TRANSACTIONAL WAVECACHE - EXECUÇÃO ESPECULATIVA FORA-DE-ORDEM DE OPERAÇÕES DE MEMÓRIA EM UMA MÁQUINA DATAFLOW

Leandro Augusto Justen Marzulo

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof. Felipe Maia Galvão França, Ph.D.

Prof. Claudio Luis de Amorim, Ph.D.

Prof. Maria Clicia Stelling de Castro, D.Sc.

RIO DE JANEIRO, RJ - BRASIL
DEZEMBRO DE 2007

MARZULO, LEANDRO AUGUSTO JUSTEN

Transactional WaveCache - Execução Especulativa Fora-de-Ordem de Operações de Memória em uma Máquina Dataflow [Rio de Janeiro] 2007

XV, 125 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistema e Computação, 2007)

Dissertação - Universidade Federal do Rio de Janeiro, COPPE

1. Dataflow
2. Memória Transacional
3. Execução Fora-de-Ordem
4. WaveCache
5. WaveScalar

I. COPPE/UFRJ II. Título (série)

"Tudo deveria ser feito da forma mais simples possível,
mas não mais simples do que isto."
Albert Einstein

Agradecimentos

Aos membros da banca examinadora, pela generosidade ao disponibilizar tempo para apreciação deste trabalho.

Aos meus orientadores, Felipe França e Vítor Santos Costa, pelo acompanhamento deste trabalho e por estarem sempre disponíveis para tirar dúvidas, mesmo pela Internet. Obrigado pela amizade e companheirismo.

À Ivomar, pela leitura crítica de uma versão preliminar deste texto.

Ao Programa de Engenharia de Sistemas e Computação (PESC/COPPE/UFRJ) por todo o suporte fornecido que me permitiu desenvolver essa dissertação de mestrado. Em especial gostaria de agradecer a: Cláudia, Solange, Roberto Rodrigues, Mercedes, Itamar, Adilson e Lourdes.

À Universidade do Estado do Rio de Janeiro, pela base sólida de minha formação. Em especial a: Vera Werneck, Maria Clícia, Paulo Eustáquio, Rosa Costa, Galúcio e Alexandre Sztajnberg, professores que fizeram grande diferença em minha formação.

Aos meus professores do mestrado: Gerson Zaverucha (Teoria de Conjuntos e Lógica), Felipe França (Arquitetura de Computadores I e II), Ricardo Farias (Estrutura de Dados e Algoritmos), Paulo Augusto Veloso (Teoria da Computação), Claudio Amorim (Arquiteturas Avançadas), Valmir Barbosa (Algoritmos Distribuídos) e Vítor Santos Costa (Sistemas Operacionais), além do professor Edil Severino Tavares (Arquitetura de Computadores II), que me orientou por um período de aproximadamente 4 meses e me propiciou o contato com diversos assuntos relacionados com o tema deste trabalho.

Aos autores referenciados que, com seus trabalhos, possibilitaram a realização desta pesquisa.

À Capes, pela bolsa de mestrado fornecida, que permitiu custear minhas despesas durante dois anos de mestrado.

Aos vários amigos que fiz na UFRJ: Bruno, Ivomar, Talita, Alexandre, Bernardo, Danilo, Fabiano, André Nathan, Patrécia, André (do samba), Vivian, Cristiane, João Victor, Thiago Henrique, João Maurício, Luciana, Carlos Melo, Ramon, Priscila. Obrigado a todos por seu companheirismo e agradáveis momentos que me proporcionaram.

Aos meus pais, por me tornarem um homem de caráter e por me fazerem sempre valorizar a importância dos estudos. Ao meu irmão, por ser um eterno amigo e companheiro. Ao meu padrinho, pela amizade, carinho e conversas descontraídas. À Carol pelo companheirismo, carinho e compreensão nos momentos de ausência em que me dedicava a este trabalho. À todos os meus amigos e familiares que, junto com meu conhecimento, compõem o bem mais precioso que possuo.

À Deus, por sua sabedoria eterna, e por ter me dado inúmeras oportunidades e força para aproveitá-las. Obrigado por tudo.

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

TRANSACTIONAL WAVECACHE - EXECUÇÃO ESPECULATIVA FORA-DE-ORDEM DE OPERAÇÕES DE MEMÓRIA EM UMA MÁQUINA DATAFLOW

Leandro Augusto Justen Marzulo

Dezembro / 2007

Orientadores: Felipe Maia Galvão França
Vítor Manuel de Morais Santos Costa
Programa: Engenharia de Sistemas e Computação

Esta pesquisa realiza uma análise comportamental de um mecanismo de ordenação de operações de memória implementado em uma arquitetura *dataflow*. O mecanismo permite um relaxamento no modelo existente, com execução especulativa e fora-de-ordem de operações de memória de diferentes ondas, baseado em pricípios de Memória Transacional. O mecanismo de ordenação existente foi modificado, originando à *Transactional WaveCache*, cuja corretude e impacto de desempenho foram avaliados através de simulação dirigida à execução utilizando o simulador arquitetural do *WaveScalar*. Em resultados obtidos da execução de um conjunto de programas, alcançou-se acelerações entre 1,52, e 1,91, para aplicações com baixo grau de concorrência no uso da memória. Em aplicações com grau de concorrência alto, houve desaceleração de 1,05 a 1,16. Para aplicações com grau de concorrência intemediário, foram obtidas desacelerações de 1,01 a acelerações de 1,33. As acelerações são justificadas pela extração de concorrência no uso da memória, resultando em maior paralelismo no sistema de execução.

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

TRANSACTIONAL WAVECACHE - SPECULATIVE AND OUT-OF-ORDER MEMORY OPERATIONS EXECUTION IN A DATAFLOW MACHINE

Leandro Augusto Justen Marzulo

December / 2007

Advisors: Felipe Maia Galvão França
Vítor Manuel de Morais Santos Costa
Department: Computing and Systems Engineering

This research performs an analysis of the behavior of a memory operations ordering mechanism implemented in a dataflow architecture. Such mechanism introduces run-time memory disambiguation, concerning memory operations belonging to different waves, based on Transactional Memories concepts. In order to do so, the original memory ordering mechanism was modified, originating the Transactional WaveCache. The use of the WaveScalar architectural simulator was employed to evaluate the correctness and performance of the mechanism. According to the results of the execution of a set of artificial applications, speedups from 1,52, to 1,91 were reached for applications with low concurrency in terms of memory accesses, considering the ideal degree of speculation. For applications with high concurrency there were slowdowns from 1,05 to 1,16. Application with intermediary concurrency presented a slowdown of 1,01 for an application and speedups up to 1,33 for the others. This speedup is achieved through extraction of concurrency in the memory system, resulting in more paralelism in the execution system.

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Algoritmos

Capítulo 1

Introdução

A crescente demanda por processadores mais velozes e baratos tem levado cientistas e estudiosos do assunto a pesquisarem novos e mais ousados projetos de forma a atender essa necessidade. Grupos preocupados com o fim da Lei de Moore, têm colocado seus esforços na pesquisa de novos materiais para substituírem o silício, quando o mesmo não puder mais contribuir com a evolução dos processadores de forma satisfatória. Porém, existe uma preocupação mais imediata acerca do assunto: o "**muro da escalabilidade do processador**". A Lei de Moore ainda é válida e existe uma constante evolução na tecnologia de silício que permite que mais transistores sejam colocados em uma mesma área do *chip*. No entanto, estas melhorias tecnológicas não são convertidas em desempenho de forma proporcional. Equipes cada vez maiores são necessárias para criar e testar processadores mais rápidos e mais complexos para **tentar** aproveitar ao máximo os avanços na tecnologia de produção de circuitos integrados.

1.1 Motivação

Diversas alternativas para tornar projetos de processadores mais escaláveis já foram pesquisadas. Essas alternativas estendem o modelo de Von Neumann (execução guiada pelo *program-counter*) explorando paralelismo de granularidade grossa [61, 30], usando mecanismos de verificação redundante [6], *pipelining*, predição de

desvio [45], *renomeamento de registradores* [44] e Escalonamento dinâmico [60] ou provendo execução guiada por fluxo de dados de forma limitada explorando tecnologia de compiladores [40]. Ao tentar estender o modelo de Von Neumann para extrair paralelismo de um programa, o que se consegue é apenas resgatar parte desse paralelismo. Além disso, a complexidade e o custo de projeto dos microprocessadores que exploram agressivamente o Paralelismo ao Nível de Instrução cresceu. Segundo Olukotun [42], a complexidade da lógica adicional necessária para encontrar instruções paralelas dinamicamente é, de forma aproximada, proporcional ao quadrado do número de instruções que podem ser disparadas simultaneamente. Estas são motivações para a busca de um modelo de execução alternativo que seja intrinsicamente paralelo e distribuído. Um dos produtos desses esforços é o modelo *dataflow*, onde a execução das instruções se baseia no fluxo de dados, ou seja, nas dependências de dados verdadeiras. Algumas arquiteturas [11, 13, 16, 17, 27, 43, 51, 54, 56] surgiram baseadas nesse modelo.

Segundo Swanson em [58], garantir a semântica das operações de memória em linguagens imperativas é um dos principais desafios que tem impedido que as arquiteturas *dataflow* se tornem uma alternativa viável para o modelo de Von Neumann. Ao permitir que a execução do programa seja guiada apenas pelo fluxo de dados, sem nenhum mecanismo para identificar a seqüência das operações de memória, não há como garantir que as mesmas ocorram na ordem esperada pelo programa.

A arquitetura WaveScalar [56, 58, 59, 57] é a primeira arquitetura *dataflow* a garantir a ordenação das operações de memória, requerida pelas linguagens imperativas. Desta forma, o modelo pode ser adotado, sem a necessidade de reescrever os programas para adequá-los ao mesmo. O WaveScalar apresenta desempenho de 2 a 11 vezes superior a arquiteturas CMP (para o *benchmark* Splash2) e desempenho similar a um superescalar com execução fora de ordem, porém com 30% menos área de *chip* utilizada (para os *benchmarks* SpecInt, SpecFP e MediaBench) [59].

Como descrito no Capítulo 2, a arquitetura WaveScalar propõe um mecanismo para garantir a ordem de operações de memória chamado de *Wave-Ordered Memory*. No entanto, esse mecanismo é, sabidamente, um forte ponto de serialização

da arquitetura, sendo o principal objetivo desta pesquisa propor modificações no seu subsistema de memória visando um aumento de paralelismo e de concorrência do mesmo.

1.2 Trabalhos relacionados

O mecanismo de ordenação de operações de memória apresentado no Capítulo 3 é inspirado em princípios de Memória Transacional. Ele não tem como objetivo facilitar o uso de *locks* ou aumentar o paralelismo de *threads*. No entanto, o mecanismo permite a execução na memória de diferentes blocos de operações (transações) de forma especulativa e fora-de-ordem. As questões e desafios endereçados por grupos de pesquisa em Memória Transacional são semelhantes as encontradas neste trabalho. Portanto, uma apresentação destas questões é necessária para o melhor entendimento do mecanismo. Esta apresentação é feita nesta seção. Uma revisão bibliográfica mais completa sobre o assunto, pode ser encontrada em [48, 52], bem como nas outras referências citadas nesta seção.

O algoritmo de Tomasulo [60] é uma técnica de escalonamento dinâmico que permite que a execução de instruções prossiga na presença de dependências. Ele controla quando os operandos de instruções estão disponíveis, minimizando perigos *Read After Write RAW*), e introduz o renomeamento de registradores. A técnica é aplicada em máquina de Von Neumann, mas faz com que a execução de instruções em unidades de ponto flutuante seja feita seguindo o fluxo de dados. Como o mecanismo de ordenação de operações de memória apresentado neste trabalho é aplicado a uma máquina *dataflow*, é importante fazer uma breve revisão do algoritmo de Tomasulo para que se possa entender de que forma são tratadas as operações de memória. Conforme o algoritmo é explicado, também são apontadas as diferenças em relação a uma máquina *dataflow*. Uma explanação mais abrangente sobre o asunto pode ser encontrada em [60, 45].

1.2.1 Memórias Transacionais

O termo Memória Transacional foi definido, em 1993, por Herlihy e Moss [24] como "uma nova arquitetura para multiprocessadores que objetiva tornar a sincronização livre de bloqueios tão eficiente (e fácil de usar) quanto técnicas convencionais baseadas em exclusão mútua". A técnica se apresentou como uma alternativa aos bloqueios. Com o surgimento dos CMPs (*single-Chip MultiProcessors*), o assunto entrou em evidência, com a crescente necessidade da criação de modelos de programação paralela e técnicas que facilitassem o desenvolvimento de aplicações *multithread*.

Códigos protegidos com bloqueios não podem executar em paralelo e, em geral, possuem baixo desempenho. Além disso, os bloqueios podem causar problemas como *deadlock*, quando utilizados incorretamente. Com a Memória Transacional o programador não precisa utilizar bloqueios para proteger o acesso às variáveis compartilhadas. Ele apenas indica as porções de código que devem executar atomicamente. Cada região atômica é uma transação, que pode executar em paralelo com as demais. Caso uma violação seja verificada entre duas transações, uma delas deve ser cancelada (*rollback*) e as alterações feitas pela mesma devem ser descartadas. Se uma transação termina sem violações, ela é efetivada (*commit*) e as alterações feitas pela mesma são mantidas.

A principal vantagem no uso de Memórias Transacionais é a facilidade na definição de regiões atômicas, sem a necessidade de se preocupar com *locks* para as diferentes variáveis compartilhadas. Embora a maior parte dos trabalhos na área mostre que o desempenho de Memórias Transacionais é superior aos *locks*, nenhum dos deles apresenta uma conclusão de qual é o aumento médio de desempenho [50, 3, 20, 8, 24]

1.2.1.1 Versionamento de dados

Para que uma transação possa ser executada, é necessário manter tanto a versão antiga quanto a versão corrente dos dados modificados. Quando uma transação é abortada, os dados antigos são restaurados e os correntes descartados e, caso a

transação termine sem a ocorrência de violações, o valor corrente é mantido e o antigo é descartado. Existem duas formas de armazenar as versões dos dados em uma transação:

Versionamento adiantado (*eager versioning*): os valores correntes são armazenados na memória e os valores antigos são colocados em um *undo log*. A efetivação de transações que utilizam este método é mais rápida, pois os dados correntes já estão armazenados em memória, bastando eliminar o *backup* no *undo log*. Por outro lado, a tarefa de abortar a transação é mais lenta, pois é preciso utilizar o *undo log* para restaurar o conteúdo original da memória;

Versionamento tardio (*lazy versioning*): Os valores alterados são armazenados localmente em um *buffer* de escrita e são gravados na memória apenas quando a transação é efetivada. Neste método, abortar uma transação é mais rápido que fazer a sua efetivação, pois no primeiro é preciso apenas descartar o *buffer* de escrita, enquanto na efetivação é passar todo o conteúdo do *buffer* de escrita para a memória.

1.2.1.2 Detecção de conflitos

Cada transação possui um conjunto de leitura (*read set*) e um conjunto de escrita (*write set*) que armazenam os endereços acessados pela transação na leitura e escrita de dados, respectivamente. Uma violação (ou conflito) é detectada quando a interseção entre os conjuntos de escrita ou de escrita e leitura de duas transações distintas é não vazia. A detecção pode ser feita de duas formas:

Detecção de conflitos adiantada (*eager conflict detection*) ou pessimista: O conflito é detectado no instante em que uma posição de memória é acessada. Este método pode evitar que a transação seja executada desnecessariamente, mas também pode cancelar transações que poderiam ser executadas normalmente, caso a transação que conflitou com a mesma fosse abortada mais tarde por conflitos com uma terceira transação;

Detecção de conflitos tardia (*lazy conflict detection*) ou otimista: A detecção só é feita na efetivação da transação.

1.2.1.3 Aninhamento de transações

Outra questão importante que deve ser tratada pelos mecanismos de Memória Transacional é o aninhamento de transações. Uma transação é dita aninhada quando é definida no contexto de outra. A transação externa pode ser chamada de transação pai e a interna de filha. Sem resolver a questão do aninhamento, não seria possível criar novas transações a partir de operações encapsuladas em bibliotecas transacionais. Existem três abordagens para este problema:

Falattening: Faz com que as transações filhas se tornem parte da transação pai, sendo assim, caso a filha aborte, a pai também o será. Esta é a abordagem mais simples de ser implementada, mas pode aumentar a probabilidade de conflitos e limitar a concorrência, devido ao aumento do tamanho das transações;

Aninhamento fechado (*closed nesting*): Cada transação aninhada tem versionamento e detecção/resolução de violações independentes, além disso, podem acessar os estados da transação pai. Caso a transação filha sofra um cancelamento, ela poderá ser reiniciada, evitando o cancelamento da transação pai. Quando a transação filha é efetivada, seus conjuntos de leitura e escrita são mesclados com a transação pai e os dados alterados pela filha só se tornarão permanentes na efetivação da transação pai;

Aninhamento aberto (*open nesting*) [**33, 38**]: Permite efetivação e cancelamento independentes para transações aninhadas. Ações compensatórias são necessárias, por exemplo, se uma transação filha tiver sido efetivada e depois a transação pai seja cancelada, a transação pai deve corrigir todos os valores alterados pela filha, além de corrigir outras transações que tenham utilizado os valores em questão.

1.2.1.4 Virtualização

Segundo JaeWoong [8] sistemas de memória transacional deveriam garantir a execução correta mesmo quando as transações excedem o *time slice* do escalonador, a capacidade das *caches* e memória ou incluir mais níveis independentes de aninhamento do que o hardware suporta. Em outras palavas, sistemas de memória transacional devem prover virtualização de tempo, espaço e de aninhamento, de forma transparente.

1.2.1.5 Implementação: Hawdware x Software x Híbrida

HTM	Conflitos	Versionamento	Aninhamento	Virtualização		
				E[a]	T[b]	A[c]
Knight (1986)	tardio	tardio	–			
HMTM (1993)	adiantado	tardio	–			
Oklahoma (1993)	tardio	tardio	–			
TLR (2002)	adiantado	tardio	flat[d]	√[e]	√[5]	√[4]
TCC (2004)	tardio	tardio	–	√[f]	√	
UTM (2005)	adiantado	adiantado	flat	√	√	
LTM (2005)	adiantado	tardio	flat	√		
VTM (2005)	adiantado	tardio	flat	√	√	
Flat LogTM (2006)	adiantado	adiantado	flat	√		
Nested LogTM (2006)	adiantado	adiantado	open/closed	√		√[g]
PTM (2006)	adiantado	adiantado[h]	flat	√	√	
XTM (2006)	tardio	tardio	open/closed	√	√	√

[a] Espaço
[b] Tempo
[c] Aninhamento
[d] Bloqueios dentro de transações são especulados.
[e] Volta a adquirir bloqueios.
[f] Induz serialização.
[g] A partir de um certo nível, transações aninhadas são mescladas à transação de nível mais alto.
[h] Uma abordagem híbrida também pode ser usada.

Tabela 1.1: Características das principais HTMs [48]

Os sistemas de Memória Transacional em Hardware (HTM) [28, 24, 55, 46, 18, 20, 19, 3, 47, 37, 7, 8] são aqueles que possuem suporte arquitetural para a execução de transações. Sua grande vantagem é o desempenho. No entanto, os problemas de aninhamento e virtualização são muito complexos para se tratar em hardware, sendo

assim, a maioria das soluções não trata esses problemas ou os resolve parcialmente. A Tabela 1.1 mostra um resumo das características das principais abordagens de Memória Transacional em Hardware.

Os sistemas de Memória Transacional em Software (STM) [53, 15, 23, 21, 22, 31, 32, 49, 1] implementam em software o conceito de transação. Possui como vantagens a flexibilidade de implementação e exploração da semântica transacional e o fato de poder ser executada em máquinas atuais. A principal desvantagem é o baixo desempenho. A STM pode ser disponibilizada como uma biblioteca (API) ou pode ser integrada a um sistema de execução e/ou compilador.

Abordagens híbridas [29, 10, 50, 49, 1] combinam HTM e STM na tentativa de obter o desempenho da HTM e a flexibilidade de STM. As transações podem ser executadas em hardware, com alto desempenho ou em software, com "recursos ilimitados". Geralmente as transações são iniciadas em hardware e podem ser reiniciadas em softwares caso os recursos de hardware não sejam suficientes.

1.2.2 Escalonamento dinâmico com o uso da abordagem de Tomasulo

O algoritmo criado por Robert Tomasulo [60] realiza a emissão de instruções na ordem do programa, mas permite a execução e conclusão fora-de-ordem. Esta é uma primeira diferença em relação a máquinas *dataflow*, onde a emissão segue também o fluxo de dados e não a ordem do programa. Instruções são carregadas quando resultados produzidos por outras instruções sao destinadas a ela, conforme é visto no Capítulo 2.

O algoritmo de Tomasulo elimina os perigos RAW permitindo que as instruções sejam executadas apenas quando seus operandos estejam disponíveis (regra de disparo *dataflow*). Para isto são utilizadas as estações de reserva, que armazenam operandos destinados a cada uma das unidades funcionais. Quando todos os operandos de uma instrução destinados a uma determinada unidade funcional estiverem disponíveis em sua estação de reserva, a mesma é executada. A eliminação de perigos

do tipo WAR e WAW pode ser realizada com renomeamento de registradores.

Outra diferença de uma máquina de Von Neumann com Tomasulo e uma máquina *dataflow* é a existência de predição de desvios na primeira. Como é visto no Capítulo 2, no modelo de Von Newman, o *Program-Counter* é usado para determinar a próxima instrução a ser emitida e executada. Com Tomasulo a execução não precisa seguir a ordem do programa, mas como a emissão continua a ser feita em ordem, os desvios se tornam um limitador de paralelismo. Técnicas de predição de desvio são usadas para amenizar este limitador, mas caso a predição tenha sido incorreta, é necessário fazer um *rollback* das instruções emitidas e possivelmente executadas incorretamente. Em uma máquina *dataflow* isto não ocorre, pois, como a emissão de instruções segue o fluxo de dados, instruções de desvio são implementadas de forma que selecionem um dos dois possíveis subgrafos *dataflow* para enviar os operandos. A desvantagem é que é necessário uma instrução desse tipo para cada operando necessário nos dois blocos de instruções que seguem o desvio. Também é possível fazer predicação de valores, seguindo os dois possíveis caminhos de um desvio simultaneamente, e depois, selecionando os resultados de acordo com a condição lógica do desvio.

Para manter a corretude dos acessos a memória, o cálculo dos endereços de acesso é feito de forma seqüencial na unidade de endereços. Os endereços calculados são enviados para os *buffers* de carga e amazenamento. No caso de uma carga, o endereço é a única informação necessária para que a instrução seja executada, e no caso de um armazenamento é necessário também o dado a ser gravado, que será enviado ao respectivo *buffer* por uma das unidades de ponto flutuante. Como os endereços efetivos são calculados na ordem do programa, ao fazer esse cálculo é possível verificar a existência de conflitos entre a operação corrente (cujo cálculo de endereço acaba de ser feito) e as operações já existente nos *buffers* de carga e armazenamento. Essa verificação segue as seguintes regras:

- Se a operação em questão é um *Load*, o buffer de armazenamento é percorrido, em busca de endereços conflitantes. Caso tal endereço exista, esta instrução só poderá realizar o acesso à memória depois que a última escrita da qual ela

depende seja executada, eliminado perigos RAW. É possível também encaminhar, assim que disponível, o dado usado na última escrita para a operação de carga, evitando que a mesma faça o acesso à memória;

- Se a operação é um *Store*, os *buffers* de armazenamento e carga são percorridos para estabelecer a dependência da operação corrente e outro *Store* (*Write After Write*) e/ou com um *Load* (*Write After Read*).

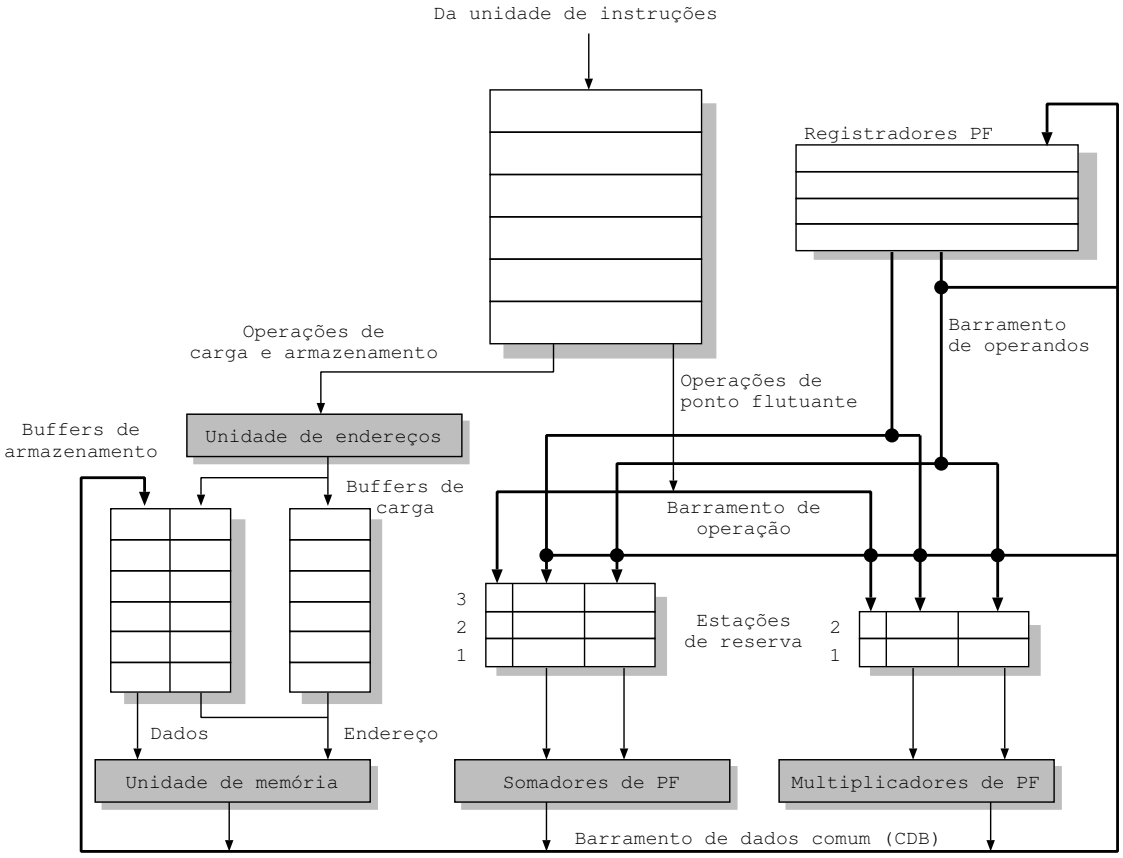


Figura 1.1: Estrutura básica de uma unidade MIPS de ponto flutuante com o uso do algoritmo de Tomasulo [45]

Para fazer a interconexão entre as estações de reserva, banco de registradores, saídas das unidades funcionais e o *buffer* de armazenamento é utilizado o Barramento de Dados Comum ou *Common Data Bus* (CDB). No WaveScalar, o número de elementos de processamento existente é muito superior ao número de unidades funcionais presentes na máquina onde foi aplicado o algoritmo de Tomasulo. Sendo

assim, um barramento compartilhado por todos esses elementos implicaria em altos custos de comunicção. Por isso, a comunicação entre os elementos de processamento no WaveScalar é feita usando uma rede hierárquica, onde os elementos são divididos em grupos. Uma interface de memória centralizada também aumentaria os custos de comunicação, desta forma em cada grupo de elementos existe uma interface de memória, tornando mais complexa a tarefa de identificar conflitos RAW, WAR e WAW. A Figura 1.1 mostra a estrutura básica de uma unidade MIPS de ponto flutuante com o uso do algoritmo de Tomasulo.

1.3 Objetivos e Contribuições

Este trabalho propõe modificações ao mecanismo de ordenação de memória do WaveScalar, buscando um maior "relaxamento" na ordem em que as mesmas são executadas. Este mecanismo chamado Transactional WaveCache permite execução especulativa e fora-de-ordem de operações de memória de diferentes ondas. Ele é o primeiro mecanismo que permite este tipo de relaxamento. Os outros mecanismos criados para reordenar operações de memória no WaveScalar trabalham dentro dos limites de uma onda. Além do impacto no desempenho do processador que incorpora tal mecanismo com o uso de aplicações artificiais, é objetivo desta pesquisa:

- Verificar a corretude dos programas executados quando utilizado o mecanismo;
- Fazer uma análise da relação entre o aumento no número de instruções para aplicações executadas com o uso do mecanismo e o grau de especulação do sistema adotado;
- Relacionar a alteração de desempenho com a capacidade do mecanismo em extrair concorrência das aplicações utilizadas;
- Relacionar a alteração de desempenho com diferentes graus de especulação do mecanimo nas aplicações utilizadas;
- Indicar possíveis alternativas de implementação para estudos futuros.

1.4 Estrutura do Trabalho

Este trabalho está dividido em 5 partes. A primeira parte compreede o presente capítulo introdutório com a motivação da pesquisa, trabalhos relacionados e objetivo. O Capítulo 2 apresenda a arquitetura WaveScalar, utilizada para implementação do mecanismo de reordenação de operações de memória, apresentado no Capítulo 3. A quarta parte é composta pelo Capítulo 4, onde são descritos os experimentos realizados e é feita uma análise dos resultados obtidos. Por fim, o último capítulo aborda as principais conclusões e os possíveis trabalhos futuros.

Capítulo 2

WaveScalar

2.1 Von Neumann x Dataflow

O modelo de Von Neumann vem regendo o projeto e construção de computadores desde o início da história da computação. Neste modelo a execução de programas é guiada pelo *program-counter* (PC), que indica a próxima instrução a ser executada. O PC é incrementado a cada instrução de forma a apontar para a próxima linha de código. Instruções de desvio podem alterar o seu valor fazendo com que haja um salto para outro trecho do programa. Dizemos que um programa executado nesses moldes obedece ao fluxo de controle. É claro ver que o modelo de Von Neumann é intrinsicamente seqüencial por duas razões: (1) o processador tem sua execução guiada pelo fluxo de controle (as instruções são executadas em seqüência) e (2) o sistema de memória precisa garantir a ordenação (ou ordenação aparente) das operações de memória de forma a preservar a ordem imposta pelos programas.

O modelo foi incrementado através de mecanismos como *pipelining*, predição de desvio [45], *renomeamento de registradores* [44] e escalonamento dinâmico [60] para tentar resgatar o paralelismo perdido pelo uso do modelo. Este tipo de esforço vem tornando os projetos de pocessadores cada vez mais complexos e custosos, além de dificultar o teste de projeto. Embora o desempenho do processador de Von Neumann tenha crescido exponencialmente por mais de trinta anos, as grandes estruturas associativas, *pipelines* profundos e redes de *bypass* usadas para extrair

paralelismo do modelo deixaram de ser escaláveis [2].

O modelo *dataflow* é uma alternativa para resolver os problemas de falta de paralelismo e escalabilidade do modelo de Von Neumann e algumas arquiteturas [13, 11, 51, 54, 17, 27, 16, 43, 56] surgiram baseadas nesse modelo. A execução de instruções em uma máquina *dataflow* é guiada pelo fluxo de dados, ou seja, uma determinada instrução inicia a sua execução de acordo com uma regra de disparo [12], baseada em dependências de dados verdadeiras. Sendo assim, uma instrução entra em execução tão logo tenha disponível os operandos de que necessite.

O modelo *dataflow* não exige uma ordem total na execução do programa (as instruções não precisam ser executadas na seqüência em que aparecem no código), exige apenas uma ordem parcial baseada nas dependências verdadeiras de dados. Desta forma é possível executar diversos trechos de código simultaneamente, contanto que não possuam dependência de dados entre si. Sendo assim, podem haver múltiplas instâncias de uma mesma instrução sendo executadas (por exemplo, em iterações diferentes de um *loop*). Para que isso ocorra sem que uma instância de uma instrução utilize dados de outra instância, é necessário *marcar* os operandos de cada instância para que possam ser identificados.

Segundo Swanson [58], as vantagens do *dataflow* são a de explicitar o paralelismo entre os diferentes caminhos do fluxo de dados e o modelo de execução descentralizado, que elimina a necessidade de um *program-counter* ou qualquer outra estrutura centralizada para controlar a execução das instruções. Por outro lado, as transferências de controle são mais caras e a falta de uma ordem total na execução das instruções torna difícil garantir a ordenação das operações de memória requerida pelas linguagens imperativas.

No modelo *dataflow* os programas são representados por grafos de fluxo de dados, ao invés de seqüências de instruções. Os nós no grafo representam instruções e as arestas são canais de comunicação que transportam dados entre as instruções, substituindo o tradicional banco de registradores. Na Figura 2.1 é possível observar, para um programa exemplo, o código em linguagem de alto nível (a) e o grafo

dataflow correspondente (b). Nesse exemplo, supondo que `A`, `B` e `C` já estão disponíveis, é possível executar as operações de adição entre `A` e `B` e a adição do imediato 15 a `C` simultaneamente, enquanto a operação de multiplicação precisa aguardar os operandos produzidos pelas instruções subjacentes.

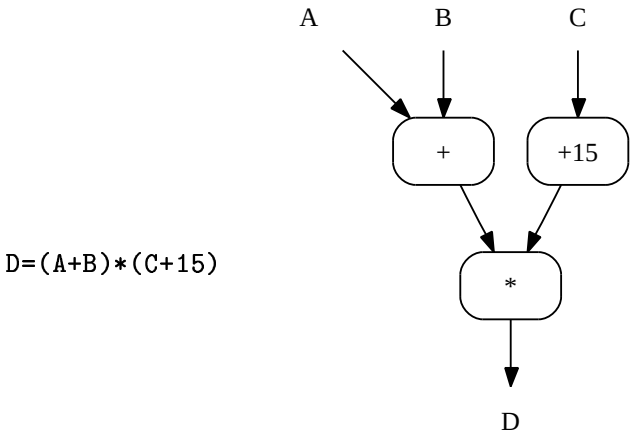


Figura 2.1: Fragmento de um grafo *dataflow*.

As primeiras máquinas *dataflow* [13, 11, 51, 54, 17, 27, 16, 43] precisavam de novas linguagens [41, 4, 26, 39, 12, 5, 34] que atendessem o paradigma do *dataflow*. Operações de memória eram executadas respeitando a ordem em que elas apareciam no código, tornando-se necessário inserir arestas no grafo *dataflow* para indicar as dependências entre tais operações de memória e as demais instruções. Por esse motivo, o modelo *dataflow* era incompatível com as linguagens até o momento usadas (como C, C++, java, entre outras), dificultando a aceitação do mesmo. Afinal, as empresas gastam tempo e dinheiro criando programas que funcionem de maneira confiável e, um modelo que promete maior desempenho não é o suficiente para que essas empresas abandonem os anos de desenvolvimento gastos em seus sistemas estáveis. A arquitetura WaveScalar [56, 58] foi criada para endereçar esse problema, sendo a primeira arquitetura *dataflow* compatível com as linguagens que se baseiam na premissa de que a ordenação das operações de memória imposta pelo programa é respeitada.

2.2 O conjunto de instruções do WaveScalar

O conjunto de instruções do WaveScalar se baseia no conjunto de instruções da máquina Alpha [14, 25], com modificações para que se possa representar o programa no modelo *dataflow*. Tais modificações são discutidas nessa seção.

O grafo de fluxo de dados é representado em linguagem *assembly* do WaveScalar da seguinte forma: cada instrução (um nó no grafo) é uma linha de código, onde os argumentos são as arestas. Arestas de saída (resultados) são representadas a esquerda do símbolo '←' e arestas de entrada (operandos) ficam à direita. Assim, o código *assembly* correspondente ao grafo da Figura 2.1 é o seguinte:

```
.label  begin
Add     temp_1 ← A, B
Add     temp_2 ← C, #15
Mul     D ← temp_1, temp_2
```

Algumas observações são importantes:

1. As arestas não correspondem a nenhuma entidade arquitetural (como registradores), permitindo o uso arbitrário de pseudo-registradores em compiladores;

2. A ordem em que as instruções aparecem no código não afetará a sua execução, pois a mesma segue o fluxo de dados e não de controle;

3. Cada instrução possui um endereço único usado para chamadas de função. *Labels* também podem ser usados para representar instruções específicas, inclusive com a possibilidade de aplicar operações aritméticas sobre os mesmos (no exemplo acima, begin+2 se refere a instrução 'Mul').

2.2.1 Desvios

No modelo guiado por fluxo de controle, desvios no código são executados alterando o *program-counter* para apontar para a instrução de destino. Como no modelo *dataflow* não existe PC, a solução é criar uma instrução que permita fazer

uma seleção entre um subgrafo ou outro para que os dados sejam direcionados para as instruções corretas. No WaveScalar isso pode ser feito através de duas instruções:

Steer (ρ): Recebe um valor de entrada e um booleano que seleciona uma dentre duas saídas possíveis, direcionando o valor de entrada para o subgrafo selecionado;

Select (ϕ): Recebe dois valores de entrada e um booleano que seleciona qual dos valores deverá ser enviado pela sua ÚNICA porta de saída. É usada para predicação, removendo a instrução de STEER de um possível caminho crítico e permitindo que ambos os subgrafos sejam executados para uma posterior seleção de resultado. Essa solução é mais custosa (desperdício de recursos), porém aumenta o paralelismo.

Na Figura 2.2 é dado um exemplo de um trecho de código contendo uma estrutura *IF-THEN-ELSE* (a) e o grafo *dataflow* que representa o mesmo, usando as duas alternativas apresentadas: a instrução de **STEER** (b) e a instrução ϕ (c).

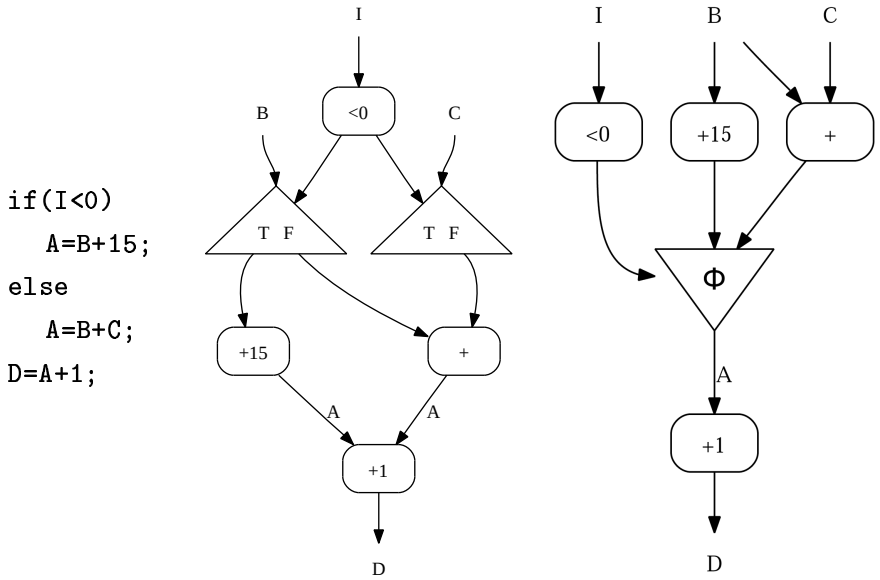


Figura 2.2: Implementação de desvios no WaveScalar.

2.2.2 *Loops*

Como citado no início desse capítulo, o modelo de execução de instruções do WaveScalar é controlado por uma regra de disparo baseada no fluxo de dados, ou seja, uma instrução só executa quando recebe de outras instruções os operandos de que necessita. Este fato levanta uma questão importante acerca dos *loops*. Se uma mesma instrução é executada diversas vezes, como saber para qual iteração dessa instrução no *loop* será enviada a mensagem com o operando que causará o seu disparo? Essa questão pode ser endereçada de duas maneiras:

***Dataflow* estático**: [13, 11] Impedir a execução de diferentes iterações simultaneamente. Embora seja uma solução simples, ela limita fortemente o paralelismo, pois uma iteração $\boldsymbol{X}$ só poderá iniciar sua execução mediante o término da iteração $\boldsymbol{X - 1}$;

***Dataflow* dinâmico**: [54, 17, 27, 16, 43] Permitir a existência de mais de uma instância de uma mesma instrução, sendo que cada instância pertence a uma iteração do *loop*. Dessa forma, os operandos acessados e os resultados produzidos pela mesma instrução, em diferentes iterações, poderão ser identificados por um *tag* que representa àquela instância. A regra de disparo também é alterada adicionando uma verificação das *tags* dos operandos para que os mesmos sejam enviados às instâncias corretas das instruções.

A solução adotada no WaveScalar para os laços é o *dataflow dinâmico*. O código fonte é dividido em ondas (*waves*) numeradas. Cada iteração é considerada uma onda distinta, bem como os trechos de código localizados antes e depois do *loop*. Para que uma instrução seja executada para uma onda $\boldsymbol{X}$, para cada operando enviado para uma instrução, é verificada sua *tag*. Quando todos os operandos que contenham o número $\boldsymbol{X}$ em suas *tags* estiverem disponíveis, a execução da instrução é disparada. Instruções executadas com operandos de entrada marcados com $\boldsymbol{X}$, produzirão operandos de saída marcados também com $\boldsymbol{X}$. Para enviar operandos de uma iteração para outra, é necessário fazer o incremento do número da onda

em suas *tags*. Isto é feito dinamicamente por uma instrução chamada WA (*Wave-Advance*), que repassa operandos de uma onda para a próxima. A Figura 2.3 mostra um trecho de código contendo um *loop for* (a) e o grafo *dataflow* correspondente (b). É possível observar a divisão do código em três blocos, marcados com linhas pontilhadas. O primeiro bloco é a onda que se encontra antes do *loop*. O segundo bloco é o *loop* com suas dez ondas (iterações) e o último é a onda posterior ao *loop*. As instruções de WA do segundo bloco recebem os operandos `i`, `a` e `b` de entrada do laço e as do terceiro bloco enviam os operandos `a` e `b` de saída. No subgrafo que representa o *for*, é possível visualizar três colunas:

- A coluna do meio representa o incremento da variável *i*, de controle do *loop*. Esta coluna é independente das demais, podendo ser executada em paralelo com as outras colunas para todas as iterações, sem aguardar o término de cada iteração;

- A coluna da esquerda é responsável pela execução da linha *a=(a+1)*2* e é dependente do resultado da comparação realizada na coluna do meio. Ela pode ser executada em paralelo com as instruções das demais colunas, mas o avanço para a iteração seguinte é limitado pela velocidade de execução da coluna do meio;

- A coluna da esquerda é responsável pela execução da linha *b=(b+1)*3* e também é dependente da comparação feita na coluna do meio.

2.2.3 Funções e Procedimentos

Para realizar chamadas de funções ou procedimentos no modelo de Von Neumann é necessário (1) empilhar os valores contidos nos registradores usados pelo chamador, (2) preencher registradores específicos com os argumentos e endereço de retorno e (3) alterar o *program-counter* para o endereço da função ou procedimento. O retorno da função ou procedimento é feito (1) armazenado os valores de retorno (se existentes) em registradores específicos ou na pilha, (2) executando o salto para o endereço

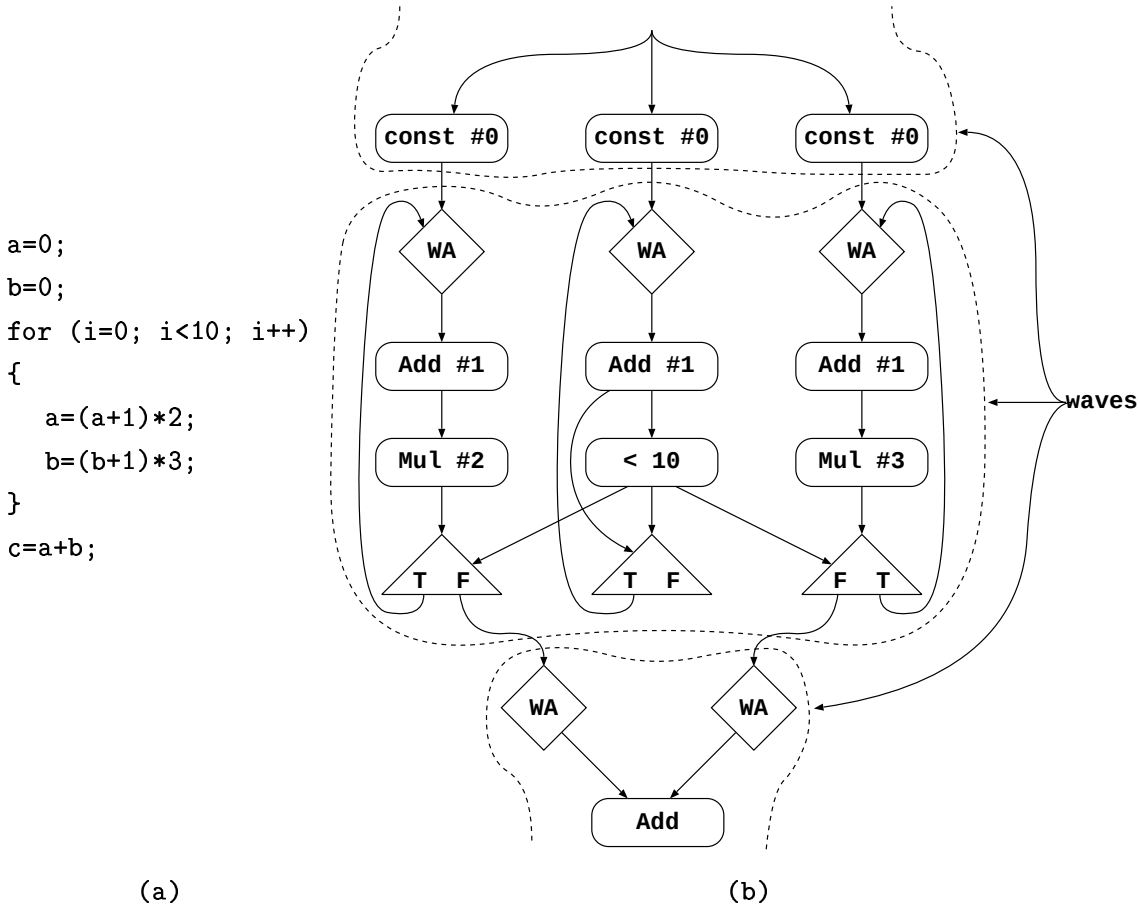


Figura 2.3: *Loop* e Ondas no WaveScalar.

de retorno armazenado em um dos registradores, e (3) restaurando o conteúdo dos registradores (armazenados na pilha) para o chamador.

No WaveScalar não existem registradores, eliminando a necessidade de salvar operandos na pilha. É necessário apenas enviar os operandos para as instruções que os utilizam, dentro da função, criando uma dependência de dados entre o código da mesma e o seu chamador. Porém, a função a ser chamada pode não ser indicada de maneira estática no código, sendo seu endereço definido apenas em tempo de execução, como por exemplo, quando se utiliza uma função implementada em uma biblioteca compartilhada. Duas instruções foram criadas para serem utilizadas em conjunto na implementação das chamadas de funções ou procedimentos:

Indirect-Send: Recebe o endereço base da instrução destino, o deslocamento e um argumento de entrada da função, fazendo com que o argumento seja enviado para a instrução *Landing-Pad* indicada dentro da função chamada;

Landing-Pad: Exerce a função de destino, recebendo o argumento enviado através da instrução *Indirect-Send* e encaminhando-o para as instruções da função chamada.

Na Figura 2.4 é dado um trecho de código contendo a definição de uma função e o seu ponto de chamada (a). Em seguida é exibido o grafo *dataflow* correspondente (b). As linhas tracejadas representam as arestas estabelecidas dinâmicamente pelo uso das instruções *Indirect-Send* e *Landing-Pad*. Uma função pode ser chamada em diversos pontos, podendo cada uma dessas chamadas ser executada em paralelo. Para permitir a identificação de cada uma das instâncias da função para que o envio de operandos ocorra corretamente, as instruções *Landing-Pad*, são sempre seguidas por instruções *Wave-Advance*.

O conjunto de instruções do WaveScalar, exibido nessa seção não difere, em essência de outros projetos de máquina *dataflow* tais como [11, 13, 16, 17, 27, 43, 51, 54]. É um conjunto de instruções que permite a execução de programas apenas baseado no fluxo de dados. O diferencial trazido pelo WaveScalar está em seu subsistema de memória, que foi projetado para permitir a execução de linguagens

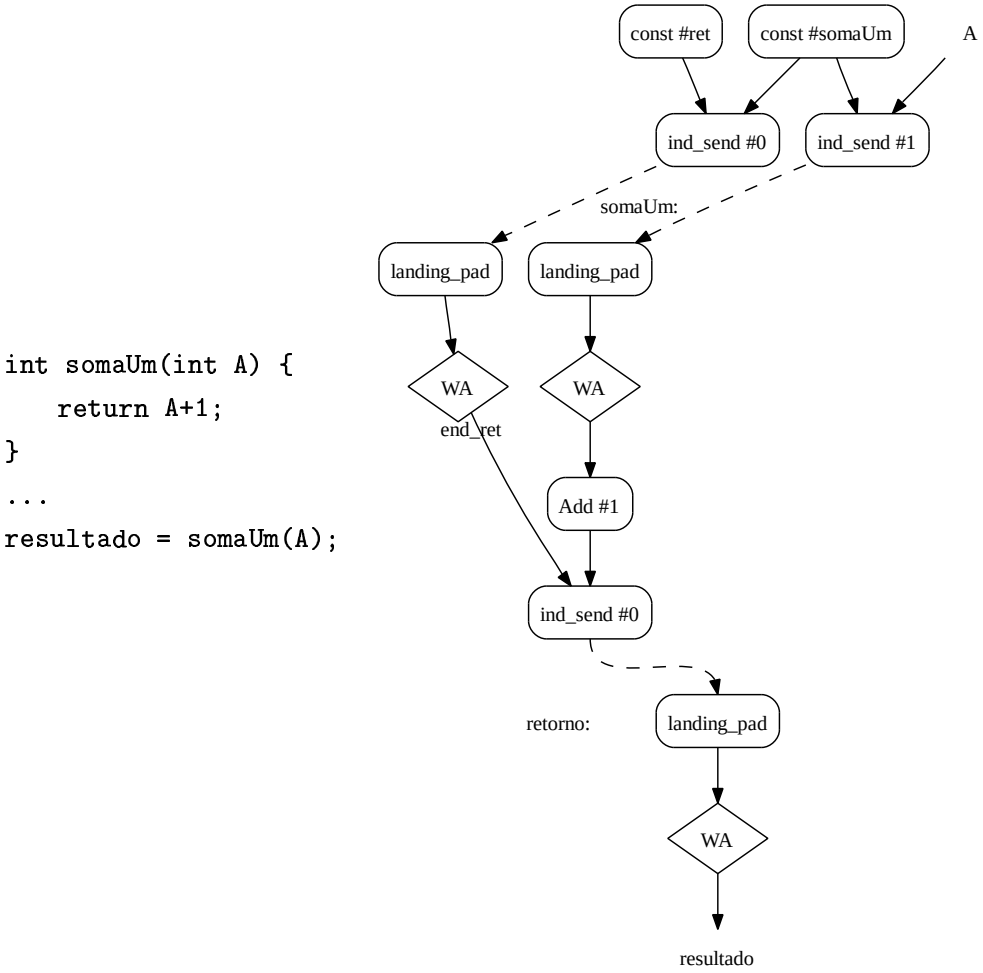


Figura 2.4: Funções e procedimentos no WaveScalar.

imperativas (C, Java, Pascal e outras) [59, 58, 56]. O subsistema de memória do WaveScalar é discutido em detalhes na próxima seção.

2.3 *Wave-Ordered Memory*

Para o usuário de um programa, não importa de que forma ele é executado, contanto que os resultados sejam apresentados de maneira correta e o mais rápido possível. Para que os resultados sejam corretos, é necessário que os mesmos sejam calculados corretamente e escritos na memória segundo uma ordem que, para a máquina de Von Neumann é estabelecida pelo próprio fluxo de controle do programa.

Como o modelo *dataflow* é totalmente paralelo e distribuído, a não ser que uma ordem para as operações de memória seja explicitada o resultado pode ser imprevisível.

As arquiteturas *dataflow* criadas antes do WaveScalar foram acompanhadas por novas linguagens [41, 4, 26, 39, 12, 5, 34], onde era possível mostrar dependências entre instruções de acesso à memória e outras instruções. Desta forma, uma determinada operação de memória só seria realizada depois que todos os seus pré-requisitos fossem satisfeitos. As linguagens imperativas (como C, C++, Java, etc) foram criadas com base no modelo de Von Neumann e, portanto, tais linguagens são incompatíveis com as arquiteturas *dataflow* que precederam o WaveScalar. Swanson argumenta que esse fato foi o principal obstáculo para o sucesso de tais arquiteturas, pois tornou inviável a transição do modelo de Von Neumann para o modelo *dataflow* [59]. Para fazer tal transição seria necessário reescrever todos os programas usando uma linguagem compatível.

O WaveScalar é a primeira arquitetura *dataflow* que atende os requisitos de acesso a memória das linguagens imperativas. Seu subsistema de memória garante execução das operações de memória seguindo a ordem em que as mesmas aparecem no código, sem que seja necessário explicitar dependências no grafo de fluxo de dados.

2.3.1 Chaves de ordenação

Para garantir a ordenação das operações de memória, durante o processo de compilação, é atribuída uma chave $<A,C,P>$ a cada operação de memória dentro de uma onda. Esta chave contém um número que identifica a operação corrente (C), o número da operação de memória anterior (A) e número da operação posterior (P). Utilizando as informações contidas nas chaves de cada operação de memória, é possível formar uma cadeia entre tais operações dentro de uma onda, garantindo assim uma **ordem parcial** para cada onda. Depois disso, ordena-se cada uma destas cadeias pelo número da onda, obtendo a **ordem total**. Algumas observações são necessárias acerca da chave que acompanha as operações de memória:

- A primeira operação de memória não possui operação anterior, portanto, na chave que acompanha a operação, o número A é marcado com uma constante que indica esta situação e o caractere "." é usado no texto para representá-la;

- O mesmo acontece com o número P da última operação de memória de uma onda;

- No caso de desvios, por exemplo um *If-Then-Else*, onde existe mais de um caminho possível no fluxo de dados, podem ocorrer operações de memória em qualquer um dos caminhos do fluxo. Neste caso, a operação de memória que precede o desvio desconhece o número da operação posterior (P). Uma constante é adotada para representar esta indeterminação e o caractere "?" é usado no texto para representá-la.

- O mesmo acontece para a operação seguinte ao bloco do desvio, que tem o número (A) indeterminado.

Na Figura 2.5 são exibidas apenas as operações de memória extraídas de uma estrutura *If-Then-Else* (retangulos pontilhados). As setas pontilhadas representam a ordem de execução a ser respeitada segundo as chaves de ordenação. É possível observar também a cadeia das operações executadas no subsistema de memória, assumindo que o lado direito do desvio foi seguido.

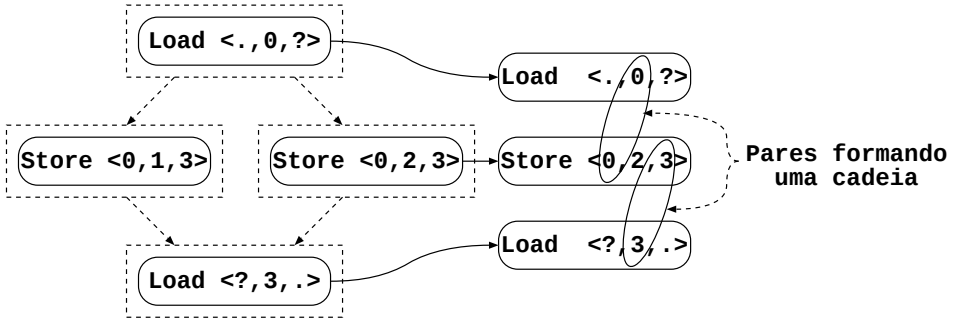


Figura 2.5: Cadeia de operações de memória em uma estrutura *If-Then-Else*.

Quando, em um desvio, não houve operações de memória em um de seus possíveis *caminhos*, não há como estabelecer uma cadeia entre as operações que precedem e

Figura 2.6: **Resolvendo a ambigüidade**: Um *If-Then-Else* sem (a) e com (b) a instrução **MenNop**.

sucedem este bloco, caso o *caminho* em questão seja escolhido durante a execução. Isto ocorre pois instruções de acesso à memória que precedem e que sucedem um desvio possuem em suas chaves de ordenação o valor indeterminado ("?") para os números (P) e (A), respectivamente. A operação que formaria a cadeia que as conecta deveria estar justamente dentro do bloco do desvio. Para resolver esta ambigüidade foi criada a instrução **MenNop**, que deve ser inserida nos *ramos* dos desvios onde não houve operações de memória. Na Figura 2.6 é possível observar um exemplo do problema descrito acima (a) e sua respectiva solução (b).

2.3.2 Regras de Ordenação

O modelo estabelece para cada onda uma estrutura, chamada de *operation buffer*, que mantém ordenadas pelo número (C) as requisições de acesso à memória. Durante a execução, requisições podem chegar em qualquer ordem. Cada requisição recebida é colocada no *operation buffer* de sua respectiva onda e é feita uma verificação para determinar se a requisição já está pronta para ser resolvida. Quando uma requisição estiver pronta para fazer o acesso à memória, ela é removida do *operation buffer* e é resolvida pelo subsistema de memória. Dentre os *operation buffers* que ainda possuem requisições pendentes, àquele que está associado a onda de menor número é chamado de *operation buffer* corrente. Apenas requisições do *operation buffer* corrente são resolvidas, garantindo assim a ordenação entre as ondas.

Dada uma requisição de acesso à memória $\boldsymbol{M}$ e sua chave <$A_{\boldsymbol{M}}$,$C_{\boldsymbol{M}}$,$P_{\boldsymbol{M}}$> e sendo $\boldsymbol{F}$ a última requisição resolvida e sua chave <$A_{\boldsymbol{F}}$,$C_{\boldsymbol{F}}$,$P_{\boldsymbol{F}}$> a verificação para determinar se M está pronta, segue os seguintes critérios:

- se $C_{\boldsymbol{M}} == 0$, $\boldsymbol{M}$ pode executar (está pronta);
- se $A_{\boldsymbol{M}} == C_{\boldsymbol{F}}$, $\boldsymbol{M}$ pode executar (está pronta);
- se $P_{\boldsymbol{F}} == C_{\boldsymbol{M}}$, $\boldsymbol{M}$ pode executar (está pronta).

Quando M está pronta, ela é executada e assume o lugar de $\boldsymbol{F}$. O procedimento é repetido para a requisição seguinte até que se atinja uma requisição que não esteja pronta ou o fim da onda. Quando o fim da onda é atingido, o *operation buffer* da próxima onda se torna o *operation buffer* corrente. Como uma onda pode ter diversos pontos de saída que são conhecidos apenas durante a execução, a tarefa de identificar o término de uma onda pode ser complexa e custosa. Para evitar essa verificação a instrução CWA (*Canonical Wave Advance*) foi criada. Um CWA é inserido em cada onda durante o processo de compilação para criar uma cadeia entre a última operação de memória da onda em questão e a primeira operação de memória da próxima onda, segundo as seguintes regras:

- O número C de uma operação de *CWA* é sempre 1;
- A número P da última operação de memória de uma onda é 1;
- A primeira operação de memória de uma onda possui A=1, exceto quando a mesma se encontra em um dos ramos de um desvio, quando A="?".

2.3.3 Melhorias no subsistema de memória: *Ripple numbers* e *Decoupled Stores* (*Partial Stores*)

Duas melhorias no subsistema foram criadas e implementadas pelos próprios idealizadores do WaveScalar em [59]: Os *Ripple numbers* e *Partial Stores*.

Os *Ripple numbers* foram criados para proporcionar um certo grau de paralelismo na execução das operações de memória. Sem este esquema as operações de memória são executadas seqüencialmente segundo a ordem em que aparecem no código fonte. O esquema proporciona um relaxamento de operações de *Load* que ocorram entre dois *Stores*, permitindo que sejam reordenadas entre si. Como é garantido que nenhuma alteração na memória ocorre entre os dois *Stores*, qualquer que seja a ordem de execução dos *Loads*, eles sempre estarão lendo os valores corretos da memória. Para a implementação deste novo esquema foi criado um quarto número na chave de ordenação chamado de *Ripple number* (R). Dada uma operação de memória M com sua chave $<A_{\boldsymbol{M}},C_{\boldsymbol{M}},P_{\boldsymbol{M}}>.R_{\boldsymbol{M}}$ e S o último *Store* que ocorreu antes de M com sua chave $<A_{\boldsymbol{S}},C_{\boldsymbol{S}},P_{\boldsymbol{S}}>.R_{\boldsymbol{S}}$, temos que:

- $R_{\boldsymbol{M}} = C_{\boldsymbol{M}}$, se M for um *Store*;

- $R_{\boldsymbol{M}} = C_{\boldsymbol{S}}$, se M for um *Load*.

Usando os *Ripple numbers* e criando mais uma regra de ordenação é possível estabelecer uma cadeia entre um *Store* e os *Loads* que o sucedem. Dada a última operação de memória executada M, um *Load* L pode ser executado se ele é o próximo na cadeia de operações (como anteriormente) ou se $R_{\boldsymbol{L}} \leq C_{\boldsymbol{M}}$. *MenNops* são tratados como *Loads*. A Figura 2.7 mostra a formação dessas cadeias em um bloco básico (a) e em uma estrutura *If-Then-Else* (b). Observa-se em (b) que o *Load* posterior ao desvio possui o *Ripple number* igual a dois, que é o maior *Ripple* observado nos dois ramos do desvio. Segundo [59], o ganho de paralelismo obtido com esta melhoria está entre 0 e 5%.

Os *PartialStores* foram criados para tirar vantagem do fato dos endereços de acesso à memória, para operações de *Store*, geralmente estarem disponíveis antes de seu dados. Para implementar um *PartialStore* são criadas duas instruções: uma para o envio do endereço da operação de *Store* e outra para o envio do dado. Quando o subsistema de memória recebe de uma operação de *Store* apenas o endereço, essa operação é tratada normalmente pelo mecanismo de ordenação. Chegando a sua vez de ser atendida, se o endereço ainda não tiver sido recebido, a operação é colocada

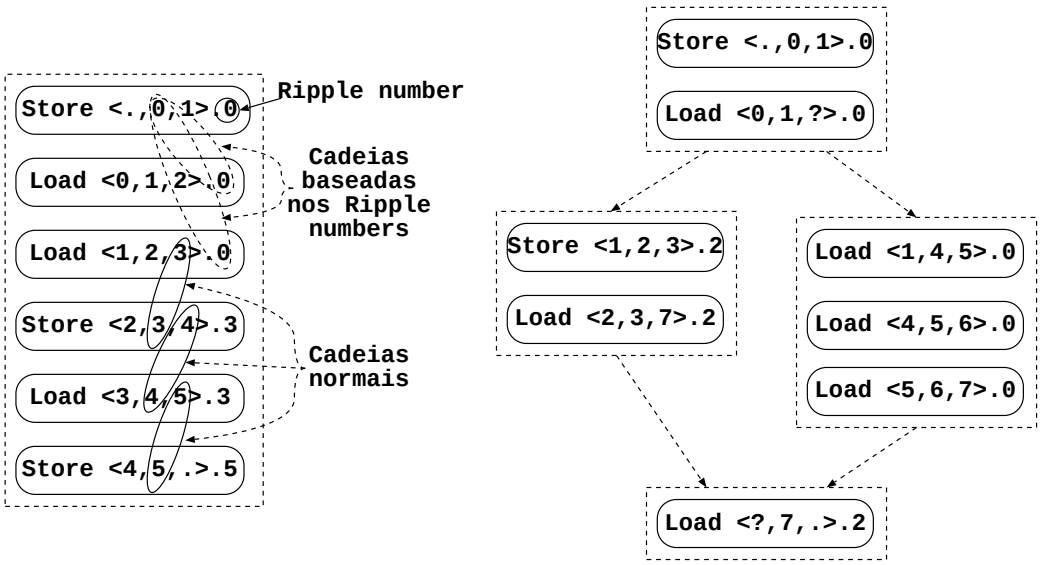


Figura 2.7: **Ripple numbers**: Uso em um bloco básico (a) e em uma estrutura do tipo *If-Then-Else* (b).

em uma fila chamada de *partial store queue*. Todas as operações seguintes que estiverem relacionadas com o mesmo endereço são também enfileiradas. Assim que o subsistema de memória receba o dado para o *Store* que criou a fila, é possível resolver todas as operações que se sucedem rapidamente. Caso o dado chegue antes da operação ser atendida ela é resolvida como um *Store* normal assim que chegar a sua vez. Se o dado para um *Store* chega antes do endereço, o subsistema de memória não atende a requisição e só o fará quando o endereço estiver disponível. Segundo [59], o ganho de paralelismo obtido com esta melhoria está entre 21 e 46% e com uma média de 30%.

2.4 A arquitetura WaveScalar

Nas seções anteriores, o conjunto de instruções e o modelo de ordenação das operações de memória foram apresentados. Nessa seção é descrita a arquitetura *WaveCache*, que implementa este modelo. A *WaveCache* é composta basicamente de um conjunto de *Processing Elements* (*PEs*) idênticos e simples, o *hardware* responsável pela ordenação das operações de memória (*wave-ordered memory*) e uma

rede de interconexão hierárquica com uma estrutura de comunicação diferente para cada nível da hierarquia.

A arquitetura tem como bloco de construção o *Cluster*, contendo quatro *Domains*, cada um com oito *PEs*, agrupados em *Pods* (com dois *PEs* cada). Cada *Cluster* possui uma cache L1, o *hardware* que provê a interface com a *wave-ordered memory* e um *switch* que permite a comunicação com *Clusters* adjacentes (norte, sul, leste e oeste). Os *Clusters* são replicados no *Die* formando uma matriz. A cache L2 é conectada nas bordas da matriz, juntamente com os diretórios usados no protocolo de coerência. A Figura 2.8 mostra uma visão final da arquitetura, cujos detalhes são exibidos ao longo da seção.

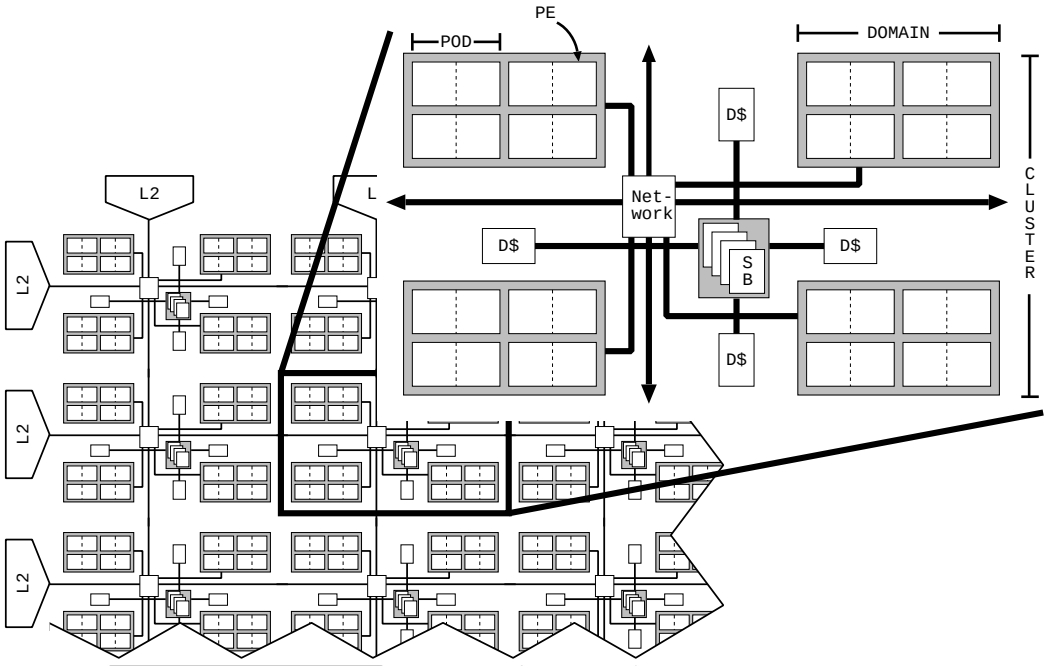


Figura 2.8: Uma visão geral da arquitetura *WaveCache* [59].

2.4.1 Preparando a execução: carregando um programa no WaveScalar

Quando um programa é executado no WaveScalar, cada instrução estática é atribuída a um *PE*. Obviamente, o número de instruções no programa pode ser maior

que o número de *PEs* disponíveis e, portanto, múltiplas instruções são dinâmicamente mapeadas para um mesmo *PE*, seguindo um algoritmo de *placement* [35, 36]. Com a evolução da execução do programa, algumas instruções se tornam desnecessárias e são substituídas pelas instruções ativadas que estão em memória. O algoritmo de mapeamento das instruções possui dois objetivos conflitantes: minimizar a comunicação entre *PEs* na troca de operandos, tentando alocar com a maior proximidade possível as instruções dependentes (no mesmo *PE* ou *Pod*); e espalhar instruções independentes por diversos *PEs* para explorar o paralelismo do modelo *dataflow*.

2.4.2 O *Processing Element*

As funções principais de um *Processing Element* são de implementar a regra de disparo do modelo *dataflow* e executar instruções. Cada *PE* possui uma ALU, estruturas de memória para armazenar operandos, lógica para controlar a execução e comunicação, e de um *buffer* para armazenar instruções. O *PE* possui um *pipeline* de cinco estágios, com redes de *bypass* que permitem a execução de instruções dependentes no mesmo *PE*. A Figura 2.9 mostra a arquitetura de um *PE*, destacando os estágios de seu *pipeline*, descritos a seguir:

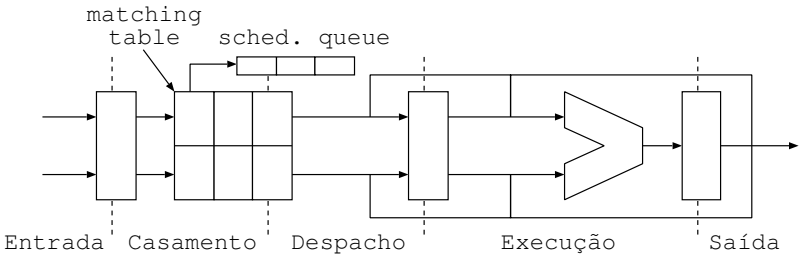


Figura 2.9: Um *Processing Element* [59].

Entrada: Chegada de operandos no *PE*, enviados por outro *PE* ou pelo próprio através da rede de *bypass* da ALU.;

Casamento de *tag* dos operandos (*Match*): Os operandos são colocados em uma tabela chamada *matching table*, onde suas *tags* são verificadas. A *matching table* é associada a *tracker board*, que determina quando uma instrução

já possui todos os seus operandos disponíveis, estando pronta para a execução. Quando isto ocorre a instrução é movida para uma fila de instruções prontas para execução, chamada *scheduling queue*;

Despacho: O *PE* seleciona uma instrução da *scheduling queue*, lê seus operandos da *matching table* e os envia para o estágio de execução. Se o destino do resultado da instrução é local, o *PE* inicia especulativamente a execução da instrução destino para o próximo ciclo (a instrução é inserida na *scheduling queue*);

Execução: Executa a instrução e envia os resultados para o estágio de saída, exceto em duas situações: quando a mesma foi disparada especulativamente e ainda não possui todos os operandos de que necessita e quando o *buffer* de saída está cheio. No primeiro caso, a instrução é eliminada da *scheduling queue* e no segundo ocorre um *stall* no estágio de execução até que haja espaço disponível;

Saída: Os resultados da instrução são enviados pelo barramento de saída para o próprio *PE* ou outro remoto. É feita a difusão da informação pelo barramento que conecta os *PEs* em um *Domain*, onde o destino responderá com um ACK ou NACK, informando se aceitou ou não a mensagem. O tempo máximo decorrido entre o envio da mensagem e o recebimento da resposta é de 4 ciclos. Para que o *PE* não fique aguardando todo esse tempo por uma resposta, ele assume que a mensagem será aceita e a move para um *reject buffer* com 4 entradas, deixando o *buffer* de saída livre para o próximo ciclo. Quando todos os destinatários tiverem recebido a mensagem, ela é excluída do *reject buffer*; caso não tenha sido aceita (NACK), ela é reinserida no *buffer* de saída e reenviada para os destinatários que a rejeitaram.

2.4.3 A rede de interconexão

Os *Processing Elements* comunicam-se entre si para a passagem de resultados. Para reduzir os custos de comunicação a rede de interconexão organiza os *PEs* de

forma hierárquica. A Figura 2.10 apresenta as diferentes redes de comunicação:

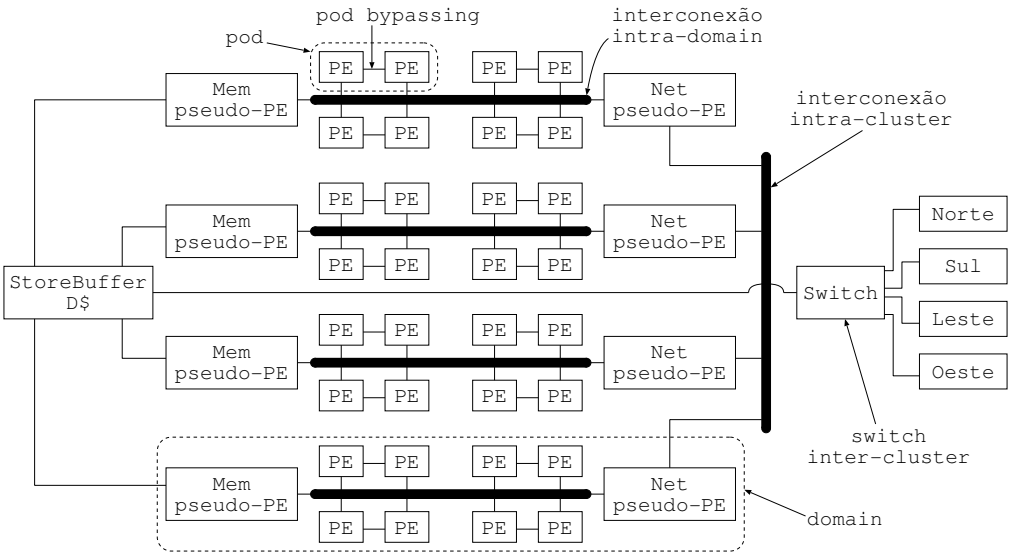


Figura 2.10: A rede de interconexão de um cluster [59].

POD: Existe uma conexão entre cada par de *PEs*, de forma que cada um deles possa verificar a rede de *bypass* de seu vizinho, compartilhando informações sobre o escalonamento de instruções e possibilitando a execução de instruções dependentes da mesma forma como isto é feito localmente;

Intra-domain: Quatro *Pods* são agrupados em um *Domain*. A comunicação entre os *PEs* de um *Domain* é feita por meio de um barramento de difusão. Cada *Domain* inclui ainda um *Mem pseudo-PE* que comporta-se como *gateway* para o subsistema de memória e um *Net pseudo-PE* que funciona como *gateway* para *PEs* em outros *Clusters*;

Intra-cluster: Quatro *Domains* são agrupados em um *Cluster* e a comunicação entre os mesmos é feita ponto a ponto através de um *switch* local que conecta os seus *Net pseudo-PEs*;

Inter-cluster: A comunicação entre *clusters* é feita usando uma NOC (*network on-chip*) que liga os *switches* de cada *cluster* com seu vizinhos (norte, sul, leste e oeste), além de conecta-los aos seus respectivos *StoreBuffers* (detalhado na

Seção 2.4.4) e à rede intra-cluster. A rede inter-cluster é também usada para o tráfego de informações do protocolo de coerência das caches L1.

2.4.4 O *StoreBuffer*

O *StoreBuffer* é a estrutura responsável pelo mecanismo de ordenação das operações de memória (*wave-ordered memory*). Cada *cluster* contém um StoreBuffer, que será responsável pela ordenação de algumas ondas. Quando um *PE* deseja realizar uma operação de memória, ele envia a requisição ao *Mem pseudo-PE* de seu *Domain* e o mesmo a encaminha ao *StoreBuffer* de seu *Cluster*. Existe uma estrutura chamada de *mapa de ondas* que relaciona o número de cada onda com o *StoreBuffer* responsável pela mesma. O *StoreBuffer* lê o número da onda ao qual pertence a requisição e verifica no mapa de ondas se já existe algum *StoreBuffer* com a custódia daquela onda, podendo ocorrer três possíveis situações:

1. Se a custódia da onda for do *StoreBuffer* local, o mesmo atenderá a requisição;
2. Se a custódia da onda for de um *StoreBuffer* remoto, o *StoreBuffer* local irá encaminhar a requisição através da rede de interconexão inter-cluster;
3. Se a custódia da onda não está definida, o *StoreBuffer* local irá assumir a custódia e atenderá a requisição

O mapa de ondas é armazenado na memória principal e uma visão consistente do mesmo, por todos os *StoreBuffers*, é garantida pelo protocolo de coerência das caches L1. A leitura é feita normalmente e a escrita, através de operações atômicas do tipo *read-modify-write*.

Quando uma onda é completada, o *StoreBuffer* verifica no mapa de ondas, o responsável pela onda seguinte e envia uma mensagem para informá-lo que as operações das ondas anteriores estão completas e que ele pode começar a atender as requisições da próxima. Segundo Swanson [59], devido ao algoritmo de *placement*, a execução de uma única *thread* tende a estar localizada em um único *Cluster*. Sendo

assim, as leituras e atualizações do mapa de ondas requerem um tráfego de coerência mínimo. Medições feitas em [59] mostram que na média, 40% do tráfego da rede permanece detro do próprio *PE* ou seu vizinho no *POD*, 52% do tráfego permanece dentro do domínio. O número de mensagens trocadas entre *clusters* corresponde a 1,5% do tráfego total da rede. Com relação à natureza das mensagens, em média 80% do tráfego é para troca de dados de operandos e menos de 20% para memória e tráfego de coerência. Desta forma, as mensagens entre *StoreBuffers* são geralmente locais, ou seja, o *StoreBuffer* local possui, também, a custódia da próxima onda.

Um *StoreBuffer* possui os seguintes componentes arquiteturais: uma *Ordering Table*, uma *Next Table*, um *ISSUED Register* e duas *Partial Store Queues*. Dada uma requisição de memória M a ser tratada e sua respectiva chave de ordenação $<A_M,C_M,P_M>$, a requisição M será processada em *pipeline* pelo *StoreBuffer* responsável. Os estágios deste *pipeline* são descritos a seguir e seu esquema é exibido na Figura 2.11.

Entrada: Aceita até quatro novas requisições por ciclo. Insere as informações da requisição, como endereço, operação e dados (se estiverem disponíveis para o caso de um *Store*) na posição C_M da *ordering table*. Caso P_M esteja definido (diferente de "?") a posição C_M da *next table* é preenchida com P_M. Se A_M estiver definido, a posição A_M da *next table* é preenchida com C_M;

Escalonamento: Mantêm o *ISSUED register* que aponta para a próxima operação de memória a ser despachada para a cache de dados. Este registrador é utilizado para fazer a leitura de quatro entradas de cada uma das tabelas (*ordering/next tables*). Se algum *link* entre as operações puder ser formado (as entradas da *next table* estiverem preenchidas), as operações são despachadas para o estágio de saída e o *ISSUED register* é adiantado;

Saída: Lê e processa as operações despachadas e, devido ao esquema que permite a existência de *PartialStores* (Seção 2.3.3), podem ocorrer quatro situações possíveis:

1. Quando a operação é um *Load* ou *Store* cujo dado esteja presente, a

operação prossegue para a cache de dados;

2. Quando a operação é um *Load* ou *Store* e existe uma *partial store queue* para seu endereço, a operação é enviada para a mesma;
3. Quando a operação é um *Store* cujo dado não esteja presente (*PartialStore*) e não existe uma *partial store queue* para a mesma, uma *partial store queue* livre é alocada e a operação é enviada para a mesma;
4. Quando a operação é um *Load* ou *Store* e não existe uma *partial store queue* disponível ou a *partial store queue* do endereço está cheia, a operação é descartada e o *ISSUED register* é atrasado. A operação será atendida mais tarde.

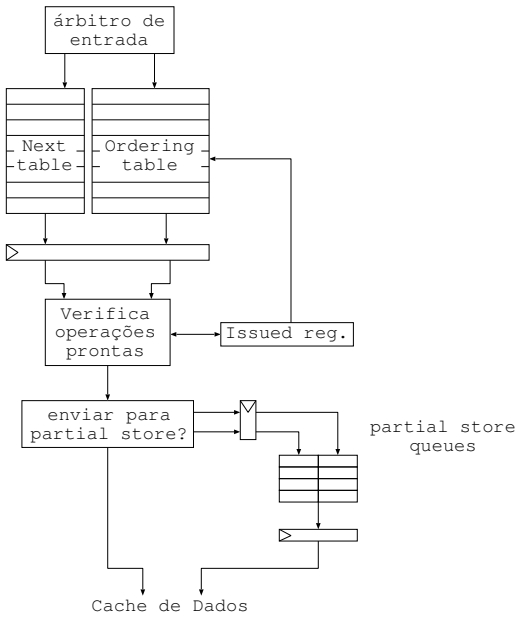


Figura 2.11: O StoreBuffer [59].

Embora operações de memória de apenas uma onda sejam atendidas por vez, o *StoreBufer* possui quatro *next/ordering tables* para receber os pedidos de acesso à memória de até quatro ondas pelas quais é responsável. Isto permite que as cadeias entre as operações de memória de ondas futuras sejam montadas com antecedência, para que possam ser executadas, tão logo a onda anterior seja terminada. Se é feita uma solicitação de acesso à memória e não existe uma *next/ordering table* associada

àquela onda e não existam *next/ordering tables* livres, a solicitação não será recebida e o *PE* solicitante tentará novamente, mais tarde.

2.5 As ferramentas do WaveScalar

Ferramentas foram implementados para que o WaveScalar pudesse ser avaliado. Os componentes mais importantes são o sistema de alocação de instruções (já discutido na Seção 2.4.1, um tradutor binário e simuladores (funcional e arquitetural), apresentados a seguir:

Tradutor binário: Converte os binários da máquina *Alpha AXP* para o *assembly* do WaveScalar. Por apresentar limitações (descritas em [59]), é necessário fazer uso de um compilador C que siga as convenções de chamada da máquina Alpha, como o compilador *CC* para o *Tru64*;

Simuladores: Foram criados dois simuladores: o funcional (chamado de Surfer) e o arquitetural (Kahuna). O simulador usado e modificado para incorporar as mudanças descritas neste trabalho é o simulador arquitetural. O Kahuna modela detalhadamente os componentes arquiteturais descritos neste capítulo, incluindo, o *pipeline* do *PE*, a hierarquia de memória e a rede de interconexão. No caso da modelagem do *StoreBuffer*, embora em [59] conste a informação de que o simulador modela este componente exatamente como descrito na Seção 2.4.4, no Capítulo 3 são apresentadas algumas diferenças entre o modelo teórico e a sua implementação.

Os passos necessários para rodar um programa em um dos simuladores do WaveScalar são exibidos na Figura 2.12 e descritos a seguir:

1. O código C é compilado usando um compilador que siga as convenções de chamada da máquina Alpha, como o compilador *CC* para o *Tru64*;
2. O binário Alpha é transformado em *assembly* para WaveScalar pelo tradutor binário;

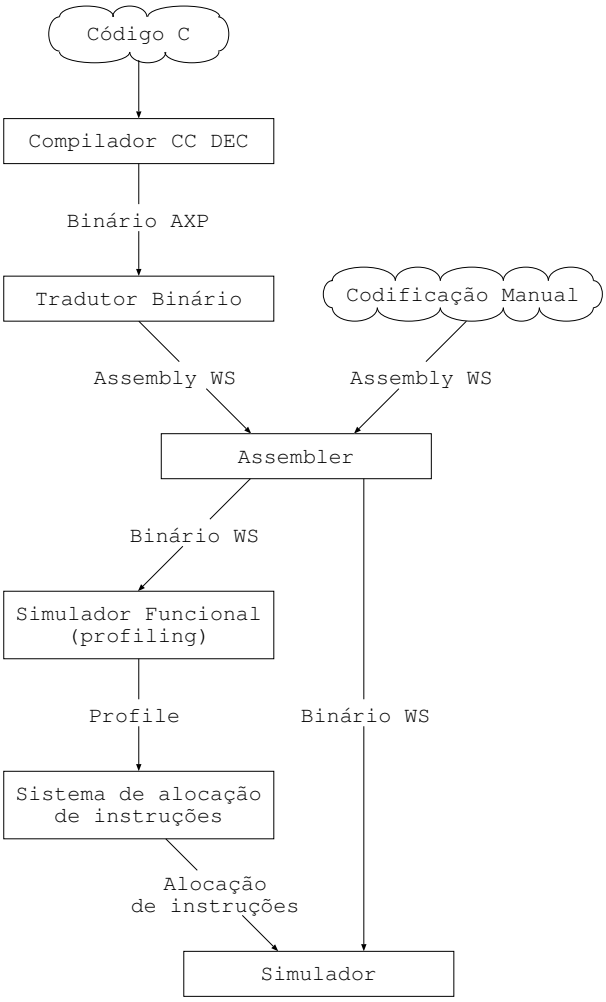


Figura 2.12: Passos para a execução de um programa no simulador do WaveScalar [59].

3. O *assembly* gerado pelo tradutor, juntamente com possível código *assembly* manualmente criado, são transformados em código binário para WaveScalar pelo Assembler;

4. O código é analisado pelo simulador funcional para extrair informações de *profiling* para ser usado pelo sistema de alocação de instruções;

5. O código finalmente é executado pelo simulador.

Os simuladores do WaveScalar possuem um simulador Alpha embutido, para executar as funções que possuem instruções que ainda não podem ser manipuladas pelo tradutor binário. Quando um programa é executado pelo Surfer ou Kahuna, o simulador da máquina Alpha entra em ação e o controle só é passado para o WaveScalar quando a função definida como ponto de entrada é encontrada. A partir daí, todas as funções são executadas no próprio simulador do WaveScalar, exceto àquelas marcadas para execução na máquina Alpha. Para informar o simulador a função que será o ponto de entrada no WaveScalar, bem como em qual dos simuladores (Alpha ou WaveScalar) cada função deverá ser executada, uma lista das funções com suas marcações deve ser salva em um arquivo com o mesmo nome do executável do WaveScalar, porém com a extensão *.compile*.

2.5.1 Otimizações no tradutor binário

No WaveScalar original uma onda $\boldsymbol{X}$ só recebe operandos proveniente da onda $\boldsymbol{X-1}$ e envia todos os operandos produzidos para a onda $\boldsymbol{X+1}$. Instruções de *Wave-Advance* só incrementam o número da onda em 1 unidade na *tag* do operando para o qual o *WA* é executado. Isto faz com que uma grande quantidade de instruções *WA* e *Steer* sejam executadas desnecessariamente.

A arquitetura do WaveScalar foi modificada para permitir a execução de instruções *WA* com incremento variado. Além disso, toda instrução pode fazer o incremento no número da onda, permitindo que ainda mais intruções de *WA* sejam eliminadas. O tradutor binário foi alterado para incluir uma melhoria, chamada de

otimização W que permite gerar código com instruções *WA* com incremento superior a 1, causando uma redução no número de instruções e um aumento de paralelismo no modelo de execução.

Na Figura 2.13, o programa (a) possui um operando C que é produzido antes do *loop FOR* e utilizado depois do mesmo. Quando compilado sem a otimização W o grafo *dataflow* (b) é gerado para o programa. O grafo mostra que, o número da onda na *tag* do operando deve ser incrementado 11 vezes para C atingir o seu destino. Este incremento é feito por 11 execuções de instruções *WA*. Além disso, a instrução *Steer*, de controle do *loop*, deve ser executada 10 vezes para que C não saia do *loop* antes do tempo e com o número da onda incorreto. Quando o programa é compilado usando a otimização W o grafo (c) é gerado. Nele, a instrução *WA* recebe também o valor do incremento, fazendo com que o operando C não passe pelas ondas intermediária, sendo entregue diretamente ao seu destino.

2.6 Informações adicionais do WaveScalar

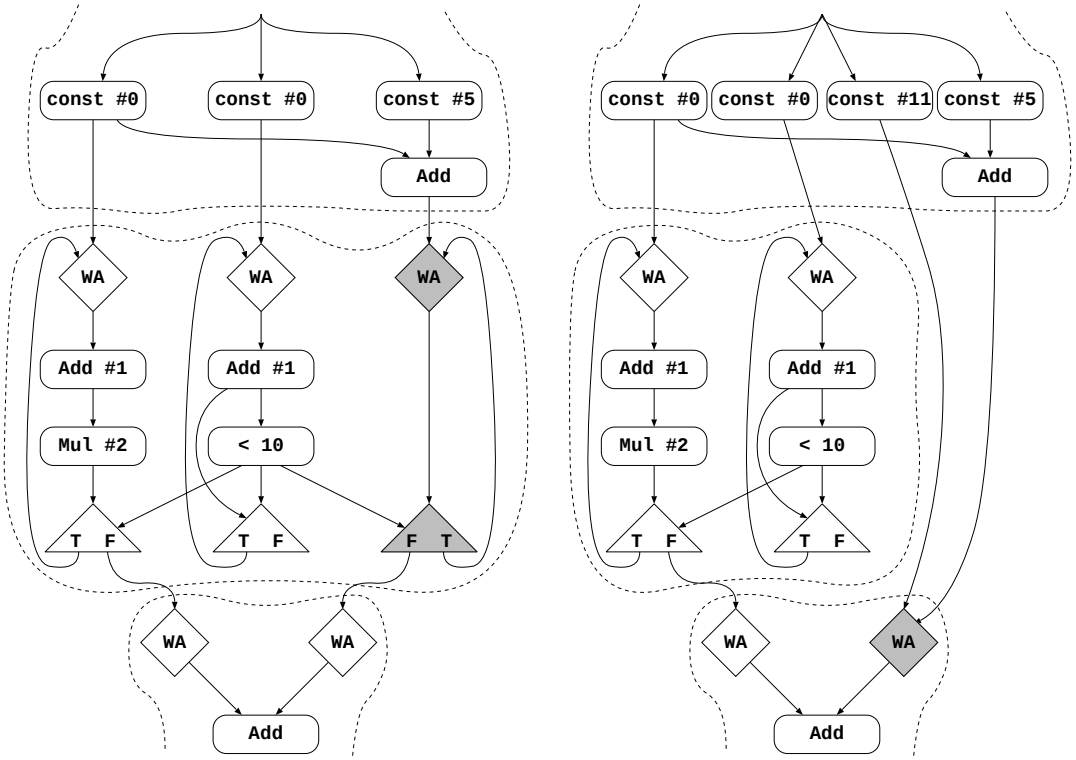
Neste capítulo foram apenas apresentados os conceitos e mecanismos do WaveScalar essenciais para este trabalho. Outras funcionalidades existentes são apresentadas superficialmente nesta seção, estando o material completo para um estudo mais aprofundado do assunto, disponível em [59, 57, 58]. As funcionalidas adicionais e informações interessantes são as seguintes:

Suporte a Múltiplas *threads*: Para implementar esta funcionalidade foi adicionado o *ThreadID* ao *tag* dos operandos (que possuía apenas o número da onda) e instruções foram criadas para manusear o número da onda e da *thread*, separadamente. O mecanismo de ordenação foi alterado para levar em conta também o número da *thread* (e não só o número da onda e a chave de ordenação) e mecanismos de sincronização de *threads* foram criados;

Interface de memória sem ordenação: O mecanismo de ordenação de memória do WaveScalar é necessário para rodar programas criados usando linguagens

```
a=0;
b=5;
c=a+b;
for (i=0; i<10; i++)
{
   a=(a+1)*2;
}
d=a+c;
```

(a)



(b) (c)

Figura 2.13: Grafo de um programa (a) compilado sem a otimização W (b) e com a otimização W (c).

iterativas, mas ele introduz um limite ao paralelismo. Com o intuito de fornecer outras alternativas aos programadores, o WaveScalar também disponibiliza uma outra interface de memória que não impõe a ordenação expressa pela seqüência em que as operações de memória aparecem no código. É uma interface semelhante às usadas em outras arquiteturas *Dataflow*, onde as dependências entre as operações de memória e as demais instruções precisam ser explicitadas com arestas inseridas no código. Para isso, foi criada uma nova operação de *Store* que retorna um valor *dummy* para apenas permitir expressar a dependência entre um *Store* e as instruções que o sucedem.

Embora o esquema propicie extrair todo o paralelismo permitido pela natureza do programa, todo o trabalho de definir as dependências entre as operações de memória fica a cargo do programador que utilizará a linguagem *assembly*, ou informará ao compilador as informações necessárias para que o mesmo construa o grafo *dataflow* de maneira correta;

Explosão de Paralelismo: Segundo [59], ocorre quando uma parte do programa (como a computação do índice de um *loop*) executa a frente do restante do programa, produzindo um grande número de operandos que não serão consumidos por um bom tempo. Isto pode causar o *overflow* das *matching tables*, degradando o desempenho, visto que elas serão despejadas na memória. Uma técnica conhecida como *k-loop bounding* [9] é utilizada para garantir que no máximo $\boldsymbol{k}$ iterações possam estar executando simultâneamente;

Análise área x desempenho: Em [57] é apresentado um estudo que determinou o número ideal de *Clusters*, *Domains* por *Cluster* e *PEs* por *Domain*, em função do desempenho e área utilizada no *chip*.

Capítulo 3

Transactional WaveCache

O mecanismo de ordenação das operações de memória do WaveScalar (*wave-ordered memory*), descrito no Capítulo 2, possibilitou a execução de programas escritos em linguagens imperativas, retirando a responsabilidade do programador de definir as dependências entre as operações de memória e, conseqüentemente, a ordem em que as mesmas deveriam ser executadas. Tal mecanismo impõe uma ordem total na execução destas operações, e é um forte ponto de serialização da arquitetura, pois nem todas as operações de memória são dependentes entre si e poderiam ser executadas em qualquer ordem.

Para ilustrar esta situação, na Figura 3.1 são dados dois trechos de código em C (a) e (b), cada um deles seguido de um grafo que mostra a relação entre as operações de memória para cada iteração do *loop* (ondas do WaveScalar). É possível observar que no trecho (a), em cada iteração do *loop* são feitos acessos à endereços de memória diferentes dos utilizados nas demais iterações, sendo portanto as operações de memória de uma iteração independentes das demais, podendo então ser executadas fora de ordem. A única operação que possui dependências dentro de uma iteração é o *Store*, que depende dos dois *Loads* subjacentes. No trecho (b), além do *Store* de cada iteração depender dos *Loads* subjacentes, a operação de *Load* do elemento $\boldsymbol{I + 1}$ do vetor $\boldsymbol{C}$ é dependente do *Store* que ocorre na iteração anterior.

Existem, portanto, situações em que a ordenação das operações de memória não precisa ser totalmente respeitada para que o programa continue produzindo resul-

tados corretos. Alterações que permitam um relaxamento no modelo de ordenação podem aumentar o grau de paralelismo do sistema para estes casos. Alternativas de relaxamento do modelo, que permitem a execução fora de ordem de operações dentro de uma onda, foram apresentadas no Capítulo 2 (Seção 2.3.3). Tais alternativas apresentam ganhos de paralelismo de 0 a 47% [59].

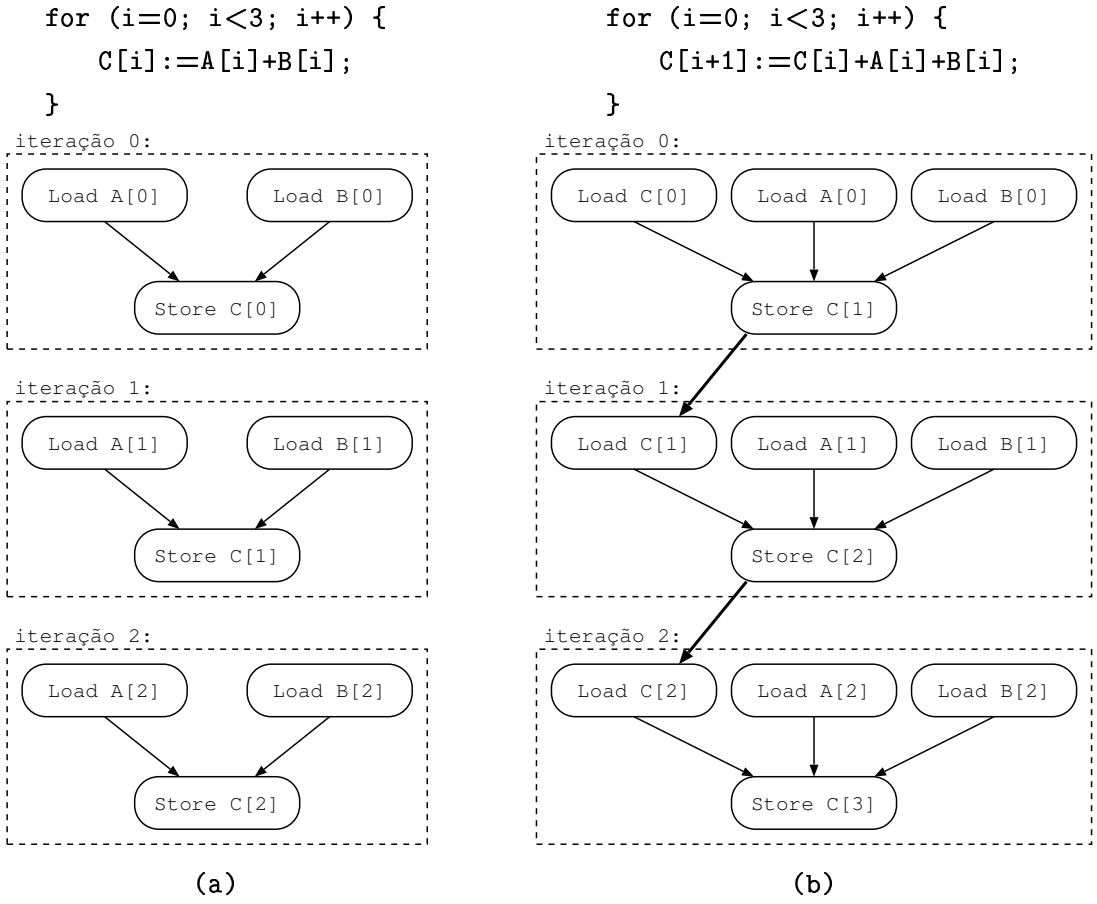


Figura 3.1: Exemplos de programas com ondas independentes (a) e dependentes (b).

Neste trabalho é apresentada uma modificação do modelo de ordenação de operações de memória que mantém a ordem parcial para cada onda, mas propõe o relaxamento da ordem total, baseada nos princípios de Memória Transacional. Esta modificação permite que operações de ondas diferentes executem fora de ordem, de forma especulativa. Este mecanismo, denominado Transactional WaveCache, bem como suas semelhanças e diferenças em relação a mecanismos de Memória Transacional já existentes para máquinas de Von Neumann, são descritos em detalhes ao

decorrer desse capítulo.

3.1 Transações no WaveScalar

Como descrito no Capítulo 1, para o uso da técnica de Memória Transacional, trechos de códigos são definidos como regiões atômicas pelo programador. As operações de memória contidas nestas regiões são tratadas como transações, que só podem ser completadas (*commit*) caso nenhuma posição de memória utilizada pela mesma tenha sido acessada por instruções fora dos limites da transação, desde seu início até o seu término. Caso contrário, a transação é cancelada e todos as alterações na memória são restauradas (*rollback*). Na Transactional WaveCache, as ondas são automaticamente consideradas como regiões atômicas, sendo a ordenação das operações de memória dentro de uma onda garantida pelo mecanismo de ordenação já existente. Ao definir transações como ondas, é possível permitir a execução das ondas especulativamente e fora-de-ordem na memória. Caso algum conflito (*hazard*) seja detectado, a transação é cancelada.

Para manter a informação do estado de cada transação é associado a cada onda um *bit* $\boldsymbol{T}$ que indica se a onda terminou suas operações de memória. O *bit* em questão é armazenado no mapa de ondas descrito no Capítulo 2. Além disso, existe em cada *StoreBuffer* um atributo chamado de *lastCommitedWave* que indica o número da última onda que sofreu um *commit*. A relação do *bit* $\boldsymbol{T}$ e do atributo *lastCommitedWave* com os possíveis estados para as transações são dadas a seguir:

1. Uma onda $\boldsymbol{X}$ pode ser considerada como não especulativa quando todas as ondas anteriores a ela tiverem o seu Estado Transacional *commited*, ou seja, $\boldsymbol{X}$ é especulativa se $\boldsymbol{X} > lastCommitedWave+1$, e $\boldsymbol{X}$ é uma onda não-especulativa, se $\boldsymbol{X} = lastCommitedWave+1$;

2. O valor inicial de *lastCommitedWave* para todo *StoreBuffer* é -1, pois a primeira onda de um programa tem número 0 e ela é não especulativa;

3. Operações de memória que pertencem a uma onda especulativa são operações

especulativas;

4. Inicialmente, todas as onda possuem $\boldsymbol{T}$ =0;

5. Quando todas as operações de memória associadas a uma onda já foram executadas, o *bit* $\boldsymbol{T}$ da onda em questão é ativado ($\boldsymbol{T}$ =1);

6. Uma onda $\boldsymbol{X}$ está pendente caso $\boldsymbol{X}$ >*lastCommitedWave*+1 e $\boldsymbol{T}$ =1; uma onda pendente está pronta para se tornar *commited* assim que a onda anterior também o for, porém, caso seja verificado algum *hazard* antes disso, ela poderá sofrer o *rollback* como qualquer onda especulativa;

7. Quando ($\boldsymbol{T}$ =1 e $\boldsymbol{X}$ =*lastCommitedWave*+1), uma onda $\boldsymbol{X}$ pode ser considerada como *commited*.

As preocupações com a virtualização de espaço e de tempo não são importantes para este modelo, pois sempre haverá uma transação não especulativa em execução. Transações muito grandes ou que demorem para terminar, em algum momento se tornarão não especulativas e poderão executar até o fim, pois não serão mais conflitantes com transações anteriores. Além disso, as estruturas de dados utilizadas para armazenar dados de transações não serão utilizadas por ondas não especulativas.

O número $\boldsymbol{A}$ da primeira operação de memória de uma onda é sempre 1, exceto para aquelas cujas primeiras operações de memória estão em ramos de um desvio ($\boldsymbol{A}$ ="?"). Para permitir a execução das ondas fora-de-ordem, o mecanismo de ordenação entre ondas foi alterado para operações de memória de qualquer onda possam ser executadas, começando pela operação que possua $\boldsymbol{A}$ =1 sem a necessidade de aguardar o término da onda anterior. Caso a primeira operação de uma onda $\boldsymbol{X}$ esteja em um ramo de um desvio, suas operações de memória não poderão ser executadas até que seja recebida a requisição da operação de *CWA* da onda $\boldsymbol{X-1}$ pelo *StoreBuffer* responsável. Quando isto ocorrer, uma mensagem será enviada ao *StoreBuffer* de $\boldsymbol{X}$, para que o mesmo adote a operação que possua número $\boldsymbol{C}$ igual ao número $\boldsymbol{P}$ do *CWA* como primeira operação de memória desta onda.

3.2 O contexto da especulação

Em mecanismos de Memória Transacional para máquinas de Von Neumann é necessário salvar o conteúdo do banco de registradores (ou a porção do banco utilizada pelo programa) antes do início da transação, além do histórico das modificações feitas na memória, durante a execução da mesma. Caso algum *hazard* ocorra, a memória e o banco de registradores são restaurados e a transação é reexecutada.

No WaveScalar não existe banco de registradores. Os operandos são produzidos por *PEs*, colocados em suas respectivas filas de saída e enviados a outros *PEs*, que os armazenam em suas *matching tables*. Além disso, quando uma instrução é executada, seus operandos de entrada são consumidos e se perdem. Para permitir a reexecução de uma transação é preciso identificar todos os operandos utilizados por suas instruções, que foram produzidos por instruções externas. O conjunto destes operandos é chamado de **contexto de entrada de uma transação** ou **contexto de entrada de uma onda**, visto que na Transactional WaveCache, cada onda é uma transação.

No WaveScalar, as instruções de *Wave-Advance* têm como objetivo receber um operando produzido por uma instrução de uma onda $\boldsymbol{X-1}$, incrementar o número da onda da *tag* do operando e encaminhá-lo para as instruções de destino na onda $\boldsymbol{X}$. Sendo assim, o contexto de entrada de uma onda é formado pelos operandos produzidos por todas as instruções de *Wave-Advance* associadas a ela. Não é possível saber, com antecedência, quantas instruções de *Wave-Advance* serão executadas para compôr o contexto de entrada da onda. Uma porção dela pode iniciar sua execução sem a necessidade do restante (caso o fluxo de dados assim permita) e um *hazard* pode ser causado antes que o contexto de entrada tenha sido completamente salvo.

De fato, não é necessário que o contexto de entrada completo seja salvo para permitir a reexecução de uma onda. O único motivo para armazenar os operandos de entrada de uma onda é que os mesmos são consumidos na execução especulativa, sendo necessária uma cópia para ser utilizada no caso de uma reexecução. Porém, para a parte da onda que ainda não iniciou a sua execução, isto não é um pro-

blema, pois seus operandos ainda serão produzidos pela onda anterior. Caso uma reexecução seja necessária, a porção do contexto que foi salva será utilizada e o restante, quando produzido pela onda anterior, será introduzido já na nova execução. Quando um *hazard* acontece, os operandos do contexto de entrada são reenviados às instruções destino dos *WAs* correspondentes, causando a reexecução das mesmas e, conseqüentemente, a reexecução de toda a onda e as subseqüentes. No entanto, durante a reexecução, podem haver operandos da execução anterior nas *matching tables* de alguns *PEs*. Tais operandos devem ser ignorados, para que não se misturem com operandos da execução atual, causando resultados indesejados.

Na Figura 3.2 é exibido o grafo *dataflow* referente a uma uma onda $\boldsymbol{X}$ com dois pontos de entrada e seus operandos A e B associados. O operando B é usado como endereço para um *Load* e o operando A e o resultado do *Load* são usados por uma operação qualquer (ANY). Observa-se a seguinte seqüência na execução das instruções da onda em questão, caracterizando o problema mencionado no parágrafo anterior:

Instante 1: O operando é produzido por uma instrução pertencente a onda $\boldsymbol{X-1}$;

Instante 2: A instrução *WA* é executada e o número da onda do operando B é atualizado para $\boldsymbol{X}$. O valor de B é salvo como parte do contexto de entrada da onda $\boldsymbol{X}$;

Instantes 3 a 5: A instrução de *Load* é executada;

Instante 6: Uma instrução de *Store* é executada em uma onda inferior a $\boldsymbol{X}$, para o mesmo endereço usado pelo *Load* em $\boldsymbol{X}$. O operando B é restaurado do contexto de entrada da onda $\boldsymbol{X}$ e enviado a instrução de *Load* (destino de *WA*) que inicia sua reexecução;

Instante 7: O operando A é produzido por uma instrução pertencente a onda $\boldsymbol{X-1}$. O *Load* ainda não terminou sua reexecução;

Instante 8: A instrução *WA* é executada com o operando A, que é adicionado ao contexto de entrada da onda $\boldsymbol{X}$. O *Load* ainda não terminou sua reexecução;

Instante 9: A instrução *ANY* é executada usando os operandos A e o valor retornado pelo *Load* da execução anterior, produzindo o operando C. O novo *Load* termina sua reexecução produzindo L';

A instrução *ANY* foi executada com operandos A e L, gerando assim um resultado incorreto. Para solucionar este problema é necessário que os operandos levem em seus *tags* o número da (re)execução a que pertencem dentro da onda e a regra de disparo deve ser modificada, para que permita que apenas operandos com o mesmo número de execução causem o disparo de uma instrução. Na Figura 3.3, a situação da Figura 3.2 é repetida, desta vez com o número da (re)execução (NExe) presente na *tag* dos operandos. Observa-se a seguinte seqüência na execução das instruções da onda em questão:

Instante 1: O operando é produzido por uma instrução pertencente a onda $\boldsymbol{X-1}$;

Instante 2: A instrução *WA* é executada para o operando B, incrementando o número da onda em sua *tag*. O valor de B é salvo como parte do contexto de entrada da onda $\boldsymbol{X}$. Como esta é a primeira execução desta onda, o número de execução (NExe) associado é 0 e, portanto, o operando B carrega o número 0 no campo NExe de seu *tag*;

Instantes 3 a 5: A instrução de *Load* é executada, produzindo o operando L.X.0 (onda=X e NExe=0);

Instante 6: Uma instrução de *Store* é executada em uma onda inferior a $\boldsymbol{X}$, para o mesmo endereço usado pelo *Load* em $\boldsymbol{X}$, caracterizando um *hazard*. O número da execução associado a onda $\boldsymbol{X}$ é incrementado. Todos os operandos de entrada que atingirem a onda à partir deste instante ou operandos restaurados do contexto de entrada salvo, levarão o número 1 no campo NExe de suas *tags*. O operando B é restaurado do contexto de entrada da onda $\boldsymbol{X}$ e enviado a instrução de *Load* (destino de *WA*) que inicia sua reexecução;

Instante 7: O operando A é produzido por uma instrução pertencente a onda X-1. O *Load* ainda não terminou sua reexecução;

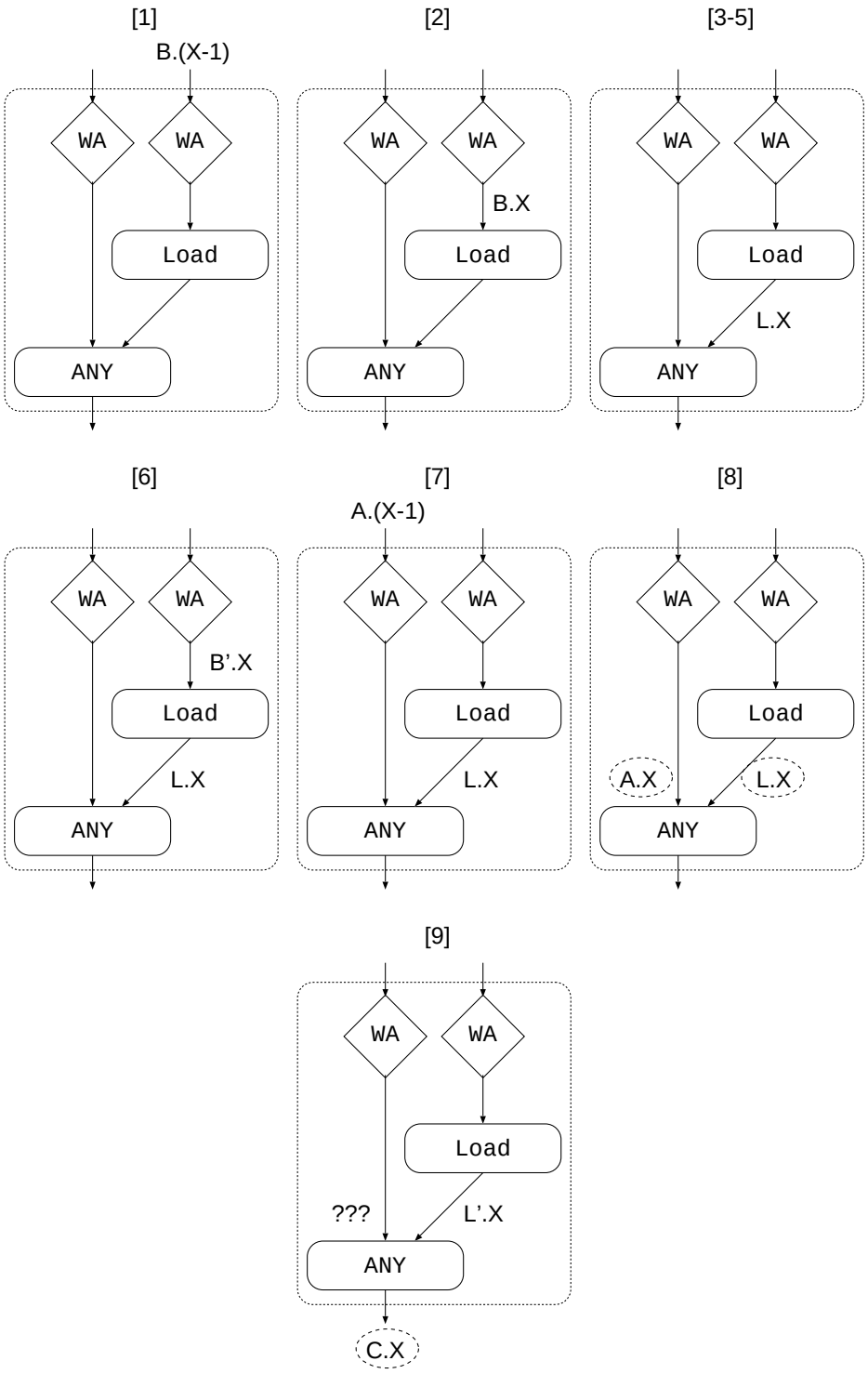


Figura 3.2: Problema com operandos de (re)execuções diferentes.

Instante 8: A instrução *WA* é executada com o operando $\boldsymbol{A}$, que é adicionado ao contexto de entrada da onda $\boldsymbol{X}$. O *Load* ainda não terminou sua reexecução. A operação *ANY* já possui dois operandos de entrada, porém ela não é executada, pois os operandos são provenientes de execuções diferentes (o operando $\boldsymbol{A}$ da execução 1 e o operando $\boldsymbol{L}$ da execução 0);

Instante 9: O *Load* termina sua execução, fazendo com que o operando L.X.0 (da execução 0) seja substituído pelo operando L.X.1 (da execução 1);

Instante 10: *ANY* é executada com os operando corretos, produzindo C.

O Número da execução (CurrentNExe) atual de uma onda deve ser conhecido pelo *StoreBuffer* responsável pela onda. Sendo assim, o NExe é adicionado ao mapa de ondas, e, quando uma reexecução é causada por uma onda $\boldsymbol{X}$, todas as ondas $\geq$X recebem um novo currentNExe que não pode ser igual ao das execuções anteriores. Por isso, o *StoreBuffer* possui um atributo chamado *lastNExe* que armazena o último número de execução utilizado. Este número é incrementado a cada reexecução e repassado ao currentNExe de cada onda reexecutada.

Para que um operando de uma execução antiga, não seja repassado para outras ondas, assumindo o NExe atual da onda destino e sendo consumido incorretamente, os *StoreBuffer*, só aceitarão operandos enviados por instruções de *WA* para uma onda onda $\boldsymbol{X}$ quando $\text{NExe}_{\boldsymbol{operando}} = \text{NExe}_{\boldsymbol{X-1}}$. Para os operandos aceitos, se for verificado $\text{NExe}_{\boldsymbol{X}} > \text{NExe}_{\boldsymbol{X-1}}$, significa que a onda $\boldsymbol{X}$ foi reexecuta e o operando em questão acaba de ser enviado pela onda X-1 e ainda não foi armazenado no contexto de entrada. Neste caso, o operando recebido deve ter seu NExe atualizado para $\text{NExe}_{\boldsymbol{X}}$ e deve ser imediatamente reenviado para instrução *WA* correspondente. Esta técnica, não só mantém o modelo coerente, como também elimina operandos de execuções antigas que estariam consumindo recursos desnecessariamente. Cada onda ao ser incluída no mapa de ondas assume o número currentNExe da onda anterior. No caso de uma reexecução, a onda onde foi encontrado o conflito e todas as posteriores terão seu NExe atualizado para currentNExe+1.

O contexto de entrada de uma onda é armazenado em uma estrutura chamada de

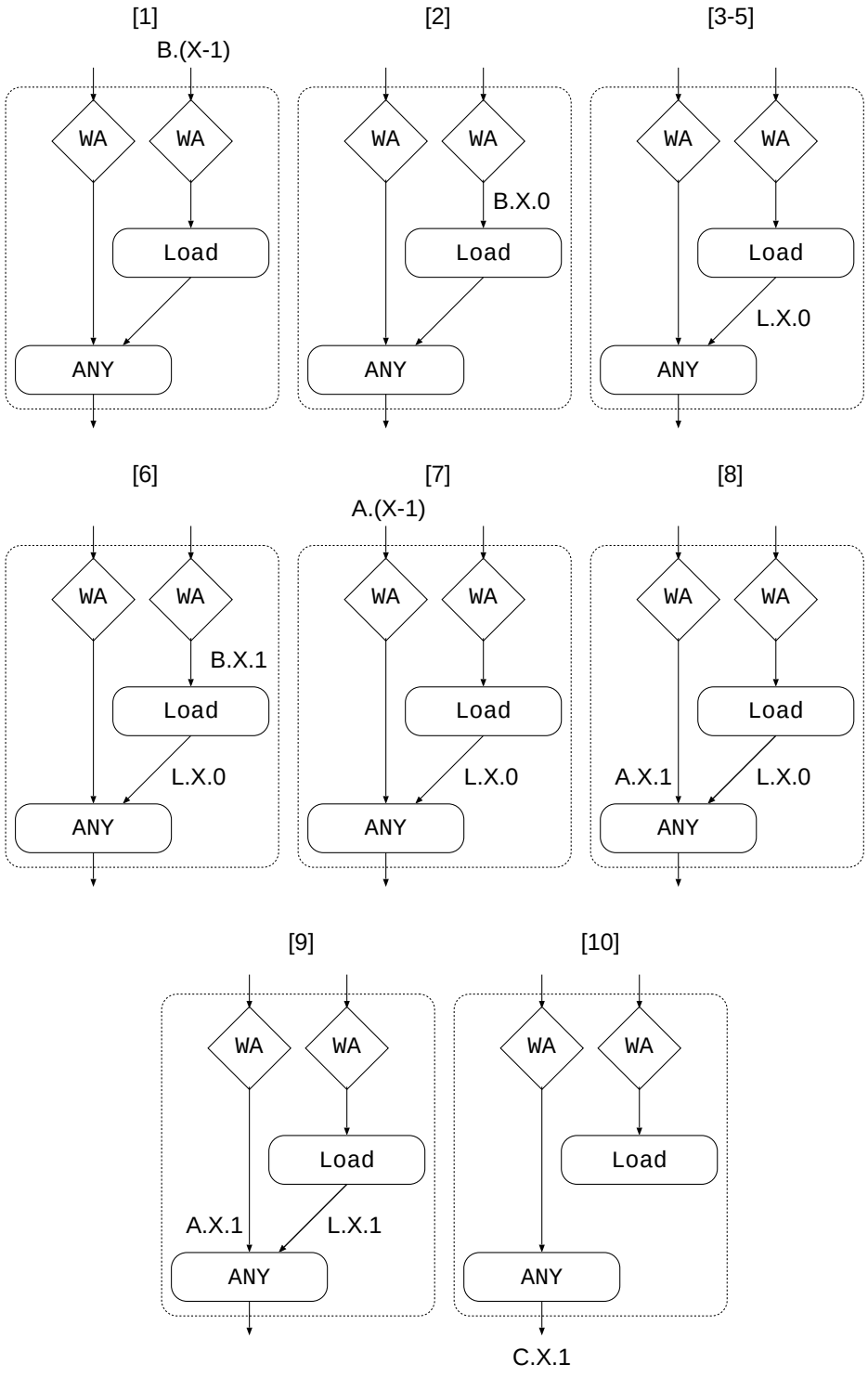


Figura 3.3: Solução: Identificar operandos com o número da execução.

Wave-Context table localizada no *StoreBuffer* responsável pela onda. Como visto no Capítulo 2, cada *StoreBuffer* só pode receber solicitações de memória de até quatro ondas. Acompanhando este limite arquitetural, o limite de quatro *Wave-Context tables* em cada *StoreBuffer* permite que até quatro ondas especulativas sejam executadas. Cada *Wave-Context table* possui um número identificador. Para saber qual das *Wave-Context tables* está associada a uma onda, no mapa de ondas é adicionado o número identificador da *Wave-Context table* associada a cada onda. O número de linhas de uma *Wave-Context table* determina o número máximo de operandos de entrada que pode existir em uma onda especulativa. Caso uma onda possua mais operandos de entrada do que suporta uma *Wave-Context table*, assim que a mesma for totalmente ocupada, os operandos restantes não serão entregues a onda até que ela se torne uma onda não especulativa, quando o contexto de entrada não é mais necessário.

No simulador, o limite de *next/ordering tables* não foi implementado, permitindo o recebimento de solicitações de qualquer onda. Para efeitos de simulação este limite arquitetural do número de *Wave-Context tables*, bem como o número de linhas de cada *Wave-Context table* não são considerados. Estudos futuros são necessários para determinar o número ideal de *Wave-Context tables* por *StoreBuffer* e o número de linhas em cada *Wave-Context table*. No Capítulo 5 (Seção 5.2.5) este assunto é discutido mais amplamente.

A estrutura com as *Wave-Context tables* é exibida na Figura 3.4. Ela armazena para toda instrução *WA* o operando (valor e *tag*) e os endereços das instruções que receberão o operando na onda seguinte

Em esquemas de Memória Transacional para máquinas de Von Neumann, a própria *cache* do processador pode ser usada para armazenar o histórico das alterações na memória, pois apenas transações de um processo ou *thread* pode estar executando por vez em um processador. Nestes esquemas, o protocolo de coerência da *cache* é modificado para incluir *bits* de estado transacional em cada linha. No WaveScalar existe uma *cache* por *Cluster* e o mesmo pode executar diferentes *threads* simultaneamente. Portanto, a *cache* não pode ser utilizada com esta finalidade a não ser

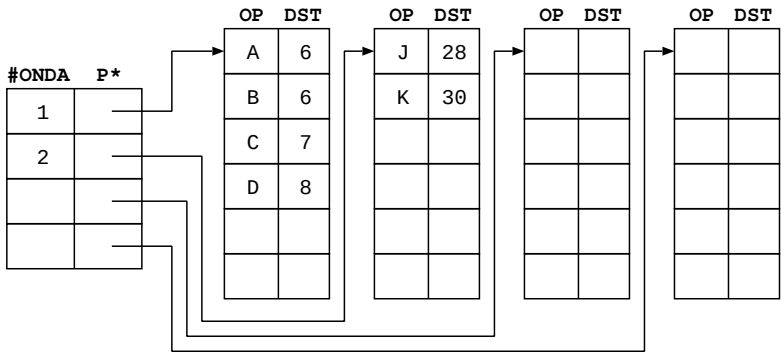


Figura 3.4: A estrutura *Wave-Context Tables*.

que o mecanismo de *placement* garanta que diferentes *threads* não sejam alocadas a *PEs* de um mesmo *Cluster*. Esta alternativa não é adotada neste trabalho.

Uma estrutura onde seja possível identificar alterações e acessos realizados para cada transação é definida, levando em conta que diversas ondas poderão estar executando especulativamente. Sendo assim, ao atender uma requisição de acesso à memória pertencente a uma dessas ondas, deve ser possível verificar a lista de operações executadas especulativamente para o mesmo endereço. As operações em questão podem pertencer a ondas diferentes, portanto, é necessário que todas as ondas especulativas tenham uma visão única deste histórico.

Como visto no Capítulo 2, em cada *Cluster* existe um *StoreBuffer* e cada um possui a custódia de algumas ondas. Desta forma, para que todas as ondas possuam a mesma visão do histórico, basta que os *StoreBuffers* o tenham. Existem duas alternativas para este problema: armazenar o histórico na memória, sendo a visão única garantida pelo protocolo de coerência de *cache* ou armazenar o histórico em cada *StoreBuffer* e utilizar um algoritmo de difusão de informação para que todos os *StoreBuffer* possuam sempre a mesma visão do histórico. Ambas as alternativas apresentam graves problemas: na primeira, cada acesso especulativo à memória resultará em dois acessos, um para a operação propriamente dita e outro para atualizar o histórico; na segunda ocorre um aumento significativo na comunicação *inter-cluster*.

Conforme pode ser visto no Capítulo 2 (Seção 2.4.4), devido ao esquema de

placement tender a alocar todas as instruções de uma *Thread* para o mesmo *Cluster*, a maior parte das ondas de uma *Thread* é alocada ao mesmo *Cluster*. Com base nessa premissa, a alternativa de armazenar o histórico de operações de memória localmente em cada *StoreBuffer* é modificada para minimizar o impacto na comunicação *inter-cluster*, como descrito a seguir:

1. Cada *StoreBuffer* possui uma estrutura para armazenar o histórico de operações de memória, chamada de *MemOp History*;

2. Cada *StoreBuffer* possui um atributo chamado NUOI, que armazena o número da última onda iniciada;

3. O atributo NUOI de cada *StoreBuffer* é inicializado com 0 (zero) no início da execução do programa;

4. Um *StoreBuffer* só pode executar operações de memória de uma onda ou operações *CWA*, se o número da onda em questão for menor ou igual ao seu NUOI;

5. Toda operação *CWA* executada, incrementará NUOI, caso o número da onda da operação *CWA* for igual a NUOI;

6. Como visto na Seção 3.4, quando uma onda recebe o *commit*, se o *StoreBuffer* responsável pela próxima onda é diferente do *StoreBuffer* da onda atual (o que raramente ocorre), uma mensagem deve ser enviada para informá-lo do número da onda que sofreu o *commit* para que o mesmo possa marcar a próxima onda como não especulativa. Ao receber esta mensagem, o *StoreBuffer* remoto incrementará este número e o armazenará em NUOI.

O esquema descrito permite execução especulativa apenas de ondas contíguas associadas a um *StoreBuffer*, fazendo com que o subsistema de memória aguarde o término de um trecho contíguo (associado a um *StoreBuffer*) para o início do próximo (em outro *StoreBuffer*). Esta espera também ocorre no modelo de execução, visto que todas as operações de *Wave-Advance* passam agora pelo subsistema de memória.

Embora tal solução insira um ponto de sincronismo no modelo, como ondas em uma mesma *thread* raramente estarão associadas a *StoreBuffers* diferentes, poucos momentos de sincronização deverão ocorrer durante a execução do programa. Além disso, devido ao problema da explosão de paralelismo descrito no Capítulo 2, a técnica de *k-loop bounding* já limita o número de ondas que podem estar executando, e, conseqüentemente, este limite também é passado para a memória. Desta forma, a solução provida não impõe uma redução no paralelismo muito maior do que a já existente.

No simulador original do WaveScalar, as filas de requisições de memória não são implementadas de forma distribuída entre os *StoreBuffers*. Como todas as operações de uma *thread* tendem a ser alocadas ao mesmo *Cluster*, uma estrutura centralizada foi utilizada para guardar as filas de requisição de acesso à memória para facilitar a implementação. Afinal, poucas mensagens serão trocadas entre os *StoreBuffers* e o custo associado a mesma pode ser desconsiderado. Na implementação da Transactional WaveCache feita neste trabalho, a mesma política é adotada e não foi necessária a implementação do mecanismo do NUOI, embora neste texto seja descrita a solução completa (com o mecanismo do NUOI).

Para toda operação especulativa a ser executada é necessário varrer o histórico de operações para o mesmo endereço em busca de possíveis violações. Portanto, é necessário armazená-lo, agrupando as operações por endereço. Em cada *StoreBuffer*, a estrutura que armazena o histórico é composta por uma lista que contém os endereços usados por operações especulativas e ponteiros para tabelas, uma para cada endereço usado em especulações, com as seguintes informações para cada uma das operações referentes ao endereço em questão:

#ONDA: Número da onda associada a operação;

#OP: Número da operação (número C da chave de ordenação);

TIPO: Tipo da operação;

VALOR: Valor gravado para operações de *Store* ou carregado para operações de

Load;

BKP: Valor armazenado no endereço em questão antes da execução da operação (*backup*), usado apenas para operações de *Store*.

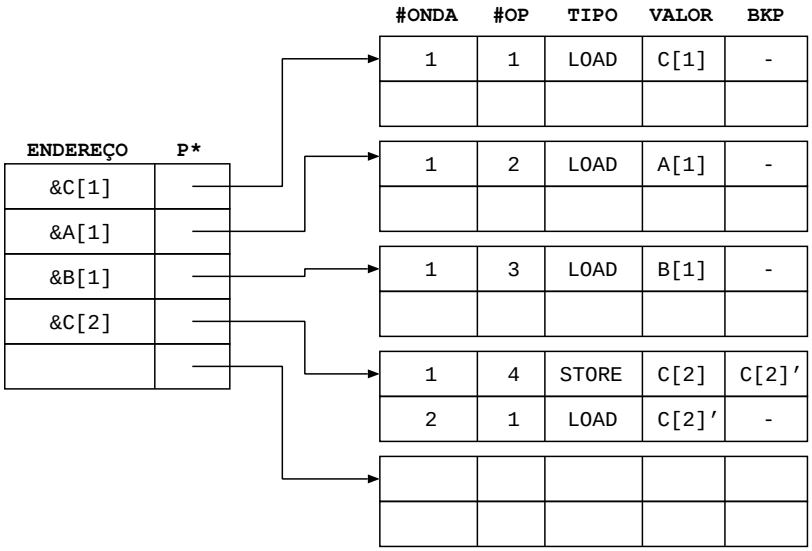


Figura 3.5: A estrutura *MemOp History*.

Considerando que para o programa exibido na Figura 3.1 (b), a onda de número 1 (iteração 1) e o início da onda de número 2 (iteração 2), tenham sido executados especulativamente, antes das operações de memória da onda de número 0 (iteração 0), é possível observar na Figura 3.5 a estrutura *MemOp History* associada. As tabelas e linhas vazias são exibidas, pois o número máximo de cada tipo de tabela, bem como seus tamanhos, são fixos e definidos na implementação (limite arquitetural). No simulador, este limite não é implementado, sendo necessários estudos futuros para determinar o tamanho ideal dessa estrutura, bem como o impacto de sua limitação na Transactional WaveCache. No Capítulo 5 (Seção 5.2.5) este assunto é discutido mais amplamente.

3.3 Detectando uma especulação incorreta

Sempre que houver dependências entre operações de ondas distintas, violações podem ocorrer ao executar tais operações especulativamente. As dependências são

determinadas quando ambas as operações são feitas sob um mesmo endereço ou quando o resultado de um *Load* é usado para compor o endereço ou dado (no caso de um *Store*) de uma operação de memória pertencente a uma onda posterior. A detecção de dependências é feita da seguinte forma: para cada nova operação, verifica-se a existência de alguma operação especulativa no mesmo endereço na estrutura *MemOp History*. Caso houver e for caracterizado um *hazard* entre as operações são tomadas as providências cabíveis.

Como o esquema de ordenação de memória apresentado neste trabalho permite apenas o relaxamento da ordenações entre ondas, operações provenientes de uma mesma onda não apresentarão *hazards*, pois serão sempre executadas em ordem. Portanto, é necessário apenas verificar a existência de *hazards* entre a operação corrente e as operações de ondas posteriores. Para descrever os possíveis *hazards* e suas soluções, são dadas duas operações de memória A e B, pertencentes as ondas ***X*** e Y, respectivamente, tal que X>Y (B ocorre antes de A segundo a ordem do programa). Sabendo que não existem operações de *Store* já executadas especulativamente entre elas e que A executou especulativamente antes de B, quando B finalmente executa, ocorrem as seguintes situações:

RAW (Read After Write): Ocorre quando A é um *Load* e B é um *Store*. Neste caso, a onda ***X*** e todas as ondas posteriores, já iniciadas, precisam ser reexecutadas (*rollback*). Este processo é descrito em detalhes na Seção 3.5. Se B é especulativa, é inserida na estrutura *MemOp History*, sendo o seu campo bkp, lido da memória, antes de fazer a modificação. Esta operação não precisa ser atômica, pois só haverá um *StoreBuffer* acessando a memória por vez;

WAW (Write After Write): Ocorre quando A e B são operações de *Store*. Neste caso, a onda ***X*** não precisa ser reexecutada. Isto ocorrerá apenas se o endereço em questão tiver sido utilizado por algum *Load* entre A e B (*RAW*). Se B for especulativa, é inserida na estrutura *MemOp History* e seu campo *bkp* recebe o valor do campo *bkp* da operação A. Independetemente de B ser ou não especulativa, o campo *bkp* de A é atualizado para o valor utilizado no *Store* B;

WAR (Write After Read): Ocorre quando A é um *Store* e B é um *Load*. Neste caso, a onda ***X*** não precisa ser reexecutada e o campo *bkp* do *Store* em questão será usado como retorno do *Load*. Se B é especulativa, é inserida na estrutura *MemOp History*.

Algoritmo 3.1 BuscaStore

Entrada: operação de memória B
Saída: operação de memória do tipo *Store* S

```
1: S ← null
2: para toda operação Op na MemOp History para o endereço de B faça
3:    Se (Op.tipo = Store) e (Op.onda > B.onda) então
4:       Se (S = null) ou (S.onda > Op.onda) ou ((S.onda = Op.onda) e (S.C > Op.C)) então
5:          S ← Op
6:       fim Se
7:    fim Se
8: fim para
9: return S
```

Algoritmo 3.2 BuscaLoad

Entrada: operação de memória B e operação *Store* S proveniente de BuscaStore
Saída: operação de memória do tipo *Load* L

```
1: L ← null
2: para toda operação Op na MemOp History para o endereço de B faça
3:    Se (Op.tipo = Load) e (Op.onda > B.onda) então
4:       Se (S = null) ou (Op.onda < S.onda) ou ((Op.onda = S.onda) e (Op.C < S.C)) então
5:          Se (L = null) ou (L.onda > Op.onda) ou ((L.onda = Op.onda) e (L.C > Op.C)) então
6:             L ← Op
7:          fim Se
8:       fim Se
9:    fim Se
10: fim para
11: return L
```

Os Algoritmos 3.4 e 3.5 descrevem os passos para detecção de *hazards* com as respectivas soluções. O Algoritmo 3.1 varre a tabela da estrutura *MemOp History* para o endereço de B em busca da primeira operação de *Store* S, cujo número da onda seja maior que Y. O Algoritmo 3.2 varre a tabela da estrutura *MemOp History* para o endereço de B em busca da primeira operação de *Load* L cujo número da onda seja maior que Y, contanto que S não esteja entre B e L.

Para preencher o campo bkp, no caso de um *Store*, caso não exista um *Store* de uma onda posterior na estrurura *MemOpHistory*, para o seu endereço (WAW),

Algoritmo 3.3 BuscaOpAnterior

Entrada: operação de memória B
Saída: operação de memória O

```
1: O ← null
2: para toda operação Op na MemOp History para o endereço de B faça
3:    Se ((Op.onda < B.onda) ou ((Op.onda = B.onda) e (Op.C < B.C))) então
4:       Se (O = null) ou (O.onda < Op.onda) ou ((O.onda = Op.onda) e (O.C < Op.C)) então
5:          O ← Op
6:       fim Se
7:    fim Se
8: fim para
9: return S
```

Algoritmo 3.4 DetectaHazardStore

Entrada: operação de *Store* B

```
1: S ← BuscaStore(B)
2: Se S = null então
3:    Se B é especulativa então
4:       O ← BuscaOpAnterior(B)
5:       Se O ≠ null então
6:          B.bkp ← O.valor
7:       senão
8:          B.bkp ← MEM[B.end]
9:       fim Se
10:      Insere B na estrutura MemOp History
11:   fim Se
12:   MEM[B.end] ← B.valor
13: senão
14:   {WAW}
15:   Se B é especulativa então
16:      B.bkp ← S.bkp
17:      Insere B na estrutura MemOp History
18:   fim Se
19:   S.bkp ← B.valor
20: fim Se
21: L ← BuscaLoad(B, S)
22: Se L ≠ null então
23:   {RAW}
24:   Reexecuta(L.onda)
25: fim Se
```

Algoritmo 3.5 DetectaHazardLoad

Entrada: operação de *Load* B

```
1: S ← BuscaStore(B)
2: Se S ≠ null então
3:    {WAR}
4:    B.valor ← S.bkp
5: senão
6:    B.valor ← MEM[B.end]
7: fim Se
8: Se B é especulativa então
9:    Insere B na estrutura MemOp History
10: fim Se
```

o campo *bkp* deve ser lido da memória. Para aplicações que não apresentam dependências do tipo *WAW* isto significa que um *Load* deve ser executado a cada *Store*. Para evitar este carregamento de valores para preencher o campo bkp, o Algoritmo 3.3 busca a operação de memória anterior a B. Se um *Load* ou *Store* já foi realizado anteriormente (para o mesmo endereço) e ainda se encontra na estrutura *MemOpHistory*, no Algoritmo 3.4 o campo valor desta operação é repassado para o campo *bkp* do *Store*.

O campo valor da estrutura *MemOpHistory* é usado apenas com esta finalidade e representa um aumento significativo no tamanho da estrutura (64 *bits* a mais por linha). Outra alternativa é remover este campo e usar o campo *bkp* para guardar o valor carregado em operações de *Load* e não armazenar valor para os *Stores*. Neste caso, o Algoritmo 3.3 deve ser modificado para buscar a operação de *Load* anterior, contanto que não exista nenhum *Store* entre ela e o *Store* que está sendo inserido na estrutura *MemOpHistory*. Caso esta operação seja encontrada, um carregamento pode ser evitado para preencher o campo *bkp* dos *Stores* especulativos.

A melhor alternativa é ainda uma questão a ser respondida. Para a alternativa adotada neste trabalho há um maior gasto de espaço pela estrutura *MemOpHistory*, em troca de menos carregamentos feitos para cada *Store*. Nesta outra alternativa ocorrem mais carregamentos desnecessários e uma economia de espaço gasto pela estrutura em questão.

3.4 Completando uma Transação (*commit*)

O *commit* é feito automáticamente pela Transactional WaveCache, sempre que uma onda não especulativa termina a sua última operação de memória. A ocorrência do *commit* em uma onda de número $\boldsymbol{X}$, faz com que a onda de número $\boldsymbol{X + 1}$ se torne uma onda não especulativa. Caso a onda $\boldsymbol{X + 1}$ tenha terminado todas as suas operações de memória, i.e., possua estado *finished*, ela também será completada (*commited*) e assim sucessivamente. Quando uma onda deixa de ser especulativa, não é mais necessário armazenar o seu histórico de operações de memória e nem seu contexto de entrada.

Como a estrutura *MemOp History* é indexada pelo endereço, para apagar tais operações seria necessário fazer uma busca seqüencial em toda a estrutura para remover as linhas referentes a operações da onda em questão. Dependendo do número de operações contidas nesta estrutura, esta operação pode ser custosa. Para tornar mais fácil a tarefa de identificar as linhas da estrutura *MemOp History* associadas a uma onda, é necessário uma tabela para cada onda, que associe o número C de cada operação com o seu endereço. Tais tabelas são localizadas nos *StoreBuffers*, junto à estrutura em questão. Também é necessário uma outra tabela para relacionar cada tabela de operações com sua sua respectiva onda. Esta estrutura é denominada *Search Catalog*.

Na prática, a tabela de ondas da estrutura *Search Catalog* pode ser mesclada com a tabela de ondas da estrutura *Wave-Context tables*, para economizar espaço. Para efeitos de modelagem, as duas estruturas são consideradas individualmente, sem esta otimização.

Considerando a estrutura *MemOp History* exibida na Figura 3.5, é possível observar na Figura 3.6 a estrutura *Search Catalog* associada. As tabelas e linhas vazias são exibidas pois o número máximo de cada tipo de tabela, bem como seus tamanhos, são fixos e definidos na implementação (limite arquitetural). No simulador, este limite não é implementado, sendo necessário um estudo para determinar o tamanho ideal dessa estrutura, bem como o impacto de sua limitação na Transactional

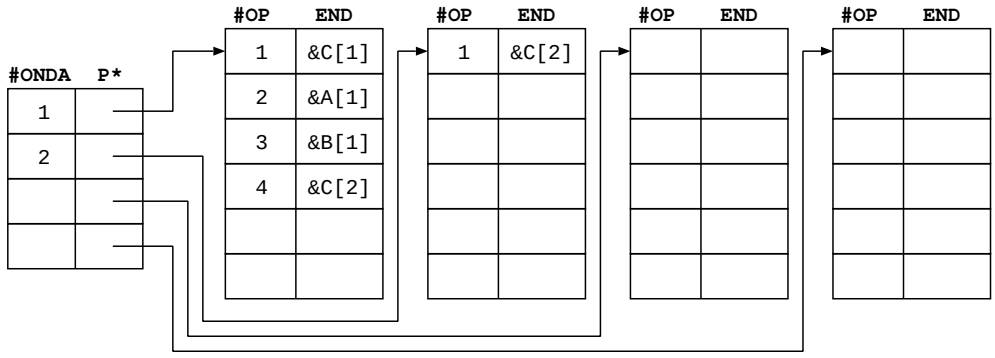


Figura 3.6: A estrutura *Search Catalog*.

WaveCache. No Capítulo 5 (Seção 5.2.5) este assunto é discutido mais amplamente.

No WaveScalar, mediante a terminação de uma onda, um algoritmo é executado para remover a entrada referente àquela onda do mapa de ondas/ *StoreBuffers*. Para tirar proveito da possível reação em cadeia decorrente de um *commit*, foram feitas modificações no algoritmo para realizar o *commit*, caso a onda que terminou suas operações de memória seja não especulativa. Depois disso, o *commit* é propagado para as ondas seguintes que já tiverem sido terminadas. O Algoritmo 3.6 descreve estas ações. A propagação do *commit* para ondas sob custódia de outros *StoreBuffers* é feita pelo envio de uma mensagem *Commited(X, lastNExe)*, causando a execução do Algoritmo 3.7 no *StoreBuffer* destino. Como na Transactional WaveCache as listas de requisições de acesso à memória não foram implementadas de forma distribuída entre os *StoreBuffers*, a propagação do *commit* é feita sem o uso das mensagens aqui descritas.

3.5 Reexecutando uma Transação (*rollback*)

Como visto na Seção 3.3, quando um *hazard* do tipo *RAW* é detectado para uma operação de *Store*, a reexecução da onda relacionada com a operação em questão e de todas as ondas posteriores é solicitada. De acordo com as definições da estrutura *MemOp History*, apresentadas na Seção 3.2, apenas um *StoreBuffer* por vez, estará executando operações de memória, pois, dadas duas ondas ***X*** e X+1 em *StoreBuffers*

Algoritmo 3.6 PropagaCommit

Entrada: onda X

1: $T_{\boldsymbol{X}} \leftarrow 1$ {Todas as operações de memória da onda $\boldsymbol{X}$ terminaram}
2: {Se a onda $\boldsymbol{X}$ não é especulativa}
3: **Se** X=*lastCommitedWave*+1 **então**
4: *lastCommitedWave* ← *lastCommitedWave*+1 {Ela é *commited*}
5: Remove $\boldsymbol{X}$ do mapa de ondas/*StoreBuffers*
6: **para** toda operação Op do *Search Catalog* da onda X **faça**
7: Remove Op da estrutura *MemOp History*
8: **Se** a tabela relacionada com Op.end da estrutura *MemOp History* estiver vazia **então**
9: Remove a linha com endereço Op.end da tabela de endereços de *MemOp History*
10: **fim Se**
11: **fim para**
12: Remove *Search Catalog* da onda X
13: Remove *Wave-Context table* da onda X
14: {Se a onda seguinte terminou suas operações de memória}
15: **Se** $T_{\boldsymbol{X+1}}$ =1 **então**
16: **Se** *StoreBuffer* da onda $\boldsymbol{X}$ = *StoreBuffer* da onda X+1 **então**
17: PropagaCommit(X+1)
18: **senão**
19: Envia mensagem Commited(X) para o *StoreBuffer* da onda X+1
20: **fim Se**
21: **fim Se**
22: **fim Se**

Algoritmo 3.7 RecebeCommit

Entrada: onda X

1: *lastCommitedWave* ← X
2: NUOI ← NUOI + 1
3: PropagaCommit(X+1)

distintos, operações da onda X+1 só poderão ser executadas quando $\boldsymbol{X}$ receber o *commit*. Porém, os demais *StoreBuffers* estarão armazenando o contexto de entrada das operações de *WA* que foram executadas. Caso seja necessária uma reexecução no *StoreBuffer* que possui a onda especulativa, a mesma será realizada, e, quando os operandos da nova execução atingirem os outros *StoreBuffers*, eles irão apagar seus contextos de entrada para que sejam refeitos, pois o NExe dos operandos atuais é maior que o dos anteriores. O processo de reexecução de uma onda consiste basicamente nos seguintes passos:

1. Excluir operações de ondas a partir de $\boldsymbol{X}$ das tabelas associadas na estrutura *Search Catalog*. O Algoritmo 3.9 detalha este processo;

2. Restaurar o conteúdo da memória para o ponto anterior a execução da primeira operação de memória de $\boldsymbol{X}$. Isto é feito restaurando o campo *bkp* das operações de *Store* S de cada endereço na estrutura *MemOp History*, contanto que S.onda$\geq$X e S seja a primeira operação de *Store* a partir de $\boldsymbol{X}$ para o endereço em questão. Aproveitando que, para restaurar o conteúdo da memória, já é necessário varrer a estrutura *MemOp History*, a exclusão das operações de ondas a partir de $\boldsymbol{X}$ (inclusive) dessa estrutura é feita em conjunto com o processo de restauração. O Algoritmo 3.10 detalha este processo;

3. Limpar as *Wave-Context tables* das ondas posteriores a onda $\boldsymbol{X}$ (exclusive). O Algoritmo 3.12 detalha este processo;

4. Excluir todas as requisições de acesso à memória do *StoreBuffer* local. O Algoritmo 3.11 detalha este processo. Requisições em *StoreBuffer* remotos são excluídas quando os mesmos receberem requisições da nova reexecução, sem a necessidade de mensagens adicionais. Na implementação da Transactional WaveCache feita neste trabalho, a exclusão de requisições é feita na estrutura central que armazena todas as requisições;

5. Incrementar o *lastNExe* no *StoreBuffer* responsável pela onda que causou a reexecução e repassar este valor para *currentNExe* de todas as ondas a partir de $\boldsymbol{X}$ (inclusive). Como apenas um *StoreBuffer* executa operações por vez, a

primeira onda P posterior a $\boldsymbol{X}$ a ser executada fora do *Cluster* ficará paralisada na execução das instruções de *WA* até que a onda P-1 sofra um *commit*. Portanto, todas as ondas presentes no mapa de ondas a partir de $\boldsymbol{X}$, estarão associadas ao *StoreBuffer* local, com exceção da onda P (se P existir). Logo, o tráfego de coerência de memória para manter o mapa de ondas atualizado continua a ser pequeno, pois raramente um *StoreBuffer* fará atualizações em linhas no mapa relacionadas a outro *StoreBuffer*. Além disso, uma reexecução não tende a ocorrer com frequência, logo, a atualização dos campos NExe do mapa de ondas também ocorrerá raramente. O Algoritmo 3.13 detalha este processo;

6. Excluir todas as requisições de acesso à memória das filas de operações (existe uma por onda) de todas as ondas superiores a $\boldsymbol{X}$ (inclusive), no *StoreBuffer* responsável. As filas das ondas sob custódias de outros *StoreBuffers* serão limpas, mediante o recebimento de operações de memória com NExe superior ao das presentes na mesma;

7. Restaura o apontador da primeira operação de memória de todas as ondas superiores a $\boldsymbol{X}$, para buscar a operações cujo número A seja igual a 1. Como o *CWA* da onda X-1 já foi executado e o operando produzido é encaminhado para a onda $\boldsymbol{X}$ (*CWAs* fazem também o papel de *WAs*), é necessário procurar este operando na WaveContextTable na onda $\boldsymbol{X}$ e atualizar o apontador da primeira operação de memória. Este processo só precisa ser feito no *StoreBuffer* associado a $\boldsymbol{X}$, ele é o único que está executando operações de memória, estando os ponteiros em questão intactos para as ondas de outros *StoreBuffers*. Este processo é feito em conjunto com a restauração do contexto de entrada da onda, causando a reexecução das instruções destino dos *Wave-Advance* e *Canonical-Wave-Advance* para os quais o contexto foi restaurado. O Algoritmo 3.14 detalha esta tarefa.

Algoritmo 3.8 Reexecuta

Entrada: onda X

```
1: LimpaSearchCatalog(X)
2: LimpaOperacoes(X)
3: RestauraMem(X)
4: LimpaWaveContextTables(X)
5: NExe ← IncrementaNExe(X)
6: RestauraWaveContext(X, NExe)
```

Algoritmo 3.9 LimpaSearchCatalog

Entrada: onda X

```
1: para toda linha L preenchida na lista de ondas da estrutura Search Catalog faça
2:   Se L.onda ≥ X então
3:     para toda operação Op em L.P faça
4:       Remove Op de L.P
5:     fim para
6:     Remove L da lista de ondas da estrutura Search Catalog
7:   fim Se
8: fim para
```

3.6 Evitando a Explosão de Paralelismo

Quando um conflito do tipo *RAW* é detectado, é feita a reexecução de diversas instruções usando um novo NExe. Embora requisições de acesso à memória provenientes de execuções antigas não sejam aceitas pelo subsistema de memória, não existe nenhum impedimento para que instruções com dados de execuções antigas sejam executadas, contanto que todos os operandos disponíveis possuam o mesmo NExe. Para evitar que *tokens* de execuções antigas sejam produzidos e transmitidos desnecessariamente, contribuindo com o problema da explosão de paralelismo e aumentando o número de instruções executadas para uma aplicação, dois mecanismo foram criados e são discutidos nesta seção.

3.6.1 Eliminação de operandos das *Matching Tables*

A etapa de casamento de operandos dos *Processing Elements* é modificada para que no recebimento de um operando com NExe para uma instrução A, operandos presentes para A na *Matching Table* com NExe menor que o NExe do operando recebido sejam eliminados. Caso o operando recebido possua NExe menor que os

Algoritmo 3.10 RestauraMem

Entrada: onda X

```
1: para cada linha L preenchida na lista de endereços da estrutura MemOp History faça
2:    TotOp ← 0
3:    TotDel ← 0
4:    S ← null
5:    para toda operação Op de L.P faça
6:       Se Op.onda ≥ X então
7:          Se Op.tipo = Store então
8:             Se S = null então
9:                S ← Op
10:            senão
11:               Se (S.onda > Op.onda) ou ((S.onda = Op.onda) e (S.C > Op.C)) então
12:                  S ← Op
13:               fim Se
14:            fim Se
15:         fim Se
16:         Remove Op de L.P
17:         TotDel ← TotDel + 1
18:      fim Se
19:      TotOp ← TotOp + 1
20:   fim para
21:   Se S ≠ null então
22:      MEM[L.end] ← S.bkp
23:   fim Se
24:   Se TotOp = TotDel então
25:      {Lista vazia}
26:      Remove L da lista de endereços da estrutura MemOp History
27:   fim Se
28: fim para
```

Algoritmo 3.11 LimpaOperacoes

Entrada: onda X

```
1: para toda fila F requisições para o StoreBuffer de X faça
2:    Se F.onda ≥ X então
3:       para toda operação Op em F faça
4:          Remove Op de F
5:       fim para
6:       Remove F
7:    fim Se
8: fim para
```

Algoritmo 3.12 LimpaWaveContextTables

Entrada: onda X

```
1: para toda linha L preenchida na lista de ondas da estrutura Wave-Context tables faça
2:    Se L.onda > X então
3:       para todo operando Op em L.P faça
4:          Remove Op de L.P
5:       fim para
6:       Remove L da lista de ondas da estrutura Wave-Context tables
7:    fim Se
8: fim para
```

Algoritmo 3.13 IncrementaNExe

Entrada: onda X

Saída: número da Execução NExe atualizado para a onda X

```
1: para toda linha L do mapa de ondas faça
2:    currentNExe ← currentNExe + 1
3:    Se L.onda ≥ X então
4:       L.NExe ← current.NExe
5:       {A primeira operação de memória tem o número 1 (padrão) ...}
6:       L.PrimeiraOp ← 1
7:       {no número A de sua cadeia}
8:       L.OlharPara ← A
9:       {o bit T é reiniciado (a onda não terminou)}
10:      L.T ← 0
11:   fim Se
12: fim para
13: return currentNExe
```

Algoritmo 3.14 RestauraWaveContext

Entrada: onda ***X*** e número da Execução NExe atualizado para a onda X

```
1: para toda linha L preenchida na lista de ondas da estrutura Wave-Context tables faça
2:    Se L.onda = X então
3:       para toda linha O em L.P faça
4:          Se O.Op é um CWA então
5:             {A primeira operação de memória de X é indicada pelo número P do CWA da onda
               anterior (faz parte do contexto de entrada de X)}
6:             {A primeira operação de memória tem o número P do CWA da onda anterior...}
7:             L.PrimeiraOp ← O.Op.Cadeia.P
8:             {no número C de sua cadeia}
9:             L.OlharPara ← C
10:         fim Se
11:         O.Op.Tag.NExe ← NExe
12:         Envia(O.Op, O.dst)
13:      fim para
14:      sai do loop (break)
15:   fim Se
16: fim para
```

presentes na *Matching Table* para A, ele não será aceito. Este mecanismo causa a remoção de *tokens* antigos, quando os operandos da reexecução nova atingem os *PEs*

3.6.2 Mapas de Execução

Para evitar o recebimento de operandos de execuções antigas que chegam a um *PE* que ainda não recebeu nenhum outro operando para a mesma instrução, é possível fazer uma relação com os demais operandos já recebidos sem a necessidade de troca de informação entre os *PEs*. Cada *Processing Element* mantém localmente uma estrutura que relaciona um NExe a cada faixa de ondas, baseado nos operandos já recebidos. Esta estrutura é chamada de Mapa de Execução. Os Algoritmos 3.18, 3.15, 3.16 e 3.17 descrevem o comportamento do mecanismo. A Figura 3.7 mostra a evolução do Mapa de execução mediante o recebimento de uma seqüência de operandos. A coluna O da estrutura representa o número da onda e a coluna N o número da execução (NExe). Cada uma das situações exibidas na figura são comentadas a seguir.

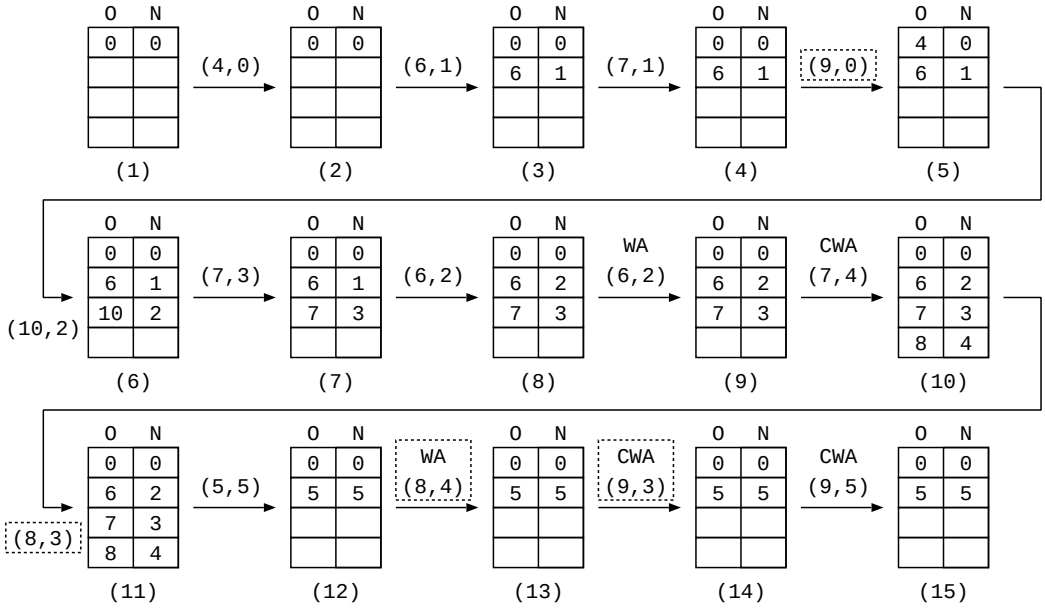


Figura 3.7: O uso do Mapa de Execução.

1. Inicialmente, o *PE* não recebeu nenhum operando e em seu Mapa de Execução consta apenas o mapeamento da Onda 0 para o NExe 0 (padrão), indicando que para todas as ondas o NExe é 0;

2. Um operando da onda 4 e NExe 0 é recebido. O Mapa de Execução não é atualizado e o operando é aceito;

3. O operando da onda 6 e NExe 1 é recebido. O Mapa de Execução é atualizado para indicar que para todas as ondas acima de 6 (inclusive) o NExe associado é 1 e o operando é aceito;

4. O operando da onda 7 e NExe 1 é recebido. O Mapa de Execução não é atualizado e o operando é aceito;

5. O operando da onda 9 e NExe 0 é recebido. O Mapa de Execução não é atualizado e o operando não é aceito, pois para ondas $\geq$ 6 o NExe deve ser 1;

6. O operando da onda 10 e NExe 2 é recebido. O Mapa de Execução é atualizado para indicar que para todas as ondas acima de 10 (inclusive) o NExe associado é 2 e o operando é aceito;

7. O operando da onda 7 e NExe 3 é recebido. O Mapa de Execução é atualizado para indicar que para todas as ondas acima de 7 (inclusive) o NExe associado é 3 e o operando é aceito;

8. O operando da onda 6 e NExe 2 é recebido. O Mapa de Execução é atualizado para indicar que para a onda 6 o NExe associado é 2 e o operando é aceito;

9. O operando da onda 6 e NExe 2 é recebido para uma instrução de *WA*. O Mapa de Execução não é atualizado e o operando é aceito;

10. O operando da onda 7 e NExe 4 é recebido para uma instrução de *CWA*. O Mapa de Execução é atualizado para indicar que para todas as ondas $\geq$ de 8 (a onda 8 é o destino do operando de saída para a instrução *CWA*) o NExe associado é 4 e o operando é aceito;

11. O operando da onda 8 e NExe 3 é recebido. O Mapa de Execução não é atualizado e o operando não é aceito;

12. O operando da onda 5 e NExe 5 é recebido. O Mapa de Execução é atualizado para indicar que para todas as ondas ≥ 5 o NExe associado é 5 e o operando é aceito;

13. O operando da onda 8 e NExe 4 é recebido para uma instrução de *WA*. O Mapa de Execução não é atualizado e o operando não é aceito;

14. O operando da onda 9 e NExe 3 é recebido para uma instrução de *CWA*. O Mapa de Execução não é atualizado e o operando não é aceito;

15. O operando da onda 9 e NExe 5 é recebido para uma instrução de *CWA*. O Mapa de Execução não é atualizado e o operando é aceito.

O número de linhas do Mapa de Execução influi na eficiência do mecanismo para remover operandos. Em aplicações que apresentem reexecuções em diversos pontos, o número de linhas pode não ser suficiente para representar a relação entre todas as ondas e seus números de execução, fazendo com que operandos de execuções antigas sejam aceitos por um *PE*. Neste trabalho, este limite não foi implementado, sendo necessário um estudo para definir o tamanho ideal desta estrutura bem como a determinação da relação entre a perda de eficiência do mecanismo e o limite imposto para a estrutura. No Capítulo 5 (Seção 5.2.5) este assunto é discutido mais amplamente.

3.7 Variando a especulação do sistema

O simulador foi alterado para permitir a escolha da quantidade de especulação desejada, ou grau de especulação do sitema. Ao selecionar grau de especulação 5, por exemplo, é criada uma janela de execução que permite que apenas operações de memória da onda não-especulativa e das 4 ondas especulativas posteriores sejam atendidas pelo subsitema de memória. A intenção deste mecanismo é limitar a

Algoritmo 3.15 ValidaOperandoWA

Entrada: Operando Op

```
1: {Operandos de instruções de WA ou CWA são destinados a onda seguinte}
2: ondaDestino ← Op.onda+1
3: Se existe a linha L no mapa de execução tal que L.onda = ondaDestino então
4:    Se (Op.NExe > L.NExe) então
5:       L.NExe = Op.NExe
6:       LimpaMapaDeExecucao(ondaDestino, Op.NExe)
7:    fim Se
8: senão
9:    M ← BuscaLinha(ondaDestino)
10:   Se (Op.NExe > M.NExe) então
11:      M.NExe = Op.NExe
12:      LimpaMapaDeExecucao(ondaDestino, Op.NExe)
13:   fim Se
14: fim Se
```

Algoritmo 3.16 BuscaLinha

Entrada: Número da onda W

Saída: Linha encontrada M

```
1: M ← null
2: para toda linha L no mapa de execução tal que L.onda < W faça
3:    Se ((M = null) ou (L.onda > M.onda)) então
4:       M ← L
5:    fim Se
6: fim para
7: return M
```

Algoritmo 3.17 LimpaMapaDeExecucao

Entrada: Número da onda W e número da execução NExe

```
1: para toda linha L no mapa de execução faça
2:    Se ((L.onda > W) e (NExe ≥ L.NExe)) então
3:       Remove L
4:    fim Se
5: fim para
```

Algoritmo 3.18 ValidaOperandoRecebido

Entrada: Operando Op
Saída: Operando deve ser aceito? (valor booleano)

```
1: Se o mapa de execução está vazio então
2:    {A linha 0,0 é a primeira a entrar no mapa}
3:    Insere uma linha com número da onda 0 e NExe 0 no mapa
4: fim Se
5: aceito ← verdadeiro
6: Se Op.onda > 0 então
7:    {Se o operando é destinado à instruções de WA ou CWA}
8:    Se ((Op.dest.tipo= WA) ou (Op.dest.tipo= CWA)) então
9:       ValidaOperandoWA(Op)
10:   fim Se
11:   Se existe a linha L no mapa de execução tal que L.onda = Op.onda então
12:      Se ((Op.NExe > L.NExe) e (Op.dest.tipo≠ WA) e (Op.dest.tipo≠ CWA)) então
13:         L.NExe = Op.NExe
14:         LimpaMapaDeExecucao(Op.onda, Op.NExe)
15:      fim Se
16:      Se Op.NExe < L.NExe então
17:         aceito ← falso
18:      fim Se
19:   senão
20:      M ← BuscaLinha(Op.onda)
21:      Se ((Op.NExe > M.NExe) e (Op.dest.tipo≠ WA) e (Op.dest.tipo≠ CWA)) então
22:         M.NExe = Op.NExe
23:         LimpaMapaDeExecucao(Op.onda, Op.NExe)
24:      fim Se
25:      Se (Op.NExe < M.NExe) então
26:         aceito ← falso
27:      fim Se
28:   fim Se
29: fim Se
30: return aceito
```

especulação, e, conseqüentemente, diminuir a possibilidade de conflitos que causem reexecuções (RAW). Quanto maior o grau de especulação, maior a possibilidade de estarem presentes na janela de execução duas ondas distantes que apresentem dependências verdadeiras entre si. A existência de tais dependências pode ocasionar conflitos. Para cada onda completada, a janela é adiantada, permitindo a execução de outras ondas especulativas.

Na Figura 3.8 obseva-se a janela de execução (marcada em cinza) formada quando é usado grau de especulação 5. Em (1), observa-se a situação inicial onde requisições de acesso à memória que vão da onda não-especulativa 0 e das ondas especulativas de 1 a 4. Quando a onda 0 é completada (*commit*) em (2), a janela é deslocada para a direita indicando que a onda 1 é a não-epeculativa e que todos as requisições de 1 a 5 podem ser atendidas. Da mesma forma, em (3) e (4) é exibida a janela de execução para quando ocorre o *commit* das ondas 1 e 2, respectivamente.

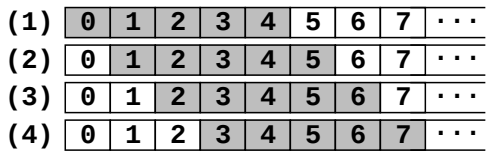


Figura 3.8: A janela de execução.

Capítulo 4

Experimentos e Resultados

4.1 Metodologia

4.1.1 Ambiente de Simulação

A fim de avaliar o mecanismo de reordenação de operações de memória proposto neste trabalho, experimentos na forma de simulação foram realizados utilizando o simulador arquitetural do WaveScalar, chamado Kahuna [59]. O simulador foi modificado, para incluir o mecanismo descrito no Capítulo 3. Os recursos computacionais utilizados para as simulações foram estações de trabalho IBM PC Pentium D com 1GB de memória e com sistema operacional Linux (Debian Sarge e Ubuntu).

O tradutor binário do WaveScalar exige que os programas tenham sido compilados para máquina Alpha. O compilador utilizado deve seguir as convenções de chamadas da máquina Alpha, o que não se aplica ao compilador gcc. Nos experimentos conduzidos em [56, 58, 59] foi utilizado o compilador cc e uma máquina Alpha com sistema operacional Tru64. Porém, como as aplicações utilizadas neste trabalho são simples e não usam chamadas de função, foi possível usar o *gcc alpha-linux cross compiler* para gerar código *assembly* da máquina Alpha à partir do código fonte em C, e depois fazer a conversão para *assembly* do WaveScalar usando o tradutor binário.

4.1.2 Aplicações

Foram criadas 7 aplicações artificiais para fazer uma avaliação inicial de desempenho. Com esta versão da implementação ainda não é possível fazer simulações com aplicações reais e, os motivos para tal são detalhados no Capítulo 5. Cada uma das 7 aplicações possuem duas versões, sendo uma com quantidade de trabalho realizada aproximadamente 10 vezes maior que a outra. A versão que realiza a menor quantidade de trabalho tem seu nome iniciado com a palavra *small*. A outra versão das aplicações são aquelas que não possuem em seus nomes a palavra *small*. No texto essas aplicações são chamadas de normais.

Seis aplicações utilizadas realizam basicamente a mesma tarefa: Cálculo de determinante de uma lista de matrizes de ordem 2, sendo 50 matrizes para a versão *small* e 500 para a versão normal. Além disso, é feita a soma de suas linhas resultando em vetores de 2 elementos. As diferenças existentes entre estas aplicações têm o propósito de alterar o grau de dependência entre as iterações dos laços existentes, além de variar a quantidade de concorrência que se pode extrair com a técnica proposta neste trabalho. A outra aplicação faz uma série de leituras e gravações em vetores de maneira a forçar o aparecimento de conflitos. Estes vetores possuem 50 elementos na versão *small* e 500 na versão normal. Na avaliação de desempenho é possível fazer uma relação destas situações com os ganhos ou perdas obtidos. As aplicações são as seguintes.

Toy-Matrices-SD e Small-Toy-Matrices-SD : O vetor de matrizes de ordem 2 é declarado, mas não é inicializado (os valores já presentes na memória são utilizados). Um laço calcula o determinante de cada matriz e armazena este resultado em um segundo vetor. Também dentro do laço um vetor de matrizes de 2x1 é preenchido com a soma das linhas de cada matriz de ordem 2;

Toy-Matrices-CD e Small-Toy-Matrices-CD: Fazem as mesmas tarefas das aplicações Toy-Matrices-SD e Small-Toy-Matrices-SD, porém, na iteração central (25 para a *small* e 250 para a normal) do laço de cálculos, o determinante calculado na iteração anterior é somado ao determinante calculado na iteração

corrente, para que seja armazenado no vetor de determinantes;

Toy-Matrices-SD-Stores e Small-Toy-Matrices-SD-Stores: Realizam as mesmas tarefas das aplicações Toy-Matrices-SD e Small-Toy-Matrices-SD, porém o vetor de matrizes é inicializado antes dos cálculos;

Toy-Matrices-CD-Stores e Small-Toy-Matrices-CD-Stores: Fazem as mesmas tarefas das aplicações Toy-Matrices-CD e Small-Toy-Matrices-CD, porém o vetor de matrizes é inicializado antes dos cálculos;

Toy-Matrices-SD-Stores-Min e Small-Toy-Matrices-SD-Stores-Min: Realizam as mesmas tarefas das aplicações Toy-Matrices-SD-Stores Small-Toy-Matrices-SD-Stores, porém o laço de cálculos é repetido 10 vezes;

Toy-Matrices-CD-Stores-Min e Small-Toy-Matrices-CD-Stores-Min: Fazem as mesmas tarefas das aplicações Toy-Matrices-CD-Stores e Small-Toy-Matrices-CD-Stores, porém o laço de cálculos é repetido 10 vezes;

Toy-Vectors-CD-WAR-WAW-RAW e Small-Toy-Vectors-CD-WAR-WAW-RAW: Fazem acessos e gravações em 5 vetores. Diversos conflitos do tipo *WAR* e *WAW* são introduzidos nesses acessos. Além disso, uma dependência que pode ocasionar um conflito *RAW* é inserida na aplicação.

Para as aplicações Toy-Matrices-SD-Stores, Small-Toy-Matrices-SD-Stores, Toy-Matrices-CD-Stores e Small-Toy-Matrices-CD-Stores no laço de inicialização ocorrem muitas operações de *Stores* intercaladas com poucas operações aritméticas. Operações de *Store* necessitam de dois operandos para que sejam disparadas: o endereço de acesso e o dado a ser gravado. Os endereços de acesso para estas aplicações são baseados na variável de controle do laço, cujo cálculo termina antes da execução do resto do corpo do *loop*. Sendo assim, o endereço de acesso geralmente está disponível antes do dado, fazendo com que uma série de *Stores* fiquem apenas aguardando o dado para que sejam disparados. Os dados dos diversos *Stores* (um de cada iteração) chegam em intervalos de tempo próximos, causando o disparo de diversos *Stores*, quase simultâneamente. A intenção é sobrecarregar o subsistema de

memória com muitas requisições resultando em alta concorrência já no WaveScalar original. Depois é feita a verificação se a Transactional WaveCache consegue aumentar a concorrência para o caso onde ela já é alta.

Para as aplicações Toy-Matrices-SD-Stores-Min, Small-Toy-Matrices-SD-Stores-Min, Toy-Matrices-CD-Stores-Min e Small-Toy-Matrices-CD-Stores-Min depois do laço de inicialização com alta concorrência nos acessos à memória, o laço de cálculos, que possui menor concorrência no WaveScalar original, é executado 10 vezes. A intenção é verificar o quanto a mais de concorrência é possível se extrair com o uso da Transactional WaveCache para estes casos.

4.1.3 Parâmetros Arquiteturais

A Tabela 4.1 descreve cada uma das configurações utilizadas nos experimentos. O número de *PEs* por *Domain* e o número de *Domains* por *Cluster* utilizado nos experimentos, é o que resulta na melhor relação área/desempenho, apontada como configuração ideal [57]. As demais configurações adotadas foram as configurações padrão do simulador arquitetural. A Tabela 4.2 exibe os principais parâmetros da arquitetura do WaveScalar utilizados nas simulações executadas neste trabalho. O ambiente de simulação utilizado provê otimizações no tradutor binário e no mecanismo de ordenação de memória que são opcionais. Os experimentos realizados variam estes parâmetros, bem como o grau de especulação da Transactional WaveCache permitido no sistema.

Execução	Descrição
SMT	Sem Memória Transacional e sem otimizações (WaveScalar original)
SMT DEC	Sem Memória Transacional e com *Decoupled Stores*
SMT RIP	Sem Memória Transacional e com Ripple Numbers
SMT DEC RIP	Sem Memória Transacional e com *Decoupled Stores* e Ripple Numbers
SMT W	Sem Memória Transacional e com otimizações no tradutor binário
SMT DEC W	Sem Memória Transacional e com *Decoupled Stores* e com otimizações no tradutor binário
SMT RIP W	Sem Memória Transacional e com Ripple Numbers e com otimizações no tradutor binário
SMT DEC RIP W	Sem Memória Transacional e com *Decoupled Stores* e Ripple Numbers e com otimizações no tradutor binário
CMT 2	Com Memória Transacional - 1 onda especulativa
CMT 3	Com Memória Transacional - 2 ondas especulativas
CMT 5	Com Memória Transacional - 4 ondas especulativas
CMT 10	Com Memória Transacional - 9 ondas especulativas
CMT 15	Com Memória Transacional - 14 ondas especulativas
CMT 20	Com Memória Transacional - 19 ondas especulativas
CMT 30	Com Memória Transacional - 29 ondas especulativas
CMT 50	Com Memória Transacional - 49 ondas especulativas
CMT 100	Com Memória Transacional - 99 ondas especulativas
CMT SLIM	Com Memória Transacional - sem limite de ondas especulativas

Tabela 4.1: Descrição dos tipos de execução

4.1.4 Métricas

As seguintes métricas foram utilizadas na realização dos estudos quantitativos:

Instruções: Número de instruções executadas;

Ciclos: Número de ciclos necessários para a execução;

IPC: Quantidades de instruções executadas por ciclo;

RAW: Número de perigos do tipo *Read After Write* detectados e resolvidos;

WAR: Número de perigos do tipo *Write After Read* detectados e resolvidos;

WAW: Número de perigos do tipo *Write After Write* detectados e resolvidos;

Speedup: Aceleração de desempenho obtida com o uso da Transactional WaveCache.

Número de *Clusters*	4 (2x2)
Domains por *Cluster*	4 (2x2)
Processing Elements pot *Domain*	8 (2x4)
Mecanismo de *Placement*	*Exp2* (dinâmico)
Cache L1	Mapeamento direto 1024 linhas de 32 bits Tempo de acesso: 1 ciclo
Cache L2	Associativa (4 vias) 131072 linhas de 128 bits Tempo de acesso: 7 ciclos
Tempo de acesso a memória	100 ciclos
Processing Elements	Tamanho das filas de operandos: 10000000 Tempo de busca nas filas: 0 Execuções por ciclo: 1 Operandos de entrada aceitos por instrução para cada ciclo: 3 Número de instruções: 8
StoreBuffers	4 portas de entrada 4 portas de saída

Tabela 4.2: Parâmetros arquiteturais utilizados na simulação

4.1.5 Validação do mecanimo: corretude dos experimentos

Para verificar a corretude dos programas executados utilizando a Transactional WaveCache, o simulador foi modificado para listar, ao final da simulação, todos os endereços de memória acessados por operações de *Store*, seguidos de seus respectivos valores. Para todos os experimentos, esta lista foi comparada com a gerada pelo simulador original (sem a Transactional WaveCache). Todos os programas executados, exibidos neste trabalho, apresentam os mesmos resultados na memória que os executados no simulador do WaveScalar original, que atende todas as requisições de acesso a memória na ordem estabelecida pelo programa.

4.2 Resultados

A Tabela 4.3 mostra as métricas coletadas para a aplicação Toy-Matrices-SD. Como esperado, não ocorrem perigos do tipo *RAW* e portanto, não são necessárias reexecuções. O mesmo ocorre para a aplicação Small-Toy-Matrices-SD, cujos dados

são exibidos na Tabela 4.4. As métricas coletadas para a aplicação Toy-Matrices-CD são exibidas na Tabela 4.5 e as da aplicação Small-Toy-Matrices-CD na Tabela 4.6. Nestes casos, o perigo do tipo *RAW* é detectado para todas as simulações que utilizam a Transactional WaveCache ocasionando uma reexecução.

Na Figura 4.1 é exibido um gráfico comparativo com as acelerações obtidas para todos os casos de simulação das aplicações Toy-Matrices-SD, Small-Toy-Matrices-SD, Toy-Matrices-CD e Small-Toy-Matrices-SD. Nota-se que a medida que o grau de especulação do sistema é aumentado, a aceleração também aumenta. Isto indica que o aumento no grau de especulação do sistema possibilita o aumento da concorrência no acesso ao subsistema de memória. Com isto, instruções dependentes de resultados de *Loads* são liberadas para execução mais rapidamente, causando um aumento no paralelismo no sistema de execução, e, conseqüentemente, um aumento no desempenho.

Execução	Instruções	Ciclos	IPC	Perigos Detectados			Speedup
				RAW	*WAR*	*WAW*	
SMT	28570	16070	1,78	0	0	0	1
SMT DEC	28570	7406	3,86	0	0	0	2,17
SMT RIP	28570	16070	1,78	0	0	0	1
SMT DEC RIP	28570	7394	3,86	0	0	0	2,17
SMT W	23057	14991	1,54	0	0	0	1,07
SMT DEC W	23057	5763	4	0	0	0	2,79
SMT RIP W	23057	14997	1,54	0	0	0	1,07
SMT DEC RIP W	23057	5764	4	0	0	0	2,79
CMT 2	28570	13863	2,06	0	0	0	1,16
CMT 3	28570	12316	2,32	0	0	0	1,30
CMT 5	28570	11412	2,5	0	0	0	1,41
CMT 10	28570	8702	3,28	0	0	0	1,85
CMT 15	28570	8416	3,39	0	0	0	**1,91**
CMT 20	28570	8455	3,38	0	0	0	1,90
CMT 30	28570	8453	3,38	0	0	0	1,90
CMT 50	28570	8453	3,38	0	0	0	1,90
CMT 100	28570	8453	3,38	0	0	0	1,90
CMT SLIM	28570	8453	3,38	0	0	0	1,90

Tabela 4.3: Dados da execução de Toy-Matrices-SD

Execução	Instruções	Ciclos	IPC	Perigos Detectados			Speedup
				RAW	*WAR*	*WAW*	
SMT	2917	1784	1,64	0	0	0	1
SMT DEC	2917	950	3,07	0	0	0	1,88
SMT RIP	2917	1784	1,64	0	0	0	1
SMT DEC RIP	2917	913	3,19	0	0	0	1,95
SMT W	2354	1662	1,42	0	0	0	1,07
SMT DEC W	2354	729	3,23	0	0	0	2,45
SMT RIP W	2354	1662	1,42	0	0	0	1,07
SMT DEC RIP W	2354	734	3,21	0	0	0	2,43
CMT 2	2917	1587	1,84	0	0	0	1,12
CMT 3	2917	1398	2,09	0	0	0	1,28
CMT 5	2917	1284	2,27	0	0	0	1,39
CMT 10	2917	1062	2,75	0	0	0	1,68
CMT 15	2917	1044	2,79	0	0	0	**1,71**
CMT 20	2917	1041	2,8	0	0	0	1,71
CMT 30	2917	1041	2,8	0	0	0	1,71
CMT 50	2917	1041	2,8	0	0	0	1,71
CMT 100	2917	1041	2,8	0	0	0	1,71
CMT SLIM	2917	1041	2,8	0	0	0	1,71

Tabela 4.4: Dados da execução de Small-Toy-Matrices-SD

A técnica de *Decoupled Stores* apresenta ganhos superiors aos obtidos com a Transactional WaveCache, e maiores ainda quando é utilizada a otimização W do compilador. No Capítulo 2 são descritas as técnicas de *Decoupled Stores* (Seção 2.3.3) e otimização W (Seção 2.5.1). Isto sugere que não é interessante deixar de utilizá-las para usar a Transactional WaveCache. No entanto, a técnica de *Decoupled Stores* atua relaxando a ordenação nos acessos à memória dentro de uma onda, enquanto a Transactional WaveCache atua permitindo o relaxamento entre ondas. Uma junção de ambas as técnicas é possível e necessária. Este assunto é apresentado no Capítulo 5 (Seção 5.2.3) como um possível trabalho futuro.

Para todas as aplicações há uma saturação nas acelações obtidas a medida em que o grau de especulação do sistema é aumentado. Isto sugere que o máximo de concorrência é extraido para as aplicação após um certo grau de especulação. O termo paralelismo não é usado aqui, pois embora exista um *StoreBuffer* por *Cluster*, apenas um *StoreBuffer* por vez pode atender requisições de acesso à Memória. Esta restrição já existia no WaveScalar original, mas era maior, pois apenas requisições

Execução	Instruções	Ciclos	IPC	Perigos Detectados			Speedup
				RAW	*WAR*	*WAW*	
SMT	33074	16445	2,01	0	0	0	1
SMT DEC	33074	7562	4,37	0	0	0	2,17
SMT RIP	33074	16505	2	0	0	0	1
SMT DEC RIP	33074	7577	4,37	0	0	0	2,17
SMT W	27562	16189	1,7	0	0	0	1,02
SMT DEC W	27562	7428	3,71	0	0	0	2,21
SMT RIP W	27562	16189	1,7	0	0	0	1,02
SMT DEC RIP W	27562	7422	3,71	0	0	0	2,22
CMT 2	45614	15702	2,9	1	0	0	1,05
CMT 3	45614	14387	3,17	1	0	0	1,14
CMT 5	45618	12905	3,53	1	0	0	1,27
CMT 10	35236	9342	3,77	1	0	0	1,76
CMT 15	36585	9390	3,9	1	0	0	1,75
CMT 20	33371	8671	3,85	1	0	0	**1,90**
CMT 30	33371	8680	3,84	1	0	0	1,89
CMT 50	33371	8673	3,85	1	0	0	1,90
CMT 100	33371	8673	3,85	1	0	0	1,90
CMT SLIM	33371	8673	3,85	1	0	0	1,90

Tabela 4.5: Dados da execução de Toy-Matrices-CD

Execução	Instruções	Ciclos	IPC	Perigos Detectados			Speedup
				RAW	*WAR*	*WAW*	
SMT	3371	1795	1,88	0	0	0	1
SMT DEC	3371	934	3,61	0	0	0	1,92
SMT RIP	3371	1795	1,88	0	0	0	1
SMT DEC RIP	3371	944	3,57	0	0	0	1,90
SMT W	2809	1745	1,61	0	0	0	1,03
SMT DEC W	2809	930	3,02	0	0	0	1,93
SMT RIP W	2809	1745	1,61	0	0	0	1,03
SMT DEC RIP W	2809	938	2,99	0	0	0	1,91
CMT 2	4658	1842	2,53	1	0	0	0,97
CMT 3	4662	1538	3,03	1	0	0	1,17
CMT 5	5673	1458	3,89	1	0	0	1,23
CMT 10	3994	1206	3,31	1	0	0	1,49
CMT 15	4140	1281	3,23	1	0	0	1,40
CMT 20	3917	1183	3,31	1	0	0	**1,52**
CMT 30	3917	1183	3,31	1	0	0	1,52
CMT 50	3917	1183	3,31	1	0	0	1,52
CMT 100	3917	1183	3,31	1	0	0	1,52
CMT SLIM	3917	1183	3,31	1	0	0	1,52

Tabela 4.6: Dados da execução de Small-Toy-Matrices-CD

de uma onda poderiam ser executadas por vez. O grau de especulação que extrai desempenho máximo para as aplicações Toy-Matrices-SD e Small-Toy-Matrices-SD é 15 e para as aplicações Toy-Matrices-CD e Small-Toy-Matrices-CD é 20. Estes também podem ser considerados os graus de especulação ideais para estas aplicações, pois a partir deste ponto, não há aumento de desempenho, ocorrendo apenas o desperdício de recursos para armazenar o histórico de transações adicionais.

Por conta do *overhead* com a reexecução, as acelerações obtidas com a Transactional WaveCache são menores para as aplicações que apresentam dependências do tipo *RAW* (Toy-Matrices-CD e Small-Toy-Matrices-CD). Além disso, para as aplicações *Small* apresentam acelerações mais modestas pois em aplicações maiores, existe um maior número de ondas e maior oportunidade em se extrair concorrência. Nos melhores casos, as aplicações Toy-Matrices-SD, Small-Toy-Matrices-SD, Toy-Matrices-CD e Small-Toy-Matrices-CD atingem acelerações (ou *speedups*) de 1,91, 1,71, 1,90 e 1,52, respectivamente.

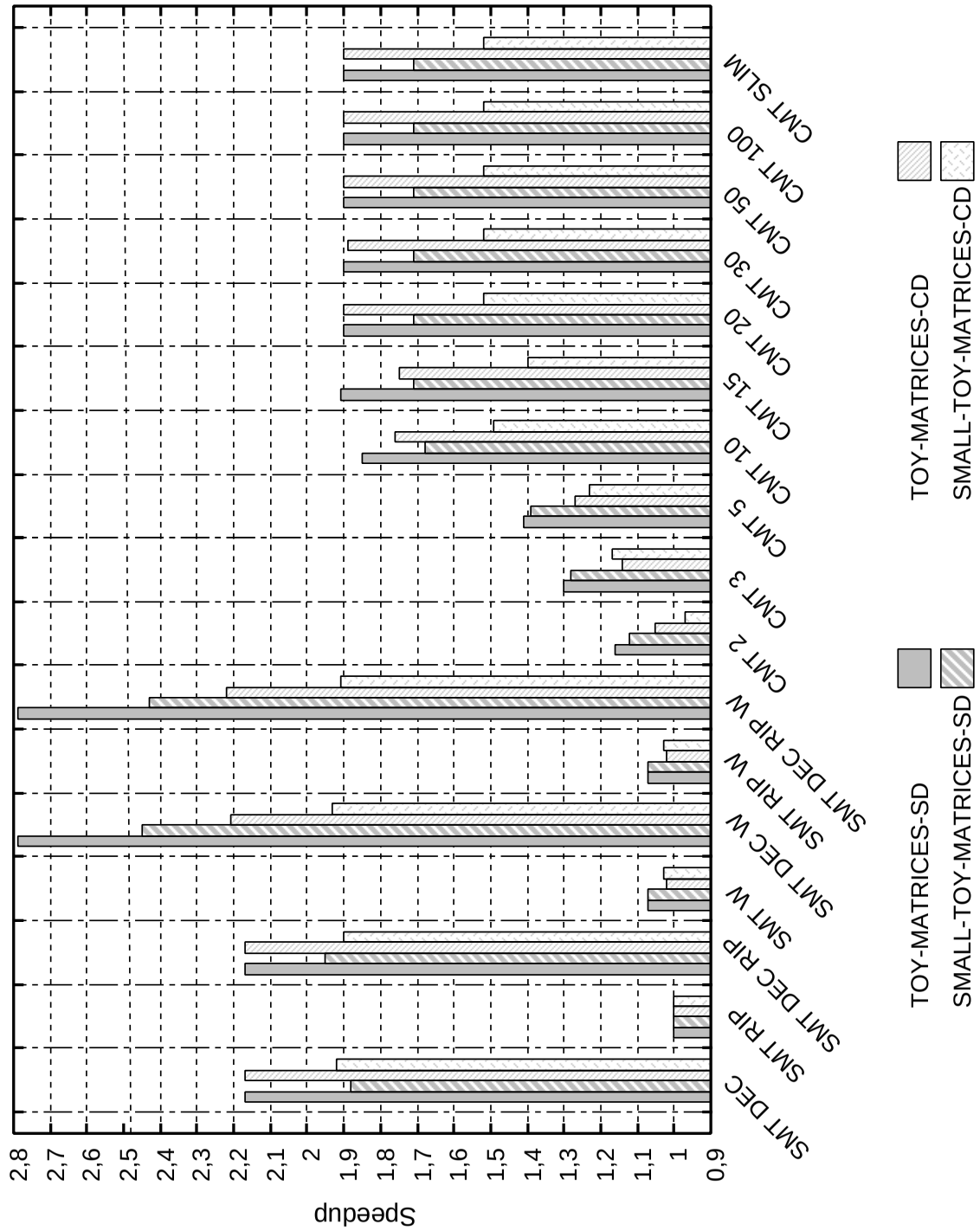


Figura 4.1: *Speedups* para Toy-Matrices-SD, Small-Toy-Matrices-SD, Toy-Matrices-CD e Small-Toy-Matrices-CD

O gráfico da Figura 4.2 mostra que, para as aplicações Toy-Matrices-SD-Stores, Small-Toy-Matrices-SD-Stores, Toy-Matrices-CD-Stores e Small-Toy-Matrices-SD-Stores houve desacelerção ou *slowdown* com o uso da Transactional WaveCache. Isto ocorre devido à alta concorrência causada pelos *Stores* já no WaveScalar original. O Transactional WaveCache não possibilitou o aumento da concorrência no acesso à memória, restando apenas o *overhead* do mecanismo.

As aplicações *Small* apresentaram menores desaceleraçõe que as demais, pois em aplicações maiores, mais *Stores* acontecem em seqüência, resultando maior concorrência já no WaveScalar original. Quanto maior a concorrência exposta originalmente, menor é a contribuição da Transactional WaveCache e mais evidentes são as desacelerações.

A Tabela 4.7 mostra as métricas coletadas para a aplicação Toy-Matrices-SD-Stores. Os dados da aplicação Small-Toy-Matrices-SD-Stores são exibidos na Tabela 4.8. Em ambas as aplicaçõe não ocorrem reexecuções, pois as mesmas não apresentam dependências do tipo *RAW*.

Nas métricas coletadas para as aplicações Toy-Matrices-CD-Stores e Small-Toy-Matrices-CD-Stores, exibidos nas Tabelas 4.9 e 4.10, é possível observar que praticamente não foram detectados perigos *RAW*. Isto se deve à alta concorrência já existente, que não permitiu a ocorrência de intercalação na execução das ondas na memória, de forma a causar o conflito.

Para o grau de especulação que ocasiona a menor desaceleração, as aplicações Toy-Matrices-SD-Stores, Small-Toy-Matrices-SD-Stores, Toy-Matrices-CD-Stores e Small-Toy-Matrices-CD-Stores atingem desacelerações de 1,13, 1,05, 1,16 e 1,05, respectivamente (inverso do *speedup*.

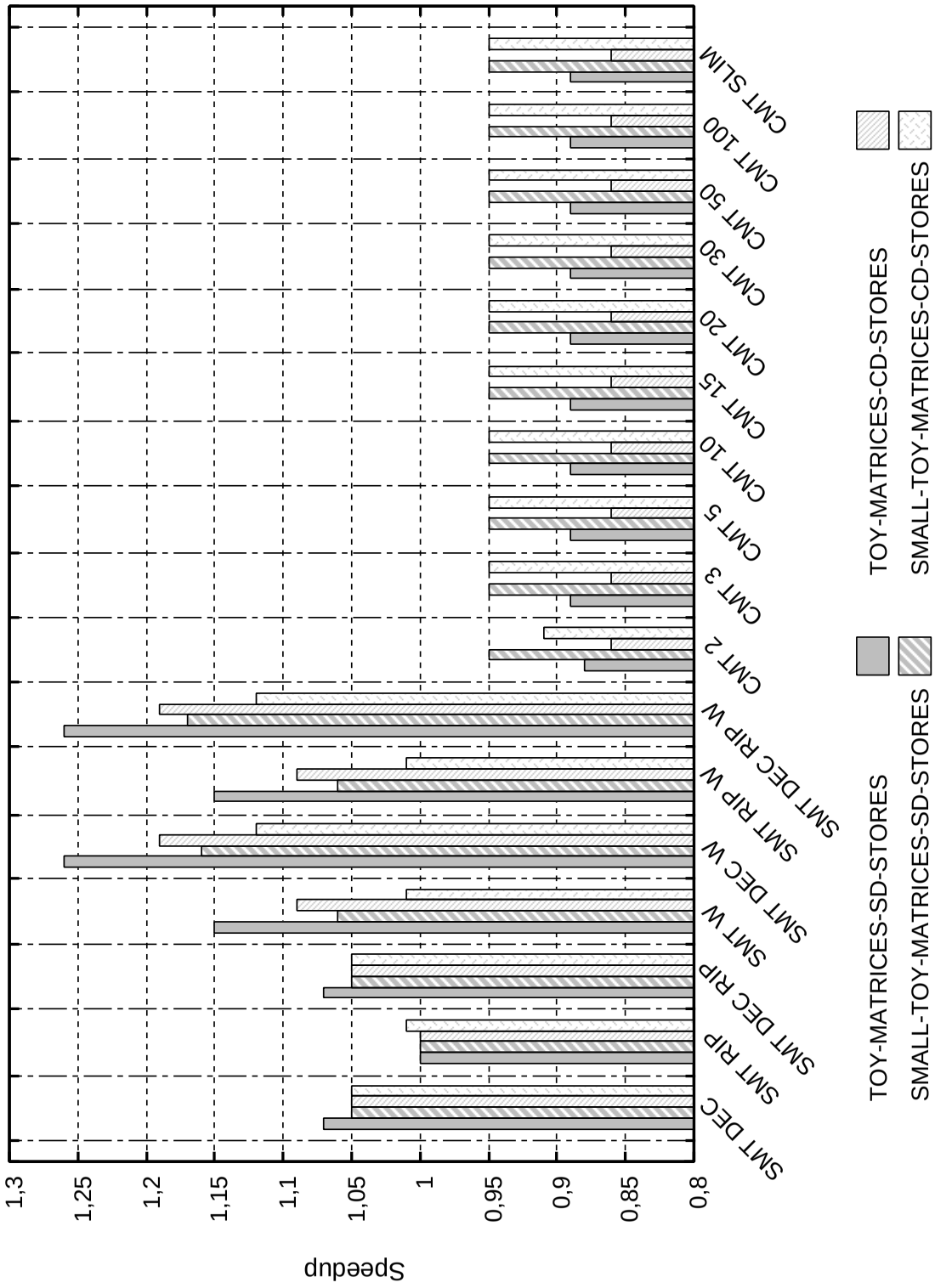


Figura 4.2: *Speedups* para Toy-Matrices-SD-Stores, Small-Toy-Matrices-SD-Stores, Toy-Matrices-CD-Stores e Small-Toy-Matrices-CD-Stores

Execução	Instruções	Ciclos	IPC	Perigos Detectados			Speedup
				RAW	*WAR*	*WAW*	
SMT	160580	32757	4,9	0	0	0	1
SMT DEC	160580	30528	5,26	0	0	0	1,07
SMT RIP	160580	32738	4,91	0	0	0	1
SMT DEC RIP	160580	30530	5,26	0	0	0	1,07
SMT W	110555	28481	3,88	0	0	0	1,15
SMT DEC W	110555	26003	4,25	0	0	0	1,26
SMT RIP W	110555	28481	3,88	0	0	0	1,15
SMT DEC RIP W	110555	26002	4,25	0	0	0	1,26
CMT 2	160580	37173	4,32	0	0	0	0,88
CMT 3	160580	36973	4,34	0	0	0	**0,89**
CMT 5	160580	36906	4,35	0	0	0	0,89
CMT 10	160580	36906	4,35	0	0	0	0,89
CMT 15	160580	36907	4,35	0	0	0	0,89
CMT 20	160580	36907	4,35	0	0	0	0,89
CMT 30	160580	36907	4,35	0	0	0	0,89
CMT 50	160580	36907	4,35	0	0	0	0,89
CMT 100	160580	36907	4,35	0	0	0	0,89
CMT SLIM	160580	36907	4,35	0	0	0	0,89

Tabela 4.7: Dados da execução de Toy-Matrices-SD-Stores

Execução	Instruções	Ciclos	IPC	Perigos Detectados			Speedup
				RAW	*WAR*	*WAW*	
SMT	16127	3613	4,46	0	0	0	1
SMT DEC	16127	3455	4,67	0	0	0	1,05
SMT RIP	16127	3613	4,46	0	0	0	1
SMT DEC RIP	16127	3455	4,67	0	0	0	1,05
SMT W	11102	3398	3,27	0	0	0	1,06
SMT DEC W	11102	3108	3,57	0	0	0	1,16
SMT RIP W	11102	3398	3,27	0	0	0	1,06
SMT DEC RIP W	11102	3097	3,58	0	0	0	1,17
CMT 2	16127	3809	4,23	0	0	0	**0,95**
CMT 3	16127	3791	4,25	0	0	0	0,95
CMT 5	16127	3787	4,26	0	0	0	0,95
CMT 10	16127	3787	4,26	0	0	0	0,95
CMT 15	16127	3787	4,26	0	0	0	0,95
CMT 20	16127	3787	4,26	0	0	0	0,95
CMT 30	16127	3787	4,26	0	0	0	0,95
CMT 50	16127	3787	4,26	0	0	0	0,95
CMT 100	16127	3787	4,26	0	0	0	0,95
CMT SLIM	16127	3787	4,26	0	0	0	0,95

Tabela 4.8: Dados da execução de Small-Toy-Matrices-SD-Stores

Execução	Instruções	Ciclos	IPC	Perigos Detectados			Speedup
				RAW	*WAR*	*WAW*	
SMT	165084	31008	5,32	0	0	0	1
SMT DEC	165084	29430	5,61	0	0	0	1,05
SMT RIP	165084	30949	5,33	0	0	0	1,01
SMT DEC RIP	165084	29428	5,61	0	0	0	1,05
SMT W	115060	28411	4,05	0	0	0	1,09
SMT DEC W	115060	26066	4,41	0	0	0	1,19
SMT RIP W	115060	28411	4,05	0	0	0	1,09
SMT DEC RIP W	115060	26081	4,41	0	0	0	1,19
CMT 2	165084	35933	4,59	0	0	0	**0,86**
CMT 3	165084	35927	4,59	0	0	0	0,86
CMT 5	165084	35912	4,6	0	0	0	0,86
CMT 10	165084	35916	4,6	0	0	0	0,86
CMT 15	165084	35916	4,6	0	0	0	0,86
CMT 20	165084	35916	4,6	0	0	0	0,86
CMT 30	165084	35916	4,6	0	0	0	0,86
CMT 50	165084	35916	4,6	0	0	0	0,86
CMT 100	165084	35916	4,6	0	0	0	0,86
CMT SLIM	165084	35916	4,6	0	0	0	0,86

Tabela 4.9: Dados da execução de Toy-Matrices-CD-Stores

Execução	Instruções	Ciclos	IPC	Perigos Detectados			Speedup
				RAW	*WAR*	*WAW*	
SMT	16581	3482	4,76	0	0	0	1
SMT DEC	16581	3312	5,01	0	0	0	1,05
SMT RIP	16581	3450	4,81	0	0	0	1,01
SMT DEC RIP	16581	3302	5,02	0	0	0	1,05
SMT W	11557	3442	3,36	0	0	0	1,01
SMT DEC W	11557	3099	3,73	0	0	0	1,12
SMT RIP W	11557	3441	3,36	0	0	0	1,01
SMT DEC RIP W	11557	3104	3,72	0	0	0	1,12
CMT 2	17243	3820	4,51	1	0	0	0,91
CMT 3	16581	3651	4,54	0	0	0	**0,95**
CMT 5	16581	3658	4,53	0	0	0	0,95
CMT 10	16581	3649	4,54	0	0	0	0,95
CMT 15	16581	3661	4,53	0	0	0	0,95
CMT 20	16581	3655	4,54	0	0	0	0,95
CMT 30	16581	3655	4,54	0	0	0	0,95
CMT 50	16581	3655	4,54	0	0	0	0,95
CMT 100	16581	3655	4,54	0	0	0	0,95
CMT SLIM	16581	3655	4,54	0	0	0	0,95

Tabela 4.10: Dados da execução de Small-Toy-Matrices-CD-Stores

Nas aplicações Toy-Matrices-SD-Stores-Min, Small-Toy-Matrices-SD-Stores-Min, Toy-Matrices-CD-Stores-Min e Small-Toy-Matrices-CD-Stores-Min a concorrência nos acessos a memória é minimizada com a repetição de 10 vezes no laço de cálculos. As tabelas 4.11, 4.12, 4.13 e 4.14 e o gráfico da Figura 4.3 mostram que isto permitiu mais oportunidades para o uso da Transactional WaveCache.

Foram obtidas acelerações de 1,33, 1,24 e 1,11 nos melhores casos, para as aplicações Toy-Matrices-SD-Stores-Min, Toy-Matrices-CD-Stores-Min e Small-Toy-Matrices-CD-Stores-Min, respectivamente. Na aplicação Small-Toy-Matrices-SD-Stores-Min houve desaceleração de 1,01. Mesmo assim, houve uma melhoria em relação a aplicação Small-Toy-Matrices-SD-Stores onde a desaceleração era de 1,05. Isto mostra que o problema dos *Stores* é realmente minimizado quando ocorrem mais rodadas de utilização dos dados preenchidos.

Nas aplicações Toy-Matrices-CD-Stores-Min e Small-Toy-Matrices-CD-Stores-Min, como o laço de cálculo dos determinantes se repete 10 vezes, ocorrem 10 possibilidades de detecção de perigos *RAW*. Dependendo da intercalação das ondas, este perigos podem ou não aparecer. Para a aplicação Toy-Matrices-CD-Stores-Min foram detectados no máximo 6 dos 10 perigos *RAW*, sendo que na maior parte das simulações, 3 perigos *RAW* foram detectados. Já na aplicação Small-Toy-Matrices-CD-Stores-Min o número máximo de perigos *RAW* detectados é 5 e na maior parte das simulações, não foram encontrados perigos deste tipo. Isto mostra que para aplicações *small*, como o número de instruções é pequeno, menor concorrência é exposta, diminuindo a possibilidade de serem detectados perigos. Este fato também, pode ser observado no desempenho, que para as aplicações nomais é superior.

Execução	Instruções	Ciclos	IPC	Perigos Detectados			Speedup
				RAW	*WAR*	*WAW*	
SMT	437457	82666	5,29	0	0	0	1
SMT DEC	437457	66456	6,58	0	0	0	1,24
SMT RIP	437457	83328	5,25	0	0	0	0,99
SMT DEC RIP	437457	66455	6,58	0	0	0	1,24
SMT W	324283	102927	3,15	0	0	0	0,80
SMT DEC W	324283	74769	4,34	0	0	0	1,11
SMT RIP W	324283	102927	3,15	0	0	0	0,80
SMT DEC RIP W	324283	74698	4,34	0	0	0	1,11
CMT 2	437457	62100	7,04	0	0	0	1,33
CMT 3	437457	62237	7,03	0	0	0	**1,33**
CMT 5	437457	64592	6,77	0	0	0	1,28
CMT 10	437457	64592	6,77	0	0	0	1,28
CMT 15	437457	64592	6,77	0	0	0	1,28
CMT 20	437457	64592	6,77	0	0	0	1,28
CMT 30	437457	64592	6,77	0	0	0	1,28
CMT 50	437457	64592	6,77	0	0	0	1,28
CMT 100	437457	64592	6,77	0	0	0	1,28
CMT SLIM	437457	64592	6,77	0	0	0	1,28

Tabela 4.11: Dados da execução de Toy-Matrices-SD-Stores-Min

Execução	Instruções	Ciclos	IPC	Perigos Detectados			Speedup
				RAW	*WAR*	*WAW*	
SMT	44154	8408	5,25	0	0	0	1
SMT DEC	44154	7005	6,3	0	0	0	1,20
SMT RIP	44154	8408	5,25	0	0	0	1
SMT DEC RIP	44154	7006	6,3	0	0	0	1,20
SMT W	32680	10443	3,13	0	0	0	0,81
SMT DEC W	32680	8316	3,93	0	0	0	1,01
SMT RIP W	32680	10443	3,13	0	0	0	0,81
SMT DEC RIP W	32680	8286	3,94	0	0	0	1,01
CMT 2	44154	9003	4,9	0	0	0	0,93
CMT 3	44154	8726	5,06	0	0	0	0,96
CMT 5	44154	8609	5,13	0	17	0	0,98
CMT 10	44154	8514	5,19	0	39	0	**0,99**
CMT 15	44154	8539	5,17	0	39	0	0,98
CMT 20	44154	8526	5,18	0	43	0	0,99
CMT 30	44154	8510	5,19	0	54	0	0,99
CMT 50	44154	8532	5,18	0	48	0	0,99
CMT 100	44154	8478	5,21	0	54	0	0,99
CMT SLIM	44154	8462	5,22	0	54	0	0,99

Tabela 4.12: Dados da execução de Small-Toy-Matrices-SD-Stores-Min

Execução	Instruções	Ciclos	IPC	Perigos Detectados			Speedup
				RAW	*WAR*	*WAW*	
SMT	472497	84985	5,56	0	0	0	1
SMT DEC	472497	63322	7,46	0	0	0	1,34
SMT RIP	472497	84985	5,56	0	0	0	1
SMT DEC RIP	472497	63349	7,46	0	0	0	1,34
SMT W	364323	107562	3,39	0	0	0	0,79
SMT DEC W	364323	82798	4,4	0	0	0	1,03
SMT RIP W	364323	107562	3,39	0	0	0	0,79
SMT DEC RIP W	364323	82787	4,4	0	0	0	1,03
CMT 2	474727	69009	6,88	6	0	0	1,23
CMT 3	473146	68554	6,9	3	0	0	**1,24**
CMT 5	473460	68577	6,9	5	0	0	1,24
CMT 10	473088	68505	6,91	3	0	0	1,24
CMT 15	473268	68536	6,91	3	0	0	1,24
CMT 20	473320	68567	6,9	3	0	0	1,24
CMT 30	473088	68510	6,91	3	0	0	1,24
CMT 50	473088	68510	6,91	3	0	0	1,24
CMT 100	473088	68510	6,91	3	0	0	1,24
CMT SLIM	473088	68510	6,91	3	0	0	1,24

Tabela 4.13: Dados da execução de Toy-Matrices-CD-Stores-Min

Execução	Instruções	Ciclos	IPC	Perigos Detectados			Speedup
				RAW	*WAR*	*WAW*	
SMT	47694	8907	5,35	0	0	0	1
SMT DEC	47694	6970	6,84	0	0	0	1,28
SMT RIP	47694	8907	5,35	0	0	0	1
SMT DEC RIP	47694	6980	6,83	0	0	0	1,28
SMT W	36720	11124	3,3	0	0	0	0,80
SMT DEC W	36720	8680	4,23	0	0	0	1,03
SMT RIP W	36720	11132	3,3	0	0	0	0,80
SMT DEC RIP W	36720	8680	4,23	0	0	0	1,03
CMT 2	51309	8627	5,95	5	0	0	1,03
CMT 3	48393	8181	5,92	3	0	0	1,09
CMT 5	47694	8065	5,91	0	0	0	1,10
CMT 10	47694	8108	5,88	0	0	0	1,10
CMT 15	47694	8065	5,91	0	0	0	1,10
CMT 20	47694	8067	5,91	0	0	0	1,10
CMT 30	47694	8055	5,92	0	0	0	**1,11**
CMT 50	47694	8055	5,92	0	0	0	1,11
CMT 100	47694	8055	5,92	0	0	0	1,11
CMT SLIM	47694	8055	5,92	0	0	0	1,11

Tabela 4.14: Dados da execução de Small-Toy-Matrices-CD-Stores-Min

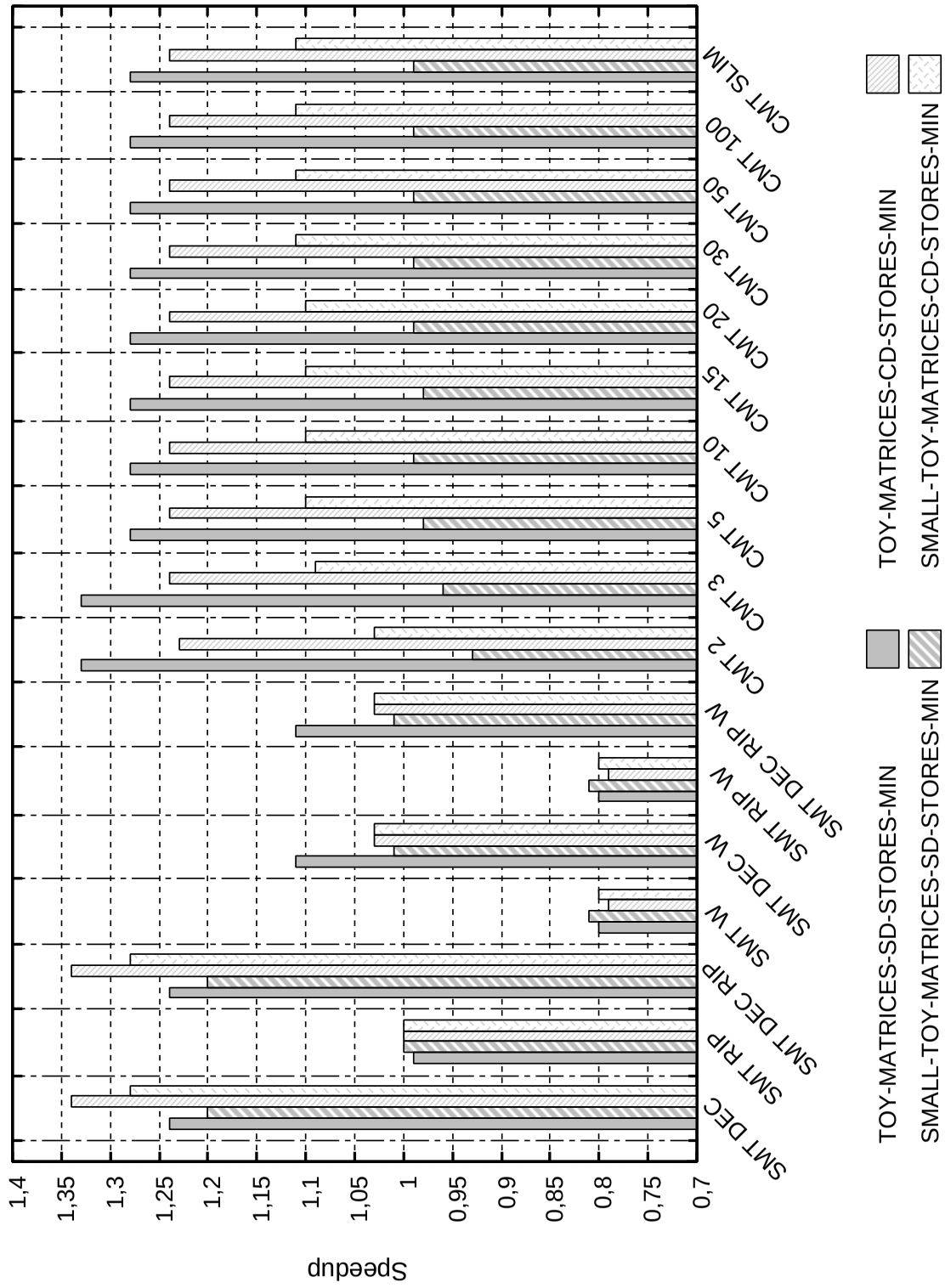


Figura 4.3: *Speedups* para Toy-Matrices-SD-Stores-Min, Small-Toy-Matrices-SD-Stores-Min, Toy-Matrices-CD-Stores-Min e Small-Toy-Matrices-CD-Stores-Min

A Figura 4.4 mostra o gráfico com os *speedups* para as aplicações Toy-Vectors-CD-WAR-WAW-RAW e Small-Toy-Vectors-CD-WAR-WAW-RAW . As tabelas 4.15 e 4.16 contém todos os dados extraídos com a simulação. A análise conjunta do gráfico e das tabelas mostra que, para as duas aplicações, o aumento do grau de especulação causa um crescimento no número de conflitos encontrados. O mecanismo consegue aumentos de desempenho, mesmo com o aumento no número de conflitos. As acelerações máximas obtidas para as aplicações Toy-Vectors-CD-WAR-WAW-RAW e Small-Toy-Vectors-CD-WAR-WAW-RAW foram de 2,40 e 2,02.

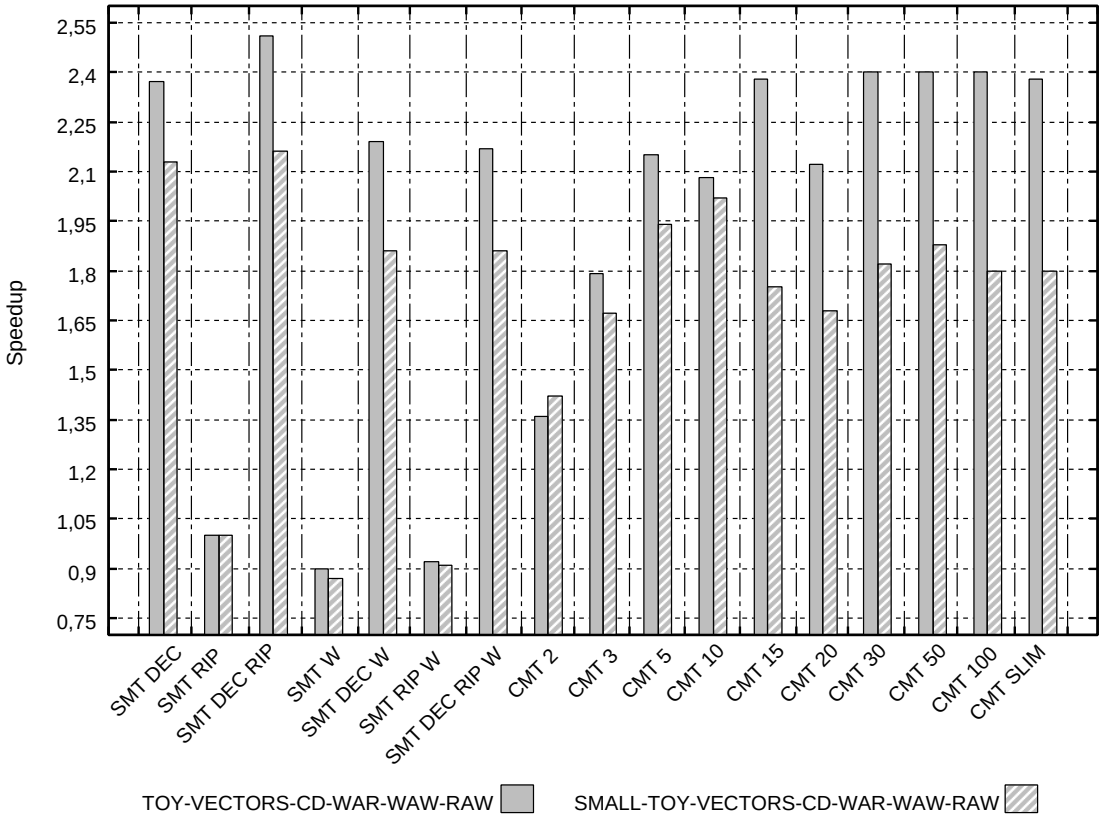


Figura 4.4: *Speedups* para as aplicações Small-Toy-Vectors-WAR-WAW-RAW e Toy-Vectors-WAR-WAW-RAW

Nas simulações onde são detectados perigos *RAW*, há uma queda na acelação de desempenho. Na aplicação Small-Toy-Vectors-CD-WAR-WAW-RAW, por exemplo, as acelerações obtidas começam em 53,31% para grau de especulação 2 e atingem 2,02 para grau de especulção 10. Para grau de especulação 15 ocorre a detecção de um perigo *RAW*, fazendo com que a aceleração caia para 1,75.

Execução	Instruções	Ciclos	IPC	Perigos Detectados			Speedup
				RAW	*WAR*	*WAW*	
SMT	74498	46483	1,6	0	0	0	1
SMT DEC	74498	19629	3,8	0	0	0	2,37
SMT RIP	74498	46328	1,61	0	0	0	1
SMT DEC RIP	74498	18525	4,02	0	0	0	2,51
SMT W	70518	51579	1,37	0	0	0	0,90
SMT DEC W	70518	21182	3,33	0	0	0	2,19
SMT RIP W	70518	50660	1,39	0	0	0	0,92
SMT DEC RIP W	70518	21378	3,3	0	0	0	2,17
CMT 2	74498	34185	2,18	0	0	0	1,36
CMT 3	74498	25898	2,88	0	0	50	1,79
CMT 5	74498	21644	3,44	0	0	101	2,15
CMT 10	87960	22386	3,93	1	1	261	2,08
CMT 15	74498	19493	3,82	0	3	284	2,38
CMT 20	87106	21927	3,97	1	1	321	2,12
CMT 30	74498	19395	3,84	0	3	358	**2,40**
CMT 50	74498	19382	3,84	0	18	329	2,40
CMT 100	74498	19402	3,84	0	84	241	2,40
CMT SLIM	74498	19556	3,81	0	236	179	2,38

Tabela 4.15: Dados da execução de Toy-Vectors-CD-WAR-WAW-RAW

Execução	Instruções	Ciclos	IPC	Perigos Detectados			Speedup
				RAW	*WAR*	*WAW*	
SMT	6996	3986	1.76	0	0	0	1
SMT DEC	6996	1875	3.73	0	0	0	2,13
SMT RIP	6996	3974	1.76	0	0	0	1
SMT DEC RIP	6996	1845	3.79	0	0	0	2,16
SMT W	6616	4607	1.44	0	0	0	0,87
SMT DEC W	6616	2140	3.09	0	0	0	1,86
SMT RIP W	6616	4392	1.51	0	0	0	0,91
SMT DEC RIP W	6616	2144	3.09	0	0	0	1,86
CMT 2	6996	2801	2.5	0	0	1	1,42
CMT 3	6996	2384	2.93	0	0	3	1,67
CMT 5	6996	2058	3.4	0	0	12	1,94
CMT 10	6996	1969	3.55	0	2	21	**2,02**
CMT 15	6996	2282	3.07	1	0	23	1,75
CMT 20	6996	2379	2.94	1	3	25	1,68
CMT 30	6996	2194	3.19	1	3	35	1,82
CMT 50	6996	2115	3.31	1	17	42	1,88
CMT 100	6996	2215	3.16	1	17	42	1,80
CMT SLIM	6996	2215	3.16	1	17	42	1,80

Tabela 4.16: Dados da execução de Small-Toy-Vectors-CD-WAR-WAW-RAW

Os gráficos das figuras 4.5 e 4.6 mostram a variação no percentual de instruções, com o uso da otimização W no WaveScalar original e com diferentes graus de especulação para a Transactional WaveCache (sem otimização W). Todas as aplicações apresentaram redução do número de instruções executadas em relação ao WaveScalar original quando a otimização W é utilizada. As aplicações do tipo SD (Toy-Matrices-SD, Toy-Matrices-SD-Stores e Toy-Matrices-SD-Stores-Min) não apresentaram alterações no percentual de instruções executadas com o uso da Transactional WaveCache, pois não ocorreram reexecuções. Para as aplicações do tipo CD (Toy-Matrices-CD, Toy-Matrices-CD-Stores e Toy-Matrices-CD-Stores-Min) ocorrem aumentos no percentual de instruções executadas com o uso da Transactional WaveCache devido as reexecuções.

Intuitivamente, o aumento no grau de especulação do sistema deveria resultar em um número maior de instruções executadas, pois quando maior o relaxamento nos acessos a memória, mais operações de *Load* seriam executadas, liberando as instruções dependentes no grafo *dataflow*. No entanto, este comportamento não é verificado. O que ocorre é que quanto maior o grau de especulação do sistema menor o número de instruções executadas desnecessariamente.

Para graus de especulação baixos, a execução de instruções que independem de resultados de *Loads*, como o incremento das variáveis de controle de laços de repetição, pode se adiantar muito em relação aos acessos à memória. Enquanto as operações de *Load* e *Store* têm sua execução limitada pela janela de especulação, as demais instruções executam livremente. Quando as ondas que contém a dependência finalmente fizerem parte da janela de especulação, o conflito será detectado e muitas instruções já terão sido executadas desnecessariamente. Com a detecção do conflito todas essas instruções serão reexecutadas.

Para janelas de especulação grandes, a execução continua sendo feita da mesma forma. Certos ramos do grafo *dataflow* executam mais rapidamente que outros. No entanto, com o maior relaxamento na execução das operações de memória, as requisições de memória geradas por estes trechos, que estão adiantados, poderão ser atendidas, pois ondas diferentes podem executar na memória dentro dos limites da

janela de especulação. Desta forma, o conflito é detectado mais rapidamente, antes que tantas instruções tenham executado desnecessariamente.

Para limitar a possibilidade de que uma parte das instruções execute muito a frente de outras, evitando a explosão de paralelismo, pode ser usada a técnica de *k-loop bounding* mencionada no Capítulo 2 (Seção 2.6). No WaveScalar original, essa técnica foi implementada no subsistema de memória, fazendo com que requisições de acesso a memória de ondas superiores a janela de tamanho $\boldsymbol{k}$ não sejam recebidas pelo *StoreBuffer* responsável. Isto faz com que todas as instruções que enviaram estas requisições fiquem aguardando a abertura da janela. As instruções que não dependem direta ou indiretamente de resultados provenientes da memória não têm sua execução limitada com esta implementação. A Transactional WaveCache não foi implementada para ser utilizada com a técnica de *k-loop bounding* do WaveScalar original. Estudos são necessários para fazer adptações desta técnica ao novo modelo de ordenação de operações de memória.

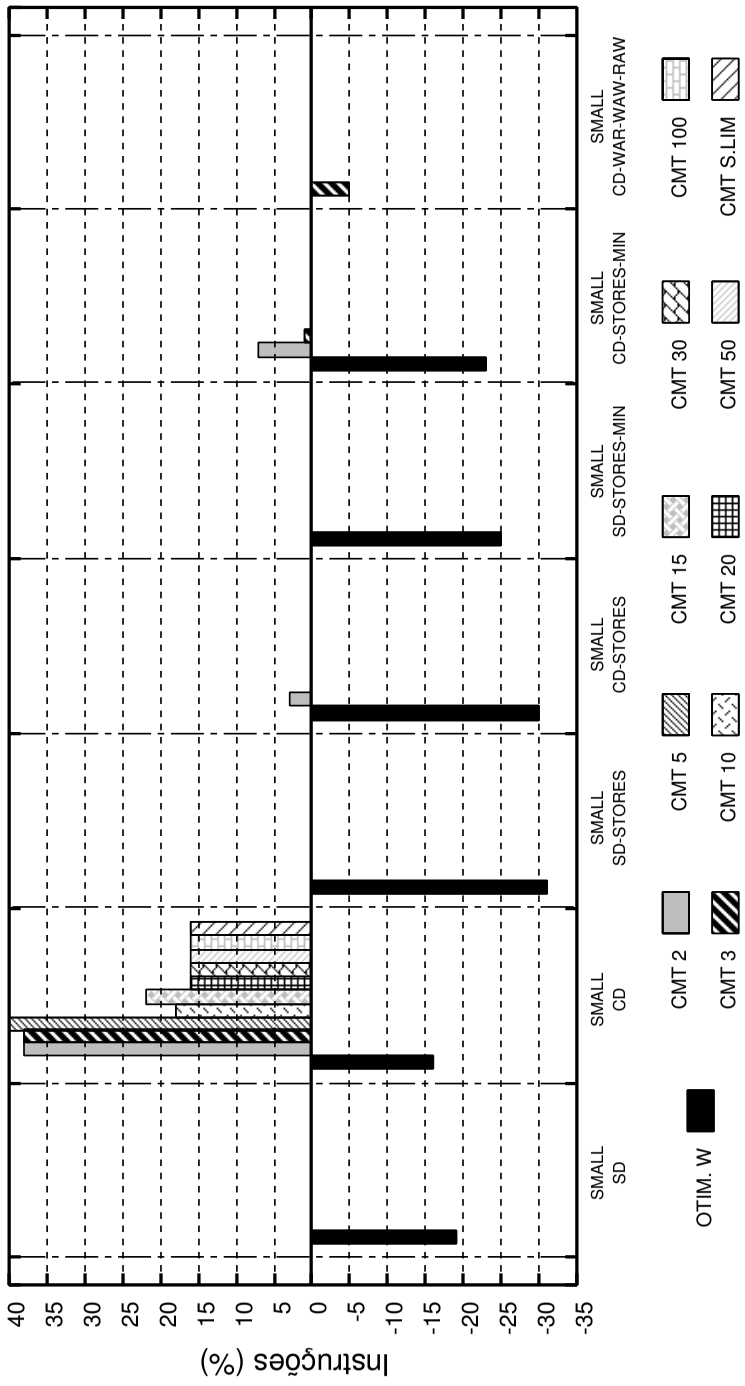


Figura 4.5: Percentual de instruções executadas em relação ao WaveScalar original e sem otimizações para as aplicações *small*

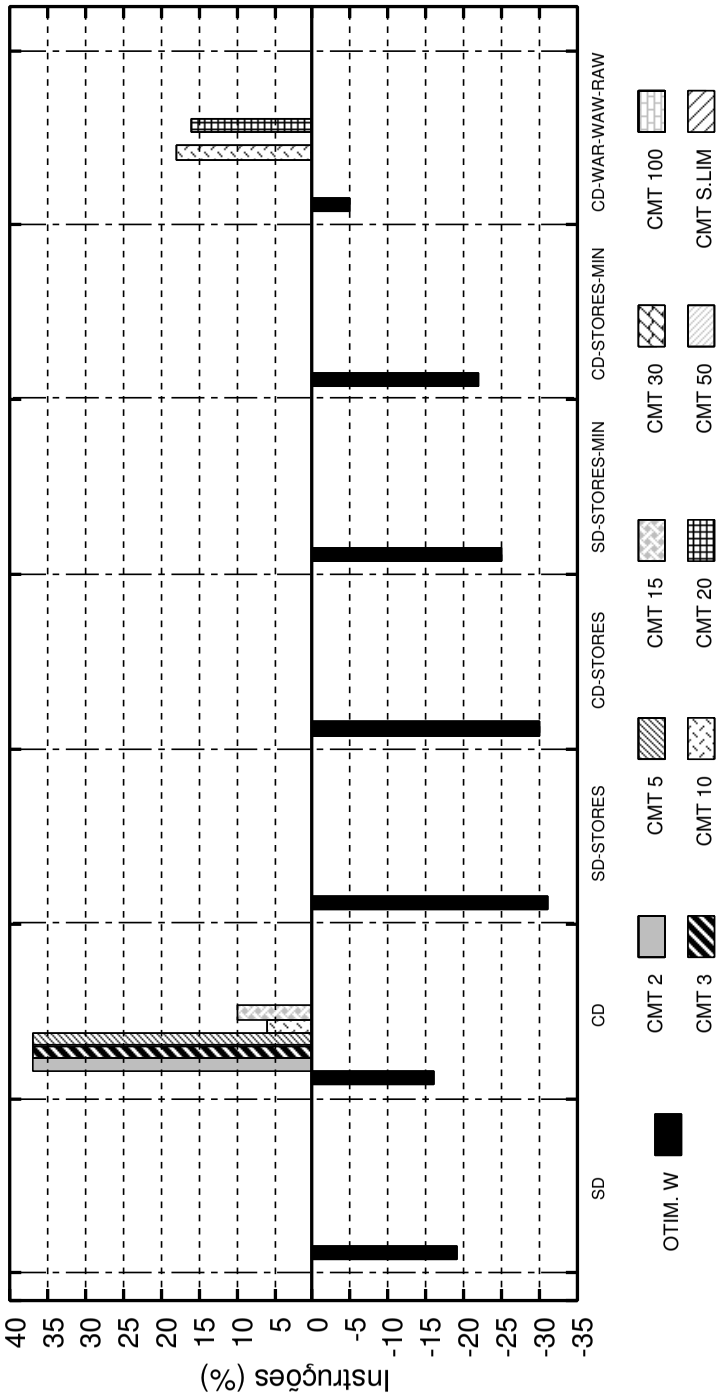


Figura 4.6: Percentual de instruções executadas em relação ao WaveScalar original e sem otimizações para as aplicações normais

Capítulo 5

Conclusões e Trabalhos Futuros

5.1 Conclusões

Este trabalho apresentou modificações, que aumentam o relaxamento na ordem em que as operações de memória são executadas, no mecanismo de ordenação de memória do WaveScalar. A Transactional WaveCache permite execução especulativa e fora-de-ordem de operações de memória de diferentes ondas. Foi possível demonstrar o impacto no desempenho do processador que incorpora tal mecanismo com o uso de aplicações artificiais. Entre as principais conclusões obtidas à partir dos experimentos, destacam-se:

- Foi possível verificar a corretude dos programas executados usando o mecanismo. Isto foi feito com a modificação do simulador para imprimir a situação final de todos os endereços alterados na memória, ou seja, uma lista com os últimos valores escritos em cada endereço. Para todas os experimentos realizados esta lista foi comparada com a lista gerada pelo simulador original (sem a Transactional WaveCache). Em nenhum dos casos foram encontradas diferenças entre as listas, o que signica que os programas geram o mesmo resultado final na memória, isso demonstra a corretude da implementação efetuada. Além disso, em nenhuma das simulações restaram *tokens* no modelo de execução, indicando que todos os *tokens* foram consumidos ou apagados (no caso de reexecuções). Também não restarão requisições não resolvidas no subsistema

de memória;

- A implementação da técnica de *k-loop bounding*, que minimiza o problema de explosão de paralelismo, foi feita no WaveScalar original apenas para evitar o recebimento de requisições de operações de memória de ondas muito maiores que a onda corrente. O mecanismo não impede que instruções que não dependem de acesso à memoria executem à frente no sistema de execução. Além disso, no simulador original existe apenas uma fila de requisições centralizada. Como na Transactional WaveCache foi necessário incluir uma fila de requisições para cada onda, a técnica de *k-loop bounding* foi desabilitada. Com isso, foi verificado que a ocorrência da explosão de paralelismo causa mais um efeito indesejado em aplicações executadas com o uso da Trasactional WaveCache: o aumento do número de instruções executadas desnecessariamente. Além disso, observou-se que o aumento do grau de especulação do sistema minimiza este efeito. Isto é devido ao fato de que quanto maior o grau de especulação, mais rapidamente serão encontrados os conflitos, deixando menos tempo para que instruções desnecessárias tenham iniciado sua execução;

- Observou-se que a técnica é mais eficiente para aplicações que apresentem originalmente um grau de concorrência baixo nos acessos à memória. Porém, mesmo para aplicações com um razoável grau de concorrência, o mecanismo consegue oferecer melhorias de desempenho. Para aplicações com baixo grau de concorrência o mecanismo oferece acelerações de 1,91, 1,71, 1,90 e 1,52, considerando o grau de especulação ideal para estas aplicações. Em aplicações com grau de concorrência alto, o mecanismo não foi eficiente na extração de mais concorrência resultando em desacelerações de 1,13, 1,05, 1,16 e 1,05. Para aplicações com grau de concorrência intemediário, foram obtidas acelerações de 1,33, 1,24 e 1,11 para três aplicações e desaceleração de 1,01 para outra. Para aplicações com baixo grau de concorrência e possibilidade alta de detecção dos três tipos de conflito foram atingidas acelerações de 2,40 e 2,02;

- O aumento no grau de especulação do sitema resulta em melhorias de desempenho ou redução do *slowdown*. Para aplicação Toy-Matrizes-SD, por exemplo,

foi observada uma aceleração de 1,16 para o grau de especulação mínimo e 1,91 para o grau de especulação ideal. Já para a aplicação Toy-Matrizes-CD, estes números foram 1,05 e 1,90;

- Nas simulações onde são detectados perigos *RAW*, há uma queda na acelação de desempenho. Na aplicação Small-Toy-Vectors-CD-WAR-WAW-RAW, por exemplo, as acelerações obtidas começam em 1,42 para grau de especulação 2 e atingem 2,02 para grau de especulção 10. Para grau de especulação 15 ocorre a detecção de um perigo *RAW*, fazendo com que a aceleração caia para 1,75.

5.2 Trabalhos Futuros

O WaveScalar foi a primeira arquitetura *dataflow* que executa programas escritos em linguagens imperativas. Este trabalho apresenta o primeiro mecanismo de que se tem conhecimento que permite a reordenação de operações de memória pertencentes a ondas distintas, em uma arquitetura deste tipo. Sendo assim, várias oportunidades de investigações são visualizadas e comentadas nesta seção.

5.2.1 Avaliação comportamental mais extensiva do mecanismo

A simulação com aplicações artificiais permite avaliar o comportamento do mecanismo para diferentes situações. O fato de conhecer o código que está sendo executado e saber o percentual de perigos de cada tipo ao qual o mecanismo é exposto, torna possível fazer uma relação entre a situação simulada, o número de perigos realmente encontrado e o desempenho atingido. Uma avaliação mais extensiva deste comportamento pode auxiliar na composição de informações de *profiling* que poderiam ser utilizadas pelo compilador para desabilitar a especulação em momentos onde haveria perda de desempenho, conforme descrito na Seção 5.2.6.

Além de avaliar o mecanismo com uma gama maior de aplicações, é necessário também verificar o impacto que diferentes otimizações de compilador (O0, O1,

O2 e O3) exercem sobre o desempenho. Simulações com diferentes parâmetros arquiteturais, como tamanho das filas de entrada e saída dos *PEs*, número máximo de requisições recebidas por *StoreBuffer* em cada ciclo e também o algoritmo de *placement* utilizado são importantes para verificar se os parâmetros arquiteturais ideais, estabelecidos para o WaveScalar original, permanecem os mesmos, quando o mecanismo é utilizado.

5.2.2 Avaliação do mecanismo com *benchmarks*

Conforme mencionado no Capítulo 2 (Seção 2.5), ainda não existe um compilador disponível para o WaveScalar. Para executar programas em seu simulador é necessário compilá-los para gerar código *assembly* da máquina Alpha e depois utilizar um tradutor binário para finalmente gerar o código assembly do WaveScalar. O tradutor binário existente possui limitações e não consegue converter efetivamente todo o código. Sendo assim, um simulador Alpha é usado em conjunto com o simulador WaveScalar. As funções que possuem instruções que não puderam ser traduzidas são automaticamente executadas no simulador Alpha.

Uma chamada de função no WaveScalar pode ser desviada para a máquina Alpha e, neste caso, o mecanismo de ordenação de memória criado neste trabalho não pô-de ser utilizado. Para que seja possível executar programas com chamadas de função, criados com o tradutor binário, o mecanismo deve ser adaptado. Um estudo mais profundo deve ser feito para determinar as modificações necessárias. Transações precisariam ser abortadas para que o controle seja passado ao simulador da máquina Alpha e a função seja executada com os dados corretos na memória. Isto pode causar um impacto no desempenho que não existiria na presença de um compilador.

O WaveScalar permite acessos a memória de 8, 16, 32 e 64 *bits*. O mecanismo criado neste trabalho não está preparado para trabalhar com acessos de tamanhos diferentes. Todos os programas usados para a avaliação do mecanismo, foram criados para fazer acessos de apenas 32 *bits*. Para o entendimento do problema com acessos de tamanho diferentes, é dado o seguinte exemplo:

- Suponha que uma operação de *Store* pertencente a onda de número 5 é executada especulativamente, gravando um valor de 64 *bits* no endereço 100;

- Em seguida, uma operação de *Store* pertencente a onda de número 3 é executada especulativamente, gravando um valor de 32 *bits* no endereço 102. Aqui existe um problema que não é tratado pelo mecanismo. Ocorre um perigo do tipo *WAW* que não é identificado, pois as informaçõs dos dois *Stores*, necessárias para detecção do conflito, foram amazenadas em tabelas diferentes na estrurura *MemOpHistory*;

- Suponha agora que um *Load* é executado especulativamente para a onda 6, carregando um dado de 64 *bits* do endereço 100. Embora a operação ocorra sem conflitos com as duas anteriores, o valor carregado está incorreto, pois o conflito *WAW* anterior não foi detectado.

A situação descrita anteriormente é outro obstáculo para a avaliação da técnica com *benchmarks*. Modificações são necessárias para inserir uma relação entre tabelas da estrutura *MemOpHistory* que contém os históricos de operações que acessam endereços próximos.

5.2.3 Junção com a técnica de *Decoupled Stores* (*Partial Stores*) e *Ripple Numbers*

Como visto no Capítulo 4, a técnica de *Decoupled Stores* traz um considerável ganho de desempenho ao WaveScalar. A Transcational WaveCache foi implementada sem fazer a integração com este mecanismo. No entanto, as duas técnicas são ortogonais: os *Decoupled Stores* permitem a reordenação de operações de memória dentro de uma onda; já a Transactional WaveCache permite a execução de operações de memória de ondas distintas, fora-de-ordem. Um estudo se faz necessário para unir as duas técnicas de maneira a aproveitar os benefícios de ambas.

A técnica de *Ripple Numbers* provê ganhos de desempenho mais modestos. No entanto, esta técnica já funciona em conjunto com a técnica de *Decoupled Stores*

e também trabalha com operações dentro de uma onda. Sendo assim, a tarefa de mesclar a Transactional WaveCache com os *Ripple Numbers* pode ser feita em conjunto com a tarefa de união com os *Decoupled Stores*.

5.2.4 Junção com a otimização W do tradutor binário

A Transactional WaveCache foi construída sem a inclusão da otimização W. Quando um conflito é detectado em um onda X, todas as ondas $\geq$X são reexecutadas, bastando reeviar os operandos de entrada de X. Com a otimização W, ondas posteriores a X podem ter recebido operandos de ondas inferiores a X e para fazer uma reexecução, todos estes operandos também deveríam ser reenviados. Um estudo é necessário para verificar qual a melhor forma de identificar e guardar estes operandos, bem como se a perda de desempenho associada seria inferior ao ganho obtido com o uso da otimização. Além disso, na Transactional WaveCache cada instrução de *WA* envia uma cópia de seu operando para o *StoreBuffer* associado a sua onda. O operando é então incluído na estrutura *WaveContextTables* para ser usado no caso de uma reexecução. A redução de instruções *WA* traz uma vantagem para este mecanismo: eliminar o *overhead* relacionado ao envio dessas mensagens.

5.2.5 Determinação do tamanho das estruturas do mecanismo

A implementação da Transactional WaveCache apresentada e avaliada neste trabalho não impõe limites as estrutuuturas *MemOpHistory*, *WaveContextTables* e *SearchCatalog* que armazenam as informações de uma transação. Para uma implementação em *hardware*, tais estruturas devem possuir tamanhos finitos. Além disso, foi considerado que a busca nessas estruturas seria realizada dentro de um ciclo de *clock*.

O tamanho dessas estruturas limita o número de ondas especulativas em execução, bem como o número de operações especulativas por onda. Existe sempre uma onda não especulativa em execução e ela não precisa armazenar informações nessas

estruturas. Portanto, ao fixar esses limites não há riscos de paradas na execução. O que pode ocorrer é uma perda de paralelismo, e, conseqüentemente, desempenho. Um estudo é necessário para determinar o tamanho ideal dessas estruturas, bem como sua relação com o desempenho, área no *chip* e tamanho do ciclo.

Os resultados do Capítulo 4 (Seção 4.2) também mostram que o aumento no grau de especulação no sistema não extrai mais paralelismo de um programa a partir de um certo ponto. A questão do grau de especulação está relacionada com a tarefa de determinar o tamanho ideal das estruturas que armazenam as informações das transações. Uma investigação para verificar o grau de especulação ideal para aplicações reais deve ser conduzido como parte do estudo que determinará o tamanho de tais estuturas.

Outra questão a ser resolvida é a presença ou não do campo valor na estrutura *MemOpHistory*. Como mencionado no Capítulo 3 (Seção 3.3), o uso deste campo permite uma diminuição dos carregamentos extras necessários para os *Stores* que não causam perigos do tipo WAW. Desta forma, o campo valor é usado para que seja possível pegar o valor da operação anterior (*Load* ou *Store*) que ainda se encontra nesta estrutura e utilizá-lo para preencher o campo *bkp* do *Store*. Caso o campo valor seja eliminado, o campo *bkp* ainda pode ser utilizado com este propósito apenas para as operações de *Load*, que não usam este campo. O número de carregamentos evitados seria menor, porém a estrurura *MemOpHistory* terá seu tamanho reduzido.

Caso o campo valor seja mantido, outros mecanismos podem também utilizá-lo. No caso de *Loads*, caso já exista um *Load* anterior na estrutura *MemOpHistory*, não é necessário fazer o acesso a memória, apenas repassando o campo valor para a operação de carga. Além disso, as instruções de *Store* não precisam necessariamente realizar a escrita na memória. A escrita pode ser feita no momento em que a transação é completada (*commit*) e apenas para o último valor escrito em cada endereço. No caso de vários *Stores* feitos para o mesmo endereço em uma onda, apenas o último seria efetivamente realizado.

Para ondas completadas que propagam o *commit* para outras que já haviam

terminado, a estrutura *MemOpHistory* pode ser analisada para todas essas ondas em conjunto, fazendo que apenas a última escrita para cada endereço de todo o grupo de ondas seja efetivamente realizada. Para diversos *Stores* que acessam o mesmo endereço em ondas diferentes, é feito somente o último. Um estudo deve ser conduzido para verificar a melhor alternativa e como deve ser feita a implementação destas melhorias.

Conforme mencionado no Capítulo 3 (Seção 3.6.2), o número de linhas do Mapa de Execução influi na eficiência do mecanismo para remover operandos. Em aplicações que apresentem reexecuções em diversos pontos, o número de linhas pode não ser suficiente para representar a relação entre todas as ondas e seus números de execução, fazendo com que operandos de execuções antigas sejam aceitos por um *PE*. Neste trabalho, este limite não foi implementado, sendo necessário um estudo para definir o tamanho ideal desta estrutura, bem como a determinação da relação entre a perda de eficiência do mecanismo e o limite imposto para a estrutura.

5.2.6 Mecanismos para desabilitar a especulação

Como mostram os resultados exibidos no Capítulo 4 (Seção 4.2), em situações onde não é possível extrair mais paralelismo na execução das operações de memória ou onde as reexecuções causadas por perigos *RAW* são freqüentes, a Transactional WaveCache pode causar perdas de desempenho. Desabilitar a especulação nesses momentos pode diminuir estas perdas ou até mesmo trazer ganhos. A criação de mecanismos que identifiquem tais situações é importante e uma investigação mais profunda é necessária.

Tais mecanismos podem ser implementados em *hardware* para extrair informações do programa em tempo de execução estabelecendo padrões que indiquem as ondas $\boldsymbol{K}$ onde ocorreriam perigos *RAW* e simplesmente desabilitar a especulação para estas ondas. Todas as ondas até $\boldsymbol{K-1}$ poderiam executar especulativamente na memória e as ondas $\boldsymbol{K}$ só poderíam executar caso fossem não-especulativas. Depois da execução de cada onda $\boldsymbol{K}$, a especulação é reativada.

Outra possibilidade é a implementação desses mecanismos em compilador para desabilitar especulação baseado em informações extraídas em tempo de compilação. Além disso, o programador pode também indicar trechos de código onde ele deseja desabilitar a especulação ou indicadores de grau de dependência entre iterações de um *loop* ou outros trechos de código, fornecendo assim informações ao compilador para que se desabilite especulações de forma mais eficiente.

Referências Bibliográficas

[1] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman, "Compiler and runtime support for efficient software transactional memory", in *Proceedings of the 2006 Conference on Programming language design and implementation*, pp. 26–37, Jun 2006.

[2] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, "Clock rate versus ipc: the end of the road for conventional microarchitectures", in *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, (New York, NY, USA), pp. 248–259, ACM Press, 2000.

[3] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded transactional memory", *IEEE Micro*, vol. 26, no. 1, pp. 59–69, 2006.

[4] Arvind and R. Nikhil, "Executing a program on the mit tagged-token dataflow architecture", *Computers, IEEE Transactions on*, vol. 39, pp. 300–318, March 1990.

[5] E. A. Ashcroft and W. W. Wadge, "Lucid, a nonprocedural language with iteration", *Commun. ACM*, vol. 20, no. 7, pp. 519–526, 1977.

[6] T. Austin, "Diva: a reliable substrate for deep submicron microarchitecture design", in *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*, pp. 196–207, 16-18 Nov. 1999.

[7] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. V. Biesbrouck, G. Pokam, B. Calder, and O. Colavin, "Unbounded page-

based transactional memory", *SIGARCH Comput. Archit. News*, vol. 34, no. 5, pp. 347–358, 2006.

[8] J. Chung, C. C. Minh, A. McDonald, T. Skare, H. Chafi, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun, "Tradeoffs in transactional memory virtualization", in *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 371–381, ACM, 2006.

[9] D. Culler, *Managing Parallelism and Resources in Scientific Dataflow Programs*. PhD thesis, Laboratory for Computer Science, M.I.T., June 1989.

[10] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid transactional memory", in *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 336–346, ACM, 2006.

[11] A. L. Davis, "The architecture and system method of ddm1: A recursively structured data driven machine", in *ISCA '78: Proceedings of the 5th annual symposium on Computer architecture*, (New York, NY, USA), pp. 210–215, ACM Press, 1978.

[12] J. B. Dennis, "First version dataflow procedure language", Tech. Rep. MAC TM61, MIT Laboratory for Computer Science, 1991.

[13] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor", *SIGARCH Comput. Archit. News*, vol. 3, no. 4, pp. 126–132, 1974.

[14] R. Desikan, D. C. Burger, S. W. Keckler, and T. Austin, "Sim-alpha: A validated, execution-driven alpha 21264 simulator", Tech. Rep. TR-01-23, UT-Austin Computer Sciences, 2001.

[15] K. Fraser, *Practical lock freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579.

[16] V. G. Grafe, G. S. Davidson, J. E. Hoch, and V. P. Holmes, "The epsilon dataflow processor", in *ISCA '89: Proceedings of the 16th annual international symposium on Computer architecture*, (New York, NY, USA), pp. 36–45, ACM Press, 1989.

[17] J. R. Gurd, C. C. Kirkham, and I. Watson, "The manchester prototype dataflow computer", *Commun. ACM*, vol. 28, no. 1, pp. 34–52, 1985.

[18] L. Hammond, B. D. Carlstrom, V. Wong, M. Chen, C. Kozyrakis, and K. Olukotun, "Transactional coherence and consistency: Simplifying parallel hardware and software", *IEEE Micro*, vol. 24, Nov-Dec 2004.

[19] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun, "Programming with transactional coherence and consistency (tcc)", in *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 1–13, ACM, 2004.

[20] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional memory coherence and consistency", in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, p. 102, IEEE Computer Society, Jun 2004.

[21] T. Harris and K. Fraser, "Language support for lightweight transactions", in *Object-Oriented Programming, Systems, Languages, and Applications*, pp. 388–402, Oct 2003.

[22] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy, "Composable memory transactions", in *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, (New York, NY, USA), pp. 48–60, ACM, 2005.

[23] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer, "Software transactional memory for dynamic-sized data structures", pp. 92–101, Jul 2003.

[24] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures.", in *ISCA*, pp. 289–300, 1993.

[25] A. Jain, W. Anderson, T. Benninghoff, D. Berucci, M. Braganza, J. Burnetie, T. Chang, J. Eble, R. Faber, O. Gowda, J. Grodstein, G. Hess, J. Kowaleski, A. Kumar, B. Miller, R. Mueller, P. Paul, J. Pickholtz, S. Russell, M. Shen, T. Truex, A. Vardharajan, D. Xanthopoulos, and T. Zou, "A 1.2 ghz alpha microprocessor with 44.8 gb/s chip pin bandwidth", in *Solid-State Circuits Conference, 2001. Digest of Technical Papers. ISSCC. 2001 IEEE International*, pp. 240–241, 5-7 Feb. 2001.

[26] P. J. M. John T. Feo and S. K. Skedzielewski, "Sisal90", *P. High Performance Functional Computing*, 1995.

[27] M. Kishi, H. Yasuhara, and Y. Kawamura, "Dddp-a distributed data driven processor", in *ISCA '83: Proceedings of the 10th annual international symposium on Computer architecture*, (Los Alamitos, CA, USA), pp. 236–242, IEEE Computer Society Press, 1983.

[28] T. Knight, "An architecture for mostly functional languages", in *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, (New York, NY, USA), pp. 105–112, ACM, 1986.

[29] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen, "Hybrid transactional memory", in *Proceedings of Symposium on Principles and Practice of Parallel Programming*, Mar 2006.

[30] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz, "Smart memories: a modular reconfigurable architecture", in *Computer Archi-*

tecture, 2000. Proceedings of the 27th International Symposium on, pp. 161–171, 2000.

[31] V. J. Marathe, W. N. Scherer III, and M. L. Scott, "Adaptive software transactional memory", in *Proceedings of the 19th International Symposium on Distributed Computing*, (Cracow, Poland), Sep 2005. Earlier but expanded version available as TR 868, University of Rochester Computer Science Dept., May2005.

[32] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott, "Lowering the overhead of software transactional memory", Tech. Rep. TR 893, Computer Science Department, University of Rochester, Mar 2006. Condensed version submitted for publication.

[33] A. McDonald, J. Chung, B. D. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun, "Architectural semantics for practical transactional memory", in *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, (Washington, DC, USA), pp. 53–65, IEEE Computer Society, 2006.

[34] J. R. McGraw, "The val language: Description and analysis", *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 1, pp. 44–82, 1982.

[35] M. Mercaldi, S. Swanson, A. Petersen, A. Putnam, A. Schwerin, M. Oskin, and S. J. Eggers, "Instruction scheduling for a tiled dataflow architecture", in *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 141–150, ACM Press, 2006.

[36] M. Mercaldi, S. Swanson, A. Petersen, A. Putnam, A. Schwerin, M. Oskin, and S. J. Eggers, "Modeling instruction placement on a spatial architecture", in *SPAA '06: Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, (New York, NY, USA), pp. 158–169, ACM Press, 2006.

[37] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "Logtm: Log-based transactional memory", in *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pp. 254–265, Feb 2006.

[38] J. E. B. Moss, "Open nested transactions: Semantics and support", in *WMPI*, (Austin, TX), February 2006.

[39] S. Murer and R. Marti, "The fool programming language: Integrating single-assignment and object-oriented paradigms", *European Workshop on Parallel Computing*, 1992.

[40] R. Nagarajan, K. Sankaralingam, D. Burger, and S. Keckler, "A design space evaluation of grid processor architectures", in *Microarchitecture, 2001. MICRO-34. Proceedings. 34th ACM/IEEE International Symposium on*, pp. 40–51, 1-5 Dec. 2001.

[41] R. Nikhil, "The parallel programming language id and its compilation for parallel machines", *Workshop on Mazzive Parallelis: Hardware, Programming and Applications*, 1990.

[42] K. Olukotun and L. Hammond, "The future of microprocessors", *Queue*, vol. 3, no. 7, pp. 26–29, 2005.

[43] G. M. Papadopoulos and D. E. Culler, "Monsoon: an explicit token-store architecture", in *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, (New York, NY, USA), pp. 82–91, ACM Press, 1990.

[44] D. A. Patterson and J. L. Hennessy, *Computer Architecture (3rd ed.): A Quantitative Approach.* Morgan Kaufmann Publishers Inc.

[45] D. A. Patterson and J. L. Hennessy, *Computer organization and design (3rd ed.): the hardware/software interface.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

[46] R. Rajwar and J. R. Goodman, "Transactional execution: Toward reliable, high-performance multithreading.", *IEEE Micro*, vol. 23, pp. 117–125, Nov-Dec 2003.

[47] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing transactional memory", in *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, (Washington, DC, USA), pp. 494–505, IEEE Computer Society, 2005.

[48] S. Rigo, P. Centoducatte, and A. Baldassin, "Memórias transacionais: Uma nova alternativa para programação concorrente", in *Minicursos do VIII Workshop em Sistemas Computacionais de Alto Desempenho, WSCAD 2007* [52].

[49] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg, "Mcrt-stm: a high performance software transactional memory system for a multi-core runtime", in *Proc. 11th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP '06)*, pp. 187–197, Mar 2006.

[50] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson, "Architectural support for software transactional memory", in *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, (Washington, DC, USA), pp. 185–196, IEEE Computer Society, 2006.

[51] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba, "An architecture of a dataflow single chip processor", in *Computer Architecture, 1989. The 16th Annual International Symposium on*, pp. 46–53, 28 May - 1 June, 1989.

[52] SBAC-PAD 2007, *Eletronic Proceedings of the 19th International Symposium on Computer Architecture and High Performance Computing*, 2007.

[53] N. Shavit and D. Touitou, "Software transactional memory", in *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, (New York, NY, USA), pp. 204–213, ACM, 1995.

[54] T. Shimada, K. Hiraki, K. Nishida, and S. Sekiguchi, "Evaluation of a prototype data flow processor of the sigma-1 for scientific computations", in *ISCA '86: Proceedings of the 13th annual international symposium on Computer architecture*, (Los Alamitos, CA, USA), pp. 226–234, IEEE Computer Society Press, 1986.

[55] J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek, "Multiple reservations and the oklahoma update", *IEEE Parallel Distrib. Technol.*, vol. 1, no. 4, pp. 58–71, 1993.

[56] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "Wavescalar", in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pp. 291–302, 2003.

[57] S. Swanson, A. Putnam, M. Mercaldi, K. Michelson, A. Petersen, A. Schwerin, M. Oskin, and S. Eggers, "Area-performance trade-offs in tiled dataflow architectures", in *Computer Architecture, 2006. 33rd International Symposium on*, pp. 314–326, 17-21 June 2006.

[58] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Osin, and S. Eggers, "The wavescalar architecture", *In submission to ACM Transactions on Computer Systems, TOCS*.

[59] S. Swanson, *The WaveScalar Architecture*. PhD thesis, University of Washington, 2006.

[60] R. M. Tomasulo, "An efficient algorithm for exploring multiple arithmetic units", *IBM Journal of Research and Development*, vol. 11, pp. 25–33, Jan 1967.

[61] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and

A. Agarwal, "Baring it all to software: Raw machines", *Computer*, vol. 30, pp. 86–93, Sept. 1997.

Apêndice A

Código fonte das aplicações utilizadas

A.1 Toy-Matrizes-SD e Small-Toy-Matrizes-SD

```
#define LIM 500 // para Toy-Matrizes-SD
#define LIM 50 // para Small-Toy-Matrizes-SD
int main()
{
     int m[LIM][2][2];
     int det[LIM];
     int v[LIM][2];
     int i;
     for (i=0; i<LIM; i++)
     {
          det[i]=(m[i][1][1]*m[i][2][2])
                 -(m[i][1][2]*m[i][2][1]);
          v[i][1]=m[i][1][1]+m[i][1][2];
          v[i][2]=m[i][2][1]+m[i][2][2];
     }
}
```

A.2 Toy-Matrizes-SD-Stores e Small-Toy-Matrizes-SD-Stores

```
#define LIM 500 // para Toy-Matrizes-SD-Stores
#define LIM 50 // para Small-Toy-Matrizes-SD-Stores
int main()
{
     int m[LIM][2][2];
     int det[LIM];
     int v[LIM][2];
     int i,j,k;
     for (i=0; i<LIM; i++)
          for (j=0; j<2; j++)
               for (k=0; k<2; k++)
                    m[i][j][k]=(k+j);
     for (i=0; i<LIM; i++)
     {
          det[i]=((m[i][1][1]*m[i][2][2])
                 -(m[i][1][2]*m[i][2][1]));
          v[i][1]=m[i][1][1]+m[i][1][2];
          v[i][2]=m[i][2][1]+m[i][2][2];
     }
}
```

A.3 Toy-Matrizes-SD-Stores-Min e Small-Toy-Matrizes-SD-Stores-Min

```
#define LIM 500 // para Small-Toy-Matrizes-SD-Stores-Min
#define LIM 50 // para Small-Toy-Matrizes-SD-Stores-Min
int main()
{
     int m[LIM][2][2];
     int det[LIM];
     int v[LIM][2];
     int i,j,k;
     for (i=0; i<LIM; i++)
          for (j=0; j<2; j++)
               for (k=0; k<2; k++)
                    m[i][j][k]=(k+j);
     for (k=0; k<3; k++)
     {
          for (i=0; i<LIM; i++)
          {
               det[i]=((m[i][1][1]*m[i][2][2])
                      -(m[i][1][2]*m[i][2][1]));
               v[i][1]=m[i][1][1]+m[i][1][2];
               v[i][2]=m[i][2][1]+m[i][2][2];
           }
     }
}
```

A.4 Toy-Matrizes-CD e Small-Toy-Matrizes-CD

```
#define LIM 500 // para Toy-Matrizes-CD
#define LIM 50 // para Small-Toy-Matrizes-CD
int main()
{
     int m[LIM][2][2];
     int det[LIM];
     int v[LIM][2];
     int i;
     for (i=0; i<LIM; i++)
     {
          if (i==250)  // para Toy-Matrizes-CD
          if (i==25)  // para Small-Toy-Matrizes-CD
          {
                det[i]=det[i-1]+((m[i][1][1]*m[i][2][2])
                      -(m[i][1][2]*m[i][2][1]));
          }
          else
          {
               det[i]=((m[i][1][1]*m[i][2][2])
                      -(m[i][1][2]*m[i][2][1]));
          }
          v[i][1]=m[i][1][1]+m[i][1][2];
          v[i][2]=m[i][2][1]+m[i][2][2];
     }
}
```

A.5 Toy-Matrizes-CD-Stores e Small-Toy-Matrizes-CD-Stores

```
#define LIM 500 // para Toy-Matrizes-CD-Store
#define LIM 50 // para Small-Toy-Matrizes-CD-Store
int main()
{
     int m[LIM][2][2];
     int det[LIM];
     int v[LIM][2];
     int i,j,k;
     for (i=0; i<LIM; i++)
          for (j=0; j<2; j++)
               for (k=0; k<2; k++)
                    m[i][j][k]=(k+j);
     for (i=0; i<LIM; i++)
     {
          if (i==250) // para Toy-Matrizes-CD-Store
          if (i==25) // para Small-Toy-Matrizes-CD-Store
          {
                det[i]=det[i-1]+((m[i][1][1]*m[i][2][2])
                      -(m[i][1][2]*m[i][2][1]));
          }
          else
          {
               det[i]=((m[i][1][1]*m[i][2][2])
                      -(m[i][1][2]*m[i][2][1]));
          }
          v[i][1]=m[i][1][1]+m[i][1][2];
          v[i][2]=m[i][2][1]+m[i][2][2];
     }
}
```

A.6 Toy-Matrizes-CD-Stores-Min e Small-Toy-Matrizes-CD-Stores-Min

```
#define LIM 500 // Toy-Matrizes-CD-Stores-Min
#define LIM 50 // Small-Toy-Matrizes-CD-Stores-Min
int main()
{
     int m[LIM][2][2];
     int det[LIM];
     int v[LIM][2];
     int i,j,k;
     for (i=0; i<LIM; i++)
          for (j=0; j<2; j++)
               for (k=0; k<2; k++)
                    m[i][j][k]=(k+j);
     for (k=0; k<10; k++)
     {
          for (i=0; i<LIM; i++)
          {
               if (i==250) // Toy-Matrizes-CD-Stores-Min
               if (i==25) // Small-Toy-Matrizes-CD-Stores-Min
               {
                    det[i]=det[i-1]+((m[i][1][1]*m[i][2][2])
                           -(m[i][1][2]*m[i][2][1]));
               }
               else
               {
                    det[i]=((m[i][1][1]*m[i][2][2])
                           -(m[i][1][2]*m[i][2][1]));
               }
               v[i][1]=m[i][1][1]+m[i][1][2];
               v[i][2]=m[i][2][1]+m[i][2][2];
          }
     }
}
```

A.7 Toy-Vetores-CD-WAR-WAW-RAW

```
#define LIM 500
int main()
{
     int v[LIM];
     int v2[LIM];
     int v3[LIM];
     int v4[LIM];
     int v5[LIM];
     int i,j;
     for (j=1; j<(LIM-3);j++)
     {
          v[j-1]=j;
          v2[j-1]=v2[j-1]+v2[j]+v2[j+1]+v2[j+2]+v2[j+3]
                  +v[j-1]+v[j]+v[j+1]+v[j+2]+v[j+3];
          v3[j-1]=v3[j-1]+v3[j]+v3[j+1]+v3[j+2]+v3[j+3]
                  +v2[j-1]+v2[j]+v2[j+1]+v2[j+2]+v2[j+3]
                  +v[j]+v[j+1]+v[j+2]+v[j+3];
          v4[j-1]=v4[j-1]+v4[j]+v4[j+1]+v4[j+2]+v4[j+3]
                  +v3[j-1]+v3[j]+v3[j+1]+v3[j+2]+v3[j+3]
                  +v2[j-1]+v2[j]+v2[j+1]+v2[j+2]+v2[j+3]
                  +v[j-1]+v[j]+v[j+1]+v[j+2]+v[j+3];
          v[0]=v4[250]+v5[j-1]+v5[j]+v5[j+1]+v5[j+2]+v5[j+3]
               +v4[j-1]+v4[j]+v4[j+1]+v4[j+2]+v4[j+3]
               +v3[j-1]+v3[j]+v3[j+1]+v3[j+2]+v3[j+3]
               +v2[j-1]+v2[j]+v2[j+1]+v2[j+2]+v2[j+3]
               +v[j-1]+v[j]+v[j+1]+v[j+2]+v[j+3];
     }
}
```

A.8 Small-Toy-Vetores-CD-WAR-WAW-RAW

```
#define LIM 50
int main()
{
     int v[LIM];
     int v2[LIM];
     int v3[LIM];
     int v4[LIM];
     int v5[LIM];
     int i,j;
     for (j=1; j<(LIM-3);j++)
     {
          v[j-1]=j;
          v2[j-1]=v2[j-1]+v2[j]+v2[j+1]+v2[j+2]+v2[j+3]
                  +v[j-1]+v[j]+v[j+1]+v[j+2]+v[j+3];
          v3[j-1]=v3[j-1]+v3[j]+v3[j+1]+v3[j+2]+v3[j+3]
                  +v2[j-1]+v2[j]+v2[j+1]+v2[j+2]+v2[j+3]
                  +v[j]+v[j+1]+v[j+2]+v[j+3];
          v4[j-1]=v4[j-1]+v4[j]+v4[j+1]+v4[j+2]+v4[j+3]
                  +v3[j-1]+v3[j]+v3[j+1]+v3[j+2]+v3[j+3]
                  +v2[j-1]+v2[j]+v2[j+1]+v2[j+2]+v2[j+3]
                  +v[j-1]+v[j]+v[j+1]+v[j+2]+v[j+3];
          v[0]=v4[25]+v5[j-1]+v5[j]+v5[j+1]+v5[j+2]+v5[j+3]
               +v4[j-1]+v4[j]+v4[j+1]+v4[j+2]+v4[j+3]
               +v3[j-1]+v3[j]+v3[j+1]+v3[j+2]+v3[j+3]
               +v2[j-1]+v2[j]+v2[j+1]+v2[j+2]+v2[j+3]
               +v[j-1]+v[j]+v[j+1]+v[j+2]+v[j+3];
     }
}
```