COPPE/UFRJ

EXECUÇÃO ESPECULATIVA EM UMA MÁQUINA VIRTUAL *DATAFLOW*.

Tiago Assumpção de Oliveira Alves

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Felipe Maia Galvão França

Rio de Janeiro
Maio de 2010

EXECUÇÃO ESPECULATIVA EM UMA MÁQUINA VIRTUAL *DATAFLOW*.

Tiago Assumpção de Oliveira Alves

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof. Felipe Maia Galvão França, Ph.D.

Prof. Valmir Carneiro Barbosa, Ph.D.

Prof. Eugene Francis Vinod Rebello, Ph.D.

RIO DE JANEIRO, RJ – BRASIL
MAIO DE 2010

Alves, Tiago Assumpção de Oliveira

Execução Especulativa em uma Máquina Virtual *Dataflow.*/Tiago Assumpção de Oliveira Alves. – Rio de Janeiro: UFRJ/COPPE, 2010.

X, 62 p.: il.; 29, 7cm.

Orientador: Felipe Maia Galvão França

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2010.

Referências Bibliográficas: p. 60 – 62.

1. dataflow. 2. TLP. 3. multicore. I. França, Felipe Maia Galvão. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Dedico este trabalho aos meus pais e à minha irmã, sem cujo apoio total e irrestrito nada teria sido possível.

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

EXECUÇÃO ESPECULATIVA EM UMA MÁQUINA VIRTUAL *DATAFLOW*.

Tiago Assumpção de Oliveira Alves

Maio/2010

Orientador: Felipe Maia Galvão França

Programa: Engenharia de Sistemas e Computação

A programação paralela se tornou mandatória para explorar totalmente o potencial das CPUs modernas. Frequentemente o paralelismo é limitado pelas dependências entre instruções ou blocos de instruções. Para contornar esse problema, em modelos de execução otimistas ou especulativos, é permitido que *threads* prossigam mesmo que possa haver dependências. Neste caso, o último estado consistente da máquina deve ser recuperado e as computações conflituosas devem ser reexecutadas.

Acreditamos que modelos de execução especulativa se adequem naturalmente ao estilo de execução *dataflow*. No entanto, um problema do modelo *dataflow* é que ou as tarefas são muito pequenas, de granularidade fina, e o sistema paralelo tem seus canais de comunicação inundados; ou as tarefas são de granularidade muito grossa, fazendo com que suas dependências sejam complexas e difíceis de serem expressadas, levando programadores a subestimarem o paralelismo potencial. Usando especulação podemos liberar o programador para considerar apenas as dependências principais. O preço a pagar é que, como parte da computação pode ser desfeita, suporte para a reexecução é necessário.

Para validar nossa hipótese, implementamos um sistema *dataflow* de granularidade grossa com suporte à especulação. Um estudo inicial em uma aplicação artificial que simula um sistema de contas bancárias, com cenários que variam na carga computacional, tamanho de transações, profundidade da especulação e contenção, sugere que há uma grande faixa de valores onde a especulação pode ser bastante efetiva. Em um segundo experimento, paralelizamos uma importante aplicação na área de mineração de dados. Obtivemos desempenho bom em ambos os casos, tendo comparado a segunda aplicação com duas versões *OpenMP* da mesma.

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

SPECULATIVE EXECUTION ON A DATAFLOW VIRTUAL MACHINE

Tiago Assumpção de Oliveira Alves

May/2010

Advisor: Felipe Maia Galvão França

Department: Systems Engineering and Computer Science

Parallel programming has become mandatory to fully exploit the potential of modern CPUs. Most often, parallelism is limited by dependencies between instructions or blocks of instructions. In order to address this problem, in optimistic or speculative models of execution, threads can go ahead even if they may possibly be dependent: state is restored and computation is undone if dependencies lead to conflict.

We argue that speculative models of execution fit naturally with the dataflow style of execution. Indeed, one major problem with dataflow execution was that either tasks were too small and fine-grained, and a parallel system could be flooded by huge numbers of tasks; or tasks were too coarse, and dependencies were complex and hard to express, driving programmers to under-estimate parallelism. Using speculative execution liberates the programmer to considerate only the main dependencies, and still allows correct dataflow execution of coarse-grained tasks. The price to pay is that computation may be undone, requiring support for re-execution.

To validate our point, we implemented a speculative coarse-grained dataflow system. An initial study on a bank server artificial application with scenarios that vary on computation load, transaction size, speculation depth and contention suggests that there is a wide range of values where speculation can be very effective. As a second experiment, we parallelized an important application in the area of data-mining. We obtain good speedups, significantly better than two OpenMP implementations.

Sumário

Lista de Figuras

Capítulo 1

Introdução

A programação paralela se torna cada vez mais necessária para explorar o potencial das CPUs modernas. À medida em que os limites do paralelismo em nível de instrução foram sendo atingidos, a indústria migrou para arquiteturas distribuídas, a exemplo das máquinas *multicore*.

Essa mudança nas arquiteturas requer também uma mudança nos modelos de programação, tendo em vista que programas otimizados para executarem sequencialmente não exploram bem o paralelismo em nível de *threads*, que deve ser o alvo da programação para essas novas arquiteturas. Por outro lado, desenvolver versões paralelas de programas é uma tarefa não-trivial, já que o programador deve considerar as dependências entre as computações de cada *thread*. Além disso, a partir das dependências identificadas, o programador deve desenvolver um esquema de sincronização entre as *threads* para garantir que os resultados produzidos reflitam os mesmos efeitos de uma versão sequencial do programa.

Tradicionalmente, as os métodos de sincronização adotados são baseados em troca de mensagens ou no uso de estruturas como *locks*, que garantem a exclusão mútua no acesso a objetos compartilhados. Em ambas as abordagens há uma troca entre complexidade e performance. Considere o caso do uso de *locks*. Uma implementação com *locks* de granularidade grossa é mais fácil de ser implementada que a equivalente com *locks* de granularidade fina, em razão da dificuldade de se determinar as possibilidades de ocorrência de *deadlocks* em programas com muitos *locks*. Por outro lado, a implementação com *locks* de granularidade grossa serializa parte do processamento que poderia ser executada em paralelo.

O modelo *Dataflow*, como o próprio nome diz, basea-se no fluxo de dados para sincronizar a execução de instruções. De maneira geral, uma instrução é executada assim que todos os seus operandos de entrada estiverem prontos. Se uma instrução A produz um valor necessário para a computação de uma outra instrução B, deve haver uma aresta de A para B no grafo *dataflow* que descreve o programa. Se há um caminho de A para C no grafo *dataflow*, C é dependente de algum valor

produzido por A e portanto só executará depois de A. Instruções entre as quais não existem caminhos no grafo são consideradas concorrentes e poderão ser executadas em paralelo. Dessa forma, a sincronização e a exploração do paralelismo são feitas naturalmente após as dependências terem sido expressas no grafo.

Tendo em mente essas características do modelo *Dataflow*, apresentamos o modelo de execução TALM (*TALM is an Architecture and Language for Multithreading*), que com base nas arquiteturas *Dataflow* explora paralelismo em nível de *threads*. No modelo além das instruções *Dataflow* simples podem ser implementadas instruções de granularidade grossa (chamadas de super-instruções). Dessa maneira, o conjunto de instruções do TALM é personalizado para cada programa, com a adição de super-instruções implementadas pelo programador de acordo com as tarefas a serem executadas pela aplicação.

No TALM, cada instrução deve ser mapeada para um elemento de processamento (EP). Como tanto as instruções simples quanto as super-instruções são executadas de acordo com a regra *dataflow*, trechos independentes do código serão executados de forma paralela, caso estejam contidos em instruções mapeadas em EPs diferentes. Assim, ao invés de implementar a sincronização entre *threads* através da troca de mensagens ou *locks*, o programador implementa as porções do código a serem paralelizadas em super-instruções e em seguida as conecta no grafo *dataflow* de acordo com suas dependências. Para operações como a redução de valores produzidos por super-instruções distintas, o programador pode usar também instruções *dataflow* simples, o que caracteriza o TALM como uma arquitetura *dataflow* híbrida.

A identificação das dependências entre as instruções, no entanto, nem sempre é uma tarefa trivial. Em diversos cenários, prever antes da execução os endereços dos acessos de leitura/escrita a recursos compartilhados pode ser muito difícil, ou até impossível. No caso dessas dependências, que chamamos de dependências dinâmicas por só poderem ser identificadas em tempo de execução, a solução adotada normalmente é serializar partes do código que podem ser dependentes. Em modelos *dataflow* isso é o equivalente a acrescentar uma aresta entre duas instruções quando, durante a descrição do grafo, não se possa prever se um valor escrito por uma deverá ser lido pela outra. Essa aresta deverá respeitar a ordem na qual as duas instruções seriam executadas no código sequencial, isto é, a aresta vai de A para B, caso a computação de A preceda a de B em uma versão sequencial do programa.

Esse tipo de solução conservadora muitas vezes acaba limitando as oportunidades para o paralelismo no programa. Assim como ocorre nas implementações com *locks* de granularidade grossa, a adição de arestas de dependência pode fazer com que diversas regiões do código que poderiam executar em paralelo sejam executadas de maneira sequencial, impedindo maiores ganhos de performance.

Uma possível abordagem para esse tipo de problema é o uso da especulação.

Nos casos em que as dependências entre instruções não possam ser determinadas de forma trivial, é adotada uma solução otimista: presume-se que não há dependências e as instruções são executadas concorrentemente de forma especulativa. Após a execução especulativa de instruções é acrescentada uma etapa responsável por verificar possíveis conflitos ocorridos em função de dependências dinâmicas. As escritas feitas por instruções especulativas devem poder ser revertidas (ou simplesmente descartadas) em caso de conflitos. É nessa etapa de verificação de conflito que as escritas especulativas são validadas e persistidas ou desfeitas/descartadas, caso algum conflito seja encontrado.

Neste trabalho propomos um modelo de execução *dataflow* especulativa baseado no conceito de memória transacional [1]. A principal característica do nosso modelo especulativo é que todo o controle da especulação é feito através da inclusão de arestas e instruções no grafo *dataflow*. Dessa forma, o controle é distribuído, sem depender de uma entidade centralizada, o que confere escalabilidade ao modelo.

Para avaliar o nosso modelo de execução, implementamos uma máquina virtual *multi-threaded*, a Trebuchet. A Trebuchet é uma implementação do modelo TALM com suporte à especulação desenvolvida para máquinas *multicore*. Nossos experimentos foram divididos em duas etapas. Primeiro paralelizamos aplicações na Trebuchet sem o uso da especulação e comparamos seu desempenho com versões *OpenMP* [2] das mesmas. Em seguida paralelizamos outras aplicações com a adoção da especulação, comparando com versões "oráculo" (simulação de cenários ideais) e com versões *OpenMP* com o uso de *locks*. A divisão dos experimentos nessas duas etapas se deve à necessidade de avaliar os custos e a performance do TALM básico e do modelo de execução especulativa de maneira separada. Os experimentos da primeira etapa, portanto, expuseram somente os custos do TALM e de sua implementação na Trebuchet, enquanto os da segunda etapa demonstraram a performance do modelo de execução especulativa que apresentamos.

O restante deste trabalho está dividido da seguinte forma: *(i)* o Capítulo 2 apresenta os trabalhos relacionados; *(ii)* o Capítulo 3 o modelo TALM; *(iii)* o Capítulo 4 descreve os detalhes da implementação Trebuchet; *(iv)* o Capítulo 5 apresenta o conceito de Ordem Parcial Condicional, utilizado para formalizar regras do modelo especulativo; *(v)* o Capítulo 6 descreve o modelo de execução *dataflow* especulativa; *(vi)* o Capítulo 7 apresenta os experimentos e resultados; *(vii)* o Capítulo 8 apresenta conclusões e trabalhos futuros.

Capítulo 2

Trabalhos relacionados

2.1 WaveScalar

Um significativo entrave para a adoção de máquinas *dataflow* como padrão foi a necessidade do uso de linguagens específicas para o desenvolvimento de aplicações para essas arquiteturas. A semântica das operações de memória de linguagens imperativas não era suportada e portanto eram necessárias linguagens derivadas de linguagens funcionais, como Lisp ou Id. A arquitetura *WaveScalar* [3] é a primeira arquitetura *dataflow* a apresentar um mecanismo para o suporte dessa semântica.

Através do uso de anotações, feitas em tempo de compilação nas instruções de acesso à memória, uma interface centralizada de leitura/escrita executa na ordem original as operações de memória. Dessa forma, mesmo que as instruções de memória sejam despachadas fora de ordem, seus acessos à memória respeitam a semântica do programa.

A *Transactional WaveCache* [4, 5] é um mecanismo de ordenação de memória alternativo para a arquitetura *WaveScalar*. O mecanismo mantém a ordem de execução das operações de memória dentro de uma iteração, mas permite que operações de memória de iterações distintas sejam executadas concorrentemente de forma especulativa. A *Transactional WaveCache* é baseada no conceito de memórias transacionais, sendo cada iteração tratada como uma região atômica. Se uma iteração termina todas as suas operações de memória, ela pode fazer o *commit*, desde que todas as iterações anteriores tenham feito *commit*.

2.2 Máquinas *Dataflow* Híbridas

Máquinas *dataflow* híbridas apresentam a execução de blocos de código *Von Neumann* escalonados seguinda a regra *dataflow* ou blocos de instruções *dataflow* escalonados com o uso de contador de programa. A arquitetura SDF [6] segue o primeiro

princípio, escalonando *threads* com base no fluxo de dados. Já a arquitetura TRIPS [7] usa o modelo inverso, a execução de instruções dentro dos blocos é *dataflow*, mas o escalonamento desses blocos segue o modelo de *Von Neumann*.

A BMDFM [8] é uma máquina virtual híbrida. Sua implementação permite a criação de blocos de granularidade grossa, bem como a descrição de programas em sua linguagem nativa, uma linguagem funcional derivada da Lisp. As diferenças fundamentais entre a nossa implementação, a *Trebuchet*, e a BMDFM estão nas suas linguagens nativas e nas suas arquiteturas. A arquitetura da BMDFM é baseada em processos do tipo *daemon* para os quais são escalonadas dinamicamente as instruções a serem executadas.

2.3 Especulação em nível de Thread

A exploração de paralelismo utilizando Especulação em nível de Thread (TLS) tem sido foco de um número considerável de trabalhos. Podemos dizer que a dificuldade na programação paralela com mecanismos de sincronização tradicional é um dos fatores fundamentais para isso. Em [9], Prabhu e Olukotun demonstram as oportunidades de ganho de performance com TLS através da paralelização manual de diversas aplicações do conjunto de *benchmarks* SPEC2000.

Em [10] é apresentada uma técnica de especulação de controle. É feita uma predição do número de vezes que o desvio condicional dos laços é tomado para reduzir o número de *threads* criadas especulativamente sem necessidade.

Em [11] e [12] são apresentados modelos baseados em memória transacional que preservam a ordem das operações de memória do código sequencial. De maneira semelhante ao nosso trabalho, nesses dois modelos a etapa de persistência das escritas na memória global compartilhada é separada da execução das *threads* e é feita em ordem. Dessa maneira, as *threads* do programa paralelizado podem executar fora de ordem, mas as suas escritas serão todas feitas respeitando a ordem original do programa.

A abordagem apresentada em [13], chamada de *Program Demultiplexing*, tem como motivação o fato de frequentemente os parâmetros de entrada de métodos (ou funções) de um programa estarem disponíveis muito antes do momento em que esses métodos são executados. Utilizando hardware com suporte a TLS e instrumentações no código original, métodos com essas características passam a ser disparados para execução especulativa assim que seus parâmetros de entrada ficam prontos. No momento em que realmente seriam executados, segundo o fluxo do programa sequencial, as escritas especulativas desses métodos são persistidas, caso não haja violações nos conjuntos de leitura. Havendo violações os métodos devem ser reexecutados de maneira não-especulativa.

Em [14] é apresentado um modelo que, assim como o implementado para os nossos experimentos, é inteiramente baseado em software. Através de ferramentas de *profiling*, são identificados os laços do programa cujos acessos à memória seguem padrões de endereçamento que podem ser previstos. Além disso, para que iterações desses laços sejam paralelizadas, esses padrões de endereçamento não devem apresentar conflitos entre leituras e escritas de diferentes iterações. Durante a execução, caso seja feito um acesso cujo endereço difere do previsto, a *thread* correspondente é interrompida e deve ser reexecutada de maneira não-especulativa.

Capítulo 3

TALM

Predominantemente, as arquiteturas de processadores comerciais, tanto monoprocessadas quanto multiprocessadas, seguem o modelo de Von Neumann, tendo sua execução guiada pelo fluxo de controle. A exemplo do algoritmo de Tomasulo [15], técnicas de execução fora de ordem, baseadas no fluxo de dados, vêm sendo adotadas como forma de explorar ILP em máquinas superescalares. Essas técnicas, no entanto, permanecem limitadas pelo despacho das instruções, que continua sendo feito em ordem.

Neste trabalho, buscamos tirar proveito da oportunidade de exploração de paralelismo do modelo *dataflow*. Para isso, desenvolvemos um modelo de execução baseado em arquiteturas *dataflow*: o TALM (*TALM is an Architecture and Language for Multithreading*). Conforme será visto, um dos principais objetivos que margearam o desenvolvimento do TALM foi a flexibilidade. Buscamos construir um modelo que pudesse ser implementado em diversas plataformas (como FPGA, GPUs ou como máquina virtual) e oferecesse a possibilidade de paralelização de programas utilizando diferentes granularidades.

Dessa maneira, neste capítulo será apresentado o modelo de maneira abstrata e a forma como seu conjunto de instruções pode ser estendido através da definição de super-instruções de diferentes granularidades de acordo com o programa a ser paralelizado.

3.1 Conjunto de Instruções

Conforme dito, um dos objetivos do TALM é permitir que o programador defina super-instruções para estender o conjunto de instruções básico. Dessa forma, podemos ter um conjunto de instruções para cada aplicação paralelizada. Para que isso seja possível, o formato da instrução do TALM deve permitir flexibilidade para suportar a definição de instruções com diferentes números de operandos de entrada e de saída.

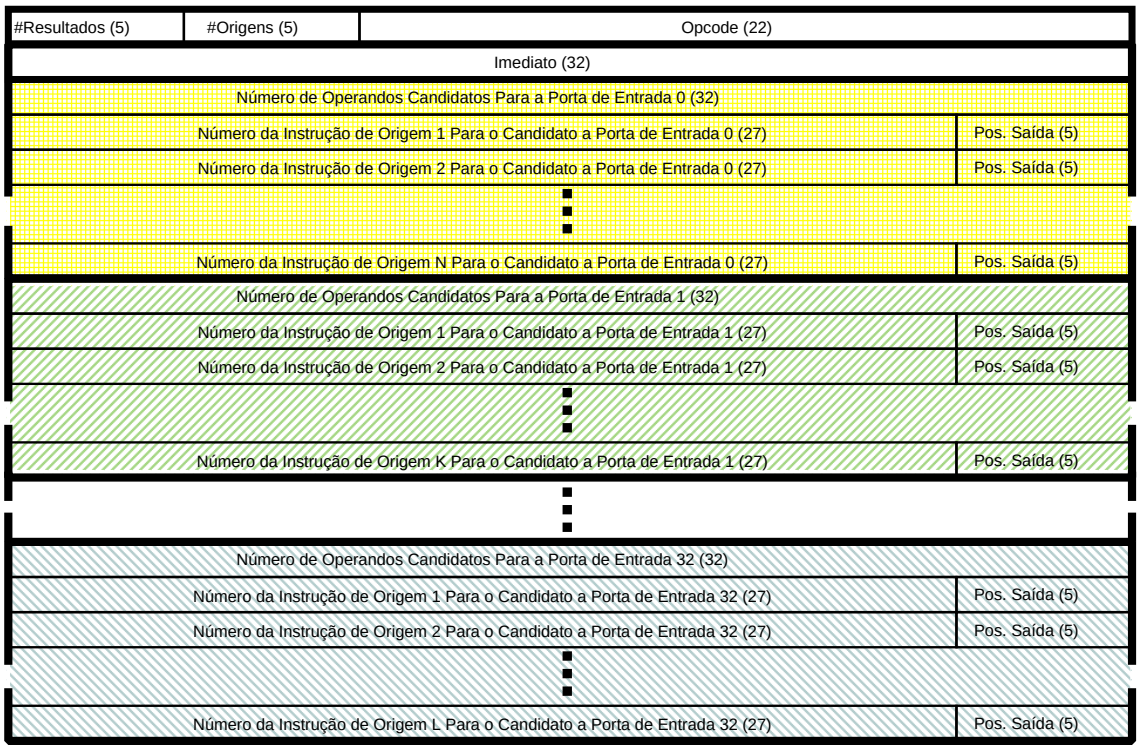


Figura 3.1: O formato de uma instrução.

A Figura 3.1 mostra o formato das instruções do TALM. Os primeiros 32 bits são obrigatórios para todas as instruções. Neles são definidos o código da operação (*opcode*), o número de operandos de origem e o número de operandos produzidos como resultado da instrução. Dessa forma, no formato dos campos da figura, uma instrução pode ter até 32 operandos de entrada e 32 operandos de saída. Os 32 bits seguintes são opcionais e definem o operando imediato para as instruções que possuem um.

Como o fluxo de dados no TALM pode seguir diferentes caminhos, um operando de entrada pode vir de mais de uma instrução de origem, dependendo do fluxo tomado na execução. Por essa razão os operandos de entrada de uma instrução são definidos na forma de listas das possíveis origens. Os primeiros 32 bits dessas listas indicam de quantas instruções o operando pode ser recebido, dependendo do fluxo. A seguir, para cada possível origem são definidos em dois campos, de 27 e 5 bits respectivamente, o número da instrução de origem e a porta de saída daquela instrução através da qual o operando pode ser enviado. Dessa forma, um valor $\langle i|p \rangle$ nesses dois campos indica que um operando pode ser recebido da p-ésima porta de saída da i-ésima instrução. Esse formato, portanto, limita o número de instruções a 2^{27}, limite esse que se mostrou ser mais do que suficiente em nossos experimentos.

O conjunto de instruções básicas (ou nativas) do TALM possui instruções lógicas e aritméticas usuais, tais como `add`, `sub`, `and`, `or` e suas variações com operando

imediato. Além dessas instruções esse conjunto possui também as instruções responsáveis pelo controle de execuções condicionais e de controle de laços.

Como no modelo dataflow não existe um contador de programa, desvios condicionais devem ser implementados como desvios no fluxo de dados. Isso é feito através da instrução `steer`, que recebe um valor O e um booleano S como seus operandos de entrada. Ao executar, a instrução `steer` envia o valor O por uma de suas duas portas de saída, dependendo do valor do booleano S. A Figura 3.2 mostra um exemplo de um grafo *dataflow* do TALM utilizando instruções `steer`, que são representadas por triângulos. No exemplo, as instruções `steer` enviam os operandos recebidos das instruções `IT` para as instruções ligadas às suas portas T, se o resultado de $<=$ for verdadeiro, ou para as instruções ligadas às portas F, se for falso.

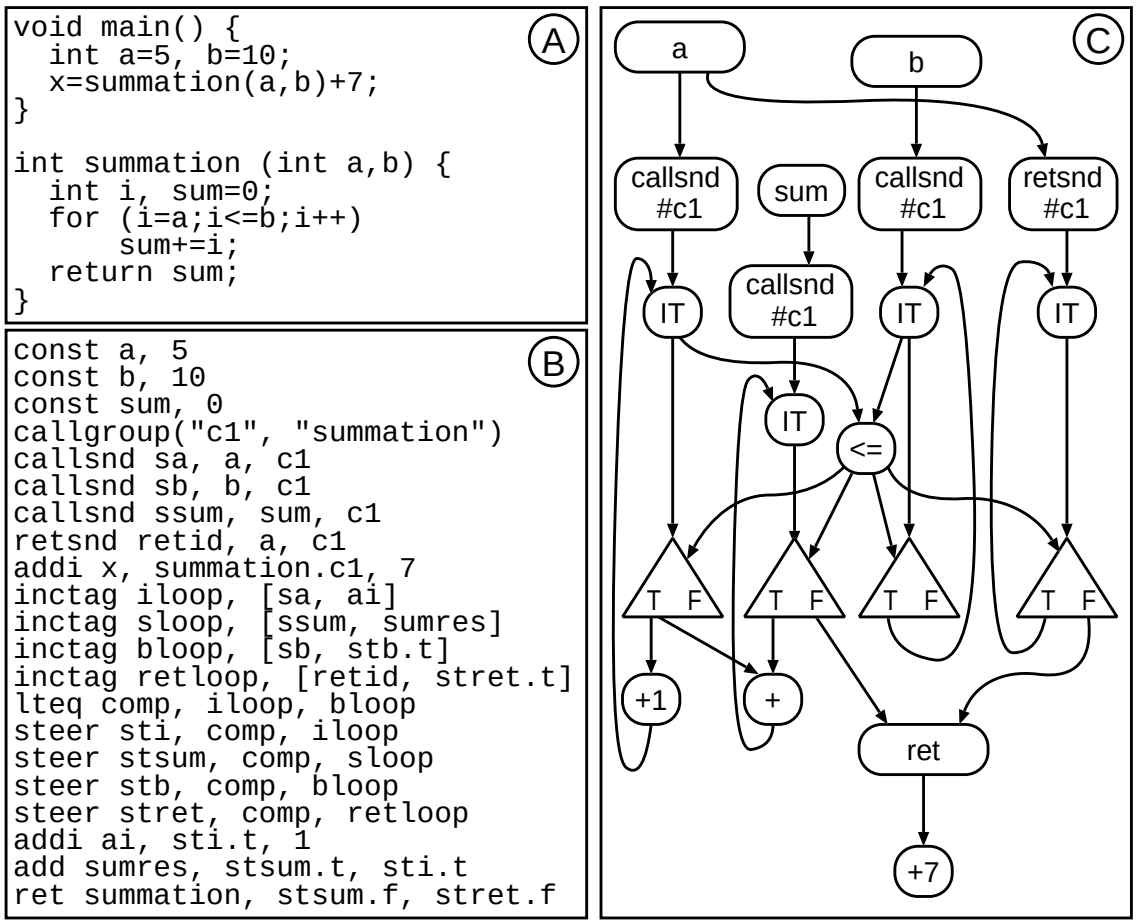


Figura 3.2: Exemplo do Assembly TALM

Por não haver despacho de instruções em ordem em uma máquina *dataflow*, é possível que partes de um laço rodem mais rápido que outras. Um exemplo desse efeito é quando as instruções responsáveis por incrementar o índice do laço e testar seu valor têm sua execução independente das outras instruções (como no laço da Figura 3.2). Dessa forma, essas instruções responsáveis pelo controle podem finalizar

uma iteração e entrar em outra antes que o resto do laço termine a iteração atual.

Tipicamente, arquiteturas *dataflow* adotam uma das seguintes abordagens para laços:

1. *Dataflow* estático: uma instrução não pode executar a iteração $N+1$ enquanto todas as instruções do laço não tiverem chegado ao fim da iteração N.

2. *Dataflow* dinâmico: instruções de um laço podem iniciar iterações sem que todas tenham finalizado a iteração anterior. Deve, então, haver uma forma de identificar operandos produzidos em diferentes iterações para evitar conflitos.

O nosso modelo adota a abordagem de *dataflow* dinâmico. Cada operando possui um rótulo correspondente à sua iteração. Quando um operando é enviado para a próxima iteração, seu rótulo deve ser incrementado para que ele passe a ser um operando daquela iteração. Isso é feito pelas instruções `inctag`, reprentadas por `IT` nas figuras.

Uma instrução em *dataflow* dinâmico pode executar quando receber todos os operandos de entrada com um mesmo rótulo de iteração. Chamamos cada uma dessas execuções disparadas pelo casamento de operandos com um mesmo rótulo de instâncias da instrução, sendo cada instância correspondente a uma iteração. A notação que usaremos para instâncias de instruções será $A(t)$, onde A é uma instrução e t o rótulo da instância. Note que no exemplo da Figura 3.2 quando a condição do laço ($\leq$) é avaliada como verdadeira os operandos são encaminhados pela porta `T` das instruções `steer` e em seguida enviados para a próxima iteração através das instruções `IT`.

No TALM, funções são blocos de instruções que podem receber operandos vindos de diferentes pontos de chamada no código, denominados *pontos de chamada estáticos*. Ao mesmo tempo, múltiplas chamadas podem ser feitas a partir de cada ponto de chamada estático, a exemplo de funções recursivas. Essas múltiplas chamadas são denominadas *pontos de chamada dinâmicos*. Para distinguir operandos de chamadas distintas da função usamos uma abordagem semelhante à adotada para distinguir operandos de iterações distintas.

Assim, cada operando possui dois rótulos, além do rótulo de iteração mencionado:

- Grupo da chamada (*callgroup*): usado para identificar o ponto de chamada do qual o operando veio;

- Número de instância da chamada: identifica as instâncias dinâmicas das chamadas feitas a partir de um mesmo ponto de chamada estático;

O número de instância da chamada é atualizado dinamicamente, através de um contador local presente nas instruções `callsnd`, que enviam operandos para as funções. O grupo da chamada é armazenado como o operando imediato das instruções

`callsnd` de um grupo. Em sua execução, a instrução `callsnd` incrementa seu contador local e produz *(i)* um operando com valor idêntico ao do operando de entrada, *(ii)* grupo da chamada igual ao imediato da instrução e *(iii)* número de instância da chamada igual ao contador local.

De maneira análoga aos laços, só há casamento de operandos com os mesmos rótulos de chamada de função. Os rótulos da função chamadora devem ser enviados à função chamada para que esta possa retornar valores com esses rótulos. A aplicação dos rótulos da função chamadora aos valores retornados é necessária para que esses valores sejam usados por instruções da função chamadora. Isso é feito pela instrução `retsnd`, que produz um operando com os rótulos de chamada (grupo da chamada e número da instância da chamada) da função sendo chamada e cujo valor é igual aos rótulos de chamada da função chamadora. O operando produzido pela `retsnd` é enviado para uma instrução `ret` dentro da função. Ao final da execução da função, a instrução `ret` poderá enviar para a função chamadora um operando com os rótulos de chamada restaurados, isto é, iguais aos da função chamadora. Esse operando poderá, então, ser usado em casamentos com outros operandos na função chamadora.

3.2 Arquitetura TALM

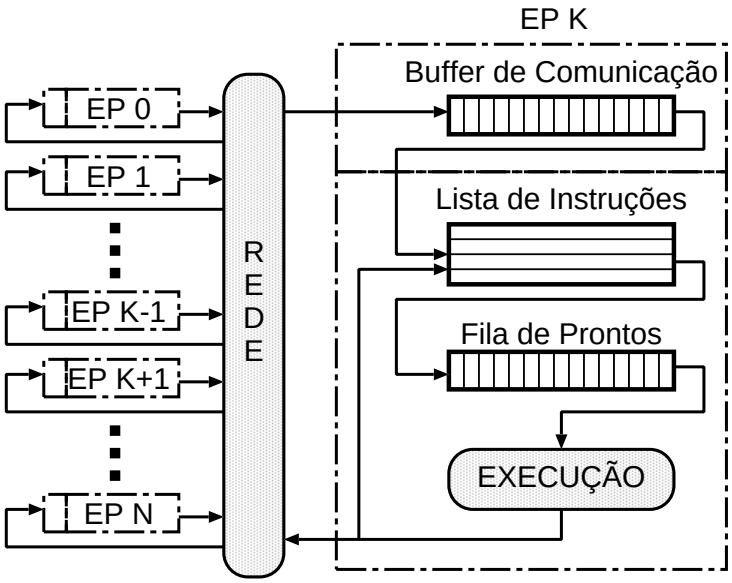


Figura 3.3: Arquitetura TALM.

A arquitetura TALM é mostrada na Figura 3.3. Essa a arquitetura é composta por um conjunto de Elementos de Processamento (EPs) idênticos interconectados por uma rede. Os EPs são responsáveis pela interpretação e execução de instruções de acordo com a regra de disparo *dataflow*. Cada EP possui um *buffer* de comuni-

cação para receber mensagens de outros EPs através da rede de interconexão. Essas mensagens podem conter operandos produzidos por instruções de diferentes EPs ou então marcadores usados por algoritmos distribuídos aplicados no sistema.

Quando uma aplicação é executada no TALM, suas instruções devem ser mapeadas para os diferentes EPs disponíveis. Além disso, cada instrução possui uma lista dos operandos recebidos correspondentes às suas diferentes instâncias. O casamento de operandos correspondentes a uma mesma instância, isto é, operandos com os mesmos rótulos de iteração e de chamada, é feito nessa lista. Quando há um casamento, os operandos são enviados junto com o código de operação da instrução para uma fila de instâncias prontas para a execução. O EP remove instâncias dessa fila de prontos para enviá-las para execução.

Além disso, como não há contadores de programa em máquinas *dataflow*, é necessário encontrar um meio de determinar quando a execução do programa chegou ao fim. No nosso modelo isso é feito através de um algoritmo distribuído de detecção de terminação global, que deve ser adaptado de acordo com a implementação da máquina. A implementação feita para a Trebuchet é descrita em detalhes na Seção 4.3.

Capítulo 4

Trebuchet: TALM para máquinas *multicore*

A Trebuchet é uma máquina virtual que implementa o TALM para máquinas *multicore*. Neste capítulo discutiremos as principais questões relacionadas à implementação e descreveremos como usar a Trebuchet para desenvolver programas paralelos.

4.1 Implementação da Máquina Virtual

O TALM é baseado em troca de mensagens, sendo as rotinas `Envia()` e `Recebe()` responsáveis pela troca de operandos entre EPs. Como a Trebuchet deve rodar em uma máquina hospedeira *multicore*, a troca de mensagens é implementada através de acessos às filas de entrada de cada EP, que ficam armazenadas em regiões de memória compartilhada. Dessa maneira, os custos de comunicação dessa implementação são relacionados aos mecanismos de sincronização (*locks*) utilizados para o acesso às estruturas na memória compartilhada. Note que além de procurar espalhar instruções independentes pelos EPs para explorar paralelismo, devemos também procurar concentrar instruções com certo grau de dependência em EPs próximos de maneira a diminuir os custos de comunicação.

A Figura 4.1a mostra a implementação da estrutura que representa uma instrução na lista de instruções de um EP. Uma breve explicação de cada um dos campos dessa estrutura segue:

- `Opcode`: contém o código de operação da instrução;
- `Imediato`: o valor do operando imediato, se a instrução o tiver;
- `#Origens`: número de operandos de entrada;
- `#Resultados`: número de operandos de saída produzidos em uma execução;

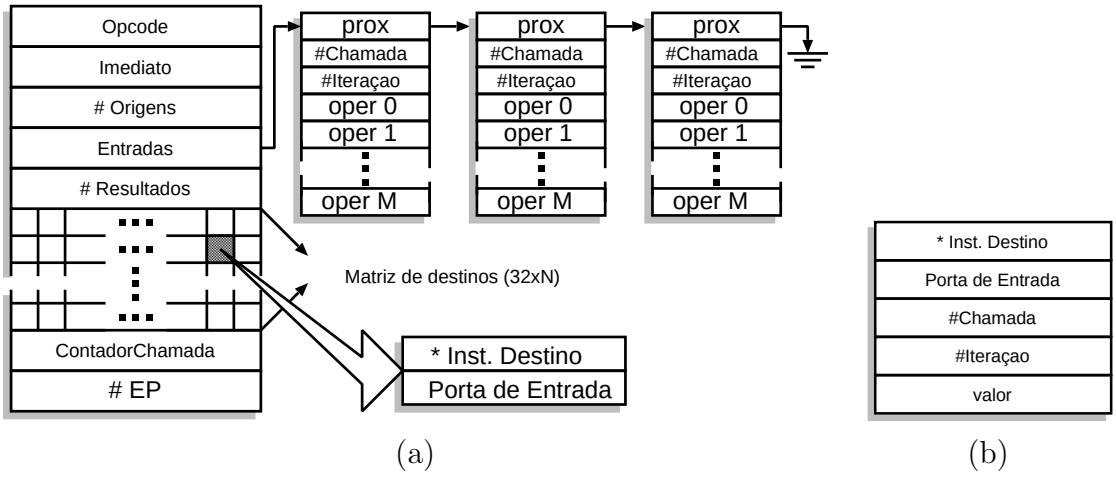


Figura 4.1: Estruturas da Trebuchet.

- `Entradas`: lista encadeada de estruturas de casamento. Para cada instância da instrução, isto é, para cada chamada/iteração que provocar uma execução dela, será criada uma estrutura de casamento para armazenar os operandos daquela instância. Nessa estrutura, `#Chamada` contém os rótulos de chamada (grupo de chamada na parte alta e número de instância na parte baixa) e `#Iteração` contém o rótulo da iteração. Quando todos os `#Entradas` operandos de uma instância chegarem na instrução, a estrutura de casamento contendo todos eles é retirada da lista encadeada e enviada para execução na forma de uma estrutura de despacho. A estrutura de despacho contém um ponteiro para a instrução sendo despachada para execução e os operandos usados no casamento;

- Matriz de destinos: para cada operando de saída, há uma linha contendo um par composto pelo ponteiro para uma instrução que deverá receber aquele operando e o número da porta de entrada dessa instrução alvo;

- `ContadorChamada`: o contador local utilizado em instruções `callsnd` e `retsnd`;

- `#EP`: o número de identificação do EP ao qual a instrução foi associada;

Na Figura 4.1b está representada a estrutura da mensagem que carrega operandos entre EPs diferentes. Além do endereço da instrução alvo e dos rótulos de chamada e de iteração, precisamos especificar também a porta de entrada da instrução alvo que deve receber o operando. Considere, por exemplo, que a instrução alvo seja uma subtração. O operando do lado esquerdo deve ser enviado para a porta 0 e o do lado direito para a porta 1.

4.2 Paralelizando aplicações com a Trebuchet

Para implementar uma aplicação paralela usando a Trebuchet é necessário compilá-la para a forma de um grafo *dataflow* representado na linguagem de montagem do TALM. A Trebuchet é implementada como uma máquina virtual e cada um de seus EPs é mapeado para uma *thread* da máquina hospedeira. Quando um programa é executado na Trebuchet, suas instruções são associadas a EPs e disparadas de acordo com a regra *dataflow*. Instruções independentes executarão em paralelo se forem mapeadas para EPs diferentes e se houver núcleos de processamento disponíveis na máquina hospedeira para executar as *threads* dos EPs concorrentemente.

Como os custos de interpretação de instruções podem ser altos, compilar um programa inteiramente para *dataflow* pode apresentar performance insatisfatória. Para contornar esse problema, o programador tem a possibilidade de estender o conjunto de instruções da Trebuchet com suas próprias instruções especiais utilizadas no seu programa. Para estender o conjunto de instruções o programador deve escrever blocos de código em linguagem imperativa com entradas e saídas especificadas. Esses blocos são chamados de *super-instruções* e são compilados na forma de uma biblioteca que é carregada dinamicamente pela Trebuchet na hora da execução. Note que as super-instruções são tratadas da mesma forma que as instruções nativas da Trebuchet e, portanto, também são disparadas de acordo com a regra *dataflow*. A Figura 4.2 descreve esse fluxo de trabalho para o desenvolvimento de um programa paralelo com o uso de super-instruções.

A Figura 4.3 mostra um exemplo do desenvolvimento de uma versão paralela de um programa simples. Em A temos o programa original em sua versão sequencial e em B é implementada uma super-instrução que executará uma parte do laço. Note que na implementação dessa super-instrução são utilizados os valores de entrada e é definido o valor de saída. Os valores de entrada são enviados por outras instruções através das arestas que chegam à instrução no grafo *dataflow*, enquanto que o valor de saída é enviado às instruções que recebem uma aresta da super-instrução no grafo.

Ainda nessa figura, em D e E vemos o grafo *dataflow* do programa na linguagem de montagem TALM e sua representação gráfica, respectivamente. Na linguagem de montagem, a macro `superinst` é utilizada para descrever uma super-instrução incluída para estender o conjunto de instruções. Neste caso, descrevemos a super-instrução implementada em B e a ela damos o nome `sP`. Em seguida incluímos quatro instruções `sP` responsáveis por executar uma parte do laço cada uma. As saídas de cada uma dessas instruções são ligadas a instruções de soma de ponto flutuante para que seja feita a redução.

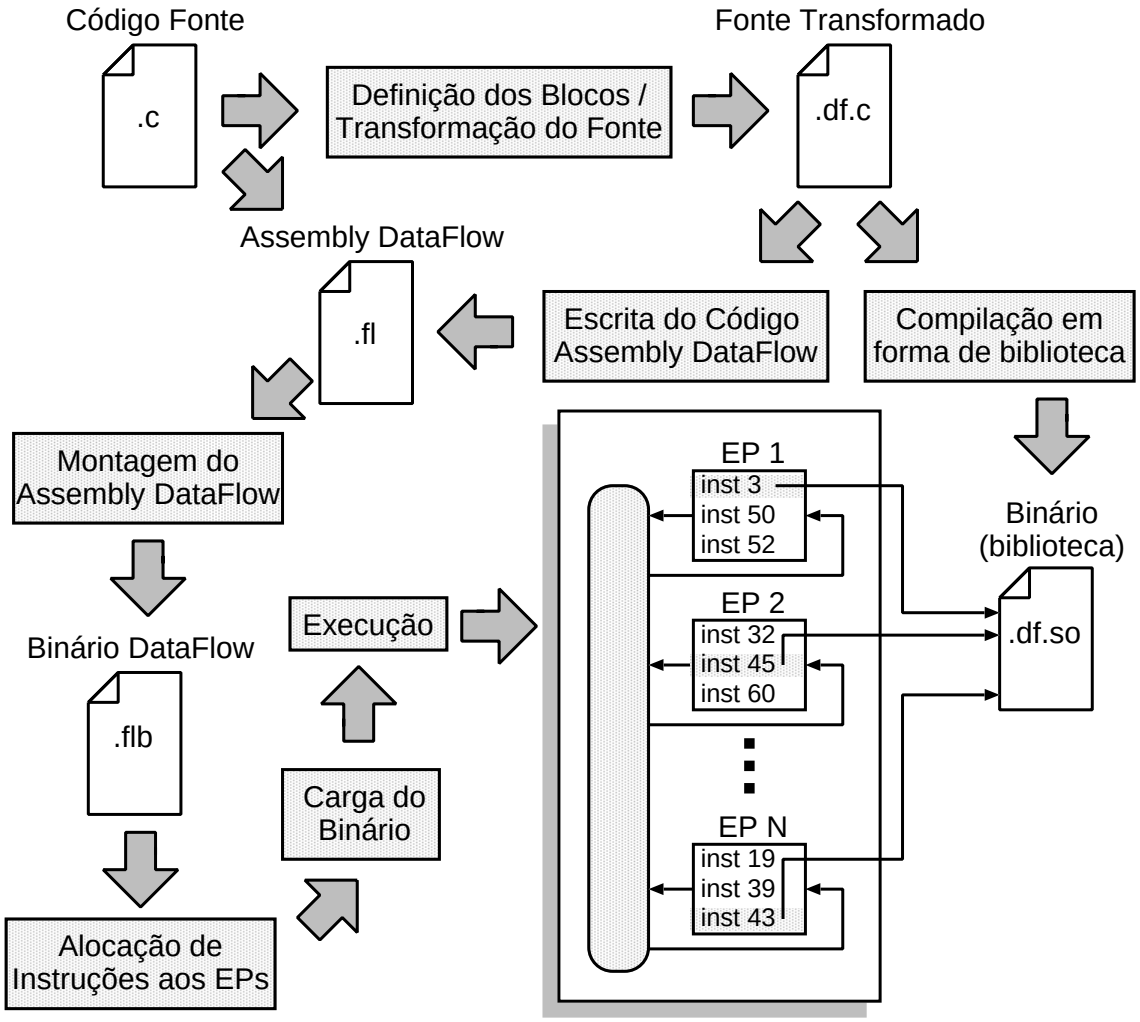


Figura 4.2: Fluxo de Trabalho da Trebuchet.

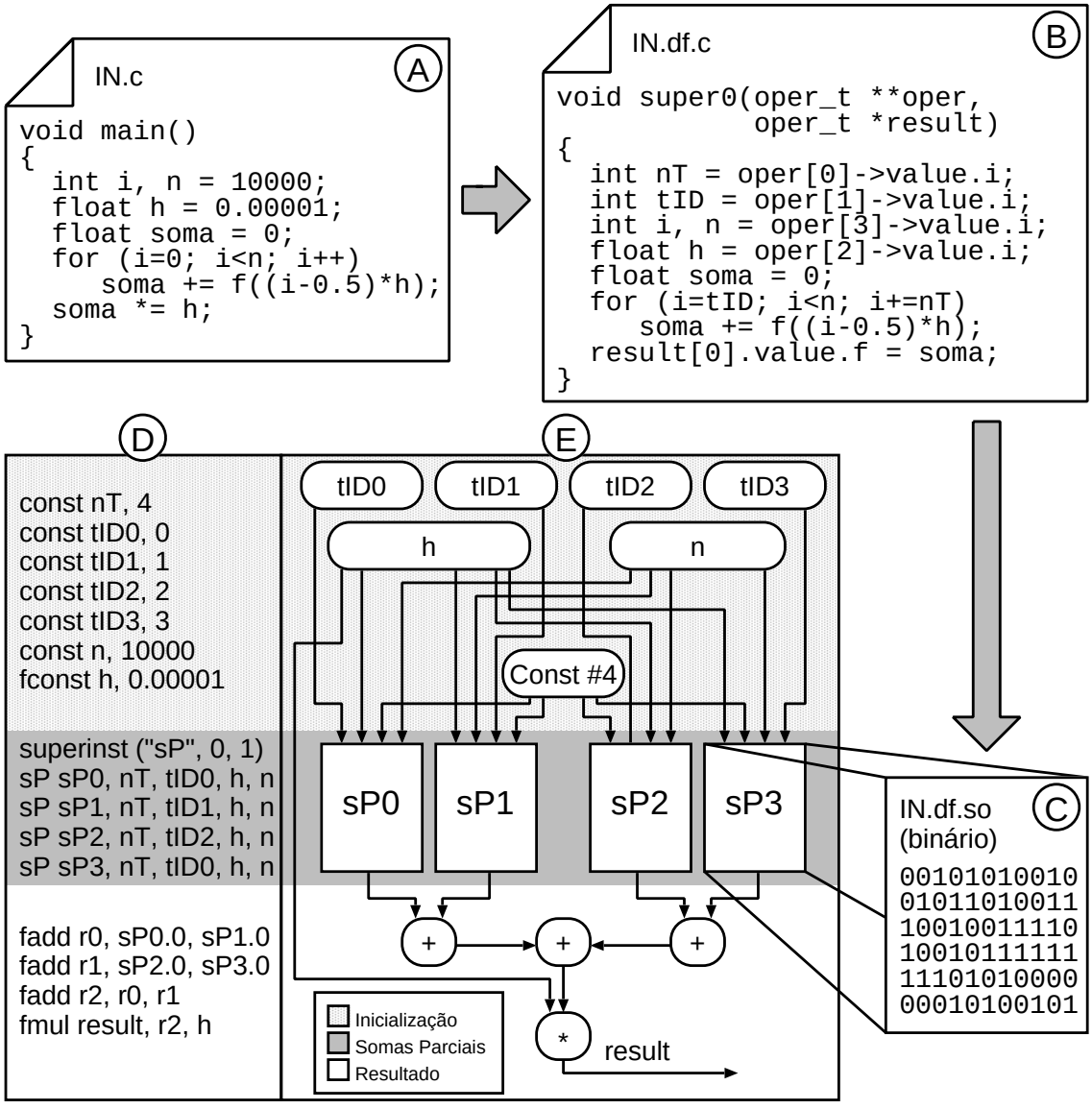


Figura 4.3: Usando a Trebuchet.

4.3 Detecção de Terminação Global

No modelo Von Neumann, o fim da execução é detectado quando o contador de programa chega ao fim do código. No caso de máquinas com múltiplos núcleos de processamento Von Neumann essa condição também é adequada, já que em determinado momento todas as *threads* deverão convergir em uma barreira para que a *thread* pai declare o fim do programa.

Em máquinas *dataflow*, esse propriedade do estado da execução não pode ser inferida tão facilmente já que os EPs possuem apenas informação sobre o seu estado local. Quando um EP se encontra ocioso, isto é, sem instruções na sua fila de prontos, ele não pode concluir que o programa terminou, pois podem chegar novos operandos em seu buffer de comunicação que desencadearão novas execuções. Por esse motivo é necessária uma maneira de determinar que não há outros EPs ainda em atividade e que não há operandos atravessando a rede de interconexão.

Para esse fim, uma implementação personalizada do algoritmo de *snapshots* distribuídos [16] foi desenvolvida. O algoritmo detecta um estado no qual todos os EPs estão ociosos e todas as arestas de comunicação entre EPs estão vazias, o que significa que o programa chegou ao fim, pois não há mais trabalho a ser feito. Há duas características fundamentais nesse algoritmo de detecção de terminação: como a topologia de rede da Trebuchet é um grafo completo, não é necessário haver um líder (cada nó pode por si mesmo obter todas as informações necessárias dos outros nós) e como precisamos apenas detectar o fim do programa nós usamos apenas um tipo de mensagem, o marcador de terminação. Observe que o canal de comunicação da Trebuchet respeita a propriedade FIFO (*First in, First out*) necessária, já que é implementado somente com filas e sem roteamento.

Um nó (EP) inicia uma detecção de terminação global ao entrar no estado ($ocioso = 1$), o que significa que não há mais instruções na sua fila de prontos. O nó então faz difusão de um marcador com um *rótulo de terminação* igual ao maior rótulo já visto incrementado de um. Outros nós entrarão nessa detecção se estiverem no estado ($ocioso = 1$) e se o rótulo recebido for maior que o maior rótulo visto por eles até o momento. Depois de entrarem nessa nova detecção os outros nós também fazem difusão do marcador recebido, indicando a todos os nós que eles estão no estado ($ocioso = 1$).

Uma vez que um nó participando de uma detecção de terminação tenha recebido marcadores daquela terminação de todos os outros nós, ele faz difusão novamente desse marcador, com o mesmo rótulo de terminação, se ainda estiver no estado ($ocioso = 1$). O segundo marcador enviado carrega a informação de que todas as arestas de entrada do nó estão vazias, já que o nó continuou no estado ($ocioso = 1$) durante a primeira etapa de recebimento de marcadores dos outros nós. Caso o nó

tivesse recebido mensagens contendo operandos durante essa primeira etapa, ele teria saído do estado ($ocioso = 1$) e, portanto, teria deixado essa detecção de terminação.

Então, nessa implementação, usamos apenas um tipo de mensagem: o marcador com o rótulo de terminação. A primeira rodada de marcadores enviados carrega a informação do estado dos nós, ($ocioso = 1$), enquanto a segunda rodada carrega a informação do estado das arestas de entrada dos nós (todas vazias). Fica claro, então, que após receber a segunda rodada de marcadores de todos os vizinhos (todos os outros nós, já que a topologia é de grafo completo), o nó pode terminar.

4.4 Descrição de *Pipeline* nos laços *Dataflow*

Pipelining é um padrão de exploração de paralelismo no qual uma operação é subdividida em operações menores que correspondem a etapas do seu processamento. O paralelismo exposto por essa técnica se deve ao fato de que todas as etapas são executadas concorrentemente, cada uma processando um conjunto de dados diferente, que será em seguida passado para uma das próximas etapas. O modelo *dataflow* possibilita que um *pipeline* seja descrito com facilidade, bastando descrever com arestas o fluxo de dados esperado.

Suponha que temos um programa que faz compressão do vídeo recebido de uma fonte e envia para um servidor de *streaming*. Podemos dividir esse programa em três etapas paralelizáveis: recebimento de *frames*, compressão das *frames* recebidas e envio das *frames* comprimidas. Considere que a primeira etapa recebe um conjunto de *frames* a cada execução e que essas *frames* serão em seguida comprimidas em conjunto e reenviadas nas etapas seguintes. Como não há dependência nas operações sobre conjuntos de *frames* distintos, a primeira etapa pode iniciar uma nova iteração (receber mais *frames*) antes que a segunda e a terceira tenham terminado a anterior. Essa implementação pode, então, ser classificada como um *pipeline*, já que após duas execuções da primera etapa as três etapas passarão a executar em paralelo.

Para descrever esse programa na forma de um *pipeline* no TALM basta implementar cada uma dessas etapas em uma super-instrução, incluir as devidas arestas entre elas e incluir o subgrafo resultante em um laço. A Figura 4.4 ilustra esse exemplo. Note que a primeira super-instrução é quem determina se o laço deve continuar, pois ela é quem receberá o aviso de terminação da fonte do vídeo. A super-instrução `Inicializa` além dos procedimentos de inicialização do programa produz o *token* que irá liberar as três primeiras iterações, para encher o *pipeline*. As iterações a partir da quarta serão liberadas pelo *token* produzido pela super-instrução `Envia`. Assim, cada execução da iteração N da super-instrução `Envia` produz um *token* que ativará a iteração $N + 3$ da super-instrução `Recebe`.

Na Figura 4.5 apresentamos um exemplo mais complexo do uso de *pipelining*.

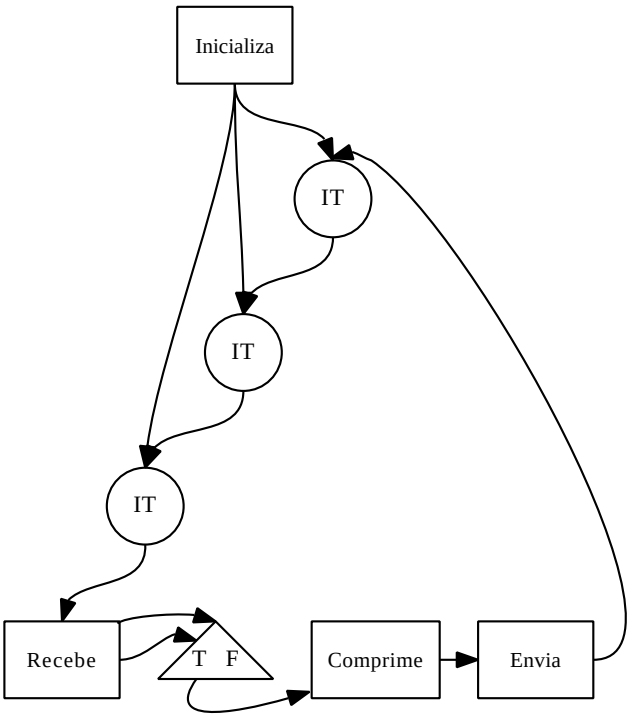


Figura 4.4: Exemplo de *Pipeline* de compressão de vídeo.

Em A temos o código sequencial, em B as super-instruções criadas para paralelizar o código e em C o grafo *dataflow* no qual as super-instruções são utilizadas.

A computação dos valores de X depende da computação dos valores de K da iteração anterior do laço `while`. Já a computação dos valores de K depende da computação dos valores de N da iteração anterior o laço `while`. Além disso, o valor da variável `stay` de controle do laço depende da computação dos valores de K. Dessa forma, a super-instrução `s0`, responsável pelo cálculo de X, pode iniciar a próxima iteração assim que `s1` terminar a iteração atual, sem precisar esperar pelo término de `s2`. Essas condições são descritas no grafo apresentado em C, onde uma execução de `s1` envia um operando que poderá disparar a execução de `s0` da próxima iteração. Portanto, `s0` e `s2` estiverem mapeadas em EPs diferentes, `s0` poderá iniciar a próxima iteração enquanto `s2` ainda executa.

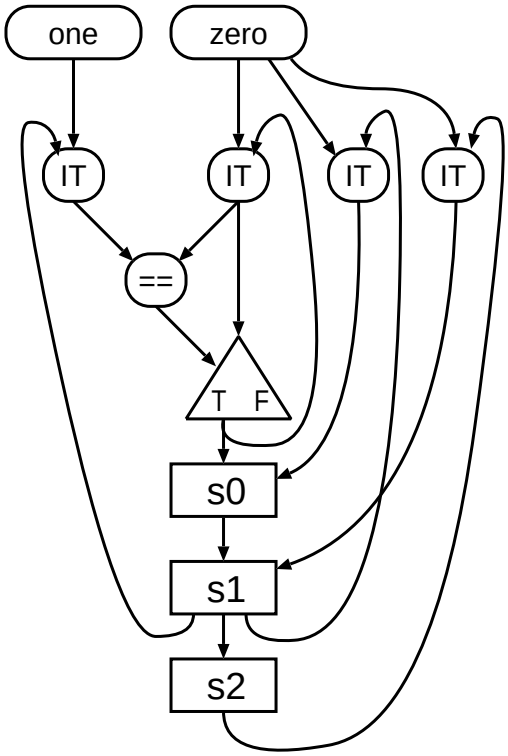
A flexibilidade para descrever esse tipo de *pipeline* é um ponto forte do TALM em comparação a outros modelos. Não seria possível obter o paralelismo da figura 4.5 usando o *template* de *pipeline* do *Intel Threading Building Blocks* [17], por exemplo, já que esse permite apenas a implementação de *pipelines* lineares.

```
stay = 1;
while stay{
   for(j=0; j<MAX; j++){
      X[j]+=j+K[j];
      K[j]+=X[j]+N[j];
      stay=(K[j]==300)?0:stay;
      N[j]+=K[j]+2;
   }
}
```

A

```
void super0(oper_t **oper, oper_t *result){
   for(j=0; j<MAX; j++)
      X[j]+=j+K[j];
}
void super1(oper_t **oper, oper_t *result){
   for(j=0; j<MAX; j++){
      K[j]+=X[j]+N[j];
      stay=(K[j]==300)?0:stay;
   }
   result[0].value.i = stay;
}
void super2(oper_t **oper, oper_t *result){
   for(j=0; j<MAX; j++)
      N[j]+=K[j]+2;
}
```

B



C

Figura 4.5: Exemplo complexo de *Pipeline*.

Capítulo 5

Ordem Parcial Condicional

Neste capítulo apresentaremos o conceito de Ordem Parcial Condicional. Esse conceito será utilizado no Capítulo 6 para formalizar as restrições impostas aos grafos do nosso modelo de execução especulativa.

5.1 Ordem Parcial

Definimos uma ordem parcial $P(C, \prec)$ como uma relação binária entre elementos de um conjunto C que satisfaz as seguintes condições:

1. *Irreflexibilidade*: $\forall a \in C, \neg(a \prec a)$
2. *Assimetria*: $\forall a, b \in C, (a \prec b) \Rightarrow \neg(b \prec a)$
3. *Transitividade*: $\forall a, b, c \in C, (a \prec b) \wedge (b \prec c) \Rightarrow (a \prec c)$

Dessa forma, dado um grafo dirigido sem ciclos $D(V, E)$, onde V é o seu conjunto de vértices e E o seu conjunto de arestas, definimos como ordem parcial relacionada ao grafo D, a ordem parcial $P(V, \prec)$ obtida a partir da seguinte regra:

$$\forall a, b \in V, (a \prec b) \iff (\text{existe caminho de } a \text{ para } b \text{ em } D) \wedge a \neq b. \tag{5.1}$$

A aplicação dessa regra a um grafo direcionado $D(V, E)$ para a obtenção da ordem parcial $P(V, \prec)$ relacionada será denotada neste trabalho por $P = p(D)$.

Podemos aplicar a Regra 5.1 a um grafo *dataflow* sem laços e sem instruções de controle. Seja um grafo *dataflow* sem laços e sem instruções de controle $D(I, E)$, onde I é o conjunto das suas instruções (vértices) e E o conjunto de arestas entre as instruções. Pelas restrições impostas e considerando que não há *deadlocks*, D é um grafo dirigido acíclico e, portanto, podemos aplicar a Regra 5.1 para obter uma ordem parcial para ele. Seja $P(I, \prec) = p(D)$ a ordem parcial relacionada a

D, $\forall a, b \in I, (a \prec b)$ significa que b depende direta ou indiretamente de operandos produzidos por a, logo, b só poderá ser executada depois de a.

Como exemplo, considere o grafo *dataflow* $D(I, E)$ da Figura 5.1 onde $I = \{A, B, C, *, +\}$. Dada a ordem parcial $P(I, \prec) = p(D)$, os pares $(x, y) \in I \times I$ tais que $x \prec y$ são:

$$(A, +), (A, *), (B, +), (B, *), (C, *), (+, *)$$

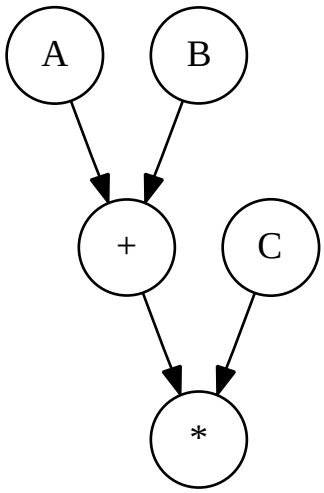


Figura 5.1: Grafo dataflow sem laços ou instruções de controle

5.2 Ordem Parcial Condicional

Nesta seção apresentaremos definições baseadas nas apresentadas em [18].

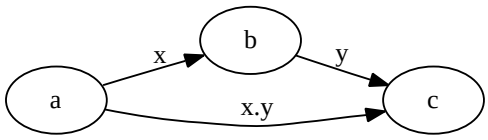
5.2.1 Grafo de Ordem Parcial Condicional

Um grafo de ordem parcial condicional (GOPC) é uma tupla $H(V, E, X, \phi)$, onde V é o seu conjunto de vértices, E o seu conjunto de arestas, X um conjunto de variáveis booleanas e a função $\phi : E \rightarrow F(X)$ atribui uma expressão booleana para cada aresta no grafo. Ao contrário da definição apresentada em [18], não precisamos atribuir expressões booleanas para os vértices do grafo.

A expressão booleana de uma aresta $e \in E$ é a função $\phi(e) \in F(X)$, onde $F(X)$ é o conjunto de todas as funções booleanas sobre as variáveis de X. No exemplo da Figura 5.2 é mostrado um GOPC e a tabela dos valores de $\phi(e)$ para as arestas e do grafo. No GOPC desse exemplo, $X = \{x, y\}$.

e	$\phi(e)$
(a,b)	x
(a,c)	$x.y$
(b,c)	y

(a)



(b)

Figura 5.2: Exemplo de grafo de ordem parcial condicional

5.2.2 Projeções

Uma projeção em um grafo de ordem parcial condicional $H(V, E, X, \phi)$ é a atribuição de um valor $\alpha \in \{0, 1\}$ a uma variável $x \in X$, sendo essa operação denotada por $H\ |_{x=\alpha}$. Essa operação fornece um GOPC $H'(V, E, X \backslash \{x\}, \phi\ |_{x=\alpha})$, onde $\phi\ |_{x=\alpha}$ significa que a variável x é substituída por α em todas as funções $\phi(e), e \in E, \phi(e) \in F(X)$.

Uma projeção completa de um GOPC $H(V, E, X, \phi)$ é uma projeção tal que para cada variável $x \in X$ é atribuído um valor $\alpha \in \{0, 1\}$. Note que uma projeção completa de $H(V, E, \{x_0, x_1, .., x_n\}, \phi)$, pode ser obtida com n projeções, isto é, $H\ |_{x_0=\alpha_0}|_{x_1=\alpha_1}|\ ..\ |_{x_n=\alpha_n}$. Por simplicidade, definimos a função de atribuição $\psi : X \rightarrow \{0, 1\}$, que faz essas atribuições de um valor booleano para cada variável em X, e aplicamos a notação $H\ |_\psi = H'(V, E, \emptyset, \phi\ |_\psi)$.

Tendo essa definição de projeção completa podemos obter um grafo direcionado $D(V_d, E_d)$ a partir de uma projeção completa $H\ |_\psi$ tal que:

$$\begin{cases} V_d & = V \\ E_d & = \{e \in E \ \mid \ \phi\ |_\psi\ (e) = 1\} \end{cases}$$

Por simplicidade, usaremos a notação $D = g(H\ |_\psi)$ para denotar o grafo dirigido obtido a partir de uma projeção completa $H\ |_\psi$. Uma projeção completa é chamada válida se e somente se o grafo obtido a partir dela for um grafo direcionado acíclico.

Seja $D(V_d, E_d)$ o grafo direcionado acíclico obtido a partir de uma projeção completa válida $H\ |_\psi$. Denominamos ordem parcial condicional a ordem parcial $P(V_d, \prec_\psi)$ relacionada a D, isto é, $P = p(g(H\ |_\psi))$.

5.2.3 Remoção de laços

Como precisaremos de um grafo dataflow sem laços para a criação de nossos grafos de ordem parcial condicional, devemos encontrar um meio de remover laços, mas manter representadas as relações de dependência entre duas iterações. Isso é feito através da linearização de duas iterações do loop como exemplificado na Figura 5.3.

Na Figura 5.3, as instruções `IT` são responsáveis por avançar operandos para a próxima iteração, conforme foi visto no Capítulo 3. Dessa forma, as arestas $(+1, IT)$ e (A, IT), representam a passagem de operandos para a iteração seguinte. Com a linearização de duas iterações podemos remover os ciclos mantendo a representação da relação de dependência entre duas iterações.

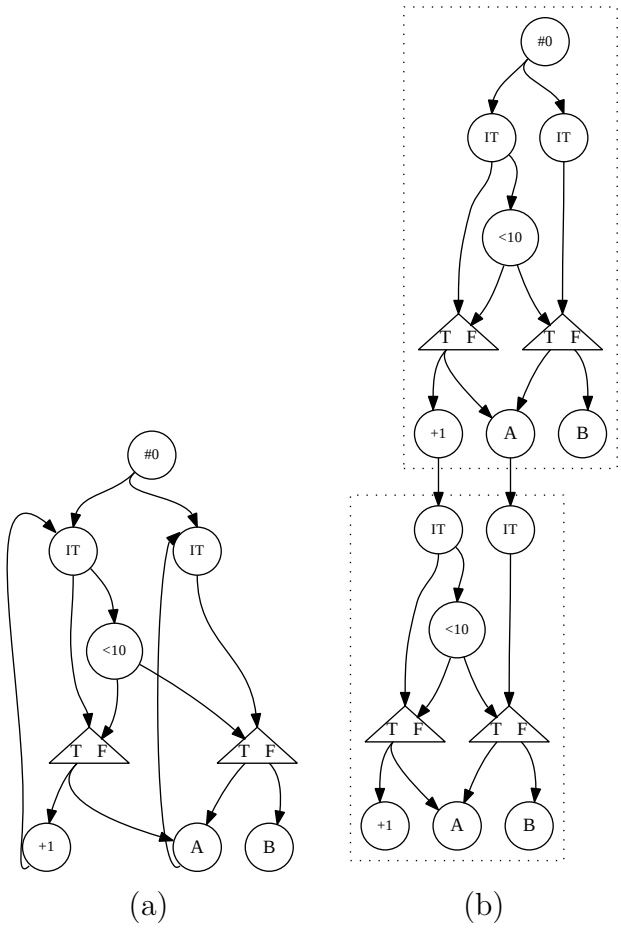


Figura 5.3: Exemplo de remoção de ciclos através de linearização de laços. Em (a) o grafo original e em (b) o grafo linearizado.

5.2.4 Criação de um GOPC a partir de um grafo dataflow

O algoritmo a seguir recebe como entrada um grafo dataflow sem laços e retorna um grafo de ordem parcial condicional. A idéia básica é atribuir uma variável a cada valor booleano produzido para alimentar instruções de controle e usar essas variáveis para escrever as expressões das arestas.

Em um grafo dataflow os vértices representam instruções e as arestas representam as relações de produção/consumo de operandos entre elas. Como os operandos de entrada de uma instrução devem ser distinguidos, dizemos que cada aresta chega em uma das portas de entrada de uma instrução. Em uma instrução de divisão,

por exemplo, arestas representando o divisor chegam em uma porta e arestas do dividendo em outra. Quando for necessário especificar portas de entrada/saída, usaremos a notação (a_p, b_q) para representar uma aresta que parte da porta de saída p da instrução a e chega na porta de entrada q da instrução b. Se não for necessário ou desejável especificar a porta de entrada ou saída o subscrito correspondente será omitido. Dessa forma (a, b) representa uma aresta saindo de uma porta qualquer de a e chegando em uma porta qualquer de b e (a_p, b) representa uma aresta partindo da porta p de saída de a.

Para descrever o algoritmo nos referiremos às portas de entrada das instruções `Steer` como porta de seleção e porta de operando. A porta de seleção recebe o operando booleano, chamado aqui de sinal de controle. Esse sinal de controle define por que porta de saída o operando recebido pela porta de operando será enviado.

A idéia central do algoritmo é propagar pelo grafo *dataflow* as condições para que operandos sejam enviados através de cada uma de suas arestas. Essa propagação começa a partir das instruções que são raízes, isto é, não necessitam de operandos de entrada, e é feita pelo procedimento recursivo *percorre()*.

Nesse procedimento são atribuídas expressões booleanas para cada aresta e que sai da instrução sendo visitada. A expressão $\phi(e)$ representa a condição para que um operando seja enviado pela aresta e. No caso de arestas que saem de instruções que não são `Steers`, essa expressão é simplesmente a condição para que a instrução da qual a aresta sai seja executada. Isso se deve ao fato de que essas instruções enviam operandos por todas as suas portas de saída sempre que executam. Logo, $\phi(e)$ para as arestas de saída de instruções que não são `Steers` será o E-lógico das condições para que ela receba um operando em cada porta de entrada.

Para as instruções `Steer`, as condições de envio pelas portas de saída 0 e 1 são diferentes, já que dependem do valor booleano de entrada. Pela porta 1, o envio depende do booleano ser verdadeiro e pela porta 0 depende do booleano ser falso. Naturalmente, ambos os casos dependem também das condições normais para que a `Steer` execute, ou seja, dependem de que a instrução receba ao menos um operando em cada porta.

Os sinais de controle do programa são representados pelas expressões $b(e)$. Conforme o algoritmo vai percorrendo o grafo, vai atribuindo uma expressão booleana $b(e)$ para cada aresta. Para as instruções que não são `Steers` é criada uma variável x_p para cada porta de saída, representando um novo sinal de controle a ser propagado. Assim, a expressão $b(e)$ de cada aresta que sai dessa instrução recebe a x_p da porta de saída correspondente.

As instruções `Steer`, por outro lado, devem propagar o sinal recebido pela porta de entrada 1 (porta de operando) até que ele chegue a uma instrução que não é `Steer` ou à porta de seleção de uma outra `Steer`. Para isso, a expressão $b(e)$ das

arestas que saem de uma instrução `Steer` é definida como o OU-lógico dos sinais recebidos na porta de operando, associados às condições para que sejam recebidos respectivamente. Isto é, $b(e) := \bigvee\{\phi(e') \wedge b(e') : e' = (k, i_1)\}$, onde i é a instrução `Steer` sendo visitada.

O algoritmo percorre o grafo *dataflow* a partir das raízes, instruções sem operandos de entrada. Para que uma instrução seja visitada pelo procedimento *percorre*, as expressões booleanas de todas as arestas direcionadas a ela devem ter sido definidas. No caso das raízes isso já é verdade no início do algoritmo, pois não há arestas chegando nelas.

Ao final do algoritmo, teremos as expressões $\phi(e)$ definidas para todas as arestas e, portanto, obteremos um GOPC.

Algoritmo 5.1 Criação de um GOPC a partir de um grafo *dataflow* sem laços

---

Entrada: Grafo *dataflow* $D(I, E)$ sem laços. Se há laços, os linearize como na figura 5.3.
Saída: Um GOPC $H(I, E, X, \phi)$
Inicialize o conjunto X como vazio.
para todo $i \in I$ tal que i é raiz **faça**
percorre(i)
Adicione a X todas as variáveis usadas em ϕ.
procedimento percorre(i)
Se $\forall e = (j, i) \in E$ $\phi(e)$ já foi definida **então**
Se i não é `Steer` **então**
Atribua uma nova variável x_p a cada porta p de saída de i.
exp_saida := 1
para todo porta p de entrada de i **faça**
exp_parcial_p := $\bigvee\{\phi(e) : e = (j, i_p) \in E\}$
exp_saida := exp_saida $\wedge$ exp_parcial_p
para todo $e = (i_p, j) \in E$ **faça**
$b(e) := x_p$
$\phi(e)$:= exp_saida
senão
exp_saida := $\bigvee\{\phi(e) : e = (j, i_1) \in E\}$
para todo $e = (i_0, j) \in E$ **faça**
condição := $\bigvee\{\phi(e') \wedge \neg b(e') : e' = (k, i_0) \in E\}$
$\phi(e)$:= exp_saida $\wedge$ condição
para todo $e = (i_1, j) \in E$ **faça**
condição := $\bigvee\{\phi(e') \wedge b(e') : e' = (k, i_0) \in E\}$
$\phi(e)$:= exp_saida $\wedge$ condição
para todo $e = (i, j) \in E$ **faça**
$b(e) := \bigvee\{\phi(e') \wedge b(e') : e' = (k, i_1) \in E\}$
para todo $e = (i, j) \in E$ **faça**
percorre(j)
fim procedimento

Na Figura 5.4 é mostrado um exemplo de execução do algoritmo 5.1 (em (a)) e

o grafo obtido a partir de uma projeção do GOPC obtido (em (b)). Em (a), cada aresta e foi marcada com o par ordenado $(\phi(e), b(e))$ correspondente à execução do algoritmo. Em (b), temos o grafo obtido a partir da projeção $H\ |_{x_1=0}$, que é uma projeção completa pois x_1 é a única variável presente em expressões ϕ. Observe que no cenário dessa projeção, D não seria executada, já que não recebe o operando de que precisa. Além disso, temos que E e F sempre executam, já que a condição da aresta (E, F) é $(x_1 \vee \overline{x_1} = 1)$. Essa condição expressa o fato do operando de entrada de E poder ser produzido por D ou por C.

Caso as duas arestas que chegam em E se dirigissem a portas de entrada diferentes, a condição da aresta (E, F) seria $(x_1 \wedge \overline{x_1} = 0)$ e assim E e F nunca executariam. Temos então que além de possibilitar a formalização de características dos grafos *dataflow*, a transformação em um GOPC também possibilita otimizações, como eliminar instruções que nunca serão executadas.

5.3 Grafo *Dataflow* bem formado

Usaremos neste trabalho o conceito de Ordem Parcial Condicional apresentado neste capítulo para formalizar definições sobre grafos *dataflow*. Mostraremos nesta seção uma formalização da nossa definição de grafo *dataflow* bem formado.

Adotamos o conceito de grafo *dataflow* bem formado para descrever programas nos quais uma instrução nunca recebe um operando que não será consumido e recebe apenas um operando por cada porta de entrada. Isto é, se uma instrução recebe um dos operandos de entrada, deve receber todos (um para cada porta de entrada) para fazer o casamento e executar, consumindo-os. Pelo restante deste trabalho consideraremos apenas grafos *dataflow* bem formados ao descrever o nosso modelo de execução. Formalizamos essa definição a seguir.

Definição 1. Seja $D(I, E)$ um grafo *dataflow*, onde I são suas instruções e E as arestas direcionadas de troca operandos entre elas. E seja H o GOPC obtido através da aplicação do Algoritmo 5.1 a D. D é considerado um grafo *dataflow bem formado* se e somente se, em todos os grafos obtidos a partir de projeções completas $H\ |_{\psi}$, para toda instrução $i \in I$ o número de arestas $(\cdot, i)$ for igual a 0 ou igual ao número de portas de entrada de i, chegando uma aresta em cada porta de entrada.

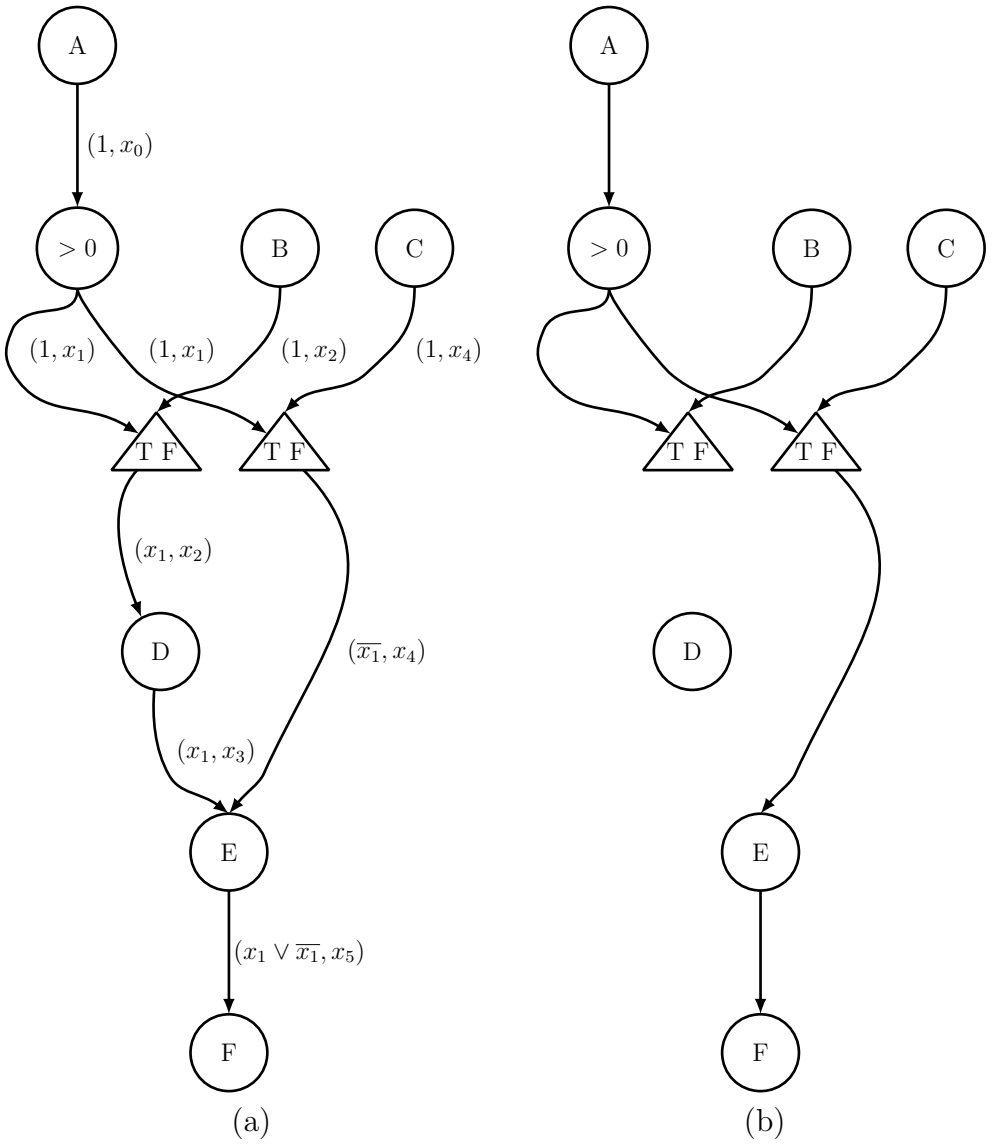


Figura 5.4: Exemplo de execução do Algoritmo 5.1 e uma projeção do GOPC obtido.

Capítulo 6

Execução *Dataflow* Especulativa

No modelo TALM, dependências de dados estáticas são resolvidas através da inclusão de arestas adicionais no grafo *dataflow*. Essa inclusão de arestas tem o efeito de serialização das operações de memória, já que pela regra *dataflow* se uma instrução recebe aresta de outra, a primeira executa somente depois da segunda ter executado. Por outro lado, dependências dinâmicas, isto é, dependências que só podem aparecem em tempo de execução, não podem ser resolvidas de maneira estática com alterações no grafo *dataflow* do programa.

Na arquitetura Wavescalar [3] a solução adotada é um esquema de anotações feitas pelo compilador que garantem uma ordem total entre todas as instruções de leitura/escrita na memória. Isso seria o equivalente a adicionar arestas entre todas as instruções de memória. Esse tipo de solução não se adequa ao propósito do TALM, pois elimina todo o paralelismo entre instruções que fazem acesso à memória, o que pode ser extremamente custoso no caso de instruções de granularidade grossa.

Por outro lado, o uso de transações, baseadas no modelo adotado em bancos de dados, tem se mostrado promissor para sistemas com *multithreading* (e.g.[11, 12, 19]). A estratégia é baseada na hipótese de que as dependências de dados estáticas são consideravelmente mais frequentes que as dinâmicas. Dessa forma, podemos executar instruções que fazem acesso à memória (e não têm dependências estáticas entre si) de maneira concorrente especulativamente. Caso alguma dependência seja encontrada, pode ser necessário reexecutar a instrução cujo conjunto de leitura foi modificado por outras transações. Caso contrário, os dados produzidos pelas instruções podem ser persistidos.

Outro método para sincronização do acesso a estruturas de dados compartilhadas é o uso de *locks* associados às mesmas. Delimitando-se seções críticas do código com *locks* pode-se garantir que as mesmas serão executadas atomicamente pelas *threads*. No entanto, o uso de *locks* limita o paralelismo (seções críticas que acessam a mesma estrutura de dados não executam concorrentemente) e introduz novas dificuldades à programação, devido à possibilidade de haver *deadlocks* no código com locks. Além

disso, esse método não garante que as alterações feitas nas estruturas de dados sejam vistas como obedecendo a ordem total do programa sequencial.

Para este trabalho decidimos usar especulação em nível de *threads* baseada em transações otimistas com a etapa de persistência das escritas feita em ordem, de maneira semelhante aos modelos apresentados em [12] e [11]. Essa escolha foi feita levando em conta as dificuldades e limitações das técnicas mencionadas acima e os custos de invalidação e reexecução causados pelo uso de transações. A motivação é que, em programas nos quais as dependências dinâmicas ocorrem com frequência proporcionalmente menor, os custos inerentes ao uso de transações são compensados pelos ganhos tanto no aumento de concorrência do programa paralelo quanto na facilidade de programação.

6.1 Modelo especulativo do TALM

Para permitir flexibilidade tanto na escrita de programas quanto na implementação de instâncias do TALM, os mecanismos de especulação desse modelo são baseados na inclusão de novas instruções e arestas no próprio grafo *dataflow* do programa. No modelo, qualquer instrução pode ser marcada como especulativa. Em sua implementação, uma instrução especulativa não pode afetar o estado da máquina, isto é, memória ou arquivos (não há registradores ou *flags* em máquinas *dataflow*), ou levantar exceções durante a sua execução. Para esse fim, tanto a escrita em um recurso da máquina quanto o levantamento de exceções devem ser separados da execução da instrução.

Isso é feito através do uso do modelo transacional na execução de instruções especulativas e da introdução de instruções `Commit`. A idéia básica é que a execução das instruções especulativas é feita em transações cujas escritas são persistidas pelas instruções `Commit` correspondentes. Dessa maneira, as instruções executam concorrentemente e possivelmente fora de ordem, mas as alterações do estado da máquina são feitas atomicamente e em ordem. No exemplo da Figura 6.1, as instruções especulativas $S1$, $S2$ e $S3$ são executadas fora de ordem e concorrentemente. Suas instruções `Commit` correspondentes, no entanto, executam em ordem, sequencialmente e somente após a instrução especulativa correspondente ter terminado.

A instrução `Commit` é responsável pela validação do conjunto de leitura da instrução correspondente, pela persistência do conjunto de escrita no estado da máquina e por levantar eventuais exceções ocasionadas pela execução da instrução. Para isso, é criada uma aresta entre a porta de saída 0 da instrução especulativa e a porta de entrada 0 da instrução `Commit`. A instrução especulativa enviará, através dessa aresta, *(i)* as estruturas de dados contendo seus conjuntos de leitura e de escrita, *(ii)* uma mensagem indicando a presença de uma exceção, encontrada durante a execução

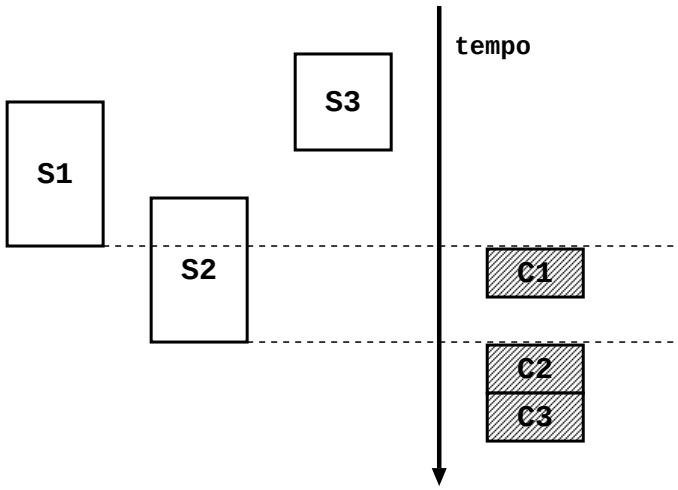


Figura 6.1: Exemplo de execução de instruções especulativas.

especulativa, e *(iii)* a estrutura de despacho da instância da instrução especulativa, contendo todos os operandos usados na sua execução. Ao operando enviado através dessa aresta contendo as estruturas citadas daremos o nome de "mensagem de *commit*".

As transações de uma instrução especulativa possuem um conjunto de escrita E_r para cada recurso r da máquina cujo estado pode ser alterado pela instrução `Commit` correspondente. Por outro lado, para cada recurso r cujo estado possar ser lido por uma transação de uma instrução especulativa, há um conjunto de leitura L_r associado a essa transação.

Os conjuntos de leitura e de escrita de uma instrução são compostos por tuplas $(e, v, t)_r$, onde e é o endereço do objeto no recurso r, v o seu valor (lido ou escrito) e t o seu tamanho. Um exemplo de objeto de um recurso seria uma palavra de memória. Nesse caso, e seria o endereço dessa palavra na memória da máquina, v o valor (lido no caso de ser um conjunto de leitura e escrito no caso de ser um conjunto de escrita) e t o tamanho da palavra de memória da arquitetura. Recursos como a tela (saída padrão) ou um arquivo aberto somente para escrita são exemplos de recursos que comportam somente conjuntos de escrita.

6.2 Preservação da semântica do programa

A maneira empregada no TALM para fazer com que a semântica de um programa seja preservada, mesmo com a execução especulativa e fora de ordem de parte de suas instruções, é adotar como regra que as instruções `Commit` sejam executadas respeitando a ordem total do programa. Dessa forma, mesmo que instruções sejam executadas fora de ordem, a visão que o programa tem do estado da máquina é alte-

rada somente em ordem. Essa ordenação das instruções `Commit` é garantida através da inserção de arestas *go-ahead*. Incluimos essas arestas para criar um caminho entre cada instrução `Commit` e a outra instrução `Commit` que, pela ordem total do programa sequencial, deve ser a subsequente. Pela regra de escalonamento *data-flow*, haver um caminho da instrução C_A para a C_B garante que C_B será executada somente depois de C_A.

Em cenários nos quais a instrução `Commit` subsequente só é definida em tempo de execução, devido aos diferentes caminhos que o fluxo do programa pode seguir, o subgrafo das instruções `Commit` deve refletir os diferentes caminhos de execução possíveis para as instruções especulativas correspondentes.

Considere o exemplo da Figura 6.2. Na ordem total do programa sequencial, após a instrução A podem vir as instruções B ou C. Após B ou C, deve vir a instrução D. Para que essa ordem seja respeitada na execução das instruções `Commit` correspondentes (CA, CB e CC), o subgrafo de instruções `Commit` deve receber uma instrução `Steer` utilizando a mesma variável booleana do grafo do programa que indica qual instrução (B ou C) será executada. Dessa forma, CB executará se e somente se B executar e o mesmo vale para CC e C.

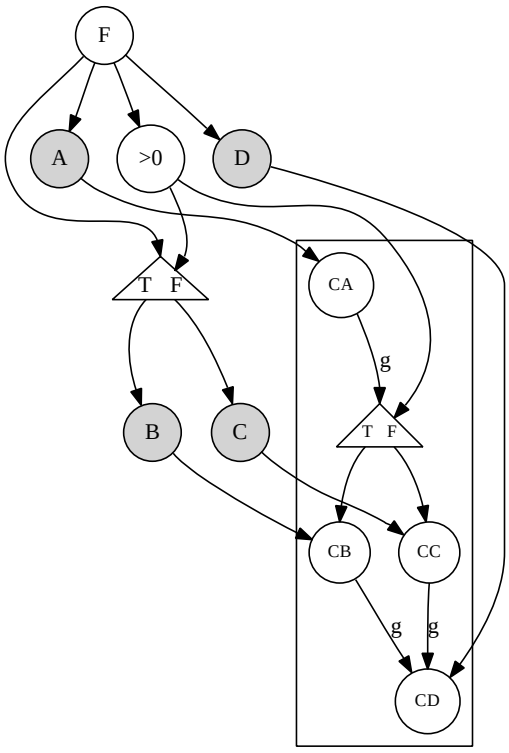


Figura 6.2: Exemplo de programa dataflow no qual as instruções especulativas a serem executadas só são determinadas em tempo de execução. As instruções especulativas estão sombreadas e arestas *go-ahead* marcadas com g.

Vamos formalizar essa condição através da seguinte regra:

Restrição 1. Seja $D(I, E)$ o grafo dataflow do programa com instruções especulativas e `Commit` correspondentes. E seja $H(I, E, X, \phi)$ o GOPC obtido através da aplicação do Algoritmo 5.1 a D. Para toda instrução especulativa $i \in I$ seja $c \in I$ a instrução `Commit` correspondente tal que $(i, c) \in E$, o OU lógico das expressões do conjunto $\{\phi(e) : e = (x, c), x \neq i, e \in E\}$ deve ser igual ao OU lógico das expressões do conjunto $\{\phi(e) : e = (\cdot, i), e \in E\}$.

6.3 Identificação de execuções especulativas

Conforme será visto na Seção 6.5, uma instância de uma instrução especulativa pode ter que ser reexecutada se um de seus conjuntos de leitura for considerado inválido pela sua instrução `Commit` correspondente. Neste caso, instâncias de instruções que recebem operandos daquela que foi reexecutada também precisarão ser reexecutadas, com os operandos produzidos pela nova execução. Dessa maneira, torna-se necessário identificar cada execução para que se saiba qual operando recebido é o mais recente.

As execuções das instâncias de instruções especulativas são identificadas pelo par $w = (x, id)$, onde x é o valor de um contador local da instrução que é incrementado a cada execução da mesma e id é um identificador único global da instrução. Repare que o contador poderia ser exclusivo de cada instância da instrução, sendo incrementado a cada reexecução da mesma instância, e nesse caso os rótulos de iteração e chamada seriam usados em w para distinguir as instâncias. Por simplicidade, adotamos no modelo o contador de execuções da instrução, comum a todas as suas instâncias.

Esse identificador (x, id) da instância da instrução especulativa é usado para marcar os operandos produzidos por ela. Instruções não especulativas podem receber no máximo um operando especulativo e os operandos que elas produzirem receberão a marcação do operando especulativo recebido. Essa restrição com relação ao recebimento de operandos especulativos por instruções não especulativas é formalizada a seguir:

Restrição 2. Seja $D(I, E)$ o grafo *dataflow* do programa com instruções especulativas e instruções `Commit` correspondentes. E seja $H(I, E, X, \phi)$ o GOPC obtido através da aplicação do Algoritmo 5.1 a D. Considere uma projeção completa $H\,|_\psi$ qualquer e o grafo $G = g(H\,|_\psi)$ obtido a partir dela. Para toda instrução não especulativa a deve haver no máximo uma instrução especulativa s tal que haja um caminho de s para a em G que não passa por outra instrução especulativa ou por uma instrução `Commit`.

Quando uma instrução recebe um operando especulativo op_i com marcação (c_i, id_i) e, posteriormente, recebe pela mesma porta de entrada outro operando especulativo op_j destinado à mesma instância da instrução e com marcação (c_j, id_j), o valor de c_i é comparado com o de c_j. Se $c_j > c_i$, op_j substitui op_i nos casamentos de operandos daquela instância da instrução, mas op_i continua armazenado para ser descartado futuramente pela coleta de lixo, que será vista na Seção 6.6. Caso contrário, op_j é descartado. A decisão de não descartar o operando antigo (op_i) imediatamente se deve ao fato da implementação Trebuchet utilizar ponteiros (da memória compartilhada) para fazer referência aos operandos nas mensagens de *commit*. Dessa forma, uma mensagem de *commit* que esteja sendo encaminhada poderia fazer referência a um operando que não existe mais na memória, se adotássemos esse descarte imediato.

Como o contador usado na marcação de operandos é local, essa comparação de c_i e c_j só tem sentido se a instrução que produziu op_i for a mesma que produziu op_j (ou seja, $id_i = id_j$). Assim, as diferentes execuções de uma instância de instrução podem receber um mesmo operando apenas de uma instrução especulativa. No caso do exemplo da Figura 6.3, como o booleano das instruções `Steer` não é especulativo, todas as (re)execuções de D receberão apenas o operando de A ou(exclusivo) de B, satisfazendo essa restrição.

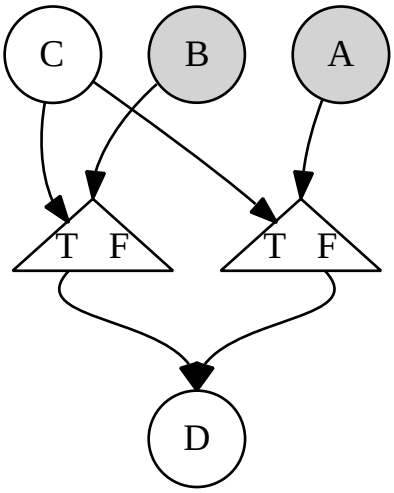


Figura 6.3: Exemplo no qual uma instrução pode receber o mesmo operando de duas instruções especulativas

6.4 Validação dos conjuntos de leitura

Quando uma instrução especulativa lê um objeto que não foi escrito pela transação corrente, uma tupla $l = (e, v, t)_r$ é acrescentada ao conjunto de leitura L_r da transação. Conforme dito anteriormente, esses conjuntos de leitura farão parte da mensagem enviada pela instância da instrução à instância da instrução `Commit` correspondente. Durante a execução da instrução `Commit` correspondente deve ser avaliado se o valor atual de cada objeto o_r é igual ao valor que foi lido durante a execução da instância da instrução especulativa.

Um conjunto de leitura é considerado válido se e somente se no momento da execução da instrução `Commit` o valor v de todos os seus elementos for igual ao valor atual dos objetos correspondentes. Isso quer dizer que, por exemplo, se para todos os elementos $(e, v, t)_m$ de um conjunto L_m de leitura da memória m o respectivo valor v for igual ao valor do objeto de tamanho t presente na posição e da memória m, esse conjunto de leitura é considerado válido.

6.5 Persistência dos conjuntos de escrita e reexecução

Considere uma instância de uma instrução especulativa $S(t)$ que após sua execução envia para a instrução `Commit` correspondente C os seus conjuntos de leitura e escrita, a estrutura de despacho utilizado na sua execução e uma mensagem contendo ou não exceções encontradas durante a execução. Caso a mensagem que $C(t)$ receber de $S(t)$ contenha conjuntos de leitura inválidos, a $C(t)$ não poderá persistir os dados presentes nos conjuntos de escrita. A instância da instrução especulativa deverá, então, ser reexecutada. Para comandar uma reexecução de $S(t)$ a estrutura de despacho recebida por $C(t)$ deve ser posta novamente na fila de instruções prontas do EP ao qual a instrução especulativa está associada. A cada execução (ou reexecução) o contador local da instrução é incrementado e, então, os operandos produzidos por uma nova reexecução terão em sua marcação um valor de contador incrementado.

Semanticamente, com relação ao estado da máquina, essa regra é equivalente a executar a instrução no momento em que a sua instrução `Commit` puder ser executada, garantindo, portanto, a ordem total do programa.

Outra condição que deve ser observada para que $C(t)$ possa persistir os conjuntos de escrita de $S(t)$ é se os operandos de entrada usados nessa execução de $S(t)$ já não são mais especulativos. Isto é, todo operando de entrada usado nessa execução de $S(t)$ deve ser resultado da última (re)execução da sua instância de instrução de origem.

No caso de operandos não especulativos, nenhuma verificação a mais deve ser

feita. Isso se deve ao fato de que instruções não especulativas jamais são reexecutadas se não recebem nenhum operando especulativo. Um operando não ter marcação de especulativo (e, portanto, ser não especulativo) indica que a instrução que o produziu não recebeu nenhum operando especulativo.

Por outro lado, para operandos especulativos devemos verificar se a sua marcação (x, id) é igual à marcação da última execução de uma instância da instrução cujo identificador é id. Isto é garantido se o marcador (x, id) também foi usado na mensagem de *commit* enviada para a instrução `Commit` e esta executou com sucesso ao receber essa mensagem, sem provocar reexecuções. Para essa verificação introduzimos as arestas de *wait* ao subgrafo das instruções de `Commit`. Através dessas arestas, uma instrução de `Commit` envia o marcador da mensagem de *commit* usada em uma execução bem sucedida para todas as instruções `Commit` de instruções especulativas que devem receber operandos com essa marcação. Assim, instruções `Commit` que recebem esse marcador por arestas *wait* só poderão executar com sucesso se um dos operandos contidos na mensagem de *commit* recebida tiver a mesma marcação. Isso indica que aquele operando está na cadeia de uma execução cujo *commit* foi bem sucedido.

Para que uma instrução receba mais de um operando especulativo, ela deve ser especulativa. Como mencionado anteriormente, os operandos que essa instrução produzir receberão as marcações de suas próprias execuções especulativas. A instrução `Commit` correspondente deve receber tantas marcações por arestas *wait* quantos forem os operandos especulativos que a instrução especulativa receberá.

Caso para cada marcação recebida por uma aresta de *wait* haja um operando na mensagem de *commit* com essa marcação, a instrução `Commit` executa com sucesso. Caso contrário, ela apenas ignora essa mensagem e espera uma reexecução.

No exemplo da Figura 6.4, $S1$, $S2$ e $S3$ são instruções especulativas e $C1$, $C2$ e $C3$ suas instruções `Commit`, respectivamente. Como $S3$ receberá um operando que está na cadeia causal de $S1$ e outro de $S2$, $C3$ só poderá executar com sucesso se $S3$ executar sem conflitos de dados e $S1$ e $S2$ não forem ser reexecutadas. Para isso, as arestas de *wait* $(C1, C3)$ e $(C2, C3)$ são incluídas, além das arestas de *go-ahead* (marcadas com g). $C1$ e $C2$ enviam a $C3$, pelas arestas de *wait*, a marcação da última execução de $S1$ e $S2$. Em sua execução, $C3$ deve constatar, através da mensagem de *commit*, que $S3$ foi executada usando operandos com essas marcações.

Se uma instrução puder receber mais de um operando com a mesma marcação, como no caso do exemplo da Figura 6.5, deverá haver uma aresta *wait* entre as instruções `Commit` para cada um desses operandos. A instrução `Commit` correspondente deverá verificar se o número de operandos com tal marcação na mensagem de *commit* é o mesmo que o número de arestas *wait* pelas quais ela recebeu essa marcação.

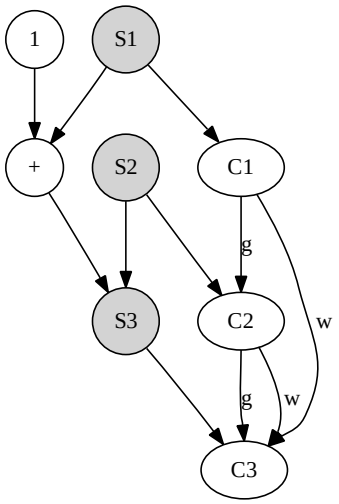


Figura 6.4: Exemplo de grafo *dataflow* com especulação.

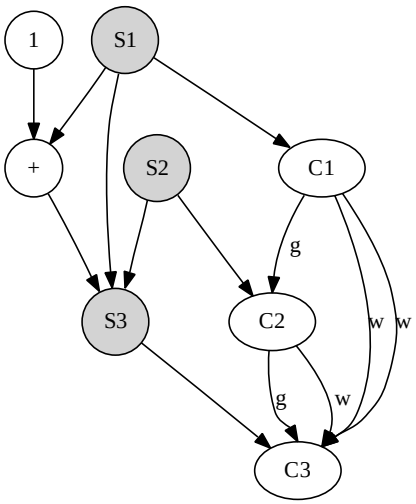


Figura 6.5: Exemplo no qual uma instrução recebe dois operandos especulativos com a mesma marcação. A instrução especulativa $S3$ recebe dois operandos com a marcação da execução de $S1$, portando $C3$ recebe duas arestas *wait* de $C1$.

Formalizamos a regra para inclusão de arestas de *wait* da seguinte forma:

Restrição 3. Seja $D(I, E)$ o grafo *dataflow* do programa com instruções especulativas e `Commit` correspondentes. E sejam H o GOPC obtido através da aplicação do Algoritmo 5.1 a D e $H\ |_\psi$ uma projeção completa desse GOPC. Considere duas instruções especulativas $a, b \in I$ e sejam $c_a, c_b \in I$ as suas instruções `Commit`, respectivamente. Seja $G(I, E_g) = g(H\ |_\psi)$ o grafo dirigido obtido a partir de $H\ |_\psi$ e $G'(I, E'_g = E_g \backslash \{(\cdot, b) \in E\})$ o grafo obtido a partir de G removendo-se as arestas que chegam em b. Para cada aresta $(x, b) \in E_g$ que, se adicionada a G' causa $P(I, \prec_\psi) = p(G''(I, E'_g \cup \{(x, b)\}), a \prec_\psi b$, sem que haja outra instrução c, especulativa ou `Commit`, tal que $a \prec_\psi c \prec_\psi b$, deve haver uma aresta *wait* de c_a para c_b em G. O número de arestas *wait* direcionadas para c_b deve ser o mínimo para satisfazer essa restrição para toda projeção completa $H\ |_\psi$.

6.6 Coleta de lixo

Estruturas de casamento que contêm operandos especulativos não podem ser removidas imediatamente após a execução da instrução, pois novas reexecuções da mesma instância da instrução podem ser disparadas, necessitando dos operandos contidos na estrutura. Além disso, operandos antigos que foram substituídos por operandos de novas reexecuções também não devem ser descartados imediatamente, conforme visto na Seção 6.3.

A solução adotada no *TALM* é empregar um modelo relaxado, não removendo as estruturas de casamento que contém os operandos até que se tenha certeza de que não ocorrerão mais execuções que os usarão. Os operandos produzidos por reexecuções vão sendo acumulados nas estruturas de casamento até que a coleta de lixo remova toda a estrutura junto com os operandos. Na Trebuchet isso foi feito com a alteração da estrutura de casamento apresentada na Figura 4.1, que ao invés de armazenar apenas um valor para cada operando, armazena uma lista encadeada de valores (produzidos por diferentes reexecuções) para cada operando.

As condições para a remoção das estruturas de casamento contendo operandos especulativos estão atreladas à execução bem sucedida das instruções de `Commit`. No caso de instruções especulativas, sabemos que não haverá mais reexecuções da mesma instância após uma execução bem sucedida da instância da `Commit` correspondente. Portanto, após a `Commit` executar com sucesso, a estrutura de casamento da instância da instrução a que ela corresponde pode ser removida.

No caso de instruções não-especulativas que recebem operandos especulativos, é preciso identificar uma instrução `Commit` que esteja na cadeia causal da instrução não-especulativa para que se possa determinar que não haverá mais reexecuções. É

o caso da instrução de incremento (+1) no grafo da Figura 6.6. Quando a instrução CA executar com sucesso, não teremos mais reexecuções de +1, mas ainda não saberemos se a última execução da instância de +1 terminou. Um indicativo de que essa última execução terminou é CB executar com sucesso, pois uma reexecução de +1 provocaria reexecução de B. Como podemos afirmar que B não reexecutará se CB executar com sucesso, essa condição é suficiente para verificar que +1 não reexecutará também.

Considere cenários como o da Figura 6.6, em que há sempre uma instrução especulativa na cadeia causal de instruções não-especulativas que recebem operandos especulativos. Podemos afirmar que, nesse tipo de cenário, operandos especulativos com marcação w podem ser removidos quando todas as instruções `Commit` que recebem o marcador w por arestas *wait* executarem com sucesso.

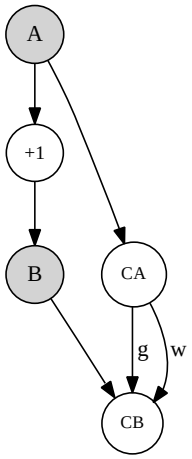


Figura 6.6: Cenário adequado ao esquema de coleta de lixo (operandos antigos) do *TALM*.

Podemos considerar duas implementações para determinar o número de mensagens com o marcador w que foram recebidas por instruções `Commit` que executaram com sucesso:

1. Um contador global é inicializado pela instrução `Commit` que envia as mensagens de *wait*. O valor inicial desse contador é o número de mensagens desse tipo enviadas. Cada instrução `Commit` que receber esse marcador por uma aresta *wait* e executar com sucesso decrementa o contador criado uma vez para cada aresta pela qual recebeu esse marcador. Se o contador chega a 0, o EP da instrução que o decrementou por último faz *broadcast* para todos os EPs de uma mensagem de coleta de lixo contendo esse marcador. Essa abordagem requer uma memória global compartilhada para os contadores.

2. De forma semelhante à abordagem anterior, a instrução que envia o marcador w por arestas *wait* inicializa um contador para esse marcador, só que em uma memória local do EP dela. O EP de cada instrução `Commit` que receber esse marcador e executar com sucesso deve enviar uma mensagem ao EP da instrução que enviou o marcador originalmente para que ela decremente o contador local. Quando o contador chega a 0, é feito o broadcast da mensagem de coleta de lixo avisando que estruturas de casamento contendo operandos com marcação w podem ser removidas.

Como nosso principal objetivo neste trabalho é utilizar a especulação para os acessos a uma memória compartilhada, optamos pela facilidade da primeira abordagem, já que haverá a memória compartilhada na arquitetura alvo.

A condição geral para a remoção de operandos descrita acima é a adotada pelo *TALM*. A restrição para que o grafo esteja adequado ao seu uso é formalizada da seguinte forma:

Restrição 4. Seja $D(I, E)$ o grafo *dataflow* do programa com instruções especulativas e `Commit` correspondentes. E seja $H(I, E, X, \phi)$ o GOPC obtido através da aplicação do Algoritmo 5.1 a D. Em qualquer projeção completa $H\ |_\psi$ não deve haver uma instrução não-especulativa $a \in I$ tal que em $G(I, E_g) = g(H\ |_\psi)$ não haja aresta saindo de a e haja uma instrução especulativa $b \in I$ tal que há caminho de b para a em G sem passar pela instrução `Commit` de b, c_b. Ou seja, dada o ordem parcial condicional $P(I, \prec_\psi) = p(G(I, E_g \backslash \{(b, c_b)\}))$ de G sem a aresta (b, c_b), a precedência $b \prec_\psi a$ não deve ser verdade.

6.7 Controle da especulação

Como todo o uso do mecanismo de especulação do *TALM* é descrito no grafo *dataflow*, o controle da profundidade da especulação também é feito no grafo do programa. Para construir uma barreira para a especulação, por exemplo, basta inserir dependências entre instruções especulativas e instruções `Commit` de outras instruções. No exemplo da Figura 6.7 a especulação da primeira etapa do programa deve ser persistida antes que o mesmo prossiga para a segunda. Essa restrição é garantida pela aresta saindo da última instrução `Commit` da primeira etapa para as instruções especulativas da segunda.

Na Figura 6.8 é mostrado como um esquema de janela deslizante pode ser utilizado para controlar a especulação. No exemplo, no grafo original (em **(a)**) as instâncias de A e de B podem começar sem que as instâncias das iterações anteriores tenham feito o *commit*. No grafo modificado (em **(b)**) é implementado um esquema de janela deslizante no qual o *commit* da iteração N libera a execução

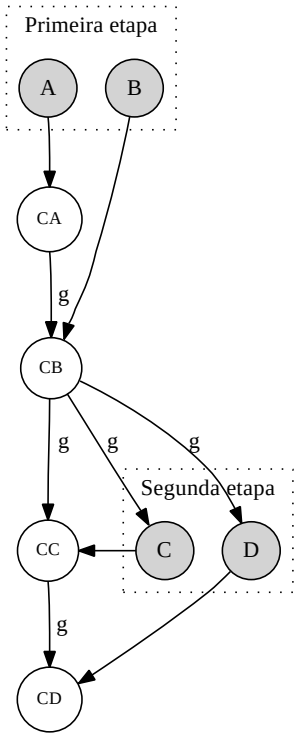


Figura 6.7: Exemplo de barreira para a especulação.

especulativa da iteração $N + W$, sendo $W = 2$ o tamanho da janela no exemplo. Para que a iteração N libere a $N + W$ é utilizada a variante da instrução `inctag` com operando imediato igual ao tamanho da janela (representada por $IT + 2$ na Figura 6.8). Essa instrução incrementa o rótulo de iteração do operando de entrada com o valor do seu operando imediato. No exemplo, as instruções especulativas da iteração N são dependentes da última instrução `Commit` da iteração $N - 2$.

Repare que é inserida ainda uma instrução `inctag` a mais, responsável por inicializar a primeira iteração, enquanto que a variante com imediato inicializa a segunda iteração. Note também que foi necessário implementar uma camada a mais de condições com as instruções `steer`, já que as duas últimas iterações produzirão operandos que seriam encaminhados, mas nunca consumidos.

6.8 Especulação na Trebuchet

Na atual implementação da Trebuchet o único recurso que pode ser acessado para escrita ou leitura especulativamente é a memória. Para fazer esses acessos foi implementada uma memória transacional de *software* (STM) cujo funcionamento é semelhante às STMs convencionais, com a diferença que a etapa de *commit* é feita pelas instruções `Commit` correspondentes. Essa diferença impactou a implementação

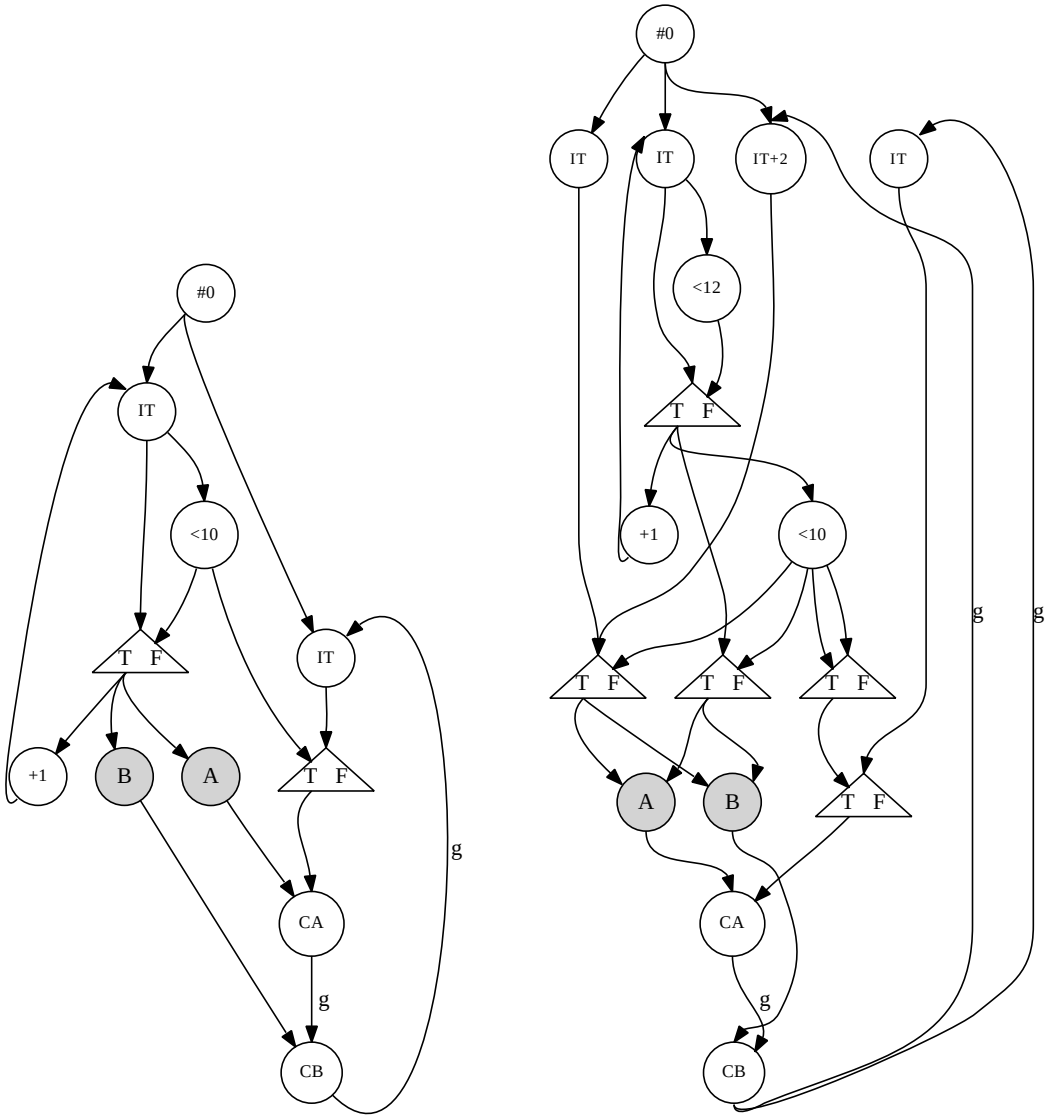


Figura 6.8: Exemplo de implementação de janela deslizante para a especulação.

da STM da Trebuchet em dois pontos principais:

- Como a etapa de persistência dos conjuntos de escrita é feita pelas instruções `Commit`, que são executadas em ordem, a STM da Trebuchet não precisa de locks para garantir atomicidade nas escritas.
- A transação, que é composta pelos conjuntos de leitura e escrita, de uma execução de instrução especulativa deve ser enviada para o EP para o qual a instrução `Commit` correspondente foi mapeada.

Ao terminar uma execução a instrução especulativa envia a mensagem de *commit* para sua instrução `Commit`, conforme descrito na Seção 6.1. Ao receber uma mensagem de *commit* e um token *go-ahead* a instrução `Commit` executará, verificando se o conjunto de leitura recebido na mensagem é válido e se os operandos recebidos na mensagem não são mais especulativos, exatamente como discutido na Seção 6.5.

Para a construção dos conjuntos de leitura e de escrita, foram implementadas as funções `LOAD()` e `STORE()` e suas variantes para os diferentes tipos da linguagem *C*. Essas funções devem ser usadas na implementação de instruções especulativas, pois substituem acessos diretos à memória por acessos à memória transacional da Trebuchet. Os conjuntos de leitura e de escrita são implementados como tabelas *hash* cujas chaves são os endereços dos acessos. Por usarmos endereços de memória como chave, a função *hash* utilizada faz o deslocamento à direita da chave no tamanho do alinhamento da função `malloc` da arquitetura, para eliminar os zeros nos bits menos significativos.

A função `LOAD()`, que recebe como parâmetro o endereço a ser lido, primeiramente busca o endereço no conjunto de escrita da transação, pois se já houve uma escrita nesse endereço o valor retornado deve ser o dela. Caso não tenha havido escrita nesse endereço, a função lê o valor diretamente da memória e inclue o valor lido na tabela *hash* correspondente ao conjunto de leitura.

A função `STORE` recebe como parâmetros o endereço no qual deve ser feita a escrita e o valor a ser escrito. Em seguida ela busca o endereço no qual deve escrever no conjunto de escrita. Se já tiver havido escrita nesse endereço, o valor encontrado no conjunto de escrita deve ser sobreescrito. Caso contrário, é feita uma nova inserção no conjunto de escrita com o valor.

Como padrão, espera-se que serão feitas leituras e escritas de apenas um tipo de dado em um espaço de endereçamento. Consideramos essa restrição bastante razoável já que normalmente as estruturas de dados das aplicações recebem apenas um tipo. Por essa razão, as buscas nas tabelas *hash* para leitura e escrita procuram apenas pelo endereço passado. Observe que essa abordagem não funciona se o programa fizer escritas sobrepostas, isto é, desrespeitando o alinhamento de memória ou escrevendo dados de diferentes tamanhos em um mesmo espaço.

Para dar suporte a esse tipo de escrita, a STM da Trebuchet deverá suportar escritas sobrepostas, que fará com que múltiplas buscas sejam feitas para cada leitura ou escrita. Com esse suporte habilitado, ao invés de buscar apenas por escritas feitas no endereço passado como parâmetro, as funções `LOAD()` e `STORE()` buscarão também por escritas feitas nas posições de memória vizinhas que poderiam se sobrepor à operação atual. Uma escrita a é considerada sobreposta com uma b se $e_a < e_b < e_a + t_a$ ou $e_b < e_a < e_b + t_b$, onde e_i é o endereço da escrita i e t_i o tamanho do objeto escrito. Uma nova escrita deve sobrescrever todas as sobrepostas, na intercessão entre a escrita atual e a que está sendo sobreposta.

6.8.1 Prova de serialização estrita

Para mostrar a corretude do nosso modelo escolhemos o critério de serialização estrita das operações. A grosso modo, uma execução é estritamente serializável se for equivalente a uma execução sequencial que preserva a ordem dos *commits*. Nesta seção apresentamos uma definição para serialização estrita baseada em [20] e estendida para o modelo *dataflow* e em seguida provamos que o nosso modelo de execução especulativa se adequa a esse critério.

Sejam $E_i = \{\triangleleft_i, \triangleright_i, \not\triangleright_i, R_i(o,v), W_i(o,v), cons_i(m), prod_i(m)\}$ os eventos modelados que podem ocorrer em uma execução i de uma instrução especulativa e de sua instrução `Commit`, onde:

- $\triangleleft_i$ = abertura da transação.
- $\triangleright_i$ = fechamento da transação com sucesso (*commit*).
- $\not\triangleright_i$ = cancelamento da transação, quando alguma das condições para o *commit* não são respeitadas no momento em que a instrução `Commit` executa.
- $R_i(o,v)$ = leitura do objeto o (do recurso ao qual pertence), obtendo o valor v como resposta.
- $W_i(o,v)$ = escrita do valor v no objeto o (no seu recurso).
- $cons_i(m)$ = consumo de um operando especulativo com marcador m para a execução.
- $prod_i(m)$ = produção de um operando especulativo com marcador m.

Os atributos entre parênteses dos eventos serão omitidos quando não forem necessários ao contexto. Por simplicidade, mas sem perda de generalidade, consideramos que pode haver apenas um evento $R_i(o,\cdot)$ e um $W_i(o,\cdot)$ para cada execução i. No caso das escritas em um mesmo objeto consideramos apenas a última. Nos referiremos simplesmente por *execução especulativa* para denotar os eventos da execução de

uma instrução especulativa e em seguida da sua instrução `Commit` correspondente. Quando for conveniente, usaremos o termo *transação* para fazer referência a uma execução especulativa.

Chamamos de *histórico* uma sequência σ de eventos correspondentes a execuções de instruções especulativas. Um histórico σ é chamado de sequencial se os eventos estiverem agrupados por transação, sendo $\triangleleft_i$ o primeiro evento de cada transação i e $\triangleright_i$ ou $\not\triangleright_i$ o último. Isto é, σ satisfaz a expressão regular $(\triangleleft_i(R_i + W_i + cons_i + prod_i)^*(\triangleright_i + \not\triangleright_i))^*$. Note que um histórico ser sequencial é o equivalente a todas as execuções terem sido feitas atomicamente.

Um histórico σ é **consistente** se, removendo os eventos de transações abortadas, obtemos um histórico no qual (considerando $\prec$ o operando de ordem parcial em σ):

1. Se houver $W_j(o, v)$, $R_i(o, u)$, $\triangleright_i$ e $\triangleright_j$ tais que $\triangleright_j \prec \triangleright_i$ e não houver $W_k(o, x)$ e $\triangleright_k$ tais que $\triangleright_j \prec \triangleright_k \prec \triangleright_i$, temos $v = u$.

2. Se houver $cons_i(m)$ há $prod_j(m)$ tal que $prod_j(m) \prec cons_i(m)$, isto é, o operando com marcação m foi produzido por uma execução que não foi cancelada, já que estamos desconsiderando execuções desse tipo, e que foi persistida antes de i.

Um histórico é estritamente serializável se removendo os eventos de transações abortadas e reordenando os restantes pela ordem dos eventos de *commit* ($\triangleright$) obtivermos um histórico sequencial consistente. A seguir mostraremos que todo histórico de eventos produzido pelo nosso modelo de execução especulativa é estritamente serializável.

Teorema 1. *Os históricos produzidos por execuções do modelo especulativo do TALM são consistentes.*

Demonstração. Chamamos de $s(\sigma)$ o histórico obtido removendo-se os eventos de transações abortadas de σ. Para provar a condição (1) de consistência para todo σ produzido pelo modelo basta supor que:

- Há em $s(\sigma)$ $W_j(o, v)$, $R_i(o, u)$, $\triangleright_i$ e $\triangleright_j$ tais que $\triangleright_j \prec \triangleright_i$ e que não há $W_k(o, x)$ e $\triangleright_k$ tais que $\triangleright_j \prec \triangleright_k \prec \triangleright_i$.

- $u \neq v$.

Temos $\forall t R_t, W_t \prec \triangleright_t$, decorrente do fato de que as instruções `Commit` só executam depois das suas instruções especulativas correspondentes. Assim, conforme descrito na Seção 6.5, em $\triangleright_j$ a instrução `Commit` responsável pela execução j persiste os conjuntos de escrita da execução, portanto atribuindo o valor v ao objeto o. No evento $\triangleright_i$, no entanto, a instrução `Commit` da execução i tentará validar o valor u

de o encontrado no conjunto de leitura. Como j é a última execução antes de i a escrever em o, o valor para o encontrado no recurso será v e o conjunto de leitura será invalidado, o que fará com que a transação seja abortada. Obtemos, assim, uma contradição já que por definição $s(\sigma)$ não tem eventos de transações abortadas.

Pela Restrição 3, para cada operando especulativo que uma instrução especulativa receber, a instrução `Commit` correspondente deverá receber o marcador desse operando por uma aresta *wait*, se o operando tiver sido produzido por uma execução que foi persistida. Sejam A uma instrução especulativa, C_A sua instrução `Commit` e i uma execução de A e C_A. Suponha que haja $cons_i(m)$ em $s(\sigma)$ e que m é a marcação dada por uma execução cancelada. Nesse caso, m não foi recebida por nenhuma aresta de *wait* e então pelo menos uma das marcações recebidas por essas arestas não fez casamento com a marcação de nenhum operando, logo C_A teria cancelado i. Como em $s(\sigma)$ não há eventos de execuções canceladas, temos então que m vem de uma execução que persistiu. Já que m é uma marcação dada a um operando por uma execução que persistiu, deve haver $prod_j(m)$ tal que $prod_j(m) \prec cons_i(m)$ em $s(\sigma)$. Logo, a segunda condição para consistência também é satisfeita e o histórico é consistente.

□

Teorema 2. *Se σ é um histórico consistente, então o histórico obtido a partir da reordenação dos eventos de $s(\sigma)$ de acordo com a ordem dos* commits *também é consistente.*

Demonstração. A consistência de $s(\sigma)$ vem trivialmente do fato de as condições para consistência descartarem execuções canceladas. Agora considere σ' o histórico obtido a partir da reordenação dos eventos de $s(\sigma)$ de acordo com a ordem dos *commits*. Como a ordem dos *commits* em σ' é a mesma que em σ, σ' respeita a primeira condição para consistência. Além disso, como $\triangleright_j \prec \triangleright_i \Rightarrow prod_j(m) \prec cons_i(m)$, pois caso contrário haveria *deadlock*, a ordem desses eventos também se mantém e então a segunda condição também é respeitada.

□

Temos então, pelo Teorema 1 e pelo Teorema 2, que todo histórico produzido pelo modelo especulativo do TALM é estritamente serializável.

Capítulo 7

Experimentos e Resultados

Os experimentos para medir o desempenho da Trebuchet foram divididos em duas etapas. Na primeira etapa optamos por demonstrar que, mesmo com os custos intrínsecos de interpretação de uma máquina virtual, podemos obter resultados competitivos. Para isso selecionamos aplicações que poderiam ser facilmente paralelizadas usando a biblioteca *OpenMP* e desenvolvemos versões delas usando a linguagem TALM, de maneira semelhante ao exemplificado na Figura 4.2. As características principais que margearam a escolha das aplicações usadas nessa etapa foram a regularidade do paralelismo, isto é, uma vez paralelizada a aplicação tem cargas de trabalho balanceadas, e a independência entre iterações dos laços.

Paralelizar aplicações com essas características é uma tarefa simples, se resumindo à inclusão de anotações nos laços, no caso da versão *OpenMP*, ou à definição de super-instruções que executarão partes do laço, no caso da versão TALM. Dessa forma, como desenvolver versões que exploram o paralelismo dessas aplicações em ambas as plataformas é fácil, a comparação do desempenho obtido pelas duas versões deverá mostrar apenas os custos computacionais de cada plataforma e não vantagens de um modelo de programação sobre o outro.

A segunda etapa dos nossos experimentos busca mostrar o potencial do modelo especulativo desenvolvido bem como apontar os custos da implementação atual. Selecionamos duas aplicações para isso: um *benchmark* sintético altamente parametrizado desenvolvido com o objetivo de avaliar os diversos custos (*rollback*, custos da memória transacional etc) da implementação da Trebuchet e uma aplicação real de mineração de dados. Em ambas as aplicações podemos realizar experimentos com diferentes cenários, aumentando ou diminuindo a pressão em cada um dos possíveis gargalos da máquina virtual.

Para os experimentos da primeira etapa foi utilizada uma máquina Intel® Core™i7 (2,66 GHz), com quatro núcleos de processamento com *Hyper Threading™* e 6GB de memória DDR-3 1600 MHz *Triple Channel* (3x2GB).

Para os experimentos com especulação foi utilizada uma máquina com quatro

chips AMD Six-Core Opteron^{TM}i7 (2100 MHz) (totalizando 24 núcleos) e 64 GB de RAM DDR-2 667MHz (16x4GB). Em razão do limitado tempo de uso dessa máquina que nos foi concedido, optamos por executar apenas os experimentos de especulação nela para testarmos a escalabilidade do modelo de execução especulativa. Outra limitação no uso dessa máquina foi no número de núcleos de processamento usados. Dos 24 núcleos, apenas 18 nos foram reservados, portanto nossos experimentos foram até 18 *threads*.

7.1 Avaliação dos custos de interpretação da Trebuchet

Nesta seção serão descritos os experimentos feitos para mostrar a performance da Trebuchet em comparação com a biblioteca *OpenMP*.

7.1.1 Aplicações

Foram selecionadas três aplicações para serem paralelizadas nesta etapa: um traçado de raios, cálculo recursivo de determinantes e multiplicação de matrizes. Essas aplicações foram escolhidas por apresentarem as características de paralelismo mencionadas acima.

A aplicação de traçado de raios faz a renderização de uma cena composta por esferas cujas posições e cores são lidas de um arquivo. A renderização é feita em dois laços aninhados que correspondem à varredura de linhas e colunas da imagem. A versão *OpenMP* é implementada através da inclusão de anotações que definem o laço mais externo da renderização como paralelo, dividindo o trabalho entre *threads*.

Para a versão Trebuchet foram definidas super-instruções para as operações de leitura e escrita dos arquivos de entrada e de saída. Em seguida, a região do programa a ser paralelizada foi implementada em uma super-instrução que executa uma parcela do laço de renderização. Foram criadas duas versões dessa super-instrução de renderização, uma em granularidade fina, que executa apenas o laço interno, e uma em granularidade grossa, que abrange também o laço externo. No caso da granularidade fina, o laço externo é descrito com instruções simples no grafo *dataflow* e, portanto, a implementação que usa essa versão deverá incorrer em mais custos de interpretação. Por outro lado, a versão de granularidade grossa pode apresentar custos de predição incorreta de desvio, devido aos laços aninhados. Na versão de granularidade fina os custos de predição incorreta de desvio são menores, pois o controle do laço externo é feito no grafo *dataflow*.

Na aplicação de cálculo recursivo de determinante de matrizes, o laço que dispara o primeiro nível da recursão é implementado em uma super-instrução, que é repli-

cada para dividir o trabalho. A aplicação foi executada para matrizes de diferentes tamanhos (ordens 8, 9, 10, 11, 12 e 13) para permitir a avaliação da quantidade de computação necessária para compensar os custos inerentes à paralelização, tanto para a versão *dataflow* quanto para a versão *OpenMP*.

No programa de multiplicação de matrizes foi definida uma super-instrução cujas múltiplas instâncias paralelizam o laço externo que varre as matrizes fazendo a multiplicação. Cada instância fica responsável pela produção de algumas linhas na matriz resultante.

7.1.2 Resultados

Para o traçado de esferas os resultados (Figura 7.1) mostram acelerações da versão Trebuchet de até $4,81$ em relação ao programa sequencial e de até $1,11$ quando comparada com a versão *OpenMP*. Na figura, a versão de granularidade fina é apresentada como DF-F e a de granularidade grossa é apresentada como DF-C. A comparação entre as duas versões mostra que os custos de interpretação do laço externo na versão de grão fino não foram tão significativos, podendo também ter sido amenizados pela diminuição de predições de desvios incorretas.

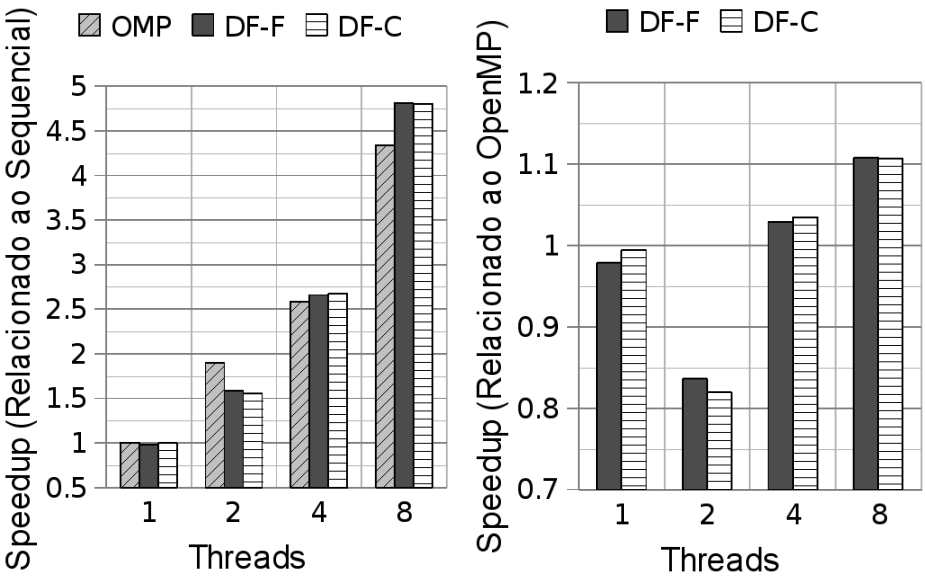


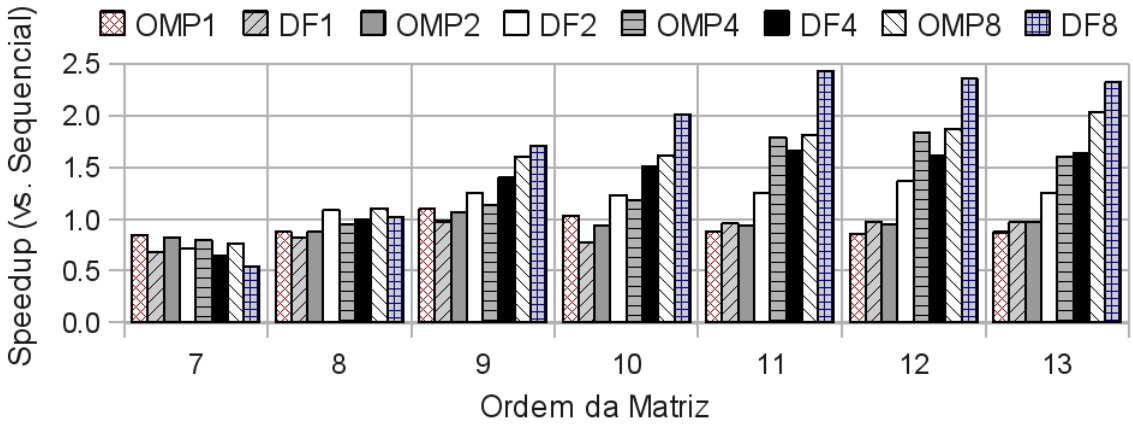
Figura 7.1: Resultados: traçado de raios.

Na Figura 7.2 são apresentados os resultados da aplicação de cálculo de determinante de matrizes para matrizes de diferentes ordens. As barras OMPi representam a versão *OpenMP* com i *threads* e as barras DFi representam a versão Trebuchet com i EPs. Na Figura 7.2a temos a comparação das acelerações de ambas as versões obtidas em relação ao programa sequencial e na Figura 7.2b a aceleração de cada

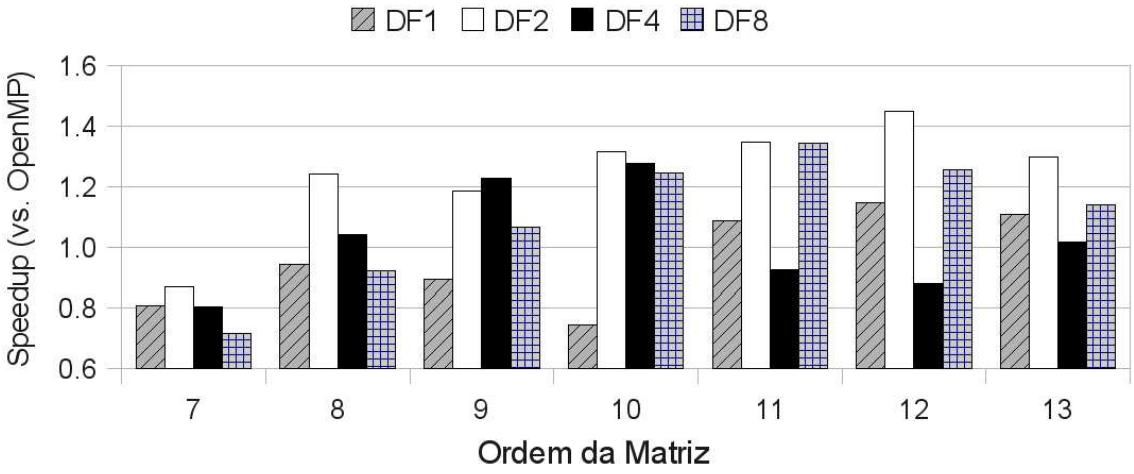
versão DFi em relação à versão OMPi correspondente.

Observamos que nos cenários com maior quantidade de computação (matrizes de ordem maior) a implementação *dataflow* apresenta ganhos maiores que a versão *OpenMP*. Isso indica um maior potencial de escalabilidade da implementação Trebuchet. Por outro lado, para os cenários com menos computação, a Trebuchet apresenta performance pior que a biblioteca *OpenMP*. Esse resultado reflete os custos fixos da execução da aplicação na Trebuchet, como carga do binário na memória e distribuição das instruções entre os EPs, por exemplo. Esses custos devem ser amortizados com a computação a ser paralelizada.

Foram obtidas acelerações de até $2,4$ em relação à versão sequencial e de até $1,3$ em relação à implementação *OpenMP*.



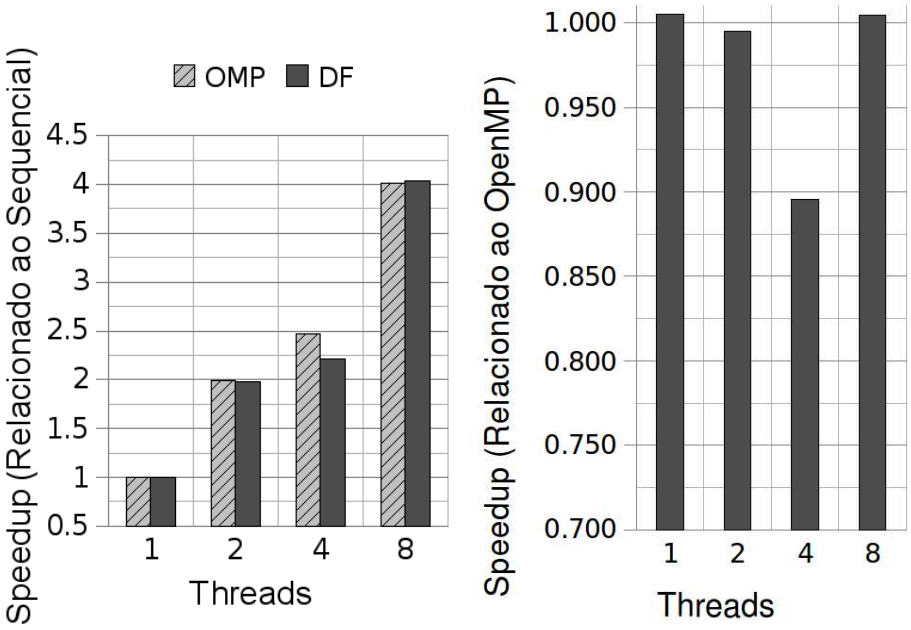
(a)



(b)

Figura 7.2: Resultados: Determinante.

A Figura 7.3 mostra os resultados obtidos para a multiplicação de matrizes. Esses resultados apresentam uma performance equivalente de ambas as versões, com acelerações de até $4,03$ em relação à versão sequencial.



(a) (b)

Figura 7.3: Resultados: Multiplicação de matrizes.

7.2 Experimentos com especulação

Para demonstrar o potencial do modelo especulativo desenvolvido foram implementadas duas aplicações, sendo uma delas um *benchmark* sintético, empregado para medir os diversos custos da especulação na Trebuchet, e uma aplicação real de mineração de dados.

7.2.1 Aplicações

A primeira aplicação, a qual chamamos de *bank* simula o servidor de um banco que recebe como entrada ordens de transferências de valores entre contas bancárias. Uma ordem de transferência é composta por uma conta de origem, uma conta de destino e um valor. Se a conta origem tiver saldo suficiente, o valor é transferido para a conta destino, caso contrário a operação é cancelada e um contador de número de operações canceladas deve ser incrementado.

A implementação dessa aplicação é composta por uma etapa de leitura e uma de processamento, que são executadas dentro de um laço. A cada iteração do laço a primeira etapa simula a leitura de um bloco de transferências que será processado pela etapa seguinte. Para os nossos experimentos a etapa de leitura é simulada por uma função de geração de números pseudo-aleatórios utilizando uma semente constante. Na etapa de processamento, a cada transferência é simulada uma carga de processamento, implementada na forma de uma decomposição LU de uma matriz cuja

dimensão varia em uma faixa determinada por um parâmetro da aplicação. Note que, as transferências lidas devem ser processadas em ordem pela etapa de processamento, já que se uma transferência esgotar o saldo de uma conta as transferências seguintes que tentarem retirar valores dessa mesma conta devem ser canceladas.

Usamos a especulação para paralelizar a etapa de processamento. O bloco de transferências lidas é dividido em blocos menores que são processados especulativamente por diferentes EPs em paralelo. Conflitos ocorrem se durante uma mesma iteração do laço principal dois EPs diferentes fizerem uma transferência usando uma mesma conta. Nesse caso, o EP que processou a transferência que deve vir depois deve fazer o *rollback* da super-instrução de processamento.

A profundidade da especulação, isto é, o número de iterações que podem executar especulativamente por vez, é controlada por um esquema de janela deslizante semelhante ao apresentado na Figura 6.8. Dessa forma, vemos que essa aplicação é altamente parametrizada, oferecendo as seguintes variáveis a serem ajustadas e seus respectivos impactos em custos da Trebuchet:

- Dimensão da matriz da decomposição LU: regula a quantidade de computação por acesso à memória transacional. Quanto menor a quantidade de computação, maior é a representatividade dos custos de acesso à STM no tempo total de execução.

- Número de contas no banco: Quanto menor o número de contas, maior a possibilidade de duas transferências executadas em paralelo operarem sobre uma mesma conta, acarretando conflitos. Assim, esse parâmetro regula o número de *rollbacks*.

- Tamanho da janela deslizante: Esse parâmetro regula quantas iterações podem executar especulativamente por vez. Uma janela muito grande provoca muitos conflitos e uma janela muito pequena explora menos paralelismo, devido à constante necessidade de sincronização para que a execução prossiga.

- Tamanho do bloco de transferências processadas a cada iteração: Quanto mais transferências são processadas especulativamente por vez, maior o número de endereços armazenados nas tabelas *hash* das transações. Com mais endereços, a probabilidade de colisão nas tabelas fica maior e, portanto, o custo de acesso à STM aumenta.

- Número de transferências: Buscamos encontrar o número mínimo de transferências para o qual o uso da Trebuchet oferece ganho de performance.

Para esse experimento, implementamos também uma versão dessa aplicação que recebe como entrada um cenário ideal, sem possibilidade de conflitos, ao invés de

uma entrada aleatória. Essa versão, portanto, não necessita do suporte à especulação e serviu como base de comparação (quão próximo chegamos do paralelismo ideal).

A segunda aplicação é uma implementação paralela do algoritmo k-medoids. O algoritmo k-medoids é um método de *clusterização* similar ao k-means [21]. Ambos os algoritmos particionam um conjunto de pontos em *clusters*, procurando a cada iteração minimizar a soma das distâncias entre os pontos e os centros dos *clusters* aos quais pertencem. A principal diferença entre ambos algoritmos é que no k-medoids os centros dos clusters são pontos pertencentes ao conjunto.

A computação desse algoritmo é dividida em duas etapas principais. Na primeira etapa cada ponto é associado ao *cluster* cujo centro é o mais próximo dele. Na segunda etapa o algoritmo seleciona novos centros para cada *cluster*. O novo centro escolhido para um *cluster* é o ponto daquele *cluster* cuja soma das distâncias a todos os outros pontos do *cluster* for a mínima.

A primeira etapa é paralelizada no número de pontos sem a necessidade do suporte à especulação. Nós usamos a especulação para paralelizar a segunda etapa, também no número de pontos, já que atualizações feitas em paralelo em um mesmo *cluster* podem causar conflitos. A escolha por paralelizar essa etapa também no número de pontos se deve ao fato de que paralelizar no número de *clusters* pode resultar em desbalanceamento de carga, tendo em vista que o número de pontos em um cada *cluster* é heterogêneo.

Nós implementamos duas versões *OpenMP* da aplicação para comparar com a nossa implementação *dataflow* especulativa. Uma versão usa *locks* de granularidade fina para a manipulação dos vetores de cada *cluster*. A segunda adiciona uma camada de sincronização após cada *thread* computar separadamente candidatos para centros de seus respectivos *clusters*. Essa nova camada de sincronização seleciona o melhor candidato para centro de cada *cluster* entre os selecionados por cada *thread*.

7.2.2 Resultados

Para a aplicação *bank* executamos uma série de experimentos, cada um representando um cenário no qual varia um dos parâmetros apresentados na seção anterior. Para cada cenário comparamos o desempenho obtido com o desempenho ideal (da aplicação sem conflitos na entrada). Apresentamos também o número de *rollbacks* que ocorrem com as variações. Nas figuras *Real-n* representa os resultados para a versão com especulação com n EPs enquanto que *Ideal-n* representa os resultados do caso ideal para n EPs.

Para todos os experimentos, conforme o número de EPs utilizados aumenta, mais transferências são processadas em paralelo e, portanto, maior o número de *rollbacks*. Esse aumento no número de *rollbacks* é o motivo pelo qual a versão especulativa não

escala tão bem quanto a ideal.

A dimensão da matriz de redução LU regula a quantidade de computação feita para cada acesso à memória transacional. Conforme dito, quanto menor a computação por acesso, mais representativos são os custos da STM na execução total. Esse comportamento é observado na Figura 7.4. Vemos que quanto menor a dimensão da matriz, maior a diferença entre a aceleração obtida com a especulação e a ideal, mesmo o número de *rollbacks* variando pouco.

Na Figura 7.5 vemos a influência da variação do número de contas bancárias na aceleração obtida. Vemos a rápida diminuição no número de *rollbacks* com o aumento do número de contas, já que havendo um universo de contas maiores a possibilidade de conflito entre duas transferências é menor.

No cenário de entrada ideal, para que seja impossível haver conflitos não há duas operações de transferência sobre uma mesma conta, portanto o número de contas disponíveis deve ser no mínimo o dobro do número de transferências. Por esse motivo, no gráfico da Figura 7.5 para números pequenos de contas os resultados ideias não são apresentados.

As figuras 7.6, 7.7 e 7.8 mostram os resultados para a variação do tamanho da janela de especulação, do tamanho da transação e do número total de operações de transferência feitas.

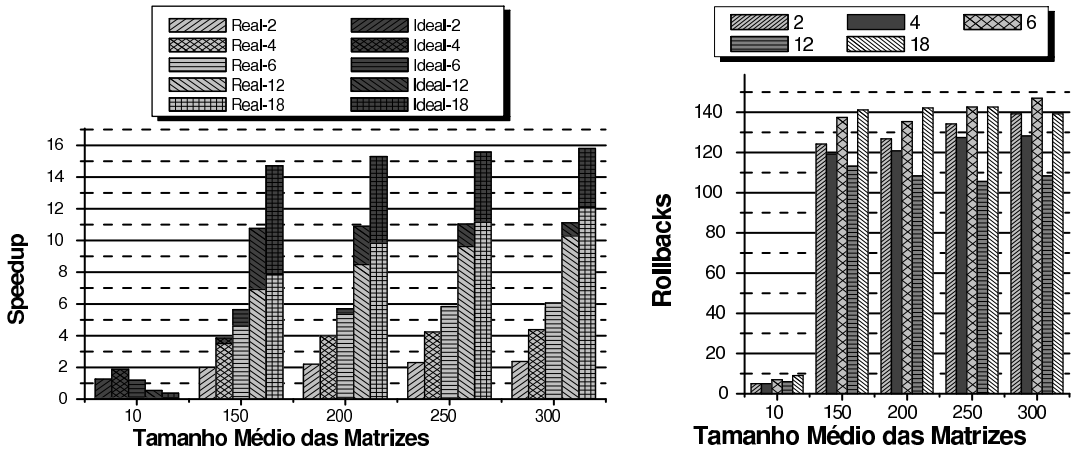


Figura 7.4: Bank - Variação da carga de computação por acesso à STM.

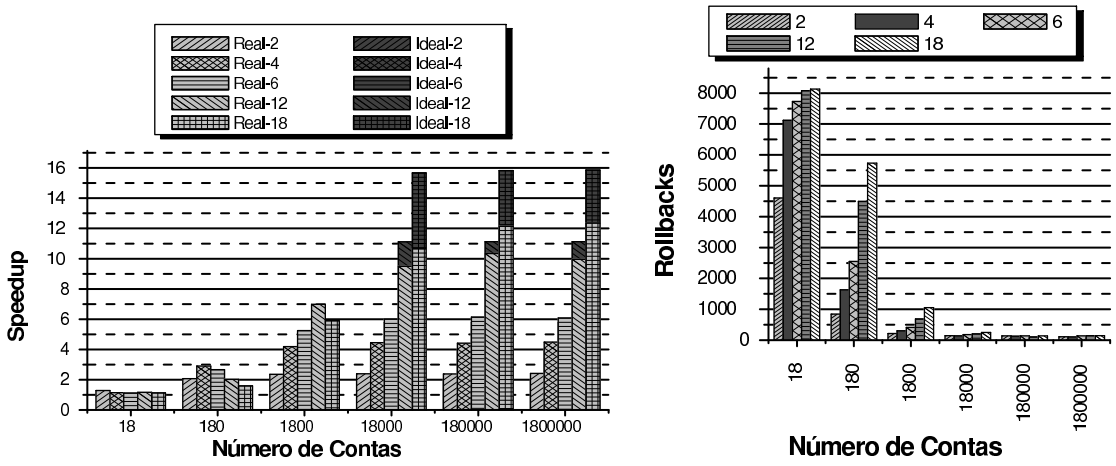


Figura 7.5: Bank - Variação do número de contas

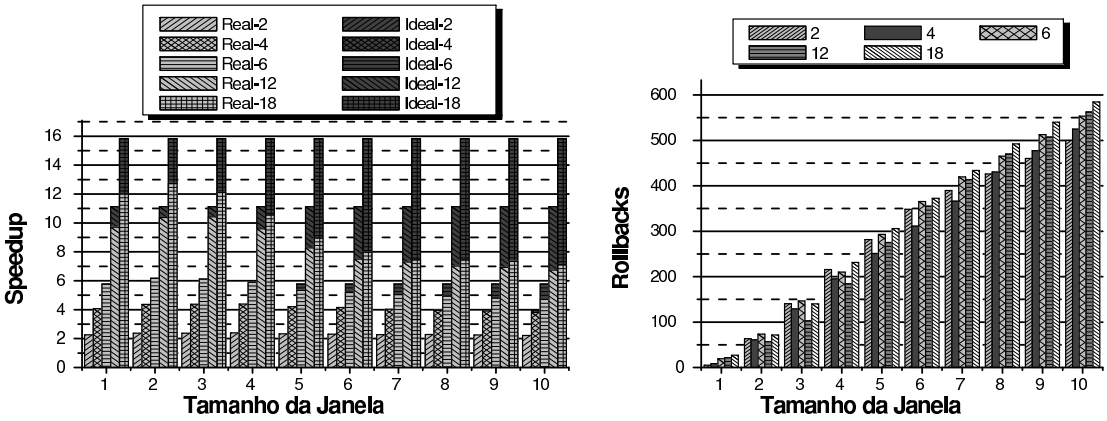


Figura 7.6: Bank - Variação do tamanho da janela

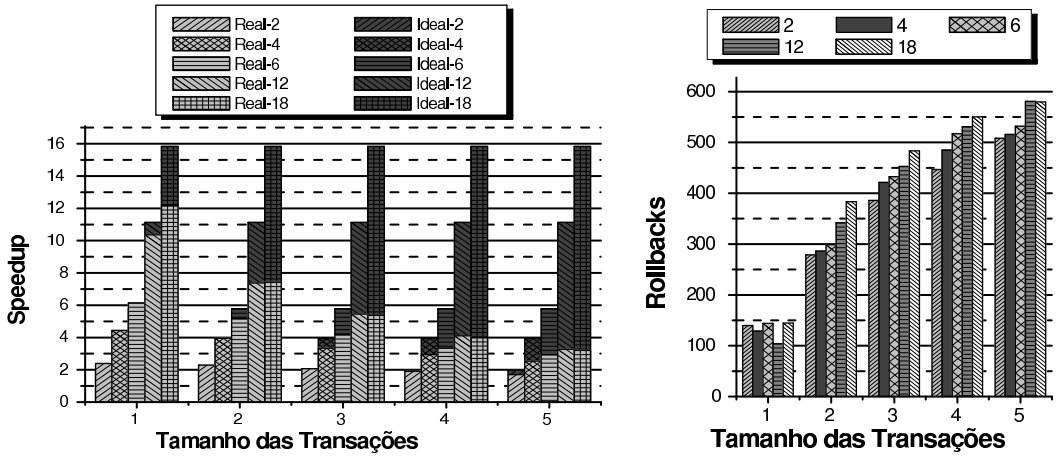


Figura 7.7: Bank - Variação do tamanho das transações.

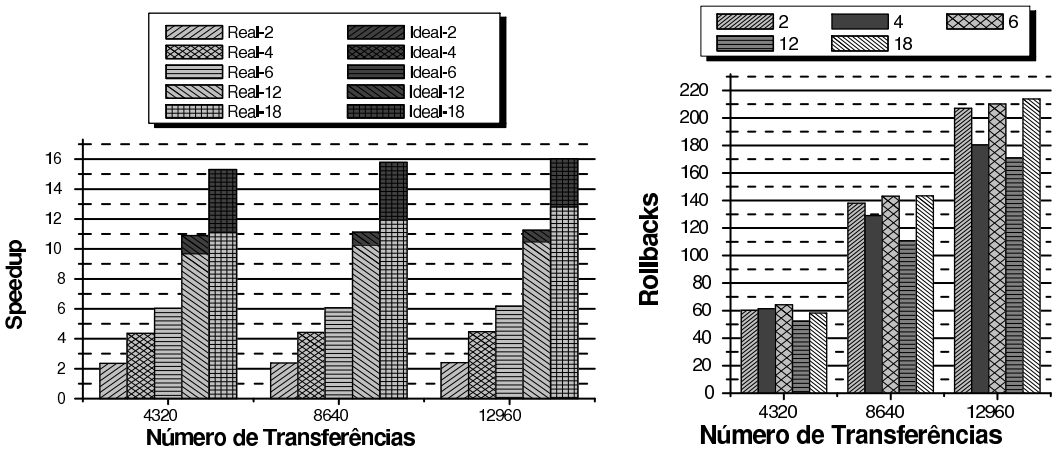


Figura 7.8: Bank - Variação do número de operações de transferência.

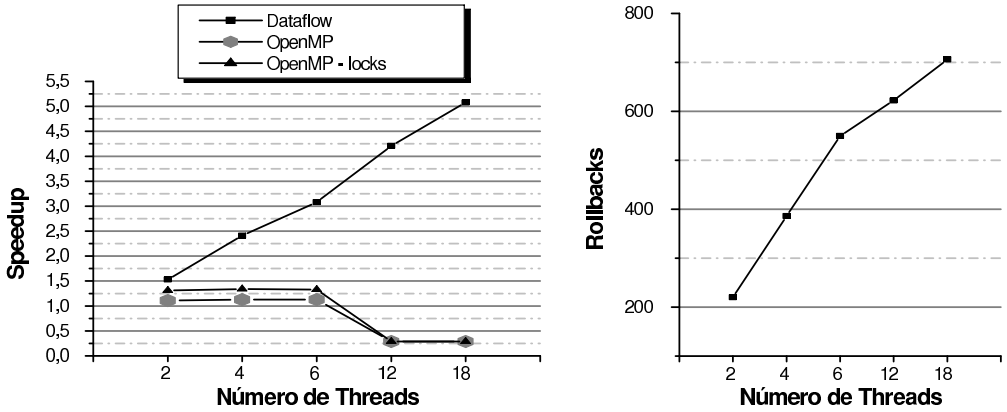


Figura 7.9: k-medoids - Comparação com implementações *OpenMP*.

As Figura 7.9 mostra os resultados para a aplicação *k-medoids*. Apesar de a versão *Dataflow* ter apresentado maior performance e escalabilidade que as versões *OpenMP*, as acelerações obtidas não são muito expressivas levando em conta o número de EPs utilizados. A baixa eficiência dessas acelerações se deve ao grande número de *rollbacks*, ocasionados por conflitos na computação concorrente dos centros dos *clusters*.

De maneira geral, em todos os experimentos observamos interessantes acelerações e boa escalabilidade, mesmo com a presença de *rollbacks*.

Capítulo 8

Discussão e trabalhos futuros

Neste trabalho apresentamos um modelo de execução *dataflow* híbrido que permite a execução de instruções de diferentes granularidades, o TALM. Em seguida apresentamos um modelo de suporte à especulação para o TALM. Para demonstrar o potencial do TALM com execução especulativa implementamos o modelo na forma de uma máquina virtual para máquinas *multicore*, a Trebuchet. Apresentamos também uma prova de que execuções especulativas do modelo são estritamente serializáveis.

Nos nossos experimentos avaliamos os diferentes custos do suporte à especulação, comparando a aplicação *bank* com um cenário ideal e a aplicação *k-medoids* com duas implementações *OpenMP*. Em ambos os casos foram obtidas acelerações e um bom grau de escalabilidade, que se mostrou comparável ao caso ideal na aplicação *bank* e bastante superior à escalabilidade das versões *OpenMP* da aplicação *k-medoids*.

Atribuímos a boa escalabilidade presente nos nossos resultados à simplicidade e ao aspecto distribuído do nosso modelo. Como não há componentes centralizados no nosso modelo de execução especulativa é natural que implementações dele escalem. Além disso, como a maior parte da validação é feita com base na troca de operandos, a implementação de operações especulativas não adiciona muita complexidade ao modelo de execução *dataflow* normal.

Apesar de já apresentar bons resultados a implementação Trebuchet ainda possui muitas questões a serem trabalhadas para poder trazer o potencial do TALM a mais aplicações e programadores. A memória transacional implementada ainda é muito simples, sem suportar recursos como alocação dinâmica de memória dentro da especulação ou supressão de exceções, recurso este que é previsto no modelo especulativo do TALM.

Para a alocação dinâmica deverá ser implementado um encapsulamento da rotina `malloc()` para que a memória alocada por uma transação entre em seu histórico e seja desalocada em caso de *rollback*. A exemplo de outras STM, o programador substituiria chamadas da `malloc()` dentro de transações pela equivalente transacional

`tm_malloc()`.

Com relação à performance, apesar da simplicidade conferida pelo modelo, algumas melhorias como o uso de filtros *Bloom* [22] podem ser adotadas para diminuir o tempo médio de acesso à STM. Além disso, procuraremos estudar uma implementação que permita a execução de instruções `Commit` de maneira concorrente quando houver certeza de que não atuarão sobre os mesmos objetos, tendo em vista que o modelo dá essa possibilidade.

Para facilitar o desenvolvimento de aplicações para a Trebuchet pretendemos desenvolver ferramentas que permitam a automatização de partes ou até de todo o processo. Uma possível abordagem para o desenvolvimento deverá ser o uso de um compilador que, a partir de um código sequencial com anotações, gerará todo o código necessário para a versão Trebuchet. Outra abordagem será o uso de uma ferramenta gráfica para a descrição do grafo *dataflow*, permitindo maior flexibilidade para descrição de mecanismos como os apresentados no Capítulo 6. No que tange a especulação, as restrições para grafos do modelo descritas no Capítulo 6 deverão servir como base para a criação de algoritmos de geração de grafos especulativos e de validação de grafos existentes.

Os exemplos apresentados na Seção 4.4 (descrição de *piplines*) e 6.7 (controle da especulação) podem parecer complexos para programadores não habituados ao paradigma *dataflow*. Apesar de mostrar como técnicas avançadas de exploração de paralelismo podem ser implementadas sem a necessidade de um controle centralizado, esse nível de detalhamento pode não ser atraente para esses programadores inexperientes. Tal barreira para a utilização do modelo poderá ser contornada em um trabalho futuro pela implementação de macros ou *templates*, semelhante aos utilizados na biblioteca *Intel Threading Building Blocks* [17]. O uso de *templates* oferece ao programador a troca de performance e flexibilidade por facilidade de programação, através de um nível mais alto de abstração.

Referências Bibliográficas

[1] HERLIHY, M., MOSS, J. E. B. "Transactional Memory: Architectural Support for Lock-Free Data Structures". In: *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp. 289–300, May 1993.

[2] DAGUM, L., MENON, R. "OpenMP: An Industry-Standard API for Shared-Memory Programming", *IEEE Comput. Sci. Eng.*, v. 5, n. 1, pp. 46–55, 1998. ISSN: 1070-9924. doi: http://dx.doi.org/10.1109/99.660313.

[3] SWANSON, S. *The WaveScalar Architecture*. Tese de Doutorado, University of Washington, 2006.

[4] MARZULO, L. A. J., FRANCA, F. M. G., COSTA, V. S. "Transactional WaveCache: Towards Speculative and Out-of-Order DataFlow Execution of Memory Operations". In: *SBAC-PAD '08: Proceedings of the 2008 20th International Symposium on Computer Architecture and High Performance Computing*, pp. 183–190, Washington, DC, USA, 2008. IEEE Computer Society. ISBN: 978-0-7695-3423-7. doi: http://dx.doi.org/10.1109/SBAC-PAD.2008.29.

[5] MARZULO, L. A. J. *Transactional WaveCache - Execução Especulativa Fora-de-Ordem de Operações de Memória em uma Máquina Dataflow*. Tese de Mestrado, COPPE - UFRJ, dez. 2007.

[6] KAVI, K., GIORGI, R., ARUL, J. "Scheduled Dataflow: Execution Paradigm, Architecture, and Performance Evaluation", *IEEE Transactions on Computers*, v. 50, n. 8, pp. 834–846, 2001. ISSN: 0018-9340. doi: http://doi.ieeecomputersociety.org/10.1109/12.947003.

[7] BURGER, D., KECKLER, S. W., MCKINLEY, K. S., et al. "Scaling to the End of Silicon with EDGE Architectures", *Computer*, v. 37, n. 7, pp. 44–55, 2004. ISSN: 0018-9162. doi: http://dx.doi.org/10.1109/MC.2004.65.

[8] POCHAYEVETS, O. *BMDFM: A Hybrid Dataflow Runtime Parallelization Environment for Shared Memory Multiprocessors*. Tese de Doutorado, Technical University of Munich, 2004.

[9] PRABHU, M. K., OLUKOTUN, K. "Exposing speculative thread parallelism in SPEC2000". In: *PPoPP '05: Proceedings of the 10th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 142–152, New York, NY, USA, 2005. ACM. ISBN: 1-59593-080-9. doi: http://doi.acm.org/10.1145/1065944.1065964.

[10] ISLAM, M. M. "Predicting Loop Termination to Boost Speculative Thread-Level Parallelism in Embedded Applications". In: *SBAC-PAD 2007: 19th International Symposium on Computer Architecture and High Performance Computing, 2007*, pp. 54–61, 2007.

[11] HAMMOND, L., CARLSTROM, B. D., WONG, V., et al. "Transactional Coherence and Consistency: Simplifying Parallel Hardware and Software", *IEEE Micro*, v. 24, pp. 92–103, 2004. ISSN: 0272-1732. doi: http://doi.ieeecomputersociety.org/10.1109/MM.2004.91.

[12] VON PRAUN, C., CEZE, L., CASCAVAL, C. "Implicit parallelism with ordered transactions". In: *PPoPP'07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming. 2007*, pp. 79–89, 2007.

[13] BALAKRISHNAN, S., SOHI, G. S. "Program Demultiplexing: Data-flow based Speculative Parallelization of Methods in Sequential Programs". In: *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pp. 302–313, Washington, DC, USA, 2006. IEEE Computer Society. ISBN: 0-7695-2608-X. doi: http://dx.doi.org/10.1109/ISCA.2006.31.

[14] BRUENING, D., DEVABHAKTUNI, S., AMARASINGHE, S. "Softspec: Software-based Speculative Parallelism". In: *ACM Workshop on Feedback-Directed and Dynamic Optimization*, Monterey, California, Dec 2000. Disponível em: <http://groups.csail.mit.edu/commit/papers/00/Softspec-FDDO.pdf>.

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

[16] CHANDY, K. M., LAMPORT, L. “Distributed snapshots: determining global states of distributed systems”, *ACM Trans. Comput. Syst.*, v. 3, n. 1, pp. 63–75, February 1985. ISSN: 0734-2071. doi: 10.1145/214451.214456. Disponível em: <http://dx.doi.org/10.1145/214451.214456>.

[17] REINDERS, J. *Intel threading building blocks*. Sebastopol, CA, USA, O’Reilly & Associates, Inc., 2007. ISBN: 9780596514808.

[18] MOKHOV, A., YAKOVLEV, A. “Conditional Partial Order Graphs and Dynamically Reconfigurable Control Synthesis”. In: *DATE*, pp. 1142–1147, 2008.

[19] RAJWAR, R., GOODMAN, J. R. “Transactional Execution: Toward Reliable, High-Performance Multithreading.” *IEEE Micro*, v. 23, n. 6, pp. 117–125, Nov-Dec 2003.

[20] SCOTT, M. L. “Sequential Specification of Transactional Memory Semantics”. In: *ACM SIGPLAN Workshop on Transactional Computing*, Jun 2006. Held in conjunction with PLDI 2006.

[21] HARTIGAN, J. A., WONG, M. A. “A K-Means Clustering Algorithm”, *Applied Statistics*, v. 28, pp. 100–108, 1979.

[22] BLOOM, B. H. “Space/time trade-offs in hash coding with allowable errors”, *Commun. ACM*, v. 13, n. 7, pp. 422–426, 1970. ISSN: 0001-0782. doi: http://doi.acm.org/10.1145/362686.362692.