecanismo de configuração dinâmica, pois o sistema precisa ser desativado, reconfigurado e reativado.

Por outro lado, na reconfiguração dinâmica, a responsabilidade da geração das configurações é tirada do compilador e o próprio sistema irá se encarregar de fazê-lo em tempo de execução. Essa abordagem permite que a AR possa ser reconfigurada com o sistema em operação. Desse modo, é possível que um bloco de hardware, acoplado à AR e ao GPP, gere as configurações em tempo de execução e, eventualmente, carregue uma nova configuração na AR. Um dos pioneiros nessa abordagem foi a arquitura Warp (LYSECKY; STITT; VAHID, 2004), que detecta dinamicamente trechos críticos da aplicação e mapeia para a arquitetura reconfigurável, que foi implementada em FPGA, de forma totalmente transparente.

2.1.4 Subsistema de Interconexão

Quando se projeta uma arquitetura reconfigurável, é desejável ter mecanismos que possibilitem a comunicação entre as unidades funcionais da arquitetura. O sistema de interconexão é o meio pelo qual os dados fluirão dentro da arquitetura e permitirá a comunicação e transferência de dados entre as unidades funcionais. As arquiteturas reconfiguráveis possuem quatro tipos de subsistema de interconexão: i) rede em malha, ii) rede de barramento, iii) *crossbar* e iv) conexão direta. Nas redes em malha, a escolha mais popular entre as CGRAs (WIJTVLIET; WAEIJEN; CORPORAAL, 2016), normalmente utiliza-se uma topologia em malha-2D e a comunicação entre as unidades funcionais é feita vizinho-a-vizinho, como pode ser visto na Figura 4. Essa estrutura é muito utilizada, pois possui alta escalabilidade. No subsistema de interconexão rede de barramento os dados são enviados através de barramentos globais aos quais todas as unidades funcionais têm acesso, contudo essa abordagem é pouca utilizada. Na interconexão *crossbar* todas as unidades funcionais estão conectadas às unidades funcionais das camadas adjacentes, como pode ser visto na Figura 5. Contudo, essa abordagem não é escalável, portanto, não é recomendada para arquiteturas que possuam muitas unidades funcionais. Por fim, temos a interconexão direta, onde cada unidade funcional é conectada diretamente às demais. Normalmente essa conexão é feita por meio da configuração do subsistema de interconexão que informará as interconexões das unidades funcionais.

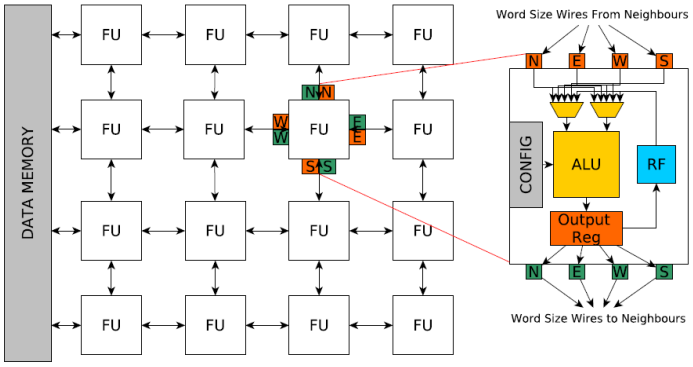


Figura 4 – Exemplo de um subsistema de interconexão em malha 2D. Retirado de (KARUNARATNE et al., 2017)

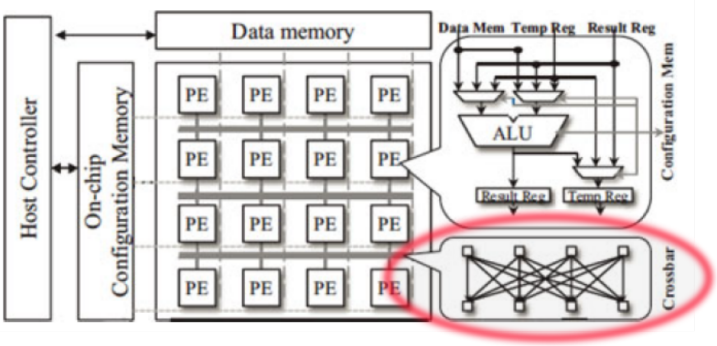


Figura 5 – Exemplo de um subsistema de interconexão *crossbar*. Adaptado de (LIANG et al., 2016)

3 Estado da Arte

Neste capítulo serão apresentados alguns trabalhos relevantes da literatura na área de arquitetura reconfiguráveis. Primeiramente, são mostradas arquiteturas reconfiguráveis voltadas para ambiente monoprocessado. Posteriormente, arquiteturas reconfiguráveis que visam acelerar *threads* em processadores *multicore* são mostradas. Por fim, uma comparação dos trabalhos citados com a proposta desta dissertação é feita.

3.1 CGRAs Relacionadas

Trabalhos relatando propostas de implementação e uso de arquiteturas reconfiguráveis começaram a surgir de forma mais regular em meados da década de 1990 (TESSIER; POCEK; DEHON, 2015). Desde então, um grande número de propostas foram publicadas.

A arquitetura SPLASH (GOKHALE et al., 1991) foi uma das pioneiras das arquiteturas reconfiguráveis. Essa arquitetura organizou 16 FPGAs em um *array* sistólico linear para resolver o problema de comparação de sequências de DNA e conseguiu prover melhor desempenho que os supercomputadores da época (CM-2 e Cray-2). Essa arquitetura conseguiu mostrar o potencial de aceleração que a computação reconfigurável poderia prover nas aplicações e, adicionalmente, possibilitava a programabilidade do sistema. Os autores utilizaram o termo programabilidade, pois, na época, o termo reconfigurabilidade/reconfiguração não era bem consolidado ainda.

Hauck et al. (1997) propuseram a arquitetura denominada Chimaera. A principal motivação foi diminuir o custo de comunicação entre a AR e o processador. Por este motivo, propuseram uma FGRA fortemente acoplada ao processador. Essa arquitetura foi uma das primeiras a propor uma AR fortemente acoplada ao processador compartilhando recursos, nesse caso, o banco de registradores. Registradores sombra (*shadow registers*) foram adicionados à arquitetura para reduzir ainda mais o custo de comunicação, já que eles mantinham uma cópia parcial do banco de registradores e o *array* reconfigurável operava sobre esses registradores. A AR foi acoplada a um processador MIPS R4000. A geração da configuração era feita estaticamente por um compilador em C. Esse compilador era responsável por mapear de maneira automática grupos de instruções MIPS em RFUOPs (*Reconfigurable Funcional Unit Operation*). Observa-se que o mecanismo de reconfiguração funcionava como uma extensão da ISA do processador MIPS, onde instruções novas são executadas na AR. Assim como na arquitetura Chimaera, a arquitetura proposta neste trabalho também utiliza uma arquitetura reconfigurável fortemente acoplada ao processador, e compartilha alguns recursos como: cache L1 de dados e banco de registradores.

Depois do Chimaera, Goldstein et al. (1999) propuseram uma AR que também possuía como foco a redução do custo de reconfiguração da arquitetura denominada *PipeRench*. A grande contribuição deste trabalho foi o uso da técnica denominada reconfiguração pipeline. A ideia dessa técnica é quebrar uma configuração grande em pedaços menores que serão executados e reconfigurados sob demanda. Esse processo de permitir execução e reconfiguração sob demanda foi chamado de processo de virtualização pelos autores. Utilizando-se da técnica de reconfiguração pipeline, o tempo de configuração foi reduzido, pois reconfigura-se menos hardware. Já o uso do processo de virtualização reduz a quantidade de hardware alocado para execução de uma configuração, pois o *PipeRench* adapta as configurações para caberem no hardware reconfigurável. Essa técnica de virtualização e reconfiguração pipeline é mostrada na Figura 6. Na parte de cima da figura, é mostrada a execução de uma aplicação que foi dividida em 5 estágios de pipeline e leva 7 ciclos para executar. Na parte inferior, apresenta-se o uso da técnica de reconfiguração pipeline e da virtualização, onde necessita-se somente de 3 estágios virtuais de pipeline para executar a mesma aplicação no hardware reconfigurável. Nesse exemplo, o hardware reconfigurável tem três estágios físicos de pipeline. O estágio 1 é configurado no ciclo 1 e executa durante 2 ciclos. No próximo ciclo, o estágio 2 está sendo configurado e o estágio 1 está executando, assim como tradicionalmente um pipeline funciona. Quando atingir o ponto onde necessita-se configurar o quarto estágio da aplicação, como não há um quarto estágio físico de pipeline, virtualiza-se o primeiro estágio, que não está mais em uso, portanto, o estágio 4 é configurado no primeiro estágio virtual de pipeline. Perceba que a virtualização é possível porque o estágio 1 e 4 serão executados em diferentes períodos de tempo. Portanto, o *PipeRench* aloca computação no tempo, diferentes estágios virtuais, e no espaço, computações feitas em paralelo dentro de cada estágio virtual.

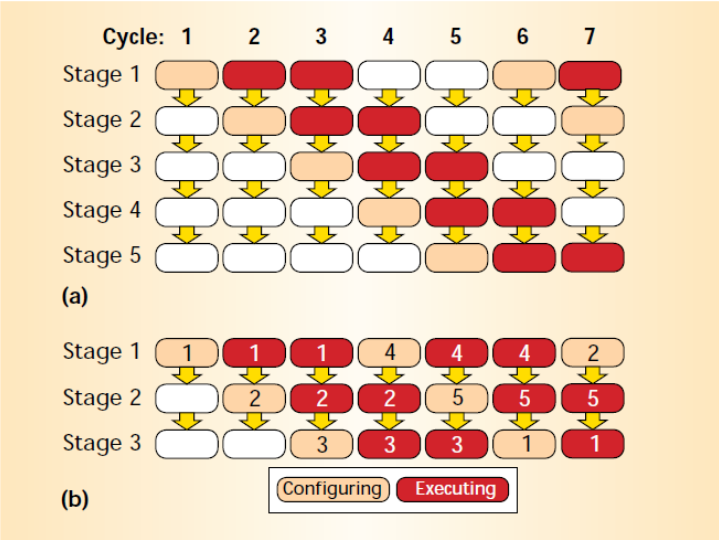


Figura 6 – Técnica de reconfiguração pipeline proposta no PipeRench. Retirado de (GOLDSTEIN et al., 1999)

A arquitetura do *PipeRench* é organizada em estágios físicos chamados de *stripes*.

Cada *stripe* possui uma rede de interconexão e um conjunto de PEs (*Processing Elements*), como pode ser visto na Figura 7. A comunicação entre os PEs pode ser feita através de saída salvas em registradores dos PEs (*Pass registers*) e através de barramentos globais para PEs que não são de *stripes* adjacentes. A geração de configuração é feita através de um compilador parametrizável. O compilador recebe como parâmetro uma descrição da arquitetura: número de PEs por *stripe*, Granularidade do PE, topologia de interconexão, entre outras características da arquitetura. Inspirado na virtualização dos *stripes* do *PipeRench*, a arquitetura reconfigurável proposta neste trabalho tem somente uma coluna de PEs (Elementos de Processamento) por núcleo, o que seria equivalente a um *stripe* no *PipeRench*, que pode ser virtualizada alocando-se computações no tempo. Dessa forma, economiza-se área em chip devido ao reuso dos PEs.

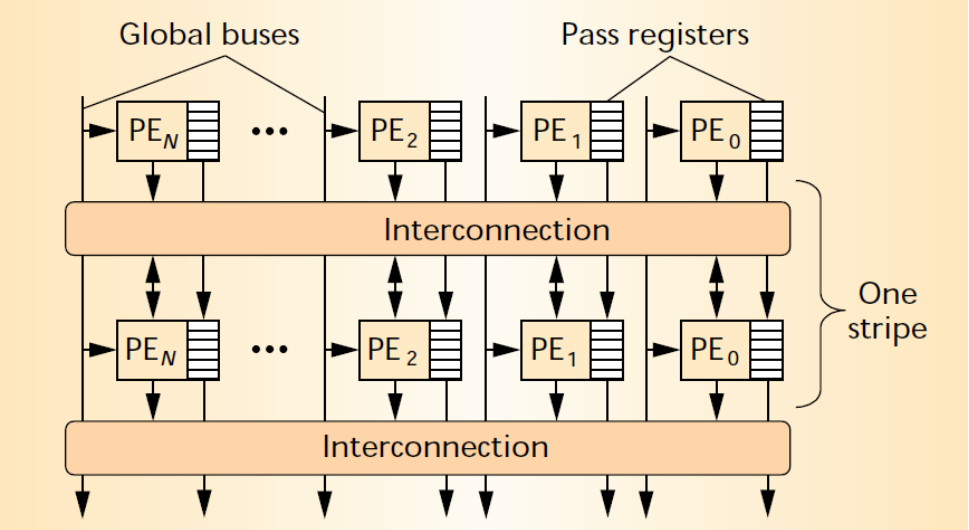


Figura 7 – Visão geral do PipeRench. Retirado de (GOLDSTEIN et al., 1999).

MorphoSys (SINGH et al., 2000) é uma CGRA fracamente acoplada a um processador TinyRisc (ABNOUS et al., 1992) que visa acelerar aplicações de compressão de vídeo, criptografia de dados e *target recognition*. A comunicação com o processador é feita através de uma controladora de DMA (*Direct Memory Access*) e um *frame buffer*. A arquitetura é composta de 64 elementos de processamento, chamados de *Reconfigurable Cell* (RC) pelos autores, arranjadas em um *array* $8x8$, dividido em 4 quadrantes $4x4$ (ver Figura 8). Cada elemento de processamento é configurado por uma palavra de configuração de 32 bits. Instruções dedicadas foram inseridas no processador TinyRISC para permitir a comunicação e execução de trechos de código no *Morphosys*. Um compilador C foi desenvolvido para adicionar essas instruções dedicadas que suportam a execução no *Morphosys*, mas a configuração é feita manualmente. Os resultados de desempenho foram obtidos utilizando as aplicações alvo: compressão de vídeo, criptografia de dados e *target recognition*. O *Morphosys* mostrou-se capaz de melhorar o desempenho dessas aplicações, comparando-se com outras ARs e com ASICs.

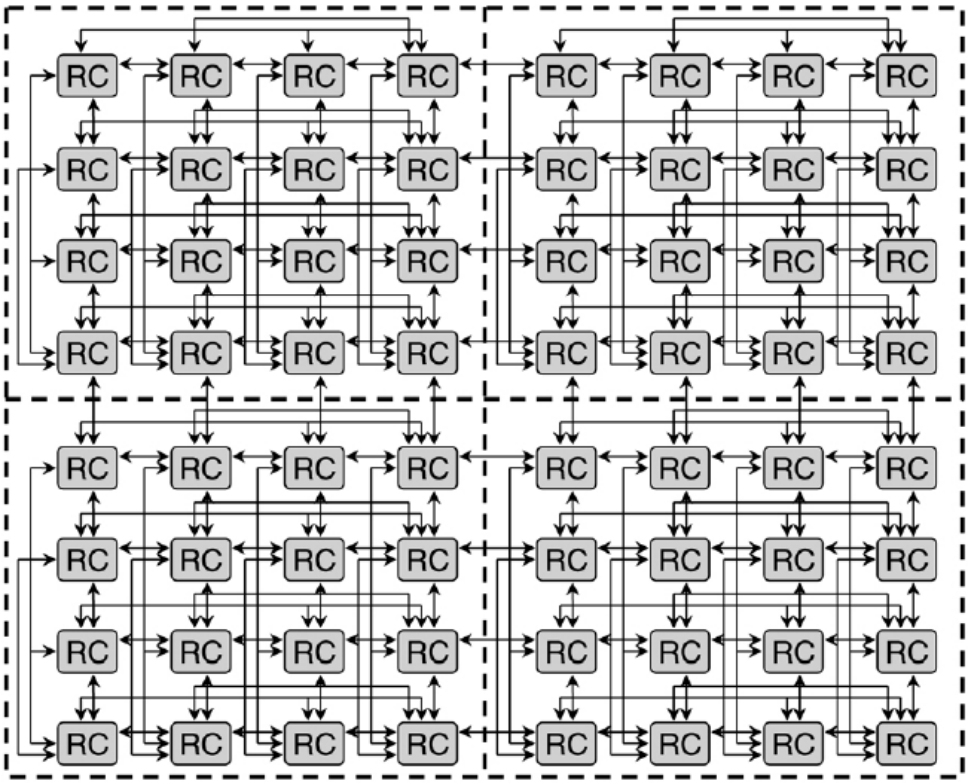


Figura 8 – *Array* reconfigurável proposto pelo MorphoSys. Retirado de (SINGH et al., 2000).

Visando prover redução no *time-to-market* de arquiteturas reconfiguráveis, surgiu a arquitetura *Warp processor* (LYSECKY; STITT; VAHID, 2004). Até então, as arquiteturas reconfiguráveis necessitavam de apoio de compiladores especiais para fazer o particionamento das regiões a serem aceleradas, o que, segundo os autores, fazia com que o *time-to-market* desses sistemas ficasse muito elevado. A proposta do *Warp processor* é uma das primeiras da literatura a dinamicamente detectar e mapear trechos críticos das aplicações para executar em unidades de hardware customizadas. Uma das vantagens citadas pelos autores da detecção e mapeamento dinâmico dos trechos críticos é que o processo de reconfiguração e execução na arquitetura reconfigurável é totalmente transparente ao usuário e, portanto, não há nenhuma necessidade de o programador alterar seu código fonte para que ele usufrua de tal benefício. O *Warp Processor* é composto de dois processadores: um responsável pela execução da aplicação e outro por executar o algoritmo de particionamento. Além dos dois processadores, a arquitetura usa uma FPGA para executar as regiões críticas. Um bloco de hardware adicional foi inserido na arquitetura para detectar regiões críticas. Quando o bloco de hardware detecta uma região crítica, ele envia para o processador de particionamento fazer o particionamento daquele trecho e mapeá-lo em FPGA. É importante destacar que esse mapeamento em FPGA inclui vários passos, tais como: particionamento, síntese, mapeamento e roteamento e envio do *bitstream* para a FPGA. Esse foi um dos grandes desafios citados pelos autores: minimizar o tempo de reconfiguração do FPGA, bem como da geração da configuração. A proposta do *Warp*

processor possui foco no algoritmo de particionamento hardware/software e pouca atenção é dada para a arquitetura reconfigurável propriamente dita. Diferentemente da proposta do *Warp processor*, este trabalho possui foco na proposta da arquitetura reconfigurável para acelerar aplicações em um ambiente *multicore*.

Com objetivo semelhante ao do *Warp processor*, surgiu outro trabalho (BECK et al., 2008) que também propôs redução no *time-to-market* e mapeamento de trechos críticos para execução em arquitetura reconfigurável de forma transparente. Contudo, esse trabalho, visando mitigar o custo de reconfiguração que os autores do *Warp processor* mencionaram como sendo a parte mais difícil do projeto, propõe uma CGRA fortemente acoplada a um processador, o que, como já explanado no Capítulo 2, implicará na redução do tempo de reconfiguração. Beck et al. (2008) propuseram uma CGRA fortemente acoplada a um processador MIPS pipeline e a um tradutor binário. O tradutor binário é um bloco de hardware capaz de gerar configuração em tempo de execução, a partir da execução das aplicações no processador. Assim, não é necessário alterar o código binário das aplicações. Uma visão esquemática da CGRA proposta pode ser vista na Figura 9. A CGRA é organizada em uma estrutura de matriz de unidades funcionais, onde cada instrução é alocada em uma interseção entre uma linha e uma coluna. Instruções alocadas em uma mesma linha são executadas em paralelo, enquanto que instruções alocadas em linhas diferentes são executadas de forma sequencial. Vale ressaltar que a execução sequencial não significa execução em ciclos diferentes, pois, dependendo do atraso de propagação de cada unidade funcional, mais de uma operação pode ser realizada por ciclo. As unidades funcionais da arquitetura foram divididas em três grupos. Essa divisão foi feita levando-se em conta a latência das operações realizadas em cada grupo. O grupo 1 corresponde às unidades lógicas e aritméticas (operações de menor latência), o grupo 2 às unidades de *load/store* e o grupo 3 aos multiplicadores. As unidades funcionais do grupo 1 conseguem executar até três instruções dependentes em único ciclo. A CGRA funciona como uma unidade funcional no estágio de execução do processador. Quando um trecho de código é executado na CGRA, o pipeline é parado (*stall*) até que a CGRA termine sua execução. A arquitetura proposta em (BECK et al., 2008) permite modificar sua estrutura de matriz de unidades funcionais variando a quantidade de linhas e colunas disponíveis no *array* reconfigurável. Na arquitetura proposta neste trabalho, adotou-se o tradutor binário como mecanismo para gerar configuração para a arquitetura reconfigurável em tempo de execução. O tradutor binário teve que ser adaptado para gerar configuração para o *array* reconfigurável proposto. Uma das principais vantagens de se utilizar o tradutor binário é que o uso do *array* reconfigurável se torna transparente ao usuário e nenhuma alteração, ou recompilação, do código-fonte é necessária para que o *array* reconfigurável seja utilizado.

Feng e Yang (2013) propuseram uma CGRA voltada para Processamento Digital de Sinais (PDS). A arquitetura proposta é composta por 16 elementos de processamento, quatro registradores de configuração e uma rede de interconexão compartilhada. Os

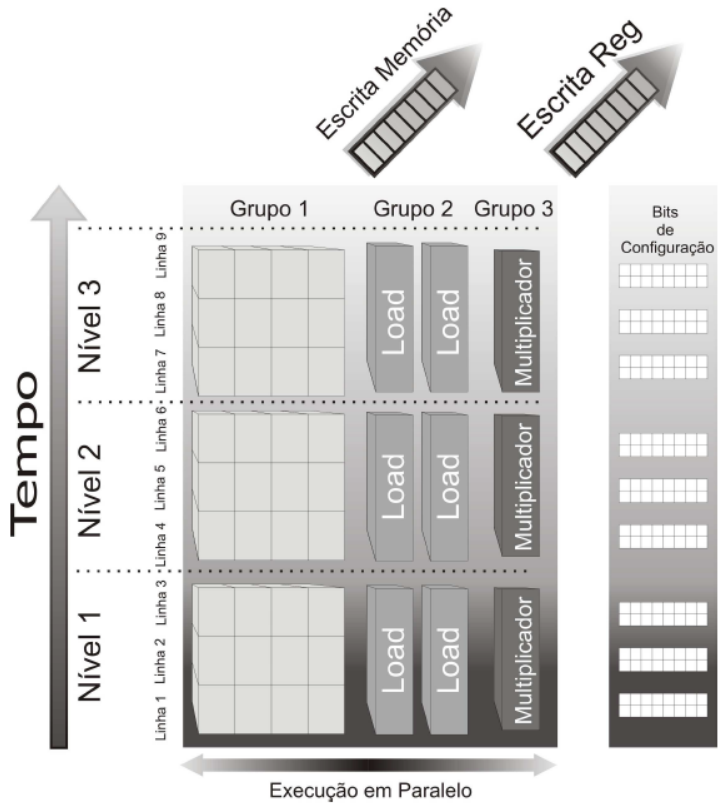


Figura 9 – Visão esquemática da CGRA proposta em (BECK et al., 2008).

elementos de processamento são distribuídos em 4 estágios, cada estágio com 4 elementos de processamento. Segundo os autores, a principal contribuição do trabalho foi a redução do custo de implementação, considerando a área em chip como unidade de medida. A redução da área deveu-se à utilização de um subsistema de interconexão simplificado.

Estudos mais recentes estão propondo arquiteturas reconfiguráveis para acelerar aplicações específicas (XU et al., 2015; PATEL; BLEAKLEY, 2016; WANG; CAO; YANG, 2017; LI; PEDRAM, 2017; YUAN et al., 2017; ZHAO; WANG; LIU, 2017). Isso mostra uma tendência à especialização da arquitetura reconfigurável para favorecer determinados comportamentos das aplicações, algo semelhando à proposta do ASIC. Dessa forma, consegue-se prover mais aceleração e eficiência energética, pois a arquitetura reconfigurável terá características que atendem às necessidades das aplicações para as quais a arquitetura foi projetada. Os três primeiros estudos propuseram uma CGRA para acelerar, respectivamente, o algoritmo de casamento de padrão de digitais (XU et al., 2015), o algoritmo REACT (algoritmo de processamento de bio-sinais para detectar em tempo real crises de epilepsia) (PATEL; BLEAKLEY, 2016) e o algoritmo de cifragem AES (WANG; CAO; YANG, 2017). Os três últimos trabalhos propuseram CGRA para acelerar aplicações de *deep learning*. (LI; PEDRAM, 2017) visam a aceleração do treinamento de redes neurais profundas. Já as outras duas abordagens (YUAN et al., 2017; ZHAO; WANG; LIU, 2017) possuem foco na aceleração em um tipo específico das redes neurais profundas, as redes

neurais convolucionais.

3.2 *Arrays* Reconfiguráveis para *Multicore*

Os trabalhos citados anteriormente são compostos de vários elementos de processamento (no mínimo 16), o que torna a utilização dessas arquiteturas em ambiente *multicore* muito custosa em termos de área. Arquiteturas reconfiguráveis voltadas para *multicore* também têm sido propostas, como em (WATKINS; ALBONESI, 2010) (RUTZIG; BECK; CARRO, 2013) (YAN et al., 2014) (SOUZA et al., 2016). Essas arquiteturas visam explorar as vantagens dos processadores *multicore* e das arquiteturas reconfiguráveis e acelerar as aplicações explorando tanto o ILP (*Instruction Level Parallelism*) quanto o TLP (*Thread Level Parallelism*).

Gottlieb et al. (2002) propuseram a arquitetura Amalgam que integrava 4 processadores de propósito geral com 4 unidades reconfiguráveis. A arquitetura é organizada em 8 *clusters* independentes, sendo que 4 *clusters* são processadores de propósito geral e os demais são *clusters* de unidades reconfiguráveis. A comunicação entre os *clusters* e dos *clusters* com o sistema de memória é feita através de uma rede em chip. Uma visão geral da organização da arquitetura Amalgam é apresentada na Figura 10. A cache de dados L1 foi dividida em quatro bancos para que fosse possível realizar até 4 operações na memória por ciclo. Cada *cluster* de processador de propósito geral possui um processador que faz o despacho em ordem de até 2 instruções por ciclo e não permite execução fora de ordem. Já os *clusters* de unidade reconfigurável contêm banco de registradores e blocos de lógica reconfigurável. Uma visão esquemática desses *clusters* pode ser visto na Figura 11. Ambos *clusters* têm uma porta de leitura e uma porta de escrita na rede em chip, que são usadas tanto para comunicação *inter-cluster* como para referências a memória. Segundo os autores, a unidade reconfigurável consegue implementar eficientemente tanto tarefas de granularidade grossa quanto tarefas de granularidade fina. Para validar a arquitetura, os autores desenvolveram um simulador próprio que fornece precisão a nível de ciclos, o AmalSim. Não foi dado detalhes da linguagem utilizada. A configuração dos *clusters* das unidades de reconfiguração é feita através de um arquivo de configuração que é uma das entradas do simulador AmalSim. Uma limitação citada por (YAN et al., 2014) da arquitetura Amalgam é que as unidades reconfiguráveis são separadas fisicamente, dessa forma, torna impossível a junção de duas unidades reconfiguráveis para mapear uma lógica mais complexa.

Chiu, Chou e Chen (2010) propuseram uma arquitetura *multicore* reconfigurável chamada de *hyperscalar*. Essa arquitetura consiste de múltiplos núcleos que podem ser unidos dinamicamente para formar núcleos superescalares com o objetivo de acelerar *threads*. Para tornar a união dos núcleos possível, foi proposto um banco de registradores

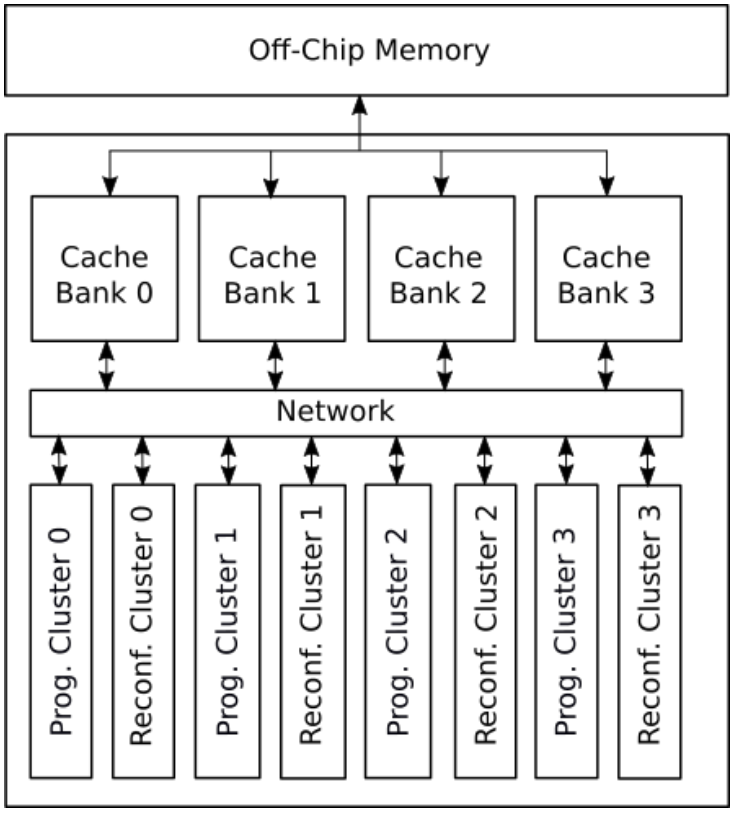


Figura 10 – Organização da arquitetura Amalgam. Retirado de (GOTTLIEB et al., 2002).

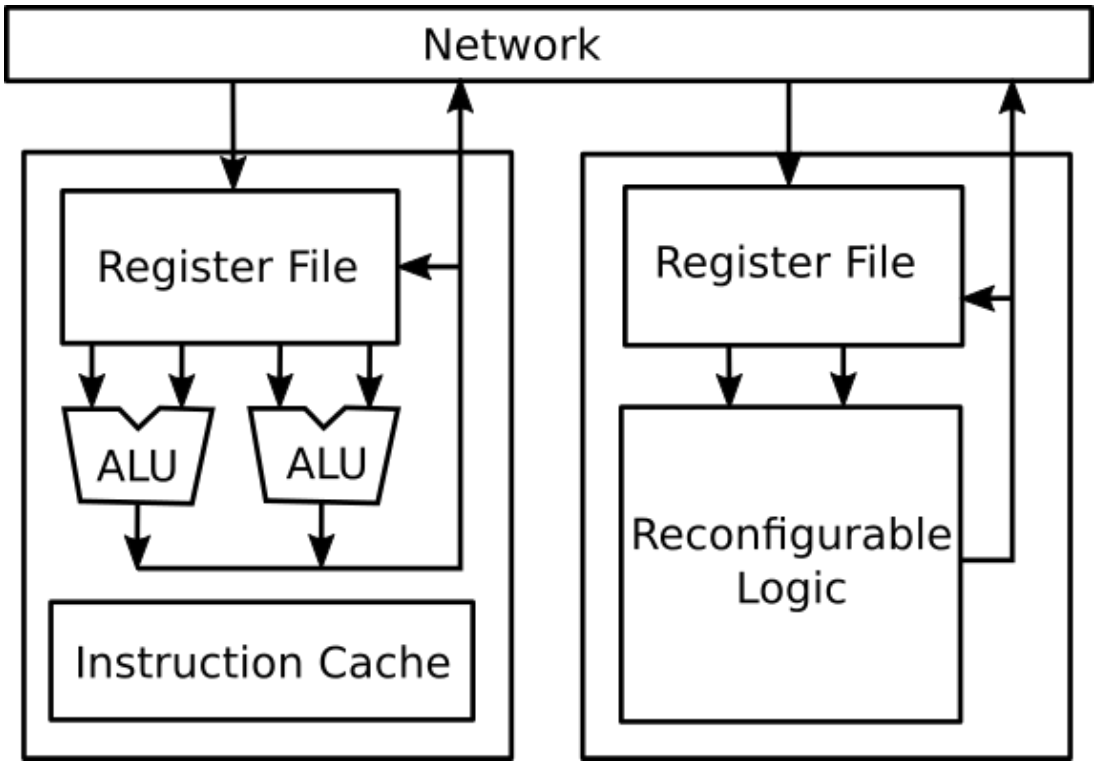


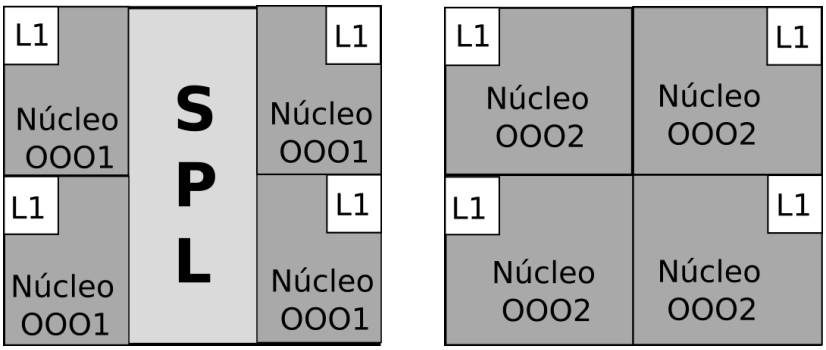
Figura 11 – *Cluster* de processador de propósito geral e *cluster* de unidade reconfigurável Amalgam. Retirado de (GOTTLIEB et al., 2002).

virtual compartilhado (*virtual shared register file*) para permitir que instruções de uma *thread* sejam executadas em um núcleo unido, acessando o mesmo conjunto de banco de registradores. Um analisador de instruções também foi proposto para realizar a verificação de dependência entre as instruções. Além disso, três instruções foram definidas (*UNI,SetPC* e *ADJ*) para o programador ou sistema operacional aumentar ou diminuir o número de núcleos unidos. O *hyperscalar* consegue prover alto desempenho quando o TLP é baixo e alta vazão quando o TLP é alto.

Diferente da proposta do *hyperscalar*, onde a arquitetura do *multicore* é reconfigurada, o trabalho desta dissertação propõe uma arquitetura reconfigurável para processadores *multicore*. Nessa proposta, o processador *multicore* não é reconfigurado, mas é acoplado a uma arquitetura reconfigurável que acelera a execução de trechos críticos das aplicações. A arquitetura proposta visa acelerar *threads* explorando o ILP e TLP.

Watkins e Albonesi (2010) propuseram o *ReMAP* (*Reconfigurable Multicore Acceleration and Parallelization*), uma arquitetura reconfigurável compartilhada para acelerar e paralelizar aplicações em arquiteturas multiprocessadas heterogêneas. A arquitetura é composta de dois tipos de *clusters*. O cluster do tipo 1 é composto por quatro núcleos OOO1 mais um SPL (*Specialized Programmable Logic*) de 24 linhas (ver Figura 12a). O *cluster* tipo 2 é composto por 4 núcleos OOO2 (ver Figura 12b). Os núcleos chamados de OOO1 são processadores *single issue* e fazem execução fora de ordem, já os núcleos OOO2 são processadores *dual issue* e também realizam execução fora de ordem. A arquitetura é constituída de dois *clusters* do tipo 1 e dois do tipo 2. No *cluster* do tipo 1, os quatro núcleos compartilham o SPL utilizando um mecanismo de *round-robin*. O SPL é composto de 24 linhas, onde cada linha contém 16 células. As células operam dados do tamanho de 8 bits. O SPL é integrado ao núcleo do processador como uma unidade funcional reconfigurável. A programação no *ReMAP* é feita gerando as configurações manualmente, mas os autores pretendem dar suporte à configuração via compilador. Segundo os autores, o *ReMAP* foi capaz de obter 45% de ganho de desempenho quando comparado com um sistema de área equivalente, núcleos maiores e com hardware dedicado de comunicação. A arquitetura proposta nesta dissertação, ao contrário do ReMAP, utiliza elementos de processamento de granularidade grossa. A utilização de granularidade grossa favorece na obtenção de menores tempos de configuração e possibilita a aceleração de maior número de trechos de uma aplicação ("THEODORIDIS; SOUDRIS; VASSILIADIS, "2007").

Stitt e Vahid (2011) estenderam seu trabalho com o *Warp Processor* (LYSECKY; STITT; VAHID, 2004) para chips multiprocessados e propuseram uma técnica de otimização dinâmica de multiprocessadores chamada de *Thread Warping*. Essa técnica utiliza a síntese dinâmica, proposta em (LYSECKY; STITT; VAHID, 2004), como base para, dinamicamente, sintetizar *threads* em FPGA. O funcionamento do *thread Warping* pode ser visto na Figura 13. No passo 1, a aplicação criará uma *thread* que será colocada em



(a) Cluster tipo 1 (b) Cluster tipo 2

Figura 12 – Clusters do ReMAP. Retirado de (WATKINS; ALBONESI, 2010)

.

uma fila pelo sistema operacional, pois o número de *threads* existentes podem exceder a quantidade de processadores disponíveis. Essa *thread* então esperará até ser escalonada pelo processador, como mostrado no passo 2. O *thread warping* usa o suporte do sistema operacional para monitorar a fila de *threads*, analisar as *threads* em espera e executar a ferramenta de CAD (*Computer Aided Design*). No passo 3, essa ferramenta é executada em um único núcleo do processador e criará aceleradores para aquela *thread* que serão mapeados em FPGA. Depois que a ferramenta de CAD terminar o mapeamento e sintetizar em FPGA, o sistema operacional começará a escalonar *threads* tanto na FPGA como nos núcleos do processador, como é demostrado no passo 4. Embora esse trabalho proponha o uso de uma arquitetura reconfigurável (FPGA) para acelerar *threads*, pouca atenção é dada pelos autores na arquitetura reconfigurável *multicore*. O trabalho foca na ferramenta de CAD e no suporte dado pelo sistema operacional. Diferentemente do *thread warping*, este trabalho foca na proposta de uma arquitetura reconfigurável capaz de prover aceleração em um ambiente *multicore*.

Rutzig, Beck e Carro (2013) propuseram um *array* reconfigurável para sistemas multiprocessados, denominado CReAMS (*Custom Reconfigurable Array for Multiprocessor Systems*). A arquitetura é capaz de explorar TLP (*Thread Level Paralelism*) replicando unidades de hardware denominadas DAPs (*Dynamic Adaptive Processor*). O DAP é uma extensão da arquitetura reconfigurável proposta em (BECK et al., 2008), já citada anteriormente, e também é fortemente acoplada ao processador. A comunicação entre os DAPs é feita através de uma rede em chip com topologia em malha 2D utilizando roteamento XY. O DAP é composto de 3 blocos principais: Caminho de dados Reconfigurável, Núcleo do processador e o tradutor binário. Para realizar a análise de desempenho, a arquitetura CReAMS foi comparada com outros processadores *multicore* que, segundo os autores, tinham área equivalente. O CReAMS, quando comparado com *MPSparcV8*, conseguiu reduzir, em média, o tempo de execução em 15% e o EDP (*Energy-Delay Product*) foi 47% menor. Quando comparado com o *Out-of-Order Superscalar SparcV8*, mostrou ganho

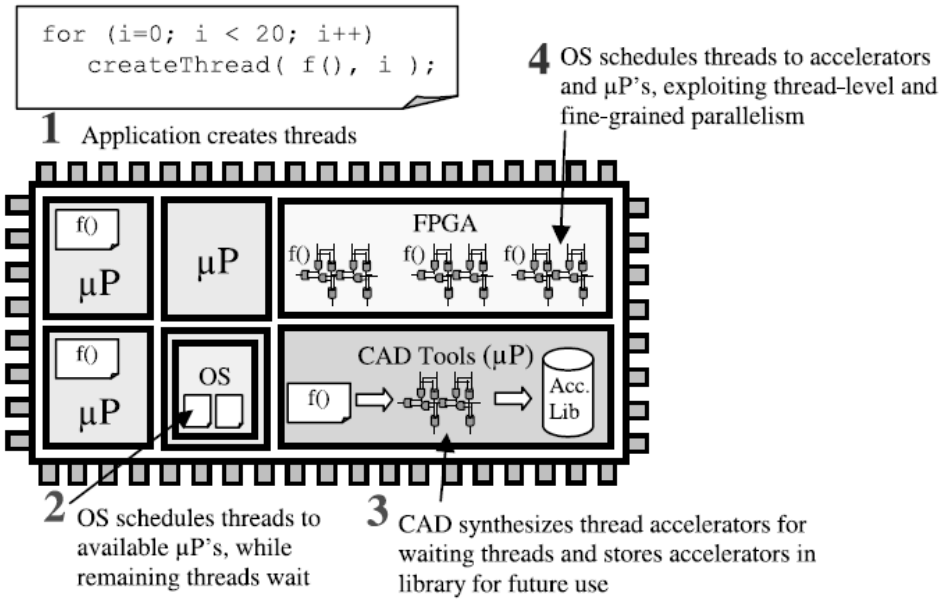


Figura 13 – Visão geral do *Thread warping*. Retirado de (STITT; VAHID, 2011).

de 28% em desempenho. Diferentemente do CReAMS, que um usa *array* reconfigurável exclusivo para cada DAP, a arquitetura aqui proposta utiliza uma arquitetura reconfigurável composta por um *array* reconfigurável compartilhado por todos os núcleos. Então, ao invés de disponibilizar hardware reconfigurável exclusivamente para cada um dos núcleos, como o CReAMS faz, a proposta deste trabalho é fornecer lógica reconfigurável sob demanda para os núcleos do processador *multicore*. Isso trará um melhor uso da lógica reconfigurável do sistema, quando houver cargas de trabalho desbalanceadas, pois o núcleo que estiver com mais trabalho terá mais recursos para utilizar na lógica reconfigurável.

Yan et al. (2014) propuseram uma arquitetura que combina um processador *multicore* de propósito geral com arquitetura reconfigurável. A organização da arquitetura reconfigurável é logicamente divida em RPUs (*Reconfigurable Processing Unit*) que são acopladas, como co-processadores, aos núcleos do processador. O acoplamento é feito com o auxilio de uma rede *crossbar*, dessa forma todos os processadores têm conexão com todas RPUs. Uma estrutura reconfigurável baseada na FPGA *Xilinx Virtex-5* (XILINX, 2018) foi utilizada para as RPUs. As RPUs possuem, portanto, granularidade fina. Para mapear funções complexas na lógica reconfigurável, duas ou mais RPUs podem ser unidas, como é costume em FPGAs. Um gerente de RPUs foi projetado para controlar a execução na RPU, gerenciar a configuração e o subsistema de interconexão. A memória de contexto de configuração usa uma estrutura *multi-context* para reduzir o tempo de reconfiguração. Nessa estrutura, vários contextos de configuração são armazenados concorrentemente na mesma lógica reconfigurável, mas somente um contexto está ativo, enquanto que os outros estão em espera (*standby*). Uma visão geral da arquitetura proposta pode ser vista na Figura 14. Para dar suporte à execução na arquitetura reconfigurável, o conjunto de

instruções do processador foi estendido e 7 novas instruções foram adicionadas. Essas instruções são usadas para transferência de dados e comunicação entre o núcleo do processador e a RPU. As configurações são geradas manualmente, mas os autores afirmam que elas podem ser geradas através de um compilador utilizando-se de macros e modificações no *assembler*. A arquitetura proposta nesse trabalho utiliza uma abordagem semelhante à proposta de (YAN et al., 2014), pois também visa permitir o compartilhamento da lógica reconfigurável entre os núcleos do processador. A diferença está na granularidade da arquitetura e, consequentemente, na forma como esse compartilhamento é feito. Na proposta apresentada em (YAN et al., 2014), as RPUs são utilizadas para implementar funções em circuitos dedicados, caso falte recurso na RPU, unem-se duas ou mais RPUs para implementar aquela função. O compartilhamento é feito em tempo de compilação. Na proposta deste trabalho de mestrado, a arquitetura permite o compartilhamento de elementos de processamento que é feito dinamicamente e em tempo de execução pelo escalonador de configuração. Desse modo, melhora-se a utilização do hardware reconfigurável de forma transparente e sem modificação ou recompilação do código-fonte.

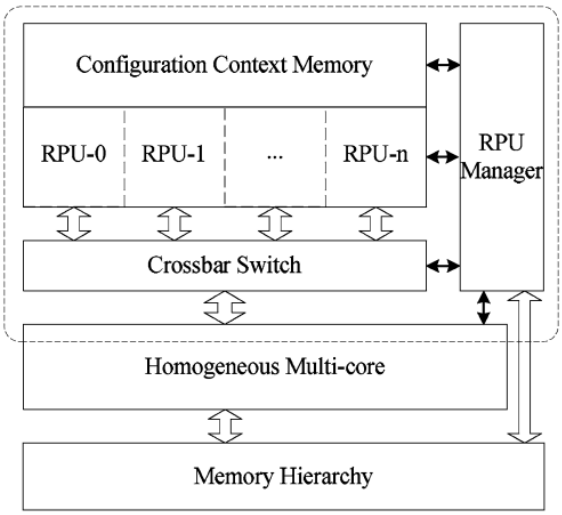


Figura 14 – Visão geral da arquitetura proposta em (YAN et al., 2014). Retirado de (YAN et al., 2014).

Souza et al. (2016) propuseram uma arquitetura reconfigurável para processadores *multicore* denominada HARTMP (*Heterogeneous Arrays for Reconfigurable and Transparent Multicore Processing*). O HARTMP é homogêneo na arquitetura (implementa a mesma ISA do processador) e heterogêneo na organização (quantidades diferentes de unidades funcionais reconfiguráveis disponíveis para cada núcleo). O sistema reconfigurável do HARTMP é o mesmo proposto no CReAMS, composto por DAPs fortemente acopladas ao núcleo do processador. A diferença existente entre a proposta do HARTMP e do CReAMS é que o CReAMS é homogêneo tanto na arquitetura quanto na organização. Para ter a heterogeneidade na organização, o HARTMP gerou DAPs com diferentes quantidade de unidades funcionais. A heterogeneidade do HARTMP é completamente transparente

para o programador ou sistema operacional devido ao mecanismo de tradução binária implementado. As instruções que são analisadas e despachadas para a lógica reconfigurável são geradas pelo tradutor binário, portanto, nenhuma modificação em código é necessária. Para análise de desempenho e de ganho energético foram geradas três configurações homogêneas do CReAMs chamadas de Ho1 (menor), Ho2 (média) e Ho3 (maior) e duas heterogêneas para serem comparadas. O HARTMP foi capaz obter ganho de desempenho e energético de 59% e 11%, respectivamente, quando comparado com sua versão homogênea (CReAMS). Diferente do CReAMS (RUTZIG; BECK; CARRO, 2013), o HARTMP gera DAPs com quantidades diferentes de unidades funcionais, alocando mais recursos para núcleos onde há maior carga de trabalho visando eficiência mesmo em aplicações onde tem baixo TLP. Contudo, essa configuração do tamanho dos DAPs é determinada estaticamente e, uma vez determinada, não há como ter modificações em tempo de execução. Portanto, para um conjunto heterogêneo de aplicações (TLPs variados), a configuração do DAP pode ser adequada para umas aplicações e para outras não. Diferentemente do HARTMP, que um usa *array* reconfigurável de tamanho variado para cada DAP, a arquitetura aqui proposta utiliza uma arquitetura reconfigurável composta um *array* reconfigurável que é compartilhado entre os núcleos. Então, ao invés de disponibilizar hardware reconfigurável com uma quantidade específica de recursos e de forma exclusiva para cada um dos núcleos, a proposta deste trabalho é fornecer lógica reconfigurável sob demanda para os núcleos do processador multicore. Os recursos utilizados por cada núcleo na lógica reconfigurável serão ajustados em tempo de execução de acordo com a necessidade de cada núcleo.

A Tabela 1 apresenta as característica das arquiteturas que foram apresentadas nesse capítulo e da arquitetura proposta nesta dissertação de mestrado. As características consideradas são: i) granularidade, ii) acoplamento, iii) reconfiguração, iv) se a arquitetura é voltada para *multicore* e v) se permite o compartilhamento de recursos reconfiguráveis entre os núcleos do processador.

Tabela 1 – Classificação das Arquiteturas Reconfiguráveis

	Granularidade	**Acoplamento**	**Reconfiguração**	**Voltada para *multicore***	**Compartilha recursos reconfiguráveis entre os núcleos**
CHIMAERA Hauck et al. (1997)	Fina	Forte	Estática	Não	Não
PipeRench Goldstein et al. (1999)	Grossa	Fraco	Dinâmica	Não	Não
MorphoSys Singh et al. (2000)	Grossa	Fraco	Dinâmica	Não	Não
Warp Processor Lysecky, Stitt e Vahid (2004)	Fina	Fraco	Dinâmica	Não	Não
Beck et al. (2008)	Grossa	Forte	Dinâmica	Não	Não
Feng e Yang (2013)	Grossa	Fraco	Estática e Dinâmica	Não	Não
Gottlieb et al. (2002)	Fina	Fraco	Estática	Sim	Não
ReMAP Watkins e Albonesi (2010)	Fina	Forte	Dinâmica	Sim	Não
Thread Warping Stitt e Vahid (2011)	Fina	Fraco	Dinâmica	Sim	Não
CReAMS Rutzig, Beck e Carro (2013)	Grossa	Forte	Dinâmica	Sim	Não
Yan et al. (2014)	Fina	Fraca	Estática	Sim	Sim
HARTMP Souza et al. (2016)	Grossa	Forte	Dinâmica	Sim	Não
DREAMS **Proposta desta dissertação**	Grossa	Forte	Dinâmica	Sim	Sim

4 A Arquitetura Proposta

Nesta dissertação de mestrado se apresenta a proposta, implementação e avaliação de uma arquitetura reconfigurável denominada DREAMS (*Dynamic Reconfigurable Array for Multicore Systems*). A arquitetura DREAMS é constituída por três componentes principais: um conjunto de núcleos de processamento; um *Array* reconfigurável e um escalonador de configurações. O *array* reconfigurável é fortemente acoplado aos núcleos de processamento. Um visão esquemática da arquitetura é apresentada na Figura 15. Cada um destes componentes possui características peculiares que serão apresentadas e discutidas nas seções 4.1, 4.2 e 4.3.

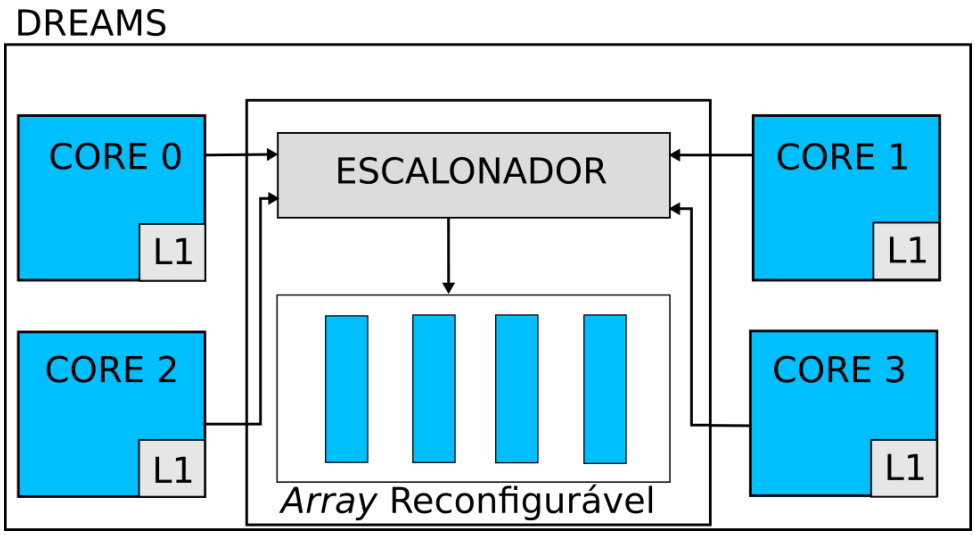


Figura 15 – Visão esquemática do DREAMS

4.1 Os Núcleos de Processamento

A arquitetura DREAMS é composta por 4 núcleos de processamento. Cada núcleo é um processador MIPS com pipeline de 5 estágios, e está acoplado a um tradutor binário. O tradutor binário foi adaptado da proposta de Beck (2009), e gera configurações, em tempo de execução, para serem executadas na arquitetura reconfigurável. Adicionalmente, um controlador e uma cache de configuração foram adicionadas ao núcleo de processamento. A cache de configuração armazena as configurações geradas pelo tradutor binário, enquanto que o controlador gerencia, baseando-se no valor do registrador *Program Counter* (PC), o que deve ser executado no *array* reconfigurável e no processador. Uma visão esquemática do núcleo de processamento é apresentada na Figura 16.

Cada núcleo de processamento possui caches L1 de dados e de instruções, ambas de 16KB. As caches L1 são privadas a cada núcleo. Um mecanismo baseado em diretório foi

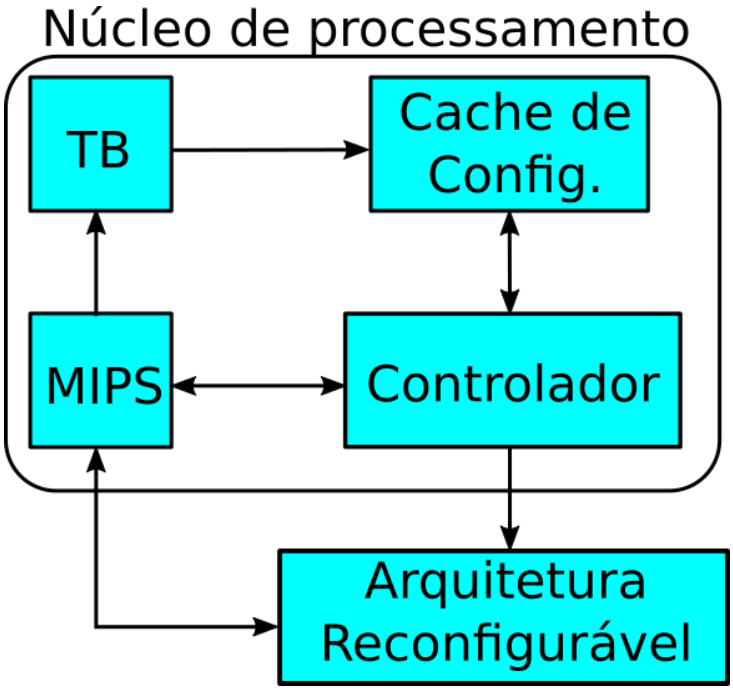


Figura 16 – Visão esquemática do núcleo de processamento.

implementado para manter a coerência das caches L1 dos núcleos. Foi utilizado o protocolo MSI (*Modified, Shared, Invalid*) como protocolo de coerência. Detalhes desse protocolo podem ser vistos em (PATTERSON; HENESSY, 2014). Adicionalmente, foi implementado o modelo de consistência sequencial. O modelo de consistência define quando uma escrita em uma cache privada de um dado núcleo estará visível para os demais. No modelo de consistência sequencial, o resultado de qualquer execução é o mesmo como se os acessos à memória realizados por cada processador fossem executados em ordem, e os acessos executados entre diferentes processadores fossem intercalados arbitrariamente conforme um ordem global (PATTERSON; HENESSY, 2014). Embora o modelo de consistência sequencial seja de simples implementação e uso, facilitando as atividades do projetista e do programador, ele reduz o desempenho do sistema, principalmente em sistemas com muitos processadores. Isso ocorre porque o acesso à memoria realizado por um dado núcleo só será iniciado quando todos os acessos anteriores tiverem sido finalizados.

4.1.1 O Tradutor Binário

O tradutor binário (TB) é um bloco de hardware que detecta dinamicamente sequências de instruções para serem executadas na arquitetura reconfigurável. As principais vantagens de se utilizar o tradutor binário são: compatibilidade de software (não precisa recompilar o código fonte) e redução no *time-to-market* do sistema reconfigurável. O TB implementado foi inspirado na proposta de Beck (2009).

O TB é responsável por gerar configurações em tempo de execução e, para que isto seja possível, ele avalia cada instrução executada pelo processador. Para cada instrução

avaliada, o TB verifica se a instrução é suportada pelo *array* reconfigurável (ver Tabela 2) e, caso seja, aloca essa instrução para a configuração corrente. Vale ressaltar que essa verificação e alocação é feita através de um algoritmo que foi implementado na forma de um pipeline de cinco estágios, que será explicado detalhadamente mais a frente. Depois da configuração gerada, ela então é armazenada em uma cache de configuração, indexada pelo valor do PC da primeira instrução. O fim da configuração é determinado pela chegada de uma instrução não suportada pelo *array* reconfigurável. Quando isso ocorre, os seguintes passos são executados: i) Salvar em cache a configuração que estava no TB; ii) Iniciar uma nova configuração.

Contudo, a decisão de salvar, em cache de configuração, uma configuração que acaba de ser gerada não é trivial. Sabe-se que existe um custo considerável na comunicação entre o processador de propósito geral e a arquitetura reconfigurável e, portanto, deve ser verificado quais configurações executadas no hardware reconfigurável trarão ganhos de desempenho. No trabalho de Beck et al. (2008), um requisito para uma configuração ser salva é ela ter pelo menos três instruções. Na arquitetura proposta nesta dissertação, com base na análise do custo de comunicação entre os núcleos de processamento e o *array* reconfigurável, bem como em resultados experimentais, chegou-se a conclusão que as configurações contendo no mínimo 20 instruções poderiam ser executadas no *array* reconfigurável. Vale ressaltar que essa restrição não se aplica nas configurações dos *loops* mais internos das aplicações, pois essas sempre são salvas, mesmo que possuam menos que 20 instruções. Os *loops* mais internos das aplicações são, geralmente, os trechos mais executados de uma aplicação e, portanto, mesmo com blocos básicos pequenos, consegue-se prover aceleração quando executado no *array* reconfigurável.

O TB é composto por tabelas que armazenam informações temporárias da configuração corrente. As tabelas utilizadas para implementação do TB são:

- **Tabela de escrita**: Armazena quais registradores serão escritos em uma configuração executada no *array* reconfigurável. Isso serve para realizar a verificação de dependências verdadeiras (RAW - *Read After Write*) e falsas (WAW - *Write after Write*) entre as instruções;

- **Tabela de recursos**: Armazena informação sobre a disponibilidade de unidades funcionais, ou seja, informa se um EP ou unidade de *lw/sw* está em uso (alocado para outra instrução). Essa tabela é utilizada para verificação de *hazards* estruturais;

- **Tabela de leitura**: Armazena quais registradores serão lidos em uma configuração executada no *array* reconfigurável. Essa tabela, além de informar os registradores fontes, serve para detecção de falsas dependências do tipo WAR - *Write After Read*;

- **Tabela de operação**: Armazena os códigos das operações para serem realizadas nas unidades funcionais;
- **Tabela de imediatos**: Armazena os operandos imediatos de uma configuração executada no *array* reconfigurável.

O TB foi implementado em hardware na forma de um pipeline de 5 estágios, sendo eles: DP (*Decode Phase*), DC (*Dependency Check*), AP (*Allocation Phase*), RR (*Register Renaming*) e UT (*Update Tables*). Esse pipeline é acoplado ao pipeline do processador MIPS que compõe o núcleo de processamento da arquitetura DREAMS (ver Figura 18). Os estágios do pipeline do processador MIPS são os mesmos apresentados em (PATTERSON; HENESSY, 2014):

- **IF** (*Instruction Fetch*): Realiza a busca de instruções e atualiza o PC (Program Counter);
- **ID** (*Instruction Decode*): Realiza a busca de operandos e decodifica a instrução;
- **EX** (*Execution*): Realiza a operação na ULA (Unidade Lógica e Aritmética);
- **MEM** (*Memory Access*): Realiza o acesso à memória de dados;
- **WB** (*Write-Back*): Faz a escrita do resultado no banco de registradores do processador.

O algoritmo de tradução Binária implementado foi inspirado na proposta apresentada em (BECK, 2009). Algumas adaptações foram feitas e serão detalhadas na seção 4.1.2. Antes de explicar o algoritmo e o que é realizado em cada estágio do pipeline do TB é importante destacar que a coluna reconfigurável é vista como uma matriz $6xN$, como pode ser visto na Figura 17. Na matriz da Figura 17, as linhas representam os recursos físicos de uma coluna (melhor descritos mais adiante). As colunas, por sua vez, representam a virtualização desses recursos físicos. Essa virtualização é possível porque as operações em diferentes colunas são feitas em tempos (ciclos) diferentes. Portanto, as N colunas representam as computações alocadas no tempo no *array* reconfigurável, enquanto que as linhas representa as computações alocadas no espaço e são executadas em paralelo (no mesmo ciclo).

Os estágios do TB são descritos a seguir:

- **DP** (*Decode Phase*): Decodifica a instrução, retornando o registrador alvo e os registradores operandos da instrução corrente;

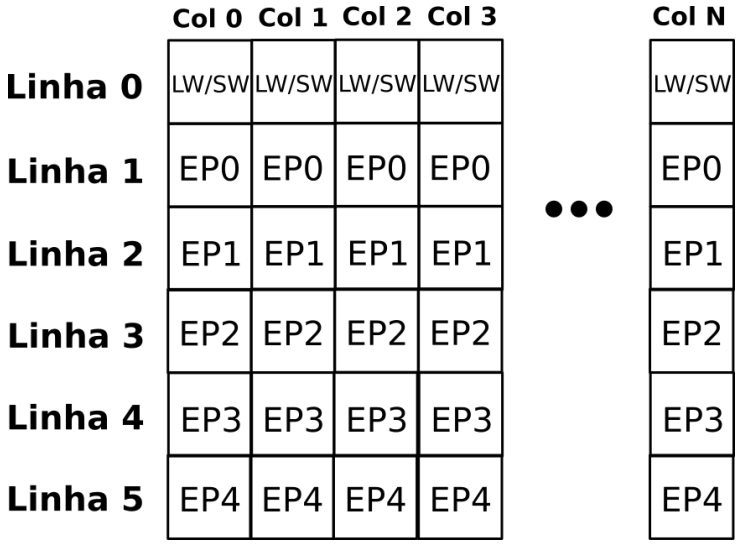


Figura 17 – Representação em Matriz do *Array* Reconfigurável

- **DC** (*Dependency Check*): Neste estágio de pipeline é realizada a verificação de dependências verdadeiras, ou seja, do tipo RAW (*Read after Write*). A verificação é feita realizando uma busca da coluna 0 à coluna N na tabela de escrita pelos registradores operandos retornados da fase de decodificação (DP), onde N é a última coluna alocada. O retorno desse estágio é uma coluna O onde a instrução corrente pode ser alocada, sem considerar a disponibilidade de recurso ainda. Se pelo menos um dos operandos for encontrado na tabela de escrita de uma coluna S, a coluna O será $S + 1$. Considerando a busca *bottom-up* (0 a N), a coluna S é a última onde pelo menos um dos operandos da instrução corrente aparece na tabela de escrita. Se nenhum dos operandos da instrução corrente existir em nenhuma coluna dessa tabela, a coluna O é igual a 0;

- **AP** (*Allocation Phase*): Este estágio é responsável por fazer a alocação da instrução corrente em uma determinada coluna e linha. O retorno desse estágio é uma coluna C e uma linha R que corresponde ao recurso alocado (unidade funcional de uma coluna) para a instrução corrente. Utilizando a coluna O retornada do estágio anterior (DC), é feita uma busca na tabela de recurso a partir da coluna O, ou seja, a coluna C é inicializada com o valor da coluna O. Para cada linha da coluna C é verificado se há recurso disponível para a instrução corrente, caso haja, a linha R será igual a linha disponível na coluna. Por outro lado, caso não haja recurso disponível na coluna C, a coluna C é incrementada ($C = C + 1$) e o processo de verificação de disponibilidade de recurso é repetido. Se o valor C atingir o número de colunas disponível na arquitetura reconfigurável e não tiver recurso disponível, a configuração então é encerrada e essa instrução não será alocada na configuração corrente.

- **RR** (*Register Renaming*): Como o próprio nome sugere, este estágio é responsável por renomear os registradores para eliminar as falsas dependências do tipo WAW

e WAR. Utilizando a coluna C retornada pelo estágio anterior, AP, é feita uma busca pelo registrador destino da instruções corrente a partir da coluna C até N, onde N é o número de colunas alocadas, na tabela de leitura e de escrita. Caso o registrador destino seja encontrado na tabela de leitura, detecta-se uma WAR. Caso o registrador destino seja encontrado na tabela de escrita, detecta-se um WAW. Para ambos os casos, o registrador destino será renomeado para um dos 16 registradores virtuais disponíveis e atualizará a entrada do registrador real na tabela de renomeação. Caso a busca não detecte WAW ou WAR, o registrador destino não será renomeado. Contudo, ainda falta garantir que os acessos futuros de um registrador renomeado sejam feitos corretamente. Então, nesse estágio de pipeline os registradores operandos da instrução corrente, que vieram da fase de decodificação e foram propagados através do pipeline, serão atualizados para apontar para o registrador virtual, caso eles tenham uma entrada na tabela de renomeação. Dessa forma, garante-se que os operandos lidos vão estar sempre apontado para o registrador correto, seja ele virtual ou real;

- **UT** (*Update Tables*): Neste estágio ocorrem as escritas nas tabelas. Todas as tabelas na linha R e coluna C, retornadas do estágio anterior, serão escritas com seus respectivos valores. Por exemplo, na tabela de leitura serão inseridos os valores dos registradores operandos da instrução, na tabela de operação será inserido o código da operação da instrução e assim sucessivamente.

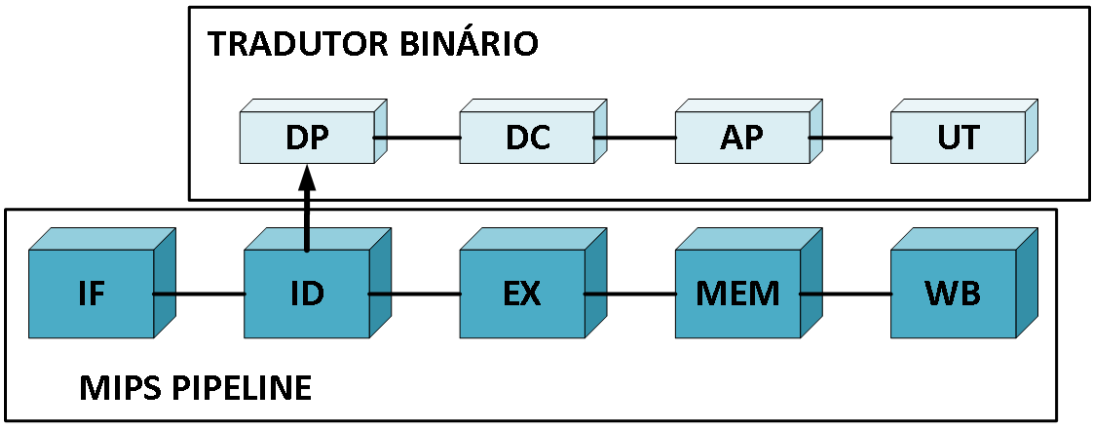


Figura 18 – Estágios do Tradutor Binário integrado acoplado a um núcleo MIPS pipeline

4.1.2 Modificações no Tradutor Binário

Devido às diferenças entre a arquitetura proposta em (BECK, 2009) e o *array* reconfigurável proposto por este trabalho, algumas modificações no tradutor binário tiveram que ser realizadas. Essas modificações foram feitas para permitir a eliminação de falsas dependências (WAR e WAW) e a execução especulativa no *array* reconfigurável.

Ambas aumentam a capacidade de exploração de ILP pelo *array* reconfigurável que, consequentemente, conseguirá prover mais aceleração nos trechos executados pelo *array*.

4.1.2.1 Tratando Falsas Dependências

As falsas dependências WAW e WAR também são conhecidas como dependências de nome, pois a dependência está no nome do registrador e não há valor transmitido entre as instruções (PATTERSON; HENESSY, 2014). Para entender melhor as falsas dependências, considere o seguinte exemplo:

```
1  add  $s0,$s4,$s2
2  sll  $s4,$t0,$t3
3  sub  $s4,$t0,$t8
```

Figura 19 – Exemplo de trecho de código com falsa dependência.

Nesse trecho de código existe uma falsa dependência do tipo WAR entre a instrução *add* e *sll*. Isso impossibilita a execução em paralelo dessas duas instruções, pois teria uma escrita e leitura ao mesmo tempo no registrador *$s4*, o que pode gerar problema com a coerência dos dados. Também há uma falsa dependência do tipo WAW entre a instrução *sll* e *sub*, o que impossibilita a execução em paralelo dessas instruções, pois duas instruções poderiam escrever no registrador *$s4* ao mesmo tempo. Observe que essas dependências são somente de nome, pois não há um fluxo de dados entre essas instruções dependentes, diferentemente do que ocorre em uma dependência verdadeira RAW. A Figura 20 mostra o mesmo código utilizado no exemplo com a eliminação dessas dependências somente modificando o nome dos registradores para registradores virtuais (ou escondidos) *X* e *Y*.

```
1  add  $s0,$s4,$s2
2  sll  X,$t0,$t3
3  sub  Y,$t0,$t8
```

Figura 20 – Trecho de código com renomeação de registrador.

Na arquitetura reconfigurável proposta em (BECK, 2009), as falsas dependências são eliminadas através da tabela de contexto que armazena um ponteiro para o valor corrente de um operando. Essa tabela de contexto elimina as dependências WAR e WAW fazendo um adiantamento de dados (*bypass*) dos valores de saída das unidades funcionais. Essa abordagem foi inspirada no algoritmo de Tomasulo (TOMASULO, 1967) que faz a renomeação de registradores utilizando suas estações reservas, eliminando a necessidade de ler um operando necessariamente do banco de registradores. No *array* reconfigurável proposto neste trabalho, todas as operações são salvas em registrador. Portanto, não é viável o uso do mesmo mecanismo adotado em (BECK, 2009). Por este motivo, o trabalho

proposto nesta dissertação adotou uma outra técnica muito conhecida para eliminação de falsas dependências: renomeação de registradores. Essa técnica consiste em renomear os registradores que possuem falsa dependência, como demonstrado anteriormente na Figura 20.

Para dar suporte à técnica de renomeação de registradores, 16 registradores virtuais foram adicionados no *array* reconfigurável. Adicionalmente, o algoritmo do tradutor binário teve que ser modificado para fazer a renomeação de registradores dinamicamente. Uma tabela que armazena as referências atuais para os registradores renomeados, denominada tabela de renomeação, foi adicionada no tradutor binário. A modificação no tradutor binário para suportar a renomeação de registradores foi feita adicionando um estágio de pipeline denominado de *Register Renaming* (RR) que fica entre os estágios *Allocation Phase* e *Update Table*.

4.1.2.2 Dando Suporte à Execução Especulativa

A quantidade de ILP disponível dentro de um único bloco básico, normalmente, é muito pequena. Em média, um bloco básico com instruções MIPS tem de 3 a 6 instruções (PATTERSON; HENESSY, 2014). Uma solução dada normalmente para esse gargalo é explorar ILP de múltiplos blocos básicos. Isso é feito utilizando um preditor dinâmico de salto e permitindo a execução especulativa. Um preditor dinâmico de salto busca instruções de acordo com sua predição, assumindo que o salto vai ser tomado ou não, e isso leva à redução de bolhas inseridas no pipeline já que a próxima instrução buscada não vai esperar avaliação do salto pelo processador. Já na execução especulativa, a arquitetura busca e executa instruções assumindo que sua predição de salto está correta. Note que, na execução especulativa, precisa-se implementar um mecanismo para tratar os casos onde a predição errou.

No trabalho de Beck (2009), para dar suporte a execução especulativa foram feitas modificações no tradutor binário e na arquitetura reconfigurável. Para geração de configuração com execução especulativa no tradutor binário foi utilizado um ponto de saturação igual a 2, que indica que aquele salto deve ter tomado o mesmo caminho 2 vezes para que se agrupe as configurações dos blocos básicos. A Figura 21 exemplifica o uso do ponto de saturação. Nesse exemplo, o bloco básico 1 é encontrado e alocado no array reconfigurável normalmente. Isso é representado na Figura 21(b). Depois disso, o salto condicional pode tomar dois caminhos, o bloco básico 2 ou o 3. Nesse exemplo, o caminho seguido pelo salto foi 3. Então, o ponto de saturação para esse salto é incrementado, ficando com o valor 1. Da próxima vez que o bloco básico 1 for encontrado novamente, ele não necessitará alocar as instruções no *array* reconfigurável, pois já foi feito isso previamente na primeira passagem por esse bloco básico. Caso o caminho tomado pelo salto for o mesmo, o ponto de saturação será incrementado novamente e atingirá o valor 2. Nesse

ponto, as instruções do bloco básico 3 serão alocadas na mesma configuração do bloco básico 1. Isso é mostrado na Figura 21(c). Por outro lado, caso o caminho tomado pelo salto seja diferente, o ponto de saturação seria decrementado e o processo se repetiria.

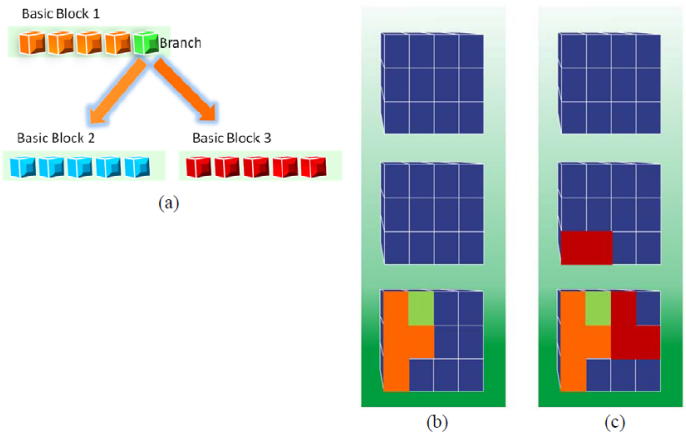


Figura 21 – Funcionamento do ponto de saturação. Retirado de (BECK, 2009).

Como mencionado anteriormente, a arquitetura que executa instruções de forma especulativa necessita de mecanismos para tratar casos onde a predição foi incorreta. Para lidar com isso, o *array* reconfigurável proposto em (BECK, 2009) adicionou um novo grupo de unidade funcional para executar os saltos condicionais. Adicionalmente, novos multiplexadores e *buffers* foram inseridos na arquitetura. Os *buffers* são utilizados como memória de rascunho, para que as escritas das instruções especulativas não sejam feitas no próprio banco de registradores, e os multiplexadores são feitos para fazer o *write-back* das instruções especulativas quando o salto for avaliado e saber que sua predição estava correta. Caso ocorra erro na predição, os valores dos buffers são descartados. É importante destacar que, a proposta de (BECK, 2009) permite a execução especulativa de múltiplos níveis, então, se ocorre erro na predição, os valores descartados são somente referente a esse nível de especulação, e os valores referentes ao nível anterior serão salvos e, posteriormente, feito o *write-back* para o processador.

Na arquitetura proposta nesta dissertação, somente um nível de especulação é suportado. Um contador para controlar o nível de especulação foi inserido no tradutor binário. Quando encontra-se um salto condicional, esse contador é incrementado e caso chegue outro salto condicional, aquela configuração é interrompida por permitir somente um nível de especulação. Não foi utilizado ponto de saturação e o tradutor binário gera sempre a configuração referente à primeira passagem do código. Utilizando o exemplo da Figura 21 como base, e o caminho tomado pelo salto sendo o bloco básico 3, a configuração gerada seria uma configuração referente a união dos blocos básico 1 e 3. Essa configuração seria gerada na primeira passagem. O motivo da não utilização do ponto de saturação é o custo de restaurar o estado de uma configuração já gerada no tradutor binário, pois isso requer que uma configuração seja buscada e depois convertida para todas as tabelas

que existem no tradutor binário. O autor (BECK, 2009) não detalhou como o estado de uma configuração é restaurado, mas essa restauração tem um custo significativo. Além do contador que controla o nível de especulação, uma nova tabela, tabela de *rollback*, foi inserida no tradutor binário para informar ao *array* reconfigurável quais valores devem ser salvos quando a predição daquela configuração estiver incorreta. Já nas modificações no *array* reconfigurável, as abordagens foram bem semelhantes. Para dar suporte à execução dos saltos condicionais, essas operações foram incluídas nas unidades funcionais. Um *buffer*, denominado de *buffer* de reordenação, foi inserido para armazenar os valores da computação de instruções que são executadas de forma especulativa até que a predição seja avaliada como correta.

Apesar da arquitetura proposta nesta dissertação sempre gerar a configuração do caminho tomado pelo salto na primeira passagem, foi implementado mecanismo para verificar se aquele caminho tomado está errando sua predição. Esse mecanismo foi implementado no controlador e será explicado na seção 4.1.3.

4.1.3 O controlador

O controlador possui três funções principais: verificar trechos que devem ser executados na arquitetura reconfigurável, controlar a execução na arquitetura reconfigurável e remover configurações que erram duas vezes consecutivas sua predição.

Para verificar trechos que devem ser executados na arquitetura reconfigurável, o controlador verifica se existe uma configuração na cache de configuração para o valor de PC corrente. Caso não exista, o fluxo de execução continua sendo feito no processador. Por outro lado, se o controlador detectar que existe uma configuração para o PC corrente, ele buscará a configuração da cache e enviará sinais de controle para o processador informando-o que iniciará a execução no *array* reconfigurável. Inicialmente, o controlador irá configurar o *array* para que ele esteja pronto para executar. A configuração do *array* é feita com informações de controle sobre a execução, como por exemplo: qual PC de retorno ao fim da execução da configuração. Em paralelo à essa configuração, também estará sendo feito o carregamento do contexto de entrada. Entende-se por contexto de entrada os valores do banco de registradores do processador no momento em que se inicia uma configuração. Nesse carregamento é feita uma cópia do banco de registradores do processador para o banco de registradores do *array* reconfigurável. Tudo isso é considerado como fase de configuração do *array*. Depois da fase de configuração, caso não ocorra cache *miss*, a cada ciclo, o controlador enviará uma palavra de configuração para executar no *array* reconfigurável. Se houver cache *miss*, a próxima palavra só será enviada quando o cache *miss* for resolvido. Quando chegar ao fim da execução da configuração, o contexto de saída é salvo realizando-se uma cópia do banco de registradores do *array* reconfigurável para o processador. Então, depois de finalizada a execução no *array* reconfigurável e salvo seu

contexto de saída, o controle da execução é devolvido para o processador.

Existe um ponto importante de se destacar sobre o fluxo descrito anteriormente. O controlador não necessariamente retornará o controle de execução de volta para o processador. Isso se deve a um mecanismo implementado no controlador, que visa mitigar o custo da configuração do *array* reconfigurável em casos de configurações de *loops*. Entende-se por configurações de *loops*, uma configuração que, ao final de sua execução, o PC de retorno é igual ao PC de início daquela mesma configuração. Esse mecanismo serve para que configurações de *loops* tenham seu custo de reconfiguração reduzido ao mínimo possível e, consequentemente, resultem em maior aceleração. Perceba que, sem esse mecanismo, a arquitetura reconfigurável, ao final da execução, devolveria o controle da execução para o processador, que depois iria novamente transferir o controle da execução para o *array* reconfigurável. Configurações com esse tipo de característica normalmente pertencem a um *loop* mais interno de uma aplicação e, portanto, otimizar sua execução reduz o tempo total da execução de forma significativa.

Para evitar que uma configuração que esteja errando muito sua predição afete o desempenho da arquitetura, foi implementado um mecanismo inspirado no preditor de 2 bits. Esse mecanismo foi implementado adicionando uma tabela associativa de 16 entradas que armazena o valor do PC da configuração e a quantidade de erros consecutivos. Quando o número de erros chega a 2, é enviada uma solicitação de invalidação para a cache de configuração. Dessa forma, a configuração é invalidada e aquele trecho de código será executado no processador para que o tradutor binário gere uma nova configuração. Contudo, essa abordagem possui um problema. A configuração que vai ser gerada novamente será feita considerando o caminho do salto tomado na próxima execução. Portanto, é possível que aconteça muitas invalidações em uma configuração, dependendo do padrão dos saltos da aplicação.

No entanto, para as aplicações selecionadas, os trechos que foram mapeados para executar na arquitetura reconfigurável possuíam um comportamento muito regular. Com isso, não ocorreram muitas invalidações. Os resultados de taxa de erro de predição serão mostrados no capítulo 5.

4.2 O *Array* Reconfigurável

O *array* reconfigurável é composto por 4 colunas reconfiguráveis e é fortemente acoplado aos núcleos de processamento. Cada coluna possui 3 EPs e 1 unidade de *load/store*. Os EPs de uma dada coluna do *array*, prioritariamente, atenderão as demandas de computação de um dado núcleo de processamento, mas seu uso por outros núcleos é possibilitado por um escalonador (seção 4.3). O *array* reconfigurável possui 4 bancos de registradores para armazenar o contexto de entrada e saída de cada núcleo, ou seja, BR0 é banco de

registradores referente ao núcleo 0, BR1 é o banco de registradores referente ao núcleo 1 e assim sucessivamente. A unidade de *load/store* não é compartilhada entre os núcleos de processamento e, portanto, cada núcleo só poderá utilizar da unidade de *load/store* pertencente à coluna a qual tem acesso prioritário. Todas as computações realizadas no *array* reconfigurável são feitas lendo e escrevendo valores nos bancos de registradores do *array*. Uma visão geral da organização do *array* reconfigurável é apresentada na Figura 22.

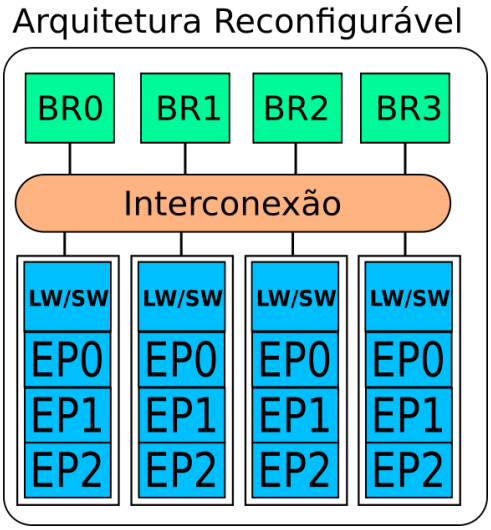


Figura 22 – Organização do *array* reconfigurável do DREAMS.

4.2.1 A Coluna Reconfigurável

A coluna reconfigurável é dotada de 3 elementos de processamento e uma unidade de *Load/Store* e é controlada por uma máquina de estados (FSM - *Finite State Machine*). O diagrama de blocos da coluna pode ser visto na Figura 23. A máquina de estados controlará a execução na coluna reconfigurável enviando os códigos de operações correspondentes à configuração recebida para serem executadas nos EPs e na unidade de *load/store*. Além disso, a máquina de estados também envia sinais de controle para o controlador do núcleo de processamento notificando-o quando ocorrer cache *miss*, erro na predição de um salto ou chegar ao fim da execução de uma configuração. A configuração, além de informar as operações realizadas na coluna, informa quais registradores precisam ser lidos. Os registradores são lidos dos bancos de registradores (BR) do *array* reconfigurável através do sistema de interconexão. Os resultados da computação da coluna são salvos escrevendo os valores nos bancos de registradores do *array*. A escrita também é realizada utilizando o sistema de interconexão e os registradores destinos são informados na configuração.

Cada coluna é prioritariamente dedicada a um núcleo, mas pode ser utilizada por outros núcleos. Sendo assim, cada núcleo possui uma coluna como recurso prioritário, mas pode usar recursos ociosos de outras colunas. A princípio, o resultado das operações realizadas em uma coluna é salvo no banco de registradores daquela coluna. Entretanto,

quando o EP de uma coluna é emprestado para execução da configuração de outro núcleo, o resultado da operação será salvo no banco de registradores da coluna prioritária do núcleo que usa o recurso, e não do núcleo que cede o recurso.

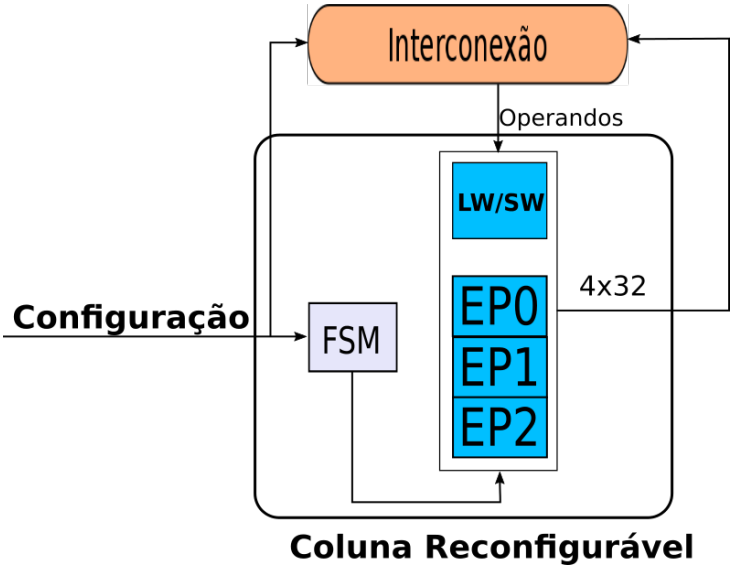


Figura 23 – Visão esquemática da coluna reconfigurável

4.2.2 O Elemento de Processamento

O EP (ver Figura 24) é composto por uma ULA e recebe como entrada dois operandos (Op. 1 e Op. 2) e a configuração que informa qual operação a ser realizada. Os valores dos operandos de entrada também são definidos pela configuração e podem ser um valor lido de um dos bancos de registradores do *array* reconfigurável ou um valor imediato, fornecido pela própria configuração. A saída do EP será escrita em um dos bancos de registradores do *array* reconfigurável. Se o EP, que necessariamente pertence a uma coluna e, portanto, prioritariamente atende a um dos núcleos, estiver, momentaneamente atendendo a outro núcleo, a saída será salva no banco de registradores da coluna dedicada ao núcleo que o utiliza e não no banco de registradores da coluna a qual pertence. A configuração informa qual banco e qual registrador será escrito por cada um dos EPs. Sendo assim, uma configuração determina os dados de entrada no EP, sua operação e onde escrever o resultado da computação. As ULAs dos EPs realizam todas as operações lógicas e aritméticas necessárias à execução de instruções MIPS. A multiplicação, entretanto, por implicar na necessidade de inclusão de um operador crítico em termos de área, foi incluída somente em uma ULA, a ULA do EP0 (ver Figura 23). Como se pode observar, o EP utiliza granularidade grossa, pois opera sobre palavras.

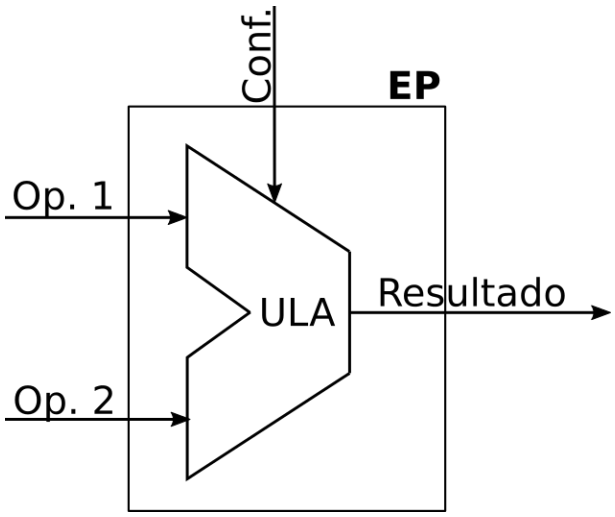


Figura 24 – Organização do elemento de Processamento

4.2.3 A Unidade de *Load/Store*

A unidade de *load/store* executa as operações de acesso à memória no *array* reconfigurável e é composta por um somador e por um controlador. Uma visão esquemática da unidade de *load/store* pode ser vista na Figura 25. O somador faz o cálculo de endereço utilizando as entradas fornecidas pela configuração. Como as instruções executadas seguem a ISA (*Instruction-Set Architecture*) MIPS, os cálculos de endereço de acesso à memória são feitos utilizando um registrador base e um imediato. Então, a entrada *Op. 1* é um valor lido de um registrador base e *Imm* é um valor imediato fornecido pela configuração. Já o controlador decodifica a configuração que está sendo recebida na unidade de *load/store* e verifica se é uma requisição de leitura ou escrita na memória. Todos esse sinais de saída, *wren*, *address* e *data*, são enviados para a cache L1 de dados que é compartilhada entre o processador e o *array* reconfigurável.

Observe que, diferentemente dos EPs, a latência das operações na unidade de *load/store* pode levar muito mais do que um ciclo de *clock* nos casos em que ocorrem cache *miss*. Sendo assim, faz-se necessário um mecanismo para controlar o envio das próximas palavras de configuração somente quando a sua predecessora tiver finalizado suas operações. Então, para sincronizar o envio de novas palavras de configuração, a máquina de estado da coluna reconfigurável notifica que ocorreu um cache *miss* para o controlador do núcleo de processamento que, por sua vez, não enviará palavras de configuração até que o cache *miss* seja resolvido.

4.2.4 A configuração

Uma configuração define um conjunto de operações que devem ser executadas na coluna reconfigurável durante um ou mais ciclos. Isso é feito mapeando um conjunto de instruções MIPS (ver Tabela 2) para a execução na coluna reconfigurável. A versão atual

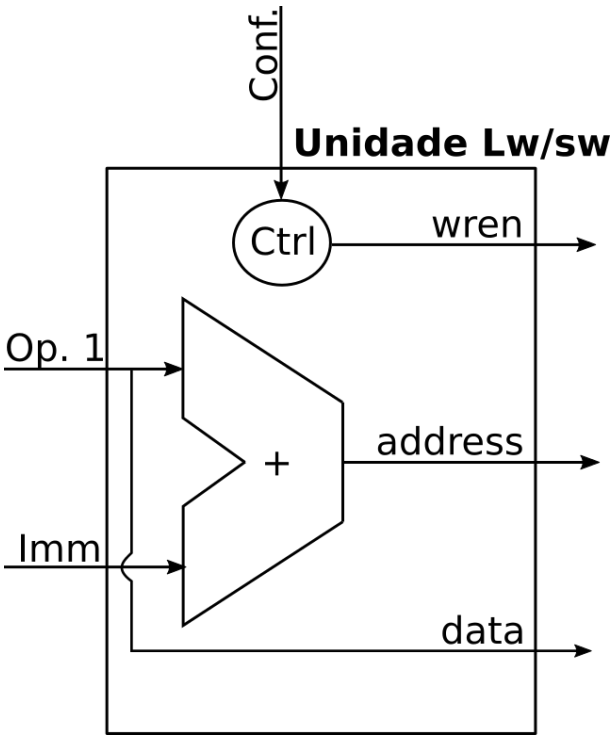


Figura 25 – Organização da unidade de *load/store*

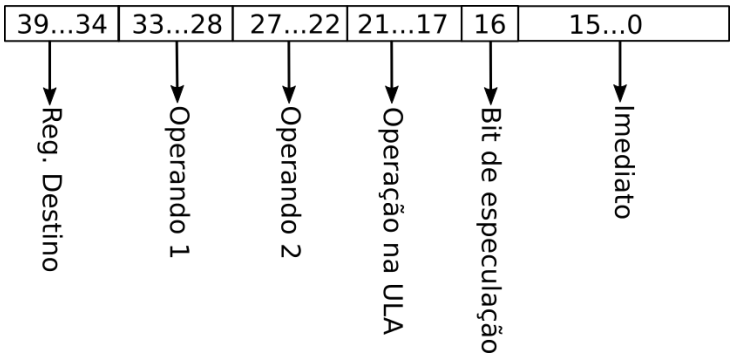
da coluna reconfigurável não suporta operações de ponto flutuante.

Tabela 2 – Instruções MIPS suportadas pelo *array* reconfigurável

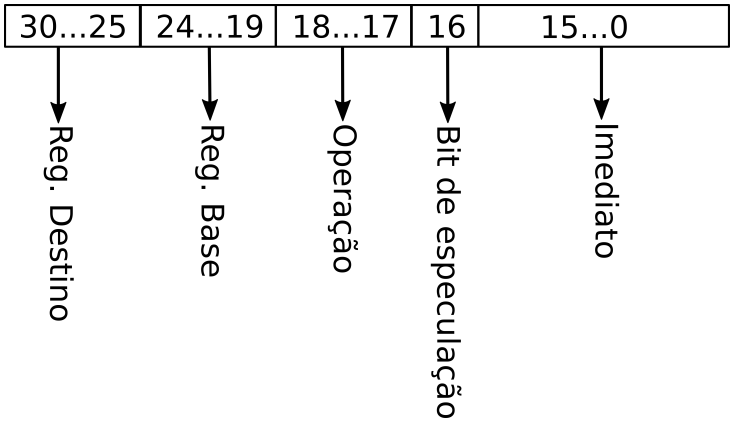
Categoria	**Instruções**
Aritméticas	*add, addi, mult, sub*
Transferência de Dados	*lw, sw, mfhi, mflo*
Lógicas	*or, ori, and, andi, xor, nor, slt, slti*
Saltos condicionais	*beq,bne,bnez,beqz,bltz*
Deslocamento de Bit	*srl, srli, sll, slli, sra, srai, lui*

Uma configuração é composta por palavras de configurações, onde cada palavra define as operações realizadas em um ciclo de execução na coluna reconfigurável. Para definir as operações realizadas em cada ciclo pela coluna reconfigurável, é preciso configurar os elementos de processamento e a unidade de *load/store*. Portanto, uma palavra de configuração é responsável por configurar os 3 elementos de processamento e a unidade de *load/store*. Contudo, a palavra de configuração utilizada neste trabalho configura uma unidade de *load/store* e até cinco elementos de processamento. Isso foi feito para permitir a exploração de mais elementos de processamentos e ILP, utilizando o mecanismo implementado na arquitetura DREAMS de compartilhamento de EPs entre os núcleos de processamento. Dessa forma, o excedente da configuração será executado em EPs de outras colunas reconfiguráveis que não estiverem em uso. Os campos de configuração de um elemento de processamento em particular podem ser vistos na Figura 26a e os campos

de configuração da unidade de *load/store* podem ser vistos na Figura 26b.



(a) Bits de configuração do EP.



(b) Bits de configuração da unidade de *load/store*

Figura 26 – Campos de configuração do EP e da unidade de *lw/sw*.

O mapeamento de um bloco de código MIPS no *array* reconfigurável é feito gerando-se uma configuração que é composta de palavras de configurações. A quantidade de palavras de configuração gerada para configurar esse bloco de código é influenciada pelo tamanho do bloco de código, pela dependência de dados existente entre essas instruções e pela quantidade de unidades funcionais disponíveis.

4.3 O Escalonador de Configurações

O escalonador de configurações aloca as configurações para serem executadas no *array* reconfigurável. Cada controlador de núcleo de processamento enviará sua palavra de configuração para o escalonador que, seguindo algumas regras, irá alocar a configuração para ser executada no *array* reconfigurável. O escalonador receberá, portanto, como entrada as palavras de configurações de cada núcleo e gerará como saída a configuração para o *array* reconfigurável como um todo. A organização do escalonador de configuração pode ser vista na Figura 27.

A implementação do escalonador de configuração foi feita em um pipeline de dois estágios. No primeiro estágio, dois blocos de lógica combinacional recebem como entrada

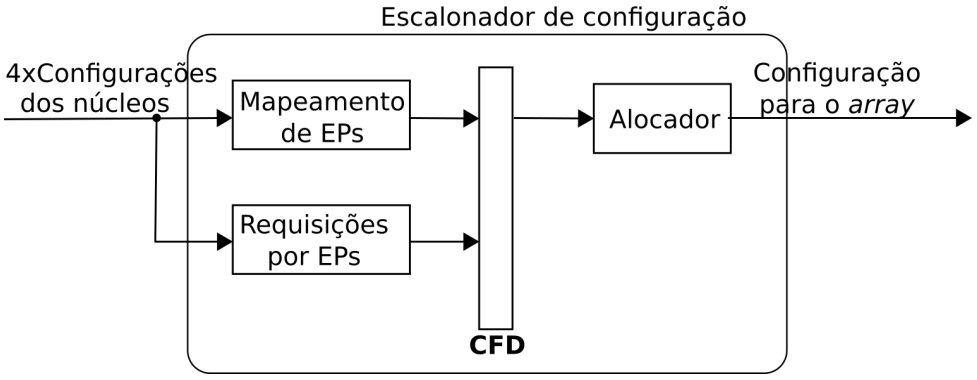


Figura 27 – Organização do escalonador de configuração.

as palavras de configuração de cada núcleo e escrevem os seus resultados no registrador de pipeline CFD (*Configuration Fetch and Decode*). No bloco de mapeamento dos EPs, é feita uma verificação da disponibilidade dos EPs no *array* reconfigurável. A saída desse bloco é um mapa de bits para cada coluna. Um exemplo de saída é 000, indicando que nessa coluna os 3 EPs estão disponíveis. No bloco de requisição por EPs, é verificado se algum núcleo precisa de um ou dois EPs emprestados. Um núcleo precisará de um EP emprestado quando sua palavra de configuração tiver mais que os 3 EPs, pois cada núcleo tem prioridade em uma coluna reconfigurável. A saída desse bloco são dois bits para cada núcleo. Cada bit sinaliza a necessidade de um EP emprestado. Note que o máximo de EPs que um núcleo pode solicitar é dois, pois a palavra de configuração aloca no máximo cinco EPs e sua coluna reconfigurável possui três. No segundo estágio, utilizando as informações que são geradas no primeiro estágio de pipeline, referentes à disponibilidade dos EPs e à requisição por EPs dos núcleos, o alocador gera a configuração do *array* reconfigurável. O alocador respeita a prioridade de cada núcleo, assim, primeiramente, aloca as configurações de cada núcleo na sua coluna reconfigurável, depois, se tiver EPs ociosos, ele atende à demanda dos núcleos que possam estar necessitando de EPs. Caso um núcleo necessite de EP e não tenha EP disponível no *array*, a palavra de configuração é dividida em duas e as operações não executadas são alocadas na coluna prioritária deste núcleo no ciclo seguinte.

4.4 Discussão

A arquitetura proposta neste trabalho usa menos EPs que os trabalhos encontrados na literatura, apenas uma coluna com 3 EPs e 1 unidade de *Load/Store* por núcleo, e possui um subsistema de interconexão de baixa complexidade. Além disso, o sistema é homogêneo, tanto na arquitetura quanto na organização, mas se comporta como um sistema *multicore* heterogêneo na organização, onde a quantidade de EPs utilizada por um determinado núcleo será definida pela configuração gerada. Isso faz com que a arquitetura reconfigurável execute as aplicações com eficiência, mesmo quando os núcleos têm cargas de trabalho diferentes, pois o *array* reconfigurável de cada núcleo irá se adaptar à carga

de trabalho de cada núcleo utilizando a virtualização de suas colunas. As vantagens da arquitetura aqui proposta em relação aos trabalhos mencionados que são voltados para o ambiente *multicore* são:

- Diferentemente do ReMAP (WATKINS; ALBONESI, 2010), no DREAMS os quatro núcleos do processador podem executar trechos de código em lógica reconfigurável ao mesmo tempo e a arquitetura não depende de ferramentas de sofware para geração de configuração. As configurações são geradas dinamicamente via tradutor binário, portanto, não há necessidade de modificação no código binário das aplicações nem de recompilação. Além disso, o ReMAP usa granularidade fina enquanto que o trabalho aqui proposto usa granularidade grossa. Arquiteturas reconfiguráveis de granularidade grossa possuem tempo de reconfiguração menor, consomem menos energia e utilizam menos memória para armazenamento de configuração do que as arquiteturas de granularidade fina ("THEODORIDIS; SOUDRIS; VASSILIADIS, "2007").

- No CReAMS um hardware reconfigurável é acoplado a cada núcleo do processador. O processador tem acesso exclusivo ao seu hardware reconfigurável. Isso se torna ineficiente quando tem-se cargas de trabalho desbalanceadas entre as *threads*, pois o hardware reconfigurável dos núcleos onde tem-se pouco trabalho computacional será subutilizado. Para mitigar esse problema, o DREAMS propõe um arquitetura reconfigurável que é compartilhada entre os núcleos do processador e fornece lógica reconfigurável sob demanda para os núcleos do processador *multicore*.

- Diferentemente do trabalho proposto em (YAN et al., 2014), o DREAMS utiliza granularidade grossa e compartilha os EPs entre os núcleos. Além disso, o DREAMS gera as configurações dinamicamente utilizando tradutor binário, enquanto que em (YAN et al., 2014) as configurações são geradas manualmente. Para dar suporte à execução em lógica reconfigurável, Yan et al. (2014) estenderam o conjunto de instruções. Com isso, para que o código legado possa usufruir dos benefícios da inserção da arquitetura reconfigurável no sistema, alterações no código binário devem ser feitas. No DREAMS, nenhuma modificação no código binário das aplicações é necessária para que se utilize a arquitetura reconfigurável e se usufrua do ganho de desempenho que ela fornece.

- O HARTMP propõe uma arquitetura heterogênea na organização, ou seja, quantidades diferentes de unidades funcionais reconfiguráveis disponíveis para cada núcleo, com a intenção de melhorar o desempenho em aplicações onde o TLP é baixo ou desbalanceado. Então, pode-se alocar DAPs com mais recursos para os núcleos onde há uma carga maior de trabalho. Ao invés de tentar solucionar o problema de *threads* com cargas de trabalho desbalanceadas entre os núcleos do processador alocando

uma quantidade de recursos diferente para cada núcleo, o DREAMS propõe um arquitetura reconfigurável que é compartilhada entre os núcleos do processador e tem seus recursos alocados dinamicamente por um escalonador de configurações. Observe que a quantidade de recursos que o HARTMP aloca para cada núcleo é definida estaticamente e, uma vez determinada, não há como ter modificações em tempo de execução.

5 Resultados

Este capítulo mostrará o impacto no desempenho da inserção do *array* reconfigurável no processador *multicore*. Quatro aplicações foram utilizadas para avaliação de desempenho: decomposição *Lu* do benchmark openMP (DORTA; RODRIGUEZ; SANDE, 2005), *bitcount* do ParMiBench (IQBAL; LIANG; GRAHN, 2010), multiplicação de matriz e laplaciano. Para cada aplicação foi gerada uma versão em linguagem C, para ser compilada utilizando o *cross-compiler* GNU GCC para MIPS, e uma versão em linguagem Go, para ser compilada utilizando o compilador resultante da utilização do *framework COGNITE*, desenvolvido no laboratório CESLa (*Circuits and Embedded System Lab*) (CARVALHO, 2018). Uma comparação dos códigos gerados pelos dois compiladores é feita considerando-se o ILP, tamanho dos blocos básicos e número de instruções de acesso à memória que o compilador gera. A predição de salto das configurações geradas e seu impacto no desempenho da arquitetura também é avaliada. Por fim, é feita uma avaliação do simulador desenvolvido em *SystemC* considerando-se a vazão de instruções que ele consegue prover.

5.1 Análise de Desempenho

Para análise de desempenho da arquitetura DREAMS, uma comparação com um processador *multicore* sem a arquitetura reconfigurável foi feita para verificar qual a aceleração provida pela inserção da DREAMS no sistema. O processador *multicore* utilizado na versão com e sem a arquitetura DREAMS é o mesmo e é composto por 4 núcleos MIPS pipeline de 5 estágios e possui despacho único. A memória cache também é a mesma nas duas versões e possui a mesma capacidade de armazenamento: 16 KB de cache L1 de instruções e 16 KB de cache L1 de dados. Foi utilizado um modelo baseado em diretório para manter a coerência entre as caches L1 dos núcleos do processador e um modelo de consistência sequencial.

As aplicações selecionadas como *benchmarks* foram: *Lu*, *Bitcount*, Laplaciano e multiplicação de matrizes. Essas aplicações foram selecionadas por terem alto TLP, ou seja, alto paralelismo na execução de suas *threads*. Apesar de terem essa característica em comum, cada uma dessas aplicações possui um comportamento diferente em suas *threads*. O *bitcount* foi selecionado para verificar como a DREAMS se comporta com aplicações *control-flow*, já que essa aplicação possui muitos saltos e blocos básicos pequenos. A multiplicação de matriz foi selecionada, pois suas *threads* possui computação intensiva e essa característica pode favorecer a arquitetura aqui proposta. O Laplaciano também possui computação intensiva em suas *threads*, mas, diferentemente da multiplicação de matriz, no final de cada iteração do cálculo do pixel, possui um trecho composto por um *if/else*. Essa

aplicação foi adicionada ao *benchmark* para analisar o impacto da inserção de *if's* dentro de um laço de repetição e o impacto de possíveis predições erradas no momento da geração da configuração, uma vez que os saltos condicionais dos *if's* tem um comportamento menos regular que os dos saltos condicionais dos laços de repetição. A aplicação *Lu* foi selecionada pois suas *threads* possuem trechos *controw-flow* e de computação intensiva.

5.1.1 Multiplicação de Matrizes

A aplicação multiplicação de matrizes recebe como entrada duas matrizes 20x20 e gera como saída o resultado da multiplicação dessas duas matrizes. Na versão desenvolvida em C e compilada com o *cross compiler* GCC, a inserção da arquitetura DREAMS conseguiu acelerar a aplicação em 43%. Duas configurações foram geradas pelo tradutor binário com ILP médio de 1,55. Nesta aplicação, 95,3% do tempo de execução da aplicação foi executada na arquitetura reconfigurável. O número máximo de operações alocadas para serem executadas em paralelo foi 3. Isso significa que, para essa aplicação, e usando o GCC, a coluna dedicada a cada processador foi o suficiente para execução da aplicação. O tamanho médio das configurações foi de 35 instruções sendo 10 de acesso à memória. Por conta da configuração possuir o salto condicional que controla um laço de repetição, a taxa média de erro de predição da configuração ficou em 3%.

No código desenvolvido em GO e compilado utilizando o compilador da *framework* COGNITE, a inserção da arquitetura DREAMS acelerou a aplicação em 30,99%. Somente uma configuração foi gerada pelo tradutor binário. A configuração gerada foi a do *loop* mais interno e possui 27 instruções, duas delas sendo de acesso à memória. O ILP médio foi de 2,07 e, no máximo, 5 instruções foram executadas em paralelo na arquitetura reconfigurável. A execução na arquitetura reconfigurável correspondeu a 82,2% do tempo de execução da aplicação. A taxa média de erro de predição da configuração ficou em 4,7%.

Mesmo com um ILP maior, o código gerado pelo compilador GO teve menos aceleração que o código gerado pelo GCC. A aceleração foi menor, pois o código gerado pelo compilador GO já é mais otimizado e, sem a arquitetura DREAMS, o tempo de execução já é acelerado em 25,63% quando comparado com código gerado pelo compilador GCC. Além disto, a diferença de ILP médio não foi significativa para compensar os blocos básicos maiores que o GCC gera. Além disso, o código gerado pelo GCC teve duas configurações enquanto que o código da versão em GO teve somente uma. Isso implica em mais uso da arquitetura reconfigurável e, portanto, maior aceleração. Contudo, a versão em GO, conseguiu executar mais que 3 instruções em paralelo na arquitetura reconfigurável e utilizar do escalonador de configuração. O código gerado pelo compilador GO, sem o uso da arquitetura DREAMS, acelerou em 25,63% quando comparado com a versão em gerado pelo compilador GCC. Com o uso da arquitetura DREAMS, o código gerado pelo compilador GO foi, aproximadamente, 10% mais rápido que o código gerado pelo

compilador GCC. Uma comparação do tempo de execução, em ciclos, das execuções com o código GO e GCC pode ser vista na Figura 28.

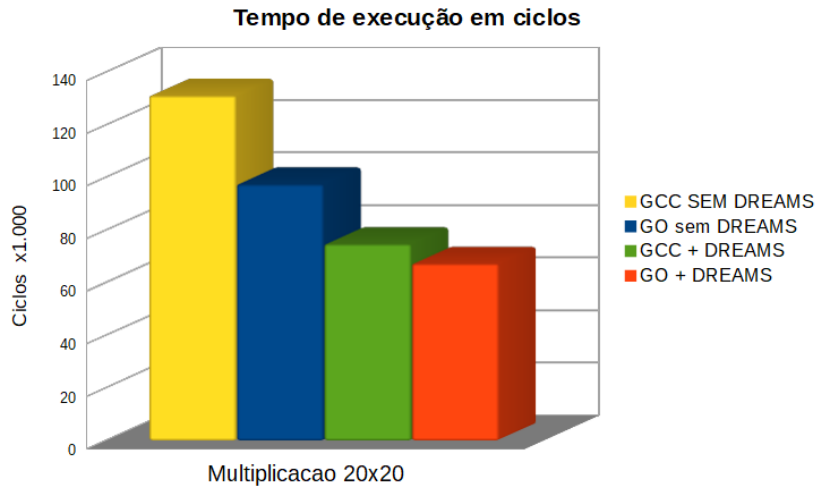


Figura 28 – Tempo de execução da multiplicação de matriz.

5.1.2 *Bitcount*

A aplicação *bitcount* na versão do ParMiBench contabiliza os bits de 75.000 inteiros. Essa versão não foi possível executar por limitações de memória da máquina que estava sendo utilizada para fazer as simulações. Então, reduziu-se o tamanho da entrada para contabilizar os bits de 18.750 inteiros. Nessa versão, a execução exigiu 10GB de memória RAM.

A implementação em C do *bitcount* foi acelerada em 41,7% pela arquitetura DREAMS. Duas configurações foram geradas pelo tradutor binário para essa aplicação. As configurações geradas possuem tamanho médio de 16 instruções sendo 6 delas acessos à memória. O ILP médio das configurações geradas foi 1,49. O máximo de instruções executadas em paralelo para essa aplicação foi 2. Portanto, para essa aplicação, a coluna dedicada a cada núcleo de processamento foi o suficiente para execução da aplicação. A taxa média de erro de predição da configuração ficou em 11,1%.

Como já explicado anteriormente, o tradutor binário salvará somente as configurações de computação intensiva do *loop* mais interno da aplicação ou que possuem 20 instruções ou mais. Uma das configurações do *bitcount* implementado em C possui exatamente 20 instruções e cai no segundo caso. Então, para avaliar esse limiar que foi proposto de 20 instruções, gerou-se duas versões do tradutor binário: uma que gerava somente configurações de computação intensiva do *loop* mais interno e outra que gerava essa configuração com 20 instruções. Essa última versão do tradutor binário acelerou em aproximadamente 3% a aplicação quando comparado com a versão que gerava somente configuração de computação intensiva de *loop* mais interno. Como essa configuração possui um

ILP médio baixo, apenas 1,49, e conseguiu mesmo assim prover aceleração, determinou-se que 20 instruções seria a quantidade mínima para gerar configurações.

A implementação em GO do *bitcount* foi acelerada em 26,87% pela arquitetura DREAMS. Somente a configuração do *loop* mais interno foi gerada. A configuração gerada possui 16 instruções sendo uma delas acesso à memória. O ILP médio da configuração foi de 1,2. O máximo de instruções executadas em paralelo na arquitetura reconfigurável por essa configuração foi 2. A taxa média de erro de predição da configuração ficou em 10,6%.

A arquitetura DREAMS acelerou menos quando utilizou código gerado pelo compilador GO. Nessa aplicação não houve diferença significativa no tamanho das configurações e essa aceleração maior no código gerado pelo GCC foi devido uma porcentagem maior da aplicação ter sido executada na arquitetura reconfigurável. O *bitcount* implementado em C teve 91,3% da aplicação executada na arquitetura reconfigurável e a implementação em GO teve 72,09%. O código gerado pelo GCC utilizou mais a arquitetura reconfigurável, pois o tradutor binário gerou duas configurações, enquanto a versão implementada em GO gerou somente uma configuração. Além disso, o ILP médio das configurações geradas pela versão em C foi superior a versão em GO. Nas versões em C e em GO o ILP médio ficou, respectivamente, em 1,49 e 1,2. Em ambas versões, as configuração geradas foram pequenas e com pouco ILP. Esse padrão é esperado em aplicações com comportamento *control-flow*. Nenhuma das duas versões fizeram uso do escalonador de configuração, já que em ambas o ILP máximo da configuração foi 2. Isso mostra que a arquitetura DREAMS, apesar de conseguir acelerar aplicações com esse comportamento, não é a mais apropriada, pois gera *overhead* desnecessário no seu escalonador de configuração. O código gerado pelo compilador GO é mais eficiente e executa 21% mais rápido que que o código gerado em C, considerado que ambos executam sem a arquitetura DREAMS. Com o uso da arquitetura DREAMS, o código gerado pelo compilador GO foi, aproximadamente, 2% mais rápido que o código gerado pelo compilador GCC. Os tempos de execução do *bitcount*, em ciclos, nas versões GO e C podem ser vistos na Figura 29.

5.1.3 Laplaciano

Na aplicação laplaciano, o filtro é aplicado em uma imagem de resolução SQCIF ($128x96$ pixels) em nível de cinza e quantização de 8 bits. A janela utilizada no filtro foi de $3x3$.

Na implementação em C, a inserção do DREAMS no processador *multicore* proveu uma aceleração de 28,59%. Somente uma configuração foi gerada e corresponde ao trecho que calcula o valor do pixel central do filtro. Essa configuração possui 73 instruções. Apesar da configuração possuir muitas instruções, o ILP médio foi de apenas 1,58. Isso acontece devido ao mau uso dos registradores pelo *cross compiler* GCC que gera muitas dependências verdadeiras entre as instruções, o que impossibilita a execução em paralelo

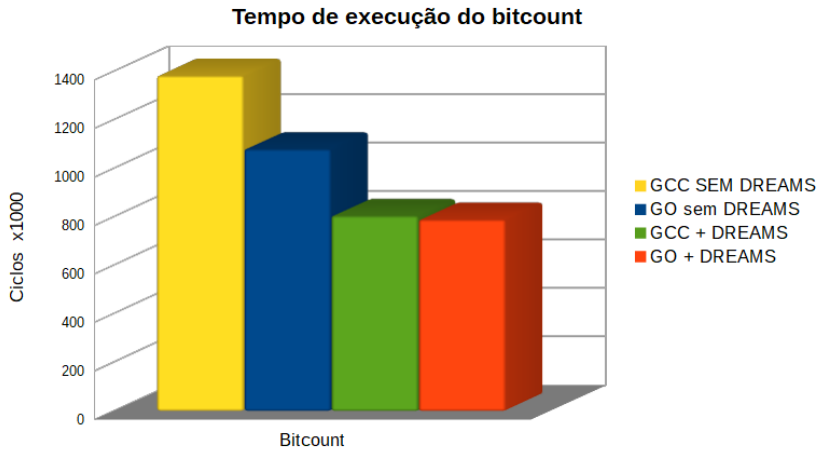


Figura 29 – Tempo de execução do *bitcount*.

dessas instruções. Para se ter uma noção, nessa configuração, o *cross compiler* utiliza somente 4 registradores. Das 73 instruções que a configuração possuía, 18 eram de acessos à memória.

A implementação em GO do laplaciano foi acelerada em 28,2% pela arquitetura DREAMS. Nessa versão, também teve somente uma configuração gerada correspondendo ao mesmo trecho que foi gerado para a versão em C. A configuração, porém, foi menor e possuía 41 instruções. O ILP médio da configuração foi de 3,1, o que mostra o potencial de exploração de ILP pelo compilador em GO, que usa de forma mais eficiente seus registradores. Das 41 instruções da configuração, somente 5 eram de acesso à memória.

Como pode-se observar, a aceleração obtida nessa aplicação, para ambas versões, é inferior à apresentada nas outras aplicações, que ficam em torno de 40% a aceleração fornecida para as versões em C. Essa redução na aceleração é devido à configuração gerada não ser uma configuração *loop*. Isso ocorre por conta do *if* que tem no final de cada iteração do laço. Como a arquitetura aqui proposta suporta somente 1 nível de especulação, ela irá especular o salto do laço de repetição, mas quando chegar no *if* a configuração encerrará. Sendo assim, a cada execução do laço, o início do laço será executado na arquitetura reconfigurável e o final do laço, o *if*, no processador. Perceba que isso faz com que haja muito troca de contexto e o custo de comunicação com a arquitetura reconfigurável ao final da execução seja muito maior. Uma consequência direta disso é a porcentagem de uso da arquitetura reconfigurável. Enquanto que nas outras aplicações o trecho executado na arquitetura reconfigurável estava em torno de 80% do tempo total da execução, nessa aplicação a execução na arquitetura correspondeu a 62% para ambas versões.

Para essa aplicação, que possui alguns blocos básicos grandes, o compilador GO foi capaz de explorar mais ILP, mesmo possuindo blocos básicos menores que o GCC. Isso pode ser observado pelo tamanho das configurações geradas e pelo ILP médio que as configurações obtiveram. Como já falado anteriormente, o uso dos registradores pelo

GCC é um fator limitante para exploração de ILP, mesmo quando se tem blocos básicos grandes, pois ele opera sobre um número muito limitado de registradores, o que gera muitas dependências verdadeiras de dado entre as instruções. Por outro lado, o compilador GO, apesar de gerar código com blocos básicos menores, consegue explorar mais ILP, pois faz melhor uso dos registradores MIPS. Para se ter uma ideia, no trecho acelerado dessa aplicação, o compilador GO faz uso de 14 registradores, por outro lado, o GCC utilizou somente 4. Isso mostra que, para aplicações que tenha trechos com muita computação e pouco controle, o compilador GO consegue gerar código que explora melhor os recursos da arquitetura DREAMS, pois configurações melhores serão geradas. O tempo de execução, em ciclos, do laplaciano pode ser visto na Figura 30.

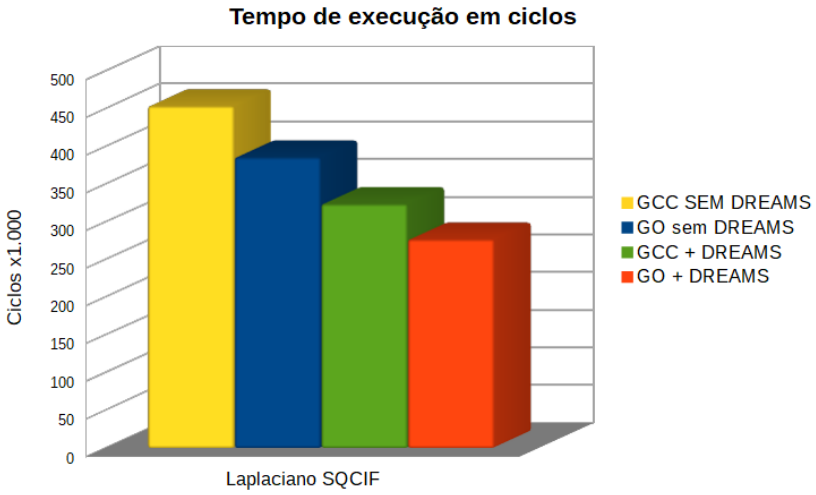


Figura 30 – Tempo de execução do laplaciano.

5.1.4 Decomposição LU

A aplicação LU foi selecionado do benchmark ParMiBench e é utilizada em aplicações de álgebra linear. Essa aplicação possui como característica ter trechos de computação intensiva e alguns trechos *control-flow*.

Na implementação em linguagem C, a arquitetura DREAMS conseguiu prover uma aceleração de 44,5%. Essa aplicação teve 5 configurações geradas pelo tradutor binário. As configurações possuíam, em média, 33 instruções. O ILP médio das configurações foi de 1,6. E a execução na arquitetura reconfigurável correspondeu a 89,78% do tempo total da execução.

Pela primeira vez uma aplicação que foi compilada utilizando o GCC conseguiu fazer uso do escalonador de configurações, pois seu ILP máximo foi 4. Até então, as aplicações implementada em C tinham alcançado no máximo ILP igual a 3. Essa aplicação também foi a primeira em que ocorreu o caso de uma configuração ser invalidada na cache de configuração por errar duas vezes consecutivas a predição do salto. Então, avaliou-se

o impacto dessa invalidação, pois isso pode prejudicar muito a aceleração obtida pelo *array* reconfigurável. Esse prejuízo ocorre porque quando uma configuração é invalidada, todo o processo de gerar a configuração será feito novamente e gerar configuração implica estar executando código no processador. Dependendo do padrão dos saltos da aplicação, a configuração poderá ser excluída várias vezes. Para as aplicações multiplicação de matrizes, laplaciano e *bitcount* não ocorreu nenhuma mudança na configuração. Ou seja, a configuração nunca errava duas vezes consecutivas sua predição. Isso ocorreu devido aos trechos mapeados para serem executados na arquitetura reconfigurável conterem salto condicional onde a condição tem um comportamento muito regular. Esse comportamento é visto em um *for* de $i = 0$ até n, por exemplo. Para esse tipo de comportamento, o mecanismo implementado, inspirado no preditor de salto de 2 bits, funciona muito bem e raramente irá errar duas vezes consecutivas. Já na aplicação LU, em uma das configurações geradas, o salto não possuía um comportamento regular, o que fez essa configuração ser invalidada, em média, 10 vezes, considerando-se os 4 núcleos. Esse número é pequeno e afetou pouco o desempenho, pois essa configuração foi executada 4.121 vezes e teve uma taxa de erro de predição de apenas 4,6%.

A implementação em GO foi acelerada em 30,21% pela arquitetura DREAMS. Foram geradas 4 configurações pelo tradutor binário. A configuração, em média, possui tamanho de 22 instruções sendo duas delas acessos à memória. O ILP médio das configurações ficou em 2,31. A execução na arquitetura reconfigurável correspondeu à 80% do tempo total de execução da aplicação.

Assim como nas outras aplicações, a arquitetura DREAMS acelerou menos o código gerado pelo compilador GO. Apesar do código GO gerar um código que permite maior exploração de ILP, os blocos básicos menores e a quantidade inferior de configurações geradas fez com que o código gerado pelo GCC fosse mais acelerado que código em GO. Para essa aplicação, ambos os códigos gerados conseguiram utilizar do escalonador de configuração. Um comparação dos tempos de execução podem ser visto na Figura 31.

Os resultados das quatro aplicações podem ser vistos na Tabela 3.

5.2 Avaliação do Simulador

Para avaliar a qualidade do simulador desenvolvido em *SystemC* (ACCELLERA, 2018), foi medida a vazão de instruções atingida pelo simulador. O simulador da arquitetura DREAMS foi todo desenvolvido em linguagem C++ utilizando a biblioteca *systemC*. Foram utilizados dois níveis de descrição: RTL (*Register Transfer Level*) e TLM *Transaction-Level Modeling* no desenvolvimento do simulador da arquitetura DREAMS. Alguns componentes da arquitetura foram implementados em RTL, tais como: mips pipeline, escalonador de configuração e controlador, outros em TLM, como: *array* reconfigurável, tradutor binário,

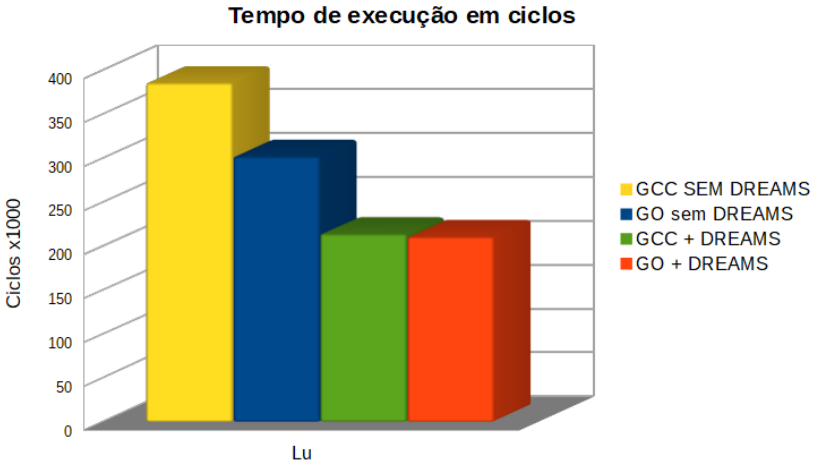


Figura 31 – Tempo de execução do LU.

Tabela 3 – Resultados de desempenho.

Aplicação	Compilador	Com DREAMS	Ciclos	% execução na arquitetura reconfigurável	ILP médio	ILP máximo	Erro de predição	Aceleração
Multiplicação de Matrizes	Go	N	97.575	-	-	-	-	-
		S	67.334	82,2%	2,07	5	4,7%	30,99%
	GCC	N	131.217	-	-	-	-	-
		S	74.793	95,3%	1,5	3	3,00%	43,00%
Laplaciano	Go	N	386.317	-	-	-	-	-
		S	277.375	62,50%	3,1	5	0,7%	28,2%
	GCC	N	454.099	-	-	-	-	-
		S	324.272	62,60%	1,58	3	0,7%	28,59%
Decomposição Lu	Go	N	302.040	-	-	-	-	-
		S	210.794	85,4%	2,31	5	5,02%	30,21%
	GCC	N	385.625	-	-	-	-	-
		S	213.998	89,78%	1,6	4	5,16%	44,50%
Bitcount	GO	N	1.080.055	-	-	-	-	-
		S	789.809	72,09%	1,2	2	10,60%	26,87%
	Gcc	N	1.382.922	-	-	-	-	-
		S	805.343	91,30%	1,49	2	5,56%	41,70%

entre outros. O nível RTL descreve o sistema no nível de transferência de dados entre registradores e é mais lento, porém fornece descrição mais detalhada e a nível de portas. Esse tipo de descrição é o utilizado pelas HDL (*Hardware Description Language*) como VHDL e Verilog. Por outro lado, a descrição a nível TLM é mais rápida e descreve o sistema como transações que são realizadas através de chamadas de funções. Apesar de prover uma descrição mais alto nível do sistema, focando mais na funcionalidade do sistema, este

modelo de descrição consegue ter modelos precisos à nível de ciclos.

As aplicações foram executadas em um computador com a seguinte configuração: processador Intel Xeon com 8 núcleos, 16GB de memória e sistema operacional Ubuntu 14.04 LTS. Foi utilizada a versão 2.3.1 do *systemC*. A vazão, em instruções por segundo, fornecida pelo simulador pode ser vista na Figura 32.

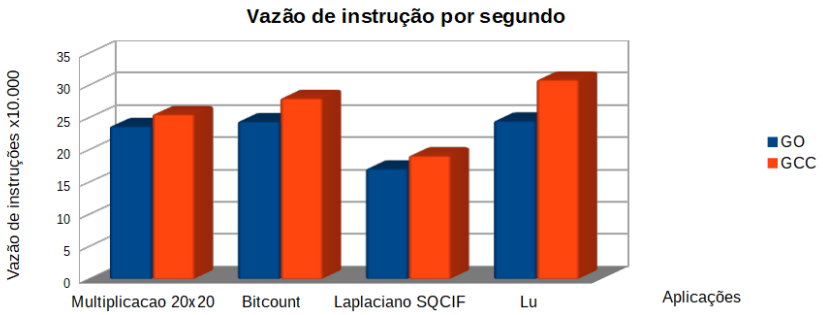


Figura 32 – Vazão de instruções do simulador

Em média, o simulador possui uma vazão de aproximadamente 23.000 instruções por segundo. A aplicação que teve menor vazão foi o laplaciano, pois foi a aplicação que foi menos acelerada pela arquitetura DREAMS. Por outro lado, a aplicação LU teve a maior vazão, pois foi a mais acelerada pela a arquitetura DREAMS. Então, quanto mais aceleração a arquitetura DREAMS prover para uma aplicação, maior será a vazão do simulador.

5.3 Considerações Finais

As quatros aplicações implementadas em C e compiladas utilizando o *cross compiler* GNU GCC tiveram mais aceleração que sua contraparte em GO. Contudo, o código gerado pelo compilador Go é melhor em vários aspectos: é mais rápido, possui menos instruções, utiliza mais registradores, tem menos instruções de acesso à memória e faz mais uso do escalonador de configurações. A única aplicação em C que conseguiu utilizar do escalonador de configuração foi a LU. Para utilizar do escalonador de configuração, a aplicação precisa ter um ILP considerável e executar mais que 3 instruções em paralelo na arquitetura reconfiguração. Contudo, o código gerado pelo GCC utiliza poucos registradores, limitando a exploração de ILP, pois gera-se muita dependência verdadeira entre as instruções.

Observou-se também que, para a aplicação laplaciano, que possuiu um bloco básico grande, o compilador Go conseguiu explorar consideravelmente mais ILP que o GCC e proveu quase a mesma aceleração. Os melhores tempo de execução obtidos foram do código gerado pelo compilador GO.

O compilador Go se mostrou mais adequado para ser utilizado pela arquitetura DREAMS, pois conseguiu utilizar o escalonador de configurações na maioria das aplicações,

com exceção do *bitcount*. Contudo, para poder se beneficiar do fato do compilador utilizar mais registradores e permitir explorar mais ILP, a aplicação deve ter blocos básicos grandes e com muita computação, como no laplaciano, por exemplo. Desse modo, com blocos básicos maiores, permite-se que o tradutor binário explore melhor o ILP e consiga mais aceleração. Para aplicações *control-flow*, os blocos básicos utilizando o compilador GO são menores que os blocos gerados pelo GCC, e como o espaço para exploração de ILP é menor, consegue-se menos aceleração utilizando a arquitetura DREAMS.

Para as aplicações utilizadas no *benchmark* as predições das configurações geradas foram boas. Isso ocorreu por conta que os trechos selecionados para executarem na arquitetura reconfigurável possuem um comportamento muito regular. A taxa de erro de predição das configurações ficou em torno de 7% e não teve muito impacto no desempenho da arquitetura.

Na aplicação *control-flow*,*bitcount*, a arquitetura DREAMS conseguiu acelerar,porém não utilizou-se do escalonar de configuração. Portanto, para esse tipo de aplicação, a arquitetura consegue acelerar, mas não é a mais apropriada, pois gerar um *overhead* desnecessário no seu escalonador de configurações que não será utilizado em aplicações com esse tipo de comportamento.

6 Conclusão e Trabalhos Futuros

Este trabalho apresentou uma arquitetura reconfigurável de granularidade grossa para processador *multicore* denominada DREAMS. O objetivo é acelerar múltiplas *threads* (ou processos) executando simultaneamente em diferentes núcleos de um processador *multicore*. A arquitetura utilizou uma adaptação do mecanismo de tradução binária proposto em (BECK, 2009) para geração de configuração e reconfiguração da arquitetura em tempo de execução. A arquitetura reconfigurável apresentada possui um *array* reconfigurável que está organizado em quatro colunas reconfiguráveis. Os recursos do *array* reconfigurável são compartilhado entre os núcleos de processamento. O compartilhamento é feito dinamicamente pelo escalonador de configurações que, em tempo de execução, verifica a necessidade de computação de cada *thread* em execução nos núcleos de processamento e aloca recursos do *array* reconfigurável para a execução. O processo de alocação obedece a dois princípios. Inicialmente obedece a uma política de prioridade, que visa garantir que cada núcleo disponha de uma quantidade mínima de recursos computacionais. O segundo princípio visa oferecer recursos adicionais aos núcleos com necessidades computacionais superiores àquela definida como mínima. A arquitetura proposta utiliza consideravelmente menos recursos que as arquiteturas de grão grosso para processadores *multicores* existentes na literatura. Enquanto que o CReAMS (RUTZIG; BECK; CARRO, 2013) utiliza 1.152 e o HARTMP (SOUZA et al., 2016), na sua menor versão, utiliza 144 unidades funcionais reconfiguráveis para um processador *multicore* de 4 núcleos, a versão proposta neste trabalho utiliza somente 16.

A arquitetura DREAMS foi implementada utilizando a linguagem de programação C++ e utilizando a biblioteca *systemC*. Foi utilizado dois níveis de descrição na implementação: RTL e TLM.

O tradutor binário foi modificado para gerar configurações para arquitetura proposta. Para aumentar o ILP explorado pelo tradutor binário, uma técnica de renomeação de registradores foi adicionada ao tradutor binário. A maneira como gerar configuração unindo dois blocos básico, através de especulação, também foi modificada, pois era muito custoso recuperar restaurar o estado de uma configuração já gerada no tradutor binário. A proposta deste trabalho gera sempre a configuração referente ao primeiro caminho tomado pelo salto. Contudo, existe um mecanismo que verifica se essa predição foi boa. O mecanismo implementado foi inspirado no preditor de salto de 2 bits. Então, quando uma configuração erra duas vezes consecutivas sua predição, a configuração é invalidade e o tradutor binário irá gerá-la novamente.

Um estudo do ILP que o tradutor consegue explorar com o código utilizado pelo

cross compiler GNU GCC foi realizado. Detectou-se que, havia a necessidade de um compilador que utilizasse mais eficientemente seus registradores. Por este motivo, utilizou-se de um compilador resultante da utilização do *framework COGNITE*, desenvolvido no laboratório CESLa (*Circuits and Embedded System Lab*) (CARVALHO, 2018) que visava gerar código que permitisse maior exploração de ILP.

Para avaliar o desempenho, foram utilizadas 4 aplicações: *bitcount*, LU, laplaciano e multiplicação de matrizes. Foram geradas versões em C e em GO das aplicações para verificar como a arquitetura reconfigurável se comporta nos códigos gerado pelos dois compiladores. A análise de desempenho foi realizada comparando uma versão do processador *multicore* com a DREAMS e uma versão sem a DREAMS para verificar a aceleração que a inserção da arquitetura provia no sistema. O código gerado pelo compilador *cross compiler* GNU GCC teve mais aceleração que o código gerado pelo compilador em GO. Contudo, o compilador em GO gerou código em que as configurações gerados pelo tradutor binário possuíam mais ILP. De um modo geral, com exceção da aplicação LU, as aplicações compiladas utilizando o *cross compiler* não conseguiu utilizar o escalonador de configuração, pois não havia ILP suficiente devido ao mau uso dos registradores por esse compilador. Por outro lado, o compilador em GO conseguiu utilizar o escalonador de configuração em quase todas as aplicações, com exceção do *bitcount*. Utilizando o *cross compiler* GNU GCC e o compilador GO, em média, a arquitetura DREAMS proveu uma aceleração de aproximadamente 39% e 28%, respectivamente.

Para avaliar a qualidade do simulador desenvolvido, foi contabilizado o tempo das simulações e a quantidade de instruções executadas no simulador para medir a vazão que o simulador tem. Em média, o simulador possui uma vazão de aproximadamente 23.000 instruções por segundo.

Desejou-se ter uma comparação com as arquitetura CReAMS (RUTZIG; BECK; CARRO, 2013) e HARTMP (SOUZA et al., 2016), pois ambas utilizam o tradutor binário e são proposta de CGRA para *multicore*. Contudo, não foi possível, pois as aplicações que foram utilizadas nesses trabalhos possuem operações de ponto flutuante e a versão atual da arquitetura aqui proposta não possui uma unidade de ponto flutuante. Alem disto, os dados disponibilizados tanto nos artigos quanto em teses e dissertação disponíveis sobre esta arquitetura, não são suficientes para se ter certeza que o mesmo experimento (aplicações em execução nas mesmas condições) foi realizado. Como trabalho futuro, pretende-se implementar uma unidade de ponto flutuante no núcleo de processamento para que essa comparação seja possível de ser realizada. Adicionalmente, uma implementação das aplicações que foram utilizadas como *benchmark* dessas arquiteturas também deve ser feita.

Também pretende-se, como trabalho futuro, fazer uma análise de potência e área da arquitetura DREAMS, já que isso, potencialmente, é uma grande contribuição deste

trabalho. Para que isso seja possível, uma versão em VHDL ou Verilog da arquitetura implementada em C++ com *systemC* será desenvolvida.

Referências

ABNOUS, A. et al. Design and implementation of the 'tiny risc' microprocessor. *Microprocessors and Microsystems - Embedded Hardware Design*, v. 16, n. 4, p. 187–193, 1992. Citado na página 13.

ACCELLERA. *SystemC Library*. 2018. <http://www.accellera.org/downloads/standards/systemc>. Acessado em 30/07/2018. Citado na página 51.

AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. New York, NY, USA: ACM, 1967. (AFIPS '67 (Spring)), p. 483–485. Citado na página 1.

AZARIAN, A.; AHMADI, M. Reconfigurable computing architecture survey and introduction. In: *2009 2nd IEEE International Conference on Computer Science and Information Technology*. [S.l.: s.n.], 2009. p. 269–274. Citado na página 5.

AZARIAN, A.; AHMADI, M. Reconfigurable computing architecture survey and introduction. In: *2009 2nd IEEE International Conference on Computer Science and Information Technology*. [S.l.: s.n.], 2009. p. 269–274. Citado na página 8.

BECK, A. C. S. *Transparent Reconfigurable Architecture for Heterogeneous Applications*. Tese (Doutorado) — Universidade Federal do Rio Grande do Sul, Porto Alegre, 2009. Citado 12 vezes nas páginas 15, 3, 6, 25, 26, 28, 30, 31, 32, 33, 34 e 55.

BECK, A. C. S. et al. Transparent reconfigurable acceleration for heterogeneous embedded applications. In: *2008 Design, Automation and Test in Europe*. [S.l.: s.n.], 2008. p. 1208–1213. ISSN 1530-1591. Citado 6 vezes nas páginas 15, 2, 16, 20, 24 e 27.

CARVALHO, E. S. *COGNITE - um Framework Gerador de Código para Arquiteturas Multicore*. Dissertação (Mestrado) — Universidade Federal do Piauí, Teresina, 2018. Citado 2 vezes nas páginas 45 e 56.

CHIU, J. C.; CHOU, Y. L.; CHEN, P. K. Hyperscalar: A novel dynamically reconfigurable multi-core architecture. In: *2010 39th International Conference on Parallel Processing*. [S.l.: s.n.], 2010. p. 277–286. ISSN 0190-3918. Citado 2 vezes nas páginas 1 e 17.

DORTA, A. J.; RODRIGUEZ, C.; SANDE, F. de. 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 45.

FENG, C.; YANG, L. Design and evaluation of a novel reconfigurable alu based on fpga. In: *Proceedings 2013 International Conference on Mechatronic Sciences, Electric Engineering and Computer (MEC)*. [S.l.: s.n.], 2013. p. 2286–2290. Citado 2 vezes nas páginas 15 e 24.

GOKHALE, M. et al. Building and using a highly parallel programmable logic array. *Computer*, v. 24, n. 1, p. 81–89, Jan 1991. ISSN 0018-9162. Citado na página 11.

GOLDSTEIN, S. C. et al. Piperench: a coprocessor for streaming multimedia acceleration. In: *Proceedings of the 26th International Symposium on Computer Architecture (Cat. No.99CB36367)*. [S.l.: s.n.], 1999. p. 28–39. ISSN 1063-6897. Citado 4 vezes nas páginas 15, 12, 13 e 24.

GOTTLIEB, D. B. et al. Clustered programmable-reconfigurable processors. In: *2002 IEEE International Conference on Field-Programmable Technology, 2002. (FPT). Proceedings.* [S.l.: s.n.], 2002. p. 134–141. Citado 5 vezes nas páginas 15, 2, 17, 18 e 24.

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 5.

HAUCK, A. D. S. *Reconfigurable computing : the theory and practice of FPGA-based computation*. [S.l.]: Morgan-Kaufmann, 2004. (The Morgan Kaufmann series in systems on silicon). ISBN 0123705223,978-0-12-370522-8. Citado na página 7.

HAUCK, S. et al. The chimaera reconfigurable functional unit. In: *Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No.97TB100186)*. [S.l.: s.n.], 1997. p. 87–96. Citado 4 vezes nas páginas 6, 7, 11 e 24.

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 6.

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 45.

KARUNARATNE, M. et al. Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect. In: *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. [S.l.: s.n.], 2017. p. 1–6. Citado 2 vezes nas páginas 15 e 10.

LI, Y.; PEDRAM, A. Caterpillar: Coarse grain reconfigurable architecture for accelerating the training of deep neural networks. In: *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. [S.l.: s.n.], 2017. p. 1–10. Citado na página 16.

LIANG, S. et al. A coarse-grained reconfigurable architecture for compute-intensive mapreduce acceleration. *IEEE Computer Architecture Letters*, v. 15, n. 2, p. 69–72, July 2016. ISSN 1556-6056. Citado 2 vezes nas páginas 15 e 10.

LYSECKY, R.; STITT, G.; VAHID, F. Warp processors. In: *Proceedings of the 41st Annual Design Automation Conference*. New York, NY, USA: ACM, 2004. (DAC '04), p. 659–681. ISBN 1-58113-828-8. Disponível em: <http://doi.acm.org/10.1145/996566.1142986>. Citado 4 vezes nas páginas 9, 14, 19 e 24.

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 8.

MIYAMORI, T.; OLUKOTUN, K. Remarc: Reconfigurable multimedia array coprocessor. In: *IEICE Transactions on Information and Systems E82-D*. [S.l.: s.n.], 1998. p. 389–397. Nenhuma citação no texto.

MOORE, G. E. Cramming more components onto integrated circuits. *Electronics*, v. 38, n. 5, p. 114, Abril 1965. Citado na página 1.

PAL, R. K.; PAUL, K.; PRASAD, S. Rekonf: Dynamically reconfigurable multicore architecture. In: *Journal of Parallel and Distributed Computing*. [S.l.: s.n.], 2014. p. 3071–3086. Citado na página 1.

PATEL, K.; BLEAKLEY, C. J. Coarse grained reconfigurable array based architecture for low power real-time seizure detection. *Journal of Signal Processing Systems*, v. 82, n. 1, p. 55–68, Jan 2016. ISSN 1939-8115. Disponível em: <https://doi.org/10.1007/s11265-015-0981-9>. Citado na página 16.

PATTERSON, D. A.; HENESSY, J. L. *Computer organization and design the hardware/software interface*. 5th. ed. Oxford, USA: Morgan Kaufmann, 2014. Citado 4 vezes nas páginas 26, 28, 31 e 32.

RUTZIG, M. B.; BECK, A. C. S.; CARRO, L. A transparent and energy aware reconfigurable multiprocessor platform for simultaneous ilp and tlp exploitation. In: *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*. [S.l.: s.n.], 2013. p. 1559–1564. ISSN 1530-1591. Citado 7 vezes nas páginas 2, 17, 20, 23, 24, 55 e 56.

SANKARALINGAM, K. et al. Trips: A polymorphous architecture for exploiting ilp, tlp, and dlp. *ACM Trans. Archit. Code Optim.*, ACM, New York, NY, USA, v. 1, n. 1, p. 62–93, mar. 2004. ISSN 1544-3566. Disponível em: <http://doi.acm.org/10.1145/980152.980156>. Citado na página 7.

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 5 vezes nas páginas 15, 5, 13, 14 e 24.

SOUZA, J. D. et al. A reconfigurable heterogeneous multicore with a homogeneous isa. In: *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*. [S.l.: s.n.], 2016. p. 1598–1603. Citado 6 vezes nas páginas 2, 17, 22, 24, 55 e 56.

STITT, G.; VAHID, F. Thread warping: Dynamic and transparent synthesis of thread accelerators. *ACM Trans. Des. Autom. Electron. Syst.*, v. 16, n. 3, p. 32:1–32:21, jun 2011. ISSN 1084-4309. Disponível em: <http://doi.acm.org/10.1145/1970353.1970365>. Citado 4 vezes nas páginas 15, 19, 21 e 24.

TEHRE, V.; KSHIRSAGAR, R. Survey on coarse grained reconfigurable architectures. v. 48, p. 1–7, 06 2012. Citado na página 5.

TESSIER, R.; POCEK, K.; DEHON, A. Reconfigurable computing architectures. *Proceedings of the IEEE*, v. 103, n. 3, p. 332–354, March 2015. ISSN 0018-9219. Citado na página 11.

"THEODORIDIS, G.; SOUDRIS, D.; VASSILIADIS, S. "a survey of coarse-grain reconfigurable architectures and cad tools". In: _____. *"Fine- and Coarse-Grain Reconfigurable Computing"*. "Dordrecht": "Springer Netherlands", "2007". p. "89–149". ISBN "978-1-4020-6505-7". Citado 2 vezes nas páginas 19 e 42.

TOMASULO, R. M. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, v. 11, n. 1, p. 25–33, Jan 1967. ISSN 0018-8646. Citado na página 31.

WANG, C.; CAO, P.; YANG, J. Efficient aes cipher on coarse-grained reconfigurable architecture. *IEICE Electronics Express*, v. 14, n. 11, p. 20170449–20170449, 2017. Citado na página 16.

WATKINS, M. A.; ALBONESI, D. H. Remap: A reconfigurable heterogeneous multicore architecture. In: *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. [S.l.: s.n.], 2010. p. 497–508. ISSN 1072-4451. Citado 7 vezes nas páginas 15, 2, 17, 19, 20, 24 e 42.

WIJTVLIET, M.; WAEIJEN, L.; CORPORAAL, H. Coarse grained reconfigurable architectures in the past 25 years: Overview and classification. In: *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. [S.l.: s.n.], 2016. p. 235–244. Citado 3 vezes nas páginas 5, 6 e 9.

XILINX. *Virtex-5 FPGA configuration user guide*. 2018. <https://www.xilinx.com/support/documentation/user_guides/ug190.pdf>. Acessado em 10/07/2018. Citado na página 21.

XU, J. et al. Coarse-grained architecture for fingerprint matching. *ACM Trans. Reconfigurable Technol. Syst.*, ACM, New York, NY, USA, v. 9, n. 2, p. 12:1–12:15, dez. 2015. ISSN 1936-7406. Disponível em: <http://doi-acm-org.ez54.periodicos.capes.gov.br/10.1145/2791296>. Citado na página 16.

YAN, L. et al. A reconfigurable processor architecture combining multi-core and reconfigurable processing units. *Telecommunication Systems*, v. 55, n. 3, p. 333–344, Mar 2014. ISSN 1572-9451. Disponível em: <https://doi.org/10.1007/s11235-013-9791-1>. Citado 6 vezes nas páginas 15, 17, 21, 22, 24 e 42.

YUAN, Z. et al. Coral: Coarse-grained reconfigurable architecture for convolutional neural networks. In: *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. [S.l.: s.n.], 2017. p. 1–6. Citado na página 16.

ZHAO, B.; WANG, M.; LIU, M. An energy-efficient coarse grained spatial architecture for convolutional neural networks alexnet. *IEICE Electronics Express*, v. 14, n. 15, p. 20170595–20170595, 2017. Citado na página 16.