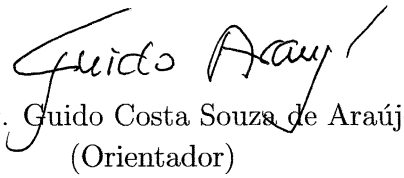
Instituto de Computação
Universidade Estadual de Campinas

Mecanismo para Execução Especulativa de Aplicações Paralelizadas por Técnicas DOPIPE Usando Replicação de Estágios

André Oliveira Loureiro do Baixo

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por André Oliveira Loureiro do Baixo e aprovada pela Banca Examinadora.

Campinas, 24 de julho de 2012.



Prof. Dr. Guido Costa Souza de Araújo
(Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA POR
MARIA FABIANA BEZERRA MULLER - CRB8/6162
BIBLIOTECA DO INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E
COMPUTAÇÃO CIENTÍFICA - UNICAMP

Baixo, André Oliveira Loureiro do, 1986-
B166m Mecanismo para execução especulativa de aplicações paralelizadas por técnicas DOPIPE usando replicação de estágios / André Oliveira Loureiro do Baixo. – Campinas, SP : [s.n.], 2012.

Orientador: Guido Costa Souza de Araújo.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Arquitetura de computador. 2. Processamento paralelo (Computadores). 3. Compiladores (Programas de computador). I. Araújo, Guido Costa Souza de,1962-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em inglês: Mechanism for speculative execution of applications parallelized by DOPIPE techniques using stage replication
Palavras-chave em inglês:
Computer architecture
Parallel processing (Electronic computers)
Compilers (Computer programs)
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Guido Costa Souza de Araújo [Orientador]
Maurício Breternitz Júnior
Rodolfo Jardim de Azevedo
Data de defesa: 24-07-2012
Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 24 de Julho de 2012, pela Banca examinadora composta pelos Professores Doutores:

Dr. Maurício Breternitz Jr.
AMD

Prof. Dr. Rodolfo Jardim de Azevedo
IC / UNICAMP

Prof. Dr. Guido Costa Souza de Araújo
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Mecanismo para Execução Especulativa de Aplicações Paralelizadas por Técnicas DOPIPE Usando Replicação de Estágios

André Oliveira Loureiro do Baixo[1]

Julho de 2012

Banca Examinadora:

- Prof. Dr. Guido Costa Souza de Araújo (Orientador)
- Dr. Maurício Breternitz Jr.
  Advanced Micro Devices, Austin, TX (AMD)
- Prof. Dr. Rodolfo Jardim de Azevedo
  Instituto de Computação, Universidade Estadual de Campinas (UNICAMP)
- Prof. Dr. Mário Lúcio Côrtes (Suplente)
  Instituto de Computação, Universidade Estadual de Campinas (UNICAMP)
- Prof. Dr. Alexandro Baldassin (Suplente)
  Instituto de Geociências e Ciências Exatas, Universidade Estadual de São Paulo (UNESP)

[1]Suporte financeiro de: Bolsa do CNPq (processo 135910/2009-9) 2009–2010 e Bolsa da FAPESP (processo 2010/02913-5) 2010–2011.

Resumo

A utilização máxima dos núcleos de arquiteturas multi-processadas é fundamental para permitir uma utilização completa do paralelismo disponível em processadores modernos. A fim de obter desempenho escalável, técnicas de paralelização requerem um ajuste cuidadoso de: (a) mecanismo arquitetural para especulação; (b) ambiente de execução; e (c) transformações baseadas em *software*. Mecanismos de *hardware* e *software* já foram propostos para tratar esse problema. Estes mecanismos, ou requerem alterações profundas (e arriscadas) nos protocolos de coerência de cache, ou exibem uma baixa escalabilidade de desempenho para uma gama de aplicações. Trabalhos recentes em técnicas de paralelização baseadas em DOPIPE (como DSWP) sugerem que a combinação de versionamento de dados baseado em paginação com especulação em *software* pode resultar em bons ganhos de desempenho. Embora uma solução apenas em *software* pareça atrativa do ponto de vista da indústria, essa não utiliza todo o potencial da microarquitetura para detectar e explorar paralelismo. A adição de *tags* às caches para habilitar o versionamento de dados, conforme recentemente anunciado pela indústria, pode permitir uma melhor exploração de paralelismo no nível da microarquitetura. Neste trabalho, é apresentado um modelo de execução que permite tanto a especulação baseada em DOPIPE, como as técnicas de paralelização especulativas tradicionais. Este modelo é baseado em uma simples abordagem com *tags* de cache para o versionamento de dados, que interage naturalmente com protocolos de coerência de cache tradicionais, não necessitando que estes sejam alterados. Resultados experimentais, utilizando *benchmarks* SPEC e PARSEC, revelam um ganho de desempenho geométrico médio de 21.6× para nove programas sequenciais em uma máquina simulada de 24 núcleos, demonstrando uma melhora na escalabilidade quando comparada a uma abordagem apenas em *software*.

Abstract

Maximal utilization of cores in multicore architectures is key to realize the potential performance available from modern microprocessors. In order to achieve scalable performance, parallelization techniques rely on carefully tunning speculative architecture support, runtime environment and software-based transformations. Hardware and software mechanisms have already been proposed to address this problem. They either require deep (and risky) changes on the existing hardware and cache coherence protocols, or exhibit poor performance scalability for a range of applications. Recent work on DOPIPE-based parallelization techniques (e.g. DSWP) has suggested that the combination of page-based data versioning with software speculation can result in good speed-ups. Although a software-only solution seems very attractive from an industry point-of-view, it does not enable the whole potential of the microarchitecture in detecting and exploiting parallelism. The addition of cache tags as an enabler for data versioning, as recently announced in the industry, could allow a better exploitation of parallelism at the microarchitecture level. In this paper we present an execution model that supports both DOPIPE-based speculation and traditional speculative parallelization techniques. It is based on a simple cache tagging approach for data versioning, which integrates smoothly with typical cache coherence protocols, and does not require any changes to them. Experimental results, using SPEC and PARSEC benchmarks, reveal a geometric mean speedup of 21.6x for nine sequential programs in a 24-core simulated CMP, while demonstrate improved scalability when compared to a software-only approach.

Agradecimentos

Agradeço primeiramente a Deus por todo o suporte e direção que me provê a cada dia.

Aos meus pais, Antônio e Waléria, e ao meu irmão Alexandre, por todo o apoio durante esses anos de mestrado.

Ao meu orientador, professor Guido, pelo direcionamento, paciência, dedicação, compreensão, amizade e ajuda durante todo o tempo em que trabalhamos juntos.

Ao amigo João Paulo Porto pela paciência, pela companhia durante alguns meses de trabalho e pelo conhecimento que me transmitiu.

Aos inúmeros amigos que fizeram parte desta caminhada, e fizeram com que a jornada fosse mais leve e prazerosa.

Ao CNPq e à FAPESP pelo auxílio financeiro que tornou este trabalho possível.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

O consumo de potência e a dissipação de calor, na última década, fizeram com que a Lei de Moore [25] deixasse de valer para a frequência de *clock* nos processadores modernos (*circa* 2012). No entanto, a demanda por desempenhos cada vez maiores continua. Por outro lado, o número de transistores por unidade de área continuou dobrando, de acordo com a Lei de Moore. Assim, a indústria de processadores, que até então fazia uso do aumento da freqência de *clock* para melhorar o desempenho das aplicações, passou a utilizar o maior número de transistores disponíveis no projeto de mais núcleos de processamento.

A presença de núcleos adicionais torna possível a execução simultânea real (sem *time sharing*) de processos. Portanto, um processador que tenha uma carga de quatro processos pesados em termos de consumo de CPU (CPU-*bound*) poderia, a princípio, ter seu desempenho melhorado em até quatro vezes na presença de três núcleos adicionais, já que cada um dos processos poderia ser executado em um núcleo separadamente. Contudo, uma aplicação convencional, cujo código não leva em consideração a presença de mais elementos de processamento, não terá benefício algum proveniente do aumento do número de núcleos. Desta forma, nenhuma aplicação escrita até o advento de processadores com múltiplos núcleos iria experimentar qualquer aumento de desempenho.

A escrita de aplicações científicas paralelas eficientes é geralmente facilitada pela estrutura regular do fluxo de controle e dos padrões de acesso à memória que possuem. Tais fatores fazem também com que muitas aplicações científicas sequenciais já existentes possam ser paralelizadas através do uso de técnicas convencionais de paralelização como DOALL e DOACROSS [2], gerando programas paralelos eficientes.

Por outro lado, aplicações de propósito geral exibem fluxos de controle complicados, padrões irregulares de acesso à memória e dependências que não podem ser resolvidas estaticamente. Isso faz com que a paralelização de tais programas sequenciais resulte em códigos paralelos que subutilizam a capacidade de processamento dos múltiplos núcleos. Por outro lado, a escrita de novas aplicações paralelas de propósito geral é muito difícil,

passível de erros e difícil de depurar, mesmo fazendo uso de bibliotecas que auxiliam os processos de paralelização [12, 32] e depuração [21].

Com o intuito de prover um processo de paralelização automática para aplicações de propósito geral, foram criadas técnicas DOPIPE, tais como *Decoupled Software Pipeline* (DSWP) [27] e *Parallel Section Decoupled Software Pipeline* (PS-DSWP) [30]. Tais métodos procuram escalonar diferentes partes do código de laços em diferentes núcleos de processamento, de forma a criar um modelo de execução similar a um *pipeline*, no qual cada núcleo corresponde a um estágio do suposto *pipeline*. Tais técnicas foram implementadas em compiladores [27] e forneceram bons ganhos de desempenho para aplicações de propósito geral executadas em máquinas de até quatro núcleos, não apresentando ganhos de desempenho significativos para um número maior de núcleos.

Por outro lado, levando em consideração o aumento do número de transistores, e o consequente aumento no número de núcleos de processamento, máquinas de 8 ou 24 *cores* já são uma realidade. Com isto, tornou-se necessária a criação de novos métodos de paralelização de aplicações de propósito geral, ou a adaptação de técnicas já existentes, a fim de obter maior desempenho das máquinas com mais núcleos.

Estudos anteriores [7, 40] mostraram que a incapacidade da técnica PS-DSWP de prover paralelismo escalável deve-se à impossibilidade de determinar estaticamente se algumas dependências realmente ocorrem em tempo de execução. Como exemplo podemos citar apontadores que podem ou não apontar para a mesma posição de memória em determinados pontos da execução do programa (*alias problem*). Tais informações podem ser dependentes da entrada do programa, por exemplo, impossibilitando uma análise exata em tempo de compilação.

A impossibilidade de determinar com exatidão a ocorrência de dependências faz com que o compilador seja conservativo, assumindo que as dependências duvidosas *ocorrem* durante a execução. Se por um lado tal procedimento garante a geração de um código correto, por outro a quantidade de paralelismo extraída pode ser diminuída significativamente. Desta forma, o compilador poderia aumentar o paralelismo do código gerado se a suposição de que tais dependências (ou uma parte delas) não ocorrem durante a execução. Contudo, para que o código gerado seja correto, o compilador deve assumir que o ambiente do qual a aplicação será executada é capaz de detectar e corrigir possíveis violações de dependência que venham a ocorrer. O procedimento que assume que dependências duvidosas não ocorrem durante a execução é chamado de *especulação*.

A incorporação de especulação à técnica PS-DSWP [7, 40] mostrou que é possível extrair paralelismo escalável de aplicações de propósito geral para computadores com vários elementos de processamento. Do mesmo modo, o uso de especulação combinado com as técnicas tradicionais de paralelização DOALL e DOACROSS provê ganhos de desempenho escaláveis para aplicações científicas [2]. Tais combinações são conhecidas

como *Thread Level Speculation* (TLS).

Visando viabilizar técnicas especulativas que permitem o compilador extrair paralelismo escalável, é necessário a utilização de mecanismos em *hardware* e *software*. Conforme citado acima, tais mecanismos devem ser capazes de detectar e corrigir violações de dependência que venham a ocorrer durante a execução do programa.

Com o intuito de permitir a execução de aplicações paralelizadas através das técnicas tradicionais de TLS, várias soluções em *hardware* foram propostas [10, 15, 39]. Tais mecanismos detectam violações de memória em tempo de execução e desfazem as operações realizadas em paralelo até um certo ponto da execução do programa caso alguma violação seja detectada.

Entretanto, estes mecanismos resultam em complicações adicionais à arquitetura existente, tais como:

- Adição e dependência de estruturas centralizadas em *hardware* que precisam ser acessadas durante a execução de *loads* e *stores*, gerando contenção (o que pode afetar o desempenho negativamente).

- Geração de tráfico adicional em rajada, aumentando significativamente o consumo de energia [33] e promovendo contenção.

- Modificações profundas e custosas nas caches e nos protocolos de coerência, tornando difícil a adoção de tais mecanismos pela indústria.

Além disso, mecanismos tradicionais de *hardware* que viabilizam TLS são incapazes de viabilizar a execução de aplicações paralelizadas através de técnicas DOPIPE aliadas a especulação de dependências, podendo gerar resultados incorretos e execuções ineficientes. Assim, embora tais mecanismos sejam adequados àqueles programas que podem ser eficientemente paralelizados pelas técnicas tradicionais de TLS, a maioria das aplicações de próposito geral (que necessita da combinação de técnicas DOPIPE com especulação para obter desempenho escalável com o número de *cores*) permanecem confinadas aos desempenhos providos por máquinas de até quatro núcleos.

Com o intuito de prover um mecanismo que possibilitasse um desempenho escalável para aplicações de propósito geral em máquinas com mais núcleos, foi proposto um mecanismo em *software* chamado SMTX [29]. Através do uso de uma unidade central responsável por detectar conflitos entre acessos à memória feitos por todos os núcleos, tal sistema permite a execução de aplicações paralelizadas por técnicas DOPIPE combinadas com especulação. O objetivo é fazer com que os núcleos reportem suas leituras e escritas para essa unidade central, que por sua vez re-executa tais leituras e escritas realizadas por todos os núcleos. Contudo, conforme mostrado em [29], o desempenho dessa unidade centralizada e o fato de todos os núcleos terem que enviar seus acessos à memória para

esta unidade formam um gargalo durante a execução de várias aplicações, impactando drasticamente a escalabilidade do desempenho dos programas paralelizados.

Em contra partida, através da interação com o protocolo de coerência de cache já existente nas arquiteturas atuais, o esquema proposto neste trabalho não depende de nenhuma unidade centralizada. A vantagem dessa abordagem descentralizada fica evidente quando é realizada uma comparação entre o desempenho das aplicações que são críticas para o SMTX com o desempenho das mesmas aplicações no sistema aqui proposto. Em particular, a diferença é dramática no caso de aplicações que saturam a unidade central do SMTX ou o seu sistema de memória. Conforme será mostrado na seção experimental (Capítulo 4), mesmo para as aplicações que não são críticas para o SMTX, o mecanismo aqui proposto provê melhores ganhos de desempenho.

Neste trabalho é proposta a adição de três *tags* simples às linhas de cache para tornar possível a execução distribuída, eficiente e escalável de códigos paralelizados tanto pelas técnicas tradicionais de TLS, como também por técnicas DOPIPE combinadas com especulação e replicação de estágios, provendo assim desempenho escalável não apenas para aplicações científicas, mas também para aplicações de propósito geral. O impacto das *tags* adicionais é pequeno e pode ser amortizado, já que o número de transistores disponíveis continua a crescer com a tecnologia. Por exemplo, a IBM anunciou recentemente que seu processador *BlueGene/Q* possui *tags* de cache combinadas com uma grande quantidade de lógica adicional a fim de criar caches com múltiplas versões de dados. Isso sugere que o impacto de *tags* de cache pode ser pequeno e que manter múltiplas versões de dados nas caches é viável. Assim, como já existe uma máquina com as mesmas características necessárias para a implementação do mecanismo aqui proposto, tudo indica que nossa técnica pode ser implementada na prática.

Conforme será explicado adiante, nosso sistema é imune à degradação da escalabilidade de desempenho presente na técnica em *software* citada anteriormente. Além disso, nosso sistema apresenta vantagens significativas em relação aos mecanismos já propostos em *hardware* para TLS:

- Nosso mecanismo interage naturalmente com os protocolos de coerência de cache já presentes nas máquinas atuais, não requerendo nenhuma alteração em relação ao protocolo ou ao seu *hardware*.

- O sistema aqui proposto possibilita a execução eficiente de códigos paralelizados através de técnicas DOPIPE com replicação de estágios e especulação, provendo assim desempenho escalável para aplicações de propósito geral. Nosso sistema também permite a execução de programas paralelizados por meio das técnicas tradicionais de TLS (DOALL e DOACROSS combinados com especulação).

- A diferença entre o *hardware* de máquinas já existentes e aquele presente na nossa abordagem é pequena. O processador *BlueGene/Q* mencionado acima já possui *tags* de cache combinadas com uma lógica adicional com o intuito de prover caches com múltiplas versões de dados. Isso é o necessário para viabilizar o mecanismo aqui proposto, já que este já interage com os protocolos de coerência de cache existentes.

- Nosso sistema resolve a maioria dos problemas encontrados nas abordagens de suporte a TLS já propostas, como geração de tráfego em rajada, introdução de novos protocolos de coerência de cache e dependência em relação a *hardwares* e estruturas especiais centralizadas que precisam ser acessadas durante leituras e escritas, causando degradação de desempenho e consumo adicional de energia.

As *tags* aqui introduzidas são gerenciadas através de uma lógica simples e eficiente que não afeta o protocolo de coerência de cache. Tais *tags* também habilitam, de forma eficiente, a detecção de violações de consistência sequencial durante a execução de códigos paralelizados com especulação, permitindo também operações de *commit* e squash eficientes.

A fim de avaliar o impacto da técnica aqui proposta, bem como para comparar seu desempenho com o de outros esquemas, foram utilizadas nove aplicações, incluindo *benchmarks* das suítes SPEC e PARSEC. Para garantir a corretude do mecanismo, foi desenvolvida uma prova formal do seu funcionamento, que se encontra no apêndice A.

Esta dissertação está organizada da seguinte forma. O capítulo 2 introduz algumas técnicas de paralelização pertinentes a esse trabalho e fornece a motivação para o suporte a aplicações paralelizadas por técnicas DOPIPE combinadas com replicação de estágios e especulação. O capítulo 3 descreve nosso modelo arquitetural e mostra como este permite a execução correta das aplicações paralelizadas. Este capítulo fornece ainda exemplos que descrevem como o nosso mecanismo interage naturalmente com os protocolos de coerência de cache já existentes e como as violações são detectadas. Os resultados experimentais da técnica proposta são apresentados no capítulo 4. O capítulo 5 traz uma revisão dos principais trabalhos na literatura relacionados à nossa abordagem e o capítulo 6 conclui a dissertação.

Capítulo 2

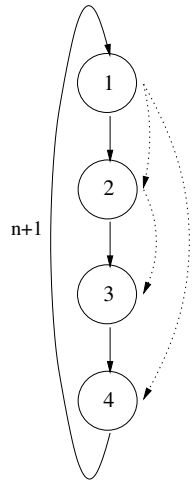
Técnicas de Paralelização

Este capítulo descreve uma série de técnicas de paralelização que vem sendo utilizadas com o intuito de tirar proveito da capacidade de máquinas com múltiplos elementos de processamento. Conforme será explicado neste capítulo, a combinação de tais técnicas com especulação pode ser capaz de extrair ainda mais paralelismo de aplicações sequenciais, exigindo no entanto, mecanismos em *hardware* e/ou *software* para garantir que a execução seja correta.

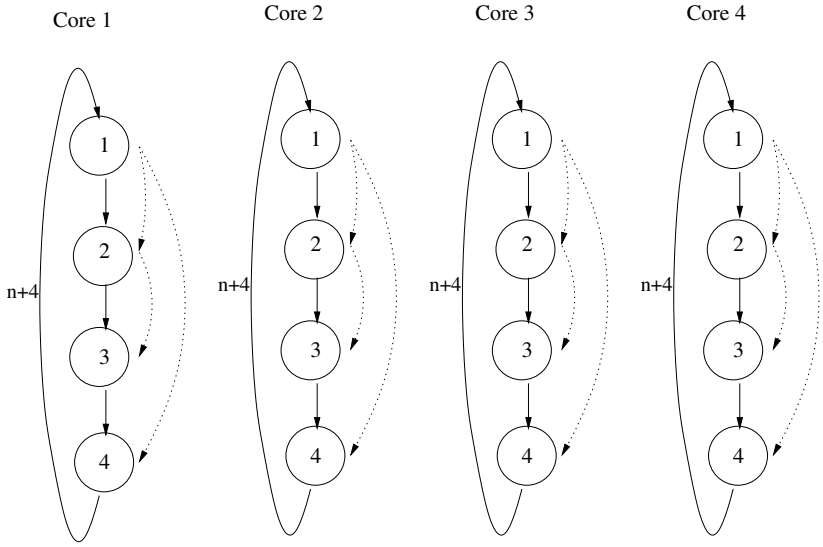
Considere o fragmento de programa mostrado na figura 2.1. Supondo que a função `f` não introduz nenhuma dependência entre as iterações (por exemplo, modificando `p`), pode-se concluir que as iterações são independentes e podem ser processadas em paralelo. Esta técnica de paralelização é chamada DOALL. Como exemplo, a figura 2.2(a) ilustra a execução sequencial de um laço sem dependências entre as iterações. Nesta figura, as setas cheias representam o fluxo de execução, e as setas pontilhadas representam dependências de dados. A figura 2.2(b) ilustra a execução paralela deste mesmo laço utilizando a técnica DOALL em um sistema com quatro *cores*. Repare que, como não existem dependências entre as iterações do laço, cada iteração pode ser executada em um *core* isoladamente. Para isso, basta que $n + 1$ seja transformado em $n + 4$ (para quatro *cores*), onde n é a variável de indução do laço. Da mesma forma, se esse laço fosse executado em um número maior de *cores*, as instruções do laço seriam simplesmente replicadas em todos os *cores*, ajustando apenas a variável de indução de acordo com o número de elementos de processamento.

```
int *p = &y[7];
for (i=0; i < N; i++)
  x[i] += f(p);
```

Figura 2.1: Exemplo de código para paralelização DOALL.



(a) Execução sequencial.

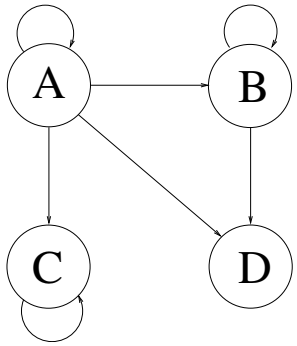


(b) Execução paralela DOALL.

Figura 2.2: Paralelização DOALL.

```
A: while (condition)
   {
     statement B;
     statement C;
     statement D;
   }
```

(a) Exemplo de laço para paralelização DOACROSS.



(b) Grafo de dependências do laço mostrado em (a).

Figura 2.3: O laço mostrado em (a) possui o Grafo de Dependências de Programa mostrado em (b). Este laço pode ser paralelizado tanto pela técnica DOACROSS como pela técnica DSWP.

No entanto, para muitos programas, a premissa de que `f` não introduz dependências entre iterações não é válida, impossibilitando o compilador de aplicar a transformação DOALL. Por outro lado, se as dependências introduzidas por `f` não se manifestarem em tempo de execução, então a técnica DOALL ainda poderia ser aplicada a fim de extrair paralelismo. Contudo, não é possível em geral determinar estaticamente se tais dependências irão ocorrer durante a execução da aplicação. Contudo, se o ambiente de execução é capaz de detectar e corrigir violações de dependências, o compilador pode especular que as dependências entre iterações, tais como aquelas introduzidas por `f`, não se manifestam na execução, possibilitando assim a paralelização utilizando DOALL. Essa técnica, que combina DOALL com especulação, é chamada DOALL especulativo [31].

Ainda que várias aplicações científicas exibem tais laços regulares, aplicações de propósito geral normalmente contém fluxos de controle e padrões de acesso à memória irregulares. Por exemplo, a figura 2.3(b) representa o grafo de dependências de programa do laço simples mostrado na figura 2.3(a). A aplicação de DOALL especulativo neste laço iria ignorar todas as dependências entre iterações, provavelmente criando uma alta taxa de violações durante a execução do programa.

A figura 2.4 mostra a paralelização DOALL especulativa do programa mostrado na figura 2.1, especulando que não existem dependências entre as iterações introduzidas por `p`. No entanto, repare que, caso `p` seja modificado na iteração zero, por exemplo, as iterações seguintes podem ler um valor incorreto de `p`; basta que `p` seja lido na iteração um antes que a modificação feita na iteração zero se torne visível. Note que a paralelização DOALL especulativa nada mais é que a aplicação da técnica DOALL ignorando possíveis dependências (nesse exemplo, especulando que `p` não introduz dependências en-

Core 0

```
for (i=0; i < N; i += 4)
  x[i] += f(p);
```

Core 1

```
for (i=1; i < N; i += 4)
  x[i] += f(p);
```

Core 2

```
for (i=2; i < N; i += 4)
  x[i] += f(p);
```

Core 3

```
for (i=3; i < N; i += 4)
  x[i] += f(p);
```

Figura 2.4: Paralelização especulativa DOALL do laço mostrado na figura 2.1.

tre iterações).

Por outro lado, o método de paralelização DOACROSS poderia ser utilizado, resultando no escalonamento de execução mostrado na figura 2.5. A técnica DOACROSS escalona cada iteração do laço em *threads* alternadas, de forma que as dependências entre iterações sejam comunicadas entre as *threads*. Tal comunicação é normalmente realizada através da memória, utilizando sincronização. A figura 2.5 mostra, no quadro da esquerda, a paralelização DOACROSS do exemplo mostrado na figura 2.3. Na figura 2.5, A1 representa a *statement* A da figura 2.3 na iteração um. Da mesma forma, B2 representa a *statement* B na iteração dois e assim por diante.

Infelizmente, escalonar cada iteração para uma *thread* faz com que a latência de comunicação entre as *threads* seja colocada no caminho crítico da execução, tornando a técnica DOACROSS sensível à eficiência do mecanismo de comunicação disponível. Este problema pode ser percebido na figura 2.5. Repare que, na paralelização DOACROSS, A2 poderia ser executado no ciclo posterior à execução de A1. Contudo, como A2 está escalonado em um outro *core*, este tem que esperar que a dependência produzida por A1 no *core* 1 chegue ao *core* 2 para começar a execução de A2.

Por fim, muitas vezes a utilização da técnica DOACROSS requer que os laços sejam contados, que operem apenas em vetores, que tenham padrões regulares de acesso à memória, ou que não possuam fluxo de controle (ou que este seja muito simples) [9, 17].

2.1 *Decoupled Software Pipeline* (DSWP)

A fim de remover a latência de comunicação entre *threads* do caminho crítico de execução, eliminando assim a dependência em relação à eficiência do mecanismo de comunicação e melhorando o desempenho das aplicações de propósito geral paralelizadas, foi proposta a técnica *Decoupled Software Pipeline* (DSWP) [27]. Além disso, tal técnica não apresenta

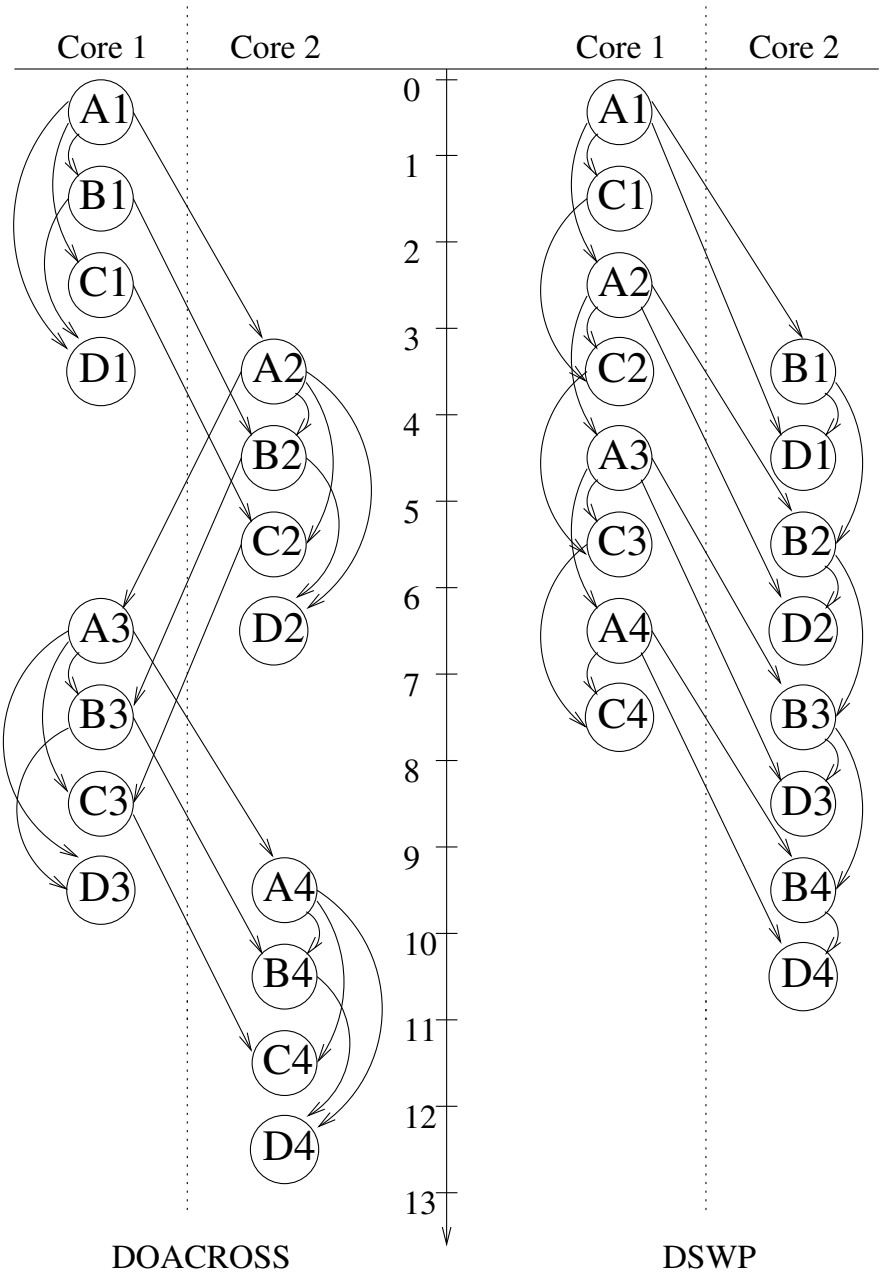


Figura 2.5: Aplicação de DOACROSS e DSWP no laço mostrado na figura 2.3. A latência de comunicação é de três ciclos.

as demais limitações do método DOACROSS mencionadas na seção anterior.

Ao contrário da técnica DOACROSS, o método DSWP escalona diferentes partes do código do laço a ser paralelizado para diferentes *threads*, conforme mostrado na figura 2.5. Repare na figura que, enquanto a técnica DOACROSS exige que todas as instruções de uma iteração completa do laço sejam colocadas na mesma *thread*, o método DSWP permite que instruções de uma mesma iteração sejam colocadas em *threads* diferentes. Assim, note que na figura 2.5, as instruções A, B, C e D da primeira iteração do laço da figura 2.3(a) (A1, B1, C1 e D1) são escalonadas em conjunto para o *core* 1 pela técnica DOACROSS. Da mesma forma, as instruções A, B, C e D da segunda iteração (A2, B2, C2 e D2) são escalonadas em conjunto para o *core* 2, e assim por diante.

Por outro lado, repare que, na figura 2.5, a técnica DSWP escalona a instrução A da iteração um (A1) para o *core* 1, ao passo que a instrução B também da iteração um (B1) é escalonada para o *core* 2. Caso exista uma dependência entre duas instruções X e Y e estas estejam em *threads* distintas, a *thread* que executa Y fica parada até que a dependência de X seja calculada e transmitida para a *thread* que executa Y. Por exemplo, como existe uma dependênca da instrução A para a instrução B no laço da figura 2.3, repare que, na figura 2.5, a técnica DSWP faz com que o *core* 2 fique parado até que a dependência de A1 chegue para que então seja possível executar B1. Para a realização da comunicação de dependências entre *cores*, a implementação da técnica DSWP [27] utiliza uma fila de comunicação em *hardware* entre os *cores*.

Por fim, repare na figura 2.5 como a latência de comunicação entre as *threads* afeta negativamente o desempenho das aplicações paralelizadas através da técnica DOACROSS. Como A2 depende de A1, o segundo núcleo precisa interromper a execução e ficar esperando o dado proveniente do primeiro núcleo. Depois disso, o mesmo acontece entre A2 e A3, mas na direção contrária. Isto faz com que, a cada iteração executada, um dos *cores* fique ocioso por dois ciclos, à espera de dados. Na figura 2.5 o *core* 2 fica parado, enquanto o *core* 1 executa B1 e C1, porque precisa esperar por uma dependência para poder executar A2. Na próxima iteração, é o *core* 1 que pára enquanto o *core* 2 executa B2 e C2. Isso continua acontecendo para todas as iterações até o fim da execução do laço. Logo, a cada iteração executada, a técnica DOACROSS desperdiça dois ciclos.

Já na técnica DSWP, a instrução A é sempre escalonada na mesma *thread*, evitando assim a formação de um ciclo de comunicação entre os *cores*. Conforme veremos mais adiante, o método DSWP gera sempre um fluxo de comunicação *unidirecional* entre os *cores*.

Com o fluxo unidirecional, a técnica DSWP permite um escalonamento de execução em forma de *pipeline*, no qual os *cores* são os elementos de execução. Repare que, na figura 2.5, os *cores* 1 e 2 podem ser vistos como um *pipeline* de dois estágios (a comunicação é sempre unidirecional do *core* 1 para o *core* 2). A vantagem do modelo de execução em *pipeline*

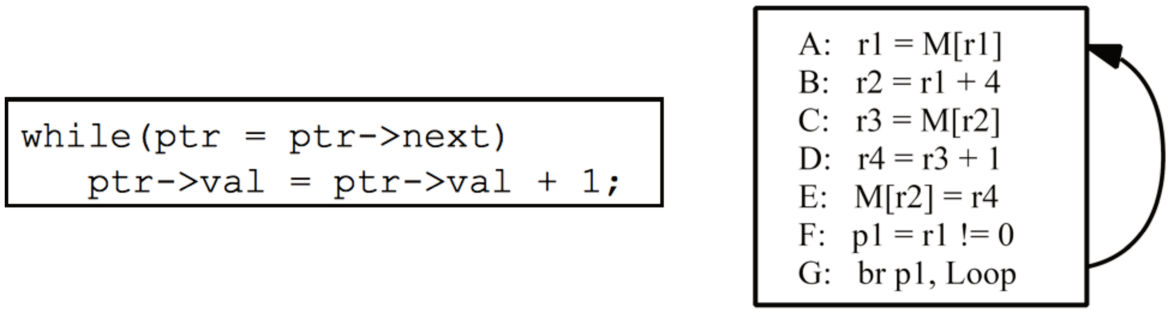


Figura 2.6: Exemplo de travessia de lista ligada a ser paralelizada usando DSWP.

pode ser observado na figura 2.5. Note que, após o terceiro ciclo, o *pipeline* fica cheio, evitando assim que os *cores* fiquem parados à espera de dependências, como acontece a cada iteração para a técnica DOACROSS, conforme explicado acima. Desta forma, os *cores* só precisam ficar parados enquanto o *pipeline* está sendo cheio (no início da execução do laço) e esvaziado (no final da execução do laço).

2.1.1 O Algoritmo DSWP

O processo de encontrar um escalonamento DSWP válido é descrito pela sequência de passos abaixo. Para isto, considere o laço mostrado na figura 2.6 que é responsável por percorrer uma lista ligada incrementado de um o campo `val` de cada elemento da lista. A representação em linguagem de montagem deste laço pode ser vista ao lado da mesma figura.

1 - Construir o grafo de dependências do programa

O primeiro passo do algoritmo DSWP consiste em construir o grafo de dependências do programa. Neste grafo, cada vértice representa uma instrução do laço e cada aresta representa uma dependência, sendo que a instrução correspondente ao vértice de origem da aresta deve ser executada antes da instrução correspondente ao vértice de destino desta aresta. Este grafo deve ser conservativo, incluindo dependências nos casos em que suas ausências não puderem ser asseguradas. O grafo de dependências do programa deve conter todas as dependências de dados, controle e memória presentes tanto dentro de uma mesma iteração como entre iterações diferentes.

O grafo correspondente ao laço da figura 2.6 pode ser visto na figura 2.7. Neste grafo, as dependências de dados são representadas por arestas verdes, as dependências de controle por arestas vermelhas e as dependências de memória por arestas alaranjadas. Além disso, as arestas correspondentes a dependências de dados estão rotuladas com o registrador que contém o dado e as arestas de controle e memória não possuem rótulo. As

arestas tracejadas indicam dependências entre iterações e as arestas sólidas representam dependências dentro de uma mesma iteração.

2 - Encontrar as componentes fortemente conexas do grafo criado

O segundo passo consiste em encontrar as componenentes fortemente conexas do grafo de dependências criado. Repare que cada uma dessas componentes representa um conjunto de instruções que fazem parte de um ciclo de dependências. Como a técnica DSWP exige que o fluxo de comunicação seja unidirecional (para a formação do *pipeline* de execução), as instruções pertencentes a uma determinada componente fortmente conexa devem ser escalonadas na mesma *thread*. Logo, se apenas uma componente fortemente conexa for encontrada, o código não é passível de paralelização por meio de DSWP e o algoritmo termina.

3 - Fazer a junção de componentes

Em seguida, as componentes encontradas devem ser unidas a fim de formar o grafo de componentes fortemente conexas. Tal grafo, correspondente ao exemplo da figura 2.7 pode ser visto na figura 2.8.

4 - Determinar as partições

No quarto passo, são determinadas quais componentes fortemente conexas irão ser executadas por cada *thread*. O conjunto de componentes destinado a uma determinada *thread* é denominado uma *partição*. Um conjunto de partições válidas deve obedecer as seguintes regras:

- O número de partições deve ser maior que um (uma partição significa apenas uma *thread*, ou seja, a aplicação não foi paralelizada) e menos que o número de *threads* que podem ser executadas concorrentemente no sistema.
- Cada componente fortemente conexa pertence a exatamente uma partição.
- Para cada aresta no grafo de componentes fortemente conexas conectando as componentes X e Y, existem apenas duas possibilidades:
  - X deve pertencer à mesma partição de Y, ou
  - X deve pertencer a uma partição que tenha sido formada antes da partição que contém Y.

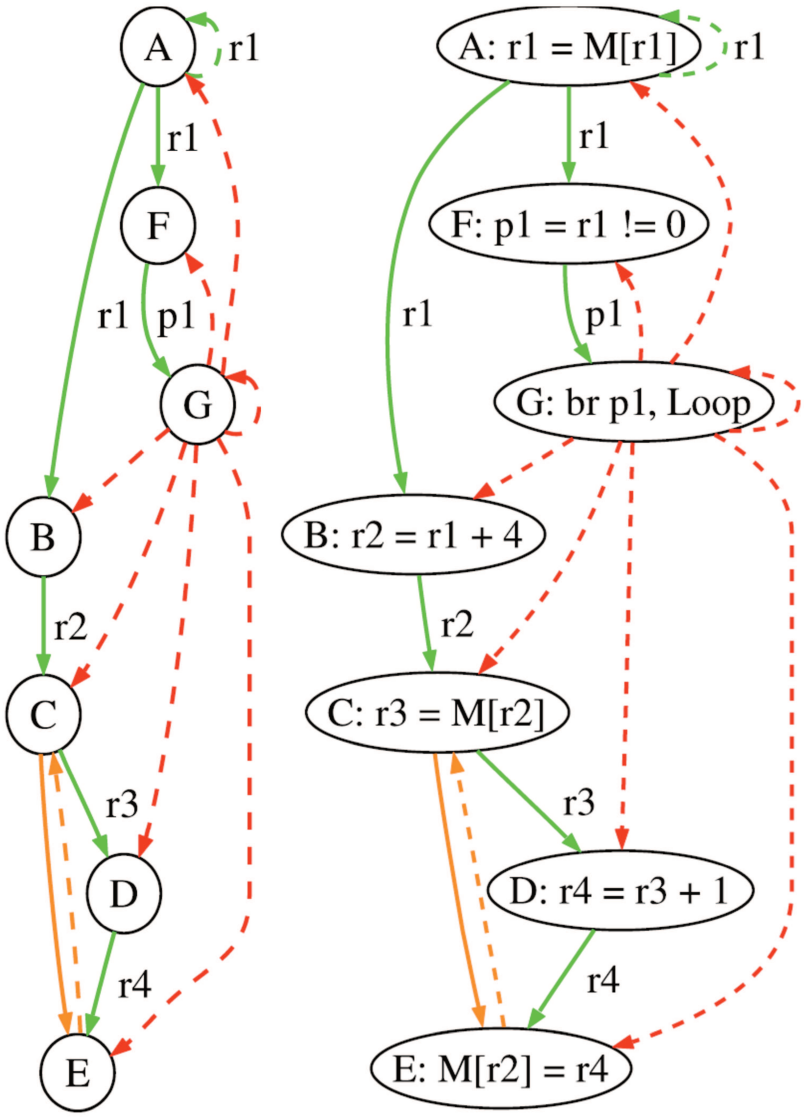


Figura 2.7: Grafo de dependências do laço presente na figura 2.6.

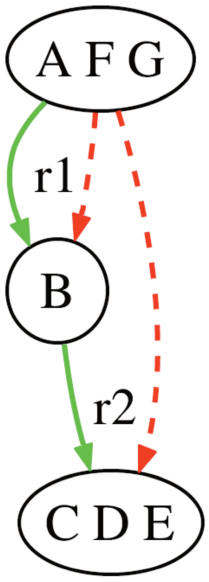


Figura 2.8: Componentes fortemente conexas do grafo da figura 2.7.

Se as duas componentes (origem e destino) pertencem à mesma partição, a comunicação de dependências entre elas fica confinada a uma comunicação local (dentro da mesma *thread*). Se a componente de origem pertence a uma partição que foi formada antes da partição que contém a componente de destino, a comunicação de dependências é sempre unidirecional, garantindo a formação de um *pipeline*. A garantia do fluxo unidirecional de dependências entre as partições pode ser percebido da seguinte maneira. Para a primeira partição, se alguma componente necessita de alguma dependência, esta deve ser produzida também na primeira partição, já que não existe partição que tenha sido formada antes da primeira. Para a segunda partição, se alguma componente precisa de alguma dependência, esta deve ter sido produzida na segunda ou na primeira partição. Procedendo da mesma maneira para as demais partições, nota-se que o fluxo de comunicação de dependências está sempre na mesma direção.

A figura 2.9 mostra como o *pipeline* de execução é formado a partir das partições formadas neste passo. Supondo que foram formadas duas partições P e Q (P1 é a partição P na iteração um, P2 a partição P na iteração dois e assim por diante; o mesmo vale para Q), e que a comunicação se dá no sentido de P para Q, repare que basta escalonar as partições para os *cores* de acordo com a ordem do fluxo de dependências entre as partições. Comunicações entre as iterações da partição P são satisfeitas trivialmente porque estão no mesmo *core*. Por outro lado, quando a partição Q precisar de algum dado produzido pela partição P, haverá uma instrução introduzida na partição Q (conforme será explicado

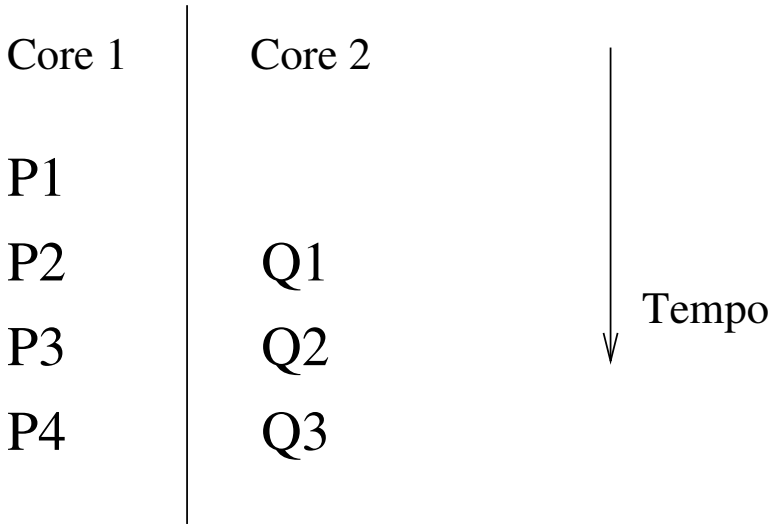


Figura 2.9: Formação do *pipeline* de execução a partir das partições.

adiante) que pára a execução de Q até que o dado chegue do *core* 1. Assim, as *threads* podem ser executadas de forma independente. De uma forma geral, se temos n partições $P_1, P_2, \ldots, P_n$, n cores e o fluxo de comunicação se dá da esquerda para a direita, basta escalonar P_1 para o *core* 1, P_2 para o *core* 2 e assim por diante.

Assim, supondo que o sistema para o qual o código está sendo compilado pode executar duas *threads* simultaneamente, devemos formar duas partiçoes (primeira condição acima). Utilizando o grafo de componentes fortemente conexas da figura 2.8 como exemplo, poderíamos formar uma partição com as componentes AFG e B e outra partição com a componente CDE. Alternativamente, poderíamos formar uma partição com a componente AFG e outra partição com as componentes B e CDE. Essa partição é mostrada na figura 2.10 para o grafo de componentes fortemente conexas da figura 2.8. Na figura 2.10, os retângulos verde e vermelho representam a comunicação de dependências através da fila de comunicação entre os *cores*. Repare que todas as dependências entre instruções (figura 2.7) foram mantidas na figura 2.10. Contudo, como existe a dependência A $\rightarrow$ B e estas duas instruções estão em *threads* diferentes, o valor produzido em A deve ser comunicado para B através da fila de comunicação (retângulo verde na figura 2.10). Da mesma forma, as dependências de controle G $\rightarrow$ B, G $\rightarrow$ C, G $\rightarrow$ D e G $\rightarrow$ E precisam ser comunicadas entre as *threads* (retângulo vermelho na figura 2.10), já que a instrução G foi colocada em uma *thread* e B, C, D e E foram colocadas em outra.

Por outro lado, se o sistema que irá executar a aplicação aceitar três *threads* executando de forma concorrente, podemos formar duas ou três partições. Portanto, além dos dois possíveis particionamentos citados, poderíamos formar uma partição com a componente AFG, outra partição com a componente B e finalmente uma partição com a componente

CDE.

O particionamento desejado é aquele que, considerando os tempos de execução de cada componente e os custos das comunicações que ocorrerão em tempo de execução, minimize o tempo total de execução. Como tais custos, na maioria das vezes, não podem ser determinados em tempo de compilação, são utilizadas informações de *profile* a fim de estimar o número de ciclos necessários para executar cada possível particionamento.

Contudo, o problema de determinar o particionamento de menor tempo de execução é NP-completo e dependente da arquitetura alvo, tornando assim necessário o uso de heurísticas. A heurística utilizada em [27] tenta maximizar o balanceamento de carga entre as *threads*, uma vez que quanto mais balanceados forem os estágios de um *pipeline*, maior será sua eficiência. Tal heurística mantém um conjunto de componentes cujos antecessores no grafo de componentes fortemente conexas já foram atribuídos a alguma partição. Isso garante a comunicação unidirecional necessárias na formação de um *pipeline*. Em seguida a heurística escolhe a componente deste conjunto que possui o maior tempo estimado de execução e adiciona tal componente à partição que está sendo formada. Caso existam duas componentes com o mesmo custo estimado de execução, é escolhida aquela que minimiza o custo de dependências que saem da partição que está sendo formada, a fim de minimizar o custo de comunicação entre as *threads*. Quando o custo total da partição que está sendo formada se aproxima do custo total de execução dividido pelo número de *threads*, a heurística finaliza a formação da partição e inicia a formação da próxima.

Quando um particionamento é finalizado, o algoritmo simula a inserção de instruções de envio e recebimento de dependências e partir daí decide se a paralelização irá trazer benefícios ou não. Caso nenhuma partição gere um código paralelo mais eficiente que o código sequencial, o algortimo termina, gerando código convencional.

Por outro lado, caso algum particionamento seja encontrado, o algoritmo prossegue para o quinto passo a fim de realizar a distribuição do código para as partições geradas.

5 - Distribuir o código para as partições determinadas

Os passos a serem seguidos para a distribuição do código são:

- Computar o conjunto de blocos básicos de cada partição. Esse conjunto é composto pelos blocos básicos que, no código original, possuíam alguma instrução que esteja na partição presente e também pelos blocos básicos que, no código original, continham alguma instrução da qual uma instrução na partição presente depende. A presença desses blocos é necessária para a introdução de instruções que realizarão a comunicação de dependências. Finalmente, são criados os blocos básicos para cada partição.

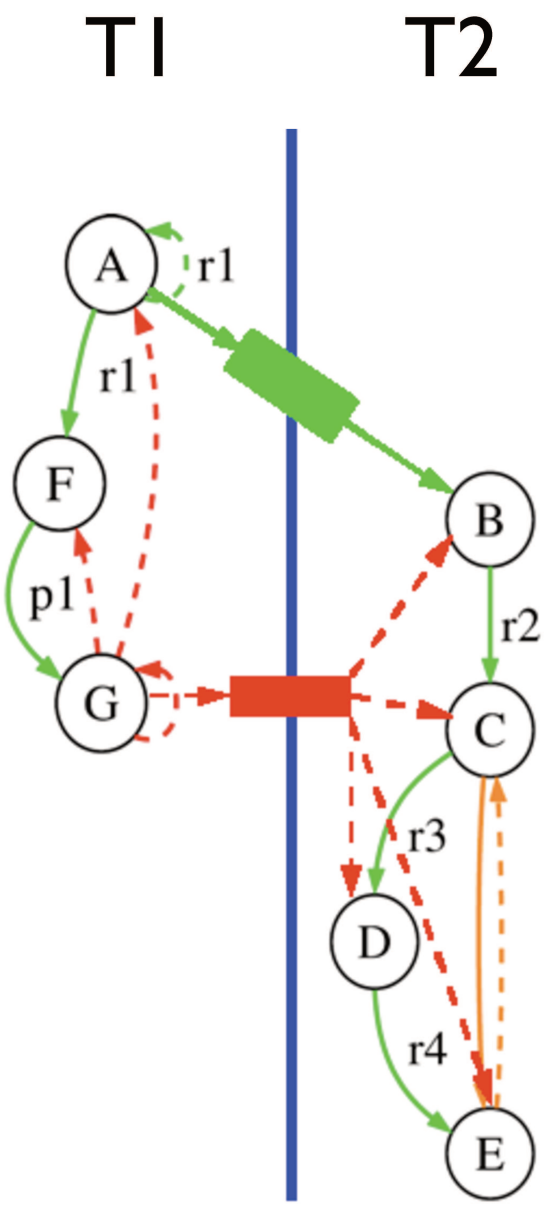


Figura 2.10: Componentes fortemente conexas do grafo da figura 2.8.

- As instruções atribuídas a cada partição são então colocadas nos blocos básicos correspondentes, preservando a ordem original das instruções no blaco básico.
- Ajustar os alvos dos desvios. Quando o alvo de algum desvio não estiver na mesma *thread*, o alvo é redirecionado para o pós-dominador mais próximo do alvo original.

6 - Inserir os fluxos de comunicação

Por fim, são inseridas as intruções para comunicação de dependências entre as *threads*. No caso de uma dependência de dados, o valor deste dado é comunicado entre as *threads*. Para dependências de controle, a direção do desvio é transmitida, e para dependências de memória, qualquer valor é transmitido, já que é necessário apenas garantir a ordem de execução das operações de memória. A implementação da técnica DSWP assume que existem filas de comunicação em *hardware* entre os *cores* [27]. É através destas filas que a comunicação das dependências é realizada.

A figura 2.11 ilustra a inserção das instruções de comunicação no código das *threads*. Primeiramente, note que as instruções que serão executadas pela primeira *thread* estão acima da linha azul, e as instruções que serão executadas pela segunda *thread* estão abaixo desta linha. Essas instruções são simplesmente copiadas para o código de cada *thread* (T1 e T2).

Depois disso, repare que existe uma dependência de controle de todas as instruções da segunda partição em relação à instrução G. Assim, para que a instrução G continue determinando se as instruções da segunda partição serão ou não executadas, é necessário duplicar G na partição dois. Esta duplicação dá origem então à instrução G'. Contudo, a segunda *thread* não precisa computar a condição do desvio utilizada por G, já que esta é computada pela primeira *thread*. Assim, logo após computar a condição do desvio, uma instrução é inserida na primeira *thread* (segundo `produce`) para enviar o resultado da condição para a segunda *thread*. A segunda *thread* recebe então a informação através da instrução `consume` correspondente, e a instrução G' realiza ou não o salto, dependendo da informação recebida.

Por fim, note que a instrução B depende de um valor produzido pela instrução A para ser executada. Assim, uma instrução `produce` é inserida após a instrução A na primeira *thread*, de modo que, assim que a instrução A for executada, o valor calculado já é transmitido para a segunda *thread* para que B possa ser executada. Em seguida, é inserida uma instrução `consume` na segunda *thread*, antes da instrução B, para receber a dependência de A, e utilizar esse valor para executar B.

A técnica DSWP foi implementada em um compilador e gerou, em média, ganhos de desempenho de 15% (em relação à execução sequencial), para aplicações de propósito geral em uma máquina simulada de dois núcleos de processamento [27]. Além disso, foi

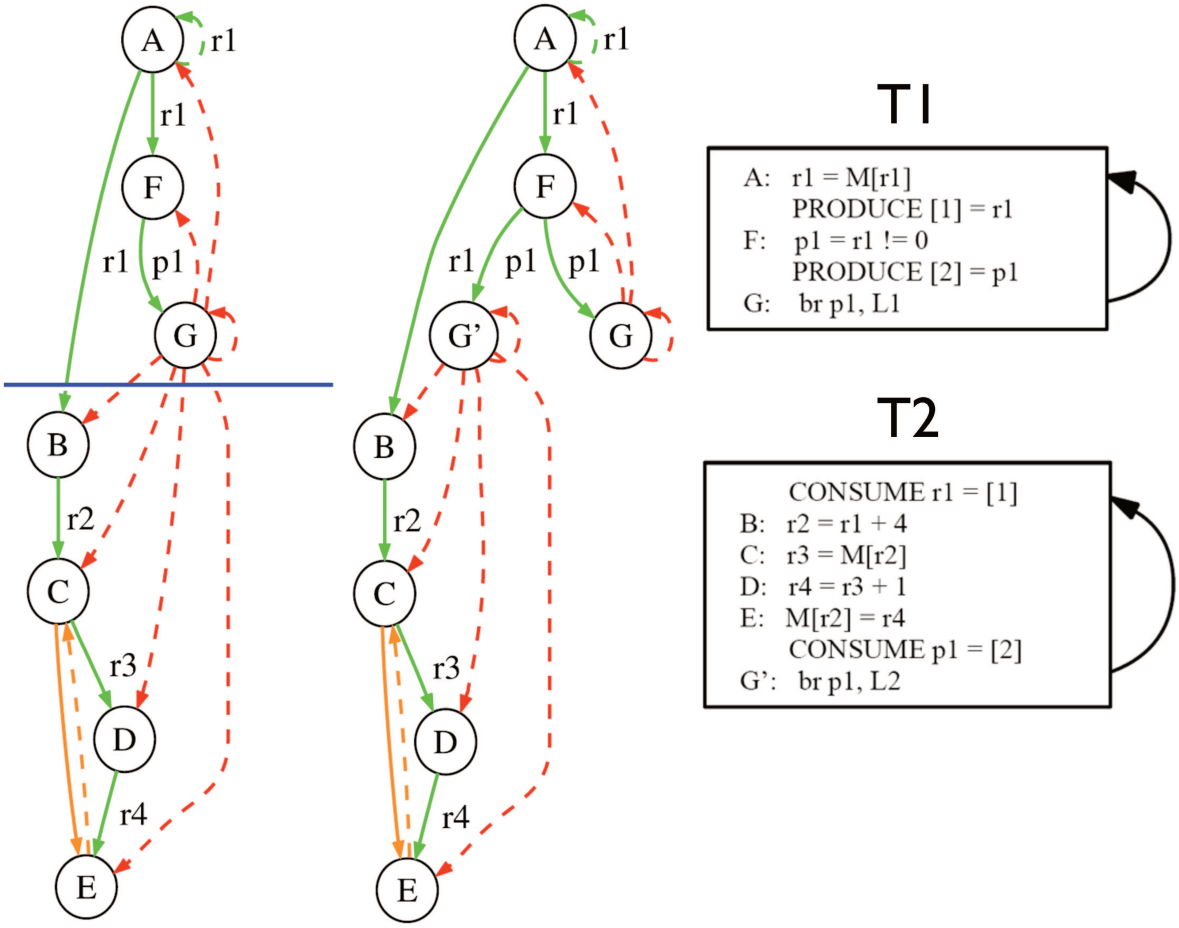


Figura 2.11: Formação do código de cada *thread*.

mostrado [27] que o desempenho da técnica depende pouco da latência de comunicação entre as *threads*, o que pode prejudicar consideravelmente o desempenho de aplicações paralelizadas pelo método DOACROSS, conforme mencionado anteriormente.

2.2 DSWP e especulação

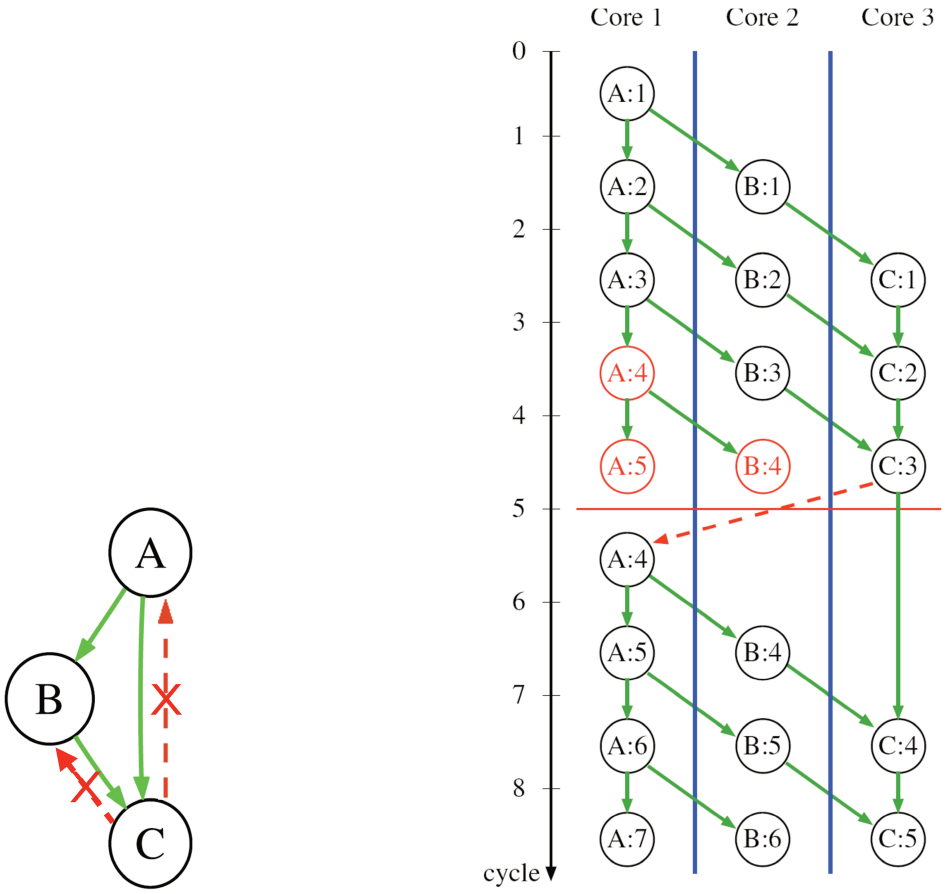
Embora a técnica DSWP tenha sido capaz de gerar programas paralelizados que apresentam bons ganhos de desempenho em uma máquina com dois *cores*, o constante aumento do número de transistores disponíveis possibilitou que a indústria adicionasse mais núcleos de processamento às máquinas de prateleira. Logo, com o objetivo de aproveitar os recursos de processamento oferecidos pela indústria, tornou-se necessário extrair mais paralelismo das aplicações de propósito geral.

Desta forma, foi proposta [42] a incorporação de especulação à técnica DSWP. Especulando que dependências que ocorreram com uma frequência muito baixa em execuções anteriores provavelmente não ocorrerão novamente, ciclos de dependências podem ser quebrados. Como tais ciclos fazem com que instruções sejam colocadas na mesma componente fortemente conexa, e consequentemente fiquem em uma mesma *thread*, a quantidade de paralelismo diminui. Com a introdução de especulação, instruções pertencentes a ciclos de dependência que foram quebrados pela especulação, podem agora ser escalonados para *threads* distintas, aumentando assim o paralelismo e possibilitando um melhor balanceamento de carga entre as *threads*. Para determinar a frequência de ocorrência das dependências, em tempo de execução, são utilizadas informações de *profile*.

A figura 2.12(a) mostra um ciclo de dependências que obriga as instruções A, B e C a permanecerem na mesma *thread*. Contudo, se as dependências de controle (representadas por arestas vermelhas) se manifestarem poucas vezes, o compilador pode especular que tais dependências nunca irão ocorrer, quebrando assim a componente fortemente conexa. Assim, cada instrução pode ser escalonada para uma *thread*, conforme mostrado na figura 2.12(b).

Na mesma figura note que, a dependência de controle de C para A se manifestou na terceira iteração. Neste caso, o sistema que está executando a aplicação deve ser capaz de detectar tal violação e desfazer tudo aquilo que já foi executado de forma errada. Neste exemplo, a quarta iteração da porção A já havia sido executada, mas como esta especulou de forma equivocada a dependência de C para A, a quarta iteração deve ser re-iniciada, bem como todo o código que a sucede no código original.

A técnica DSWP com especulação (Spec-DSWP) foi implementada em um compilador [42] e os programas gerados foram executados em um simulador que foi adaptado para tratar a detecção e a correção de violações da consistência sequencial. Tal técnica conseguiu, em média, ganhos de desempenho próximos a 40%, em comparação com a



(a) Todas as instruções pertencem à mesma componente fortemente conexa.

(b) Execução de código paralelizado com especulação.

Figura 2.12: Quebrando ciclos de dependências com especulação.

execução sequencial, em uma máquina com quatro núcleos.

2.3 *Parallel Stage Decoupled Software Pipeline* (PS-DSWP)

Embora o método DSWP combinado com especulação tenha obtido ganhos de desempenho consideravelmente superiores à sua versão sem especulação de dependências, um *speed-up* médio próximo de 40% em relação à execução sequencial em uma máquina com quatro núcleos deixa evidente que os núcleos estão sendo sub-utilizados. Tal sub-utilização é ainda mais acentuada tendo em vista que o número de núcleos disponilizados pela indústria em um *chip* continua a aumentar (*circa* 2012).

Dessa forma, pode-se concluir que, embora a técnica DSWP seja capaz de paralelizar

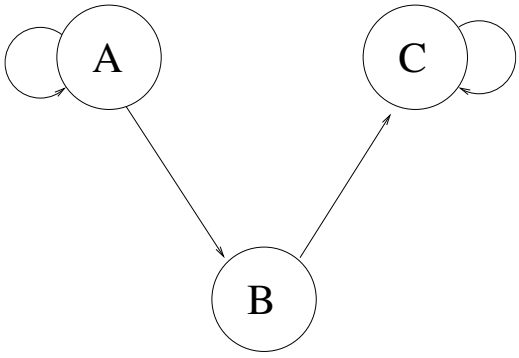


Figura 2.13: Grafo de dependências para o escalonamento PS-DSWP da figura 2.14.

laços de aplicações de propósito geral, esta não consegue gerar ganhos de desempenho escaláveis com o número de núcleos de processamento. Por outro lado, o método DOALL discutido no capítulo 2 normalmente obtém ganhos de desempenho escaláveis, embora a aplicação dessa técnica de paralelização seja muito restrita, já que ela exige a ausência de dependências entre iterações.

Assim, com o objetivo de combinar a aplicabilidade de DSWP com a escalabilidade de desempenho do método DOALL, surgiu a técnica de parallelização *Parallel Stage Decoupled Software Pipeline* (PS-DSWP) [30]. O modelo de execução deste técnica é ilustrado na figura 2.14, a partir do grafo de dependências da figura 2.13.

Repare que o modelo geral de execução é um *pipeline* (DSWP) formado pelos estágios A, B e C, onde o estágio B é executado em paralelo (DOALL). Portanto, o método PS-DSWP consiste em paralelizar a aplicação conforme o método DSWP (produzindo uma execução em *pipeline*), mas permitindo que estágios que não possuem dependências entre iterações (que é a condição para que o método DOALL seja aplicável) sejam exucutados simultaneamente como na técnica DOALL.

Na figura 2.14 o estágio B do *pipeline* formado não apresenta dependências entre iterações (isto é, um estágio qualquer Bx não depende de By, onde $x > y$). Logo, este estágio é executado em paralelo como no modelo DOALL. Por outro lado, repare que tanto o estágio A como o estágio C dependem da iteração anterior deles mesmos. Portanto, estes não podem ser executados de forma DOALL.

A vantagem de permitir que estágios sem dependências entre iterações sejam executados em paralelo fica ainda mais evidente quando estes estágios são os que consomem o maior tempo de execução dentro do *pipeline*. Na figura 2.14 pode-se notar que o estágio B é maior que os demais. Logo, se este estágio não fosse executado em paralelo, a latência do *pipeline* seria dominada por ele, limitando assim o ganho de desempenho produzido pela técnica DSWP.

A técnica PS-DSWP foi implementada em um compilador de pesquisa [30], e os pro-

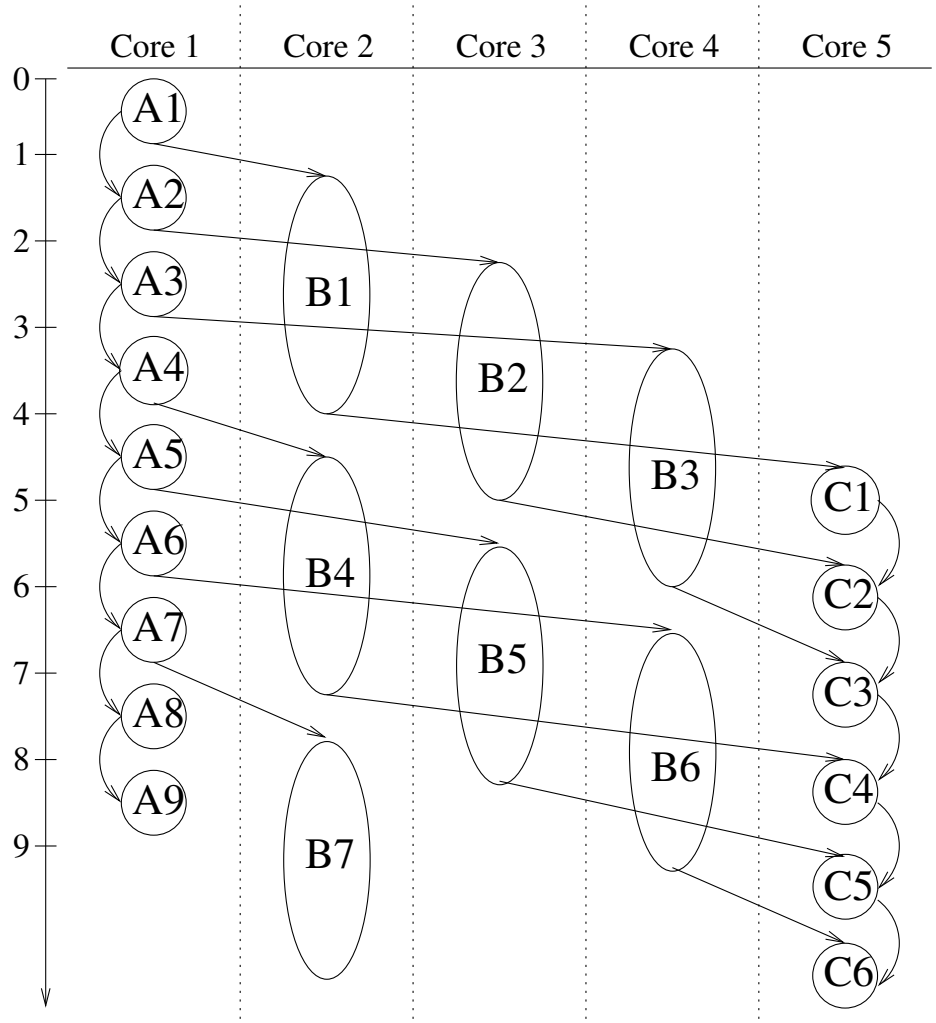


Figura 2.14: Modelo de execução do método *Parallel Stage Decoupled Software Pipeline*.

gramas gerados foram executados em uma máquina simulada de seis núcleos. Foram selecionados cinco laços para serem paralelizados pela técnica PS-DSWP e executados na máquina citada. O ganho de desempenho obtido foi, na média, 114% em relação à execução sequencial utilizando seis *threads*. Para estes mesmos laços, a técnica DSWP forneceu um ganho médio de apenas 36% na mesma máquina com seis núcleos.

2.4 PS-DSWP e especulação

.

Embora a técnica PS-DSWP tenha se mostrado capaz de produzir ganhos de desempenho bastante superiores à técnica DSWP, pode-se perceber que os núcleos disponíveis ainda estão sendo sub-utilizados. Mais uma vez, o número de núcleos de processamento por máquina continua a aumentar, sendo que atualmente as máquinas Xeon da Intel, por exemplo, possuem 24 núcleos.

Não obstante, não é comum, durante o processo de paralelização de laços, encontrar grandes estágios de *pipeline* que não possuem dependências entre iterações. Na prática, no entanto, muitas dessas dependências se manifestam com pouca ou nenhuma frequência em tempo de execução [8]. Assim, é comum em aplicações de propósito geral, paralelizadas por DSWP, serem encontrados estágios grandes que não podem ser executados segundo o modelo DOALL por causa de dependências infrequentes. Logo, se estas dependências forem especuladas nos estágios grandes, é possível produzir um *pipeline* no qual estágios grandes possam ser executados em paralelo, como no modelo PS-DSWP.

A combinação da técnica PS-DSWP com especulação de dependências para produzir grandes estágios que possam ser executados em paralelo é chamada de *Speculative Parallel Stage Decoupled Software Pipeline* (Spec-PS-DSWP). Esta técnica [7, 40] é capaz de prover ganhos de perfomance escaláveis com o número de *cores*.

Contudo, visto que o método Spec-PS-DSWP faz uso de especulação de dependências, é necessário que as aplicações paralelizadas por essa técnica sejam executadas em um sistema que possua algum mecanismo capaz de detectar e corrigir violações de dependência. Muitos mecanismos em *hardware* foram propostos [10, 15, 39] a fim de permitir a especulação de dependências aplicada aos métodos DOALL e DOCROSS. Infelizmente, além desses mecanismos apresentarem problemas clássicos como geração de tráfico em rajada, introdução de novos protocolos de coerência de cache e dependência de *hardwares* e estruturas especiais centralizadas, eles não aceitam o modelo de execução em *pipeline* combinado com a replicação de estágios e especulação, podendo gerar resultados incorretos.

Entretanto, como esse modelo de execução é capaz de prover ganhos de desempenho escaláveis, mecanismos para a execução de aplicações paralelizadas por Spec-PS-DSWP

Técnica	Detecção de paralelismo	Desempenho	Lógica de suporte
DOALL	Baixa	Alto	–
DOACROSS	Baixa	Baixo	–
TLS	Média	Alto	Média
DSWP	Alta	Médio	–
PS-DSWP	Alta	Médio	–
Spec-PS-DSWP	Alta	Alto	Média

Tabela 2.1: Propriedades das técnicas de paralelização.

tornam-se necessários. O mecanismo de *Multi-threaded Transactions* (MTX) [41] permite o modelo de execução requerido pela técnica Spec-PS-DSWP, mas tal mecanismo depende de mudanças caras e complexas no *hardware* já existente, bem como um novo protocolo de coerência de cache.

Por outro lado, o mecanismo de *Software Multi-threaded Transactions* (SMTX) também aceita o modelo requerido pela técnica Spec-PS-DSWP, e não requer mudanças no *hardware*. Este mecanismo é ilustrado na figura 2.15. Neste modelo, um dos *cores* (na figura, o *core* seis) é utilizado como unidade de *commit*. Enquanto os demais *cores* executam a aplicação paralelizada, esta unidade verifica se ocorreu alguma violação de memória por parte de algum dos *cores*. Caso não tenha ocorrido nenhuma violação em uma iteração qualquer, esta iteração sofre *commit*. Contudo, para o mecanismo SMTX funcione, é necessário que todos os *cores* enviem as operações de memória realizadas para a unidade de *commit* (os envios são representados na figura 2.15 pelas setas saindo dos *cores* de um a cinco e chegando no *core* seis), para que esta possa fazer a verificação de violações. Além disso, a unidade de *commit* precisa refazer todas as operações de memória realizadas pelos demais *cores* para assegurar que não houveram violações. Infelizmente, este trabalho centralizado na unidade de *commit*, a necessidade de todos os *cores* comunicarem os acessos à memória realizados, juntamente com a pressão no sistema de memória, criaram um gargalo para a escalabilidade de desempenho provida pela técnica Spec-PS-DSWP para várias aplicações.

A tabela 2.1 apresenta, de forma resumida, as técnicas de paralelização estudadas, juntamente com a capacidade que cada uma tem de detectar paralelismo presente nos laços de aplicações e o desempenho dos programas paralelos gerados. Para as técnicas que fazem uso de especulação, foi incluída ainda a complexidade da lógica de suporte necessária para a execução dos programas gerados. Note que, como a técnica Spec-PS-DSWP é a única capaz de prover, ao mesmo tempo, uma alta capacidade de detecção de paralelismo e alto desempenho, torna-se importante desenvolver mecanismos que permitam a execução de aplicações paralelizadas por esta técnica.

No capítulo seguinte será introduzido um novo mecanismo que aceita não apenas os

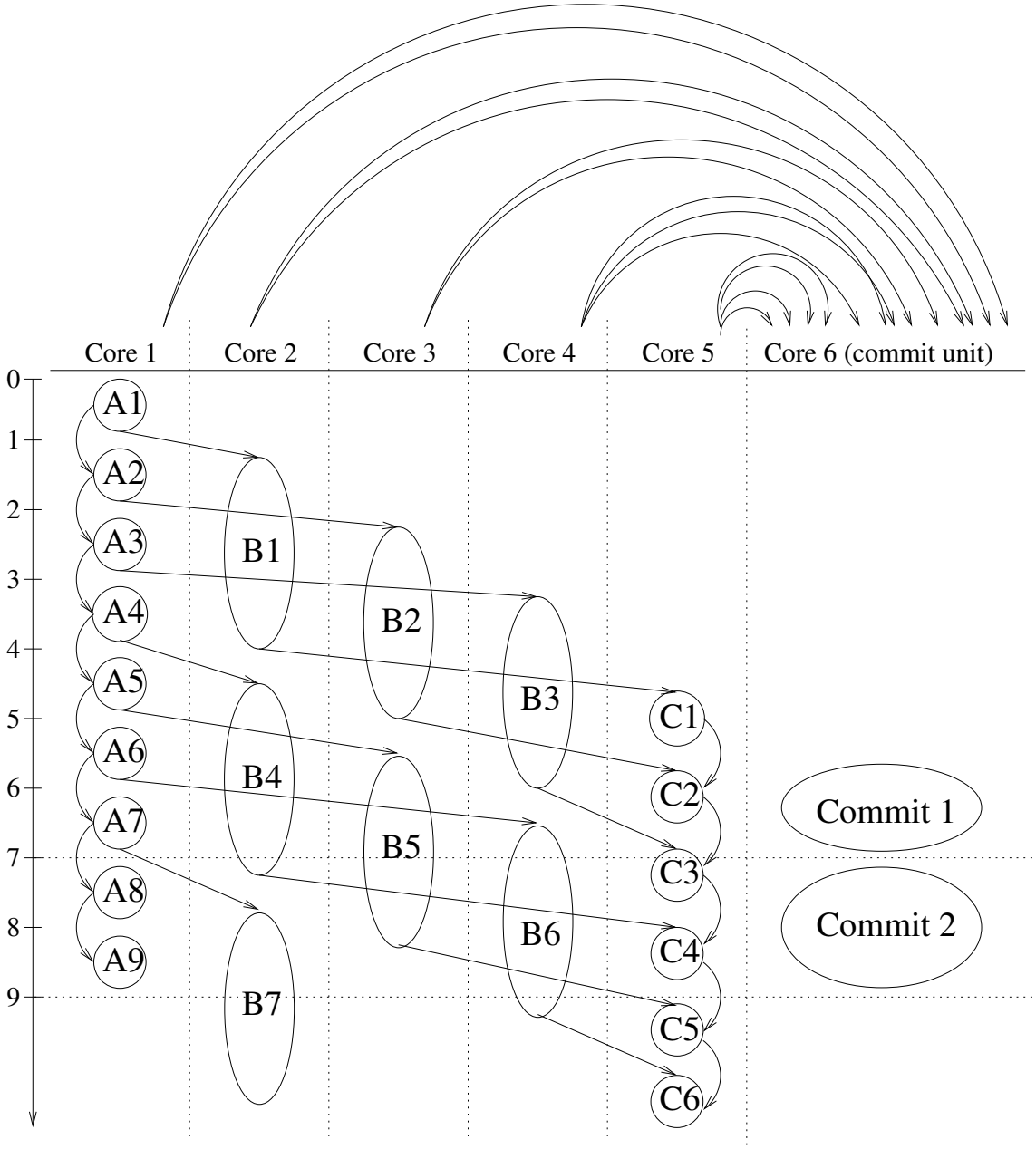


Figura 2.15: O mecanismo SMTX.

modelos tradicionais de execução DOALL e DOACROSS combinados com especulação, mas também o modelo de execução necessário para as técnicas DSWP e PS-DSWP combinadas com especulação (Spec-DSWP e Spec-PS-DSWP). Tal mecanismo usa uma combinação de *tags* em cache com uma lógica simples e leve que interage naturalmente com o protocolo de coerência de cache já existente, sem requerer qualquer alteração neste. Com essa simples modificação foi obtido desempenho escalável em um sistema com vários núcleos de processamento. Como o processador *BlueGene/Q* da IBM já possui *tags* em cache juntamente com uma lógica adicional para permitir execução especulativa [16], pode-se esperar que o mecanismo aqui proposto possa ser adotado pela indústria sem muitos esforços.

Capítulo 3

Mecanismo Arquitetural

O mecanismo proposto nesta dissertação assume um *chip multi-processor* (CMP) com n núcleos de processamento conectados por um barramento, e uma hierarquia de caches utilizando um protocolo de coerência baseado em invalidações [11]. O último nível de cache (aquele mais próximo da memória principal) é compartilhado. Por fim, este último nível de cache é inclusivo, o que significa que todos os dados presentes nos demais níveis de cache também estão presentes neste último nível. Tal modelo é similar ao dos processadores i7 da Intel que possuem uma cache L3 compartilhada e inclusiva. O processador *BlueGene/Q* da IBM também possui uma cache L2 compartilhada de 32MB.

O modelo aqui proposto assume que as dependências de registradores são tratadas corretamente pela geração de código, de modo que o nosso modelo precisa tratar apenas das dependências de memória. Tal objetivo é atingido através da expansão das linhas de cache com *tags* de coerência entre iterações.

A figura 3.1 ilustra dois *sets* de uma cache de associatividade quatro. A figura representa as *tags* de coerência por meio de três *tags* de n bits (onde n é o número de núcleos) nomeadas IDTag, DepsTag e VersionsTag. Cada uma dessas *tags* desempenha um ou mais papéis distintos no nosso sistema, conforme descrito nas seções 3.1, 3.2 e 3.3.

Nosso mecanismo pode ser implementado fazendo uso de apenas $log(n)$ bits por linha de cache para todos os níveis de cache, com exceção do último (compartilhado e inclusivo)

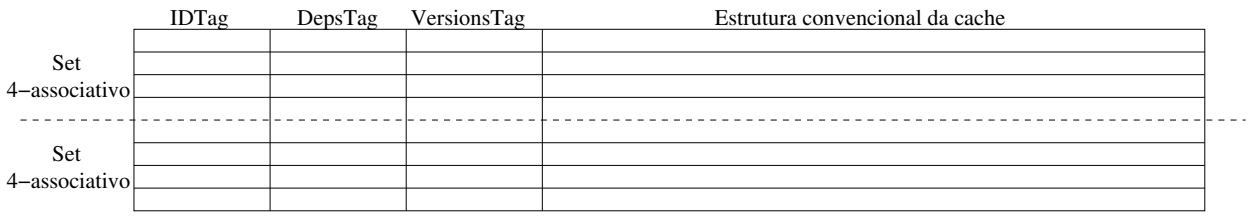


Figura 3.1: Novas linhas de cache.

que requer $2n + log(n)$ bits. Neste trabalho o sistema será apresentado usando $3n$ bits por linha de cache com o intuito de simplificar a exposição e facilitar o entendimento. A idéia de inserir *tags* nas linhas de cache para permitir a manutenção de múltiplas versões de dados nas caches já é utilizada no processador *BlueGene/Q* da IBM, conforme mencionado anteriormente.

Vale ressaltar que o protocolo de coerência de cache já existente gera *naturalmente* todas as transações necessárias para manter atualizadas as *tags* adicionais. Tal fato é demonstrado na seção 3.5 para uma série de exemplos detalhados e provado no apêndice A. Por fim, note que não existe nenhum mecanismo especial para transferência de dados entre as caches. Tais transferências ocorrem sempre através do barramento de interconexão, como já é feito nas máquinas com múltiplos núcleos de processamento atualmente.

3.1 *Tag* de ID da iteração (IDTag)

A IDTag representa simplesmente a iteração na qual a linha de cache correspondente foi produzida.

Nosso modelo de execução permite que várias iterações de um laço paralelizado sejam executadas ao mesmo tempo, o que requer a capacidade de versionar os dados gerados. A forma mais simples de habilitar o versionamento de dados é através da geração de um identificador único para cada iteração a ser executada. Todavia, essa abordagem pode não ser a melhor solução, uma vez que nosso mecanismo necessita de um limite superior para o identificador da iteração, pois este precisa ser representado em *hardware*. Além disso, para a implementação da verificação de conflitos, o sistema necessitaria de muitos comparadores, gerando assim uma sobrecarga na potência e na área, sem mencionar a possibilidade de afetar o caminho crítico do circuito da cache.

O modelo de execução aqui proposto impõe um limite superior para o número de iterações que estão sendo executadas em um dado momento. Em um sistema com n núcleos, podem haver no máximo n iterações do laço sendo executadas concorrentemente a qualquer momento. Assim, cada linha de cache possui uma IDTag de n bits que representa o identificador da iteração na qual essa linha foi produzida com um único bit ativado. Conforme explicado na seção 3.4, essa decisão simplifica consideravelmente a implementação das operações de *commit* e *squash*.

A fim de ilustrar o funcionamento da IDTag, considere por exemplo um sistema de quatro *cores*. Neste sistema, a primeira iteração de um laço paralelizado é representada pelo padrão de bits 1000. A iteração dois do laço é representada pelo identificador 0100. Da mesma forma, a terceira iteração é representada por 0010. Após a iteração quatro (cujo identificador é o padrão de bits 0001), os identificadores são reaproveitados, fazendo com que a iteração cinco do laço seja representada pelo padrão 1000, a iteração seis pelo

IDTag				Localidade
0	1	0	0	X
1	0	0	0	Y
0	0	1	0	Y

Figura 3.2: Dados escritos pelo *core* 3.

padrão 0100 e assim por diante. Este reaproveitamento é possível porque existem no máximo quatro iterações sendo executadas concorrentemente em qualquer instante de tempo. Portanto, as iterações um (com IDTag 1000) e cinco (com IDTag 1000) de um laço paralelizado nunca estarão sendo executadas ao mesmo tempo.

A figura 3.2 mostra três exemplos de entradas na cache e suas respectivas IDTags. Assuma, por exemplo, que a figura 3.2 representa o estado da cache L1 do *core* três após a execução das iterações um, dois e três de um laço paralelizado, em uma máquina com quatro *cores*. Observando a figura 3.2, é possível inferir que ambas as iterações um (cujo identificador é 1000) e três (cujo identificador é 0010) escreveram no endereço Y, ao passo que a iteração dois (cujo identificador é 0100) escreveu no endereço X. Este exemplo mostra que o mesmo endereço (Y) pode agora estar presente em mais de uma linha de cache do mesmo *set*. Note que, embora as duas linhas tenham o mesmo endereço, seus endereços *especulativos* são diferentes, pois uma linha foi produzida na iteração um (IDTag 1000) e a outra foi produzida na iteração três (IDTag 0010). Aqui, será utilizada a notação Y_{1000} para representar o endereço especulativo Y da iteração cujo identificador é o padrão de bits 1000. Assim, Y_{0010} representa o endereço especulativo Y da iteração 0010.

Qualquer valor da IDTag diferente de 0000 indica uma linha de cache especulativa que não pode ser removida do último nível de cache (compartilhado e inclusivo) e não pode ser escrita na memória principal. Portanto, um dado especulativo nunca é escrito na memória. Como veremos adiante, tal característica possibilita a implementação de operações de *squash* extremamente eficientes. Quando uma determinada violação de consistência é detectada pelo sistema, basta que os dados especulativos das caches sejam invalidados, já que as memórias principal e secundária não possuem dados especulativos. Além disso, como uma violação foi detectada, os dados especulativos não são válidos e não precisam ser escritos na memória principal, evitando assim a geração de tráfico adicional em rajada. Este é um dos principais problemas encontrados por abordagens tradicionais de suporte a TLS, já que tal tráfego é responsável por gerar contenção, perda de desempenho e

aumento do consumo de energia [33].

O reaproveitamento de identificadores para a IDTag gera uma complicação referente à ordenação entre identificadores. Por exemplo, em um sistema de quatro *cores*, a iteração representada pelo padrão de bits 0001 está logicamente antes ou depois daquela representada pelo identificador 1000? A resposta depende do significado atual dos identificadores. Se o padrão de bits 1000 representa a iteração um e o padrão 0001 representa a iteração quatro, então $1000 < 0001$, onde o sinal "<" indica que o identificador 1000 (iteração um) ocorreu antes de 0001 (iteração quatro). Por outro lado, se o identificador 1000 representa a iteração cinco, então temos que $1000 > 0001$. A fim de solucionar este problema de ordenação de identificadores, no nosso modelo de execução cada *core* possui um identificador que determina a posição relativa deste em relação aos demais núcleos de processamento.

Por exemplo, o núcleo um sabe que não podem haver iterações do laço paralelizado em execução que estejam logicamente *depois* da iteração que ele está executando. Da mesma maneira, o núcleo dois sabe que existe no máximo uma iteração sendo executada no sistema que está logicamente depois da iteração que ele está executando, e assim por diante até o núcleo quatro que sabe que todas as iterações que estão sendo executadas no sistema estão logicamente *depois* da iteração que ele está executando. Isto pode ser notado na figura 3.3. Repare que, em qualquer instante, nenhum outro núcleo está executando uma iteração que esteja *depois* da iteração que está sendo executada pelo núcleo um. Da mesma forma, o núcleo dois sabe que existe no máximo uma iteração sendo executada no sistema (a do núcleo um) que está depois da iteração que ele está executando. Por fim, o núcleo quatro sabe que todas as iterações que estão sendo executadas no sistema estão depois da iteração que ele está executando.

Com essa ordenação entre os *cores* em mente, é sempre possível determinar, para cada *core*, o conjunto de iterações logicamente *posteriores* e *anteriores* à sua iteração que estão sendo executadas no sistema em um dado instante.

3.2 *Tag* de consistência de intervalo (DepsTag)

A DepsTag é utilizada com o intuito de detectar violações na ordem das operações de memória. A DepsTag de uma linha de cache X representa o intervalo de iteraçoes no qual qualquer escrita na linha de cache X deve ativar a recuperação de falha de especulação.

Quando uma operação de leitura da linha de cache X é requisitada em alguma iteração I_k, o mecanismo arquitetural deve ser capaz de detectar qual é a versão *mais apropriada* da linha X para ser usada pela iteração I_k. A versão *mais apropriada* da linha de cache X deve vir da mais recente iteração I_j que é logicamente *anterior* a I_k em tempo de execução (ou seja, $j < k$).

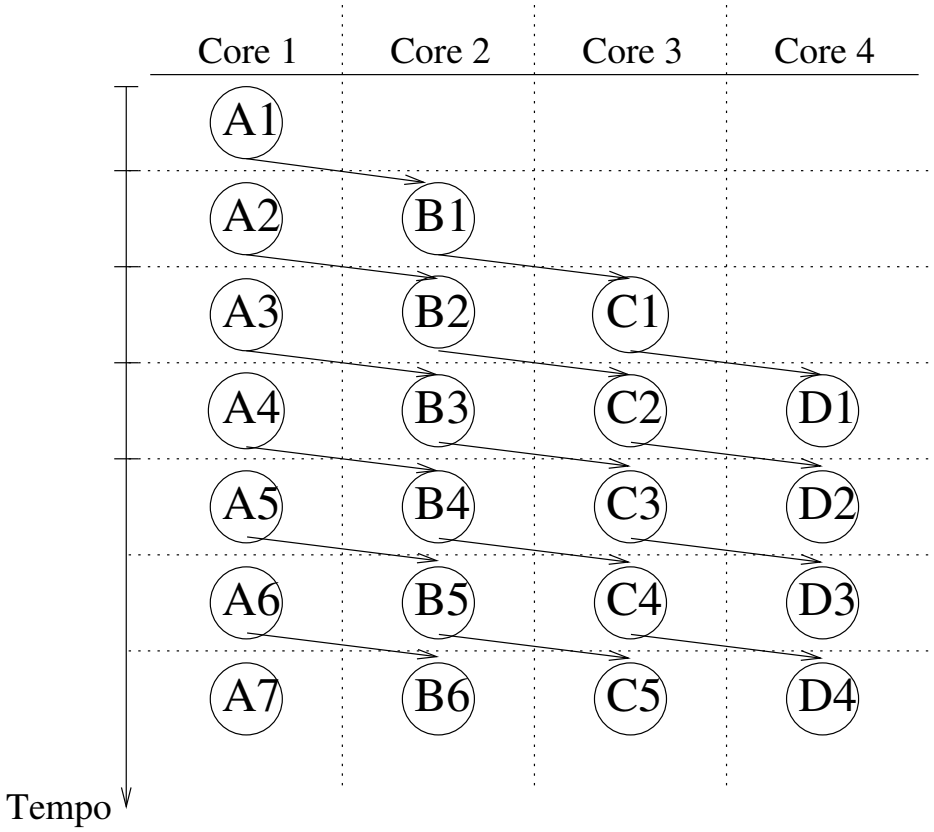
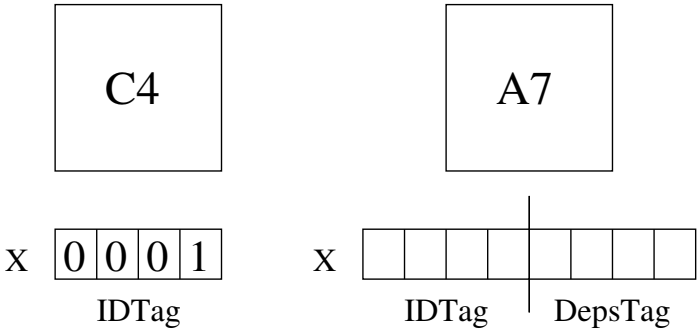


Figura 3.3: Escalonamento DSWP.

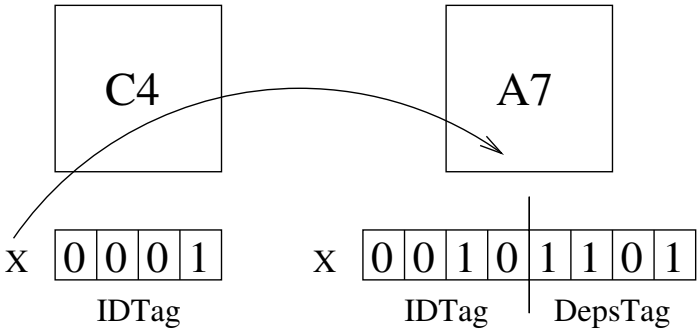
A fim de evitar falhas de especulação, o sistema precisa manter o registro de um intervalo de iterações no qual qualquer escrita na linha de cache X deve invalidar o seu valor atual para todas as iterações que estão sendo executadas naquele momento. Supondo que a iteração I_k faz a leitura da linha X da iteração I_j, o intervalo de invalidação é $[I_j, I_k)$, já que qualquer nova escrita na linha de cache X realizada em uma iteração dentro deste intervalo irá produzir um valor que é mais apropriado para a iteração I_k do que aquele produzido pela iteração I_j. Portanto, qualquer escrita na linha X realizada em uma iteração dentro do intervalo $[I_j, I_k)$ irá causar uma falha de especulação.

O exemplo exposto pela figura 3.4(a) mostra a IDTag, descrita anteriormente, para a linha de cache X depois que uma escrita na linha X é realizada no estágio C4 do *pipeline*. Agora, assuma que o estágio A7 faça uma leitura da linha X, criando assim o endereço especulativo X_{0010}. A figura 3.4(b) mostra a IDTag para a linha X_{0010}. Suponha que a versão mais apropriada da linha de cache X presente no sistema, para uma leitura no estágio A7 é a linha produzida pelo estágio C4. Como a iteração sete faz a leitura de um valor produzido pela iteração quatro, qualquer escrita na linha de cache X realizada em uma iteração dentro do intervalo $[4, 7)$ produz um valor mais apropriado para a leitura feita em A7 do que o que foi produzido em C4. Este intervalo é representado através da ativação dos bits quatro, cinco e seis na DepsTag (atenção para a numeração circular) para o novo endereço especulativo X_{0010} (cujo identificador é o padrão 0010). As *tags* para esse novo endereço especulativo são mostradas na figura 3.4(b). Qualquer modificação realizada na linha de cache X em qualquer iteração neste intervalo dispara o processo de recuperação de falha de especulação.

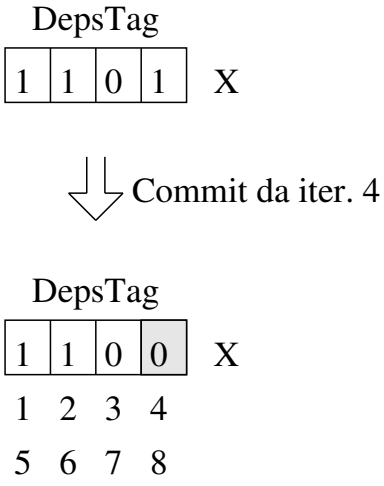
No entanto, o reaproveitamento de identificadores pode gerar um problema para a DepsTag. Depois que os bits desta *tag* são ativados, o sistema não tem como saber se o padrão de bits 0001 já representa a iteração oito, ou se ainda representa a iteração quatro. Por exemplo, suponha que o dado X da iteração quatro (identificador 0001) seja lido na iteração sete (identificador 0010). Neste caso, o X gerado por essa leitura teria sua DepsTag ajustada para o padrão 1101. Isto quer dizer que, se houver uma escrita em X na iteração quatro (representada por 0001), cinco (1000) ou seis (0100), o *hardware* deve sinalizar uma violação de ordenação de memória, já que qualquer um desses dados seria mais apropriado para uma leitura na iteração sete do que o dado que foi lido (da iteração quatro). Suponha então que ocorra uma escrita na iteração 0001. Agora, não é possível saber se esta escrita na iteração 0001 foi uma escrita na iteração quatro ou oito (ambas são representadas pelo padrão 0001). Se o identificador 0001 estiver representando a iteração quatro, uma violação de ordenação de memória deve ser sinalizada, conforme explicado anteriormente. No entanto, se o padrão 0001 estiver representando a iteração oito, nada deve acontecer, já que o dado X foi lido pela iteração sete, e um valor escrito na iteração oito não é apropriado para uma leitura na iteração sete.



(a) Escrita em C4



(b) Leitura em A7



(c) Após o Commit da Iteração 4

Figura 3.4: Mantendo o Registro de Dependências

Para resolver esta ambiguidade, no *commit* de toda iteração i, o sistema faz com que o i-ésimo bit da DepsTag seja zerado. Por exemplo, se a iteração quatro (cujo identificador é o padrão de bits 0001) sofre *commit*, o sistema *precisa* desativar o bit quatro na DepsTag. Este procedimento é evidenciado pela figura 3.4(c). Repare que, depois que a iteração quatro sofre *commit*, a DepsTag 1101 é alterada para 1100, já que o quarto bit é desativado. Desta forma, o identificador 0001, utilizado para representar a iteração quatro, pode ser reaproveitado de maneira segura, a fim de representar agora a iteração oito.

3.3 A *tag* de Versões (VersionsTag)

Quando um *core* realiza uma leitura ou uma escrita em uma linha de cache X, na iteração i, se X_i ainda não está disponível na cache, a requisição irá gerar um *cache miss* na cache privada L1 deste *core*. Logo, o protocolo de coerência de cache irá naturalmente gerar uma transação de barramento com a finalidade de conseguir o dado desejado de algum lugar na hierarquia de memória. Desta forma, com o aparecimento de tal transação no barramento compartilhado, todas as caches saberão que uma nova versão da linha de cache X (por exemplo, X_i) será criada.

A fim de armazenar quais versões de uma linha de cache X estão presentes no sistema, é utilizada uma *tag* chamada VersionsTag. A VersionsTag desempenha dois papéis importantes. Primeiramente, quando um *core* realiza a solicitação de uma linha de cache X, ele sabe exatamente qual versão desta linha de cache no sistema é a mais apropriada para a sua requisição. Desta forma, a transação de barramento gerada é específica, não requerendo que todos os demais *cores* respondam. Tal característica faz com que o tráfego no barramento se mantenha o mesmo de um sistema que não utiliza o versionamento de dados na cache. Conforme mencionado anteriormente, isso é uma grande vantagem em relação a muitas propostas anteriores de suporte a TLS nas quais, ou todos os *cores* enviam o dado mais apropriado que possuem para o *core* solicitante (e este escolhe o mais apropriado para a requisição que fez), ou a requisição de leitura ou escrita é feita para uma unidade centralizada de *hardware*. Estes métodos são potenciais causadores de contenção (no barramento ou na unidade centralizada de *hardware*), degradação de desempenho e alto consumo de energia [33].

O segundo papel desempenhado pela VersionsTag é um pouco mais sutil. Suponha que uma linha de cache X foi gerada durante a execução da iteração um (identificador 1000) e outra linha do mesmo endereço X foi gerada durante a execução da iteração dois (identificador 0100). Quando a operação de *commit* for realizada na iteração um, a linha X com identificador 1000 na IDTag passará a ter o identificador 0000 em sua IDTag, já que a itereração um foi executada corretamente, e portanto as linhas com identificador 1000 (iteração um) não são mais especulativas. Agora, quando a iteração dois (0100) sofrer

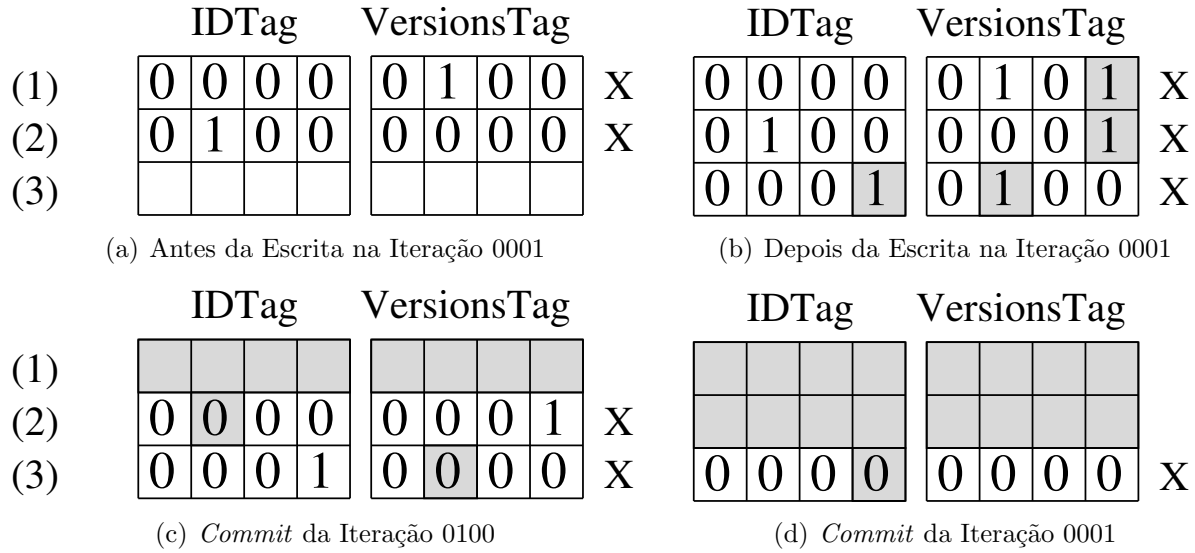
commit, a linha X com identificador 0100 também passará a ter o identificador 0000 em sua IDTag, já que a iteração dois foi executada corretamente, fazendo com que as linhas com identificador 0100 (iteração dois) não sejam mais especulativas. Mas agora temos duas versões da linha X com IDTag 0000 nas caches (uma que surgiu quando a iteração um sofreu *commit* e a outra quando a iteração dois sofreu *commit*). Repare que o dado não especulativo correto é aquele que surgiu quando a iteração dois sofreu *commit*, já que este dado é mais recente (foi produzido na iteração dois) do que aquele produzido quando a iteração um sofreu *commit* (que foi produzido na iteração um). O *hardware* precisa lidar com essa situação a fim de evitar a presença de linhas não-especulativas duplicadas nas caches.

Uma possível solução alternativa seria realizar uma invalidação em rajada de todas as linhas não-especulativas. Contudo, tal solução impõe uma forte sobrecarga no controlador de memória, já que este precisa lidar com (possíveis) grandes requisições de escritas na memória em rajada. Isso ocorre porque todas as linhas de cache com dados mais atuais que aqueles da memória precisam ser escritas na memória principal (*write back*). Além disso, linhas de cache não-especulativas que foram invalidadas desta forma podem ser requisitadas novamente por qualquer *core*, o que faria com que o sistema precisasse transferir tais dados de volta da memória, afetando assim o desempenho de forma negativa. Por fim, é sabido [33] que a geração de tráfego em rajada degrada tanto o desempenho como o consumo de energia de maneira dramática.

Uma outra possível solução seria, a cada operação de *commit*, realizar uma busca por todas as linhas de cache não especulativas que ainda são válidas que poderiam causar um conflito de duplicação de linhas. Esta solução, adotada por alguns dos mecanismos a TLS já propostos, também não é desejável, uma vez que poderia resultar em uma implementação ineficiente em *hardware* do ponto de vista de desempenho e consumo de energia [33].

A *tag* de versões VersionsTag é utilizada para resolver este problema de uma forma eficiente. Cada bit ativado na VersionsTag representa a presença no sistema de uma outra linha de cache com o mesmo endereço, mas produzida em uma iteração diferente. Por exemplo, a figura 3.5(a) mostra o estado de uma cache que contém duas cópias da linha X: uma versão não-especulativa (com o identificador 0000 na IDTag), e uma versão especulativa que foi produzida na iteração representada pelo padrão de bits 0100. Repare que a VersionsTag da linha de cache não-especulativa possui o segundo bit ativado, indicando que uma versão especulativa da iteração representada pelo padrão 0100 está presente no sistema.

A figura 3.5(b) mostra o que acontece na cache depois que uma escrita na linha X é realizada durante a execução de uma iteração representada pelo identificador 0001. Primeiramente, uma nova linha de cache é criada para X_{0001} (linha de cache 3) com sua



(a) Antes da Escrita na Iteração 0001

(b) Depois da Escrita na Iteração 0001

(c) *Commit* da Iteração 0100

(d) *Commit* da Iteração 0001

Figura 3.5: Evitando linhas de cache duplicadas do mesmo endereço X.

VersionsTag representada pelo padrão de bits 0100, indicando que a versão X_{0100} já existe no sistema. Em seguida, a VersionsTag da versão não-especulativa de X (linha de cache 1, com identificador 0000) é alterada para o padrão de bits 0101, representando a existência das linhas de cache X com identificadores 0100 (X_{0100}) e 0001 (X_{0001}) no sistema. O mesmo acontece para a VersionsTag da linha de cache X_{0100} (linha 2) que é atualizada a fim de indicar que a linha X_{0001} agora encontra-se presente no sistema.

A figura 3.5(c) evidencia o que acontece com a cache na operação de *commit* da iteração representada pelo identificador 0100 (linha de cache 2, figura 3.5(b)). A versão não-especulativa (linha 1, com identificador 0000) é invalidada e, se necessário, escrita na memória principal. Isso ocorre porque a VersionsTag desta linha de cache possui o segundo bit ativado, indicando portanto a presença de uma linha de cache X produzida na iteração com identificador 0100 no sistema (linha 2, figura 3.5(b)). Agora, quando a iteração representada pelo padrão 0100 sofre *commit*, a linha de cache correspondente (com identificador 0100) precisa se tornar a nova versão não-especulativa da linha de cache X no sistema. A VersionsTag da linha X_{0001} (linha de cache 3) tem o seu segundo bit ativado, mas tal linha não é nem invalidada e nem escrita de volta na memória principal, já que ainda é especulativa e portanto não pode "escapar" para a memória.

Finalmente, a figura 3.5(d) mostra o que acontece na operação de *commit* da iteração com identificador 0001, invalidando a linha de cache não-especulativa anterior (linha 2, figura 3.5(c)), já que esta linha tem sua VersionsTag com o quarto bit ativado. Este mecanismo gera uma escrita da linha de cache na memória principal somente quando a ausência de tal escrita criaria linhas duplicadas na cache. Esta abordagem evita portanto

a geração de tráfego em rajada de escritas na memória e preserva a localidade espacial (requisições posteriores por linhas não-especulativas poderão ser atendidas pela cache, pois não há invalidação de todas as linhas não-especulativas, como em outros modelos). Para eliminar completamente o tráfego, o *hardware* poderia, durante a operação de *commit* da linha de cache especulativa X_i, propagar o estado *dirty* de uma linha não-especulativa X (que vai sair da cache devido à operação de *commit* de X_i) para a linha X_i.

3.4 *Commits* e *Squashes*

A operação de *commit* no nosso modelo de execução significa que a especulação de uma iteração do laço foi realizada com sucesso, ou seja, produziu os mesmos resultados que teriam sido produzidos pela execução da mesma iteração de forma sequencial. Conforme discutido nas seções anteriores, a operação de *commit* consiste em simplesmente limpar o bit correspondente à iteração que está sofrendo o *commit* em IDTag, DepsTag e VersionsTag.

Por outro lado, a operação de *squash* pode ocorrer na presença dos seguintes eventos:

- Extrapolação da capacidade das caches: o código em execução realiza a leitura ou a escrita em uma linha de cache que ainda não existe no sistema e não há mais entradas nas cacahes disponíveis para acomodar a nova linha. Isso pode acontecer quando todas as entradas que poderiam ser utilizadas pela nova linha estiverem ocupadas com dados especulativos (que não podem ser escritos na memória principal). Este problema não ocorreu em nenhum dos experimentos descritos no capítulo 4.

- Violação na ordem das operações de memória: durante a execução do código paralelizado, foi realizada uma leitura de algum dado incorreto, fazendo com que os resultados da execução paralela possa divergir daqueles da execução sequencial correspondente.

- Exceções de *hardware* ou *software*: como a invocação de tratadores de interrupções (ou exceções) implicam em um desvio do fluxo de execução para um ponto fora do laço que está sendo executado, não faz sentido continuar a execução paralela.

A operação de *squash* é extremamente eficiente na arquitetura aqui proposta pois, a única ação necessária por parte do *hardware* é a invalidação das linhas de cache especulativas presentes nas caches do sistema. Contudo, note que esta invalidação não é responsável pela geração de nenhum tráfego adicional: como as linhas a serem invalidadas são especulativas, nenhuma escrita de volta para a memória principal é necessária. Portanto, esta solução é bastante eficiente em relação tanto ao desempenho, como ao consumo de energia.

Visto que falhas de especulação podem ocorrer, pelo menos um elemento de processamento no sistema deve ser capaz de realizar *checkpoints* (por exemplo, o *core* que realiza as operações de *commit*). Repare que, como como o estado da memória é mantido consistente pelas caches, a operação de *checkpoint* consiste apenas do armazenamento do estado sequencial dos registradores. Desta forma, mediante a ocorrência de alguma falha de especulação, o sistema é capaz de restaurar completamente o estado sequencial da aplicação.

Por fim, note que uma violação pode ser detectada erroneamente devido ao problema de *false sharing* [11]. Para evitar este problema, o compilador pode fazer uso de técnicas como *padding* quando não conseguir determinar estaticamente se dois dados distintos presentes na mesma linha de cache serão lidos/escritos concorrentemente. Esta técnica foi utilizada nos *benchmarks* usados para avaliar o mecanismo, e uma possível perda de desempenho decorrente da diminuição da localidade foi compensada pela ausência de *squashes* e pelo alto grau de paralelismo, conforme mostrado no capítulo 4. Se as técnicas de compilação ainda não forem suficientes, pode ser acrescentada uma *tag* nas linhas de *cache* com um *bit* para cada *byte* de dados da linha. Estes *bits* indicam quais *bytes* da linha foram lidos. Assim, uma escrita somente causaria um *squash* se fosse realizada em algum *byte* que já tenha sido lido. Além disso, esta *tag* precisaria estar presente apenas na *cache* compartilhada, já que a detecção de conflito pode ser feita apenas neste nível. Desta forma, pode-se evitar adicionar esta nova *tag* nos demais níveis da hierarquia de *caches*.

Depois da apresentação do mecanismo arquitetural proposto, a seção 3.5 explica, através de vários exemplos de execução detalhados, como as operações acontecem no nosso sistema, de modo a garantir que o resultado da execução paralela de um laço seja sempre equivalente àquele da execução sequencial correspondente. Tais exemplos mostram ainda como o nosso mecanismo funciona naturalmente com o protocolo de coerência de cache já existente. Embora não se trate de uma prova formal de corretude, espera-se que os exemplos exponham todas as possíveis situações que podem ocorrer em um sistema paralelo, e como o mecanismo aqui proposto trabalha em cada caso. Para uma prova formal de corretude, veja o apêndice A.

3.5 Exemplos

Com o intuito de simplificar a exposição, nos casos abaixo será utilizado o protocolo de coerência de cache MOSI [26], que é baseado em invalidações. A extensão dos exemplos abaixo para o protocolo de coerência de cache (também baseado em invalidações) MOESI [3], que é utilizado pelas máquinas com múltiplos núcleos da AMD, é trivial. Em todos os casos abaixo a seguinte notação será utilizada para os estados do protocolo de

coerência: M (*Modified*), O (*Owned*), S (*Shared*) e I (*Invalid*).

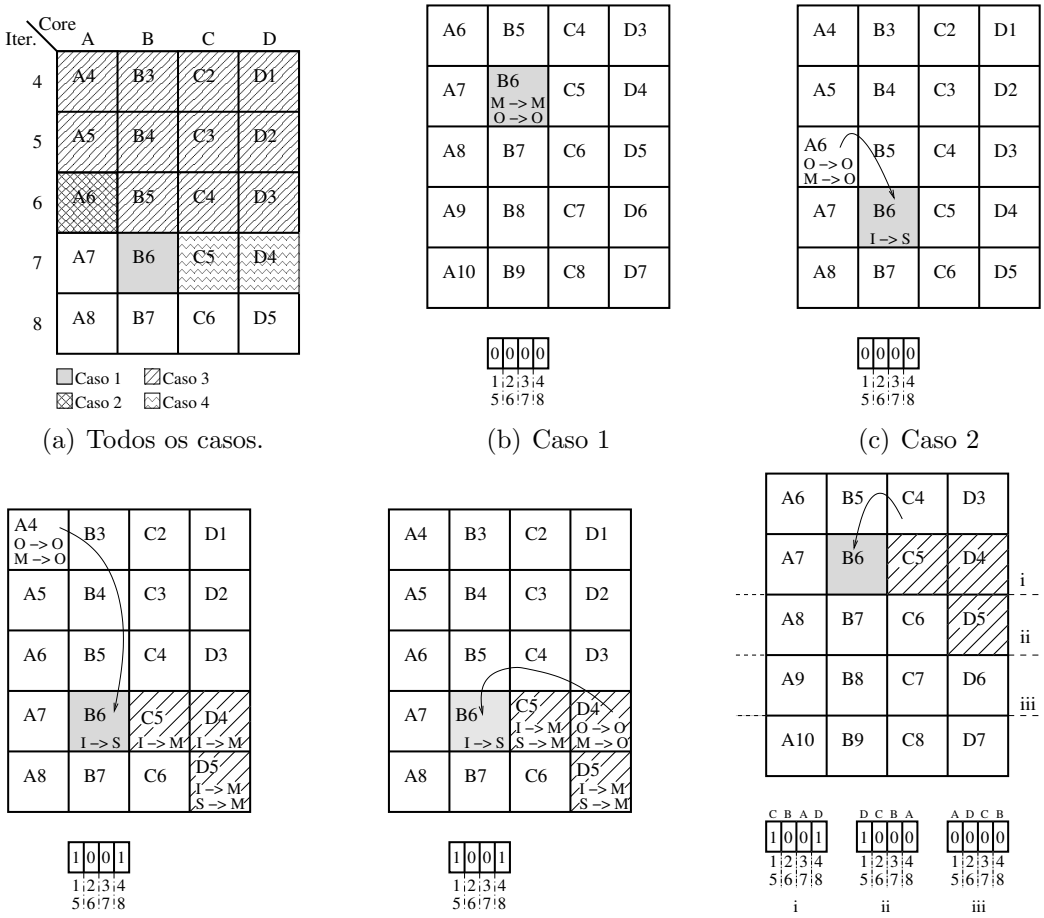
O leitor deve reparar que, em todos os casos, o mecanismo aqui apresentado não altera nenhum aspecto nem do protocolo de coerência de cache MOSI e nem de seu *hardware*. Ao invés disso, o mecanismo utiliza o comportamento natural do protocolo de coerência para prover um sistema de versionamento muito eficiente.

Os casos apresentados na figura 3.6 mostram todas as situações que podem ocorrer (veja a figura 3.6(a)) no momento que em que uma leitura do dado X é solicitada pelo estágio B na sexta iteração (B6). Quatro casos podem acontecer, dependendo de quando e em qual *core* foi produzido o dado utilizado pela leitura realizada no estágio B6. Para todos esses casos será mostrado que o sistema irá causar uma falha de especulação se um dependência *read-after-write* (RAW) for violada, implicando que o modelo de execução aqui proposto produz resultados corretos. Em cada um dos casos, o vetor de bits mostrado abaixo da figura correspondente indica a DepsTag do dado B_{X_6} (linha de cache X da iteração seis, presente no *core* B) depois que o *core* B recebeu o dado requisitado.

Caso 1 *O dado foi produzido na mesma iteração e no mesmo núcleo.*

A figura 3.6(b) ilustra o que acontece quando uma operação de leitura solicita uma linha de cache que foi escrita pelo mesmo núcleo e na mesma iteração em que a linha é solicitada. Na figura 3.6(b) a linha de cache X é lida no estágio de execução B6, mas a linha X já havia sido escrita no próprio B6. Note que, quando esta escrita aconteceu, a linha de cache B_{X_6} foi levada para o estado M pelo protocolo de coerência de cache. Agora, quando o estágio B6 realiza uma leitura de X, esta linha de cache pode estar somente nos estados M ou O (no caso de outro núcleo já ter feito uma solicitação da linha modificada X antes do estágio B6 solicitar a mesma linha). A linha B_{X_6} não pode estar nos estados S ou I do protocolo de coerência de cache porque apenas o estágio B está executando a iteração seis neste momento, e portanto nenhum outro núcleo pode ter escrito na linha de cache X na iteração seis (tal escrita iria alterar o estado da linha B_{X_6}) depois que o estágio B6 escreveu na linha X. Desta forma, a operação de leitura realizada no estágio B6 será satisfeita por um *hit* na cache, e este é obviamente o valor correto para esta leitura da linha X. Repare também que, como nenhum bit foi ativado na DepsTag da linha de cache B_{X_6} (a DepsTag é mostrada abaixo do *grid*), não acontecerá nenhuma falha de especulação por causa desta operação de leitura realizada no estágio B6.

É importante notar que uma operação de leitura nunca irá supor que uma linha de cache foi produzida na mesma iteração se isso não for verdade. Por exemplo, o leitor poderia pensar de maneira equivocada que, como as iterações dois e seis compartilham o mesmo endereço (ambas as linhas de cache B_{X_2} e B_{X_6} são representadas pelo endereço $B_{X_{0100}}$) e nossa abordagem implementa *lazy commits* (quando a iteração dois sofre *commit*, a linha de cache B_{X_2} não é nem escrita de volta na memória principal imediatamente, nem



(a) Todos os casos. (b) Caso 1 (c) Caso 2

(d) Caso 3 (e) Caso 4 (f) Execução detalhada.

Figura 3.6: Todos os casos que podem ocorrer quando uma operação de leitura é realizada no estágio B6.

é invalidada), uma operação de leitura feita no estágio B6 poderia ser satisfeita por um *hit* na cache porque a iteração dois deixou a linha $B_{X_{0100}}$ nos estados M ou O (novamente, ambas as linhas B_{X_2} e B_{X_6} são representadas pelo endereço $B_{X_{0100}}$). Contudo, quando a iteração dois sofreu *commit*, a linha de cache B_{X_2} foi alterada para B_{X_0} e portanto, enquanto o núcleo B não ler ou escrever na linha X na iteração seis (isto é, no estágio B6), a linha de cache B_{X_6} nem mesmo existirá no sistema. Logo, se existir qualquer operação de leitura que precede uma escrita no estágio B6, a primeira dessas leituras não irá encontrar uma linha de cache X_6 presente no núcleo B, causando um *miss* na cache, e sendo tratada portanto por um dos casos a seguir.

Caso 2 *O dado foi produzido na mesma iteração, mas em um núcleo diferente.*

A figura 3.6(c) ilustra a mesma operação de leitura do caso anterior, mas desta vez assumindo que o valor mais apropriado (note que o valor "mais apropriado" é o valor mais próximo, na ordem de execução sequencial, que já foi produzido no sistema no momento em que a leitura realizada no estágio B6 acontece) foi produzido na iteração seis, idependentemente de qual núcleo o fez. Se o núcleo A fornece a linha de cache X_6, tal núcleo deve ter essa linha nos estados M ou O, de acordo com o protocolo de coerência de cache. O núcleo A não poderia ter essa linha nos estados S ou I do protocolo porque isso indicaria que um outro núcleo escreveu na linha de cache X_6 depois de A (estado I do protocolo) ou antes de uma leitura realizada por A (estado S do protocolo). Em ambos os casos, não teria sido o núcleo A que produziu a linha de cache X_6, como estamos assumindo.

Como a linha de cache X solicitada foi produzida na iteração seis, nenhum bit na DepsTag da linha B_{X_6} será ativado, e portanto não haverá nenhuma falha de especulação por causa desta operação de leitura realizada no estágio B6. Tal comportamento é o esperado toda vez que o valor mais apropriado para uma operação de leitura tiver sido produzido na mesma iteração em que essa operação acontece, pois este valor é o correto para esta leitura (de acordo com a ordem de execução sequencial). Por fim, em tais casos, o protocolo de coerência de cache garante *naturalmente* que o valor mais recente produzido na mesma iteração será utilizado pela operação de leitura. Isso ocorre porque a linha de cache com o valor mais recente (produzido na iteração seis) é a única linha X_6 que *deve* estar ou no estado M ou no estado O do protocolo de coerência de cache. Por exemplo, se a linha X for lida no estágio de execução C6 e se X tiver sido escrita em ambos os estágios A6 e B6, o valor escrito em B6 seria fornecido porque quando a escrita na linha X aconteceu no estágio B6, o protocolo de coerência de cache invalidou a linha X_6 do núcleo A (A_{X_6}). Isto ocorre porque quando X_6 é escrita em B6, todas as linhas X_6 no sistema precisam ser invalidadas. Se isso não acontecesse, teríamos diferentes linhas

X_6 no sistema com valores distintos. O leitor deve reparar que o protocolo de coerência não está sendo modificado, já que todas essas ações já fazem parte dele.

Caso 3 *O dado foi produzido em uma outra iteração, em um estágio de execução que já foi finalizado.*

Este caso (mostrado pela figura 3.6(d)) ilustra uma operação de leitura realizada durante a execução do estágio B6, supondo que a linha de cache mais apropriada para essa leitura foi produzida pelo núcleo A na quarta iteração (isto é, no estágio A4). Repare que a linha de cache X_4 do núcleo A (A_{X_4}) deve estar ou no estado M ou no estado O do protocolo de coerência de cache. Se esta linha estivesse no estado S ou I do protocolo, isso indicaria que a linha de cache mais apropriada para a leitura no estágio B6 foi produzida em um outro núcleo, contrariando a nossa hipótese. Além disso, o protocolo de coerência garante naturalmente que nenhuma outra linha de cache X_4 pode estar nos estados M ou O. Logo, a linha A_{X_4} será utilizada pela operação de leitura no estágio B6.

Note que o quarto e o quinto bit são ativados na DepsTag da linha de cache B_{X_6} por causa desta operação de leitura, já que o valor lido foi produzido durante a iteração quatro. Uma falha de especulação da iteração seis deve acontecer por causa desta leitura se a linha de cache X for escrita na quarta ou na quinta iteração, pois tais valores seriam mais apropriados para uma leitura realizada na iteração seis do que um valor que foi produzido na iteração quatro. E repare que é exatamente isso que acontece: somente ocorrerá uma falha de especulação da sexta iteração quando o núcleo B detectar uma escrita na linha de cache X sendo feita ou na iteração quatro ou na iteração cinco, uma vez que os bits quatro e cinco estão ativados na DepsTag da linha B_{X_6}. Além disso, o leitor deve notar que:

- O quarto e o quinto bit da linha B_{X_6} serão desativados apenas no momento em que as iterações quatro e cinco (respectivamente) sofrerem *commit*, e este é exatamente o ponto a partir do qual essas iterações não podem escrever mais nada na linha de cache X, e portanto a linha X_6 não pode mais ser anulada, já que com certeza possui o valor correto.

- As iterações que compartilham os mesmos bits na IDTag (por exemplo, o mesmo bit é compartilhado pelas iterações um, cinco, nove, etc. em uma máquina com quatro elementos de processamento) jamais irão interferir umas com as outras, já que quando a execução da iteração cinco é iniciada, a iteração um já sofreu *commit* (só podem haver quatro iterações ativas no sistema, já que este possui quatro *cores*), e quando a execução da iteração nove começar, a iteração cinco já terá passado pelo *commit*. O mesmo raciocínio pode ser utilizado para o bit quatro com as iterações quatro, oito, doze e assim por diante.

- Depois que a leitura da linha de cache X é realizada no estágio B6, a primeira escrita que venha a ocorrer nesta linha nas iterações quatro ou cinco irá gerar uma transação de barramento, e portanto o núcleo B detectará tal escrita. Considere por exemplo a iteração cinco. A primeira vez que for realizada uma operação de escrita na linha de cache X pelo estágio C5 (ainda não aconteceu nenhuma escrita na linha X durante a execução do estágio C5, pois neste caso esta linha seria mais apropriada para a operação de leitura realizada no estágio B6 do que a linha que foi produzida na iteração quatro), se a linha X já tiver sido lida em C5, C_{X_5} deve estar no estado S do protocolo de coerência, já que como esta é a primeira vez que a linha de cache X é escrita no estágio C5, a linha C_{X_5} não poderia ter ido para o estado M do protocolo. Por outro lado, se a linha X ainda não tiver sido lida no estágio C5, então a linha C_{X_5} ainda não existe no sistema. Logo, em ambos os casos, a primeira operação de escrita na linha de cache X durante a execução do estágio C5 irá gerar uma transação de barramento, de acordo com o funcionamento natural do protocolo de coerência de cache já existente.

Caso 4 *O dado foi produzido em uma outra iteração por um estágio que está sendo executado concorrentemente.*

Neste caso, ilustrado pela figura 3.6(e), quando acontece, no estágio de execução B6, uma operação de leitura que solicita a linha de cache X, esta linha já foi escrita durante a execução do estágio D4, sendo que este é o valor mais apropriado para a leitura no estágio B6. Logo após receber a linha de cache solicitada, os bits quatro e cinco são ativados na DepsTag da linha X_6 no núcleo B (B_{X_6}). Desta forma, operações de escrita na linha de cache X realizadas nas iterações quatro e cinco devem causar uma falha de especulação, já que os valores escritos nestas iterações serão mais apropriados (de acordo com a ordem de execução sequencial) para a leitura no estágio B6 do que o valor que foi produzido na iteração quatro.

Para a iteração cinco, a primeira operação de escrita realizada na linha de cache X no estágio C5 (note que a linha X ainda não foi escrita no estágio C5, pois este valor seria mais apropriado para a leitura realizada em B6 do que o valor que foi escrito no estágio D4) irá gerar uma transação de barramento, já que a linha C_{X_5} ainda não existe no sistema. O mesmo raciocínio é válido para o estágio de execução D5 e a linha de cache D_{X_5}.

Para a iteração quatro, o núcleo D é o único que ainda pode realizar uma operação de escrita na linha X, pois os outros núcleos já terminaram de executar seus estágios da quarta iteração. Contudo, como o bit quatro está ativado na DepsTag da linha de cache B_{X_6}, e o protocolo de coerência de cache obriga o núcleo D a gerar uma transação de barramento para realizar a operação de escrita (quando a linha escrita no estágio D4 foi

lida em B6, esta linha de cache foi para o estado O do protocolo de coerência), o sistema irá se comportar da maneira correta.

Finalmente, a figura 3.6(f) ilustra a evolução do sistema no decorrer no tempo. Inicialmente, uma operação de leitura solicitando a linha de cache X ocorre no estágio de execução B6. O valor mais apropriado para satisfazer essa leitura foi produzido na iteração quatro. Obviamente, no momento em que a linha X é solicitada no estágio B6, tal linha ainda não foi escrita nem no estágio C5, nem em D4, já que em ambos os casos haveria um valor mais apropriado para a leitura no estágio B6 do que aquele que foi utilizado para satisfazer tal operação. No passo (i), apenas os núcleos C e D podem causar uma falha de especulação por causa da leitura realizada no estágio B6; note que os bits quatro e cinco estão corretamente ativados na DepsTag da linha de cache B_{X_6} (a evolução desta DepsTag é mostrada abaixo do *grid* da figura 3.6(f)). No passo (ii), como a iteração quatro acabou de sofrer uma operação de *commit*, apenas operações de escrita realizadas no estágio D5 podem causar uma falha de especulação, já que este estágio de execução é o único que ainda pode escrever na linha de cache X na iteração cinco. No passo (iii), as iterações quatro e cinco já sofreram *commit*, e portanto os bits na DepsTag da linha de cache B_{X_6} já foram desativados; mas agora não podem haver mais falhas de especulação devidas à operação de leitura realizada pelo estágio B6, e portanto o sistema se comporta da forma correta. Novamente, repare que o bit que corresponde à iteração cinco, por exemplo, não é influenciado por operações nas iterações um e nove (que compartilham o mesmo bit com a iteração cinco) porque quando a execução da iteração cinco foi iniciada, a iteração um tinha acabado de sofrer uma operação de *commit*, e quando começar a execução da nona iteração, a iteração cinco já terá sofrido *commit*.

Capítulo 4

Resultados Experimentais

O mecanismo arquitetural aqui proposto foi implementado na ferramenta SESC [36], um simulador arquitetural *cycle-accurate*. Foi simulado um sistema com 24 *cores* interconectados por um barramento de 256 bits. Este número de *cores* foi escolhido por dois motivos. Primeiramente, as máquinas com maior número de núcleos atualmente possuem 24 núcleos. Segundo, o único mecanismo proposto [29] até então capaz de aceitar a execução especulativa de aplicações paralelizadas por técnicas DOPIPE com replicação de estágios utilizou uma máquina de 24 núcleos para avaliação de desempenho. Desta forma, decidimos utilizar o mesmo número de elementos de processamento para fins de comparação.

Esta máquina possui caches de dados L1 privadas com política de escrita *write-back*, 32KB de capacidade, associatividade quatro, linhas de 32 *bytes*, política de reposição *least-recently-used*, latências de *hit* e *miss* de dois ciclos, com duas portas (uma para acessos do núcleo e outra para acesso ao barramento compartilhado). As caches L1 de instruções também são privadas e possuem 32KB de capacidade, política de escrita *write-through*, associatividade dois, latências de *hit* e *miss* de um ciclo, política de reposição de linhas *least-recently-used* e com duas portas de acesso, assim como a cache dados.

A cache de dados L2 é uma cache inclusiva (isto é: todas as linhas de cache presentes nas caches de dados privadas L1 de todos os *cores* também estão presentes na cache L2) compartilhada entre todos os elementos de processamento. Tal cache possui 32MB de capacidade de armazenamento, linhas de 32 *bytes*, associatividade 32, política de reposição de linhas *least-recently-used*, latência de *hit* de nove ciclos, latência de *miss* de onze ciclos e política de escritas *write-back*. A memória principal possui uma latência de acesso de 500 ciclos, com uma porta de acesso. Máquinas modernas com o mesmo número de *cores*, como a família Intel Xeon, possuem caches de 30MB no último nível da hierarquia. Ambos os processadores POWER7 [4] e BlueGene/Q [16] da IBM possuem caches de 32MB no último nível. Aqui foram utilizados 32MB devido ao fato do simulador utilizado exigir

que a capacidade de armazenamento das caches sejam uma potência de dois.

Os *buffers* de tradução de memória (TLB) de dados e de instruções possuem 512 *bytes* de capacidade de armazenamento, associatividade quatro, blocos de oito *bytes*, política de reposição de blocos *least-recently-used*, com duas portas de acesso.

Cada núcleo de processamento possui uma unidade de *loads* com latência de um ciclo, uma unidade de *stores* com a mesma latência, uma unidade de multiplicações de inteiros com quatro ciclos de latência, uma ALU de inteiros e uma de ponto flutuante, ambas com latências de um ciclo, uma unidade de divisão de ponto flutuante com dez ciclos de latência, uma unidade de multiplicação de ponto flutuante com dois ciclos de latência e uma unidade de divisão de inteiros com latência de doze ciclos.

No sistema simulado não existe nenhum mecanismo especial para a transferência de dados entre as caches. Tais transferências sempre acontecem através do barramento compartilhado, da mesma forma que é feito em máquinas atuais com múltiplos núcleos. O custo de tais transferências é composto por dois ciclos decorrentes da latência de *miss* na cache L1, somados ao número de ciclos necessários para acessar o barramento (este número é variável e dependente da contenção no barramento no momento em que o acesso é realizado) mais a latência de transferência intrínseca do barramento. Além disso, são adicionados o número de ciclos necessários para realizar o acesso à cache que fornecerá o dado solicitado (tal número também é variável e depende da contenção na cache correspondente no momento em que o acesso é realizado) mais um custo fixo de dois ciclos para acessar o dado requisitado. Por fim, quando a cache envia o dado solicitado, os custos (acesso ao barramento, à cache solicitante, etc.) são calculados da mesma forma que foi feito durante o processo de requisição. Os custos associados ao processo de resposta são então adicionados ao custo total da transferência de dados.

Foram selecionados alguns *benchmarks* das suítes SPEC CINT e CFP [38], duas funções importantes do aplicativo de processamento de imagens GIMP [14] e dois *benchmarks* da suíte PARSEC [6] que possui diversas aplicações com o intuito de avaliar o desempenho de computações realizadas em máquinas paralelas. Tais *benchmarks* fazem uso intenso de CPU e requerem a utilização de especulação para que o processo de paralelização produza códigos eficientes e escaláveis. Para encontrar laços e funções candidatos à paralelização especulativa foi utlizado *profiling* a nível de laços e informações de análise da infra-estrutura de compilação LLVM [20]. Os programas foram paralelizados manualmente da mesma forma que um compilador moderno faria. Embora já exista um mecanismo de suporte para a execução especulativa de aplicações paralelizadas por técnicas DOPIPE combinadas com replicação de estágios [29], ainda não existe um compilador capaz de aplicar tais técnicas em códigos sequenciais a fim de gerar os códigos paralelos automaticamente. Devido à dificuldade inerente à aplicação das técnicas de compilação e paralelização de forma manual, o processo de seleção de *benchmarks* foi influenciado pela

Benchmark	Suíte	Função	Representatividade dinâmica
130.li	SPEC CINT	`main`	100%
164.gzip	SPEC CINT	`deflate`	99%
179.art	SPEC CFP	Internal loop	25%
256.bzip2	SPEC CINT	`compressStream`	99%
blackscholes	PARSEC	`main`	100%
swaptions	PARSEC	`main`	100%
crc32	Ref. Implem.	`main`	100%
gimp-nova	GIMP	`nova`	93%
gimp-oilify	GIMP	`oilify`	99%

Tabela 4.1: Detalhes dos *benchmarks*.

tratabilidade do código fonte sequencial. Além disso, a seleção de aplicações levou em consideração a diversidade em termos de paradigmas de paralelização (Spec-PS-DSWP, Spec-DOALL, etc.) e tipos de especulação necessários (especulação de dados, especulação de fluxo de controle, etc.). Por fim, a escolha de *benchmarks* levou em conta a necessidade de prover uma comparação de desempenho entre o mecanismo arquitetural aqui proposto e o mecanismo já existente na literatura [29].

A tabela 4.1 apresenta os *benchmarks* selecionados juntamente com informações referentes a qual suíte de programas pertencem, qual a parte de cada aplicação foi paralelizada e a porcentagem do tempo de execução da parte paralelizada em relação ao tempo total de execução (representatividade dinâmica). Por exemplo, suponha que, para a aplicação `A`, a função `f()` desta aplicação foi paralelizada. Suponha que o tempo necessário para executar `A` sequencialmente seja 90 minutos. Se destes 90 minutos, 45 minutos foram gastos executando `f()`, então a representatividade dinâmica de `f()` é 50%. Da mesma forma, se ao invés de 45 minutos tivessem sido gastos 80 minutos executanto `f()`, a representatividade dinâmica de `f()` seria 88,88%.

4.1 Resultados e Análise

A figura 4.1 apresenta os ganhos de desempenho da parte paralelizada de cada *benchmark* relativos ao código sequencial correspondente sem a aplicação de qualquer modificação. Como pode ser observado pela última coluna da tabela 4.1, com exceção da aplicação `179.art`, tais ganhos de desempenho são próximos ou iguais aos ganhos de desempenho da aplicação completa. Uma descrição detalhada a respeito dos tipos de especulação e paralelização utilizados para cada um dos *benchmarks* é feita abaixo.

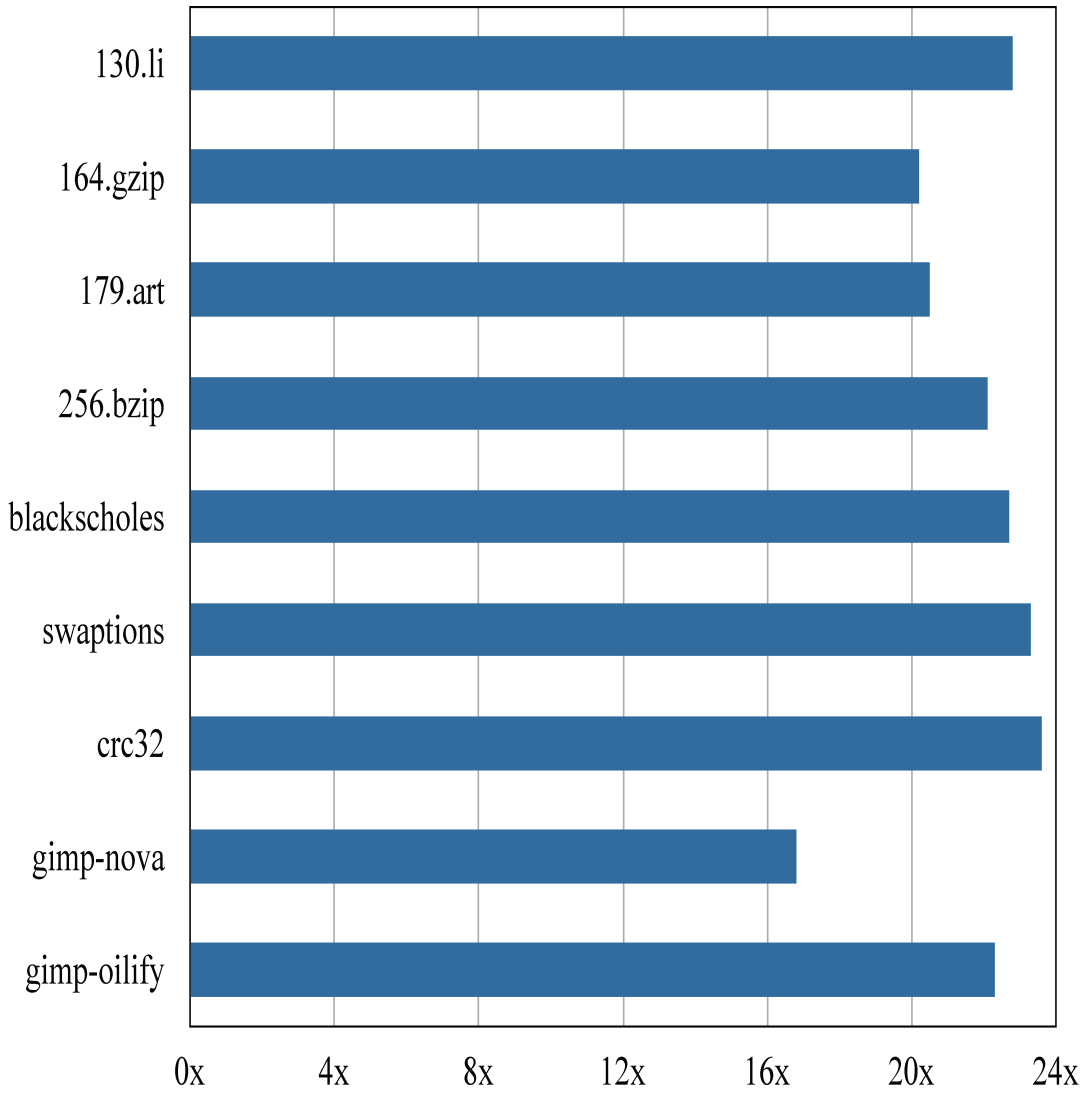


Figura 4.1: Ganhos de desempenho alcançados com a paralelização especulativa de cada aplicação.

- `130.li` é um interpretador Lisp com programação orientada a objetos. Este *benchmark* processa de maneira sequencial um conjunto de *scripts* especificados como entrada do programa. Para que tais *scripts* possam ser processados concorrentemente é necessário especular que o fluxo de controle não será desviado para a saída do programa. Além disso, é necessário versionar várias variáveis globais relacionadas ao ambiente de execução.

- `164.gzip` é um compressor e descompressor de arquivos. A parte responsável pela compressão de arquivos foi paralelizada com o uso de especulação, formando assim um *pipeline* de execução com três estágios: o primeiro estágio lê e armazena a próxima parte a ser comprimida do arquivo de entrada, o segundo estágio realiza a compressão dos dados lidos pelo estágio anterior e o terceiro estágio escreve os dados comprimidos no *buffer* de saída. O segundo estágio apresenta dependências entre iterações, o que impede que a compressão seja realizada em paralelo. Contudo, o versionamento de memória quebra tais dependências de maneira automática, permitindo assim que os blocos de dados possam ser comprimidos em paralelo. Outra dependência apresentada pelo segundo estágio que previne a compressão paralela é que o ponto onde a compressão do próximo bloco será iniciada é conhecido somente depois que o bloco atual já tiver sido comprimido. Para quebrar essa dependência foi utilizada a técnica Y-branch [7], fazendo com que a compressão de novos blocos seja iniciada em pontos fixos.

- `179.art` consiste na utilização de uma rede neural aplicada ao reconhecimento de imagens. O laço mais significativo desta aplicação requer que o versionamento de memória seja utilizado a fim de quebrar dependências, fazendo com que as iterações do laço mais externo possam ser executadas de forma paralela.

- No *benchmark* `256.bzip2` o bloco de dados lido do arquivo de entrada tem o seu CRC calculado, e este número é necessário para calcular o CRC do próximo bloco a ser lido. Uma dependência semelhante ocorre no memento de escrever o bloco de dados compactado no *buffer* de saída. A presença dessas dependências impede que as iterações do laço mais externo sejam executadas concorrentemente. Combinando a técnica DSWP com especulação é possível colocar cada uma dessas dependências em um estágio de execução separado, permitindo assim que um *pipeline* de três estágios seja formado. O primeiro desses estágios deve ser sequencial por causa da dependência de CRC citada anteriormente. No entanto, fazendo com que o CRC seja local para cada bloco, esta dependência pode ser transferida para o segundo estágio do *pipeline*. Neste estágio, o tamanho da estrutura de dados utilizada para armazenar os dados lidos do arquivo de entrada varia a cada iteração. Como os valores de tais variações não podem ser determinados em tempo de compilação, este

estágio não pode ser paralelizado. Contudo, o versionamento de memória privatiza essa estrutura de dados automaticamente em tempo de execução fazendo com que este estágio seja replicado e executado em paralelo. Além disso, o versionamento de memória é utilizado para quebrar dependências de memória que não ocorrem em tempo de execução e especulações de fluxo de controle são utilizadas para especular que condições de erro são sempre falsas.

- `blackscholes` é um *benchmark* RMS da Intel que utiliza a equação diferencial parcial Black-Scholes para calcular os preços das opções de um portfólio europeu analiticamente. Nesta aplicação é necessário aplicar especulações de fluxo de controle para permitir que as iterações do laço mais externo sejam processadas em paralelo.

- `gimp-nova` é uma transformação artística que insere uma estrela e os efeitos de luminosidade associados em uma dada imagem. Aqui, o versionamento de memória se faz necessário a fim de quebrar dependências de memória que não se manifestam em tempo de execução, permitindo assim que as linhas da imagem na qual a estrela será inserida possam ser processadas paralelamente.

- `swaptions` é um outro *benchmark* RMS da Intel que utiliza o arcabouço Heath-Jarrow-Morton para calcular os valores de um conjunto de operações de *swap*. Esta aplicação realiza simulações utilizando o algoritmo de Monte Carlo para computar os valores. Especulações de condições de erro se fazem necessárias para que possa ser extraído paralelismo do laço mais externo.

- `gimp-oilify` se trata de uma outra transformação artística que faz com que uma determinada imagem fique com a aparência de uma pintura a óleo. Especulações de fluxo de controle são utilizadas em algumas condições e o versionamento de memória quebra automaticamente algumas dependências entre iterações. Essa combinação torna possível que o laço mais externo possa ser executado em paralelo.

- O *benchmark* `crc32` simplesmente calcula o CRC de 32 bits dos arquivos especificados na entrada do programa. Especulações de fluxo de controle são necessárias para permitir que as computações dos CRCs possam ser realizadas concorrentemente.

Com o intuito de prover um mecanismo para a execução especulativa de aplicações em um modelo de *pipeline*, Raman *et. al* [29] propôs o mecanismo SMTX que faz uso de uma unidade central para operações de *commit*. Esta unidade é responsável por re-executar as operações de leitura e escrita em memória realizadas por todos os outros *cores* do sistema que estão sendo utilizados para a execução da aplicação paralelizada. A repetição de tais operações tem por finalidade detectar possíveis violações na ordem dos acessos à memória realizados pelos diferentes elementos de processamento.

Todavia, conforme mostrado em [29], essa unidade central é um fator limitante para a escalabilidade do desempenho de uma série de aplicações que realizam muitos acessos à memória, saturando assim tanto a unidade de *commit* como o sistema de memória da máquina na qual o mecanismo está sendo utilizado. Para tais aplicações o desempenho do mecanismo SMTX é afetada consideravelmente.

A figura 4.2 apresenta uma comparação realizada entre a técnica SMTX e o mecanismo arquitetural aqui proposto. Note que as comparações foram feitas apenas para o subconjunto de *benchmarks* que foram utilizados para avaliar o desempenho de *ambas* as estratégias. Repare que a natureza distribuída da arquitetura aqui proposta garante a escalabilidade de desempenho para todas as aplicações. De uma forma específica, para os *benchmarks* `164.gzip`, `gimp-nova` e `gimp-oilify`, a diferença de desempenho entre o SMTX e o mecanismo aqui apresentado é dramática. Note também que, mesmo para as aplicações que não são críticas para o mecanismo SMTX, o modelo aqui proposto atinge desempenhos superiores.

4.2 *Overhead* de Falhas de Especulação

Quando o mecanismo arquitetural detecta alguma violação na ordem em que as operações de memória foram realizadas, o sistema precisa desfazer a parcela da computação que foi realizada de maneira equivocada e re-executar o trecho de código correspondente de forma sequencial, para garantir que os resultados gerados sejam corretos e a execução paralela possa ser reiniciada a partir daquele ponto.

A fim de modelar o custo das falhas de especulação, o simulador conta todos os ciclos gastos durante a execução do código paralelo até o ponto em que a violação foi detectada. São contados ainda os ciclos necessários para a re-execução sequencial do código durante a execução do qual a falha ocorreu. Por fim, a execução especulativa do código paralelo é retomada; o simulador nunca pára de contar ciclos. Além disso, é considerado uma penalidade fixa de 2000 ciclos para modelar os custos da sinalização da falha de especulação e para invalidar as linhas de cache especulativas de todas as caches presentes no sistema de memórias. Note que, durante este processo de invalidação, nenhum tráfego adicional é gerado no sistema, já que as linhas invalidadas são especulativas e portanto não devem ser escritas de volta na memória principal.

Para medir o impacto da ocorrência de falhas de especulação durante a execução do código paralelo nos ganhos de desempenho obtidos através do processo de paralelização combinado com técnicas de especulação, uma taxa de falhas de 1% foi injetada no *benchmark* `blackscholes`. Executando este experimento na mesma máquina simulada de 24 *cores*, o ganho de desempenho obtido foi de 18.2×. Este ganho é consideravelmente inferior ao ganho de 22.7× que foi obtido durante a execução da mesma aplicação paralela

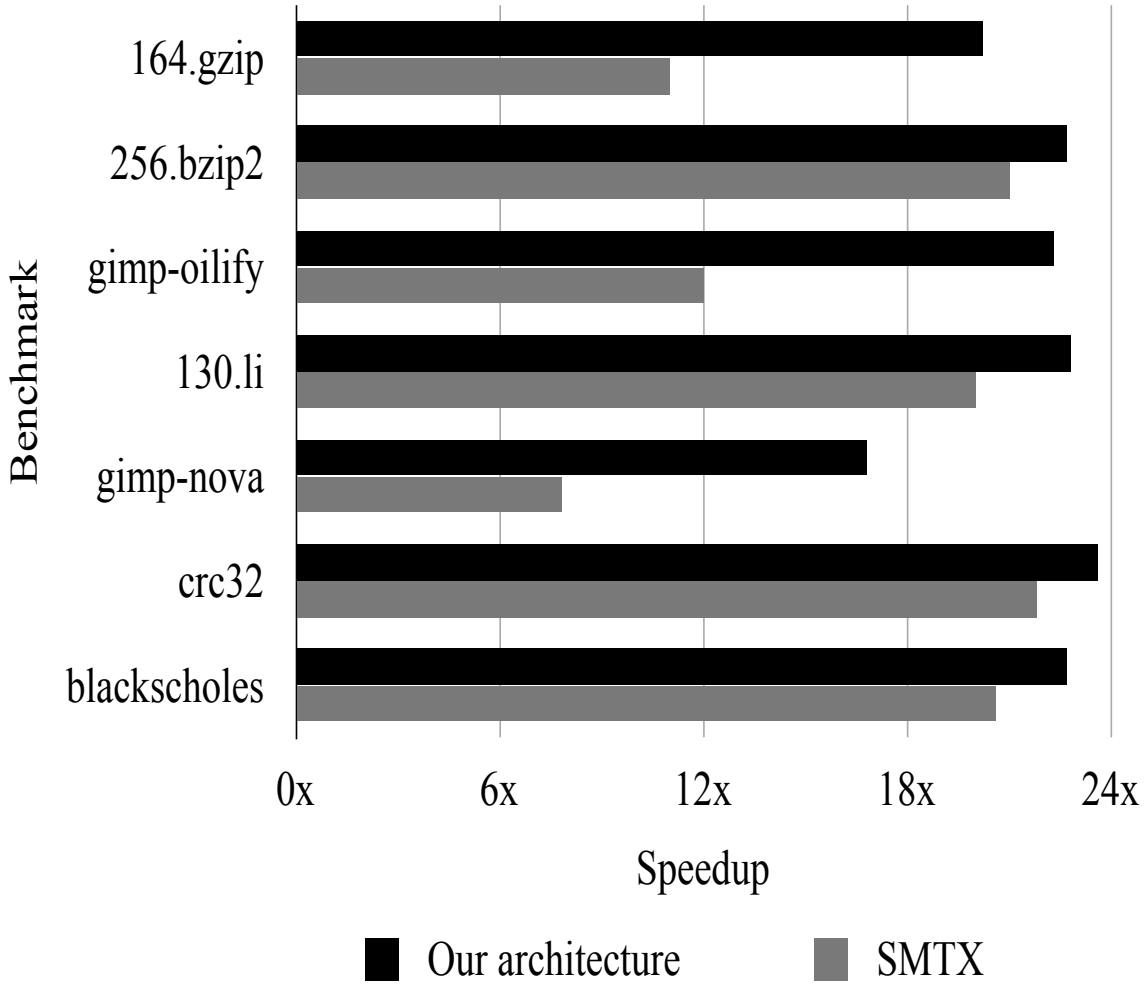


Figura 4.2: Comparação entre o arquitetural aqui proposto e o mecanismo SMTX.

sem a ocorrência de nenhuma violação de memória.

Foi realizada então uma análise com o intuito de detectar os principais fatores responsáveis por tal perda de desempenho. Dessa forma, foi observado que aproximadamente 99% deste *overhead* é proveniente da execução sequencial do trecho de código no qual a falha de especulação foi detectada durante a execução paralela, combinada com a execução (inútil) do código paralelo entre o ponto no qual a violação de ordenação de memória foi detectada e o último *checkpoint* realizado pelo sistema. Esta execução paralela é completamente descartada, já que não é possível garantir a corretude de nenhuma parcela da computação que já foi realizada depois do último *checkpoint*. A fim de mitigar o custo da re-execução sequencial do código responsável pela violação de dependência de memória, esta re-execução sequencial poderia ser paralelizada, respeitando todas as dependências (isto é, sem fazer uso de especulação). Esta possibilidade é deixada como um trabalho futuro.

Por fim, note que no mecanismo aqui proposto, a operação de *squash* envolve apenas a execução sequencial do trecho de código que causou a falha de especulação, simplesmente ignorando as linhas de cache especulativas presentes na hierarquia de memórias do sistema. Nenhum tráfego adicional é gerado no sistema de memória ou no barramento de interconexão compartilhado pelos *cores*. Logo, nenhuma parcela do *overhead* introduzido por eventuais falhas de especulação é inerente ao mecanismo arquitetural aqui proposto. Portanto, a perda de desempenho relacionada ao processo de recuperação de falhas de especulação corresponderá sempre à quantidade de ciclos necessária para executar sequencialmente o trecho de código no qual a violação na ordem das operações de memória ocorreu.

Capítulo 5

Trabalhos Relacionados

Trabalhos recentes a respeito de técnicas de paralelização de programas sequenciais [40, 7] têm demonstrado as vantagens de esquemas que combinam o modelo de execução em *pipeline* com especulação e replicação de estágios sobre esquemas tradicionais de paralelização (DOALL e DOACROSS). Através da incorporação de especulação às técnicas de paralelização baseadas em DOPIPE, é possível obter ganhos de desempenho escaláveis para aplicações de propósito geral executadas em máquinas com múltiplos elementos de processamento. No entanto, como nesses esquemas trechos das iterações de laços são espalhados pelos *cores* disponíveis, os suportes tradicionais de TLS descritos abaixo (tanto os que são baseados em *software* como aqueles baseados em *hardware*) não podem ser utilizados, já que seriam produzidos resultados diferentes daqueles produzidos pela execução sequencial correspondente. Por outro lado, o mecanismo aqui proposto aceita não apenas a execução de aplicações paralelizadas por meio de técnicas tradicionais, mas também a execução de programas paralelizados segundo o modelo em *pipeline* combinado com especulação e replicação de estágios (Spec-PS-DSWP [8]).

Um mecanismo arquitetural para TLS foi proposto por Steffan *et al.* [39]. Nesta abordagem, o protocolo de coerência de cache *Modified - Exclusive - Shared - Invalid* (MESI) é modificado com a introdução de novos estados, transições e mensagens de coerência, fazendo com que este mecanismo se torne pouco atraente para ser adotado pela indústria de *hardware*. Além disso, este mecanismo permite a geração de tráfego em rajada durante as operações de *commit* e *squash*, além de introduzir mensagens de coerência adicionais quando ocorrem escritas especulativas em linhas de cache sujas. Este modelo também depende de um módulo especial de *hardware* que, no momento em que uma tarefa sofre *commit*, solicita de maneira sequencial o *ownership* para um conjunto de linhas de cache cujos endereços ficam armazenados em um *buffer*. Desta forma, além de gerar tráfego em rajada, como durante as operações de *commit* e *squash* os processadores ficam parados, o desempenho é prejudicado e o sistema se torna pouco eficiente em relação ao consumo de

energia [33]. O mecanismo aqui proposto não sofre de nenhum desses problemas, já que o protocolo de coerência de cache não é modificado, e não é introduzido nenhum módulo de *hardware* especial. Ainda, nosso modelo elimina completamente a geração de tráfego em rajadas. Por fim, o mecanismo proposto por Steffan *et al.* permite apenas que iterações completas dos laços sejam distribuídas para os elementos de processamento em um modelo de execução semelhante ao modelo DOALL, mas que permite o uso de especulações. Por outro lado, o nosso mecanismo permite não apenas esse modelo de execução, mas também o modelo de execução em *pipeline* combinado com especulações e replicação de estágios.

Uma outra abordagem, proposta por Cintra *et al.* [10], introduz um novo protocolo de coerência de cache. Além disso, as operações de *commit* e *squash* realizam várias operações potencialmente lentas. As operações de *commit* envolvem a geração de tráfego em rajadas entre as caches L1, L2 e a memória principal, e a verificação de um conjunto de linhas de cache presentes no sistema. As operações de *squash* também necessitam de tal verificação, além de forçar a sincronização de todos os processadores em uma barreira. Ainda, todo o mecanismo arquitetural é dependente de *Memory Desambiguation Tables* (MDT) que não apenas precisam ser acessadas e buscadas em todo *miss* especulativo nas caches e em todas as operações de *squash*, mas também fazem com que os elementos de processamento do sistema fiquem parados quando as MDTs estão cheias e uma nova entrada precisa ser alocada. Além disto, essas tabelas precisam ser informadas a respeito de todas as leituras realizadas pelos processadores, independentemente de tais operações terem sido satisfeitas por *hits* ou terem sofrido *miss* na cache L1. Finalmente, este esquema não permite que linhas de cache sujas permaneçam nas caches L1 e L2 entre as inicializações das *threads*, fazendo com que cada nova *thread* inicie sua execução com a cache fria (exceto pela eventual presença de dados não-especulativos). Em contraste, o mecanismo arquitetural aqui proposto funciona com protocolos de coerência de cache já existentes e adotados pela indústria de *hardware*, e não introduz nenhum tipo de sincronização ou de geração de tráfego em rajadas. Também não é introduzida nenhuma estrutura de *hardware* especial (que é potencialmente um ponto único de falha e pode introduzir contenção no sistema) e é permitida a permanência de linhas de cache sujas nas caches depois das operações de *commit*, fazendo com que as iterações não precisem iniciar suas execuções com as caches frias, melhorando assim o desempenho do sistema.

O mecanismo em *hardware* proposto por Gopal *et al.* [15], embora não seja escalável, resolve o problema da geração de tráfego em rajada nas operações de *commit*. Todavia, tal mecanismo não apenas modifica o protocolo de coerência de cache, mas também faz uso de uma unidade de controle especial, introduzida com a finalidade de predizer as tarefas que deverão ser executadas por cada elemento de processamento presente no sistema, introduzindo assim operações de *squash* quando erros nas predições de tarefas são detectados. Além disso, essa abordagem introduz um mecanismo centralizado de

hardware responsável pelo gerenciamento de todo o processo de execução especulativa, o que representa um ponto único de falha no sistema. Finalmente, em *todos* os *misses* na cache, ou mesmo nas operações de leitura de linhas de cache que já sofreram *commit*, este *hardware* centralizado precisa atravessar uma lista ligada de linhas de cache versionadas espalhadas por todas as caches de todos os processadores da máquina. Por outro lado, além de não introduzir modificações no protocolo de coerência de cache, o modelo aqui proposto não depende de nenhum *hardware* ou estrutura especial centralizada para gerenciar o processo de execução especulativa, evitando assim a ocorrência de contenção (o que melhora o desempenho), tornando a nossa abordagem mais atrativa do ponto de vista das indústrias de *hardware*. Ainda, a abordagem aqui proposta não faz uso de nenhuma unidade de predição de tarefas a serem executadas, evitando assim a geração de *squashes* desnecessários (que são as operações responsáveis pelas maiores perdas de desempenho, conforme mostrado na seção 4.2). Tais operações de *squash* desnecessárias também ocorrem na abordagem proposta por Sohi *et al.* [37], que também faz uso de um *hardware* especial que realiza a predição de tarefas a serem executadas em um modelo de execução hierárquica.

A abordagem para suporte a TLS em *hardware* proposta por Akkary *et al.* [1] se diferencia dos esquemas usuais no sentido de permitir que a paralelização seja realizada em tempo de execução. Neste caso, além de prover o suporte usual para o uso de especulação, este mecanismo realiza uma busca por funções e laços que possam ser executados concorrentemente (por exemplo, se não há dependências entre duas chamadas de funções). Mesmo quando a presença de dependências impede a execução paralela, a arquitetura utiliza um preditor de dados para especular o valor de dependências que atrasam o início da execução de funções e iterações. Caso o preditor erre os valores, ocorre uma falha de especulação. Por outro lado, a abordagem proposta por Balakrishnam *et al.* [5] não faz uso de preditores para quebrar dependências dinamicamente, respeitando as dependências expostas pelo compilador. Dessa forma, o *hardware* só executa uma função, por exemplo, depois que suas dependências já estão satisfeitas. Tais dependências não são, no entanto, determinadas pelo *hardware*: este espera que tais informações sejam providas pelo compilador. Infelizmente, tais abordagens não aceitam o modelo de execução em *pipeline* combinado com especulação, limitando a escalabilidade de desempenho obtida para aplicações de propósito geral.

A arquitetura proposta por Hammond *et al.* introduz *hardwares* especializados dentro e fora dos processadores a fim de suportar TLS. É introduzido também um barramento especial para facilitar as comunicações dos processadores com a cache compartilhada e com os demais elementos de processamento presentes no sistema. Por fim, deve-se notar que, ao contrário do mecanismo aqui proposto, essa arquitetura não apenas promove a geração de tráfego em rajada (o que degrada o desempenho e a eficiência de energia),

mas também não permite a execução de códigos paralelizados por técnicas baseadas em DOPIPE combinadas com especulação e replicação de estágios.

Assim como na abordagem proposta por Balakrishnam *et al.* [5], Krishnan *et al.* [18, 19] faz uso de estruturas especiais de *hardware* para suportar TLS e paralelizar o código em tempo de execução, contando com informações embutidas no código binário a respeito de quais laços e funções devem ser paralelizados. Por fim, são introduzidos ainda *hardwares* especiais para lidar com dependências de dados , além das *Memory Desambiguation Tables* (MDT) que também são utilizadas por Cintra *et al.*, conforme discutido anteriormente. Assim, além de contar com os problemas inerentes ao uso de MDTs, pode-se reparar que essa abordagem requer a adição de muitas estruturas especiais de *hardware*, tornando-a pouco atrativa para a indústria. Outro mecanismo em *hardware* para TLS na qual a paralelização de programas sequenciais é realizada pela própria arquitetura foi proposta por Marcuello *et al.* [23]. Assim como no mecanismo proposto neste trabalho, essa abordagem faz uso de especulações de dependências de dados e de fluxo de controle a fim de extrair maior paralelismo de laços externos das aplicações.

Zhong *et al.* [45] propõem um mecanismo para TLS capaz de suportar paralelismo a nível de laços e de *threads* utilizando um módulo de execução separado com memórias transacionais em *hardware*. Contudo, esse mecanismo vai além: assim como foi proposto em *Wisconsin Decoupled Grid Execution Tiles* [43], essa abordagem realiza a fusão de núcleos de processamento a fim de melhor explorar o paralelismo a nível de instruções. Infelizmente, é conhecido [35] que tal fusão não é eficiente na prática. Além disso, não existe o suporte para a coerência de dados nesta arquitetura, sendo responsabilidade do compilador lidar com todas as questões relativas à coerência no momento de gerar código. O compilador deve ser portanto, consideravelmente mais elaborado que os compiladores usuais para que o código gerado seja eficiente. Outra abordagem que utiliza o agrupamento de núcleos de processamento foi proposta por Madriles *et al.* [22] através da introdução de um *hardware* especializado responsável por manter a coerência de operações de memória entre os processadores. Assim como no mecanismo arquitetural aqui proposto, esta abordagem também introduz bits nas caches privadas e na cache compartilhada dos elementos de processamento.

O principal objetivo do trabalho de Prvulovic *et al.* [28] é a identificação dos gargalos de desempenho encontrados nas abordagens de suporte a TLS tradicionais. Como resultado desta análise foi observado que o tráfego adicional de coerência de cache gerado pela utilização de especulações, o custo proveniente das escritas em rajada na memória principal durante as operações de *commit* e o limite de armazenamento de dados especulativos são os principais fatores que limitam o desempenho dos mecanismos de TLS propostos na literatura. Note que, a abordagem aqui proposta não apresenta os dois primeiros fatores limitantes. Para o terceiro fator (capacidade de armazenamento de dados especulativos),

os autores propõem que uma área da memória principal seja utilizada para armazenar os dados especulativos que não couberem nas caches. Tal abordagem pode ser adaptada ao mecanismo aqui proposto em trabalhos futuros.

De maneira similar, o trabalho de Renau [33] é focado na identificação dos fatores limitantes presentes nas abordagens tradicionais de TLS relativas ao consumo de energia, já que o excesso de energia necessário por tais esquemas nunca permitiu que estes fossem adotados em escala industrial. Os principais fatores apontados foram as operações de *squash*, operações que precisam realizar verificações em conjuntos de linhas de cache e aumento de tráfego causado pelo uso de especulações. Os autores propõem então uma série de modificações no sistema operacional, *hardware* e compilador com o intuito de encontrar um compromisso razoável entre desempenho e consumo de energia, já que soluções para diminuir o consumo de energia geralmente afetam o desempenho negativamente. Por outro lado, nossa abordagem não introduz tráfego adicional e não requer operações em grupos de linhas de cache, considerando que as operações de *commit* possam limpar o mesmo bit em todas as *tags* simultaneamente. Em relação às operações de *squash*, nosso mecanismo permite que a operação seja realizada sem nenhum custo ou geração de tráfego adicional: o código sequencial pode começar a ser executado imediatamente, desconsiderando as linhas especulativas presentes na cache. Embora o consumo de energia não tenha sido medido, isso fornece evidências de que o mecanismo aqui proposto possui uma boa eficiência neste aspecto. Já o desperdício de trabalho realizado até o momento em que uma violação de memória é detectada é um problema que permanece em aberto e é inerente a todos os suportes para execução especulativa propostos até então.

Um outro tipo de abordagem para o problema de aproveitar a capacidade de processamento provida por máquinas com múltiplos núcleos foi proposta por Devietti *et al.* [13]. Ao invés do uso de especulação de dependências, os autores propõem um mecanismo para a execução determinística de aplicações paralelas. Através da relaxação do modelo de consistência de memória e de alterações em *hardware* e *software*, foram obtidas melhoras na escalabilidade de desempenho de programas paralelos, mesmo na presença de *races*. Embora tal modelo não seja focado na paralelização de aplicações sequenciais já existentes, esta abordagem mostra-se um importante auxílio para programadores de aplicações paralelas durante os processos de depuração, testes e replicação de código. Outra abordagem que pode ser útil ao programador for proposta por Zhang *et al.* [44], onde é apresentado um arcabouço capaz de identificar dinamicamente candidatos a paralelização. Tais candidatos não são, no entanto, paralelizados automaticamente: o sistema simplesmente informa o programador a respeito de tais possibilidades.

O trabalho de Bridges *et al.* [7] propõe a utilização de anotações em código por parte do programador a fim de facilitar o uso de técnicas de paralelização e especulação por parte do compilador, fazendo assim com que códigos paralelos mais eficientes possam ser gera-

dos automaticamente a partir de códigos sequenciais. Depois de mostrar a importância do uso de especulação para extração de desempenho escalável, os autores sugerem duas anotações. A primeira indica para o compilador se o código protegido por uma estrutura condicional pode ser executado independentemente do resultado da condição, além de informar uma probabilidade de que o desvio seja tomado. Pode-se notar assim a importância das informações de *profiling* para que o compilador quebre apenas dependências com probabilidades muito baixas de ocorrerem, a fim de evitar que falhas de especulação ocorram durante a execução. A segunda anotação indica para o compilador as funções que podem ser invocadas, durante a execução paralela, em uma ordem diferente daquela utilizada na execução sequencial. Assim, os autores evidenciam a importância do programador no processo de extração de paralelismo e consequente aproveitamento dos recursos providos por máquinas com múltiplos elementos de processamento.

Sistemas de suporte a técnicas tradicionais de TLS em *software*, bem como o emprego de memórias transacionais em *software* também foram propostos [24, 34, 40] com o intuito de aceitar a execução especulativa de códigos paralelizados por técnicas tradicionais (DOALL e DOACROSS).

Por fim, Raman *et al.* [29] apresentam o SMTX, um sistema implementado em *software* com o objetivo de permitir o modelo de execução especulativa em *pipeline*. Tal objetivo é atingido através da introdução de uma unidade de *commits* centralizada que é responsável por detectar conflitos entre os acessos à memória realizados por todos os *cores*. Isso é feito fazendo com que essa unidade centralizada re-execute todas as operações de leitura e escrita realizadas por todos os elementos de processamento. Conforme mostrado em [29], tal unidade representa um gargalo de desempenho para uma gama de aplicações para as quais as re-execuções dominam o tempo total de execução ou saturam o sistema de memória. Esta saturação pode ser agravada ainda pelas comunicações necessárias para que os *cores* informem a unidade de *commit* a respeito de quais operações de memória foram realizadas para que tal unidade possa replicá-los. Por outro lado, por causa da iteração com o protocolo de coerência de cache *já existente*, o esquema proposto nesta dissertação não depende de nenhuma entidade centralizada (já conhecidas por representarem pontos de contenção nos sistemas em que são utilizadas), como é necessário para o SMTX. As vantagens dessa forma "distribuída" através da qual as verificações de violações são feitas, podem ser muito substanciais. Em particular, a diferença de desempenho é dramática para os *benchmarks* `164.gzip, gimp-oilify` e `gimp-nova` (figura 4.2). Finalmente, mesmo no caso das aplicações para as quais a unidade centralizada de *commit* do SMTX não representa um gargalo de desempenho, nosso esquema ainda atinge desempenhos superiores.

Assim, pode-se notar que, embora vários mecanismos já tenham sido propostos para permitir a execução especulativa de aplicações, o único capaz de permitir o modelo em *pipeline* (que é o único modelo que provê desempenho escalável para aplicações de propósito

geral) apresenta um gargalo de desempenho devido à sua natureza centralizada. Os demais mecanismos permitem apenas a execução de aplicações paralelizadas por técnicas tradicionais (DOALL e DOACROSS) combinadas com especulação, além de apresentarem limitações como introdução de novos protocolos de coerência de cache, geração de tráfego em rajada, introdução de unidades centralizadas em *hardware*, entre outras.

Capítulo 6

Conclusões

Neste trabalho foi apresentada uma proposta de adição de *tags* às linhas de cache com o objetivo de habilitar, de maneira eficiente, a execução de aplicações paralelizadas não apenas por meio de técnicas tradicionais de TLS como DOALL e DOACROSS, mas também através de métodos baseados no modelo de execução em *pipeline* (DOPIPE) combinado com especulação e replicação de estágios.

A incorporação de *tags* às linhas de cache já é realizada pela IBM em seu processador *BlueGene/Q*, a fim de manter múltiplas versões de dados nas caches. Assim, visto de um *hardware*, com as mesmas características necessárias para a implementação do mecanismo aqui proposto, já ser fabricado pela indústria, faz com que a nossa abordagem se torne uma técnica promissora de ser implementada na prática.

O mecanismo aqui apresentado não modifica o protocolo de coerência de cache já existente e resolve a maioria dos problemas apresentados pelos esquemas tradicionais de suporte a TLS. O modelo proposto atinge um ganho de desempenho médio de 21.6× em uma máquina simulada com 24 elementos de processamento.

6.1 Trabalhos Futuros

Embora o mecanismo apresentado nesta dissertação apresente fortes evidências de ser eficiente em termos de consumo de energia, análises detalhadas e possíveis ajustes podem ser necessários, e são considerados trabalhos futuros. Embora a limitação de espaço de armazenamento dos dados especulativos não tenha sido um problema para os *benchmarks* utilizados em nossos experimentos, pode ser que algumas aplicações não possam ser executadas no sistema por essa razão. Uma possível abordagem para este problema é permitir que parte dos dados especulativos possam ser mantidos na memória principal. Essa possibilidade também deve ser avaliada futuramente.

Uma vez que o mecanismo proposto foi projetado para uma única máquina com

múltiplos núcleos, esta pode servir como peça para a formação de *clusters* de processamento. Nesse caso, os desafios são manter a coerência de dados e realizar a seleção de versões entre dados presentes nas caches de máquinas distintas. Como as técnicas de paralelização baseadas em DOPIPE combinadas com especulação promovem um ganho de desempenho escalável, a formação de *clusters* pode melhorar muito o desempenho das aplicações paralelas, uma vez que o número de *cores* pode crescer bastante. Por fim, como o simulador utilizado já provê suporte para redes de comunicação entre máquinas, a formação de *clusters* deve ser abordada no futuro.

Referências Bibliográficas

[1] H. Akkary and M. A. Driscoll. A Dynamic Multithreading Processor. In *MICRO 31: Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 226–236, 1998.

[2] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures: A dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

[3] A.M.D. *AMD64 architecture programmer's manual volume 2: System programming*. A.M.D., 2010.

[4] R. X. Arroyo, R. J. Harrington, S. P. Hartman, and T. Nguyen. IBM POWER7 systems. *IBM Journal of Research and Development*, 55(3), 2011.

[5] S. Balakrishnan and G. S. Sohi. Program Demultiplexing: Data-flow based Speculative Parallelization of Methods in Sequential Programs. *SIGARCH Computer Architecture News*, 34(2):302–313, 2006.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: characterization and architectural implications. In *PACT '08: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, 2008.

[7] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August. Revisiting the sequential programming model for multi-core. In *MICRO '07: Proceedings of the 40th annual IEEE/ACM International Symposium on Microarchitecture*, pages 69–84, 2007.

[8] M. J. Bridges. The velocity compiler: Extracting efficient multicore execution from legacy sequential codes, 2008. PhD thesis, Department of Computer Science, Princeton University, Princeton, New Jersey, United States.

[9] D.-K. Chen. Compiler optimizations for parallel loops with fine-grained synchronization, 1994. PhD thesis, Department of Computer Science, University of Illinois, Urbana, IL, United States.

[10] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 13–24, 2000.

[11] D. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., 1998.

[12] L. Dagum and R. Menon. Open MP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[13] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman. RCDC: A Relaxed Consistency Deterministic Computer. In *ASPLOS '11: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[14] GNU Image Manipulation Program. http://www.gimp.org.

[15] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative versioning cache. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 195–205, 1998.

[16] R. A. Haring, M. Ohmacht, T. W. Fox, M. K. Gschwind, D. L. Satterfield, K. Sugavanam, P. W. Coteus, P. Heidelberger, M. A. Blumrich, R. W. Wisniewski, A. Gara, G. L. Chiu, P. A. Boyle, N. H. Christ, and C. Kim. The IBM Blue Gene/Q compute chip. *Micro*, 32(2):48–60, 2012.

[17] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

[18] V. Krishnan and J. Torrellas. Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip-Multiprocessor. In *ICS '98: Proceedings of the 12th International Conference on Supercomputing*, pages 85–92, 1998.

[19] V. Krishnan and J. Torrellas. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Transactions on Computers*, 48(9):866–880, 1999.

[20] C. Lattner and V. Adve. A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, page 75, 2004.

[21] B. Lucia, L. Ceze, and K. Strauss. ColorSafe: Architectural Support for Debugging and Dynamically Avoiding Multi-variable Atomicity Violations. In *ISCA '10: Proceedings of The 37th International Symposium on Computer Architecture*, 2010.

[22] C. Madriles, P. López, J. M. Codina, E. Gibert, F. Latorre, A. Martinez, R. Martinez, and A. Gonzalez. Boosting Single-thread Performance in Multi-core Systems Through Fine-grain Multi-threading. *SIGARCH Computer Architecture News*, 37(3):474–483, 2009.

[23] P. Marcuello and A. González. Clustered Speculative Multithreaded Processors. In *ICS '99: Proceedings of the 13th International Conference on Supercomputing*, pages 365–372, 1999.

[24] M. Mehrara, J. Hao, P.C. Hsu, and S. Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 166–176, 2009.

[25] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 1965.

[26] A.N. Moudgal and B.M. Kuttanna. Apparatus and method to prevent overwriting of modified cache entries prior to write back, 2001. US Patent 6286082.

[27] G. Ottoni, R. Rangan, A. Stoler, and D. August. Automatic thread extraction with decoupled software pipelining. In *MICRO '05: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 105–118, 2005.

[28] M. Prvulovic and M. J. Garzarán. Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization. In *ISCA '01: In proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 204–215, 2001.

[29] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August. Speculative Parallelization Using Software Multi-threaded Transactions. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 65–76, 2010.

[30] E. Raman, G. Ottoni, A. Raman, M. Bridges, and D. August. Parallel-Stage Decoupled Software Pipelining. In *CGO '08: Proceedings of the 2008 International Symposium on Code Generation and Optimization*, 2008.

[31] L. Rauchwerger and D. A. Padua. The LRPD test: Speculative runtime parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):160–180, 1999.

[32] J. Reinders. *Intel threading building blocks*. O'Reilly, 2007.

[33] J. Renau, K. Strauss, L. Ceze, W. Liu, S.R. Sarangi, J. Tuck, and J. Torrellas. Energy-efficient thread-level speculation. *IEEE Micro*, 26(1):80–91, 2006.

[34] P. Rundberg and P. Stenstrom. An all-software thread-level data dependence speculation system for multiprocessors. *Journal of Instruction-Level Parallelism*, 3, 2001.

[35] P. Salverda and C. Zilles. Fundamental Performance Constraints in Horizontal Fusion of In-Order Cores. In *HPCA '08: The 14th IEEE International Symposium on High Performance Computer Architecture*, pages 252–263, 2008.

[36] SESC: Superescalar Simulator. http://iacoma.cs.uiuc.edu/∼paulsack/sescdoc.

[37] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar Processors. *SIGARCH Computer Architure News*, 23(2):414–425, 1995.

[38] Standard Performance Evaluation Corporation (SPEC). http://www.spec.org.

[39] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. *ACM SIGARCH Computer Architecture News*, 28(2):1–12, 2000.

[40] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *MICRO '07: Proceedings of the 40th annual IEEE/ACM International Symposium on Microarchitecture*, pages 356–369, 2007.

[41] N. Vachharajani. Intelligent speculation for pipelined multithreading, 2008. PhD thesis, Department of Computer Science, Princeton University, Princeton, New Jersey, United States.

[42] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. August. Speculative Decoupled Software Pipelining. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques*, 2007.

[43] Y. Watanabe, J. D. Davis, and D. A. Wood. WiDGET: Wisconsin Decoupled Grid Execution Tiles. *SIGARCH Computer Architecture News*, 38(3):2–13, 2010.

[44] X. Zhang, A. Navabi, and S. Jagannathan. Alchemist: A Transparent Dependence Distance Profiling Infrastructure. In *CGO '09: Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 47–58, 2009.

[45] H. Zhong, S. A. Lieberman, and S. A. Mahlke. Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-thread Applications. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 25–36, 2007.

Apêndice A

Prova de Corretude do Mecanismo Arquitetural

Conforme observado por Cintra *et. al* [10], violações de dependências *write-after-read* (WAR) não ocorrem em sistemas com cache multi-versionadas. Portanto, é necessário provar apenas que as dependências de memória *read-after-write* (RAW) e *write-after-write* (WAW) ou são satisfeitas corretamente pelo nosso mecanismo, ou causam uma falha de especulação que será detectada pelo sistema.

A.1 Convenções

Para as provas de corretude abaixo, as seguintes convenções serão adotadas:

- n é o número de elementos de processamento presentes no sistema.
- Se existem n *cores*, $(1..n)$, um intervalo do tipo $[x, y)$, onde $x > y$, é o mesmo que $[x, n] \cup [1, y)$. Tal convenção se faz necessária por causa da numeração circular dos bits nas *tags* das linhas de cache.
- Será considerado que o gerenciamento das caches é feito por linhas.
- Será considerado que o protocolo de coerência de cache MOSI [26] é utilizado.

A.2 Lemas

Lema 1 *A primeira operação de escrita realizada pelo* core *c durante a execução da iteração i gera uma transação de barramento.*

Prova: Quando a iteração i começa a ser executada no sistema, o bit ($i \bmod n$) da DepsTag é desativado para todas as linhas X nas caches de todos os *cores*. Isso acontece porque a iteração ($(i-n) \bmod n$) acabou de sofrer uma operação de *commit*. Mas agora, quando o *core c* escreve na linha de cache X pela primeira vez na iteração i, o bit ($i \bmod n$) ainda está desativado, já que este estaria ativado apenas em duas situações:

- se o *core c* já tivesse realizado uma leitura da linha X. Mas neste caso, esta linha de cache estaria ou no estado S ou no estado O do protocolo de coerência de cache. Como esta linha precisa ir para o estado M quando uma escrita for realizada nela, a primeira escrita irá gerar uma transação de barramento pelo funcionamento natural do protocolo de coerência de cache.
- se este *core* já tivesse realizado alguma escrita na linha X durante a execução de uma iteração y tal que $i - n < y < i$ e $y \bmod n = i \bmod n$. Obviamente, tal y não existe.

Como uma transação de barramento não seria gerada apenas se o bit ($i \bmod n$) da DepsTag estivesse ativado na linha de cache X (isso causaria um *hit* na cache), pode-se concluir que uma transação de barramento será sempre gerada.

A.3 Dependências *Read After Write* (RAW)

Esta prova está dividida em duas partes. Primeiramente será provado que quando uma violação de dependência RAW ocorre, o mecanismo realiza a detecção corretamente, causando assim uma operação de *squash*. Isso é necessário para garantir a corretude do sistema. Em seguida será provado que se uma dependência de memória RAW é corretamente satisfeita, nenhum *squash* acontecerá. Em outras palavras, será provado que operações de *squash* somente serão realizadas quando uma violação de dependência realmente acontece. Embora não seja necessária para garantir a corretude do sistema, esta prova garante que o desempenho não é afetado por *squashes* desnecessários.

A.3.1 Uma dependência RAW é violada

Se o *core c* realiza uma operação de leitura da linha X durante a execução da iteração i, mas o valor correto da linha de cache X para esta iteração (aquele que deveria ser lido por essa iteração se o código estivesse sendo executado sequencialmente) é escrito em um momento posterior em algum lugar do sistema, será mostrado que a iteração i não sofrerá *commit*. Desta forma, é necessário provar duas afirmações:

1. Quando o valor correto (ou mais apropriado) da linha de cache X para a leitura na iteração i for escrita em uma iteração j, o bit ($j \bmod n$) está ativado na DepsTag da linha de cache $\langle X, i \bmod n\rangle$ do *core* c, e como $j < i$, a iteração i sofrerá *squash*.

2. Quando este valor correto (ou mais apropriado) da linha X é escrito por um *core*, este *core* gera uma transação de barramento, de forma que c possa detectar a falha de especulação.

Prova 1: Primeiramente, note que o valor lido não foi produzido durante a execução de uma iteração y tal que $y \bmod n = i \bmod n$. Tal premissa vale porque como existem apenas n iterações sendo executadas em qualquer instante de tempo, tem-se que $y = i$. Mas o valor lido não pode ter sido produzido na iteração i porque este seria o valor correto para essa operação de leitura, contradizendo a hipótese de que o valor correto ainda não foi escrito.

Agora será mostrado que essa operação de leitura gera uma transação de barramento porque a leitura não será atendida por um *hit* na cache. Tal prova é importante porque garante que o *core* c irá ativar os bits da DepsTag da linha de cache X para fins de detecção de violações de memória.

Tal transação não aconteceria apenas se a linha de cache $\langle X, i \bmod n\rangle$ do *core* c estivesse em um dos estados M, O ou S do protocolo de coerência de cache, **e** se o bit ($i \bmod n$) da DepsTag da mesma linha de cache em c estivesse ativado. No entanto, quando a iteração $(i - n)$ sofreu a operação de *commit* (isso já aconteceu, já que a iteração i está sendo executada), este bit foi desativado. Assim, este bit estaria ativado apenas se o *core* c realizou alguma operação de escrita ou leitura na linha X durante a execução de alguma iteração y tal que $i - n < y \leq i$ e $y \bmod n = i \bmod n$. Mas como $y \bmod n \neq i \bmod n$, para todo y no intervalo $(i - n, i)$, conclui-se que $y = i$. Mas nós já sabemos que o *core* c não escreveu na linha X durante a iteração i antes dessa leitura ser realizada. Portanto, a única possibilidade restante é que o *core* c já havia lido X durante a execução da iteração i; esta possibilidade não existe para nenhuma operação de leitura realizada em i, exceto para a primeira leitura. Logo, a primeira operação de leitura feita pelo *core* c durante a execução da iteração i precisa ter gerado uma transação de barramento; e isso é tudo que precisa ser provado, já que esta leitura ativou todos os bits necessários na DepsTag da linha de cache $\langle X, i \bmod n\rangle$, e todas as leituras da linha X durante a iteração i executadas pelo *core* c antes de qualquer escrita realizada pelo mesmo *core* na mesma linha de cache durante a execução da iteração i devem obter o mesmo valor que foi obtido pela primeira leitura. Portanto, os valores obtidos por tais leituras subsequentes estão corretos se a primeira leitura obteve o valor correto, e estão errados caso contrário.

Agora que o valor mais apropriado (mas incorreto!) da linha de cache X foi lido pelo *core* c durante a execução da iteração i, suponha que o valor lido tenha sido escrito em

algum lugar no sistema durante a execução da iteração k (que é a próxima iteração a sofrer *commit* se o valor lido não for especulativo). Note que os bits da DepsTag no intervalo $[k \bmod n, i \bmod n)$ da linha de cache $\langle X, i \bmod n\rangle$ do *core* c não estariam ativados somente se $k \bmod n = i \bmod n$. Mas como k é pelo menos a próxima iteração a sofrer *commit* e a iteração $i - n$ já sofreu *commit*, então $k > i - n$. Por outro lado, como a iteração $i + n$ ainda não foi iniciada, $k < i + n$. Mas nós acabamos de provar que $k \neq i$. Logo, $k \bmod n \neq i \bmod n$, e os bits citados estão ativados.

Agora a linha de cache $\langle X, i \bmod n\rangle$ do *core* c tem os bits da sua DepsTag ativados no intervalo $[k \bmod n, i \bmod n)$. Portanto, a iteração i sofrerá *squash* por causa desta leitura apenas quando uma escrita na linha de cache X aparecer no barramento, e tal escrita tiver sido realizada durante a execução de uma iteração w tal que $w \bmod n$ esteja no intervalo dos bits que estão ativados na DepsTag.

Suponha que quando a linha X é escrita na iteração j, o bit $(j \bmod n)$ seja desativado na DepsTag da linha de cache $\langle X, i \bmod n\rangle$ do *core* c. Isso significa que em algum ponto entre a leitura realizada na iteração i e essa escrita na iteração j, alguma iteração y tal que $y \bmod n = j \bmod n$ sofreu *commit*. Vamos supor, para efeitos de contradição, que tal y existe.

Obviamente $y < j$, pois a iteração j está sendo executada agora. Além disso, quando a operação de leitura foi executada na iteração i, a iteração $i - n$ já havia sofrido *commit*, o que nos dá que $y > i - n$. Mas $i > j \Rightarrow i - n > j - n$, e como $y > i - n$, nós temos que $y > j - n$. No entanto, não existe um y tal que $j - n < y < j$ e $y \bmod n = j \bmod n$, e portanto a prova 1 está completa.

Prova 2: Vamos supor que o valor incorreto foi produzido pelo *core* q, durante a execução da iteração k, e que o valor correto (ou um valor mais apropriado) vai ser escrito pelo *core* p durante a iteração j. Agora nós temos duas possibilidades:

- O valor correto (ou mais apropriado) vai ser escrito em um estágio que será iniciado após o termino do estágio que realizou a leitura.
- O valor correto (ou mais apropriado) vai ser escrito em um estágio que será executado concorrentemente com o estágio que realizou a leitura.

O primeiro caso é satisfeito de forma trivial pelo Lema 1. O segundo caso pode ser subdividido em dois sub-casos:

1. O valor incorreto lido durante a iteração i foi escrito em algum estágio cuja execução já foi finalizada.
2. O valor incorreto lido durante a iteração i foi escrito em algum estágio que ainda encontra-se em execução.

Novamente, o primeiro sub-caso é satisfeito trivialmente pelo Lema 1. Para o segundo sub-caso, note que $q \neq c$, pois caso contrário o valor correto já teria sido escrito, contrariando a hipótese de que o valor correto ainda não foi escrito. Agora, se $k = j$, então $p = q$ e a linha de cache $\langle X, k \bmod n$ deste *core* foi para o estado O do protocolo de coerência no momento em que este *core* enviou o valor incorreto para a iteração i. Quando este *core* escrever na linha X durante a iteração k novamente, será gerada uma transação de barramento, pois esta linha de cache (agora no estado O do protocolo) precisa ir para o estado M antes que a escrita seja realizada. Finalmente, se $k \neq j$, então $p \neq q$. Mas note que quando o *core* c solicitou a linha de cache X, se a linha produzida na iteração k foi enviada é porque a linha X não havia sido escrito durante a iteração j ainda (caso contrário este último valor seria mais apropriado). Logo, quando a linha X é escrita na iteração j, o Lema 1 garante que uma transação de barramento será produzida.

A.4 Uma dependência RAW é satisfeita corretamente

Neste caso o *core* c realiza uma operação de leitura da linha de cache X durante a execução da iteração i, e o valor correto já foi escrito. A prova é dividida nos seguintes três casos exclusivos:

- Se o *core* c já havia escrito na linha X durante a iteração i (então este é o valor correto).
- Senão, se um outro *core* já havia escrito na linha X durante a iteração i (então este é o valor correto).
- Senão, o valor correto foi escrito por algum *core* durante uma iteração k tal que $k < i$.

No primeiro caso, a linha de cache $\langle X, i \bmod n\rangle$ do *core* c possui o valor mais apropriado e este é o valor correto. Portanto, é necessário provar que:

1. A operação de leitura será satisfeita por um *hit* na cache (e portanto não haverá a geração de uma transação de barramento), e portanto o valor correto será lido.
2. Não haverá nenhum *squash* por causa desta leitura.

Prova 1: Quando a linha de cache $\langle X, i \bmod n\rangle$ do *core* c foi escrita por c durante a execução da iteração i, esta linha foi para o estado M do protocolo de coerência de cache, e depois disso, ela teria mudado para o estado I somente se um outro *core* p tal que $p \neq c$ tivesse escrito na linha X durante a execução de uma iteração y tal que ($y \bmod n =$

$i \bmod n$). Mas como apenas o *core* c está executando a iteração i neste momento e não há nenhum outro *core* executando uma iteração y tal que $y \bmod n = i \bmod n$, tal hipótese não é possível. Portanto a linha de cache $\langle X, i \bmod n \rangle$ do *core* c não está no estado I do protocolo de coerência. Se essa linha de cache não foi para o estado I, ela não pode ter ido para o estado S também, já que isso só aconteceria se um outro *core* r tivesse escrito na linha X e depois disso o *core* c realizasse uma leitura dessa mesma linha. Mas nesse caso a linha $\langle X, i \bmod n \rangle$ de c teria ido para o estado I no momento da escrita realizada por r, o que acabamos de mostrar que não acontece.

Prova 2: Não ocorrerá nenhum *squash* devido a essa operação de leitura, já que nenhum bit será ativado na DepsTag da linha de cache $\langle X, i \bmod n \rangle$ do *core* c (a linha X foi produzida na mesma iteração).

Para o segundo caso, o último *core* que escreveu na linha de cache X durante a execução da iteração i possui o valor mais apropriado, e este precisa ser o valor correto. Portanto é necessário provar que:

1. A primeira operação de leitura irá solicitar o valor mais apropriado no barramento (e leituras subsequentes irão utilizar o mesmo valor através de *hits* na cache).

2. O *core* que escreveu na linha X durante a iteração i pela última vez é o único que possui a linha de cache $\langle X, i \bmod n \rangle$ nos estados M ou O do protocolo de coerência de cache, e portanto esta é a linha que será enviada para o *core* c.

3. Não ocorrerá nenhum *squash* por causa desta operação de leitura.

Prova 1: Não ocorrerá um *hit* na cache porque quando a iteração $i - n$ sofreu *commit*, o bit $i \bmod n$ da DepsTag da linha de cache $\langle X, i \bmod n \rangle$ do *core* c foi desativado, e como c não escreveu na linha X durante a execução da iteração i, este bit ainda estará desativado quando esta leitura for executada por c durante a iteração i (este bit estaria ativado somente se em alguma iteração y tal que $i - n < y < i$ e $y \bmod n = i \bmod n$ foi realizada alguma leitura ou escrita na linha X antes que esta fosse lida pelo *core* c durante a execução de i. Mas como tal y não existe, o bit está desativado). Portanto, haverá uma transação de barramento solicitando a linha de cache X.

Prova 2: O *core* que escreveu na linha X durante a execução da iteração i pela última vez teria sua linha de cache $\langle X, i \bmod n \rangle$ no estado I do protocolo de coerência somente se um outro *core* escreveu em X durante uma iteração y tal que $y \bmod n = i \bmod n$. No entanto, como existem apenas n iterações sendo executadas a qualquer momento, e como i está sendo executada, pode-se concluir que $y = i$. Mas obviamente nenhum outro *core* poderia ter escrito na linha X durante a iteração i depois do *core* que escreveu na linha X durante a iteração i pela última vez. Logo, o *core* que escreveu na linha de cache X durante a iteração i pela última vez não pode ter a sua linha de cache $\langle X, i \bmod n \rangle$ no

estado I do protocolo de coerência. Como esta linha de cache não mudou para o estado I, ela também não pode ter ido para o estado S. Portanto, este *core* irá enviar o último valor de que foi escrito na linha de cache X.

Prova 3: Não haverá *squash* por causa desta leitura, já que nenhum bit da DepsTag da linha de cache $\langle X, i \bmod n\rangle$ do *core* c estará ativado (a linha X que foi lida foi produzida na mesma iteração).

Para o último caso é preciso provar:

1. A primeira operação de leitura irá solicitar o dado mais apropriado no barramento (leituras subsequentes utilizarão o mesmo valor através de *hits* na cache).

2. O *core* que escreveu na linha X durante a execução da iteração k pela última vez tem a sua linha de cache $\langle X, k \bmod n\rangle$ nos estados M ou O do protocolo de coerência de cache, e portanto esta linha será enviada para o *core* c.

3. Não ocorrerá nenhum *squash* por causa desta operação de leitura.

Prova 1: Não ocorrerá um *hit* na cache pelo mesmo motivo evidenciado na Prova 1 do caso acima. Portanto, o *core* c irá gerar uma transação de barramento solicitando a linha de cache X, e como o valor correto para essa leitura já foi escrito, este será o valor enviado.

Prova 2: Para provar esta afirmação, suponha que o valor correto foi escrito durante a execução da iteração k pelo *core* p. A linha de cache $\langle X, k \bmod n\rangle$ deste *core* estaria no estado I do protocolo de coerência somente um outro *core* q escreveu na linha X durante a execução de alguma iteração y tal que $y \bmod n = k \bmod n$. Mas note que y é obviamente uma das n iterações que ainda não sofreram *commit* (existem apenas n iterações que ainda não sofreram *commit* em qualquer instante de tempo), assim como a iteração k (k é, pelo menos, a próxima iteração a sofrer *commit* se o valor correto da linha de cache X vier da memória principal). Logo temos que $y = k$. Repare então que se $p = q$, a linha de cache $\langle X, k \bmod n\rangle$ do *core* p está ou no estado M ou no estado O do protocolo de coerência. Por outro lado, $p \neq q$ é impossível porque isso contradiz a hipótese de que o valor correto foi escrito durante a execução da iteração k pelo *core* p.

Prova 3: No momento em que o *core* c receber o valor (correto) de X, a DepsTag da sua linha de cache $\langle X, i \bmod n\rangle$ terá os bits ativados no intervalo $[k \bmod n, i \bmod n)$. Portanto, irá acontecer um *squash* por causa desta operação de leitura somente se houver uma escrita na linha de cache X durante a execução de uma iteração y tal que $y \bmod n \in [k \bmod n, i \bmod n)$ antes que a iteração i sofra *commit*. Suponha, para fins de contradição, que tal y existe.

Note que $y > i - n$ porque os bits (no intervalo $[k \bmod n, i \bmod n)$) da DepsTag estão ativados quando a linha X é lida durante a execução da iteração i, e neste instante, a

iteração $(i - n)$ já sofreu *commit*. Além disso, o valor foi produzido durante a iteração k, tal que $k > i - n$ (mesmo que o valor tenha vindo da memória principal, k seria pelo menos $i - n + 1$). Mas se $y > i - n$, $y \bmod n \in [k \bmod n, i \bmod n)$ e $k \leq i$, então temos que $y \geq k$. Ainda, o leitor deve perceber que quando a execução da iteração $i + n$ é iniciada (ou seja, é colocada no "*pipeline* de execução"), a iteração i já terá sofrido *commit*. Como a escrita durante a iteração y precisa ser realizada antes que a iteração i sofra *commit*, então $y < i + n$. Portanto, como nós temos ambos $k \leq y < i + n$ e $y \bmod n \in [k \bmod n, i \bmod n)$, existem duas possibilidades:

- $k \leq y < i$
- $k + n \leq y < i + n$

No primeiro caso, o valor escrito na linha de cache X durante a execução da iteração y é mais apropriado para a operação de leitura realizada na iteração i. Mas como esta escrita ocorre depois que a linha X é solicitada na iteração i, ocorre uma contradição com a hipótese de que o valor correto havia sido escrito antes do *core* c solicitar o valor durante a execução da iteração i.

Para o segundo caso, quando a operação de escrita é realizada durante a iteração y, a iteração $y - n$ já sofreu *commit*. Logo, o bit $y \bmod n$ da DepsTag estará representando a iteração y. Mas, conforme foi mostrado acima, $k > i - n$, e nós temos que $k + n > i$. Como $y \geq k + n > i$, não ocorrerá nenhum *squash* por causa desta escrita.

A.5 Dependências *Write After Write* (WAW)

Repare que, de acordo com o funcionamento natural do protocolo de coerência de cache, quando um *core* realiza uma operação de escrita em alguma linha de cache, este também lê essa mesma linha. Portanto, a prova para dependências *write after write* é exatamente a mesma prova para as dependências *read after write*.