EXPLORANDO LINHAS DE EXECUÇÃO PARALELAS COM PROGRAMAÇÃO ORIENTADA POR FLUXO DE DADOS

Leandro Augusto Justen Marzulo

Tese de Doutorado 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 Doutor em Engenharia de Sistemas e Computação.

Orientadores: Felipe Maia Galvão França
Vítor Santos Costa

Rio de Janeiro
Outubro de 2011

EXPLORANDO LINHAS DE EXECUÇÃO PARALELAS COM PROGRAMAÇÃO ORIENTADA POR FLUXO DE DADOS

Leandro Augusto Justen Marzulo

TESE 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 DOUTOR EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof. Felipe Maia Galvão França, Ph.D.

Prof. Vítor Santos Costa, Ph.D.

Prof. Valmir Carneiro Barbosa, Ph.D.

Prof. Claudio Luis de Amorim, Ph.D.

Prof. Philippe Olivier Alexandre Navaux, Ph.D.

Prof. Cristiana Bentes, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
OUTUBRO DE 2011

Marzulo, Leandro Augusto Justen

Explorando Linhas de Execução Paralelas com Programação Orientada por Fluxo de Dados/Leandro Augusto Justen Marzulo. – Rio de Janeiro: UFRJ/COPPE, 2011.

XVI, 140 p.: il.; 29, 7cm.

Orientadores: Felipe Maia Galvão França

Vítor Santos Costa

Tese (doutorado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2011.

Referências Bibliográficas: p. 97 – 104.

1. *Dataflow*. 2. Programação Paralela. 3. Compilador. I. França, Felipe Maia Galvão *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

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

Agradecimentos

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

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

Ao Programa de Engenharia de Sistemas e Computação (PESC/COPPE/UFRJ) por todo o suporte fornecido que me permitiu desenvolver este trabalho. Em especial gostaria de agradecer aos professores Valmir e Amorim e aos funcionário da secretaria e suporte (Cláudia, Solange, Sônia, Mercedes, Gutierrez, Itamar e Adilson).

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

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

À Capes, pela bolsa fornecida, que permitiu custear minhas despesas durante o Doutorado.

Ao programa Euro-Brazilian Window, pela oportunidade de fazer um intercâmbio e à Universidade do Porto pelo acolhimento. Em especial, gostaria de agradecer à Luísa Capitão e Ana Reis, por todo o suporte dado durante a minha estadia em Portugal. O intercâmbio foi uma experiência muito enriquecedora e contribuiu tanto para a minha formação acadêmica quanto pessoal. Foram muitas as amizades feitas neste período. Estas pessoas foram muito importantes, pois me fizeram sentir em casa, mesmo estando longe do meu país. São elas: Raphael, Thiago, Fernanda, Luis, Céu, Sá, Sr. Agostinho, Dona Emília, Marcos, Tiara, Joana, Melina, Roberta, Feu, Henrrique, Igor, Rafa, Damaris, Vanessa, Tiago, Mariana, Laís, Xádia, Ícaro, Lisa, Rodrigo Müller, Cecilia, Paula, Diogo, Rodrigo, Kássia, Cristina, Bia, Adriano, Sérgio, Érika e Eric.

Além de experiências e amizades, o intercâmbio também me deu alguém para amar: Camila, a quem agradeço por todo o carinho e compreensão nos momentos em que estive ausente por conta da realização deste trabalho. Obrigado por entender meu jeito apreensivo e sempre ansioso e por sempre apostar no nosso futuro. Su-

perar a distância dos 10 meses finais de intercâmbio não foi fácil, mas você sempre esteve presente (mesmo que pela internet) para me dar forças para conquistar meus objetivos. Te amo!

Aos Professores e colegas do Departamento de Ciência de Computadores da Universidade do Porto: Fernando Silva, Inês, Vinay, Fábio, Fahan e Bruno.

Aos vários amigos que fiz na UFRJ: Bruno, Ivomar, Talita, Alexandre, Bernardo, Danilo, Fabiano, André Nathan, Vivian, João Victor, João Maurício, Carlos Melo, Priscila, Lawrence, Alexandre Nery, Alexandre Sardinha, Fábio Flesch e Rafael. Obrigado a todos pela amizade e agradáveis momentos que me proporcionaram.

Ao Tiago, obrigado pela amizade e pelo companheirismo nesta jornada, enquanto colegas de trabalho. Obrigado pela dedicação, força de vontade, determinação e por escolher também este projeto para seu trabalho de mestrado e doutorado. O trabalho conjunto, certamente ampliou o leque de resultados obtidos.

Aos meus pais, por me tornarem um homem de caráter e por me fazerem sempre valorizar a importância dos estudos. Ao meu irmão, por ser um eterno amigo e companheiro. Ao meu padrinho, pela amizade, carinho e conversas descontraídas. À todos os meus amigos e familiares que, junto com meu conhecimento, compõem o bem mais precioso que possuo. Em especial, gostaria de agradecer ao Paulo, Marcus, Maca, Bruno, Guilherme, Janine, Nando, Carol Tabuada e Sabrina.

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

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

EXPLORANDO LINHAS DE EXECUÇÃO PARALELAS COM PROGRAMAÇÃO ORIENTADA POR FLUXO DE DADOS

Leandro Augusto Justen Marzulo

Outubro/2011

Orientadores: Felipe Maia Galvão França
Vítor Santos Costa

Programa: Engenharia de Sistemas e Computação

Execução guiada por fluxo de dados permite extração de paralelismo de forma natural. Entretanto, descrever dependências de controle entre tarefas *dataflow* com granularidade fina pode ser complexo e apresentar custos indesejáveis. Neste trabalho é apresentado o TALM (*TALM is an Architecture and Language for Multi-threading*) um modelo *dataflow* de programação paralela com granularidade definida pelo usuário. No TALM, os programadores identificam blocos de código, chamados de super-instruções, para executarem em paralelo, e depois os conectam em um grafo de fluxo de dados. O TALM foi implementado como um sistema de execução híbrido (Von Neumann/*dataflow*): a *Trebuchet*. Observou-se que a utilidade do TALM depende largamente da forma que os programadores especificam e conectam super instruções. Foi elaborada uma linguagem de alto nível para o TALM e, para dar suporte a esta linguagem, foi desenvolvido o *Couillard*, um compilador completo que cria, baseado em um program C anotado, um grafo de fluxo de dados e o código *C* correspondente a cada super-instrução. Este trabalho mostra que o conjunto de ferramentas desenvolvido permite obter os benefícios da execução guiada por fluxo de dados e explorar técnicas de programação paralela sofisticadas, com pequeno esforço. Para avaliar o sistema desenvolvido, foi executado um conjunto de aplicações reais em duas máquinas com múltiplos núcleos de processamento. Uma comparação com métodos populares de programação paralela mostra acelerações competitivas, enquanto é provida uma alternativa mais fácil de programação paralela.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

EXPLORING MULTITHREADED EXECUTION WITH DATAFLOW ORIENTED PROGRAMMING

Leandro Augusto Justen Marzulo

October/2011

Advisors: Felipe Maia Galvão França
Vítor Santos Costa

Department: Systems Engineering and Computer Science

Dataflow is a natural approach to parallelism. However, describing dependencies and control between fine-grained dataflow tasks can be complex and present unwanted overheads. In this work we present TALM (TALM is an Architecture and Language for Multi-threading): a dataflow model for parallel programming with user-defined granularity. In TALM, programmers identify code blocks, called super-instructions, to be run in parallel and connect them in a dataflow graph. TALM has been implemented as a hybrid Von Neumann/dataflow execution system: the *Trebuchet*. We have observed that TALM's usefulness largely depends on how programmers specify and connect super-instructions. Thus, we have created a high-level language for TALM and to support this language, we have developed *Couillard*, a full compiler that creates, based on an annotated C-program, a dataflow graph and C-code corresponding to each super-instruction. We show that our toolchain allows one to benefit from dataflow execution and explore sophisticated parallel programming techniques, with small effort. To evaluate our system we have executed a set of real applications on two different multi-core machines. Comparison with popular parallel programming methods shows competitive speedups, while providing an easier parallel programing approach.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Extrair desempenho de processadores de único núcleo se tornou uma tarefa árdua e complexa. O aumento das temperaturas limita a frequência de relógio que pode ser usada nos processadores recentes. Por outro lado, e seguindo Olukotun [1], "a complexidade da lógica adicional necessária para encontrar instruções paralelas dinamicamente é proporcional ao quadrado do número de instruções que podem ser emitidas simultaneamente". Sendo assim, embora a tecnologia de silício ainda evolua, permitindo a inserção de mais transistores em uma mesma área de circuito integrado, os projetos recentes de processadores de único núcleo não foram considerados a melhor solução para converter estes transistores extras em ganhos de desempenho significativos.

Processadores com múltiplos núcleos por *chip*, ou CMPs (*single-Chip Multiprocessors*), são agora a alternativa mais amplamente adotada. CMPs podem alcançar ganhos de desempenho significativos [2] com maior eficiência no consumo de energia [3]. É, portanto, natural que CMPs tenham dominado a indústria de processadores. Note que classes de aplicações importantes, como gerenciadores de bancos de dados, exibem uma quantidade significativa de paralelismo, e por isso podem ser imediatamente beneficiadas pelos múltiplos núcleos de processamento. Por outro lado, um grande número de aplicações foram escritas de forma estritamente sequencial. Tais aplicações precisam ser extensivamente reescritas para expor paralelismo.

Infelizmente, desenvolver aplicações com múltiplas linhas de execução não é trivial. Primeiro, é preciso encontrar porções de código (ou tarefas) que acredita-se que devam executar em paralelo. Tarefas com granularidade mais fina permitirão mais paralelismo, mas terão um custo de manutenção maior (programas com paralelismo de granularidade fina são geralmente mais complexos, especialmente quanto à sincronização); tarefas de granularidade mais grossas resultarão em menos paralelismo, mas serão de mais fácil gerenciamento. Segundo, essas tarefas precisarão colaborar em algum momento, seja para compartilhar dados ou simplesmente para sincronizar controle, por exemplo, para estabelecer o término de uma fase do algoritmo. É

responsabilidade do programador estabelecer esses mecanismos não triviais de colaboração entre tarefas. Técnicas como *barreiras* e *locks*, ou, mais recentemente, *Memórias Transacionais* [4], podem ser usadas para garantir o acesso correto à recursos compartilhados. Terceiro, apenas possuir um programa paralelo correto pode não ser suficiente. O programa deve ter melhor desempenho que a versão sequencial. De fato, geralmente é desejado que a aplicação escale quando máquinas novas, com mais núcleos de processamento, se tornam disponíveis.

Alternativas populares para paralelização são baseadas em tecnologias padrão, como *OpenMP* [5] e *MPI* [6]. O *OpenMP* segue o paradigma de memória compartilhada; em contraste, o *MPI* apresenta, fundamentalmente, um modelo de passagem de mensagens. Em *OpenMP*, a sincronização é feita normalmente através de barreiras, *locks* e semáforos. Barreiras são populares pois são de simples entendimento, especialmente se a computação pode ser dividida em fases. No entanto, elas requerem que núcleos de processamento fiquem ociosos até que **todos** os outros terminem, o que pode levar períodos de tempo relativamente longos. Alternativas como *OpenMP* e *MPI*, que requerem que o usuário provenha o fluxo de controle esperado entre tarefas, tendem a limitar o paralelismo. O problema é que o programador precisa garantir a execução segura (tarefas só podem executar em paralelo se a semântica do programa se mantem inalterada). Para tal, ele irá forçar frequentemente que as tarefas fiquem estagnadas enquanto aguardam que outras tarefas terminem a seção de código corrente. Enquanto isso, porções de código na seção seguinte poderiam ter todos os operandos de entrada disponíveis, mas pode ser inseguro ou muito complexo prosseguir com a execução (pode ser difícil determinar quando uma tarefa é segura ou não).

Balakrishnan e Sohi trataram este problema com a criação do *Program Demultiplexing* [7], um paradigma de execução onde métodos ou funções são demultiplexados e executados concorrentemente com o restante do programa. A execução de métodos demultiplexados é acionada de acordo com a regra *dataflow*, isto é, sua execução é iniciada assim que seus operandos de entrada (conjunto de leitura) estejam disponíveis, o que pode ocorrer antes que os métodos sejam efetivamente chamados, segundo a ordem do programa.

A ideia de executar operações assim que os operandos estejam disponíveis, ao invés de seguir a ordem do programa, é um princípio fundamental de modelos *dataflow*. Execução guiada por fluxo de dados é uma forma natural de expor paralelismo. Neste modelo, as instruções são executadas tão logo seus operandos estejam disponíveis [8–13]. De fato, no *dataflow* dinâmico, é possível ter instruções independentes de múltiplas iterações de um laço sendo executadas simultaneamente, pois partes do laço podem ser executadas mais rapidamente que outras e atingir iterações seguintes. Sendo assim, é uma tarefa complexa descrever controle em *dataflow*, pois instruções

apenas podem prosseguir para execução quando todos os operandos necessários em uma mesma iteração estiverem disponíveis para a instrução em questão. No entanto, essa dificuldade pode ser compensada pela quantidade de paralelismo exposta pelo modelo. Além disso, é evidente que a execução de instruções *dataflow* de granularidade fina pode apresentar um custo elevado, pois o controle especificado também será mais complexo.

Tendo essas observações em mente, este trabalho apresenta o TALM (*TALM is an Architecture and Language for Multi-threading*) [14–17], um modelo de execução, baseado em arquiteturas *dataflow*, concebido para explorar as vantagens da execução guiada por fluxo de dados para programação com múltiplas linhas de execução. Um programa no TALM é composto por blocos de código, chamados de super-instruções, e instruções simples conectadas em um grafo que descreve as dependências entre elas, isto é, um grafo de fluxo de dados. Para paralelizar um programa usando o TALM o programador marca porções de código que devem se tornar super-instruções e descrever suas dependências. Com esse método, o paralelismo surge naturalmente da execução guiada por fluxo de dados.

A maior vantagem do TALM é prover um modelo de programação paralela de granularidade variável que pode tirar vantagem de execução guiada por fluxo de dados em máquinas de Von Neumann. Além disso, é um modelo bastante flexível, visto que as principais instruções *dataflow* estão disponíveis, permitindo, portanto, a compilação total do controle de forma *dataflow*. Esta característica dá ao programador o poder de escolher a granularidade a ser adotada de acordo com sua estratégia de paralelização. Esta alternativa é contrastante com trabalhos anteriores em programação *dataflow* [8–10] que focavam em esconder a execução *dataflow* do programador. Entretanto, tais alternativas eram arquiteturas *dataflow* completas que não se tornaram realidade.

Uma primeira implantação do TALM, a *Trebuchet* foi desenvolvida como um sistema híbrido de execução (Von Neumann/*dataflow*) para arquiteturas CMP. A *Trebuchet* emula uma máquina *dataflow* que suporta tanto instruções simples quanto super-instruções. Super instruções são compiladas como funções separadas que são chamadas pelo ambiente de execução, enquanto instruções simples são interpretadas durante a execução. Embora a *Trebuchet* precise emular instruções *dataflow*, a experiência mostrou que a maior parte do tempo de execução está contida nas super-instruções. Resultados mostram que a *Trebuchet* é competitiva com aplicações em seu estado-da-arte, desenvolvidas com uso de *OpenMP*. Por outro lado, paralelismo para aplicações do tipo SPMD (*single-program multiple data*) podem ser bem exploradas usando ferramentas como o *OpenMP*. Os principais benefícios explorados pelo TALM se tornam aparentes ao experimentar aplicações que requerem técnicas mais complexas, como *software pipelining* e execução especulativa [16, 17].

Para avaliar o modelo TALM e os potenciais de desempenho da *Trebuchet* foi paralelizado um conjunto de 7 aplicações: uma aplicação de cálculo de determinante de matrizes, uma aplicação de multiplicação de matrizes, uma aplicação de renderização de cenas usando *ray-tracing*, o Equake do SpecOMP 2001 [18], o IS do NPB3.0-OMP [19], e também o LU e Mandelbrot do *OpenMP Source Code Repository* [20]. As acelerações obtidas para oito *threads*, em relação à versão sequencial foram, respectivamente 2,00, 4,16, 4,39, 3,61, 3,00, 2,19 e 7,16 vezes. Foi realizada uma comparação com uma versão *OpenMP* dessas aplicações. As acelerações obtidas com o *OpenMP* foram 1,94, 4,15, 4,39, 3,40, 3,11, 2,19 e 7,13 vezes. Esses resultados são muito promissores e mostram que a *Trebuchet* pode ser competitiva com soluções consolidadas para aplicações regulares, além de prover a flexibilidade (e facilidade de programação) do modelo *dataflow* para aplicações mais complexas.

A utilidade do TALM depende claramente em como o programador pode especificar e conectar super-instruções, incluindo a complexa tarefa de descrever controle usando instruções *dataflow*. É portanto introduzido o *Couillard*, um compilador de programas descritos em linguagem *C* e anotados com diretivas TALM. O resultado da compilação é um grafo de fluxo de dados, incluindo a descrição do controle em *dataflow*. Além disso, o *Couillard* gera o código C com as funções associadas às super-instruções, para execução na *Trebuchet*. O Couillard foi concebido para esconder, do programador, os detalhes indesejáveis de programação orientada por fluxo de dados. Por requerer apenas que o programador anote o código com as definições de super-instruções e sua dependências, o *Couillard* simplifica extremamente a tarefa de paralelizar aplicações com o TALM.

O desempenho do *Couillard* foi avaliado em duas aplicações em estado-da-arte, pertencentes ao PARSEC [21]. Experimentos demonstram que a *Trebuchet* e o *Couillard* permitem explorar técnicas avançadas de programação paralela, como *pipelines*, e esconder latência de operações de entrada e saída. Uma comparação com modelos populares de programação paralela como *Pthreads* [22], *OpenMP* [5] e *Intel Thread Building Blocks* [23] (TBB) mostram que o modelo apresentado não é apenas compatível com tecnologia em estado-da-arte, mas de fato pode atingir melhor desempenho, por permitir a fácil exposição de um ambiente de desenvolvimento sofisticado para programação paralela.

Este trabalho faz as seguintes contribuições:

- Definição da arquitetura TALM e sua linguagem de montagem.
- Criação de um montador para a linguagem de montagem TALM.
- Criação da *Trebuchet*, uma implantação do TALM para CMPs.
- Criação de um *loader* para a *Trebuchet*.

- Execução de experimentos com aplicações regulares e comparação com *OpenMP*, mostrando que a *Trebuchet* tem desempenho equivalente.
- Criação de uma linguagem em alto nível para o TALM, estendendo a linguagem *C*, para marcar super-instruções e descrever suas dependências.
- Criação do *Couillard*, um compilador para a linguagem de alto nível do TALM.
- Execução de experimentos com aplicações com padrões de paralelismo mais complexos e irregulares, mostrando uma comparação com outras técnicas como *Pthreads* [22] e *Intel Thread Building Blocks* (TBB) [23].

O restante deste trabalho está dividido da seguinte forma: *(i)* o Capítulo 2 faz uma revisão de conceitos necessários para o entendimento deste trabalho; *(ii)* o Capítulo 3 discute os trabalhos relacionados; *(iii)* o Capítulo 4 apresenta o modelo TALM; *(iv)* o Capítulo 5 apresenta a linguagem de alto nível do TALM; *(v)* o Capítulo 6 discute a implementação e uso da Trebuchet; *(vi)* O Capítulo 7 apresenta o compilador *Couillard*; *(vii)* os experimentos e resultados são apresentados no Capítulo 8; *(viii)* a conclusão e trabalhos futuros são discutidos no Capítulo 9; *(ix)* o Apêndice A exibe uma lista dos trabalhos publicados neste projeto; *(x)* o Apêndice B apresenta a *FlowPGA* que pode ser usada como base para a implementação de um futuro suporte de *hardware* para a Trebuchet; *(xi)* o Apêndice C apresenta a *Transactional WaveCache*, um mecanismo de execução especulativa de operações de memória em uma arquitetura *dataflow*; *(xii)* o Apêndice D mostra o código fonte completo do Blackscholes, uma das aplicações usadas nos experimentos, para que o leitor possa verificar o uso do modelo em uma aplicação real.

Capítulo 2

Revisão de Conceitos

Neste capítulo são explicados conceitos necessários para o entendimento deste trabalho. Inicialmente é apresentada uma revisão de conceitos de programação paralela. Em seguida é feita uma revisão do modelo de computação guiada por fluxo de dados (*dataflow*).

2.1 Programação Paralela

Com o advento dos processadores de múltiplos núcleos e sua adoção como padrão de mercado, programação paralela se tornou um assunto de extrema importância. Anteriormente, para melhorar o desempenho na execução de um programa serial, bastava obter um processador mais novo. Hoje, os processadores mais novos não operam com frequências maiores ou apresentam melhorias arquiteturais que proporcionariam um melhor desempenho para aplicações sequenciais. Atualmente, a evolução na indústria de processadores está em produzir *chips* com mais núcleos de processamento [2, 3].

Do ponto de vista da multiprogramação há um benefício imediato trazido por estes núcleos extras, pois diversos processos serão executados em paralelo, diminuindo o tempo de execução total de cargas de trabalho compostas por múltiplos processos independentes. No entanto, do ponto de vista de um processo serial, não há nada que garanta uma redução no tempo de execução. Aplicações sequencias devem então ser reescritas para usar os múltiplos núcleos de processamento disponíveis, ou seja, explorar paralelismo em nível de linhas de execução (ou *threads*).

2.1.1 Problemas Relacionados ao Uso de Programação Paralela

Escrever programas paralelos expõe ao programador uma série de novos problemas. Primeiramente, há uma mudança de paradigma. O desenvolvedor deve pensar qual é

o trabalho realizado pelo seu programa e dividir, se possível, este trabalho em tarefas que poderiam ser executadas em paralelo. Além disto caso haja alguma dependência entre as tarefas, como compartilhamento de dados ou algum tipo de comunicação, o programador terá que descrever este comportamento, seja manualmente ou usando alguma ferramenta ou técnica que facilite o processo. Por fim, o programador precisa distribuir essas tarefas para execução em múltiplos núcleos de processamento. Para resolver estes problemas de forma eficiente, muitas vezes é exigido um conhecimento que vai além dos conhecimentos de lógica de programação.

2.1.1.1 Questões Relacionadas à Memória

Escolher a melhor estratégia para dividir um programa em tarefas exige, por exemplo, um conhecimento da hierarquia de memória da máquina que irá executar o programa. Ao se paralelizar uma aplicação é extremamente importante considerar os efeitos colaterais causados por acessos à memória. Se as tarefas (ou *threads*) de uma aplicação realizam constantemente operações de *load* e *store* em uma memória compartilhada, invalidações na *cache* podem introduzir *overheads* significativos que poderiam, até mesmo, fazer com que a versão paralela se torne mais lenta que a sequencial. Além disto, a forma como a aplicação é dividida em *threads* faz com que a localidade de memória seja uma preocupação.

A Figura 2.1 mostra um trecho de código que realiza a soma de seis vetores (Y, Z, W, K, L, M) e armazena o resultado no vetor X. Suponha que deseja-se paralelizar este código com a criação de quatro tarefas, cada uma sendo responsável pela computação de uma porção do vetor X. Nesta figura, a variável `ntasks` guarda o número total de tarefas (quatro) e a variável `taskID` guarda o identificador da tarefas (de um a três). Para este exemplo são exibidos duas formas de distribuir a computação entre as tarefas: *(i)* em B, as tarefas irão trabalhar em elementos esparsos dos vetores, e; *(ii)* em C, as tarefas irão trabalhar em elementos consecutivos destes vetores. Na primeira solução, a localidade da memória não está sendo bem explorada. Se as tarefas forem mapeadas em EPs diferentes, executando em núcleos de processamento diferentes, sem *caches* compartilhadas, os blocos de cache carregados em cada núcleo não serão totalmente utilizados. Por outro lado, a segunda opção é ciente da localidade, visto que os blocos carregados terão mais elementos acessados pela tarefa correspondente. É claro que a ciência da localidade não é tão significativa para máquinas cujo tamanho do bloco da *cache* seja muito maior que o tamanho médio do objeto de memória usado pela aplicação.

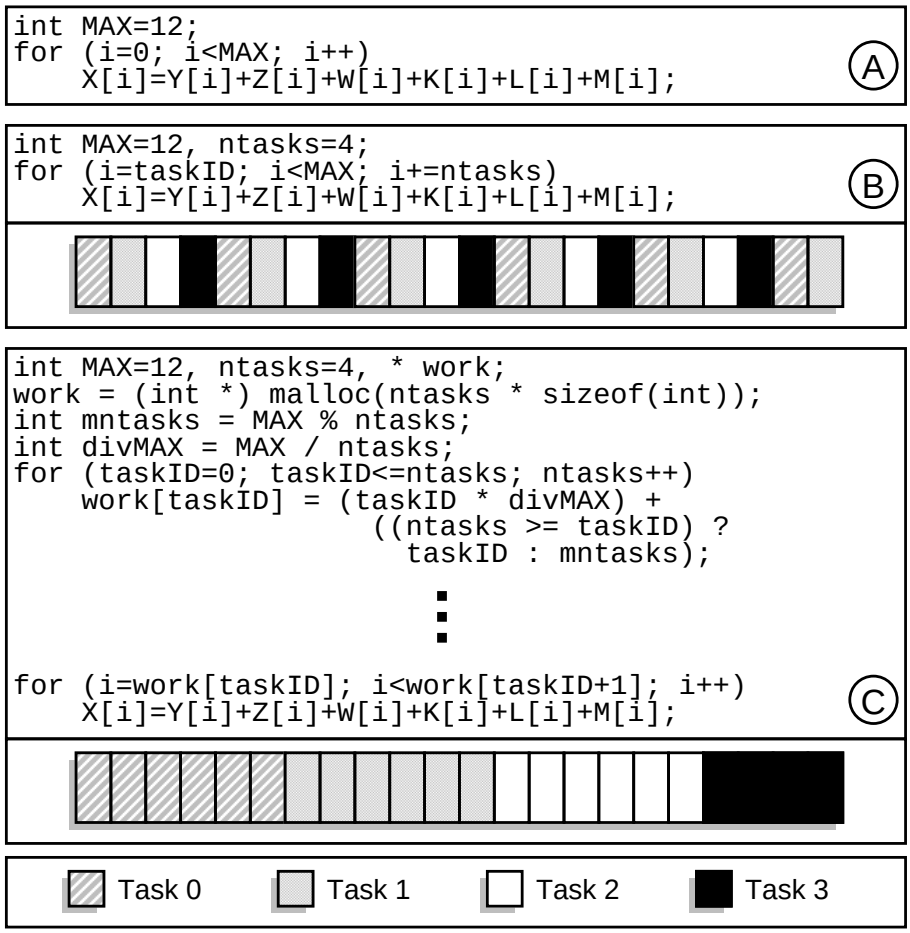

Figura 2.1: Escrevendo código ciente da localidade. O quadro A mostra o código sequencial. O quadro B mostra a divisão do trabalho em quatro *threads* sem considerar a localidade, enquanto que no quadro C a localidade é levada em conta.

2.1.1.2 Comunicação e Compartilhamento de Recursos

Tarefas concorrentes precisam se comunicar em algum ponto para, por exemplo, compartilhar dados ou estabelecer a terminação de uma fase do algoritmo. Essa comunicação pode ser feita por troca de mensagens ou por memória compartilhada.

Quando a comunicação é feita através de troca de mensagens, o programador precisa estar ciente de que existem custos associados. Se duas tarefas estão em um mesmo núcleo de processamento ou até em núcleos de um mesmo *chip*, a comunicação será feita na rede de interconexão interna e será pouco custosa. Quando as tarefas estão em núcleos em *chips* diferentes de uma mesma máquina (em uma mesma placa mãe) o custo de comunicação será o do acesso aos barramentos da placa mãe. No caso onde as tarefas estão em núcleos de máquinas distintas (nós em um *cluster* ou *grid*) o custo de comunicação será muito maior.

No caso do uso de memória compartilhada, é necessário garantir atomicidade no acesso à memória. O uso de *locks* ou, mais recentemente, *memória transacional* é empregado com este processo. O uso de *locks* demanda uma atenção maior do programador, visto que seu emprego incorreto pode acarretar em impasses, inanição ou contenção. O uso de *memórias transacionais* torna este processo mais transparente, mas existe um custo associado que pode ser grande, dependendo da aplicação.

Para estabelecer a terminação de uma fase do algoritmo, *barreiras* são muito populares pois são fáceis de entender. No entanto introduzem uma condição de espera muito forte, principalmente se mal utilizadas.

2.1.1.3 Escalonamento e Balanceamento de Carga

Ao executar um programa paralelo em um ambiente com múltiplos núcleos de processamento, é preciso determinar em qual núcleo cada tarefa será executada, ou seja, o escalonamento. É preciso levar em conta dois objetivos contrastantes: a diminuição de custos de computação e contenção devidos à compartilhamento de recurso e o aumento ganho de desempenho com execução paralela. Distribuir tarefas que se comuniquem ou compartilhem recursos frequentemente em diversos núcleos de processamento pode aumentar os custos de comunicação e contenção. Por outro lado, concentrar as tarefas em poucos núcleos de processamento diminui as oportunidades de obtenção de ganhos de desempenho com execução paralela.

Os sistemas operacionais modernos possuem mecanismos de escalonamento de *threads* de processamento. No entanto, o programador pode configurar manualmente a afinidade das *threads* de processamento com um determinado subconjunto de núcleos, para que o escalonador do sistema operacional tenha candidatos mais apropriados para cada *threads*. Além disto, ambientes de execução voltados para programação paralela podem oferecer mecanismos de escalonamento mais eficientes,

que diminuam ou dispensem a intervenção do programador.

Em aplicações cujas tarefas tenham duração mais heterogênea, é comum aparecerem problemas de balanceamento de carga. Por exemplo, quando um conjunto de tarefas é seguido de uma barreira, se algumas tarefas são muito mais longas, alguns núcleos de processamento podem ficar ociosos. O programador pode evitar este problema dividindo melhor o trabalho realizado, ou tendo um número de tarefas muito superior ao número de núcleos. Assim, um mecanismo de escalonamento dinâmico pode atribuir, em tempo de execução, outras tarefas aos núcleos que pegaram tarefas mais curtas. Esta técnica é chamada de *over-subscription*. No entanto, é preciso levar em conta que diminuir a quantidade de trabalho a ser feita em uma tarefa para aumentar o número de tarefas pode tornar mais aparentes os custos de comunicação e contenção. Além disto, o gerenciamento e o escalonamento de uma tarefa pelo sistema operacional (ou ambiente de execução) incorre em custos. Esta escolha precisa ser ponderada pelo programador. Os custos de comunicação e contenção extras podem ser escondidos com a computação de outras tarefas, se a paralelização da aplicação for bem elaborada pelo programador.

2.1.2 Técnicas, Modelos e Tecnologias para Programação Paralela

2.1.2.1 *Pthreads*

Pthreads (POSIX *Threads*) [22] é um padrão para *threads*. É definida uma API (*Application Programming Interface*) que fornece dezenas de primitivas para a criação e manipulação de *threads*, além de sincronização. As principais primitivas são:

- `pthread_create` cria uma *thread* que inicia sua execução em uma função, cujo nome e argumentos são passados na chamada da primitiva. Além disto, um outro argumento guarda um ponteiro para a *thread* criada, para ser usado para operações de sincronização.
- `pthread_exit` é chamada por uma *thread* para finalizar sua própria execução
- `pthread_cancel` cancela a execução de uma *thread* cujo ponteiro (fornecido pela `pthread_create`) é passado como argumento.
- `pthread_join` bloqueia *thread* chamadora até que a *thread* cujo ponteiro (fornecido pela `pthread_create`) foi passado como argumento termine sua execução. Tem a função de sincronização.
- `pthread_mutex_lock` adquire um *lock*. Usado para acesso à recursos compartilhados.

- `pthread_mutex_unlock` libera um *lock*.

Embora a API permita uma programação extremamente flexível, o uso de primitivas para manipulação de *threads* expõe aos programadores detalhes que nem sempre são necessários (um modelo de mais alto nível pode facilitar esta tarefa). No entanto, para programadores experientes que desejam ajustar manualmente detalhes específicos da paralelização para obter melhor desempenho, a API se mostra bastante completa.

2.1.2.2 O *MPI*

O *MPI* (*Message Passing Interface*) [6] é também uma API que permite a comunicação entre processos através do envio e recebimento de mensagens. O seu uso é mais adequado em arquiteturas distribuídas, como *clusters* de computadores. É também um modelo bastante flexível. Porém muitos detalhes são expostos ao programador. No *MPI*, o programador é responsável por definir a estratégia de paralelização e a hierarquia dos processos para computar paralelamente uma tarefa. Além disto, o fato de não ter uma memória compartilhada pode dificultar bastante a tarefa de programação, pois o programador precisa enviar mensagens para atualizar dados em *threads* em processadores distintos.

2.1.2.3 O *OpenMP*

O *OpenMP* (*Open Multi-Processing*) [5] é uma API para programação paralela em sistemas com memória compartilhada. Ele consiste em um conjunto de diretivas, bibliotecas e variáveis de ambientes para descrever tarefas paralelas. A intenção principal é prover ao programador uma interface simples e flexível para paralelizar aplicações.

O *OpenMP* é muito utilizado para aplicações que seguem o modelo *fork/join*, onde uma *thread* mestra instância *threads* escravas que vão dividir o trabalho a ser feito. As *threads* executam concorrentemente e um ambiente de execução é responsável pelo escalonamento.

O *OpenMP* disponibiliza anotações para descrição de laços paralelos onde grupos de iterações são transformados em *threads* para computar concorrentemente o trabalho feito no laço. A API faz, para este tipo de construção, a divisão automática da tarefa feita no laço, seguindo algumas diretivas de escalonamento e granularidade, definidas pelo programador.

A desvantagem do *OpenMP* é que aplicações que fogem a este modelo, como aplicações irregulares ou com padrões de paralelismo como pipeline, são mais complexas de descrever. Além disto, laços de repetição paralelos descritos com o *OpenMP* geralmente contém barreiras ao seu término. A sincronização imposta por uma

barreira deste tipo é muito forte, pois podem haver diversos blocos que poderiam prosseguir com segurança. O programador *OpenMP* tende a usar estas construções, descartando, muitas vezes, grandes potenciais de paralelismo. Um mecanismo de sincronização mais fino e ainda implícito poderia fornecer melhor desempenho.

2.1.2.4 Intel® *Threading Building Blocks* (TBB)

O Intel® *Threading Building Blocks* (TBB) [23] é uma biblioteca $C++$ projetada para prover uma camada abstrata para auxiliar programadores no desenvolvimento de código com múltiplas linhas de execução. O TBB permite que o programador especifique tarefas paralelas, o que leva a uma programação de mais alto nível do que escrever diretamente código para gerenciar *threads*. Uma outra funcionalidade do TBB é o uso de *templates* para instanciar mecanismos como *pipelines*. Os *templates*, no entanto, possuem limitações. Por exemplo, apenas *pipelines* lineares podem ser descritos usando *templates*.

2.2 O modelo *Dataflow*

Os processadores atuais, que dominam o mercado desde a criação do primeiro computador, seguem o modelo de Von Neumann. Neste modelo, a execução de instruções é guiada pelo fluxo de controle, isto é, instruções são executadas de acordo com a ordem em que aparecem no programa. Um contador de programa é usado para indicar a próxima instrução a ser executada. Este contador é alterado por instruções de desvio, usadas em laços de repetição ou para descrever execução condicional. Note que este modelo é intrinsecamente sequencial. No entanto, tenta-se resgatar paralelismo em nível de instruções com técnicas como pipelining [24], predição de desvio [24] e renomeamento de registradores [25].

O modelo *dataflow* [8, 8, 10, 26, 26–34] expõe paralelismo de forma natural. Neste modelo, as instruções são executadas de acordo com o fluxo de dados, ou seja, assim que todos os seus operandos de entrada estiverem disponíveis. O TALM, apresentado neste trabalho, é um modelo de programação que segue os princípios do modelo *dataflow*. É, portanto, necessário compreender o funcionamento de execução guiada por fluxo de dados, antes de introduzir o TALM.

2.2.1 O Conceito Básico

Programas no modelo *dataflow* podem ser descritos em um grafo de fluxo de dados onde os nós representam as instruções e uma aresta direcionada $A \rightarrow B$ indica que a instrução A produz um operando que é entrada da instrução B. De fato, este modelo é adotado em máquinas de Von Neumann com o intuito de extrair

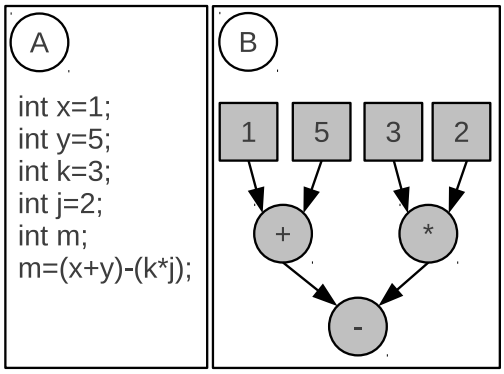

Figura 2.2: Exemplo de um programa em *dataflow*. O quadro *A* mostra um trecho de código em alto nível e o quadro *B* mostra o grafo dataflow associado. Instruções são representadas por nós no grafo e os operandos trocados entre elas são representados por arestas.

paralelismo, implementado como um mecanismo de execução fora-de-ordem com escalonamento dinâmico baseado em fluxo de dados [35]. No entanto, o paralelismo explorado está limitado pela emissão das instruções, que continua seguindo o fluxo de controle. Em arquiteturas que adotam execução totalmente guiada pelo fluxo de dados, instruções não são emitidas segundo a ordem do programa programa. De fato, instruções independentes pode executar concorrentemente sem alterar a semântica do programa. A Figura 2.2 mostra um exemplo de um programa simples em *dataflow*. No quadro *A* é mostrado o código em alto nível e no quadro *B* o grafo de fluxo de dados associado. Note que as instruções de soma e multiplicação são independentes e poderiam ser executadas em qualquer ordem, ou até mesmo em paralelo.

2.2.2 Desvios em *Dataflow*

Um dos principais problemas do modelo *dataflow* está relacionado à dificuldade e alto custo para descrever desvios de controle. Como não há registradores em uma máquina *dataflow*, para executar um desvio condicional, por exemplo, não basta apenas executar as instruções do trecho selecionado pelo desvio. É necessário converter o desvio de controle em desvio de dados. Isto é feito através de uma instrução de desvio de fluxo de dados que recebe um operando e um booleano. O operando é então encaminhado para um dos dois caminhos possíveis no programa, dependendo do valor do booleano. É necessário criar uma instrução de desvio de fluxo de dados para cada operando usado nos diferentes ramos do desvio.

A Figura 2.3 mostra um exemplo do uso de instruções de desvio em *dataflow*. O quadro *A* mostra um código em alto nível, contendo uma construção *IF-THEN-ELSE* e o quadro *B* mostra o grafo de fluxo de dados associado, onde as instruções

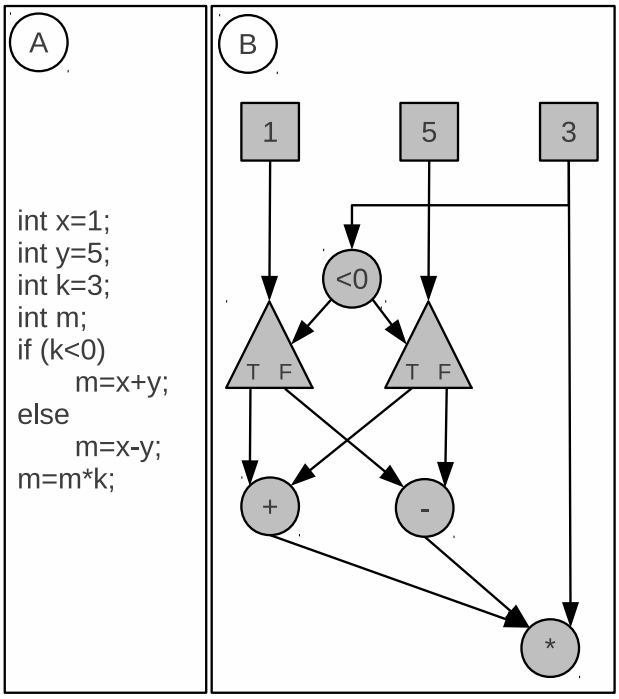

Figura 2.3: Exemplo de desvios em *dataflow*. O quadro A mostra um trecho de código em alto nível e o quadro B mostra o grafo dataflow associado. Instruções de desvio são representadas por triângulos no grafo.

de desvio são representadas por triângulos. As instruções de desvio possuem duas portas de saída, T e F, habilitadas para enviar o operando caso o booleano seja, respectivamente, verdadeiro ou falso. Note que é necessária uma instrução de desvio para cada operando necessário dentro do corpo do *IF* ou do *ELSE*.

2.2.3 Laços em *Dataflow*

O problema do controle também se estende para laços de repetição. Laços são uma grande fonte de paralelismo. Durante a execução de um laço, porções independentes de uma iteração podem rodar mais rapidamente que outras e atingirem a iteração seguinte antes que a execução da iteração corrente tenha sido completamente finalizada. Por um lado é desejável a extração desta forma válida de paralelismo. Por outro lado, é necessário ter cuidado para que operandos da nova iteração não se misturem aos da iteração anterior. Existem duas soluções para evitar que operandos de diferentes iterações casem e sejam consumidos na execução de uma instrução:

1. no *dataflow* estático, instruções em uma iteração de um laço podem executar apenas quando a iteração anterior tiver terminado;

2. no *dataflow* dinâmico, uma instrução pode ter múltiplas instâncias, uma por iteração, sendo cada operando rotulado com o número da instância ao qual

está associado.

No caso do *dataflow* dinâmico, quando um operando atinge a iteração seguinte o seu rótulo é incrementado para casar com o dos outros operandos na mesma iteração. O incremento do rótulo é feito por uma instrução para este propósito. A condição para que uma instrução seja executada no modelo é então alterada. Com o *dataflow* dinâmico, uma instrução é executada assim que todos os operandos que ela necessita estejam disponíveis e *os operandos em questão possuam o mesmo rótulo.* Portanto, em um laço de repetição, é preciso que todos os operandos usados dentro do laço tenham seus rótulos incrementados a cada iteração. A Figura 2.4 mostra um exemplo de um somatório, cujo código em alto nível está descrito no quadro A, enquanto o grafo associado é descrito no quadro B. As instruções de incremento de rótulo de operandos são descritas por losangos no grafo.

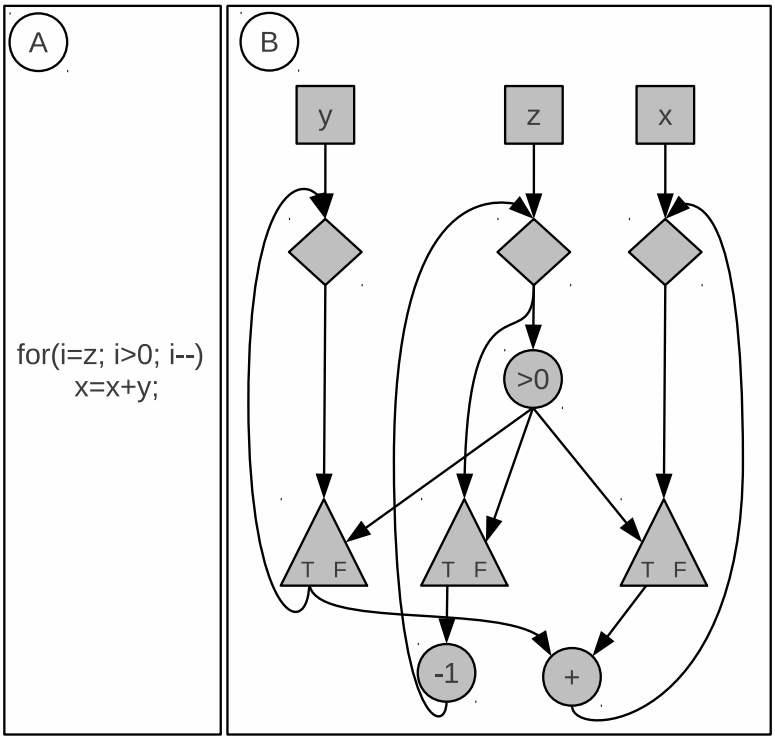

Figura 2.4: Exemplo de laços de repetição em *dataflow*. O quadro A mostra um trecho de código em alto nível e o quadro B mostra o grafo dataflow associado. Instruções de incremento do rótulo de iteração são representadas por losangos no grafo.

2.2.4 Compatibilidade com Linguagens Imperativas

Outro problema bastante importante é a incompatibilidade com linguagens imperativas devido aos acessos à memória. Nestas linguagens, a memória serve como um

elemento de estado global da máquina, exatamente como o banco de registradores. O banco de registradores é eliminado nas arquiteturas *dataflow* pois todos os operandos de entrada das instruções são enviados diretamente por outras instruções. É claro que existem estruturas de armazenamento para operandos destinados às instruções. No entanto estas estruturas não têm função de estado global, servindo apenas para guardar valores que serão consumidos durante a execução. Para que não haja problemas com acessos a memória é preciso ordenar os acessos de acordo com a ordem do programa para garantir que a semântica seja mantida. Outra alternativa é utilizar a memória para guardar informação que seja do contexto local das instruções e passar estes valores adiante na forma de operandos.

Capítulo 3

Trabalhos Relacionados

Neste capítulo são apresentados os trabalhos em estado-da-arte relacionados com este trabalho.

3.1 WaveScalar

As primeiras máquinas *dataflow* [8, 26–29, 31–33] precisavam de linguagens funcionais [30, 36–41] que atendessem o paradigma do *dataflow*. O problema é que essas máquinas eram sensíveis aos efeitos colaterais. Diz-se que uma função ou expressão tem um efeito colateral quando, além de retornar um resultado, ela altera o estado da máquina, por exemplo, alteração do valor de uma variável global, impressão de dados na tela ou em arquivo, escrita em regiões de memória visíveis por outras funções ou expressões ou o levantamento de exceções. Linguagens imperativas (como C, C++, Pascal ou Java) são conhecidas por produzirem efeitos colaterais frequentemente, enquanto linguagens funcionais raramente apresentam estes efeitos.

Um exemplo simples de como efeitos colaterais podem afetar a semântica de programas executados em *dataflow* é o acesso à memória. Operações de armazenamento em memória não produzem operandos, mas duas operações deste tipo podem fazer acesso ao mesmo endereço e a ordem de execução precisa ser respeitada para manter a semântica do programa. Neste caso, torna-se necessário inserir arestas no grafo *dataflow* para indicar as dependências entre tais operações de memória e as demais instruções.

A arquitetura WaveScalar [10, 34] foi criada para tratar este problema, sendo a primeira arquitetura *dataflow* compatível com as linguagens que se baseiam na premissa de que a ordenação das operações de memória imposta pelo programa é respeitada. A ordem esperada dos acessos à memória é verificada em tempo de compilação com o uso de *Wave-ordering annotations*. Portanto, embora a execução seja guiada pelo fluxo de dados, os acessos à memória seguem a ordem do programa para preservar a sua semântica.

3.1.1 Ondas e *Wave-ordering Annotations*

As ondas (*waves*) são fragmentos acíclicos e conexos do grafo de fluxo de controle com uma única entrada, estendendo os hiper-blocos, pois também podem possuir junções. Durante o processo de compilação todas as operações de memória recebem uma chave $< P, C, S >$ (Predecessor, Corrente e Sucessor), permitindo que o subsistema de memória estabeleça uma cadeia conectando-as em uma onda (de um laço). Uma requisição de acesso à memória só pode ser atendida se todas as requisições anteriores na cadeia e todas as ondas anteriores já tiverem sido executadas. Quando uma operação de memória é a primeira ou a última de uma ondas, P=“.” e S=“.”, respectivamente (o coringa “.” denota operação inexistente). Operações que ocorrem antes e depois de um bloco de desvio têm S=“?” e P=“?” (o coringa “?” denota operação desconhecida). Se existem operações de memória em um dos ramos de um desvio, não é possível estabelecer uma cadeia entre as operações que se encontram antes e depois deste bloco. Para resolver este problema, a instrução `MemNop` é inserida neste caminho. A Figura 3.1 mostra um trecho de código com um bloco `IF-THEN-ELSE` (a) e o grafo *dataflow* correspondente (b). As *Wave-ordering annotations* também são exibidas, com as linhas pontilhadas indicado a cadeia formada por elas.

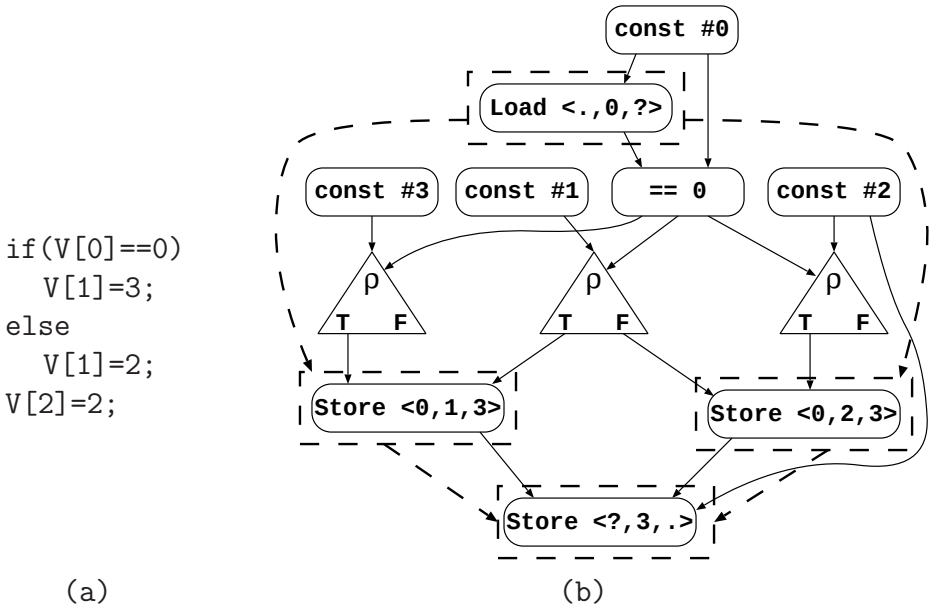

Figura 3.1: Wave-ordering Annotations. O quadro A mostra um trecho de código de alto nível e o quadro B mostra o grafo *dataflow* associado. Nas operações de memória é possível observar as anotações de memória. As arestas tracejadas denotam a ordenação descrita pelas anotações.

A *Wave-ordered memory* incorpora um mecanismo de desambiguação de memória em tempo de execução dentro de cada onda. Ele tira vantagem do fato de que algumas vezes o endereço de um *Store* está disponível antes do dado. Quando isto

ocorre, o mecanismo permite que o subsistema de memória continue atendendo requisições para outros endereços de forma segura. Os *Stores* são quebrados em duas requisições: *Store-Address-Request* e *Store-Data-Request*. *Store-Address-Requests* recebidos sem o *Store-Data-Request* correspondente são inseridos em uma fila de *partial Store* que irá armazenas outras requisições para o mesmo endereço até que o *Store-Data-Request* seja recebido, permitindo que toda a fila seja processada rapidamente. Esta técnica é chamada de *Decoupled Stores* e proporciona um aumento médio de de 30% no paralelismo no acesso à memória [42].

3.1.2 A WaveCache

A arquitetura *WaveScalar* é composta por um conjunto de elementos de processamento (*EPs*) idênticos, o *hardware* da *Wave-ordered memory* e uma rede hierárquica para suportar a comunicação. O bloco de construção da *WaveCache*, é o *Cluster*, que possui uma *cache* L1, o *StoreBuffer* que faz a interface com a *Wave-ordered memory*, e um *Switch* que provém comunicação inter e intra-*Cluster*. Cada *Cluster* tem quatro *Domains*, que por sua vez possuem oito elementos de processamento agrupados em *Pods* de dois *EPs* cada. Os *Clusters* são replicados no *die*, formando uma matriz que é conectada à *cache* L2 em suas bordas. A Figura 3.2 (reproduzida com permissão do autor, de [42]) mostra uma visão geral da *WaveCache*.

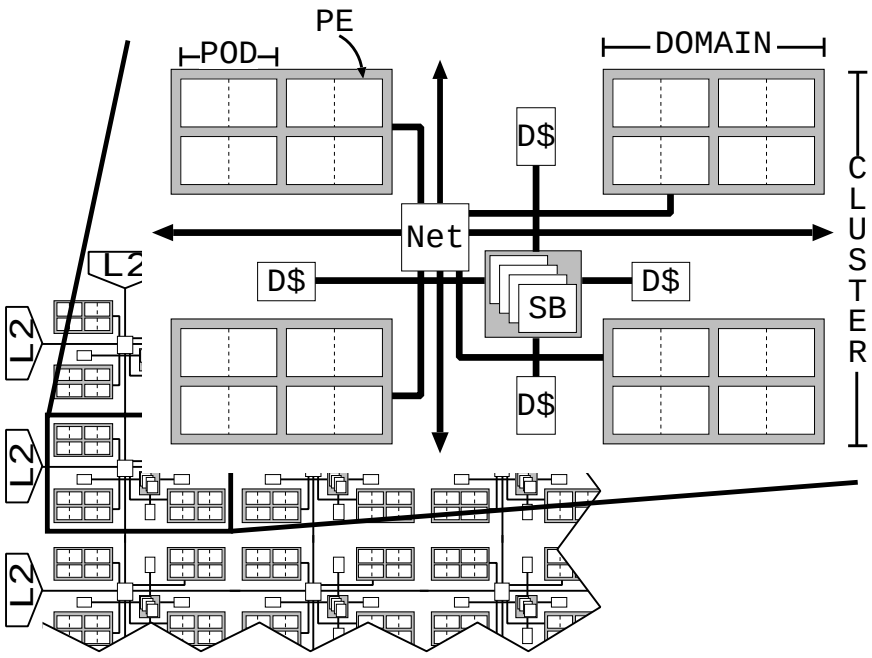

Figura 3.2: A arquitetura WaveCache, composta por diversos elementos de processamento conectados em uma rede hierárquica, além do *hardware* da *Wave-ordered memory*.

Os *EPs* implementam a regra de disparo *Dataflow* e a execução de instruções. Cada *EP* tem a sua *ALU*, estruturas de memória para armazenar operandos, lógica de controle de execução e comunicação e um *buffer* de instruções. Quando um programa é executado no *WaveScalar*, múltiplas instruções são mapeadas em um mesmo *EP*, segundo um algoritmo de *placement*. Conforme a evolução da execução do programa, algumas instruções tornam-se desnecessárias e são substituídas por outras. A regra de disparo garante que uma instrução é executada quando todos os seus operandos de entrada estão disponíveis. Cada *EP* possui também uma *Matching Table* que armazena operandos destinados à instruções mapeadas naquele *EP*, até que as mesmas estejam prontas para serem disparadas. Quando isto ocorre, tais operandos são consumidos, produzindo resultados que serão enviados para outras instruções (em *EPs* remotos ou no *EP* local). O *EP* possui um *pipeline* de cinco estágios, com redes de *bypass* que permitem a execução de instruções dependentes no mesmo *EP*. A Figura 3.3 mostra a arquitetura de um *EP*, destacando os estágios de seu *pipeline*, descritos a seguir:

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

Casamento de tag dos operandos (Match): Os operandos são colocados em uma tabela chamada *matching table*, onde suas *tags* são verificadas em busca de um casamento de operandos para uma mesma instrução e onda. Quando uma instrução já possui todos os seus operandos disponíveis é movida para uma fila de instruções prontas para execução, chamada *scheduling queue*. O casamento também pode ocorrer especulativamente, quando o *EP* supõe que alguma instrução que está executando localmente produzirá operandos para uma outra, também local.

Despacho: O *EP* seleciona uma instrução da *scheduling queue*, lê seus operandos da *matching table* e os envia para o estágio de execução.

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

Saída: Os resultados da instrução são enviados pelo barramento de saída para o próprio *EP* ou outro remoto. É feita a difusão da informação pelo barramento que conecta os *EPs* em um *Domain*, usando um protocolo de transmissão ACK/NACK.

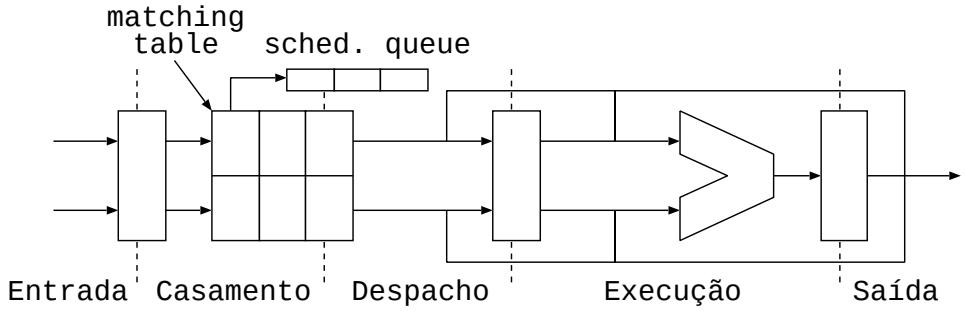

Figura 3.3: Um Elemento de Processamento, definido em um *pipeline* de cinco estágios.

Cada *Cluster* possui um *StoreBuffer* (*SB*) responsável pela ordenação de algumas onda na memória. O **Mapa de Ondas** armazena a informação de qual *StoreBuffer* possui a custódia de quais ondas. Este mapa é armazenado na memória principal e uma visão única das linhas usadas em cada *SB* é garantida pelo protocolo de coerência de *cache*. Requisições de acesso à memória são enviadas dos *EPs* aos seus *StoreBuffers* locais e roteadas, se necessário, para o *SB* remoto responsável pela onda associada à requisição. Esta é inserida em uma lista de requisições para àquela onda e será executada de acordo com o mecanismo de ordenação de operações de memória.

3.2 Arquiteturas *Dataflow* Hibridas e Técnicas, Modelos e Tecnologias em Estado-da-arte para Programação Paralela

A maquina SDF [43] pode ser considerada uma máquina *dataflow* híbrida, pois ela permite a compilação de blocos que serão executadas em uma máquina de Von Neumann, mas que serão disparados segundo o modelo *dataflow*. Por outro lado, a arquitetura TRIPS [44] é ortogonal à SDF. A execução de blocos é disparada segundo o modelo de Von Neumann, mas, dentro de cada bloco, a execução segue o modelo *dataflow*. A execução de programas no modelo TRIPS requer um *hardware* dataflow.

A BMDFM [45] é uma máquina virtual híbrida que suporta a descrição de programas tanto em granularidade fina quanto grossa. Código de granularidade fina é escrito na linguagem nativa da máquina virtual, uma linguagem funcional baseada em LISP. As instruções da máquina virtual podem ser alternadas com instruções personalizadas, escritas em C, como as super-instruções da *Trebuchet*. Segundo a terminologia do BMDFM, as instruções personalizadas são chamadas de instruções

de granularidade grossa. A BMBFM usa diversos *daemons* correspondentes aos diferentes componentes do escalonador dinâmica. O uso de uma linguagem funcional para descrição de programas para a BMDFM é um ponto negativo do trabalho.

O *Program Demultiplexing* [7] é um paradigma de execução onde métodos ou funções são demultiplexados para serem executados concorrentemente com o resto do programa, de acordo com o modelo *dataflow*. O código fonte das aplicações é modificado para permitir a execução de funções em outros núcleos de processamento antes do seu ponto de chamada. Sendo assim, resultados são produzidos e recebidos pelo fluxo de controle principal para serem utilizados quando o ponto de chamada for atingido. Os operandos de entrada (conjunto de leitura) podem ser especulados para permitir a execução demultiplexada com maior antecedência. A implementação demanda mudanças no protocolo de coerência de cache, estruturas adicionais para armazenar resultados de execuções especulativas, bem como ferramentas para selecionar e demultiplexar os métodos. Embora seja uma solução interessante, ela depende, em muitos casos, da especulação para conseguir demultiplexar os métodos com antecedência suficiente para que os resultados possam estar prontos nos pontos de chamada. Além disso a dependência de suporte de hardware faz com que este método não seja uma realidade para os usuários em um futuro próximo.

O DDMCPP é um projeto que se baseia em anotação de código para paralelização [46]. O DDMCPP é um pré-processador para o modelo *Data Driven Multithreading* [47], que suporta *dataflow* dinâmico. O modelo provê unidades de sincronização de *threads*, que também são responsáveis pelo escalonamento. O DMCPP disponibiliza um conjunto de *pragmas* para a definição de *threads* e a descrição dos operandos trocados entre elas, como no modelo *dataflow*, além de *pragmas* para descrição de laços do tipo *for* e operações de redução. O pré-processador transforma o código para incluir a troca de operandos entre *threads* e operações de sincronização. No entanto, este modelo não é muito flexível. Como o DDMCPP não é um compilador e o ambiente de execução não é uma arquitetura *dataflow*, não há a descrição de laços usando instruções de granularidade fina em fluxo de dados (como uso de instruções de incremento de rótulo de iteração). O único tipo de laço disponível é do tipo *for*, e o *dataflow* dinâmico é garantido com a criação de novos contextos no ambiente de execução (semelhante a uma chamada de função). A criação de contextos é incluída pelo pré-processador. Além disto, não é feita uma comparação de desempenho com outras ferramentas consagradas de programação paralela.

O HMPP [48] é um ambiente heterogêneo de programação paralela para *multicores* que permite a integração de diferentes aceleradores em *hardware* de maneira simples e preservando código legado. É provido um ambiente de execução, um conjunto de diretivas de compilação, chamadas de *codelets*, que podem ser executadas em *GPFPU*, *FPGAS*, máquinas remotas (com *MPI*) ou o *CPU* local. Os *Codelets*

são funções puras, sem efeitos colaterais (não afetam a memória global ou arquivos). Múltiplos *Codelets*, cada um desenvolvido para um tipo de *hardware* diferente, podem existir e o ambiente de execução vai escolher qual *Codelet* será executado, de acordo com a disponibilidade dos recursos e com as diretivas de compilação previamente definidas. O ambiente de execução também é responsável pelas transferências de dados de/para os componentes de *hardware* envolvidos na computação. A ideia de um ambiente heterogêneo é interessante, no entanto, o problema do modelo de programação paralela ainda precisa ser tratado. A programação continua sendo feita com os modelos tradicionais.

O sistema *Galois* [49–51] é um sistema de paralelização otimista baseado em objetos, para aplicações irregulares. Ele é composto por: *(i)* construções sintáticas para empacotar paralelismo otimista com iterações sobre conjuntos ordenados e desordenados, *(ii)* um sistema de execução para detectar acessos inseguros à memória compartilhada e executar as operações de recuperação necessárias e *(iii)* verificações de métodos em bibliotecas de classes. Em vez de rastrear os endereços acessados pelo código otimista, o *Galois* rastreia violações de semântica em alto nível em tipos de dados abstratos. Para cada método que realiza acessos à memória compartilhada, o programador precisa descrever quais métodos (e em quais circunstâncias) podem ser executados de forma comutativa sem conflitos. O *Galois* também introduz alternativas às verificações de comutatividade, pois estas podem ser custosas [50]. Dados compartilhados são particionados e atribuídos aos diferentes núcleos de processamento e o sistema monitora se partições estão sendo "tocadas" por *threads* concorrentes (o que acarretaria em um conflito). Indiferente do método de detecção usado, o programador precisa descrever um método inverso para cada método que acessa objetos compartilhados. Os métodos inversos são executados no caso de um *rollback*. O sistema de execução é encarregado de detectar conflitos, chamar os métodos inversos e comandar a re-execução. Embora seja um sistema bastante inovador, o *Galois* expõe ao programador alguns detalhes do modelo de especulação, ao demandar a descrição dos métodos inversos e das relações de comutatividade. Além disto, foi mostrado que especulação baseada em verificação de comutatividade pode ser custosa [50]. Foi então sugerido um mecanismo de partição de dados e a modificação do mecanismo de especulação para controlar o acesso a blocos de memória, invés de objetos em alto nível. No entanto, os métodos inversos para fazer operações de *rollback* continuam existindo.

3.3 Memórias Transacionais

O termo *Memória Transacional* (MT) foi cunhado por Herlihy e Moss [4] como "uma nova arquitetura de multiprocessador com o propósito de tornar a sincronização livre

de bloqueios tão eficiente (e fácil de usar) quanto as técnicas convencionais, baseadas em exclusão mútua". A motivação era evitar a complexidade da programação baseada em bloqueios. O advento dos *CMPs* aumentou a demanda por modelos de programação paralela mais fáceis, tornado o assunto popular.

Uma transação é um bloco código, cuja execução precisa parecer atômica. Transações podem executar especulativamente sem bloqueios na MT. Uma estrutura mantém um histórico das alterações feitas na memória e no banco de registradores. Caso haja um conflito entre duas transações uma delas é abortada e reexecutada. Se uma transação termina sem conflitos, o *commit* será feito. Existem quatro problemas clássicos que devem ser considerados ao se criar uma solução de Memória Transacional: versionamento de dados, detecção de conflitos, aninhamento e virtualização.

No versionamento de dados adiantado um histórico mantém um *backup* dos blocos de memória e registradores modificados pela transação. Esta informação é usada para restaurar as modificações feitas por uma transação caso um perigo seja detectado, e é descartada quando a transação termina com sucesso. No versionamento tardio, um *buffer* de escrita armazena as alterações feitas na memória e registradores. Tais alterações só serão aplicadas mediante um *commit* e descartadas por *rollbacks*. A detecção de conflitos também pode ser adiantada ou tardia. Na primeira, os conflitos são detectados assim que ocorrem e, na última, mediante o término da transação.

Transações são ditas aninhadas quando a execução de uma ocorre dentro da outra. Elas podem ser tratadas como uma única transação (*flattening*), ou separadamente (aninhamento fechado e aberto [52, 53]). Além disto, sistemas de Memória Transacional devem prover mecanismos de virtualização para garantir a execução correta de programas, mesmo quando transações excedem o *quantum* do escalonador de processos, a capacidade das *caches* e memória ou o número de níveis de aninhamento permitidos pelo *hardware* [54].

Implementações de Memórias Transacionais em *hardware* [4, 54, 55], proporcionam um maior desempenho, mas soluções completas para virtualização e aninhamento podem aumentar a complexidade do projeto e torná-lo mais caro. Implementações em *software* [56–58] permitem soluções mais complexas e flexíveis com menor desempenho. Implementações híbridas [58–60] tentam obter o melhor dos dois mundos.

Capítulo 4

TALM (*TALM is an Architecture and Language for Multi-threading*)

A maioria dos processadores segue o modelo de Von Neumann, onde a execução é, fundamentalmente, guiada pelo fluxo de controle. De fato, os processadores modernos possuem mecanismos de execução de instruções fora-de-ordem com base no fluxo de dados, como o algoritmo de Tomasulo [35]. Tais alternativas extraem paralelismo em nível de instrução, limitado pelo fato das instruções continuarem sendo emitidas sequencialmente.

O objetivo deste trabalho é aproveitar melhor as possibilidades disponíveis no modelo *dataflow*, mantendo ainda a compatibilidade com as arquiteturas correntes. Para tal, foi desenvolvido um modelo de execução de programas baseado em *dataflow*: o TALM (*TALM is an Architecture and Language for Multi-threading*) [14, 15, 17]. O modelo disponibiliza uma arquitetura base para um ambiente de execução guiada por fluxo de dados, com um conjunto de instruções que permite descrever o programa e o seu controle, construções de execução condicional e laços de repetição, em *dataflow*. No entanto, o diferencial do TALM é permitir a descrição de super-instruções customizadas pelo programador, cujo comportamento é descrito em linguagens imperativas. O modelo foi concebido para existir dentro do contexto de uma Máquina de Von Neumann e pode ser implementado como um emulador, uma máquina virtual ou um sistema de execução que crie a abstração de uma arquitetura *dataflow*, dentro de uma máquina de Von Neumann. O modelo dá ao programador a liberdade de escolher qual a granularidade das super-instruções e qual trabalho vai ser feito por elas. Aplicações podem ser compiladas para a linguagem de montagem do TALM e serem executadas em paralelo.

Neste capítulo são apresentadas as ideias principais do TALM: um modelo con-

cebido para tirar proveito de execução *dataflow* com instruções de granularidade customizável, em máquinas de Von Neumann

4.1 A Arquitetura TALM

A arquitetura base do TALM é exibida na Figura 4.1. Ela é composta de um conjunto de Elementos de Processamento (EPs) idênticos, conectados em uma rede. Os EPs são responsáveis pela interpretação de instruções e execução de acordo com as regras de disparo guiada pelo fluxo de dados. Cada EP possui um *buffer* de comunicação para receber mensagens com operandos enviados por outros EPs na rede. Quando uma aplicação é executada no TALM, suas instruções precisam ser distribuídas entre os EPs. Uma lista de instruções guarda a informação relacionada a cada instrução estática mapeada em um EP. Além disso, cada instrução possui uma lista de operandos relacionados às suas instâncias dinâmicas. O casamento de operandos com mesmo rótulo, isto é, pertencentes à mesma instância de uma instrução, é feito nesta lista. A instrução e os operandos relativos ao casamento são enviados para uma fila de prontos para serem interpretados e executados. A topologia da rede de interconexão depende da implementação do modelo.

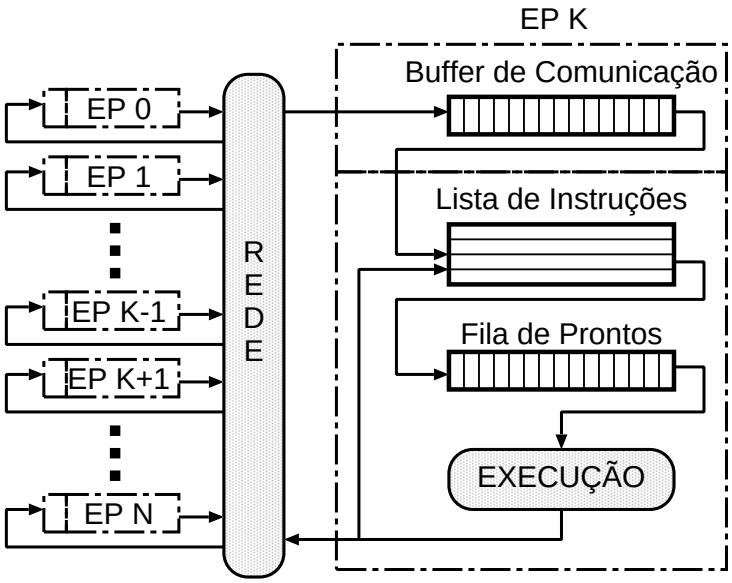

Figura 4.1: A arquitetura TALM, composta por um conjunto de elementos de processamento idênticos, ligados em uma rede. Cada elemento de processamento possui *buffers* para armazenamento de mensagens com operandos, uma lista de instruções, uma fila de instruções prontas para execução e a lógica de execução.

Os elementos de processamento são responsáveis pela execução de instruções segundo o modelo *dataflow*. Os passos de execução de um EP, independente da implementação, são os seguintes:

1. Verificação de novas mensagens com operandos.

2. Inclusão de operandos recebidos nas listas de operandos de suas respectivas instruções de destino. Cada operando será inserido em um linha correspondente à instância dinâmica da instrução.

3. Verificação se houve algum casamento de operandos, ou seja, se para alguma instância de uma instrução todos os operandos estão disponíveis. Isto ocorre quando uma linha da lista de operandos está completa.

4. Envio dos operandos "casados"para uma fila de execução.

5. Execução de uma instrução da fila de execução.

6. Envio dos resultados da execução paras as instruções destino.

Como não existe contador de programa em máquinas Dataflow, é necessário achar uma forma de determinar quando a execução de um programa chegou ao seu fim. No TALM isto é feito com um algoritmo distribuído de detecção de terminação global, que deve ser adaptado de acordo com a implantação da máquina virtual. Detalhes do algoritmo distribuído de detecção de terminação global da *Trebuchet* são discutidos profundamente na Seção 6.1.4.

4.2 O Conjunto de Instruções

O primeiro passo para projetar uma arquitetura é a concepção do seu conjunto de instruções. Neste caso, é desejável que o conjunto de instruções seja extensível, através do uso de super-instruções. Nesta seção é descrita a arquitetura do conjunto de instruções do TALM e é feita uma descrição da linguagem de montagem com seus mnemônicos e operandos.

4.2.1 O Formato das Instruções

Como super-instruções são blocos de código escritos pelo programador, cada super-instrução pode ter um número distinto de operandos de entrada saída. Sendo assim, o formato das instruções deve permitir a definição de instruções com esta característica.

A Figura 4.2 apresenta o formato de instrução do TALM. O formato da instrução foi definido para facilitar a implantação; o tamanho de todos os campos pode ser aumentado, se necessário. Os primeiros 32 bits são obrigatórios para toda instrução. Eles carregam o `Opcode` e o número de operandos de entrada e saída requeridos pela instrução (`#Origens` e `#Resultados`, respectivamente). O `Opcode` tem 22 bits pois

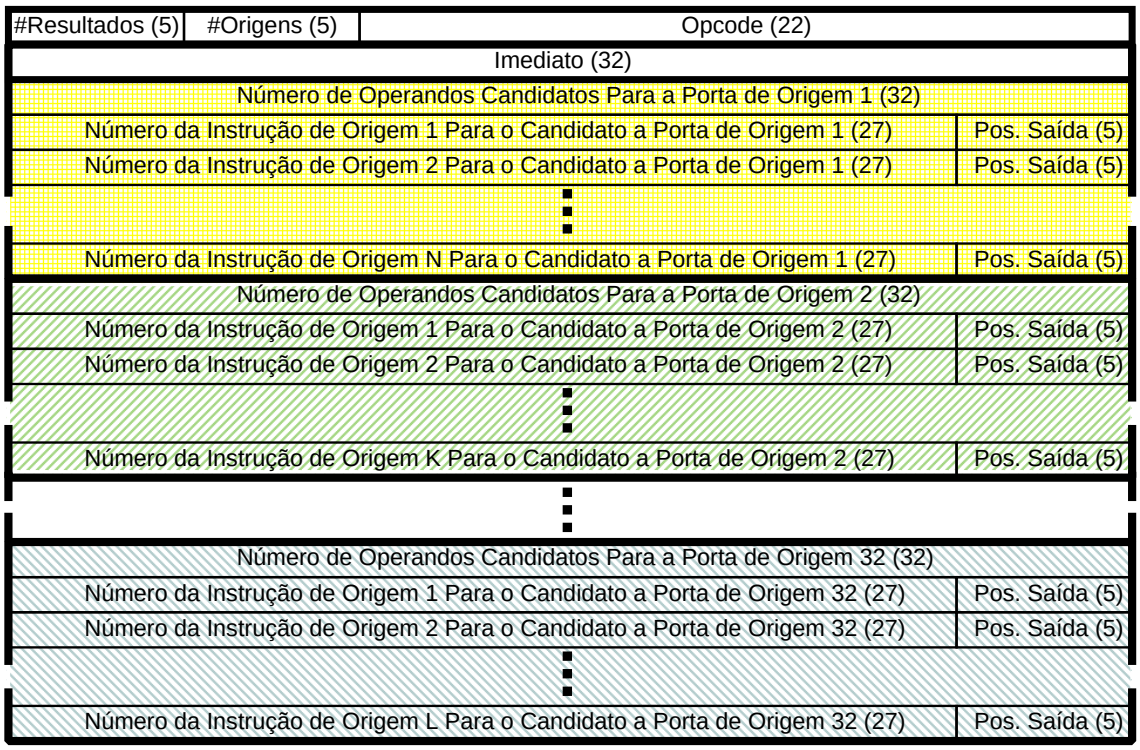

Figura 4.2: O formato de uma instrução descreve a operação executada pela mesma, um operando imediato (opcional) e uma lista de operandos candidatos para até 32 portas de entrada.

os campos `#Origens` e `#Resultados` possuem 5 bits cada. Isso permite uma rápida indexação de instruções.

O campo `#Origens` indica o número de operandos de entrada que uma instrução pode ter. Como esse campo possui 5 bits, é possível que uma instrução receba até 32 operandos. O mesmo é válido para o número de operandos de saída (o campo `#Resultados` também possui 5 bits). Os operandos são enviados e recebidos por portas de entrada e de saída das instruções. Existem, portanto, um máximo de 32 portas de entrada e 32 de saída.

O 32 bits seguintes são opcionais. Eles carregam um operando `Imediato`, caso seja necessário pela instrução. Uma política de "*compile once, run anywhere*" é adotada. Sendo assim, o tamanho do imediato é limitado a 32 bits para manter a compatibilidade com máquinas hospedeiras de 32-bits.

O TALM suporta execução condicional permitindo que cada operando possua múltiplas instruções como possíveis origens (alternativamente, seria possível especificar as diferentes instruções de destino para cada operando de saída). Note que, durante a execução, apenas um caminho será escolhido. Sendo assim, um operando é dito disponível assim que a porta de entrada é preenchida por **quaisquer** de suas possíveis origens. Para cada porta de entrada é especificado o número de origens

possíveis. É usado um campos de 32 bits para garantir o alinhamento. Em seguida, para cada possível origem são dados o endereço da instrução (com 27 bits) e a porta de saída na instrução indicada (com 5 bits). Em outras palavras, um valor $\langle i|p \rangle$ nesses dois campos indicam que um operando pode ser recebido pela *p-ésima* porta de saída da *i-ésima* instrução. Note que existem até 32 portas de entrada (`#Origens` tem 5 bits) e, portanto, 32 grupos de candidatos. Os 27 bits para o endereçamento de instruções limitam o tamanho máximo de código em 2^{27} instruções; este não é um limite rígido, dado que está sendo fixado apenas para facilitar a implementação da máquina virtual.

4.2.2 A Linguagem de Montagem do TALM

O conjunto de instruções básico fornece as operações lógicas e aritméticas mais comuns, como `add`, `sub`, `and`, `or`, `mult`, `div`, além de suas variações com imediato. Além disto, existem as instruções que implementam controle: desvios condicionais, laços, e chamadas de função. Mais detalhes sobre todas estas instruções são explicados à seguir.

4.2.2.1 Instruções Lógicas e Aritméticas sem Imediato

A sintaxe das instruções lógicas e aritméticas sem imediato é a seguinte:

```
<mnemônico> <nome>, <op1>, <op2>
```

A descrição de cada campo da instrução deste tipo de instrução é feita a seguir:

- O mnemônico da instrução indica a operação a ser realizada.
- `nome`: é o identificador da instância da instrução. Esse nome é usado para referenciar o operando de saída.
- `op1`: indica o operando usado na primeira de entrada.
- `op2`: indica o operando usado na segunda porta de entrada.

4.2.2.2 Instruções Lógicas e Aritméticas com Imediato

A sintaxe das instruções lógicas e aritméticas com imediato é a seguinte:

```
<mnemônico> <nome>, <op>, <imediato>
```

A descrição de cada campo da instrução deste tipo de instrução é feita a seguir:

- O mnemônico da instrução indica a operação a ser realizada.

- `nome`: é o identificador da instância da instrução. Esse nome é usado para referenciar o operando de saída.
- `op`: indica o operando usado na primeira porta de entrada.
- `imediato`: indica o imediato usado na operação.

4.2.2.3 Portas de Entrada com Múltiplos Operandos Candidatos

Uma instrução pode receber em uma porta de entrada um operando proveniente de múltiplas fontes. Na lista de entrada, cada elemento preenche uma porta de entrada da instrução. Como, neste caso, o operando pode vir de diferentes fontes, dependendo do fluxo de dados, é preciso descrever esta relação na lista de entrada. Colchetes são utilizados para descrever grupos de candidatos para cada porta. Como a seguir:

```
[candidato0, ..., candidatoN]
```

É bom lembrar que cada instrução tem, no máximo 32 portas de entrada e para cada porta é possível ter até 2^{32} operandos candidatos.

4.2.2.4 Desvios Condicionais

Conforme explicado na Seção 2.2, desvios de controle devem mudar o fluxo de dados de acordo com o caminho selecionado. Isto é feito pela por instruções de desvio, chamadas de `steer`, no TALM, que recebem um operando O e um booleano seletor S, fazendo com que O seja encaminhado para um de dois possíveis caminhos no grafo de fluxo de dados de acordo com o valor de S.

A sintaxe da instrução `steer` é a seguinte:

```
steer <nome>, <seletor>, <valor>
```

A descrição de cada campo da instrução `steer` é feita a seguir:

- `steer`: é o mnemônico da instrução.
- `nome`: é o identificador da instância da instrução. Esse nome é usado para referenciar o operando de saída.
- `seletor`: indica o operando seletor (um dos operandos de entrada da instrução).
- `valor`: indica o operando com o valor a ser encaminhado para um dos 2 caminhos no grafo, de acordo com o seletor (é o outro operando de entrada da instrução).

Uma instrução `steer` tem duas portas de saída. Uma delas é usada para enviar o operando caso o valor do seletor for verdadeiro e a outra e usada para enviar o mesmo operando se o seletor for falso. Se uma instrução i for usar como entrada um operando produzido por uma instrução `steer` com nome `desv`, deve especificar o operando com uma das seguintes notações:

- `desv.t`: o operando será enviado para i caso o seletor da instrução `desv` seja **verdadeiro**.
- `desv.f`: o operando será enviado para i caso o seletor da instrução `desv` seja **falso**.

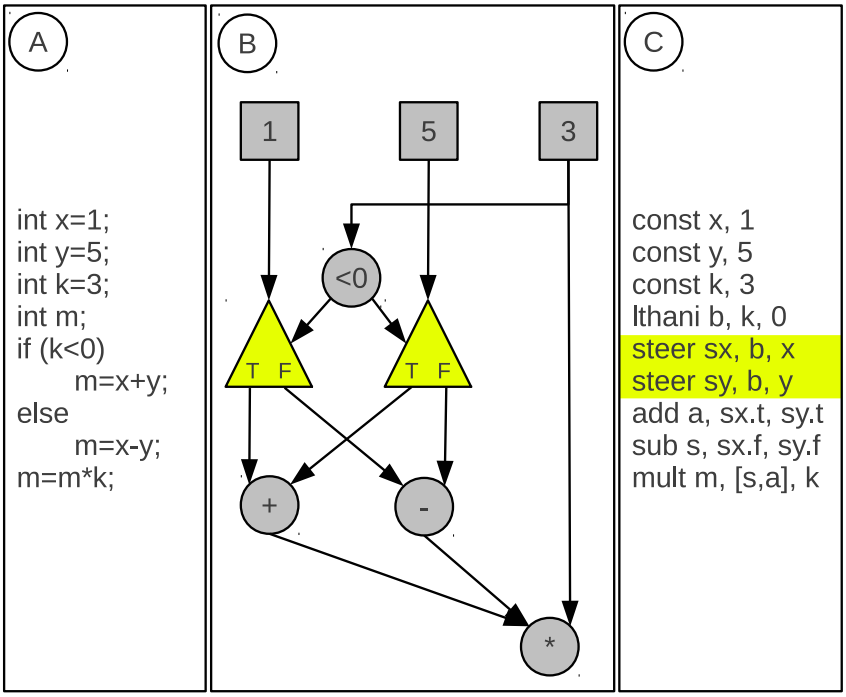

Figura 4.3: Exemplo de desvios no TALM. O quadro A mostra um trecho de código em alto nível. O quadro B mostra o grafo *dataflow* associado, onde as instruções de `steer` são representadas como triângulos. O quadro C mostra o código de montagem do TALM.

A Figura 4.3 mostra um exemplo do uso de desvios no TALM. No quadro A é exibido o código de alto nível de uma construção do tipo *IF-THEN-ELSE*. O quadro B mostra o grafo de fluxo de dados associado, onde as instruções `steer` são representadas por triângulos. No quadro C é exibido o código de montagem do TALM, onde observa-se o uso das instruções de `steer`. Neste exemplo, o booleano seletor é produzido pela instrução `lthani`, com nome `b`. Repare que `b` é usado nas duas instruções de `steer`, com nomes `sx` e `sy`, que são usados para encaminhar `x` e `y`, respectivamente, para um dos ramos do desvio. Note que a instrução `add` (de nome

a) usa `sx` e `sy` e executa caso `b` seja verdadeiro, pois especifica `sx.t` e `sy.t` como entradas. Analogamente, a instrução `sub` (com nome `s`) executa quando `b` é falso, pois especifica `sx.f` e `sy.f` como entradas. Além disto, repare como a instrução `mult` (de nome `m`) define os operandos provenientes de `s` e `a` como candidatos para operandos da primeira porta de entrada (`m` declara `[s,a]` como operandos para a primeira porta). O operando para a segunda porta de entrada de `m` é `k`, uma constante.

4.2.2.5 Laços de Repetição

O TALM adota o modelo *dataflow* dinâmico. Quando um operando atinge a iteração seguinte o seu rótulo é incrementado para casar com o dos outros operandos na mesma iteração. A escolha pelo dataflow dinâmico se deve ao fato do objetivo deste trabalho ser o de buscar o paralelismo, além da implementação do modelo ser baseada em software, o que permite uma maior flexibilidade. Conforme explicado na Seção 2.2 é necessária uma instrução especial para separar diferentes iterações em diferentes laços. Instruções irão executar quando elas tiverem um conjunto de operandos disponíveis **com o mesmo rótulo**. Esta instrução é chamada, no TALM, de *Increment Iteration Tag* (`inctag`).

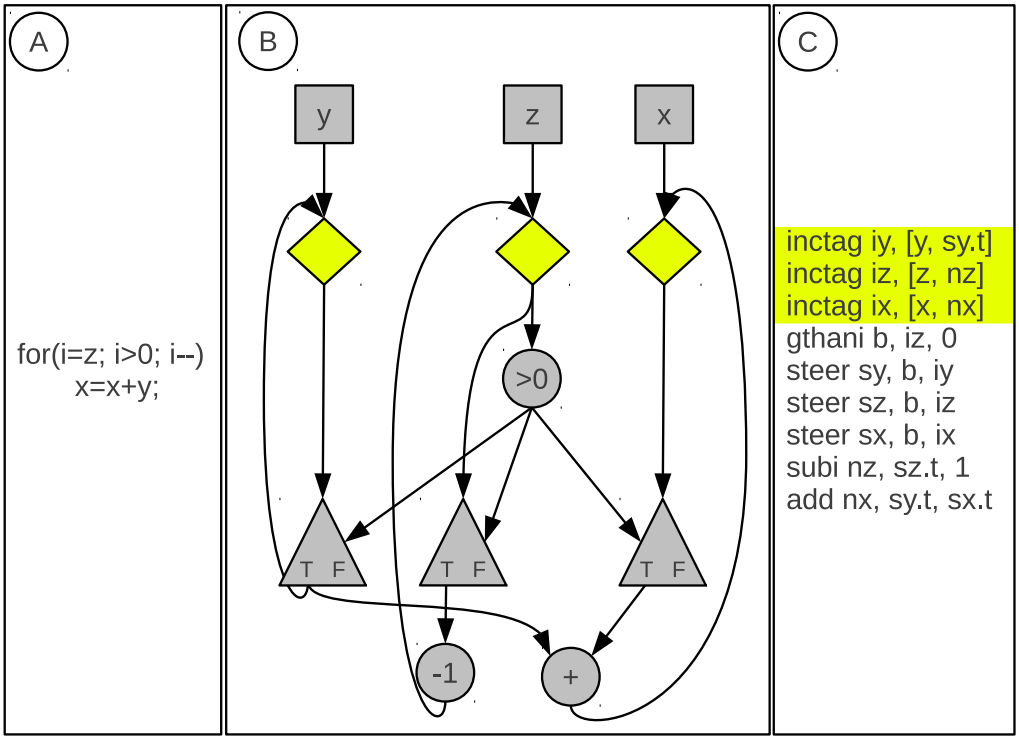

Figura 4.4: Exemplo de laços no TALM. O quadro *A* mostra um trecho de código em alto nível. O quadro *B* mostra o grafo *dataflow* associado, onde as instruções de `inctag` são representadas como losangos. O quadro *C* mostra o código de montagem do TALM.

A sintaxe da instrução `inctag` é a seguinte:

```
inctag <nome>, <operando>
```

A descrição de cada campo da instrução `inctag` é feita a seguir:

- `inctag`: é o mnemônico da instrução.
- `nome`: é o identificador da instância da instrução. Esse nome é usado para referenciar o operando de saída.
- `operando`: indica o operando de entrada cujo rótulo de iteração será incrementado.

A Figura 4.4 mostra um exemplo de um laço de repetição em *dataflow* para ilustrar o uso das instruções `inctag`. O quadro A mostra o código de alto nível. O quadro B mostra o grafo *dataflow* da aplicação, onde as instruções `inctag` sãp representadas como losangos. O quadro C mostra o código de montagem do TALM associado ao grafo. Note, por exemplo, como os nós `steer` e `inctag` interagem para compor um laço: sempre que a condição > 0 for verdadeira, o `steer sy` encaminha o seu operando para os nós `iy` e `nx`, `sx` encaminha seu operando para `nx` e `sz` encaminha seu operando para `nz`. O nó `nx` faz a operação $x = x + y$ e encaminha o valor atualizado de x para a iteração seguinte. Já o nó `nz` decrementa o valor de z para ser usado no teste condicional da iteração seguinte. Eventualmente, a condição falha e o laço termina. Repare como todas as instruções de `inctag` possuem dois operandos candidatos para entrada: um deles é sempre proveniente de fora do laço e o outro vem do corpo do laço.

4.2.2.6 Funções

No modelo dataflow, funções são blocos de instruções que podem receber operandos (parâmetros) de diferentes pontos de chamada no código. Esses pontos de chamada são denominados **estáticos**. Além disso, o mesmo ponto de chamada pode ser executado múltiplas vezes, por exemplo, quando existe um laço de repetição, ou em uma função recursiva. Esses pontos de chamada são denominados **dinâmicos**. Uma solução semelhante à dada para o laços pode ser utilizada. Deve haver uma forma de identificar operandos provenientes de diferentes pontos de chamada (estáticos ou dinâmicos) para que o casamento apenas seja feito com operandos associados ao mesmo ponto de chamada. Para tal, o rótulo dos operandos precisa de mais dois itens:

- o *call group* (`GrpCh`) é gerado em tempo de montagem pelo pré-processador, através da macro `callgroup`, e identifica pontos de chamada estáticos diferentes.

- o *call instance number* (`#Chamada`) identifica instâncias distintas de um mesmo ponto de chamada.

O *call group* é um número inteiro armazenado no campo `Imediato` de instruções `callsnd`. O *call instance number* é atualizado dinamicamente, com base em um contador local (`ContCh`) pela instrução `callsnd`. Quando uma função é chamada, instruções de `callsnd` serão usadas para enviar operandos de entrada (ou parâmetros) com o rótulo de `#Chamada` para a função chamada. O rótulo de `#Chamada` do chamador também é enviado à função chamada (com a instrução `retsnd`) para ser usado quando a mesma retorna um valor. Neste caso, o rótulo `#Chamada` no operando de retorno deve ser restaurado para casar como o do chamador. Isto é feito pela instrução `ret`.

A sintaxe da instrução `callsnd` é a seguinte:

```
callsnd <nome>, <operando>, <grupo>
```

A descrição de cada campo da instrução `callsnd` é feita a seguir:

- `callsnd`: é o mnemônico da instrução.
- `nome`: é o identificador da instância da instrução. Este nome é usado para referenciar o operando de saída.
- `operando`: contém o operando a ser enviado como um dos argumentos da função.
- `grupo`: contém o imediato (gerado pela macro `callgroup`) que identifica o ponto de chamada ao qual o argumento pertence.

A sintaxe da instrução `retsnd` é a seguinte:

```
retsnd <nome>, <operando>, <grupo>
```

A descrição de cada campo da instrução `retsnd` é feita a seguir:

- `retsnd`: é o mnemônico da instrução.
- `nome`: é o identificador da instância da instrução. Este nome é usado para referenciar o operando de saída.
- `operando`: contém o operando a ser enviado como *link* de retorno função (contendo o rótulo dos operandos do chamador).
- `grupo`: contém o imediato (gerado pela macro `callgroup`) que identifica o ponto de chamada ao qual o argumento pertence.

A sintaxe da instrução `ret` é a seguinte:

```
ret <nome>, <operando>, <link>
```

A descrição de cada campo da instrução `ret` é feita a seguir:

- `ret`: é o mnemônico da instrução.
- `nome`: é o identificador da instância da instrução. Este nome é usado para referenciar o operando de saída e deve ser o igual ao nome escolhido na macro `callgroup`.
- `operando`: contém o operando de retorno.
- `link`: contém o operando enviado pela instrução *retsnd*.

A Figura 4.5 mostra um exemplo de chamada de funções no TALM. O quadro *A* mostra o código de alto nível onde existem dois pontos de chamada da função `soma` dentro de um laço de repetição. O intuito do exemplo é mostrar o funcionamento de pontos de chamada estáticos (os dois pontos de chamada) e dinâmicos (as múltiplas chamadas no laço). O quadro *B* mostra o grafo de fluxo de dados da aplicação e o quadro *C* mostra o código de montagem associado. Note o uso da macro `callgroup` para criar grupos de chamada referentes aos dois pontos de chamada estáticos (`c1` e `c2`). Repare também como as instruções `callsnd`, `retsnd` e `ret` usam o grupo de chamada para identificar os retornos de cada chamada. Por último, note como as instruções de `inctag ix` e `ik` recebem os valores de retorno das duas chamadas da função. Os pontos de chamada dinâmicos no laço são controlados também pelas instruções de `callsnd` e `retsnd`. A cada nova chamada, o contador de chamadas é incrementado localmente nestas instruções, para que o rótulo da chamada seja distinto em cada nova chamada em um mesmo ponto.

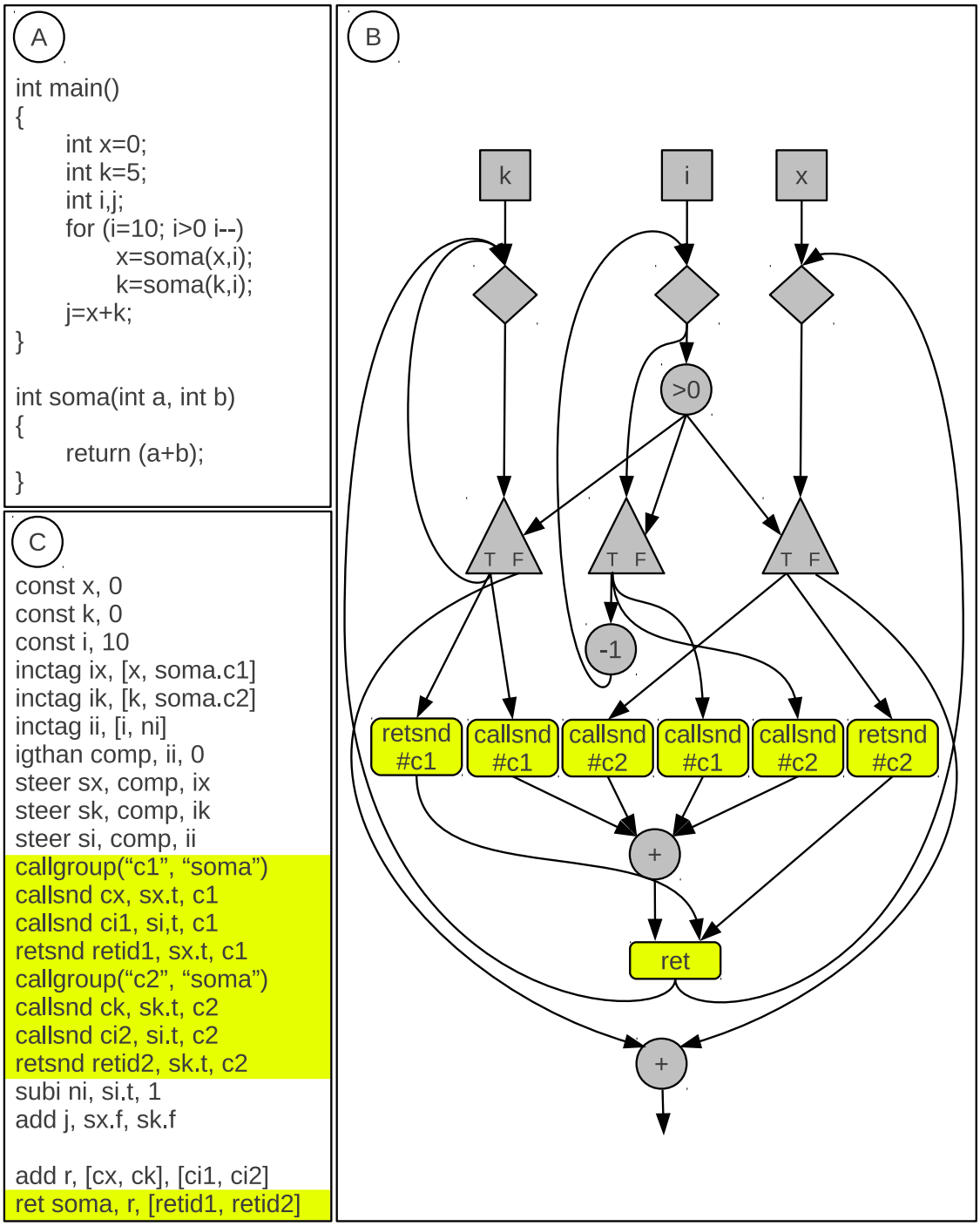

Figura 4.5: Exemplo de funções no TALM. O quadro *A* mostra um trecho de código em alto nível. O quadro *B* mostra o grafo *dataflow* associado, onde é possível observar o uso das instruções `callsnd`, `retsnd` e `ret`. O quadro *C* mostra o código de montagem do TALM.

4.2.2.7 Super-instruções

As super-instruções possuem um papel muito importante no modelo TALM. Elas representam instruções, cujo comportamento é definido pelo programador. No Capítulo 5 é explicado como definir o comportamento de super-instruções na linguagem de alto nível do TALM. No entanto, a linguagem de alto nível apenas descreve quais são os operandos de entrada e saída das super-instruções, além da computação feita pelas super-instruções. Ela não descreve como os operandos de entrada e saída são recebidos ou enviados para as super-instruções, pois isto depende da implementação do modelo TALM, isto é, da arquitetura alvo. No Capítulo 6 é discutido como codificar o comportamento completo de uma super-instrução, incluindo envio e recebimento de operandos para que seja executada pela *Trebuchet*. Já no capítulo 7 é discutido como um compilador transforma o código descrito em alto nível para um código compreendido pela *Trebuchet*.

Super-instruções permitem reduzir a complexidade do grafo de fluxo de dados das aplicações com a definição de trechos de código imperativo para executar trabalhos mais completos que de instruções simples. O grafo de fluxo de dados passa a ser usado para descrever a relação entre essas instruções de granularidade grossa para extrair paralelismo com o modelo *dataflow*.

A sintaxe para a definição de uma super-instrução em linguagem de montagem TALM é:

```
super <nome>, <#super>, <#operandos>, [entradas]
```

Uma descrição de cada campo é feito à seguir:

- `super`: é o mnemônico usado para descrever super-instruções.
- `nome`: é o identificador da instância da super-instrução, usado para referenciar os operandos produzidos por esta super-instrução nas entradas de outras instruções do programa. Por exemplo, se existe uma super instrução chamada `processa`, `processa.i` se refere ao i-ésimo operando produzido por esta super-instrução.
- `#super`: é o número da super-instrução que é associado ao código que implementa o seu comportamento. É possível então ter várias super-instruções com nomes e operandos diferentes, todas associadas à mesma implementação. Cada uma dessas super-instruções descritas no código de montagem é uma instância da super-instrução cujo comportamento é definido pelo usuário.
- `#operandos`: é o número de operandos de saída da super-instrução.

- `entradas`: é uma lista dos operandos de entrada recebidos pela super-instrução.

Existem uma variante da instrução `super` com operando imediato. A `superi` tem sintaxe semelhante à `super`, mas o último argumento (depois da lista de entradas) é um operando imediato, usado para passar o *task_id*, para ser acessado pela função `treb_get_tid()`, conforme explicado na Seção 5.3. Além disso, as variantes `specsuper` e `specsuperi` são descrever super instruções especulativas, respectivamente, sem e com imediato. O suporte à execução especulativa no TALM [16, 17] não é do escopo deste trabalho. No entanto, a ideia geral é explicada no Capítulo 9

Super-instruções também são executadas de acordo com o fluxo de dados. Elas podem receber até 32 operandos de entrada e produzir até 32 operandos de saída. Em instruções simples, como existe apenas 1 operando de saída, o nome da instrução é usado para referenciar o operando produzido pela mesma. A instrução `steer` é a única exceção, pois existem duas portas de saída que serão ativadas de acordo com o booleano seletor. Assim, em uma instrução `steer` de nome `s`, `s.t` e `s.f` são usados para referenciar as saídas de `s` caso o booleano seja, respectivamente, verdadeiro ou falso. No caso de uma super-instrução, o operando de saída é selecionado com o número da porta. Para referenciar o operando da porta de saída `i` ($0 \leq i \leq 31$) de uma super-instrução de nome `s`, usa-se `s.i`.

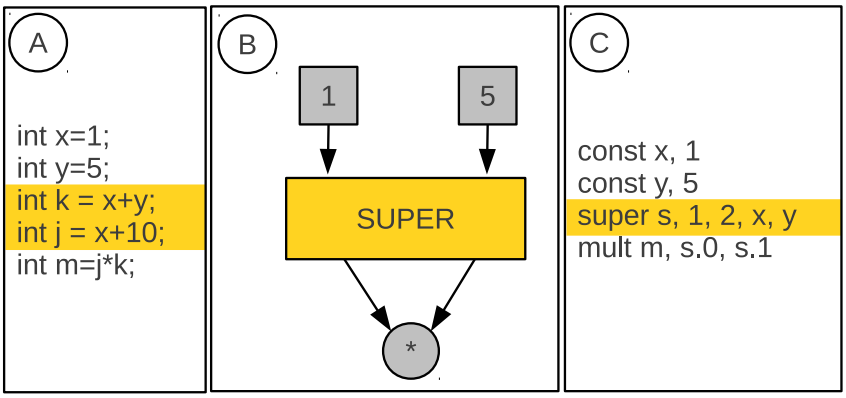

Figura 4.6: Exemplo de super-instruções no TALM. O quadro *A* mostra um trecho de código em alto nível. O quadro *B* mostra o grafo *dataflow* associado, onde a instrução `super` é representadas como um retângulo mais escuro. O quadro *C* mostra o código de montagem do TALM.

A Figura 4.6 mostra um exemplo do uso de super-instruções no TALM. Neste exemplo, para a aplicação cujo código de alto nível é mostrado no quadro *A*, deseja-se criar uma super-instrução que calcule os valores de k e j. O quadro *B* mostra o grafo *dataflow* para a aplicação e o quadro *C* mostra o código de montagem associado. Repare que o código que descreve o comportamento da super-instrução não é descrito

neste exemplo, pois ele depende da implementação do TALM. Mais detalhes sobre este processo são apresentados no Capítulo 6. Note como a instrução super é usada para definir a existência de uma super-instrução com nome `s`, com identificador 1 e dois operandos de saída. Veja como o código de montagem é usado para descrever a relação de dados de `s` com o restante do programa. Repare que os dois operandos de saída de `s` são usados pela instrução de multiplicação `m`, quando a mesma especifica `s.0` e `s.1` como operandos de entrada.

4.2.2.8 Macros e Sequencias de Repetição

Para facilitar a programação com a linguagem de montagem TALM, foram criadas algumas macros. As macros são interpretadas por um pré-processador do montador do TALM. As macros existentes e suas funções são:

- a macro `superinst`, para criação de super-instruções;
- a macro `placeinpe` para fazer alocação de instruções no elementos de processamento da arquitetura TALM;
- a macro `callgroup` para criação de pontos de chamada em funções.

A sintaxe da macro `superinst` é a seguinte:

```
superinst(<alias>, <#super>, <#operandos>, <Especulativa>, [imediato])
```

O campo alias é um novo mnemônico criado para essa super-instrução e associado ao número que representa a super-instrução e permite a posterior criação de suas diversas instâncias de maneira mais inteligível (usando nomes invés de números). O campo `Especulativa` é um booleano que informa de a super-instrução é especulativa ou não. O campo `imediato`, opcional (*False* é o valor padrão), é um booleano que informa se a super-instrução tem imediatos. O significado dos demais campos é o mesmo dos campos usado na instrução `super`. Quando essa macro é utilizada, é possível especificar instâncias (ou cópias estáticas) da super-instrução de maneira mais simples:

```
<alias> <nome>, [entradas]
```

O pré-processador converte automaticamente as linhas descritas com a sintaxe simples na sintaxe original, descrita na Seção 4.2.2.7. A Figura 4.7 mostra um exemplo de código de montagem no quadro *A*, onde a macro `superinst` é utilizada. No quadro *B* é exibido o resultado da conversão feita pelo pré processador. Repare como o booleano de imediato é usado para determinar se as super-instruções são do tipo `super` ou `superi`. Se o booleano de especulação fosse verdadeiro na declaração das super-instruções deste exemplo, elas seriam do tipo `specsuperi` e `specsuper`.

A macro `callgroup` possui a seguinte sintaxe:

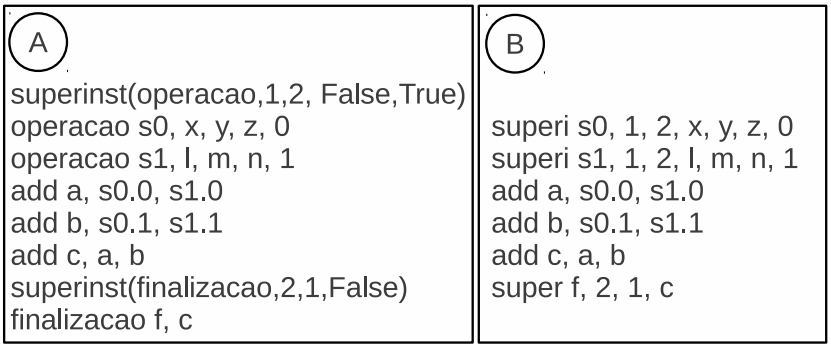

Figura 4.7: Funcionamento da macro `superinst`. O quadro A mostra um trecho de código de montagem que utiliza a macro `superinst`. O quadro B mostra o código resultante da conversão feita pelo pré-processador.

```
callgroup(<grupo>,<retorno>)
```

Ela cria número inteiro único para identificar um grupo de chamada e um nome associado ao mesmo e ao o ponto de retorno. Todos operandos enviados pela instrução `callsnd` devem especificar o nome do grupo de chamada ao qual pertencem. O pré-processador substitui o nome do grupo pelo inteiro associado, que é passado como imediato para as instruções `callsnd` e `retsnd`.

O montador do TALM gera, a partir do código de montagem, o binário para carga em uma implementação do TALM e um arquivo de alocação de instruções em EPS. Este arquivo diz, para cada instrução, qual o número do EP onde a mesma será alocada. A macro `placeinpe` é para definir esta alocação e possui a seguinte sintaxe:

```
placeinpe(<#EP>,<"DYNAMIC"|"STATIC">)
```

O segundo campo especifica se a alocação é dinâmica ou estática. Na alocação estática, a instrução descrita após a macro é mapeada no elemento de processamento (EP) cujo número é especificado no campo `#EP`. A alocação dinâmica é usada quando a definição da instrução seguinte a macro é feita com sequencias de repetição. Neste caso, a instrução definida com a sequencia de repetição será replicada em tempo de montagem e cada cópia sera alocada em um EP consecutivo, começando pelo EP cujo número está indicado no campo `#EP`.

As sequencias de repetição são usadas para replicar instruções automaticamente ou definir que uma instrução tem operandos de entrada proveniente de diversas instruções cujos identificadores tem um padrão. A Figura 4.8 mostra um exemplo de um trecho de código de montagem (descrito no quadro A), onde são definidas duas super-instruções `p` e `q`. O código convertido pelo pré-processador é mostrado no quadro B. Repare como a sequência de repetição `{i=0..NUM_TASKS-1}` é usada

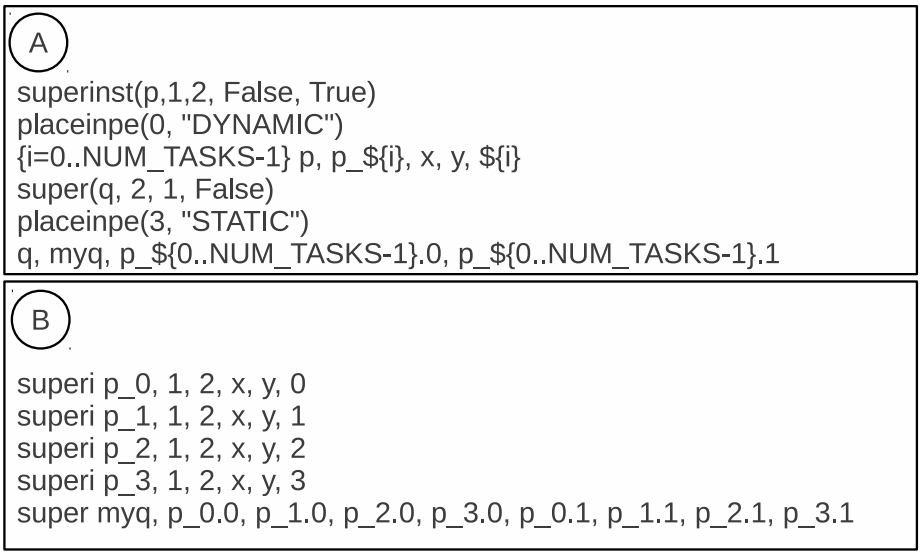

Figura 4.8: Funcionamento da macro `placeinpe` e das sequências de repetição no TALM. O quadro *A* mostra um trecho de código de montagem que utiliza a macro `placeinpe` em conjunto com sequências de repetição. O quadro *B* mostra o código resultante da conversão feita pelo pré-processador.

para replicar a super-instrução `p`. A contante `NUM_TASKS` representa o número de tarefas desejado para a aplicação. Este número é passado para o montador do TALM. No exemplo, são especificadas quatro tarefas. Ainda neste exemplo, é possível usar a sequência de repetição `{0..NUM_TASKS-1}` para definir que todo `p_${k}.0` e `p_${k}.1` ($0 \leq k \leq 3$) são entradas da super-instrução `q`. Além disto, a macro `placeinpe` é usada para especificar que, no arquivo de alocação gerado, `p_0`, `p_1`, `p_2` e `p_3` serão alocados para os EPs 0, 1, 2 e 3, respectivamente, e a alocação de `myq` será definida para o EP de número 3.

4.2.2.9 Quadro Resumo do Conjunto de Instruções

A Tabela 4.1 mostra um quadro resumo com as instruções da linguagem de montagem TALM. A coluna *Função* indica o que faz a instrução, as colunas *#Entradas* e *#Saídas* indicam, respectivamente, o número de operandos de entrada e de saída, enquanto o campo *Imediato* indica se um dos operandos de entrada é um imediato. No caso da instrução `callsnd`, o imediato é produzido pela macro `callgroup`.

Mnemônico	Função	#In	#Out	Imm	Tipo
add	+	2	1	não	`int`
sub	−	2	1	não	`int`
div	/	2	1	não	`int`

Tabela 4.1 – Continua na página seguinte

Continuação da página anterior

Mnemônico	Função	#In	#Out	Imm	Tipo
mult	+	2	1	não	`int`
mod	resto da divisão	2	1	não	`int`
and	E lógico	2	1	não	`int`
or	OU lógico	2	1	não	`int`
lthan	<	2	1	não	`int`
gthan	>	2	1	não	`int`
leq	≤	2	1	não	`int`
geq	≥	2	1	não	`int`
addi	+	2	1	sim	`int`
subi	−	2	1	sim	`int`
divi	/	2	1	sim	`int`
multi	+	2	1	sim	`int`
modi	resto da divisão	2	1	sim	`int`
andi	E lógico	2	1	sim	`int`
ori	OU lógico	2	1	sim	`int`
lthani	<	2	1	sim	`int`
gthani	>	2	1	sim	`int`
leqi	≤	2	1	sim	`int`
geqi	≥	2	1	sim	`int`
fadd	+	2	1	não	`float`
fsub	−	2	1	não	`float`
fdiv	/	2	1	não	`float`
fmult	+	2	1	não	`float`
f mod	resto da divisão	2	1	não	`float`
fand	E lógico	2	1	não	`float`
for	OU lógico	2	1	não	`float`
flthan	<	2	1	não	`float`
fgthan	>	2	1	não	`float`
fleq	≤	2	1	não	`float`
fgeq	≥	2	1	não	`float`
faddi	+	2	1	sim	`float`
fsubi	−	2	1	sim	`float`
fdivi	/	2	1	sim	`float`
fmulti	+	2	1	sim	`float`
fmodi	resto da divisão	2	1	sim	`float`

Tabela 4.1 – Continua na página seguinte

Continuação da página anterior

Mnemônico	Função	#In	#Out	Imm	Tipo
fandi	E lógico	2	1	sim	`float`
fori	OU lógico	2	1	sim	`float`
flthani	$<$	2	1	sim	`float`
fgthani	$>$	2	1	sim	`float`
fleqi	$\leq$	2	1	sim	`float`
fgeqi	$\geq$	2	1	sim	`float`
inctag	incrementa o rótulo de iteração	1	1	não	N/A
steer	desvio	2	1	não	N/A
callsnd	chamada de função	2	1	sim	N/A
retsnd	chamada de função	2	1	sim	N/A
ret	retorno de função	2	1	não	N/A
super	definida pelo usuário	até 32	até 32	não	N/A
superi	definida pelo usuário	até 32	até 32	sim	N/A
specsuper	super com especulação	até 32	até 32	não	N/A
specsuperi	superi com especulação	até 32	até 32	sim	N/A

Tabela 4.1: Conjunto de instruções do TALM.

A Tabela 4.2 mostra um resumo das macros da linguagem de montagem do TALM, com uma descrição de suas funções e sintaxe.

Nome	Função	Sintaxe
superinst	Criação de super-instruções	`superinst(alias,#super,#operandos)`
callgroup	Chamada de funções	`callgroup(chamada, retorno)`
placeinpe	Alocação de Instruções	`placeinpe(#EP, alocação)`

Tabela 4.2: Macros do TALM.

Capítulo 5

THLL (*TALM High Level Language*)

O modelo *dataflow* expõe paralelismo, tirando vantagem na forma como os dados são trocados entre os as instruções. Sob essa ótica, programar no TALM se resume em identificar tarefas paralelas e descrever como os dados são produzidos e consumidos por elas. A linguagem de alto nível do TALM, THLL (*TALM High Level Language*), foi desenvolvida para facilitar a tarefa de descrever programas paralelos usando o TALM. É uma extensão da linguagem C com as definições de super-instruções e suas variáveis de entrada e saída. Neste capítulo é apresentada a sintaxe dessas construções e posteriormente são mostrados exemplos de uso da linguagem para paralelizar aplicações com diferentes técnicas de programação paralela. A compilação desta linguagem é feita pelo *Couillard*, discutido em detalhes no Capítulo 7.

Um programa em THLL é composto por super-instruções, trechos de código que descrevem o comportamento de alguma tarefa. O programador da THLL descreve as super-instruções com seus argumentos de entrada e saída. As dependências de dados entre as super-instruções são determinadas por estes argumentos como em uma relação de produtor/consumidor. Além disto, o controle do programa pode ser escrito normalmente, usando a sintaxe da linguagem C, na qual a THLL se baseia. Laços de repetição e execução condicional podem ser descritas naturalmente. Um programa descrito em THLL será compilado para gerar um grafo de fluxo de dados e o código que descrever o comportamento das super-instruções. Este processo é realizado pelo *Couillard*, apresentado no Capítulo 7.

5.1 Blocos e Super-Instruções

O par de anotações `#BEGINBLOCK` e `#ENDBLOCK` é usado para marcar blocos de código que **não** serão compilados para *dataflow*. Esses blocos normalmente contém decla-

rações de *include files*, definições de funções auxiliares e declarações de variáveis globais usadas no código das super-instruções. Variáveis declaradas dentro destes blocos serão globalmente visíveis pelo corpo das super-instruções, mas não fora delas.

As super-instruções possuem um papel fundamental no TALM. A eficiência de um programa paralelo feito com o TALM depende da escolha correta da granularidade das super-instruções de forma a expor paralelismo sem comprometer os custos de comunicação.

A anotação das super-instruções é feita de acordo com o *statement* abaixo:

```
treb_super <single|parallel> input(<input_list>)
                             output(<output_list>)
#BEGINSUPER
    ...
#ENDSUPER
```

Super-instruções declaradas como `single` terão apenas uma instância no grafo de fluxo de dados, enquanto que instruções declaradas como `parallel` podem ter múltiplas instâncias que executarão em paralelo, dependendo da alocação nos EPs e da disponibilidade de recursos na máquina hospedeira. Na anotação das super-instruções também são definidas as variáveis de entrada, usadas pelas super-instruções na sua computação, e as variáveis de saída, produzidas pelas mesmas. A forma como estas variáveis são declaradas é explicada na Seção 5.2

5.2 Variáveis

A THLL requer que o programador especifique como as variáveis conectam as diferentes super-instruções. Mais precisamente, todas as variáveis usadas como entrada ou saída de super-instruções precisam ser previamente declaradas para garantir que os operandos sejam corretamente trocados entre instruções (sem perda de informação devido a conversões de dados incorretas). Além disso, variáveis de saída usadas em super-instruções paralelas devem ser declaradas da seguinte forma:

```
treb_parout <TIPO> <IDENTIFICADOR>;
```

O *Storage Classifier* `treb_parout` é usada para diferenciar a tipagem de variáveis de saída de super-instruções paralelas, que geralmente possuem múltiplas instâncias. Sendo assim, variáveis de saída de super-instruções paralelas também terão múltiplas instâncias, uma para cada instância da super-instrução paralela. Quando uma variável de entrada declarada como `treb_parout` é usada como entrada de uma outra super-instrução é necessário especificar qual instância está sendo referenciada. Para isto, a linguagem provê a seguinte sintaxe:

```
<IDENTIFICADOR>::< NÚMERO |
                  * |
                  mytid |
                  (mytid + NÚMERO) |
                  (mytid - NÚMERO) |
                  lattid>
```

Considere uma variável de nome x, produzida por uma super-instrução paralela V. Quando uma outra super-instrução T (`single` ou `parallel`) declara como entrada $x :: i$, sendo i um número inteiro positivo, isto significa que x produzido pela i-ésima instância de V será enviado para T. Além disto, se T declara como entrada $x :: *$, isto significa que todas as instâncias de x serão enviadas para T (uma abstração útil para quando uma super-instrução faz uma operação de *join*, por exemplo).

Geralmente é conveniente referenciar a instância corrente da super-instrução. Considere V e T super-instruções paralelas. Além disto, entenda-se por $S.i$ a i-ésima instância de uma super-instrução paralela S. Dado que x é produzido por V, se T especifica $x :: mytid$ como operando de entrada, isto significa que para cada instância k de T, $T.k$ recebe x de $V.k$.

Expressões com operadores $+$ e $-$ também são permitidas com *mytid*. Considere V e T super-instruções paralelas. Por exemplo, se V produz um operando x e T especifica $x :: (mytid - 1)$ como entrada, isto significa que para uma instância k, $T.k$ receberá x de $V.(k-1)$. Por fim, considere V e T super-instruções, sendo V paralela e T *single* ou *parallel*. Se uma variável x é produzida por V e se T tem como operando de entrada $x :: lasttid$, à instância de x produzida pela última instância paralela de V será entrada de T.

Para os casos onde existem dependências entre instâncias de uma mesma super-instrução paralela (dependências locais), é possível especificar variáveis de entrada usando a sintaxe à seguir:

```
local.<IDENTIFICADOR>::<(mytid + NÚMERO) |
                        (mytid - NÚMERO)>
```

Por exemplo, se um super-instrução paralela T produz o operando x e recebe $local.x :: (mytid - 2)$, isto significa que $T.k$ depende de $T.(k-2)$. Além disto, $T.0$ e $T.1$ não possuem dependências locais.

Quando a construção `local` é usada, é possível também especificar operando que serão enviados apenas para as instâncias independentes (ou instâncias sem dependência local) da super-instrução. É usada a seguinte sintaxe:

```
starter.<IDENTIFICADOR>::< NÚMERO |
```

```
* |
mytid |
(mytid + NÚMERO) |
(mytid - NÚMERO) |
lattid>
```

No exemplo anterior, se T também tiver como entrada *starter.c*, apenas $T.0$ e $T.1$ receberão o operando c, mas não receberão o operando x, enquanto as demais instâncias de T receberão x, mas não receberão c.

5.3 Funções Auxiliares e Argumentos de Linha de Comando

As funções `treb_get_tid()` e `treb_get_n_tasks()` foram adicionadas à *Trebuchet* e podem ser chamadas dentro das super-instruções. A primeira função retorna o `thread_id` (identificador da *thread*) associado à instância da super-instrução paralela na qual foi feita a chamada. A última retorna o número de *threads* ou instâncias paralelas da super-instrução em questão. Essas funções podem ser usadas para identificar a porção do trabalho a ser feito por cada instância.

Como aplicações são executas pela máquina virtual *Trebuchet*, argumentos de linha de comando não podem ser declarados diretamente no código da aplicação. A máquina virtual deve receber esses argumentos e repassar para a aplicação executada. Sendo assim, a *Trebuchet* armazena um vetor de argumentos de linha de comando e o número de argumentos nas variáveis `treb_superargv` e `treb_superargc`, respectivamente. O *Couillard* então, quando gera o arquivo com o código da biblioteca de funções (`.lib.c`, declara essas variáveis como `extern`, o que permite que esses argumentos sejam acessados dentro do bloco das super-instruções. Alternativamente poderiam ser incluídas funções para acessar os argumentos, mas isso é apenas uma decisão de projeto e não afeta o desempenho ou a programabilidade do sistema.

5.4 Exemplos Ilustrativos

O passo-a-passo para descrever código paralelo em super-instruções é simples. O desenvolvedor deve primeiro dividir o código em super-instruções que podem executar em paralelo. Super-instruções de inicialização e terminação serão geralmente *single*, enquanto a maior parte do trabalho estará em super-instruções *parallel*. O programador, em seguida, especifica como as super-instruções se comunicam. Se a

comunicação é puramente de controle, o programador deve adicionar variáveis extras para especificar essa conexão (uma técnica comum em programação paralela). Note que o programador ainda deve prevenir-se contra condições de corrida entre super-instruções.

Os exemplos que se seguem mostram como usar o THLL para descrever programas paralelos, usando diferentes técnicas. Os exemplos consolidam as explicações acerca da sintaxe. No Capítulo 7 é descrito como estes exemplos são compilados pelo *Couillard* para gerar o grafo de fluxo de dados e o código que descreve o comportamento das super-instruções para execução na *Trebuchet*.

5.4.1 *Fork* e *Join* com Operação de Redução

A Figura 5.1 mostra um exemplo de como a THLL é usada para paralelizar uma aplicação. Neste caso, foi escolhida uma aplicação regular que lê um vetor de 10000 números inteiros e calcula a soma destes números. A Figura 5.2 mostra um grafo com as relações de fluxo de dados entre as super-instruções. O grafo exibe o padrão de paralelismo da aplicação, mas não é o grafo *dataflow* completo, com todos os detalhes de controle, que seriam necessários pelo TALM.

Na figura é possível observar o código C escrito com a THLL. Constantes e globais são definidas na primeira seção de código, marcada com `#BEGINBLOCK` e `#ENDBLOCK`, conforme descrito na Seção 5.1.

Também observa-se, na função `main`, a existência de três super-instruções: *(i)* uma super-instrução *single* que lê o arquivo e inicializa o vetor, *(ii)* uma super-instrução `parallel` que faz uma soma parcial em cada uma de suas instâncias e *(iii)* uma super-instrução *single* que faz uma operação de redução, produzindo e imprimindo o resultado final.

Observe como a variável `treb_parout` b é passada como entrada para a última super-instrução. A notação $b :: *$ descreve que todas as instâncias de b, cada uma contendo uma soma parcial, serão necessárias para fazer a operação de redução. Observe que dentro da super-instrução que faz a operação de redução, b é tratado como um vetor, onde cada elemento é produzido por uma instância da super-instrução de processamento.

Nesse exemplo também é possível observar o acesso de argumentos de linha de comando com a variável `superargv`. No exemplo, o argumento passado é o nome do arquivo que contém os elementos do vetor.

Na super-instrução de processamento observa-se o uso das funções `treb_get_tid()` e `treb_get_n_tasks()` para fazer a divisão do trabalho entre as instâncias paralelas dessa super-instrução. Note que o laço de repetição realiza um somatório de apenas um trecho do vetor para cada instância. Neste exemplo é assumido que

```
#BEGINBLOCK  //INCLUDES, FUNCTIONS E GLOBALS
   #include <stdio.h>
   #include <stdlib.h>
   #define size 10000
   int A[size];
#ENDBLOCK
int main(){
   int a=0; treb_parout int b;
   treb_super single input(a) output(a)
   #BEGINSUPER  //Código de inicialização
      int i;
      FILE * f;
      f = fopen(superargv[0], "r");
      for(i=0; i<size; i++)
         A[i]=fscanf(f, "%d\n", &A[i]);
      fclose(f);
   #ENDSUPER

   treb_super parallel input(a) output(b)
   #BEGINSUPER  //Código de processamento
      int i, sum=0;
      int tid = treb_get_tid();
      int n_tasks = treb_get_n_tasks();
      int task_size = size / n_taks;
      int begin = tid * task_size;
      int end = begin + task_size;
      for(i=begin; i<end; i++){
         sum+=A[i];
      }
      b=sum;
   #ENDSUPER

   treb_super single input(b::*) output(a)
   #BEGINSUPER //Redução (+)
      int i, total=0;
      for(i=0; i<treb_get_n_tasks(); i++){
         total+=b[i];
      }
      printf("%d\n", total);
   #ENDSUPER
   return 0;
}
```

Figura 5.1: Exemplo de uma aplicação regular com THLL.

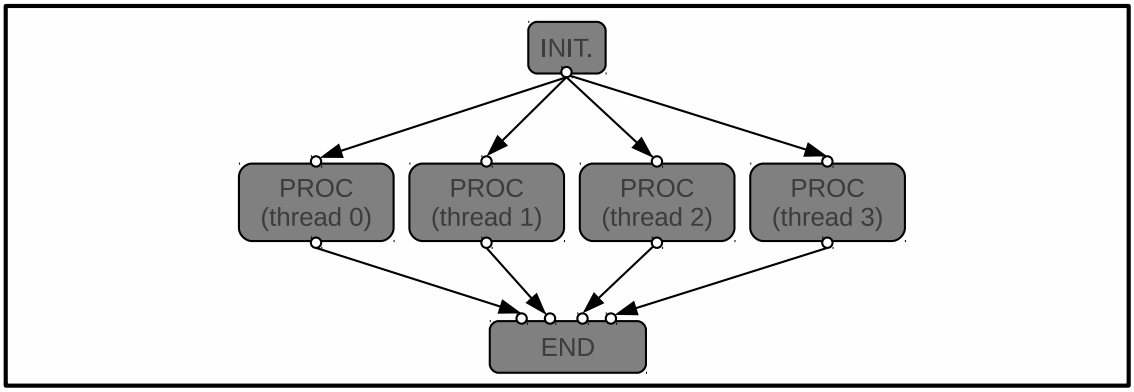

Figura 5.2: Grafo com o padrão de paralelismo de uma aplicação regular.

o número de instâncias paralelas desta super-instrução é divisível pelo tamanho do vetor.

A descrição de operações de redução, como a descrita neste exemplo, podem ser automatizadas com a criação de um *template*. A inclusão desta funcionalidade é objeto de trabalhos futuros.

5.4.2 Escondendo Latência de Operações de Entrada e Saída

A Figura 5.3 mostra um exemplo de como a THLL pode ser usada para esconder latência de operações de entrada e saída em uma aplicação paralela. Neste exemplo, dois vetores 10000 elementos precisam ser lidos de um arquivo, somados e depois o vetor resultado precisa ser escrito em um arquivo de saída. . A Figura 5.4 mostra um grafo com as relações de fluxo de dados entre as super-instruções. O grafo exibe o padrão de paralelismo da aplicação, mas não é o grafo *dataflow* completo, com todos os detalhes de controle, que seriam necessários pelo TALM.

O código mostra as diferentes etapas realizadas por super-instruções: *(i)* abertura dos arquivos, *(ii)* leitura dos arquivos de entrada, *(iii)* processamento, *(iv)* escrita do arquivo de saída e *(v)* fechamento dos arquivos.

Note que as etapas de leitura e escrita são descritas com super-instruções *parallel*, mas como existem dependências locais nestas duas super-instruções, elas serão executadas sequencialmente (embora distribuídas em diferentes EPs). Esta construção permite a execução de cada instância da etapa de processamento assim que a instância de leitura correspondente tiver sido finalizada, invés de aguardar a leitura completa. Ela também permite que a escrita de resultados processados pela instância i da super-instrução de processamento seja feita, sem ter que esperar que as instâncias x, tal que $x < i$, terminem.

```
#BEGINBLOCK  //INCLUDES, FUNCTIONS E GLOBALS
   #include <stdio.h>
   #include <stdlib.h>
   #define size 10000
   FILE * fa;
   FILE * fb;
   FILE * fc;
   int A[size];
   int B[size];
   int C[size];
#ENDBLOCK
int main(){
   int a=0, e; parout int b,c,d;
   super single input(a) output(a)
   #BEGINSUPER  //Código de inicialização
      fa = fopen(superargv[0], "r");
      fb = fopen(superargv[1], "r");
      fc = fopen(superargv[2], "w");
   #ENDSUPER

   super parallel input(starter.a, local.b::(mytid-1)) output(b)
   #BEGINSUPER //Leitura
      int i;
      int tid = treb_get_tid();
      int n_tasks = treb_get_n_tasks();
      int task_size = size / n_tasks;
      int begin = tid * task_size;
      int end = begin + task_size;
      for(i=begin; i<end; i++){
         fscanf(fa, "%d\n", &A[i]);
         fscanf(fb, "%d\n", &B[i]);
      }
   #ENDSUPER

   super parallel input(b::mytid) output(c)
   #BEGINSUPER  //Código de processamento
      int i;
      int tid = treb_get_tid();
      int n_tasks = treb_get_n_tasks();
      int task_size = size / n_tasks;
      int begin = tid * task_size;
      int end = begin + task_size;
      for(i=begin; i<end; i++){
         C[i]=A[i]+B[i];
      }
   #ENDSUPER

   super parallel input(c::mytid, local.d::(mytid-1)) output(d)
   #BEGINSUPER //Escrita
      int i;
      int tid = treb_get_tid();
      int n_tasks = treb_get_n_tasks();
      int task_size = size / n_tasks;
      int begin = tid * task_size;
      int end = begin + task_size;
      for(i=begin; i<end; i++){
         fprintf(fc, "%d\n", C[i]);
      }
   #ENDSUPER

   super single input(d::lasttid) output(e)
   #BEGINSUPER //Código de finalização
      fclose(fa);
      fclose(fb);
      fclose(fc);
   #ENDSUPER
   return 0;
}
```

Figura 5.3: Exemplo de como esconder latência de E/S com THLL.

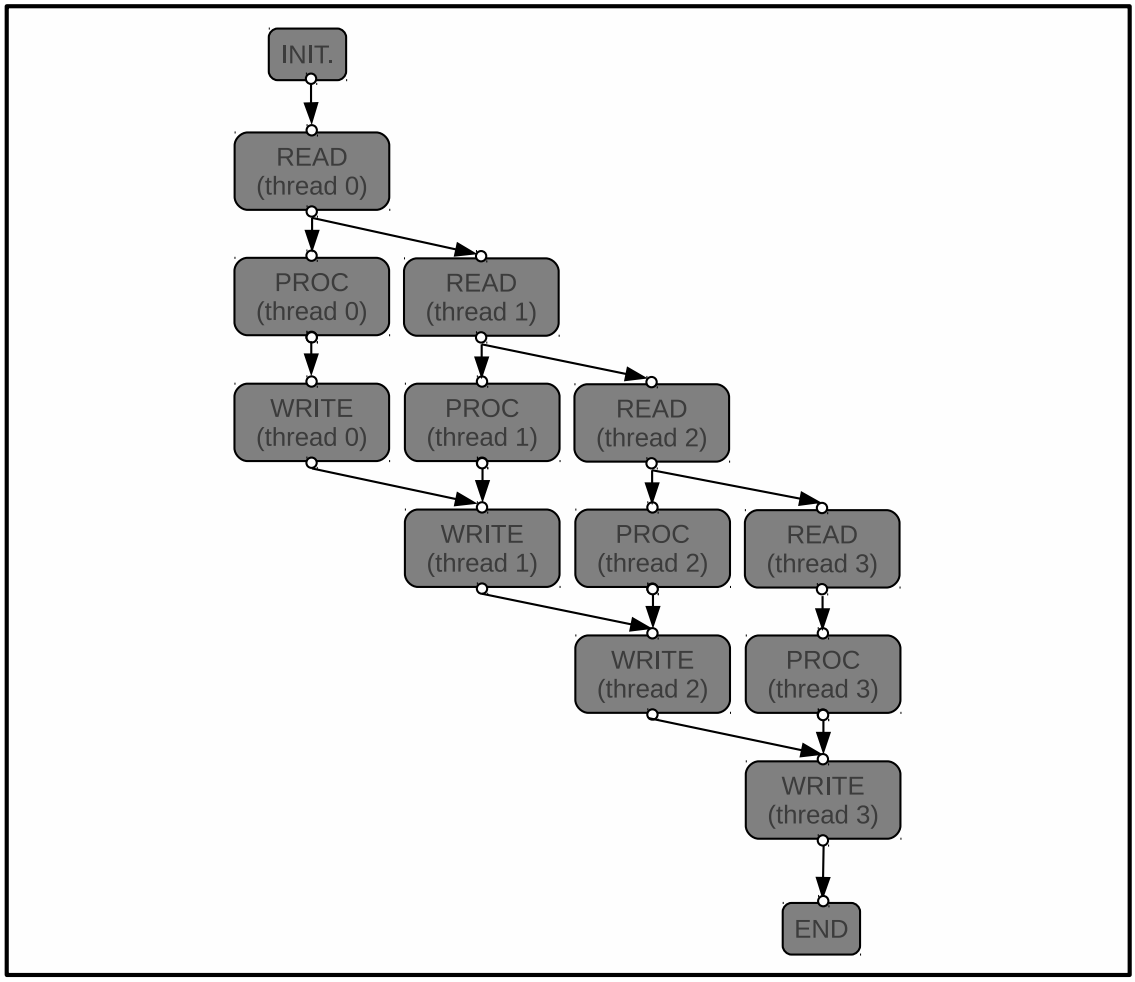

Figura 5.4: Grafo com o padrão de paralelismo para esconder latência de E/S em uma aplicação.

5.4.3 *Pipelines* Lineares

A Figura 5.5 mostra como descrever aplicações paralelas na forma de um *pipeline*. A aplicação paralelizada é a mesma descrita na Seção 5.4.2: uma soma de vetores. Neste caso, a paralelização é feita em um laço de repetição onde um bloco de 100 elementos é processado em cada iteração. O *pipeline* descrito possui 3 estágios: leitura, processamento e escrita. A Figura 5.6 mostra um grafo com as relações de fluxo de dados entre as super-instruções. O grafo exibe o padrão de paralelismo da aplicação, mas não é o grafo *dataflow* completo, com todos os detalhes de controle, que seriam necessários pelo TALM. Note que o laço de repetição é envolvido com linhas tracejadas no grafo e é possível ver o ciclo no grafo, definido pelos operandos trocados entre iterações.

Note a forma como as super-instruções são definidas para paralelizar a aplicação usando essa estratégia:

- Uma super-instrução `single` é criada para abrir arquivos utilizados.
- O laço de repetição é criado para computar blocos do vetor.
- Uma super-instrução `single` descreve o primeiro estágio. Nele é feita a alocação dinâmica de memória para guardar um bloco de elementos dos dois vetores, que são lidos de dois arquivos distintos.
- Uma super-instrução paralela é responsável pelo segundo estágio. Ela recebe o ponteiro dos blocos lidos e cada instância faz o processamento de uma parte do bloco (sub-bloco). Note que o número de instâncias é divisível pelo tamanho do bloco. O resultado é colocado em uma área de memória alocada dinamicamente por cada instância
- Uma super-instrução `single` é responsável pelo terceiro estágio. Ela libera a área de memória alocada para os blocos dos vetores de entrada, imprime o resultado parcial em um arquivo e libera os blocos de resultado alocados por cada instância da super-instrução anterior. Repare que, como houve um *join*, em `dc` é recebido um vetor de sub-blocos.

```
#BEGINBLOCK  //INCLUDES, FUNCTIONS E GLOBALS
   #include <stdio.h>
   #include <stdlib.h>
   #define size 10000
   #define block 100
   FILE *fa, *fb, *fc;
#ENDBLOCK
int main(){
   void *da, *db; parout void *dc; int x=0, a=0, b, c=0, d;
   super single input(a) output(b)
   #BEGINSUPER  //Código de inicialização
      fa = fopen(superargv[0], "r");
      fb = fopen(superargv[1], "r");
      fc = fopen(superargv[2], "w");
      b=size;
   #ENDSUPER
   while(x<b)
   {
      super single input(b) output(b, da, db)
      #BEGINSUPER //Leitura
         int * A = (int *) malloc(block*sizeof(int));
         int * B = (int *) malloc(block*sizeof(int));
         int i;
         for(i=0; i<block; i++){
            fscanf(fa, "%d", &A[i]);
            fscanf(fb, "%d", &B[i]);
         }
         da=A; db=B;
         b-=block;
      #ENDSUPER
      super parallel input(da, db) output(dc)
      #BEGINSUPER  //Código de processamento
         int i,j=0;
         int *A = (int *)da;
         int *B = (int *)db;
         int tid = treb_get_tid();
         int n_tasks = treb_get_n_tasks();
         int task_size = block / n_tasks;
         int *C = (int *) malloc(task_size*sizeof(int));
         int begin = tid * task_size;
         int end = begin + task_size;
         for(i=begin; i<end; i++){
            C[j]=A[i]+B[i];
            j++;
         }
         dc=C;
      #ENDSUPER
      super single input(da, db, dc::*, c) output(c)
      #BEGINSUPER //Escrita
         free(da);
         free(db);
         int ** C = (int **)dc;
         int i,j;
         int n_tasks = treb_get_n_tasks();
         int task_size = block / n_tasks;
         for(i=0; i<n_tasks; i++){
            for(j=0;j<task_size;j++)
               fprintf(fc, "%d\n", C[i][j]);
            free(C[i]);
         }
         free(C);
      #ENDSUPER
   }

   super single input(c) output(d)
   #BEGINSUPER //Código de finalização
      fclose(fa);
      fclose(fb);
      fclose(fc);
   #ENDSUPER
   return 0;
}
```

Figura 5.5: Exemplo de *pipeline* linear com THLL.

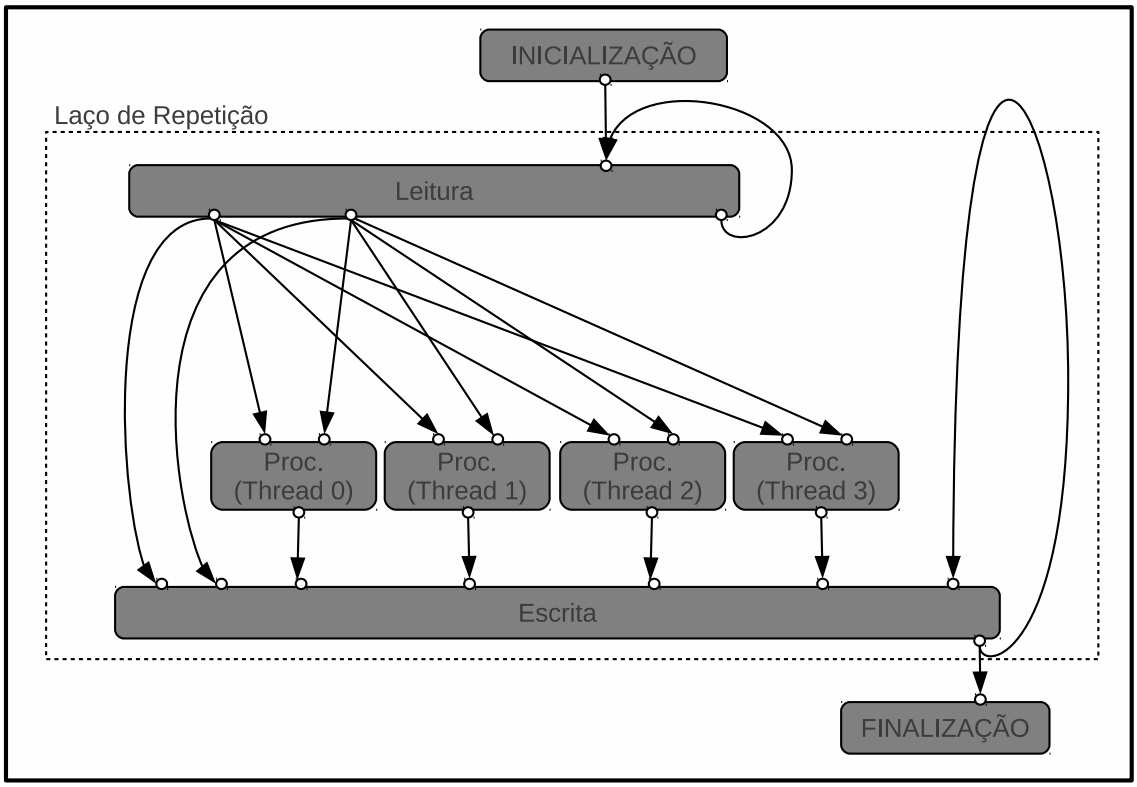

Figura 5.6: Grafo com o padrão de paralelismo de um *pipeline* linear.

5.4.4 *Pipelines* não Lineares

A Figura 5.7 mostra um exemplo de como usar a THLL para descrever um *pipeline* paralelo não linear (onde os estágios podem executar condicionalmente, ou o número de entradas e saídas para cada estágio variam). O exemplo é código esqueleto de uma aplicação que lê um arquivo contendo um conjunto de tarefas para ser processado e escreve o resultado em um outro arquivo. A fase de processamento pode ser dividida em três estágios (`Proc-1`, `Proc-2` e `Proc-3`). A etapa de processamento `Proc-2` foi dividida em 2 tarefas diferentes (`Proc-2A` e `Proc-2B`), que são executadas condicionalmente. . A Figura 5.8 mostra um grafo com as relações de fluxo de dados entre as super-instruções. O grafo exibe o padrão de paralelismo da aplicação, mas não é o grafo *dataflow* completo, com todos os detalhes de controle, que seriam necessários pelo TALM. Note que o laço de repetição e a execução condicional definida pela construção *IF-THEN-ELSE* são envolvidas por linhas tracejadas no grafo.

```
int main(){
    int a=0,b,f=0,cond=1,cond2;
    treb_parout int c,d,e;
    treb_super single input(a) output(a)
    #BEGINSUPER
        ...  //Inicialização
    #ENDSUPER
    while(cond>0)   {
        treb_super single input(a) output(b,cond2,cond)
        #BEGINSUPER
            ...  //Entrada/Condição
        #ENDSUPER
        treb_super parallel input(b) output(c)
        #BEGINSUPER
            ...   //PROC. 1
        #ENDSUPER
        if(cond2>0){
            treb_super parallel input(c::mytid) output(d)
            #BEGINSUPER
                ...  //PROC. 2A
            #ENDSUPER
        }
        else{
            treb_super parallel input(c::mytid) output(d)
            #BEGINSUPER
                ...  //PROC. 2A
            #ENDSUPER
        }
        treb_super parallel input(d::mytid) output(e)
        #BEGINSUPER
            ...  //PROC. 3
        #ENDSUPER
        treb_super single input(f,e::*) output(f)
        #BEGINSUPER
            ...  //Saída
        #ENDSUPER
    }
    treb_super single input(f) output(void)
    #BEGINSUPER
        ...  //Finalização
    #ENDSUPER
    return 0;
}
```

Figura 5.7: Exemplo de *pipeline* não linear com THLL.

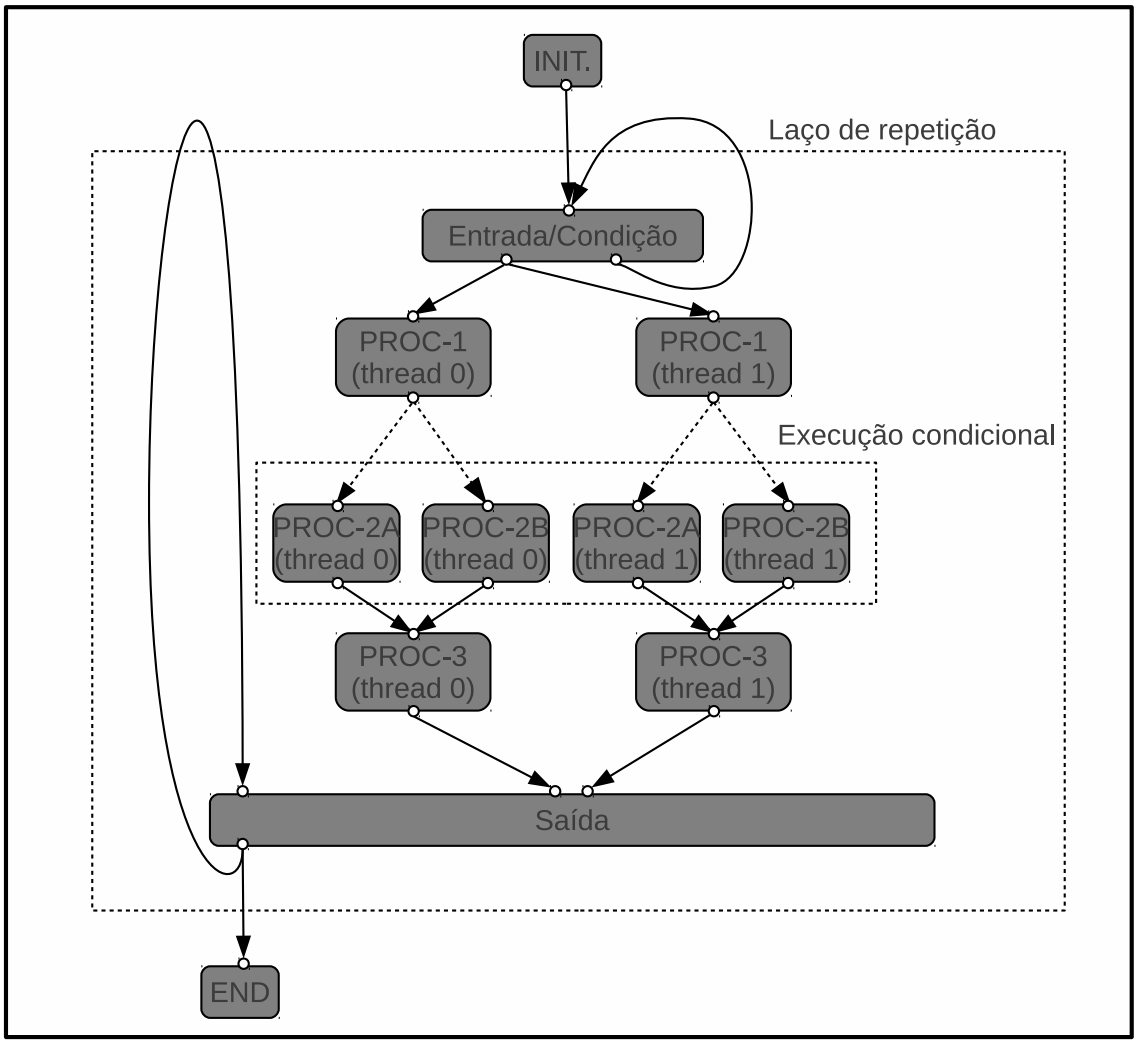

Figura 5.8: Grafo com o padrão de paralelismo de um *pipeline* não linear.

Capítulo 6

Trebuchet: TALM para CMPs

A Trebuchet é uma implementação do TALM para máquinas CMPs. Neste capítulo são discutidas as principais questões relativas a sua implementação e é feita uma descrição de como utilizá-la para paralelizar aplicações.

6.1 Implementação da Máquina Virtual

Os principais aspectos de implementação da *Trebuchet* são:

- As estruturas de armazenamento de instruções e operandos.
- A forma como as *threads* que representam os elementos de processamento virtuais são mapeados nos núcleos de processamento reais.
- Mecanismos de alocação dinâmica de instruções.
- Detecção de terminação global.

Nesta seção são abordadas as soluções adotadas para cada um destes aspectos.

6.1.1 Instruções e Troca de Operandos

Como qualquer arquitetura *dataflow*, o TALM é naturalmente baseado em passagem de mensagens. Para instruções alocadas em um mesmo elemento de processamento (EP), a troca de operandos é feita localmente com acesso direto às estruturas de armazenamento de operandos. Para instruções localizadas em diferentes EPs as rotinas `Envia()` e `Recebe()` são responsáveis pela troca de operandos entre instruções. A implementação destas rotinas depende da máquina alvo.

Como a máquina hospedeira da *Trebuchet* é uma CMP, com memória compartilhada entre os núcleos de processamento, a passagem de mensagens é de fato implementada através de regiões de memória compartilhada. Neste caso, o custo de

comunicação é dado pelas falhas na *cache* mais o custo de adquirir os *locks* requeridos para acessar estas regiões. Por este motivo, não é apenas importante alocar instruções em EPs diferentes para maximizar o paralelismo, mas também tentar manter instruções dependentes no mesmo EP, ou minimizar a distância entre elas na rede de interconexão, para evitar altos custos de comunicação.

A Figura 6.1 (a) detalha a estrutura de uma instrução na lista de instruções. A estrutura contém:

- O `Opcode` da instrução.
- Um operando `Imediato`.
- `#Origens`: o número de operandos de entrada.
- `Entradas`: uma lista encadeada para armazenar as estruturas de casamento. Cada elemento contem um ponteiro `p` para o próximo elemento da lista, o rótulo com o número da chamada e iteração (`#Chamada` e `#Iter.`, respectivamente), além de até 32 operandos. Um casamento ocorre para um elemento desta lista quando todos os operandos (`# Origens`) estão presentes.
- `#Resultados`: o número de operandos de saída produzidos por uma certa instrução.
- A *matriz de destinos* indica para onde enviar operandos produzidos por uma certa instrução. Cada linha especifica N destinos para um operando. Cada elemento desta matriz contem um ponteiro para a instrução de destino e o número da porta de entrada (`#Porta`) pela qual o operando será inserido quando chegar ao seu destino.
- O campo `ContCh` é usado por instruções `callsnd` para armazenar o valor do contador de chamadas, mencionado na Seção 4.2.2.6. Neste caso, o campo `Imediato` é usado para armazenar o *call group*, explicado na mesma seção.
- `#EP`: cada instrução tem o número do EP no qual está mapeada. Note que operandos trocados entre instruções localizadas no mesmo EP não serão enviadas pela rede de interconexão virtual. Ao invés disto, os operandos serão diretamente encaminhadas aos seus destinos na lista de instruções.

A Figura 6.1 (b) mostra uma mensagem com um operando, trocada entre EPs. A mensagem contém um ponteiro para a instrução destino, o número da porta de entrada na instrução destino e o operando em si.

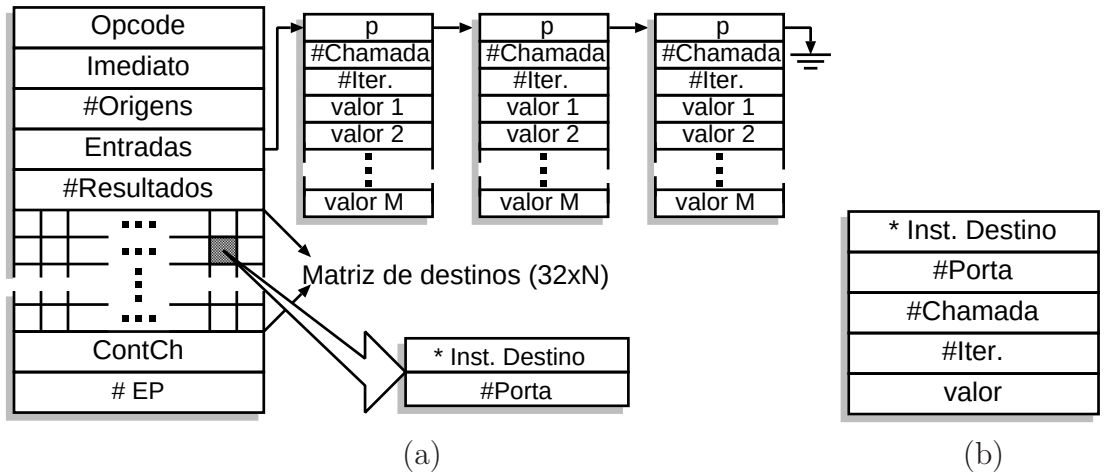

Figura 6.1: Estruturas da *Trebuchet* para armazenamento de operandos e instruções (a) e estrutura de uma mensagem com operando (b).

6.1.2 *Core Affinity*

Ao se paralelizar uma aplicação é importante alocar *threads* que se comunicam com frequência em núcleos próximos. No caso de núcleos com *Hyper Threading*^{TM} o mesmo vale para os processadores lógicos. Além disto, se as *threads* compartilham muitos objetos de memória, é desejável que elas sejam alocadas em núcleos que compartilham suas *caches*. Isto pode se feito com a chamada de sistema `sched_setaffinity()`.

Durante os experimentos descritos no Capítulo 8, foi identificada a necessidade de usar *schedule affinity* para fixar cada EP (*thread*) em um núcleo de processamento. De fato, o *OpenMP* também usa *schedule affinity* para definir grupos de núcleos nos quais uma *thread* poderia ser escalonada para execução. No entanto a solução utilizada na *Trebuchet* é mais ingênua, pois cada EP é associado a um único núcleo, ao invés de possuir um grupo de candidatos.

6.1.3 Roubo de Tarefas

A *Trebuchet* pode se apoiar tanto em escalonamento estático de instruções em EPs, quanto usar mecanismos dinâmicos, como roubo de tarefas, para solucionar problemas de balanceamento de carga. Uma versão inicial de um mecanismo de roubo de tarefas foi incluída na *Trebuchet* e usada em um dos experimentos descrito no Capítulo 8. O algoritmo é baseado no ABP [61]. A principal diferença é que a versão desenvolvida para a *Trebuchet* usa uma *double-ended queue* (*deque*) do tipo FIFO (*First In, First Out*) invés de LIFO (*Last In, First Out*), como é o caso do ABP. A ordem FIFO foi escolhida para que instruções mais antigas tenham prioridade na execução, o que é uma característica desejável nas aplicações alvo, neste momento.

A política de roubo escolhida é bem simples: sempre que um EP está ocioso ele busca, aleatoriamente, por trabalho nas filas de despacho dos EPs vizinhos. O estudo mais profundo do mecanismo de Roubo de Tarefas para a *Trebuchet* não é do escopo deste trabalho. A versão inicial foi feita movida pela necessidade das aplicações usadas nos experimentos. Uma avaliação extensiva, bem como apresentação de políticas de roubo mais eficientes são considerados para trabalhos futuros.

6.1.4 Detecção de Terminação Global

Em máquinas de Von Neumann, o termino da execução de um programa é identificado com a simples observação do contador de programa. Quando o contador de programa atinge a última instrução do programa, ocorre a terminação do mesmo. Em máquinas *dataflow* não há um contador de programa e, como cada elemento de processamento possui apenas uma visão de seu estado local, ele não pode determinar sozinho se o programa chegou ou não ao seu fim. Quando um elemento de processamento está ocioso, ou seja, com sua fila de prontos vazia, ainda podem chegar mensagens com operandos que ativarão a execução de novas instruções. A terminação só ocorrerá, quando todos os elementos de processamento estiverem vazios e não existirem mensagens com operandos em transito. Sendo assim, é necessária uma forma de saber o estado global do sistema, EPs e arestas de comunicação.

Com este fim, foi feita uma implementação do algoritmo de detecção de terminação através de *Snapshots* distribuídos [62] para a *Trebuchet*. Como a topologia da rede adotada para a *Trebuchet* é um grafo completo, não é necessária a existência de um líder (cada nó pode, por si só, reunir todas as informações que necessita acerca dos outros nós). Além disto, como é preciso apenas detectar se o programa chegou ao seu fim, só é utilizado um tipo de mensagem, o marcador de terminação.

Um nó (EP) inicia a detecção de terminação quando o mesmo entra em estado ocioso, o que significa que não existem instruções na sua fila de prontos. Ele então difunde um marcador contendo um **rótulo de terminação** igual ao maior marcador já visto mais 1. Os demais nós aderem a detecção da terminação se eles já estiverem ociosos e o marcador recebido possuir rótulo superior ao maior rótulo de terminação já visto até o momento. Após ter entrado no novo estado de detecção de terminação, os outros nós também difundem o marcador recebido, indicando para todos os seus vizinhos que eles também estão ociosos.

Um vez que um nó participante da detecção de terminação tenha recebido, de todos os seus vizinhos (todos os outros nós, no grafo completo), um marcador com aquele **rótulo de terminação**, ele difunde um outro marcador com o mesmo rótulo, se ainda estiver ocioso. O segundo marcador carrega a informação de que todas as arestas de entrada daquele estão vazias, visto que cada nó apenas difundiu o

marcador pois permaneceu ocioso até receber a primeira rodada de marcadores de todos os outros nós, o que significa que ele não recebeu mensagens com operando. Fica claro que, uma vez recebido o segundo marcador de todos os vizinhos, o nó pode terminar.

6.2 Paralelizando Aplicações com a Trebuchet

Para paralelizar uma aplicação sequencial usando a *Trebuchet* é necessário compilá-la para a linguagem de montagem *dataflow* do TALM e executá-la na *Trebuchet*. A *Trebuchet* é implementada como uma máquina virtual e cada um de seus EPs é associado a uma *thread* na máquina hospedeira. Quando um programa é executado na *Trebuchet*, as instruções são alocadas aos seus EPs e disparadas segundo o modelo *dataflow*. Instruções independentes irão executar em paralelo se estiverem mapeadas em diferentes EPs e se existirem núcleos de processamento disponíveis na máquina hospedeira para executar as *threads* dos EPs simultaneamente.

Como os custos de interpretação podem ser altos, a execução de programas totalmente compilados para o modelo *dataflow* pode apresentar desempenho insatisfatório. Para tratar este problema, blocos de código Von Neumann são transformados em super-instruções. Estes blocos são compilados normalmente para a máquina hospedeira e a interpretação de uma super-instrução se resumirá à execução direta do bloco relacionado. Note que a execução de uma super-instrução continua a ser disparada pela máquina virtual, guiada pelo fluxo de dados. Esta é uma das principais contribuições introduzidas pela *Trebuchet*: permitir a definição de blocos de Von Neumann com diferentes granularidades (não só métodos, como no *Program Demultiplexing* [7]), além de não demandar suporte de *hardware*.

A Figura 6.2 mostra o fluxo de trabalho a ser seguido para paralelizar um programa sequencial e executa-lo na Trebuchet. As ferramentas associadas a cada um dos processos descritos são indicadas na figura. Inicialmente, blocos que irão formar super-instruções são definidos. Depois é feita uma extração de código das super-instruções para transformar todos os blocos em funções que irão coletar operandos de entrada da *Trebuchet*, fazer o processamento descrito pelo programador e retornar os operandos de saída. Ferramentas de *profilling* podem ser utilizadas para determinar os melhores candidatos para paralelização.

No passo seguinte, os blocos transformados são compilados para uma biblioteca dinâmica, que será disponibilizada ao interpretador da máquina abstrata. Então, um grafo *dataflow* conectando todos os blocos é definido e o código de montagem é gerado. O código de montagem pode ter super-instruções e instruções simples.

Por último, um binário *dataflow* é gerado a partir do código de montagem, a alocação dos EPs é definida e o código binário é carregado e executado. Como

dito anteriormente, a execução de instruções simples requer interpretação completa pela máquina virtual, enquanto super-instruções são executadas diretamente pela máquina hospedeira.

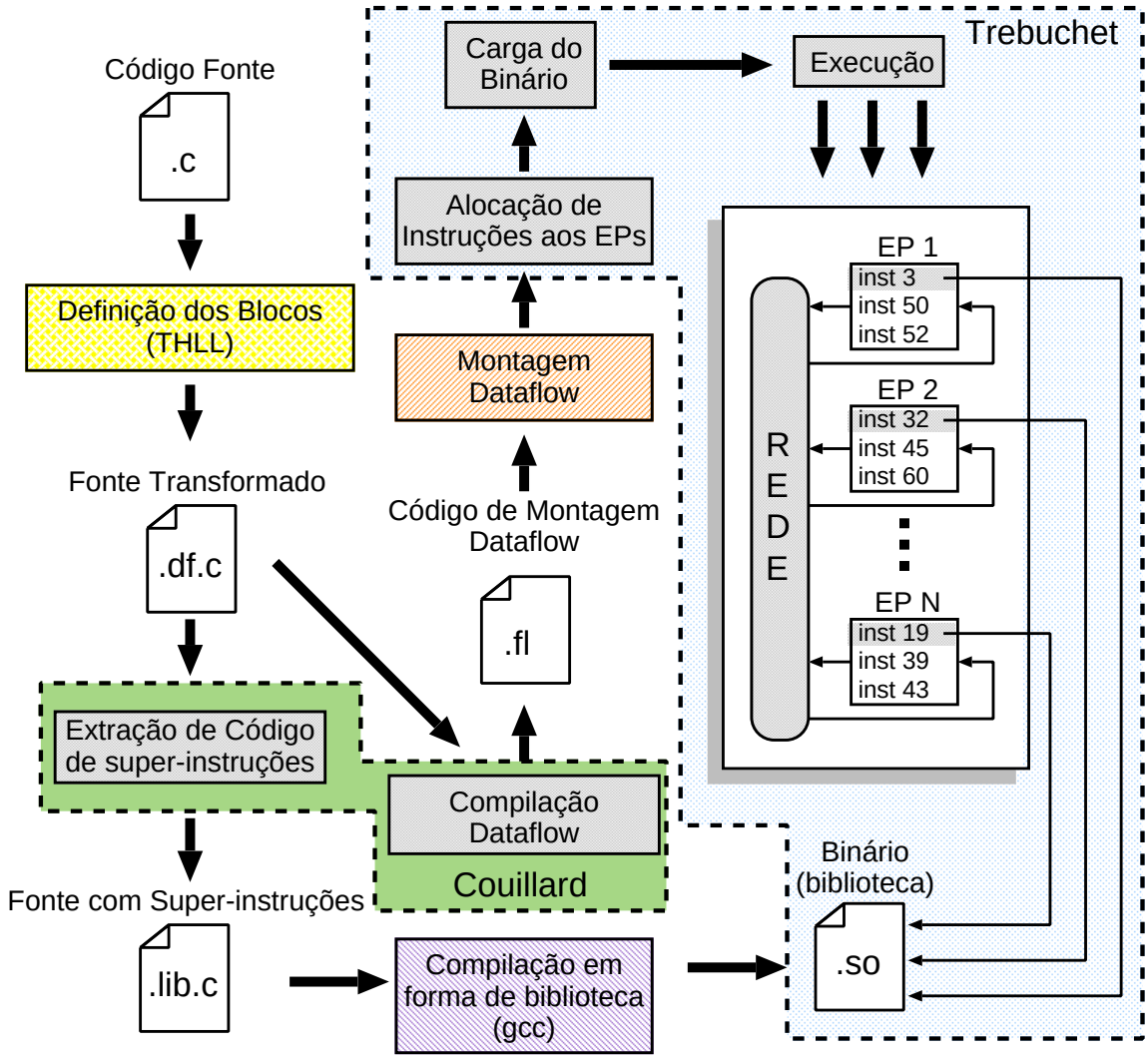

Figura 6.2: Fluxo de Trabalho para paralelizar aplicações com a *Trebuchet*.

Nos exemplos que se seguem é explicado como paralelizar aplicações manualmente, sem o uso da linguagem de alto nível do TALM, para dar uma ideia inicial do trabalho que precisaria ser feito por um compilador.

6.2.1 Paralelizando Aplicações Regulares com a *Trebuchet*

A Figura 6.3 detalha a paralelização manual (sem uso de compilador) do núcleo de uma integração numérica usando a *Trebuchet*. O código fonte sequencial é mostrado em *A*. O uso da macro `superinst` ilustrado no código de montagem *dataflow* (em *D*) permite a extensão do conjunto de instruções original do TALM com a criação de

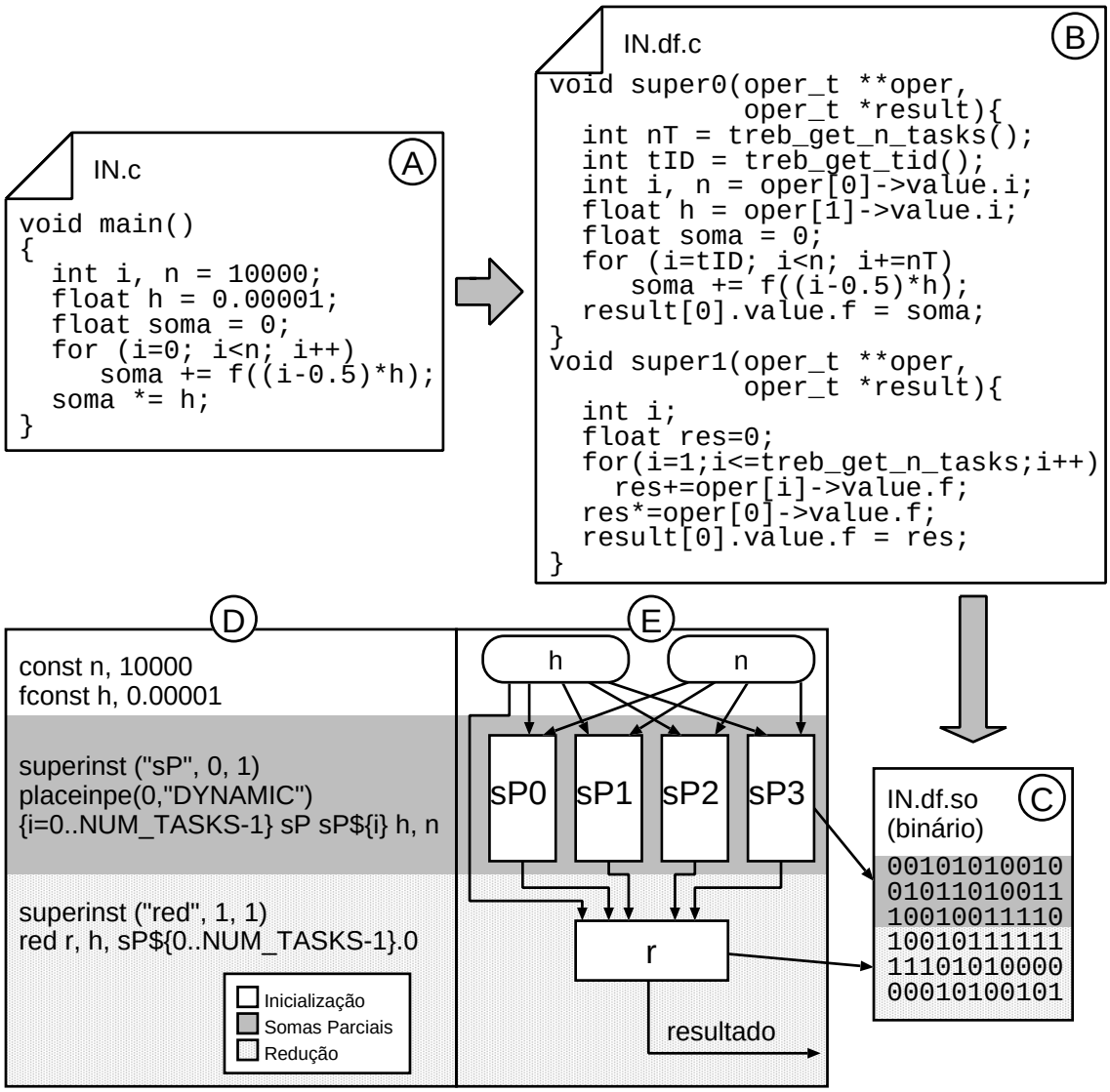

Figura 6.3: Exemplo de uso da *Trebuchet* para paralelizar uma integração numérica. O quadro A mostra o código em alto nível original (sequêncial) para a aplicação. O quadro B mostra o código que descreve o comportamento das super-instruções. O quadro C representa o binário resultante da compilação de B, usando um compilador tradicional. O quadro D mostra o código de montagem TALM que conecta as super-instruções, e o quadro E mostra o grafo *dataflow* associado.

super-instruções (como `sP` e `red`). No exemplo, a super-instrução `sP` é criada para executar uma integração parcial (uma porção do laço descrito em A). Cada instância de `sP` é sabe qual porção do laço irá executar através das funções `treb_get_tid()` e `treb_get_n_tasks()`. No exemplo, a aplicação é executada com quatro instâncias de `sP` e cada uma delas pode ser alocada a um EP (*thread*) diferente. A função `super0`, que implementa `sP`, define: *(i)* os operandos de entrada que serão enviados por outras instruções no grafo *dataflow* (E), e; *(ii)* os resultados produzidos pelas somas parciais. A super-instrução red é criada para fazer a operação de redução (somatório total) e a multiplicação por h. A função `super1` implementa `red`. Note que `sP` e `red` são super-instruções, que serão diretamente executadas pela máquina hospedeira (`super0` e `super1` na biblioteca). Observe como as sequencias de repetição são usadas para replicar a super-instrução `sP` e para passar os operandos de todas as instâncias de `sP` para `red`.

6.2.2 Paralelização com Desenrolamento de Laços

O exemplo anterior é apenas um dos possíveis métodos que pode ser usado para paralelizar uma aplicação com a *Trebuchet*. Uma outra alternativa é descrever o laço no grafo de fluxo de dados e desenrolá-lo, alocando cada cópia do corpo do laço em um EP diferente, como exibido na Figura 6.4. Este alternativa cria super-instruções de granularidade mais fina, o que poderia aumentar os custos de interpretação. Repare que neste exemplo, assume-se que o número de iterações do laço é múltiplo do número de tarefas (4, no exemplo). Por isto, não é necessário verificar para cada instância de `sP` se o valor de `i` é maior que `lim`. Neste método, quando há operações de memória dentro do laço, o programador pode criar super-instruções separadas paras as leituras em memória. Se o corpo do laço leva mais tempo pra executar que as leituras, o programador pode alocar as instruções de controle e as super-instruções de leitura em um único EP dedicado para esconder a latência de acesso à memória e manter a localidade de memória de forma simples.

Observe que o laço é desenrolado para explicitar o escalonamento das múltiplas cópias do corpo em EPs diferentes. No caso de a *Trebuchet* estar equipada com algum mecanismo de escalonamento dinâmico, o simples fato do controle do laço executar independentemente do corpo já exibe paralelismo, visto que múltiplas iterações podem estar prontas para execução em um dado momento, bastando que as instâncias da super-instrução do corpo do laço sejam alocadas à EPs distintos. Isto pode ser feito por um mecanismo de roubo de tarefas.

Repare que na linguagem de alto nível não há uma sintaxe para descrever desenrolamento de laços. Portanto, na versão atual do sistema, só é possível usar esta técnica manualmente.

NI.df.c

A

```
void super0(oper_t **oper, oper_t *result)
{
  int i = oper[0]->value.i;
  result[0].value.f=f((i-0.5)*0.00001);
}
void super1(oper_t **oper,
            oper_t *result){
  int i;
  float res=oper[0]->value.f;
  for(i=1;i=<treb_get_n_tasks;i++)
    res+=oper[i]->value.f;
  result[0].value.f = res;
}
```

NI.df

B

```
const lim, 10000
const i, 0
const soma, 0
inctag ilim, [lim, stlim.t]
inctag ii, [i, t3]
inctag isoma, [soma, r.0]
lthan comp, ilim, ii
steer stlim, comp, ilim
steer sti, comp, ii
steer stsoma, comp, isoma
superinst ("sP", 0, 1)
placeinpe(0,"STATIC"}
sP sP0, sti.t
{i=1..NUM_TASKS-1} addi t${i}, sti.t, ${i}
placeinpe(1,"DYNAMIC"}
{i=1..NUM_TASKS-1} sP sP${i}, t${i}
superinst ("red", 1, 1)
red r, h, stsoma.t, sP${0..NUM_TASKS-1}.0
divi resultado, stsoma.f, 100000
```

C

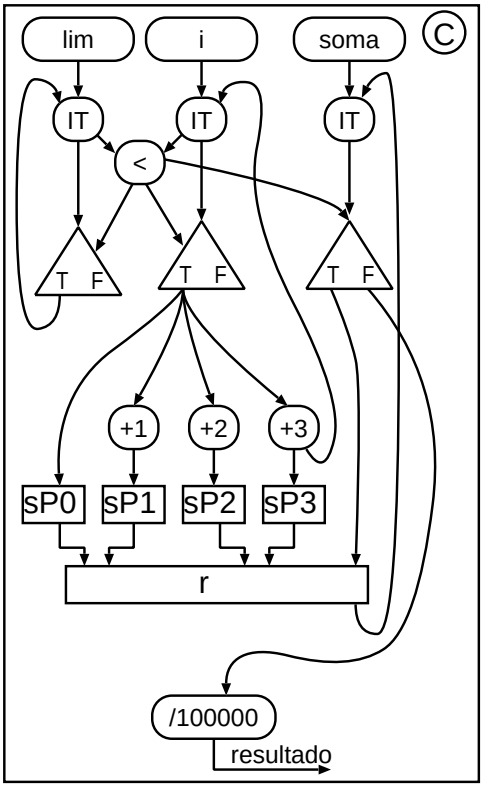

Figura 6.4: Exemplo de paralelização com desenrolamento de laço. O quadro A mostra o código das super-instruções, o quadro B mostra o código de montagem TALM que conecta as super-instruções, e o quadro C mostra o grafo *dataflow* associado.

Capítulo 7

Couillard: um Compilador para o TALM

A *Trebuchet* provê um ambiente de execução para máquinas com múltiplos núcleos de processamento, além de um montador e *loader*. No entanto, a transformação de código escrito na linguagem de alto nível do TALM para gerar o grafo de fluxo de dados (código de montagem) e a biblioteca de super-instruções não é uma tarefa trivial.

É proposto então o *Couillard*, um compilador `C` para execução guiada por fluxo de dados. O *Couillard* produz o código `C` correspondente à cada super-instrução que será em seguida compilada como um objeto compartilhado (em uma biblioteca dinamicamente ligada) para a arquitetura alvo e carregada pela *Trebuchet*. Além disto, o *Couillard* gera o código de montagem TALM correspondente ao grafo que conecta todas as super-instruções, segundo as especificações do usuário. Todo código que estiver fora do corpo das super-instruções, incluindo laços e condicionais, também será compilado para a linguagem de montagem TALM. O código de montagem gerado será usado pela *Trebuchet* para guiar a execução, de acordo com as regras *dataflow*.

Neste capítulo são discutidos alguns detalhes de implementação do *front-end* (analisador léxico e sintático) e do *back-end* (geração de código). Posteriormente, são exibidos uma série de exemplos de como são compiladas aplicações que usam diferentes técnicas de paralelismo, usando o *Couillard*.

7.1 *Front-end*

O *front-end* do *Couillard* foi desenvolvido usando a biblioteca PLY (Python Lex-Yacc) [63] e uma gramática que é um sub-conjunto do ANSI-C estendido com construções para super-instruções. Assume-se que super-instruções tomem a maior parte

do tempo de execução de uma aplicação, visto que instruções simples são mais usadas apenas para descrever as relações de controle entre super-instruções. Como o código das super-instruções é compilado por um compilador tradicional (como o gcc), o *Couillard* não precisa suportar a gramática ANSI-C completa. O *front-end* do compilador produz uma AST (*Auxiliary Syntax Tree*) que será processada para gerar uma representação na forma de grafo de fluxo de dados.

7.2 *Back-end*

O *back-end* do *Couillard* gera código de montagem para o TALM, código das super-instruções (para ser posteriormente compilado na forma de biblioteca dinamicamente ligada) e uma representação gráfica do grafo do programa, usando a notação Graphviz [64]. Depois de gerar a AST (*Abstract Syntax Tree*) de um programa, o *Couillard* monta o grafo de fluxo de dados do programa e gera três arquivos de saída:

1. Um arquivo `.dot` que descreve o grafo na notação Graphviz [64]. Este arquivo é usado para criar uma representação gráfica do grafo, usando as ferramentas do Graphviz. Embora este arquivo não seja necessário pela *Trebuchet*, ele pode ser útil para fins acadêmicos, ou para prover um visão mais clara do grafo para programadores que queiram fazer ajustes manuais em suas aplicações.

2. Um arquivo `.fl` descrevendo o grafo usando o conjunto de instruções da linguagem de montagem do TALM. Este arquivo será passado para o montador do TALM para produzir um arquivo binário `.flb` a ser carregado pela máquina virtual *Trebuchet*.

3. Um arquivo `.lib.c` descrevendo as super-instruções na forma de funções, em linguagem C, para ser compilado como uma biblioteca dinamicamente ligada, com o uso de um compilador C tradicional (como gcc). Todas as variáveis de entrada e saída descritas pelo programador são automaticamente declaradas e inicializadas dentro das funções geradas. Note também que o corpo das super-instruções não precisam passar pelo analisador sintático do *Couillard*. Eles são apenas tratados como valores associados aos nós das super-instruções na AST. Isso permitiu um enfoque apenas nas construções necessárias para conectar super-instruções em um grafo de fluxo de dados de granularidade grossa.

7.3 Exemplos de Compilação

Para um melhor entendimento acerca do trabalho realizado pelo *Couillard*, nesta seção são apresentados os resultados da compilação dos exemplos exibidos na Seção

5.4. Para todos os exemplos (com exceção do exemlo de pipeline não linear), são exibidos o código em alto nível sem o corpo das super-instruções (o código do corpo pode ser visto na Seção 5.4), o código gerado para a biblioteca de super-instruções, o código de montagem do TALM e a representação gráfica do grafo de fluxo de dados descrito pelo código de montagem.

7.3.1 Compilação de Aplicações do Tipo *Fork* e *Join* com Operação de Redução

As figuras 7.1, 7.2, 7.3 e 7.4 mostram o processo de compilação da aplicação descrita na Seção 5.4.1. Na Figura 7.1 é possível observar o código C escrito com a linguagem de alto nível do TALM. O código do corpo das super-instruções é omitido para uma melhor legibilidade (o código completo está descrito na Seção 5.4.1).

```
#BEGINBLOCK  //INCLUDES, FUNCTIONS E GLOBALS
    ...
#ENDBLOCK
int main(){
    int a=0; treb_parout int b;
    treb_super single input(a) output(a)
    #BEGINSUPER  //Código de inicialização
        ...
    #ENDSUPER

    treb_super parallel input(a) output(b)
    #BEGINSUPER  //Código de processamento
        ...
    #ENDSUPER

    treb_super single input(b::*) output(a)
    #BEGINSUPER //Redução (+)
        ...
    #ENDSUPER
    return 0;
}
```

Figura 7.1: Código de alto nível de uma aplicação regular com o TALM.

Na Figura 7.2 é possível observar o grafo de fluxo de dados dessa aplicação. Nesse exemplo, o grafo está sendo montado com quatro *threads* para a super-instrução paralela. É por isso que podem ser vistas quatro cópias dessa da super-instrução de processamento no grafo.

A Figura 7.3 mostra como a biblioteca de super-instruções é gerada pelo *Couillard*. Note que o corpo das super-instruções é simplesmente copiado para o arquivo da biblioteca e estendido com as declarações das funções e variáveis. Note como as variáveis de entrada das super-instruções são automaticamente declaradas e inicializadas como os valores provenientes das portas de entrada associadas (linhas 9 e 17). No caso da variável b, usada como entrada na super-instrução de redução ($b :: *$), re-

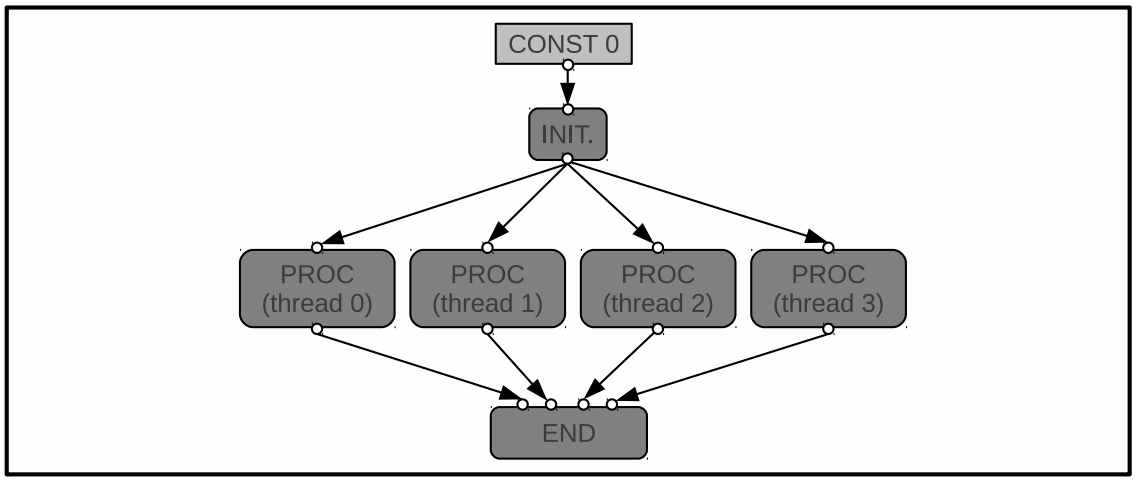

Figura 7.2: Grafo de fluxo de dados de uma aplicação regular

pare como b é declarado automaticamente como um vetor, alocado dinamicamente, para receber as diversas instâncias produzidas pela super-instrução paralela (linhas 25 à 30). As variáveis de saída também são declaradas automaticamente e as portas de saída são atribuídas com os valores das variáveis de saída associadas (linhas 12, 16, 20, 24 e 33).

A Figura 7.4 mostra o código de montagem gerado pelo compilador. Repare como a variável i é usada no código de montagem para descrever múltiplas instâncias da super-instrução paralela (linha 4). Além disso, veja como a macro `placeinpe` é usada para distribuir as instâncias dessa super-instrução em EPs consecutivos, começando pelo de número zero (linha 3). Note também como a sequência `{0..NUM_TASKS-1}` é usada para descrever em baixo nível (linha 5) a referência feita por $b :: *$ no código de alto nível.

```
1  #include "queue.h"
2  #include "interp.h"
3  extern int superargc;
4  extern char ** superargv;
5  //INCLUDES FUNCTIONS E GLOBALS
6     ...
7
8  super1(oper_t **oper, oper_t *result){
9     int a = oper[0]->value.i;
10    //Código de inicialização
11    ...
12    result[0].value.i = a;
13 }
14
15 super2(oper_t **oper, oper_t *result){
16    int b;
17    int a = oper[0]->value.i;
18    //Código de processamento
19    ...
20    result[0].value.i = b;
21 }
22
23 super3(oper_t **oper, oper_t *result){
24    int a;
25    int* b;
26    b = (int*) malloc(treb_get_n_tasks()*sizeof(int));
27    int mytask;
28    for(mytask=0; mytask<treb_get_n_tasks();mytask++){
29       b[mytask]=oper[mytask]->value.i;
30    }
31    //Código de finalização
32    ...
33    result[0].value.i = a;
34 }
```

Figura 7.3: Código da biblioteca de super-instruções de uma aplicação regular (gerado pelo *Couillard*)

```
1  const flowInstConst23693504, 0
2  super flowInstSuper23693288, 1, 1, flowInstConst23693504
3  placeinpe(0,"DYNAMIC")
4  {i=0..NUM_TASKS-1} superi flowInstPar23692568_t${i}, 2, 1,
      flowInstSuper23693288.0, ${i}
5  super flowInstSuper23022840, 3, 1,
      flowInstPar23692568_t${0..NUM_TASKS-1}.0
```

Figura 7.4: Código de montagem TALM de uma aplicação regular (gerado pelo *Couillard*)

7.3.2 Compilação de Aplicações com Técnica para Esconder Latência de Operações de Entrada e Saída

As figuras 7.5, 7.6, 7.7 e 7.8 mostram um exemplo de como compilar uma aplicação paralela que usa a técnica para esconder latência de operações de entrada e saída. A aplicação é a descrita na Seção 5.4.2. Na Figura 7.5 é mostrado o esqueleto do código com as diferentes etapas realizadas por super-instruções (o código completo pode ser visto na Seção 5.4.2).

```
#BEGINBLOCK
   //INCLUDES, FUNCTIONS E GLOBAIS ...
#ENDBLOCK
int main(){
   int a=0, e; treb_parout int b,c,d;
   treb_super single input(a) output(a)
   #BEGINSUPER
      //Código de inicialização ...
   #ENDSUPER
   treb_super parallel
      input(starter.a, local.b::(mytid-1)) output(b)
   #BEGINSUPER
      //Leitura ...
   #ENDSUPER
   treb_super parallel input(b::mytid) output(c)
   #BEGINSUPER
      //Código de processamento ...
   #ENDSUPER
   treb_super parallel input(c::mytid, local.d::(mytid-1))
      output(d)
   #BEGINSUPER
      //Escrita ...
   #ENDSUPER
   treb_super single input(d::lasttid) output(e)
   #BEGINSUPER
      //Código de finalização ...
   #ENDSUPER
   return 0;
}
```

Figura 7.5: Código de alto nível do exemplo de como esconder latência de E/S.

A Figura 7.6 mostra o grafo de fluxo de dados associado, gerado pelo *Couillard*. Neste caso, o grafo foi criado com quatro instâncias das super-instruções paralelas. Note como as dependências locais serializam a execução das etapas de escrita e leitura, embora as diferentes instâncias da etapa de processamento sejam liberadas individualmente por cada instância da etapa de leitura.

A Figura 7.7 mostra o código gerado para a biblioteca de super-instruções. Note a omissão do recebimento do operando a na super-instrução de leitura (linhas 11 à 17). Como o operando a foi definido como `starter`, ele não estará presente em algumas instâncias dessa super-instrução. Neste caso tanto o operando a quanto o operando local b servem apenas como arestas de precedência e não poderão ser usados para transportar dados. Observe na linha 27 como é recebido o operando

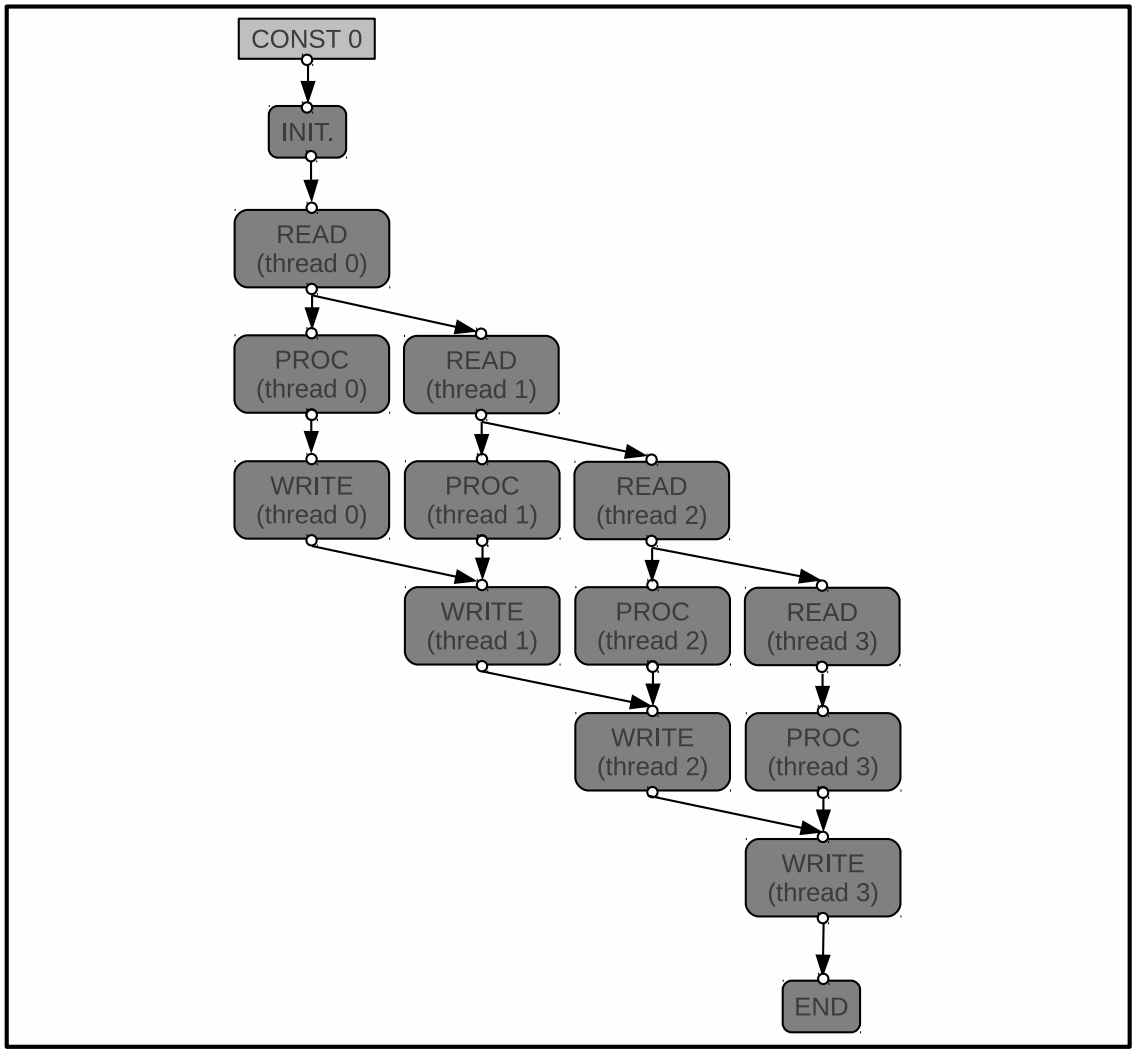

Figura 7.6: Grafo de fluxo de dados do exemplo de como esconder latência de E/S.

local d na super-instrução de escrita. Neste caso, como ele é o único operando opcional poderia sim ser utilizado para transportar dados.

```
1  #include "queue.h"
2  #include "interp.h"
3  extern int superargc;
4  extern char ** superargv;
5     //INCLUDES, FUNCTIONS E GLOBALS ...
6  super1(oper_t **oper, oper_t *result){
7     int a  = oper[0]->value.i;
8     //Código de inicialização ...
9     result[0].value.i = a;
10 }
11 super2(oper_t **oper, oper_t *result){
12    int b;
13    if (oper[0])
14       b = oper[0]->value.i;
15    //Leitura ...
16    result[0].value.i = b;
17 }
18 super3(oper_t **oper, oper_t *result){
19    int c;
20    int b = oper[0]->value.i;
21    //Código de processamento ...
22    result[0].value.i = c;
23 }
24 super4(oper_t **oper, oper_t *result){
25    int c  = oper[0]->value.i;
26    int d;
27    if (oper[1])
28       d = oper[1]->value.i;
29    //Escrita ...
30    result[0].value.i = d;
31 }
32 super5(oper_t **oper, oper_t *result){
33    int e;
34    int d  = oper[0]->value.i;
35    //Código de finalização ...
36    result[0].value.i = e;
37 }
```

Figura 7.7: Código da biblioteca de super-instruções do exemplo de como esconder latência de E/S.

A Figura 7.8 mostra o código de montagem da aplicação. Note como são usadas duas instruções de montagem com operandos de entrada diferentes para descrever os operandos `starter` e `local` da super-instrução de leitura (linhas 3 à 6), incluindo a alocação dessas instâncias em diferentes EPs (macro `placeinpe`). O mesmo acontece para a super-instrução de escrita (linhas 9 à 12).

```
1  const flowInstConst42094464, 0
2  super flowInstSuper42087920, 1, 1, flowInstConst42094464
3  placeinpe(0,"DYNAMIC")
4  {i=0..0} superi flowInstPar42088424_t${i}, 2, 1,
      flowInstSuper42087920.0, ${i}
5  placeinpe(1,"DYNAMIC")
6  {i=1..NUM_TASKS-1} superi flowInstPar42088424_t${i}, 2, 1,
      flowInstPar42088424_t${(i-(1%NUM_TASKS))%NUM_TASKS}.0, ${i}
7  placeinpe(0,"DYNAMIC")
8  {i=0..NUM_TASKS-1} superi flowInstPar42087056_t${i}, 3, 1,
      flowInstPar42088424_t${i}.0, ${i}
9  placeinpe(0,"DYNAMIC")
10 {i=0..0} superi flowInstPar42079440_t${i}, 4, 1,
      flowInstPar42087056_t${i}.0, ${i}
11 placeinpe(1,"DYNAMIC")
12 {i=1..NUM_TASKS-1} superi flowInstPar42079440_t${i}, 4, 1,
      flowInstPar42087056_t${i}.0,
      flowInstPar42079440_t${(i-(1%NUM_TASKS))%NUM_TASKS}.0, ${i}
13 super flowInstSuper42082104, 5, 1,
      flowInstPar42079440_t${NUM_TASKS-1..NUM_TASKS-1}.0
```

Figura 7.8: Código de montagem TALM do exemplo de como esconder latência de E/S.

7.3.3 Compilação de *Pipelines* Lineares

As figuras 7.9, 7.10, 7.11 e 7.12 mostram um exemplo de como compilar uma aplicação paralela implementada como um *pipeline* linear. A aplicação é descrita na Seção 5.4.3. Na Figura 7.9 é mostrado o esqueleto do código com as diferentes etapas realizadas por super-instruções (o código completo pode ser visto na Seção 5.4.3). A Figura 7.10 mostra o grafo de fluxo de dados associado, gerado pelo *Couillard*. Note como os operandos b e c são usados para serializar, respectivamente, os estágios de leitura e escrita, e deixando o estágio de processamento independente.A Figura 7.11 mostra o código gerado para a biblioteca de super-instruções. A Figura 7.12 mostra o código de montagem associado.

```
#BEGINBLOCK
   //INCS, FUNCS E GLOBAIS...
#ENDBLOCK
int main(){
   void *da, *db;
   treb_parout void *dc;
   int x=0, a=0, b, c=0, d;
   treb_super single input(a) output(b)
   #BEGINSUPER
      //Inicialização ...
   #ENDSUPER
   while(x<b){
      treb_super single input(b) output(b, da, db)
      #BEGINSUPER
         //Leitura ...
      #ENDSUPER
      treb_super parallel input(da, db) output(dc)
      #BEGINSUPER
         //Processamento ...
      #ENDSUPER
      treb_super single input(da, db, dc::*, c)    output(c)
      #BEGINSUPER
         //Escrita ...
      #ENDSUPER
   }
   treb_super single input(c) output(d)
   #BEGINSUPER
      //Finalização ...
   #ENDSUPER
   return 0;
}
```

Figura 7.9: Código de alto nível de um *pipeline* linear.

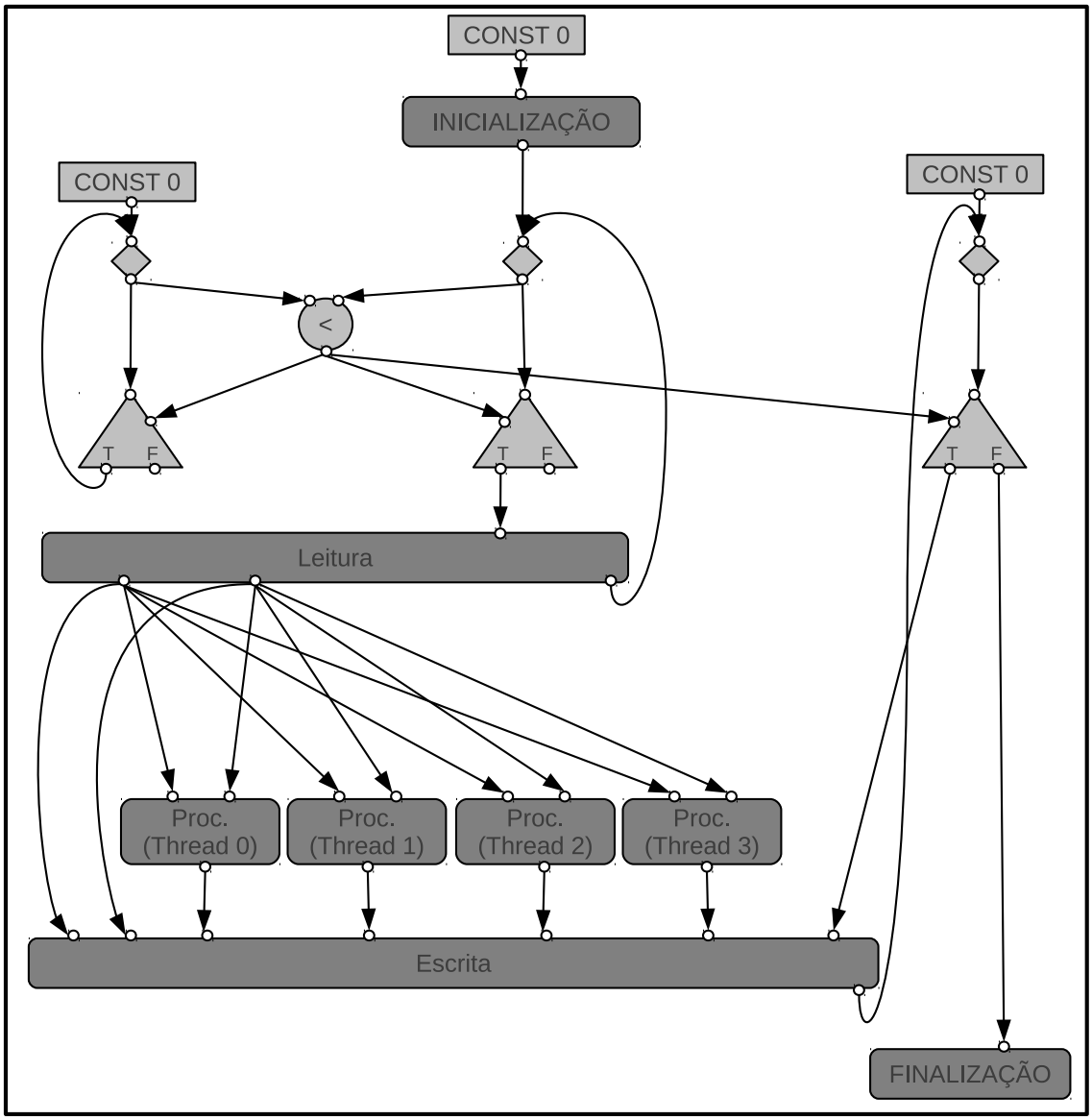

Figura 7.10: Grafo de fluxo de dados de um *pipeline* linear.

```
#include "queue.h"
#include "interp.h"
extern int superargc;
extern char ** superargv;
   //INCS, FUNCS E GLOBAIS...
super1(oper_t **oper, oper_t *result){
   int b; int a = oper[0]->value.i;
   //Inicialização ...
   result[0].value.i = b;
}
super2(oper_t **oper, oper_t *result){
   void* da; void* db;
   int b = oper[0]->value.i;
   //Leitura ...
   result[0].value.i = b;
   result[1].value.p = da;
   result[2].value.p = db;
}
super3(oper_t **oper, oper_t *result){
   void* dc;
   void* da = oper[0]->value.p;
   void* db = oper[1]->value.p;
   //Processamento ...
   result[0].value.p = dc;
}
super4(oper_t **oper, oper_t *result){
   void* da = oper[0]->value.p;
   void* db = oper[1]->value.p; void** dc;
   dc = (void**) malloc(treb_get_n_tasks()*sizeof(void*));
   int mytask;
   for(mytask=0; mytask<treb_get_n_tasks(); mytask++){
      dc[mytask]=oper[2+mytask]->value.p;
   }
   int c  = oper[2+(1*treb_get_n_tasks())]->value.i;
   //Escrita ...
   result[0].value.i = c;
}
super5(oper_t **oper, oper_t *result){
   int d; int c  = oper[0]->value.i;
   //Finalização ...
   result[0].value.i = d;
}
```

Figura 7.11: Código da biblioteca de super-instruções de um *pipeline* linear.

```
const flowInstConst33410800, 0
const flowInstConst34081752, 0
const flowInstConst34081464, 0
super flowInstSuper34080528, 1, 1, flowInstConst34081752
inctag flowInstIncTag33401096,
   [flowInstConst33410800, flowInstSteer33458008.t]
inctag flowInstIncTag33402824,
   [flowInstSuper34080528.0, flowInstSuper33433432.0]
lthan flowInstBinop33401528, flowInstIncTag33401096,
   flowInstIncTag33402824
steer flowInstSteer33431848, flowInstBinop33401528,
   flowInstIncTag33402824
super flowInstSuper33433432, 2, 3, flowInstSteer33431848.t
placeinpe(0,"DYNAMIC")
{i=0..NUM_TASKS-1} superi flowInstPar33434440_t${i}, 3, 1,
   flowInstSuper33433432.1, flowInstSuper33433432.2, ${i}
inctag flowInstIncTag33434224, [flowInstConst34081464,
   flowInstSuper33457072.0]
steer flowInstSteer33434800, flowInstBinop33401528,
   flowInstIncTag33434224
super flowInstSuper33457072, 4, 1, flowInstSuper33433432.1,
   flowInstSuper33433432.2,
   flowInstPar33434440_t${0..NUM_TASKS-1}.0,
   flowInstSteer33434800.t
steer flowInstSteer33458008, flowInstBinop33401528,
   flowInstIncTag33401096
super flowInstSuper33456568, 5, 1, flowInstSteer33434800.f
```

Figura 7.12: Código de montagem TALM de um *pipeline* linear.

7.3.4 Compilação de *Pipelines* não Lineares

A Figura 7.13 o grafo de fluxo de dados do *pipeline* paralelo não linear (onde os estágios podem executar condicionalmente) descrito na Seção 5.4.4. O código de montagem e o código da biblioteca de super-instruções são omitidos por questões de legibilidade e por não apresentarem novidades (do ponto de vista de compilação) em relação aos exemplos anteriores.

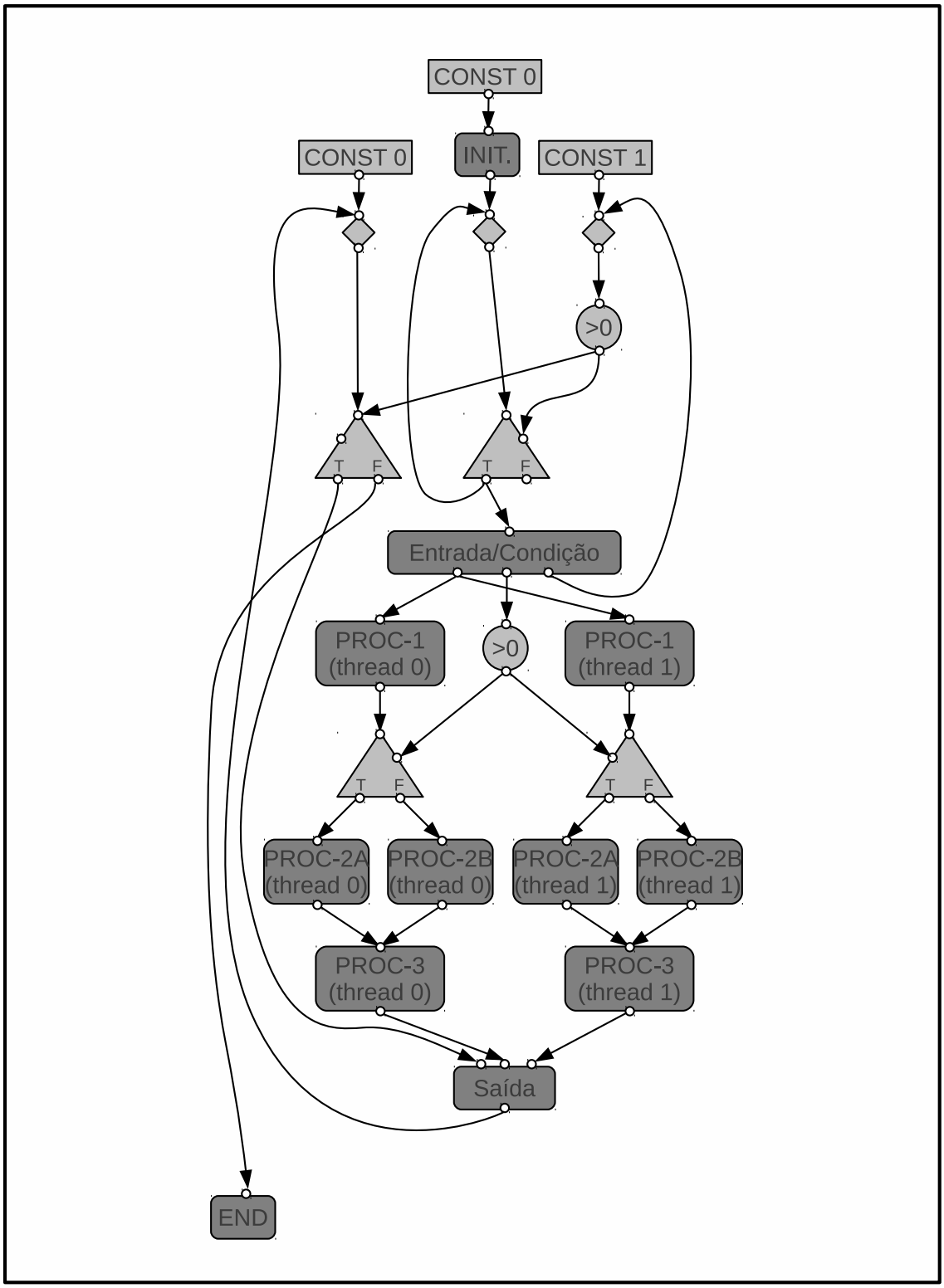

Figura 7.13: Grafo de fluxo de dados de um *pipeline* não linear com o TALM.

Capítulo 8

Experimentos e Resultados

Para avaliar o TALM, a *Trebuchet* e o *Couillard* são feitos dois conjuntos de experimentos. O primeiro conjunto visa avaliar o desempenho da máquina virtual *Trebuchet* em aplicações regulares, escritas manualmente com a linguagem de montagem TALM, e comparando o desempenho com o *OpenMP*. O segundo conjunto tem o objetivo de avaliar a facilidade e versatilidade de programação com a linguagem de alto nível do TALM e o desempenho do código gerado pelo *Couillard* em aplicação irregulares, ou com técnicas de paralelismo mais complexas.

Nesse capítulo são apresentados os *benckmarks* utilizados, os procedimentos experimentais adotados, e os resultados experimentais obtidos.

8.1 Avaliação da *Trebuchet* com Programação Manual

Para avaliar o desempenho da *Trebuchet*, foi manualmente paralelizado, ou seja, sem uso do THLL (apenas da linguagem de montagem), um conjunto de sete aplicações: uma aplicação que faz o cálculo do determinante de matrizes (DET), uma aplicação que faz multiplicação de matrizes (MXM), uma aplicação de renderização de cenas com *ray-tracing* (RT), o Equake do SpecOMP 2001 [18], o IS do NPB3.0-OMP [19] e também o Mandelbrot (Mandel) do *OpenMP Source Code Repository* [20]. Uma comparação é feita com a versão *OpenMP* dessas aplicações para verificar o quão próximo de soluções em estado-da-arte está o desempenho da *Trebuchet*.

8.1.1 Aplicações e Estratégia de Paralelização

A aplicação de cálculo de determinantes de matrizes utiliza o método clássico de cofatores e determinantes recursivo de submatrizes. Nesta aplicação, tarefas paralelas são criadas para calcular os determinantes das submatrizes do primeiro nível de re-

cursão. Cada tarefa é responsável pelo cálculo correspondente a uma parte da árvore de recursão. Para a versão *Trebuchet*, uma super-instrução é criada para a leitura da matriz de um arquivo e outra para a impressão do resultado, enquanto que o laço que executa o primeiro nível de recursão é transformado em outra super-instrução cujas instâncias dividirão a tarefa dos cálculos dos determinantes das submatrizes do nível dois em diante. A versão *OpenMP* foi feita de maneira análoga, com o uso dos *pragmas* necessários.

Para o programa de multiplicação de matrizes o código de montagem *Dataflow* inclui a descrição de uma função na qual são executadas super-instruções que paralelizam o laço externo que varre as matrizes fazendo a multiplicação. Cada instância da super-instrução fica responsável pela produção de algumas linhas na matriz resultado.

O *ray-tracing* de esferas faz a renderização de uma cena com diversas esferas cujas informações de posição, cor, reflexividade e opacidade são definidas em um arquivo. O resultado é uma imagem com resolução de 800x600 pixels salva em arquivo. A renderização é feita pixel por pixel, varrendo linhas e colunas em dois laços aninhados. Super-instruções são criadas para leitura do arquivo com as esferas e gravação da imagem de saída. A paralelização da porção de renderização é feita de duas maneiras:

- *Dataflow* de granularidade fina: o laço externo é descrito em *dataflow*, usando a linguagem de montagem do TALM, e uma super-instrução foi criada para descrever o laço interno, para avaliar o impacto causado pela interpretação de instruções mais simples.

- *Dataflow* de granularidade grossa: ambos os laços (interno e externo) são incluídos em uma super-instrução. Embora os custos de interpretação sejam menores, laços aninhados aumentam a quantidade de desvios de controle a serem executados na máquina hospedeira. Além disso, laços aninhados podem aumentar o efeito das predições de desvio incorretas e prejudicar o desempenho da aplicação.

Para ambas as soluções, diferentes instâncias da super-instrução de renderização trabalham para produzir diferentes grupos de colunas da cena final.

Equake é uma aplicação de simulação de propagação de ondas sísmicas. A maior parte do seu tempo de execução é gasto com as funções `smvp()` e `main()`. A paralelização foi baseada na versão *OpenMP* da aplicação, disponível no SpecOMP 2001 [18]. Para a versão TALM, super-instruções foram criadas de acordo com as regiões paralelas definidas na versão *OpenMP*.

IS é um ordenador de número inteiros longos. Este *kernel* realiza uma operação de ordenação que é importante em códigos de "*particle method*". Ele testa a velo-

cidade da computação de inteiros, além do desempenho da comunicação. Como foi utilizada a versão NPB3.0-OMP, as regiões paralelas já estavam descritas no código. Sendo assim, foi apenas necessário criar as super-instruções de acordo.

O LU faz uma redução LU de uma matriz 2D densa. A paralelização foi feita de forma que cada tarefa paralela compute um conjunto de linhas da matriz, como já feito na versão *OpenMP*.

O Mandelbrot computa uma estimativa da área do conjunto de Mandelbrot usando *MonteCarlo sampling*. A aplicação foi paralelizada dando para cada tarefa paralela um conjunto de pontos a ser analisado, como na versão OpenMP. Além disso, também foi preparada uma versão do Mandelbrot usando a técnica de desenrolamento de laços (descrita na Seção 6.2.2), onde cada EP recebe uma cópia do corpo do laço. Nessa versão foram criadas super-instruções separadas para fazer os acessos à memória.

8.1.2 Procedimentos Experimentais

Cada experimento foi executado múltiplas vezes (de 10 até 10 mil, dependendo do tamanho da aplicação e da entrada) para remover discrepâncias nos tempos de execução. Para as aplicações DET, MXM e RT, foram produzidas versões paralelas para 1,2 e 8 *threads* (ou EPs), enquanto que para o Equake, IS, LU e Mandelbrot foram preparadas versões de 1 até 8 *threads*. Além disto, é efetuada uma comparação com *OpenMP* para cada cenário. A Tabela 8.1 exibe o tamanho das entradas usadas para cada aplicação.

Aplicação	Tamanho da entrada
MXM	duas matrizes de 2500x2500 elementos
DET	matrizes quadradas de dimensão 9x9 até 13x13
RT	cena com 800x600 *pixels* contendo 4006 esferas
Equake	Entrada de Referência do SpecOMP2001
IS	Entrada classe C do NPB3.0-OMP
LU	matriz de 5000x5000 elementos
Mandel	1 milhão de pontos

Tabela 8.1: Tamanho da entrada para cada aplicação.

A Tabela 8.2 mostra o número total de instruções *dataflow* de granularidade fina (do conjunto de instruções nativo do TALM) e o número de super-instruções usadas nas versões paralelas de cada aplicação. A variável n representa o número de *threads*. Os números mostram que poucas instruções de granularidade fina são usadas, o que resulta em baixos custos com interpretação.

Foi usada uma máquina CMP para executar esses experimentos: um Intel®

Aplicação	Instruções	Super-Instruções
MXM	3	2+n
DET	0	2+n
RT coarse-grained	0	2+n
RT fine-grained	15+4n	2+n
Equake	6	8+6n
IS	20	9+6n
LU	5	3+n
Mandel	0	2+n

Tabela 8.2: Contagem estática de instruções *dataflow* para cada aplicação (n=número de *threads*).

Core^{TM}i7 920 (2,67 GHz), com quatro núcleos de processamento, cada um com *simultaneous multi-threading* (e portanto 2 núcleos lógicos em cada núcleo físico). A máquina tem 12 GB de memória RAM DDR-3 1066 MHz *Triple Channel* (6x2GB). O sistema operacional utilizado foi GNU/Linux (kernel 2.6.27-7 64 bits).

8.1.3 Resultados

A Figura 8.1 apresenta os resultados para o DET. Os cenários de execução *dataflow* e *OpenMP* são identificados por DF-X and OMP-X, respectivamente, onde X representa o número de *threads*. Experimentos foram realizados variando o tamanho da matriz de entrada (de ordem 9 até 13) para permitir avaliar a carga de computação necessária para compensar os custos comunicação para as versões *Trebuchet* e *OpenMP*. Os resultados mostram que a versão *dataflow* tem desempenho melhor que a versão *OpenMP*. Além disso a versão *OpenMP* tem desempenho mais afetado que a versão *dataflow* quando a carga de computação é diminuida. A versão *dataflow* apresenta, no melhor caso, um *speedup* de 2,00, enquanto a versão *OpenMP* apresenta *speedups* de 1,94. Para matrizes de ordem menor que 9, o tempo de execução é muito pequeno (entre mil e 6 mil nanosegundos). Isto faz com que os tempos aferidos sejam muito afetados por processos de sistema que podem entrar em execução.

A Figura 8.2 mostra os resultados para o MXM. Os resultados mostram *speedups* de até 4.16, quando comparados com a versão sequencial. Além disso, o desempenho da versão TALM se mostrou equivalente à versão *OpenMP*.

A Figura 8.3 mostra os resultados para o RT. *Speedups* de 4.39 são obtidos tanto para a versão *dataflow* quanto para a versão *OpenMP*. A comparação entre as versões de granularidade grossa e fina mostra que os custos de interpretação não foram significativos. Esperava-se que a versão de granularidade fina apresentasse um desempenho inferior, no entanto, com a descrição do laço externo em *dataflow* (o laço aninhado foi removido da super-instrução), pode ter havido uma redução nas

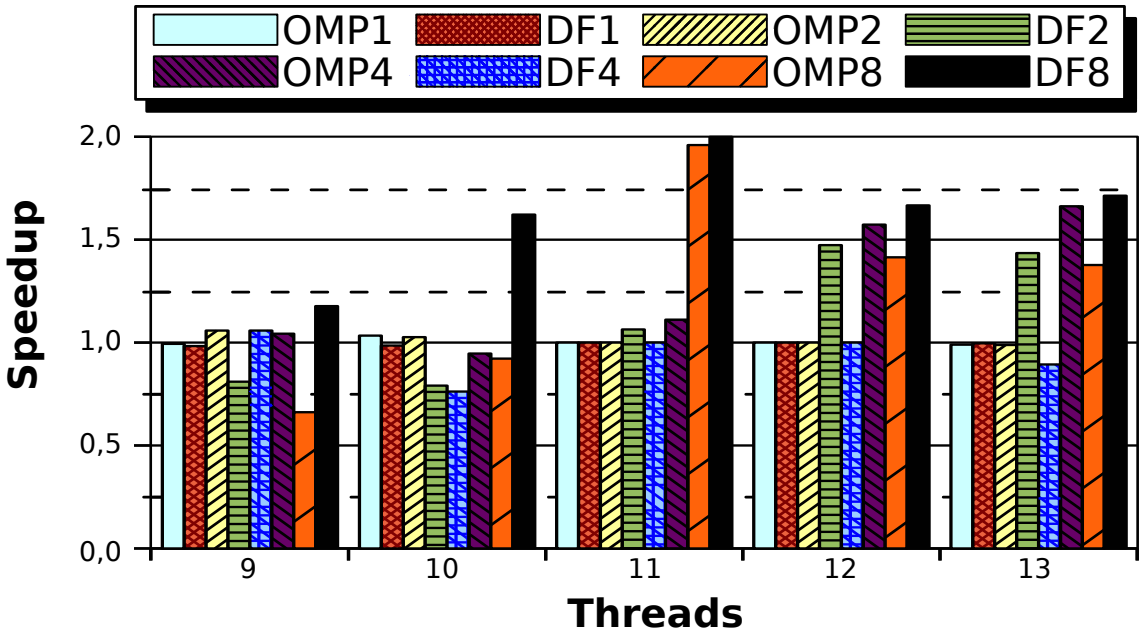

Figura 8.1: Resultados para o DET

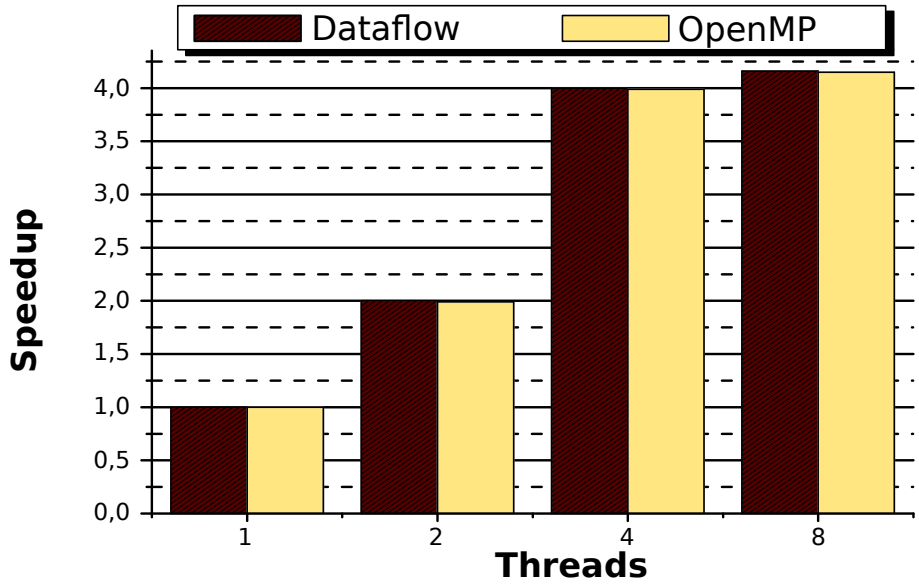

Figura 8.2: Resultados para o MXM.

predições de desvio incorretas na máquina hospedeira.

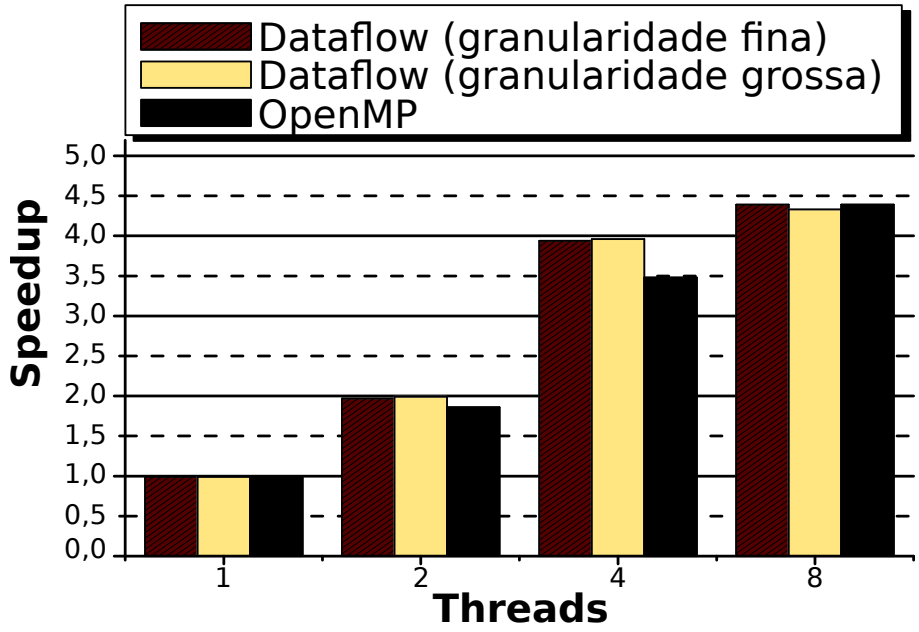

Figura 8.3: Resultados para o RT.

Os resultados para o Equake são exibidos na Figura 8.4. Observa-se que a versão *dataflow* tem desempenho superior à versão *OpenMP*. Os *speedups*, no melhor caso, para as versões *dataflow* e *OpenMP* são, respectivamente, 3,61 e 3,14.

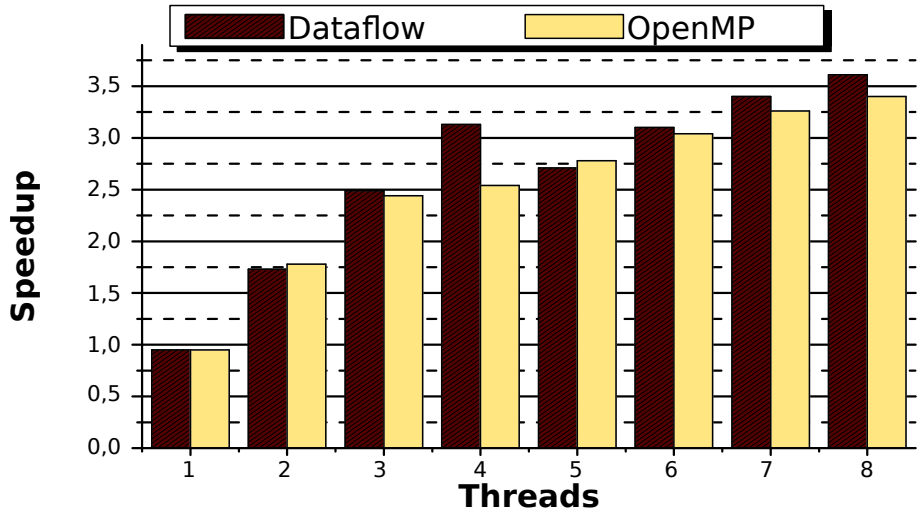

Figura 8.4: Resultados do Equake.

A Figura 8.5 mostra os resultados para o IS. A versão *dataflow* apresenta desempenho ligeiramente inferior ao da versão *OpenMP*. Os *speedups*, no melhor caso, para as versões *dataflow* e *OpenMP* são, respectivamente, 3,00 e 3,11.

A Figura 8.6 exibe os resultados para o LU. Observa-se um desempenho equivalente entre as versões *dataflow* e *OpenMP*. Os *speedups*, no melhor caso, tanto para

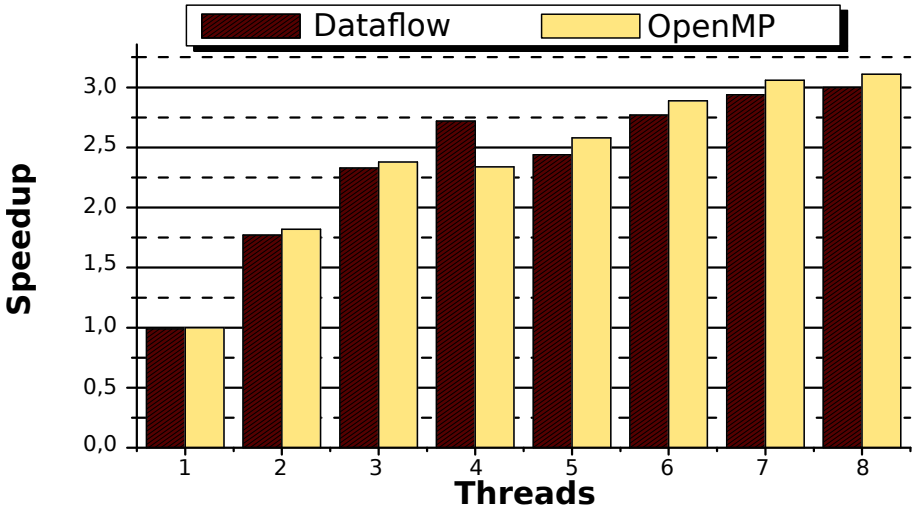

Figura 8.5: Resultados do IS.

a versão *dataflow* quanto para a versão *OpenMP* são de 2,19.

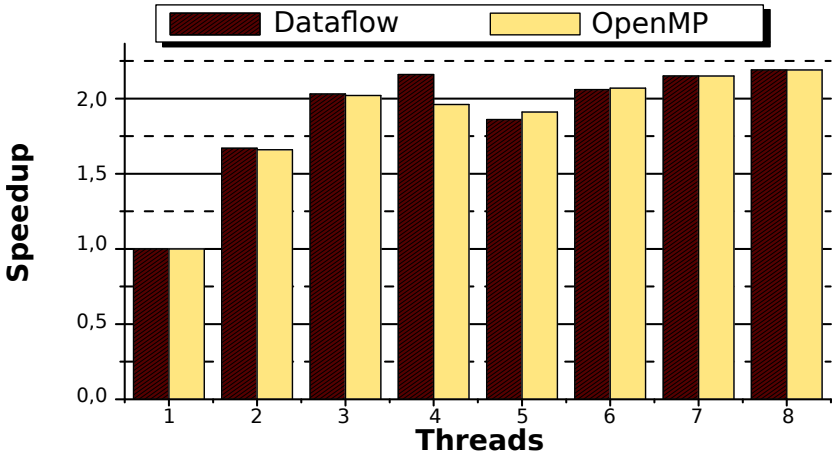

Figura 8.6: Resultados para o LU.

A Figura 8.7 exibe os resultados para o Mandelbrot. Observa-se que a versão *dataflow* tem desempenho superior à versão *OpenMP*. Os *speedups*, no melhor caso, para as versões *dataflow* e *OpenMP* são, respectivamente, 7,16 e 7.13. Note que mesmo com o uso de uma técnica ingênua de paralelização (desenrolamento de laço) foi possível obter *speedups* significativos.

8.2 Avaliação da Linguagem de Alto Nível do TALM e do *Couillard*

Neste segundo conjunto de experimentos, o objetivo é obter bom desempenho em aplicações reais usando técnicas de programação paralela mais sofisticadas, com au-

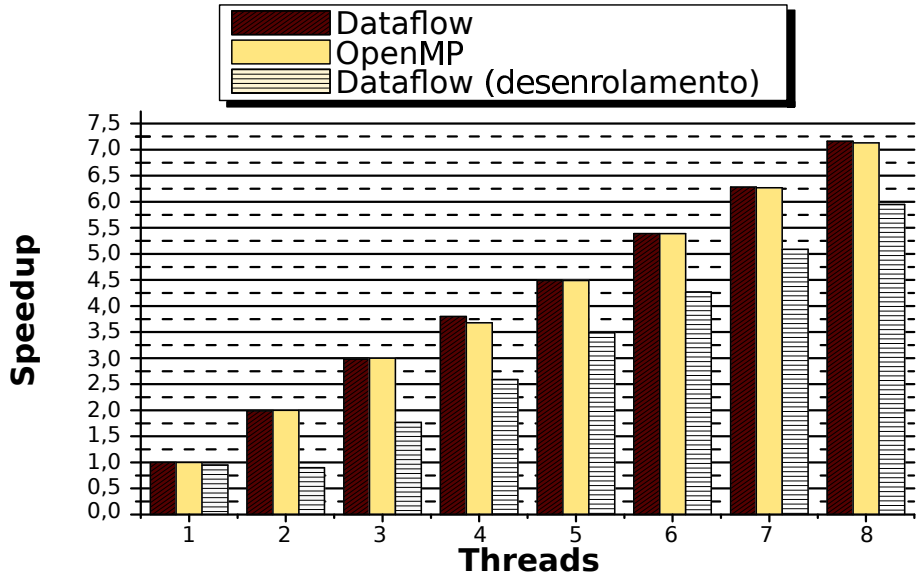

Figura 8.7: Resultados para o Mandelbrot.

xílio da linguagem de alto nível do TALM e do compilador *Couillard*. O estudo mostra o comportamento do TALM em duas aplicações em estado-da-arte, pertencente ao conjunto de *benckmarks* PARSEC [21]: Blackscholes e Ferret.

8.2.1 Aplicações e Estratégia de Paralelização

O Blackscholes é uma aplicação regular que calcula analiticamente os preços de um portfólio de opções Europeias, usando a equação parcial diferencial de Black-Scholes (PDE). A aplicação lê um arquivo contendo o portfólio. A PDE de cada opção do portfólio pode ser calculada independentemente. Um arquivo de saída é produzido com os resultados para todas as opções do portfólio. São criadas super-instruções *single* para leitura e escrita dos arquivos de entrada e saída, respectivamente. É também criada uma super-instrução *parallel* para a etapa de processamento. Cada instância dessa super-instrução fará o processamento de um grupo de opções.

O PARSEC disponibiliza versões *OpenMP*, *Pthreads* e *TBB* desta aplicação. A aplicação é paralelizada para o TALM seguindo os mesmos padrões de paralelismo das versões disponíveis no PARSEC. No entanto, foi observado que é possível esconder a latência das operações de entrada e saída e ainda aumentar a localidade de memória se as super-instruções de entrada e saída forem do tipo *parallel*. Sendo assim, também foi feita uma versão do Blackscholes de acordo com o exemplo mostrado na Seção 7.3.2.

A segunda aplicação considerada, o Ferret, possui carga irregular. Ela é baseada no *Ferret toolkit* que é usado para busca de similaridade, baseado em conteúdo. Foi desenvolvida na Universidade de Princeton e representa mecanismos de busca

emergentes e de última geração para documentos não textuais (imagens, no caso da versão do PARSEC). O Ferret é paralelizado usando o modelo *pipeline* e apenas uma versão *Pthreads* é disponibilizada com o PARSEC. No entanto, uma versão *TBB* foi obtida [65] e também utilizada no experimento.

Primeiramente, foi observado que o tamanho das tarefas do Ferret é ligeiramente pequeno, e poderia resultar em altos custos de interpretação pela máquina virtual, especialmente quando um grande número de núcleos de processamento fosse utilizado, onde os custos de comunicação se tornam mais aparentes. Sendo assim, a aplicação foi adaptada para processar blocos de cinco imagens por tarefa, ao invés de uma.

A versão paralela do Ferret para o TALM também usa o padrão *pipeline*, como descrito na Seção 7.3.3. A versão para o TALM possui 5 super-instruções:

- Uma super-instrução *single* de inicialização, onde são abertos os arquivos de entrada e saída.
- Uma super-instrução *single* de leitura que representa o primeiro estágio do pipeline.
- Uma super-instrução *parallel* de processamento que representa o segundo estágio.
- Uma super-instrução *single* de escrita que representa o terceiro estágio.
- Uma super-instrução *single* de finalização, onde os arquivos são fechados.

Esta versão do Ferret se apoia no mecanismo de roubo de tarefas (descrito na Seção 6.1.3) para fazer o balanceamento dinâmico de carga.

Também foi preparada uma versão manualmente otimizada do Ferret, usando *over-subscription* (criação de mais *threads* do que núcleos de processamento disponíveis) e contando com o escalonador do sistema operacional, ao invés do mecanismo de roubo de tarefas, para fazer balanceamento de carga. Note que as otimizações foram feitas manualmente, sem o uso do *Couillard*. Os estágios de entrada e saída foram transformados em super-instruções *parallel*, e, ao invés de ter um padrão *fork-join* na paralelização, o laço do *pipeline* foi modificado para escolher uma linha de execução em cada iteração, mas deixando o controle do laço dependente apenas da super-instrução de leitura. Sendo assim, o escalonamento é manualmente controlado.

A Figura 8.8 mostra o grafo de fluxo de dados da versão manual do Ferret. Neste exemplo, o grafo mostra a versão com duas *threads*. Note como em cada iteração do laço, apenas uma linha de execução é ativada. As etapas de leitura e escrita devem ser serializadas, mas a etapa de processamento pode ocorrer em qualquer ordem. As arestas tracejadas mostram esse comportamento. Em uma iteração i, as etapas de

leitura e escrita da *thread i mod nt* (*mod* é a operação de resto da divisão e *nt* é o número de *threads*) vão ativar a execução da iteração seguinte. Note também que, embora existam múltiplas instâncias da super-instrução de finalização, somente uma delas será executada, pois em cada iteração só uma *thread* é executada.

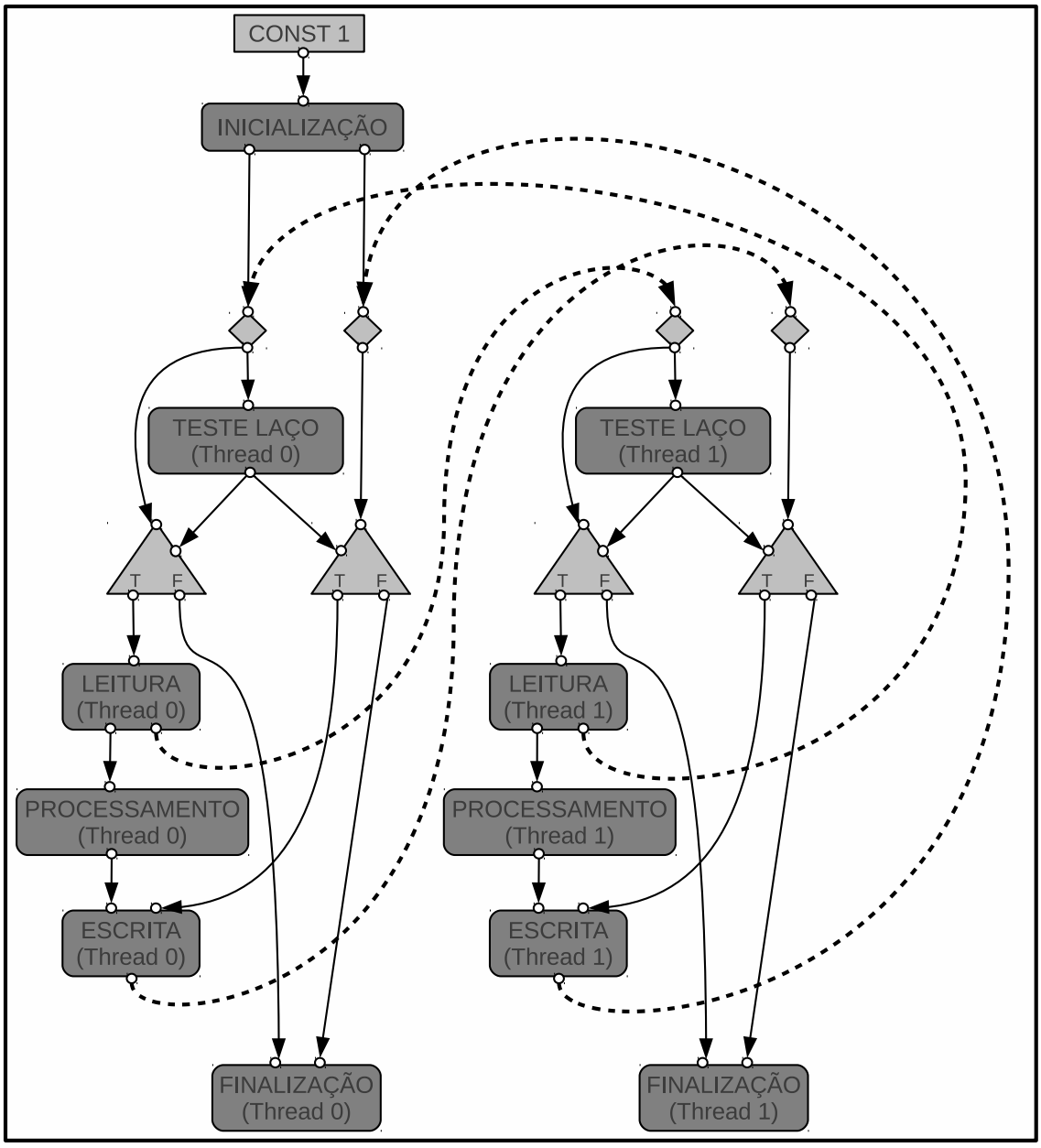

Figura 8.8: Grafo de fluxo de dados da versão manual do Ferret.

8.2.2 Procedimentos Experimentais

Cada experimento foi executado 5 vezes para diminuir discrepâncias no tempo de execução. A plataforma de execução paralela usada foi uma máquina como quatro *chips* AMD *Six-Core* Opteron^{TM}8425 HE 2100 MHz (total de 24 núcleos) e 64 GB

de RAM DDR-2 667MHz (16x4GB). O sistema operacional usado foi o GNU/Linux (kernel 2.6.31.5-127 64 bits).

A Tabela 8.3 mostra o tamanho da entrada usada para cada aplicação. No caso da versão manual do Ferret com *over-subscription*, foram usadas três vezes mais *threads* do que o número de núcleos usados em cada experimento. Para isso, foi necessário ajustar o mecanismo de *scheduling affinity* da *Trebuchet* para usar apenas o número de núcleos necessário em cada cenário.

Aplicação	Tamanho da entrada
Blacksholes	10 milhões de opções
Ferret	3500 imagens

Tabela 8.3: Tamanho da entrada para cada aplicação.

8.2.3 Resultados

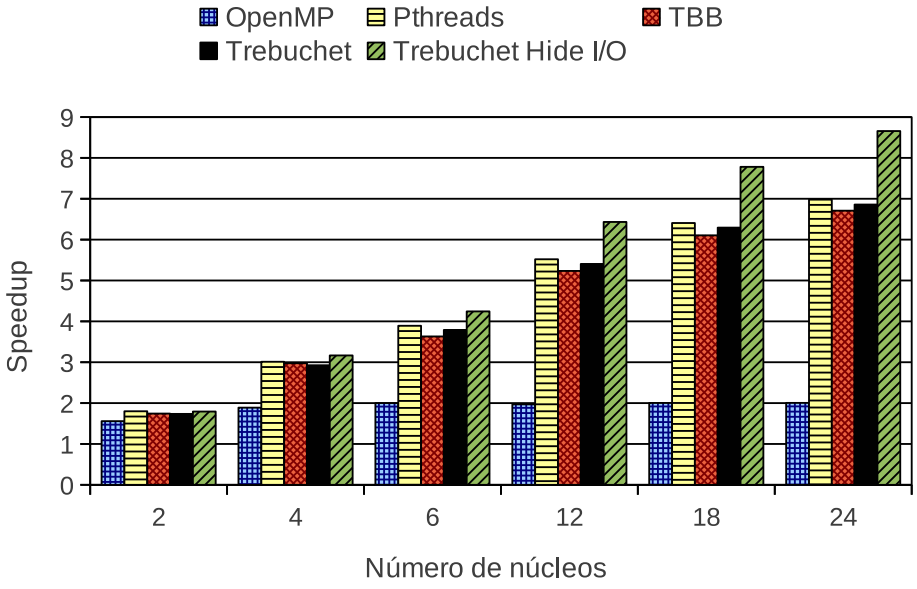

Figura 8.9: Resultados para o Blackscholes.

A Figura 8.9 mostra os resultados obtidos para a aplicação Blackscholes. Com o uso do TALM foi possível obter bom desempenho (comparável a versão *Pthreads*) com uma programação simples. No entanto, a flexibilidade da linguagem permitiu, com facilidade, a aplicação de técnicas mais sofisticadas de programação paralela e a obtenção de melhorias de desempenho.

Os resultados apresentados na Figura 8.10 mostram que na versão compilada e com uso de roubo de tarefas (*Treb Couillard (WS)* no gráfico) é possível obter

acelerações de até 17.05, para até 24 núcleos. De fato, o desempenho obtido foi superior ao apresentado pela versão *TBB* (aceleração de 16.22), e próximo ao atingido pela versão *Pthreads* (aceleração de 18.89). É possível notar que o mecanismo de roubo de tarefas apresentou uma contribuição significativa para o desempenho (a aceleração com 24 núcleos para a versão *Treb Couillard (no WS)* foi de 13.00).

Os resultados para a versão manual mostram que é possível superar o desempenho da versão *Pthreads* (aceleração de 19.47). No entanto a diferença de desempenho existente entre a versão de alto nível e a versão manual pode ser reduzida com melhorias no mecanismo de roubo de tarefas e com a adição de otimizações no código gerado pelo *Couillard*.

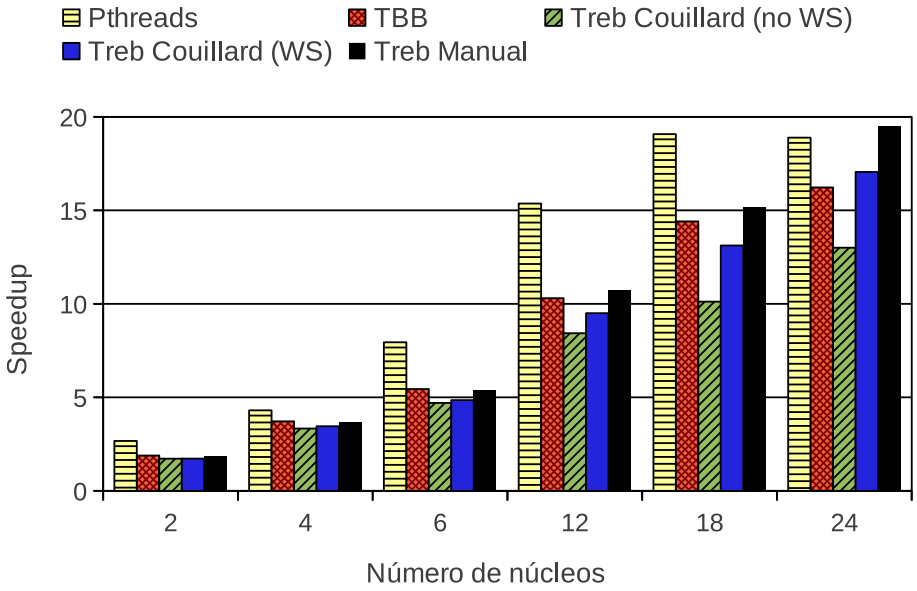

Figura 8.10: Resultados para oFerret.

Capítulo 9

Conclusões e Trabalhos Futuros

Neste trabalho foi apresentado o TALM (*TALM is an Architecture and Language for Multi-threading*), um modelo *dataflow* de execução de programas que permite execução baseada em fluxo de dados com granularidade grossa em máquinas de Von Neumann. Para demonstrar o modelo foi criada uma máquina virtual, a *Trebuchet*. Também foi apresentado o compilador *Couillard*, que facilita a programação no modelo TALM com uma linguagem de alto nível para descrever super-instruções e suas relações.

Para avaliar o modelo e o desempenho da *Trebuchet* com aplicações regulares, foram paralelizadas sete aplicações: uma aplicação de cálculo de determinante de matrizes, uma aplicação de multiplicação de matrizes, uma aplicação de renderização de cenas com *ray-tracing*, o Equake do SpecOMP 2001 [18], o IS do NPB3.0-OMP [19], e também o LU e Mandelbrot do *OpenMP Source code Repository* [20]. Os *speedups* atingidos nessas aplicações para 8 *threads*, em relação à versão sequencial foram, respectivamente, 2,00, 4,16, 4,39, 3,61, 3,00, 2,19 e 7,16 vezes. Também foi feita uma comparação com o *OpemMP*. As versões *OpenMP* das aplicações apresentaram, respectivamente, *speedups* de 1,94, 4,15, 4,39, 3,40, 3,11, 2,19 e 7,13 vezes. Os resultados são bastante promissores e mostram que a *Trebuchet* pode ser bastante competitiva com o *OpenMP*, enquanto provê a flexibilidade do modelo *dataflow* para alplicações mais complexas.

Para avaliar o *Couillard* duas aplicações em estado-da-arte foram paralelizadas usando a linguagem de alto nível do TALM. Os resultados mostram que, em experimentos realizados com uma máquina com 24 núcleos de processamento, foi obtido desempenho competitivo com versões artesanais em *Pthread* e desempenho superior ao das versões em *TBB*. A avaliação mostra que é possível melhorar significativamente o desempenho, simplesmente experimentando com a conectividade e a granularidade das super-instruções, o que apoia a afirmação de que o *Couillard* provê um *framework* flexível e escalável para computação paralela.

Este trabalho abre um leque de oportunidades para pesquisas futuras. A intenção

de que o TALM seja adotado pela comunidade como um modelo de programação paralela sempre foi o objetivo principal desse trabalho. Outros alunos e pesquisadores já estão envolvidos e trabalhando com diferentes aspectos que são importantes para tornar o TALM um modelo completo e robusto, com todo o ferramental necessário. No restante deste capítulo são identificadas oportunidades de pesquisa, muitas delas já em andamento.

Em máquinas *Dataflow*, incluindo o modelo TALM, a alocação de instruções é fortemente relacionada com o desempenho. Instruções alocadas em EPs associados à núcleos de processamento próximos na máquina hospedeira podem minimizar os custos de comunicação (para a troca de operandos). Por outro lado, alocar instruções de forma dispersa pode aumentar o paralelismo. A automatização do processo de alocação de instruções é importante para tornar a Trebuchet mais transparente ao usuário, e é incluída como trabalho futuro. Embora já existam soluções para este problema [66], a flexibilidade da Trebuchet, como uma máquina virtual, permite o estudo de soluções diferentes e mais agressivas. Afinal, a rede de interconexão da Trebuchet e a disposição dos EPs poderia ser dinamicamente alterada. Além disso, é interessante que a disposição dos EPs e a rede de interconexão se assemelhem à organização dos núcleos da máquina hospedeira, usando *schedule affinity* para associar cada *thread* (representado um EP) à um núcleo de processamento.

Um dos grandes problemas de programação paralela é evitar condições de corrida no acesso à recursos compartilhados. No modelo TALM, apresentado neste trabalho, se duas super-instruções paralelas acessam os mesmos recursos, é necessário estabelecer uma precedência com o uso de arestas entre estas super-instruções. No entanto, em muitos casos, conflitos no acesso aos recursos compartilhados não ocorrem com tanta frequência. A serialização imposta pelas arestas passa a ser um entrave para se obter ganhos de desempenho.

Foi desenvolvido então, no escopo do trabalho de mestrado de Tiago A. O. Alves [16], um modelo de execução otimista fora-de-ordem para o TALM. O modelo permite que super-instruções sejam marcadas como especulativas. Super-instruções especulativas são executadas como parte de uma transação. Um transação é formada por uma super-instrução e uma instrução `commit` associada. Transações possuem acesso apenas a cópias locais dos recursos utilizados. Quando sua execução é terminada sem conflitos, alterações locais serão persistidas pela instrução de `commit`. Na ocorrência de conflitos, as alterações locais são descartadas e a super-instrução é re-executada. Este mecanismo permite a execução de super-instruções fora-de-ordem, especulativamente, sem alterar a semântica do programa.

Cada super-instrução especulativa possui um conjunto de leitura e um conjunto de escrita. Cada acesso à memória feito por uma operação de leitura passa por uma verificação para incluir o endereço acessado no conjunto de leitura, caso seja o

primeiro acesso. O conjunto de escrita contém as modificações locais que precisam ser persistidas pela instrução de `commit`. Cada super-instrução, ao terminar sua execução, manda para a instrução de `commit` associada uma *mensagem de commit*, contendo toda a informação necessária, incluindo os conjuntos de leitura e escrita, para persistir as alterações locais.

As instruções de `commit` são conectadas entre si, em um `subgrafo de commits`. Este grafo serve para ordenar as instruções de `commit` para que as operações de persistência sejam feitas sem afetar a semântica do programa. Sendo assim, embora a execução das super-instruções leve em conta as dependências implícitas nos acessos à memória, essa relação é respeitada no *subgrafo de commits*. Além disto, ter as operações de persistência representadas por instruções de `commit` torna o modelo distribuído, pois as instruções *subgrafo de commit* podem ser alocadas aos elementos de processamento da arquitetura da mesma forma que é feito com o grafo do programa.

O modelo de execução especulativa do TALM foi implementado na *Trebuchet*. O uso do modelo implica em, ao descrever programas, usar funções especiais para fazer acessos à memória em regiões que o programador deseja que façam parte do contexto da especulação. Experimentos foram realizados com uma aplicação artificial que simula um servidor de transações bancárias, para simular cenários que variam quanto à carga computacional, o tamanho da transação, profundidade da especulação e contenção. Os resultados mostram que o uso de especulação pode ser eficiente em uma série de cenários. Em uma aplicação real, experimentos mostram que é possível extrair desempenho onde as técnicas de programação tradicionais, no caso o *OpenMP*, não são bem sucedidas. Mais detalhes sobre o assunto podem ser obtidos em [16, 17].

O uso de especulação ainda não é suportado pelo *Couillard*. A implantação dessa funcionalidade envolve a extensão da linguagem para permitir que o programador descreva os objetos de memória farão parte do contexto da especulação em cada super-instrução. Além disso, o compilador dever ser capaz de montar um *subgrafo de commit* de forma eficiente. A inclusão de suporte à especulação no *Couillard* também é objeto de pesquisa em andamento.

Outra questão de extrema importância é a diminuição com os custos de emulação. Além de melhorias na implementação da Trebuchet, co-processadores para executar determinadas instruções podem ser úteis. Esses co-processadores podem ser desenvolvidos em FPGAs conectadas ao sistema. O Apêndice B apresenta a *FlowPGA* que é uma implementação reduzida da arquitetura WaveScalar feita em FPGAs com baixo número de células lógicas. Este projeto foi um estudo de viabilidade para a implementação de um futuro suporte de *hardware* para a Trebuchet. Vale lembrar que este suporte não é mandatório, mas poderia gerar ganhos de de-

sempenho significativos.

Uma alternativa interessante é estender a *Trebuchet* para permitir a execução de super-instruções em GPGPU ou FPGA. Neste caso, o programador escreveria o código das super-instruções para a arquitetura alvo e a *Trebuchet* faria a interface para envio e recebimento de dados, semelhante ao que é feito no HMPP [48]. Essa funcionalidade transformaria o TALM em uma plataforma heterogênea de programação paralela. Neste mesmo veio de pesquisa, também é possível estender a *Trebuchet* para funcionar em *clusters* de computadores. Além disso, não há nenhuma limitação do modelo que impeça o uso de outras bibliotecas de programação paralela pra descrever o comportamento de super-instruções. Desta forma, *OpenMP*, *TBB* ou *Pthreads* poderiam ser usados em conjunto com a *Trebuchet*. Experimentos e estudos com o uso misto destas bibliotecas são objeto de trabalhos futuros.

O mecanismo de roubo de tarefas influenciou significativamente nos resultados obtidos em um dos experimentos. Um estudo mais profundo de politicas de roubo mais eficientes e do impacto dessas políticas no desempenho é objeto de pesquisa em andamento.

Melhorias no *Couillard* e introdução de novas funcionalidades também estão entre os objetivos desse projeto. A criação de *templates* para descrever aplicações que se encaixam bem em padrões de paralelismo já conhecidos pode facilitar ainda mais a tarefa de programar com o TALM.

Este trabalho é baseado na experiência adquirida ao portar aplicações reais para o modelo TALM. Sendo assim, encontrar mais aplicações que sejam candidatas interessantes para paralelização com o *Couillard* é objeto constante de pesquisa neste projeto. As aplicações usadas nos experimentos executados neste trabalho possuem características e padrões de paralelismo que permitem o uso das bibliotecas de paralelização já consagradas. Com exceção do Ferret, todas as aplicações usadas são regulares e podem ser paralelizadas com o *OpenMP*. Nestes casos, as vantagens da execução guiada por fluxo de dados não ficam tão evidentes. Aplicações irregulares tendem a proporcionar um *explosão de paralelismo*, pois o desbalanceamento de carga faz com que partes do programa, em um laço, por exemplo, executem mais rapidamente, abrindo possibilidade de disparar instruções de diversas iterações simultaneamente. Esta explosão, aliada a um mecanismo de escalonamento dinâmico é chave para a obtenção de desempenho com este modelo.

Referências Bibliográficas

[1] OLUKOTUN, K., HAMMOND, L. "The Future of Microprocessors", *Queue*, v. 3, n. 7, pp. 26–29, 2005. ISSN: 1542-7730. doi: http://doi.acm.org/10.1145/1095408.1095418.

[2] OLUKOTUN, K., NAYFEH, B. A., HAMMOND, L., et al. "The Case for a Single-Chip Multiprocessor". In: *IEEE Computer*, pp. 2–11, 1996.

[3] JIAN, L., MARTINEZ, J. F. "Power-Performance Implications of Thread-level Parallelism on Chip Multiprocessors". In: *ISPASS '05: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005*, pp. 124–134, Washington, DC, USA, 2005. IEEE Computer Society. ISBN: 0-7803-8965-4. doi: http://dx.doi.org/10.1109/ISPASS.2005.1430567.

[4] HERLIHY, M., MOSS, J. E. B. "Transactional Memory: Architectural Support for Lock-Free Data Structures." In: *ISCA*, pp. 289–300, 1993.

[5] 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.

[6] THE MPI FORUM. "MPI: A message passing interface". In: *Proc. Supercomputing '93*, pp. 878–883, nov. 15–19, 1993. doi: 10.1109/SUPERC.1993.1263546.

[7] 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.

[8] GURD, J. R., KIRKHAM, C. C., WATSON, I. "The Manchester prototype dataflow computer", *Commun. ACM*, v. 28, n. 1, pp. 34–52, 1985. ISSN: 0001-0782. doi: http://doi.acm.org/10.1145/2465.2468.

[9] ARVIND, K., NIKHIL, R. S. "Executing a Program on the MIT Tagged-Token Dataflow Architecture", *IEEE Trans. Comput.*, v. 39, n. 3, pp. 300–318, 1990. ISSN: 0018-9340. doi: http://dx.doi.org/10.1109/12.48862.

[10] SWANSON, S., MICHELSON, K., SCHWERIN, A., et al. "WaveScalar". In: *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pp. 291–302, 2003. doi: 10.1109/MICRO.2003.1253203.

[11] 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.

[12] MARZULO, L. A., FRANCA, F. M., COSTA, V. S. "Transactional WaveCache: Towards Speculative and Out-of-Order DataFlow Execution of Memory Operations", *Computer Architecture and High Performance Computing, Symposium on*, v. 0, pp. 183–190, 2008. ISSN: 1550-6533. doi: http://doi.ieeecomputersociety.org/10.1109/SBAC-PAD.2008.29.

[13] PEI, S., WU, B., DU, M., et al. "SpMT WaveCache: Exploiting Thread-Level Parallelism in WaveScalar". In: *Proceedings of the 2009 WRI World Congress on Computer Science and Information Engineering - Volume 03*, CSIE '09, pp. 530–535, Washington, DC, USA, 2009. IEEE Computer Society. ISBN: 978-0-7695-3507-4. doi: http://dx.doi.org/10.1109/CSIE.2009.35.

[14] ALVES, T. A., MARZULO, L. A. J., FRANÇA, F. M. G., et al. "Trebuchet: Explorando TLP com Virtualização DataFlow". In: *WSCAD-SSC'09*, pp. 60–67, São Paulo, out. 2009. SBC.

[15] ALVES, T. A., MARZULO, L. A., FRANCA, F. M., et al. "Trebuchet: exploring TLP with dataflow virtualisation", *International Journal of High Performance Systems Architecture*, v. 3, n. 2/3, pp. 137, 2011. ISSN: 1751-6528. doi: 10.1504/IJHPSA.2011.040466.

[16] ALVES, T. A. *Execução Especulativa em uma Máquina Virtual Dataflow*. Tese de Mestrado, COPPE - UFRJ, may 2010.

[17] MARZULO, L. A. J., ALVES, T. A., FRANCA, F. M. G., et al. "TALM: A Hybrid Execution Model with Distributed Speculation Support", *Computer Architecture and High Performance Computing Workshops, International Symposium on*, v. 0, pp. 31–36, 2010. doi: http://doi.ieeecomputersociety.org/10.1109/SBAC-PADW.2010.8.

[18] ASLOT, V., DOMEIKA, M., EIGENMANN, R., et al. "SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance". In: *In Workshop on OpenMP Applications and Tools*, pp. 1–10, 2001.

[19] JIN, H., JIN, H., FRUMKIN, M., et al. *The OpenMP Implementation of NAS Parallel Benchmarks and its Performance*. Relatório técnico, NASA Ames Research Center, 1999.

[20] DORTA, A., RODRIGUEZ, C., DE SANDE, F., et al. "The OpenMP Source Code Repository". In: *13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pp. 244–250, Washington, DC, USA, 2005. IEEE. ISBN: 0-7695-2280-7. doi: 10.1109/EMPDP.2005.41.

[21] BIENIA, C. *Benchmarking Modern Multiprocessors*. Tese de Doutorado, Princeton University, jan 2011.

[22] BUTENHOF, D. R. *Programming with POSIX threads*. Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN: 0-201-63392-2.

[23] REINDERS, J. *Intel threading building blocks : outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007. ISBN: 0596514808.

[24] PATTERSON, D. A., HENNESSY, J. L. *Computer organization and design (3rd ed.): the hardware/software interface*. San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 2003. ISBN: 1-55860-428-6.

[25] PATTERSON, D. A., HENNESSY, J. L. *Computer Architecture (3rd ed.): A Quantitative Approach*. San Francisco, CA, USA, Morgan Kaufmann Publishers Inc.

[26] DENNIS, J. B., MISUNAS, D. P. "A preliminary architecture for a basic data-flow processor", *SIGARCH Comput. Archit. News*, v. 3, n. 4, pp. 126–132, 1974. ISSN: 0163-5964. doi: http://doi.acm.org/10.1145/641675.642111.

[27] DAVIS, A. L. "The architecture and system method of DDM1: A recursively structured Data Driven Machine". In: *ISCA '78: Proceedings of the 5th annual symposium on Computer architecture*, pp. 210–215, New York, NY, USA, 1978. ACM Press. doi: http://doi.acm.org/10.1145/800094.803050.

[28] SAKAI, S., YAMAGUCHI, Y., HIRAKI, K., et al. "An Architecture Of A Dataflow Single Chip Processor". In: *Computer Architecture, 1989. The 16th Annual International Symposium on*, pp. 46–53, 28 May - 1 June, 1989.

[29] SHIMADA, T., HIRAKI, K., NISHIDA, K., et al. “Evaluation of a prototype data flow processor of the SIGMA-1 for scientific computations”. In: *ISCA ’86: Proceedings of the 13th annual international symposium on Computer architecture*, pp. 226–234, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press. ISBN: 0-8186-0719-X. doi: http://doi.acm.org/10.1145/17407.17383.

[30] DENNIS, J. B. *First version dataflow procedure language*. Relatório Técnico MAC TM61, MIT Laboratory for Computer Science, 1991.

[31] KISHI, M., YASUHARA, H., KAWAMURA, Y. “DDDP-a Distributed Data Driven Processor”. In: *ISCA ’83: Proceedings of the 10th annual international symposium on Computer architecture*, pp. 236–242, Los Alamitos, CA, USA, 1983. IEEE Computer Society Press. ISBN: 0-89791-101-6.

[32] GRAFE, V. G., DAVIDSON, G. S., HOCH, J. E., et al. “The Epsilon dataflow processor”. In: *ISCA ’89: Proceedings of the 16th annual international symposium on Computer architecture*, pp. 36–45, New York, NY, USA, 1989. ACM Press. ISBN: 0-89791-319-1. doi: http://doi.acm.org/10.1145/74925.74930.

[33] PAPADOPOULOS, G. M., CULLER, D. E. “Monsoon: an explicit token-store architecture”. In: *ISCA ’90: Proceedings of the 17th annual international symposium on Computer Architecture*, pp. 82–91, New York, NY, USA, 1990. ACM Press. ISBN: 0-89791-366-3. doi: http://doi.acm.org/10.1145/325164.325117.

[34] SWANSON, S., SCHWERIN, A., MERCALDI, M., et al. “The wavescalar architecture”, *In submission to ACM Transactions on Computer Systems, TOCS*. doi: https://www.cs.washington.edu/homes/swanson/support/swanson_WaveScalarArch.pdf.

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

[36] NIKHIL, R. “The parallel programming language Id and its compilation for parallel machines”, *Workshop on Mazzive Parallelis: Hardware, Programming and Applications*, 1990.

[37] ARVIND, NIKHIL, R. “Executing a program on the MIT tagged-token dataflow architecture”, *Computers, IEEE Transactions on*, v. 39, n. 3, pp. 300–318, March 1990. doi: 10.1109/12.48862.

[38] JOHN T. FEO, P. J. M., SKEDZIELEWSKI, S. K. "Sisal90", *P. High Performance Functional Computing*, 1995.

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

[40] ASHCROFT, E. A., WADGE, W. W. "Lucid, a nonprocedural language with iteration", *Commun. ACM*, v. 20, n. 7, pp. 519–526, 1977. ISSN: 0001-0782. doi: http://doi.acm.org/10.1145/359636.359715.

[41] MCGRAW, J. R. "The VAL Language: Description and Analysis", *ACM Trans. Program. Lang. Syst.*, v. 4, n. 1, pp. 44–82, 1982. ISSN: 0164-0925. doi: http://doi.acm.org/10.1145/357153.357157.

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

[43] 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.

[44] 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.

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

[46] TRANCOSO, P., STAVROU, K., EVRIPIDOU, P. "DDMCPP: The Data-Driven Multithreading C Pre-Processor". In: *The 11th Workshop on Interaction between Compilers and Computer Architectures*, p. 32. Citeseer, 2007.

[47] KYRIACOU, C., PARASKEVAS EVRIPODOU, TRANCOSO, P. "Data-Driven Multithreading Using Conventional Microprocessors", *IEEE Transactions on Parallel and Distributed Systems*, v. 17, n. 10, pp. 1176–1188, oct 2006. ISSN: 1045-9219. doi: 10.1109/TPDS.2006.136.

[48] DOLBEAU, R., BIHAN, S., BODIN, F. "HMPP: a hybrid multi-core parallel programming environment". In: *First Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[49] KULKARNI, M., PINGALI, K., WALTER, B., et al. "Optimistic parallelism requires abstractions". In: *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation - PLDI '07*, PLDI '07, p. 211, New York, New York, USA, 2007. ACM Press. ISBN: 9781595936332. doi: 10.1145/1250734.1250759.

[50] KULKARNI, M., PINGALI, K., RAMANARAYANAN, G., et al. "Optimistic parallelism benefits from data partitioning". In: *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pp. 233–243, New York, NY, USA, 2008. ACM. ISBN: 978-1-59593-958-6. doi: http://doi.acm.org/10.1145/1346281.1346311.

[51] KULKARNI, M., CARRIBAULT, P., PINGALI, K., et al. "Scheduling strategies for optimistic parallel execution of irregular programs". In: *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures - SPAA '08*, SPAA '08, p. 217, New York, New York, USA, 2008. ACM Press. ISBN: 9781595939739. doi: 10.1145/1378533.1378575.

[52] MCDONALD, A., CHUNG, J., CARLSTROM, B. D., et al. "Architectural Semantics for Practical Transactional Memory". In: *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pp. 53–65, Washington, DC, USA, 2006. IEEE Computer Society. ISBN: 0-7695-2608-X. doi: http://dx.doi.org/10.1109/ISCA.2006.9.

[53] MOSS, J. E. B. "Open Nested Transactions: Semantics and Support". In: *WMPI*, Austin, TX, February 2006.

[54] CHUNG, J., MINH, C. C., MCDONALD, A., et al. "Tradeoffs in transactional memory virtualization". In: *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pp. 371–381, New York, NY, USA, 2006. ACM. ISBN: 1-59593-451-0. doi: http://doi.acm.org/10.1145/1168857.1168903.

[55] ANANIAN, C. S., ASANOVIC, K., KUSZMAUL, B. C., et al. "Unbounded Transactional Memory", *IEEE Micro*, v. 26, n. 1, pp. 59–69, 2006. ISSN: 0272-1732. doi: http://dx.doi.org/2006-02-1702:00:03.800.

[56] SHAVIT, N., TOUITOU, D. "Software transactional memory". In: *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pp. 204–213, New York, NY, USA, 1995. ACM. ISBN: 0-89791-710-3. doi: http://doi.acm.org/10.1145/224964.224987.

[57] HERLIHY, M., LUCHANGCO, V., MOIR, M., et al. “Software Transactional Memory for Dynamic-Sized Data Structures”, pp. 92–101, Jul 2003.

[58] ADL-TABATABAI, A.-R., LEWIS, B. T., MENON, V., et al. “Compiler and Runtime Support for Efficient Software Transactional Memory”. In: *Proceedings of the 2006 Conference on Programming language design and implementation*, pp. 26–37, Jun 2006.

[59] KUMAR, S., CHU, M., J. HUGHES, C., et al. “Hybrid Transactional Memory”. In: *Proceedings of Symposium on Principles and Practice of Parallel Programming*, Mar 2006.

[60] DAMRON, P., FEDOROVA, A., LEV, Y., et al. “Hybrid transactional memory”. In: *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pp. 336–346, New York, NY, USA, 2006. ACM. ISBN: 1-59593-451-0. doi: http://doi.acm.org/10.1145/1168857.1168900.

[61] ARORA, N. S., BLUMOFE, R. D., PLAXTON, C. G. “Thread Scheduling for Multiprogrammed Multiprocessors”, *Theory of Computing Systems*, v. 34, n. 2, pp. 115–144, jan. 2001. ISSN: 1432-4350. doi: 10.1007/s002240011004.

[62] HUANG, S. T. “Termination detection by using distributed snapshots”, *Inf. Process. Lett.*, v. 32, pp. 113–120, August 1989. ISSN: 0020-0190.

[63] BEAZLEY, D. “PLY - Python Lex-Yacc. `http://www.dabeaz.com/ply/`”. .

[64] “Graphviz web-site. `http://www.graphviz.org`”. .

[65] NAVARRO, A., ASENJO, R., TABIK, S., et al. “Load balancing using work-stealing for pipeline parallelism in emerging applications”. In: *Proceedings of the 23rd international conference on Supercomputing*, ICS ’09, pp. 517–518, New York, NY, USA, 2009. ACM. ISBN: 978-1-60558-498-0. doi: http://doi.acm.org/10.1145/1542275.1542358.

[66] MERCALDI, M., SWANSON, S., PETERSEN, A., et al. “Modeling instruction placement on a spatial architecture”. In: *SPAA ’06: Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, pp. 158–169, New York, NY, USA, 2006. ACM. ISBN: 1-59593-452-9. doi: http://doi.acm.org/10.1145/1148109.1148137.

[67] MARZULO, L. A. J., FLESCH, F., NERY, A., et al. "FlowPGA: DataFlow de Aplicações em FPGA", *IX Simpósio em Sistemas Computacionais WSCAD - SSC*, 2008.

[68] COMPUTERS, B. A. *Quarterly technical report no. 3 April l & July 15. 1984 development of a Butterfly multiprocessor test bed: The Butterfly switch.* Relatório técnico, BBN Advanced Computers, Cambridge, Massachusetts, 1985.

Apêndice A

Lista de Trabalhos Publicados

Os seguintes trabalhos foram publicados durante a realização deste projeto:

- MARZULO, L. A., FRANÇA, F. M., COSTA, V. S. "Transactional WaveCache: Towards Speculative and Out-of-Order DataFlow Execution of Memory Operations", Symposium on Computer Architecture and High Performance Computing - SBAC-PAD - v. 0, pp. 183–190, 2008.

- MARZULO, L. A. J., FLESCH, F., NERY, A., FRANÇA, F. M. G, TAVARES, E. S. "FlowPGA: DataFlow de Aplicações em FPGA", IX Simpósio em Sistemas Computacionais WSSCAD - SSC, 2008.

- PEI, S., WU, B., DU, M., CHEN, G., MARZULO, L. A. J., FRANÇA, F. M. G. "SpMT WaveCache: Exploiting Thread-Level Parallelism in WaveScalar". In: Proceedings of the 2009 WRI World Congress on Computer Science and Information Engineering - Volume 03, CSIE '09, pp. 530–535, Los Angeles, USA, 2009. IEEE Computer Society.

- ALVES, T. A., MARZULO, L. A. J., FRANÇA, F. M. G., COSTA, V. S. "Trebuchet: Explorando TLP com Virtualização DataFlow", WSCAD-SSC'09, pp. 60–67, São Paulo, out. 2009. SBC.

- MARZULO, L. A. J., ALVES, T. A., FRANÇA, F. M. G., COSTA, V. S.. "TALM: A Hybrid Execution Model with Distributed Speculation Support", 1st Workshop on Architecture and Multi-Core Applications (on SBAC-PAD) on, v. 0, pp. 31–36, 2010. (*Best Paper Award*)

- ALVES, T. A., MARZULO, L. A., FRANÇA, F. M., COSTA, V. S.. "Trebuchet: exploring TLP with dataflow virtualisation", International Journal of High Performance Systems Architecture, v. 3, n. 2/3, pp. 137, 2011.

Apêndice B

FlowPGA: Dataflow de Aplicações em FPGA

A *FlowPGA* [67] é uma versão reduzida da arquitetura WaveScalar para ser utilizada com *FPGAs* com pequeno número de células lógicas. Uma *FPGA* com 1,5 milhões de gates foi utilizada para implementação. A corretude da implementação foi avaliada com a execução de um programa de multiplicação entre dois números positivos usando sucessivas somas. Os resultados mostram que a arquitetura *FlowPGA* tem desempenho equivalente ao *WaveScalar*. A experiência obtida com o desenvolvimento da *FlowPGA* pode auxiliar na construção de um co-processador *Dataflow* para executar blocos de código compilados para a Trebuchet. O texto que se segue foi extraído de parte do artigo apresentado no WSCAD 2008.

B.1 Conjunto de Instruções e Arquitetura

A arquitetura *FlowPGA* é baseada, com algumas simplificações, na arquitetura *WaveScalar*, permitindo o mapeamento em FPGAs com baixo número de células reconfiguráveis e viabilizando o estudo de arquiteturas *dataflow*, em um ambiente real de execução. Nesta versão ainda não está contemplada a interface de acesso à memória (*Wave-ordered memory*) e um mecanismo de *placement* dinâmico que permita carregar programas com mais instruções estáticas do que o suportado pelas listas de instruções dos *EPs*. Esse mecanismo faria o *swap* de instruções da lista, conforme a evolução do programa em execução. O *swapping* de operandos da *Matching Table* e listas de espera também não é contemplado nesta versão. Sendo assim, o sistema não está preparado para execução de programas com alto grau de paralelismo, que esgotariam o espaço de armazenamento de operandos.

As instruções da *FlowPGA* possuem um tamanho fixo e regular, para facilitar a decodificação. Cada instrução contém um número identificador (ID), o *opcode* que

indica a operação a ser realizada na ALU, um imediato, além dos IDs das instruções de destino. Nesta versão do *FlowPGA* cada instrução pode receber 1 ou 2 operandos e enviar resultados para até 2 instruções. Caso seja necessário enviar o resultado para mais de dois destinos, é usada a instrução `FANOUT` que recebe um operando de entrada e encaminha para dois destinos.

Dependendo da instrução, a posição dos operandos de entrada é relevante para a produção do resultado (em uma divisão, por exemplo, é necessário saber quem são o divisor e o dividendo). Sendo assim, para cada um dos 2 destinos de uma instrução é informada também a posição. A Figura B.1 exibe o formato de uma instrução da *FlowPGA*, com todos os seus campos, e as Tabelas B.1 e B.2 relacionam os valores para os campos posição e opcode, com seus respectivos significados.

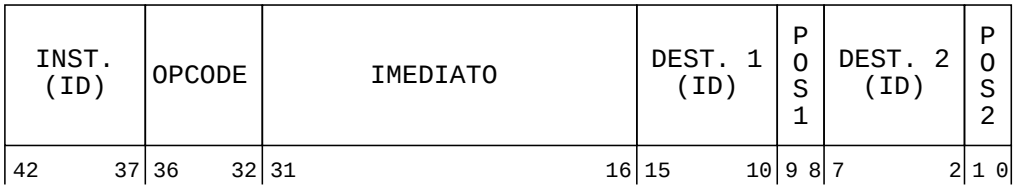

Figura B.1: Formato das instruções

Tabela B.1: Significado dos campos de posição

0	Campo destino correspondente não é utilizado
1	Operando direito
2	Operando esquerdo
3	Único operando

Tabela B.2: Significado do campo opcode

0	Soma	13	FanOut
1	Subtração	14	Saída de Resultado (USB)
2	Multiplicação	16	Soma com imediato
3	Divisão	17	Subtração com imediato
4	Mod	18	Multiplicação com imediato
5	E lógico	19	Divisão com imediato
6	Ou lógico	20	Resto com imediato
7	Negação lógica	21	E lógico com imediato
8	Comparação >	22	Ou lógico com imediato
9	Comparação <	24	Comparação > Imediato
10	Comparação =	25	Comparação < Imediato
11	Steer	26	Comparação = Imediato
12	Wave-Avance		

A execução de uma instrução gera operandos de saída que podem ser enviados para outras instruções, mapeadas em qualquer *EP*. Os operandos são enviados em forma de mensagens contendo o número da onda, ID da instrução de destino, posição

do operando na instrução destino e valor do operando. O formato da mensagem é exibido na Figura B.2.

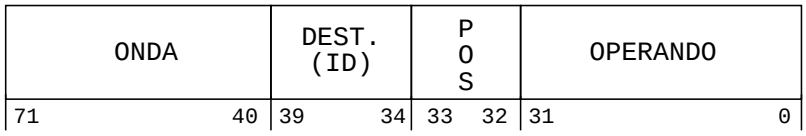

Figura B.2: Mensagens entre instruções

Cada Elemento de Processamento da *FlowPGA* foi implementado como um *pipeline* assíncrono de 3 estágios. A Figura B.3 exibe o *EP*, cujos estágios são descritos a seguir:

Entrada e Casamento: Chegada de no máximo 1 operando por ciclo no *EP*, enviados por outro *EP* ou pelo próprio através da rede de *bypass*. A mensagem de entrada é: *(i)* enviada para execução, se o campo *Posição* da mensagem possui o valor 3; *(ii)* enviada para execução juntamente com outro operando da *Matching Table*, se ocorrer um casamento; *(iii)* enviada para uma fila de espera, caso a *Matching Table* esteja cheia; *(iv)* inserida na *Matching Table*, caso o campo posição seja 1 ou 2, e não tenha ocorrido casamento.

Execução: Acessa a instrução indicada na *Instruction List*, realiza a execução e envia os resultados para o estágio de saída.

Saída: Os resultados da instrução são inseridos em um *buffer de saída* e enviados pelo barramento de saída para o próprio *EP* (pela rede de *bypass*) ou outro remoto (por um *switch Butterfly* [68]). Uma mensagem é produzida para cada destino. O *EP* pode seguir com a execução de instruções, pois cada mensagem só será apagada do *buffer* quando tiver sido recebida pelo switch, que garantirá a sua entrega.

Percebe-se que não há necessidade da *scheduling queue*, pois não há execução especulativa e, além disso, como só uma mensagem é recebida por ciclo, todo o casamento gera uma execução imediata. Pelo mesmo motivo, só é necessário armazenar, no máximo, um operando por instrução na *Matching Table*, ficando a mesma reduzida a linhas contendo apenas o número da onda, número da instrução, operando e posição. Como a *Matching Table* é percorrida em paralelo no estágio de casamento, ela deve ter um número pequeno de linhas para que esta tarefa não seja cara. Daí a existência da lista de espera. Também não é necessário o estágio de Despacho, pois não há *scheduling queue* e a rede de *bypass* envia operandos apenas para o estágio de Entrada/Casamento.

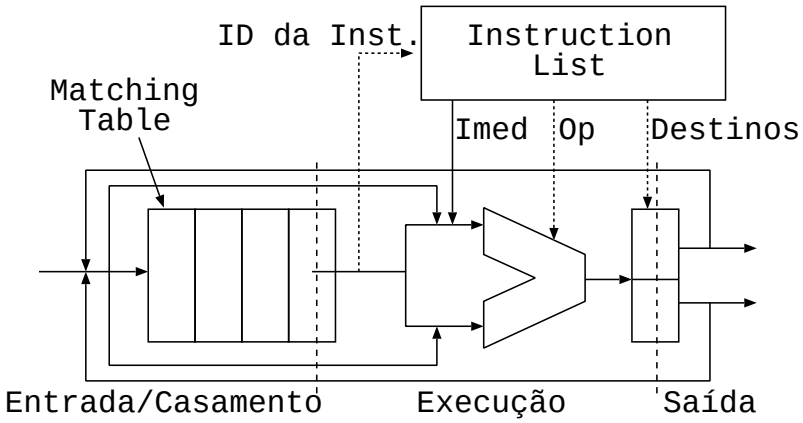

Figura B.3: Um EP na FlowPGA

A comunicação entre os *EPs* é feita por intermédio de um *Switch Butterfly*, descrito na Figura B.4. Este é implementado com um *pipeline* síncrono de 3 estágios, sendo cada um responsável pela execução de um *hop* até a entrega da mensagem ao *Destino*. É importante salientar que no *WaveScalar*, a comunicação entre *EPs* de um mesmo *Domain* é feita ponto-a-ponto. Embora isto possibilite um maior paralelismo, na presente arquitetura, optou-se pela alternativa de alterar o *switch* para simplificar o projeto e facilitar a escalabilidade do mesmo (inserir novos *EPs*). O *Switch Butterfly* 4x4 é construído recursivamente a partir de *switches* 2x2. Para ampliá-lo, basta seguir o mesmo princípio. Para saber qual processador deve receber uma determinada mensagem, o *switch* consulta uma tabela de roteamento que guarda uma lista dos *EPs*, associados a todas as instruções mapeadas.

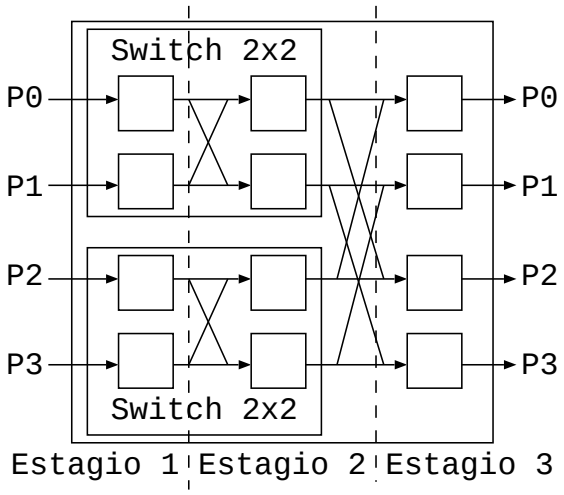

Figura B.4: Switch Butterfly 4x4

B.2 Implementação em FPGA

A arquitetura *FlowPGA*, descrita neste capítulo, foi implementada utilizando a linguagem Handel-C, tendo como alvo a placa RC10, ambas desenvolvidas pela Celoxica®. A RC10 utiliza uma FPGA Xilinx®Spartan-3E, que pode operar com um clock entre 2 e 300MHz. Ela possui 1,5 milhões de células lógicas e disponibiliza uma interface de comunicação via USB, além de uma série de dispositivos de E/S (VGA, teclado, mouse, *display* de sete segmentos, leds, entre outros). A implementação desta arquitetura se deu por meio do kit de desenvolvimento DK5, também licenciado pela Celoxica®, em conjunto com as ferramentas do ISE WebPack da Xilinx®.

A comunicação entre *EPs* e *Switch* é feita através de canais com *FIFOs* para armazenar até sete mensagens em cada. As estruturas *FIFO* atuam como os *reject buffers* do *WaveScalar*. Para a comunicação entre os estágio do pipeline assíncrono de cada *EP* foram utilizados canais sem *FIFO*. Uma interface de recebimento de mensagens externas através da USB foi elaborada para fazer a carga de operandos de inicialização da computação. A instrução `OUT`, descrita na Seção B.1 gera uma mensagem de saída contendo o operando, que é encaminhada para a interface USB e pode ser coletada pelo PC hospedeiro. A Figura B.5 mostra uma visão geral da arquitetura *FlowPGA* (envolvida pela linha pontilhada) e sua comunicação com o PC.

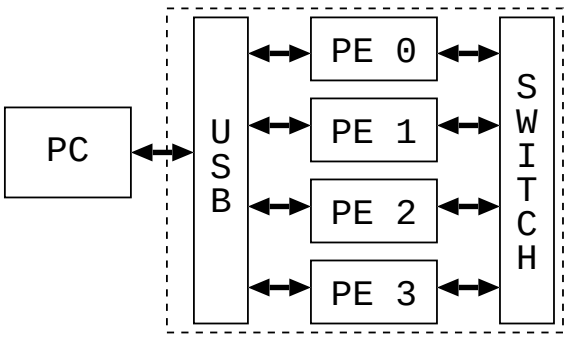

Figura B.5: Visão geral da FlowPGA

A implementação de *ALUs* com capacidade para executar todos os tipos de instrução descritos na Tabela B.2 só seria possível, para a placa RC10, em alternativas de *design* menos eficientes. Neste trabalho, houve uma preocupação maior em criar uma versão da *FlowPGA* com desempenho mais próximo possível da arquitetura *WaveScalar*. Desta forma, o projeto foi otimizado ao máximo, dentro dos limites da placa. As operações de divisão e multiplicação são as que ocupam a maior área no projeto e também as que demandam o ciclo mais longo para execução. Estas operações não foram incluídas nesta implementação, resultando em um projeto que

ocupa aproximadamente quinhentos e cinqüenta mil células lógicas e pode executar em freqüências de até 48MHz. Vale lembrar que esta é uma das possíveis alternativas de *design*. A descrição da arquitetura em uma linguagem de alto nível, como o Handel-C, facilita a adequação do projeto para outras necessidades.

B.3 Descrição do grafo *dataflow*

Para facilitar a escrita de programas para a *FlowPGA*, foi desenvolvida uma linguagem para descrever o grafo *dataflow* e definir o *placement* das instruções nos *EPs*. Nesta, informa-se a quantidade de *EPs*, as instruções a serem executadas por cada *EP* e os dados de entrada do programa. A Figura B.7 mostra o programa *FlowPGA* associado ao grafo dataflow da Figura B.6. A aplicação faz uma operação de multiplicação através de sucessivas somas.

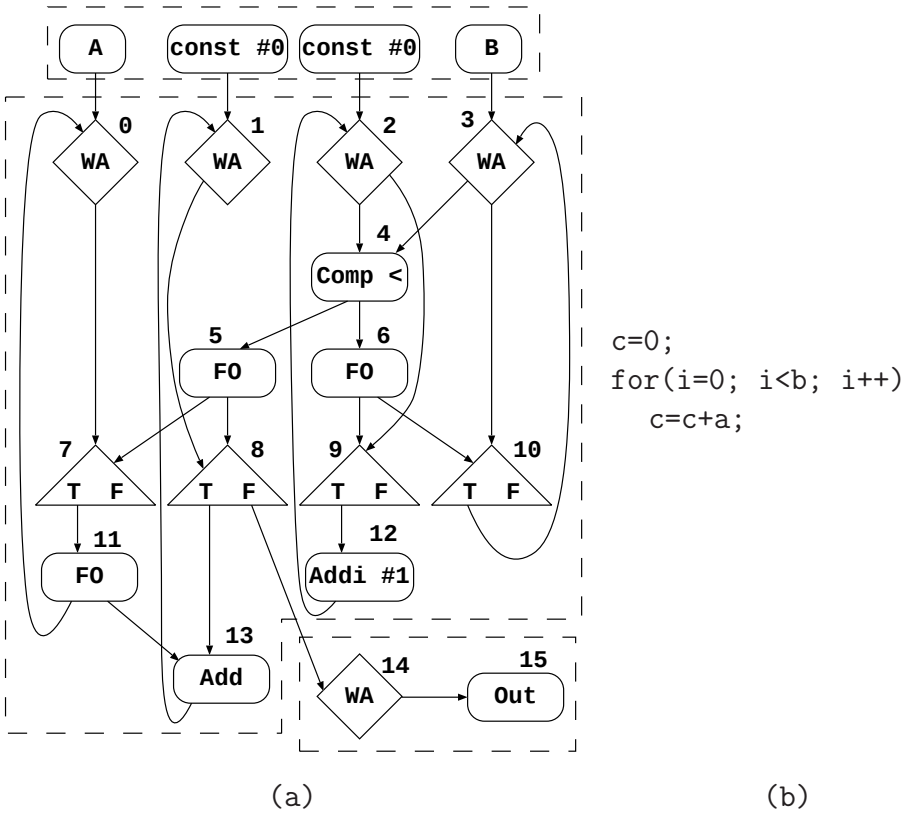

Figura B.6: Grafo de uma aplicação para o FlowPGA.

Primeiramente, a tabela de roteamento associa cada instrução estática a um *EP*. Um vetor armazena uma lista de números de *EPs* indexada pelo identificador da instrução. Dentro de cada bloco *pe(id)*, uma instrução corresponde a uma linha que obedece ao formato:

$$opcode\#im(id) \rightarrow (id) < side >, (id) < side >;$$

Neste formato, o resultado da operação indicada à esquerda da seta será enviado para as instruções de destino indicadas à direita da mesma. Além disso, este resultado poderá ser inserido no operando direito ou esquerdo destas instruções, conforme indicado pelo campo $< side >$. Caso nada seja informado a respeito da posição de inserção, o tradutor assumirá que a instrução de destino admite apenas um operando. Algumas instruções também permitem operandos imediatos, indicados no campo $\#im$.

O último bloco, denominado *init*, inicializa a computação enviando mensagens para cada um dos *EPs* descritos nos blocos anteriores. As mensagens podem conter constantes de inicialização ou variáveis, cujos valores devem ser atribuídos antes do envio.

```
pe=4;

pe(0){
    wa(0) -> (7)<l>;
    steer(7) -> (11);
    fo(11) -> (0), (13)<l>;
}
pe(1){
    wa(1) -> (8)<l>;
    fo(5) -> (7)<r>, (8)<r>;
    steer(8) -> (13)<r>, (14);
    add(13) -> (1);
    wa(14) -> (15);
    out(15);
}
pe(2){
    wa(2) -> (4)<l>, (9)<l>;
    compless(4) -> (5), (6);
    fo(6) -> (9)<r>, (10)<r>;
    steer(9) -> (12);
    addi#1(12) -> (2);
}
pe(3){
    wa(3) -> (4)<r>, (10)<l>;
    steer(10) -> (3);
}

init{
    a = 3;
    b = 2;
    a->wa(0);
    0->wa(1);
    0->wa(2);
    b->wa(3);
}
```

Figura B.7: Um programa FlowPGA

B.4 Ferramentas

Uma ferramenta de tradução foi criada para a especificação de um programa *FlowPGA*. Ela tem como entrada um grafo dataflow, descrito de acordo com a linguagem especificada na Seção B.3, e como saída o código binário correspondente. Um visualizador de mensagens também foi criado para separar em campos e exibir, de forma inteligível, as mensagens enviadas pela *FlowPGA* e coletadas pelo PC (via USB).

B.5 Experimentos e Resultados

Esta seção provê uma avaliação da arquitetura *FlowPGA* e sua semelhança comportamental em relação ao *WaveScalar*. A linguagem Handel-C permite implementar rapidamente uma solução em FPGA, mas, como a descrição é feita em alto nível, o produto final pode não ser o mais otimizado. Uma solução em Handel-C é mais genérica que uma solução em uma linguagem de descrição de hardware (como VHDL ou Verilog). Além disso, no Handel-C, cada atribuição a uma variável leva um ciclo de *clock*. Sendo assim, cada estágio do *pipeline* de um *EP* pode levar mais de um ciclo, embora o trabalho realizado em cada ciclo seja inferior ao de uma implementação descrita em HDL.

Com o objetivo de avaliar o grau de equivalência entre as duas implementações e validar a solução construída, o algoritmo da Figura B.6 (que realiza uma multiplicação usando sucessivas somas) foi descrito como um grafo dataflow (de acordo com o modelo exposto neste capítulo). O tradutor, descrito na Seção B.4, foi usado na elaboração do binário para a arquitetura *FlowPGA*. O programa foi mapeado manualmente nas listas de instruções dos processadores e, em seguida, o projeto foi compilado e gravado na FPGA. O algoritmo foi executado na *FlowPGA* e no simulador *WaveScalar*, variando o multiplicador (valor de B) entre cinqüenta e cem mil. O simulador foi configurado com as mesmas características da implementação da *FlowPGA* (1 *Cluster* com 1 *Domain* de 4 *EPs*, com até 16 instruções cada).

A Figura B.8 mostra os resultados deste experimento, observando-se que as implementações são equivalentes. Cada estágio do *pipe* assíncrono da *FlowPGA* leva entre 2 e 4 ciclos. Sendo assim, uma distância quase constante, de aproximadamente quatro vezes, é verificada no número de ciclos da execução entre as arquiteturas. É importante lembrar que em uma solução descrita em HDL cada estágio pode ser projetado para durar apenas um ciclo.

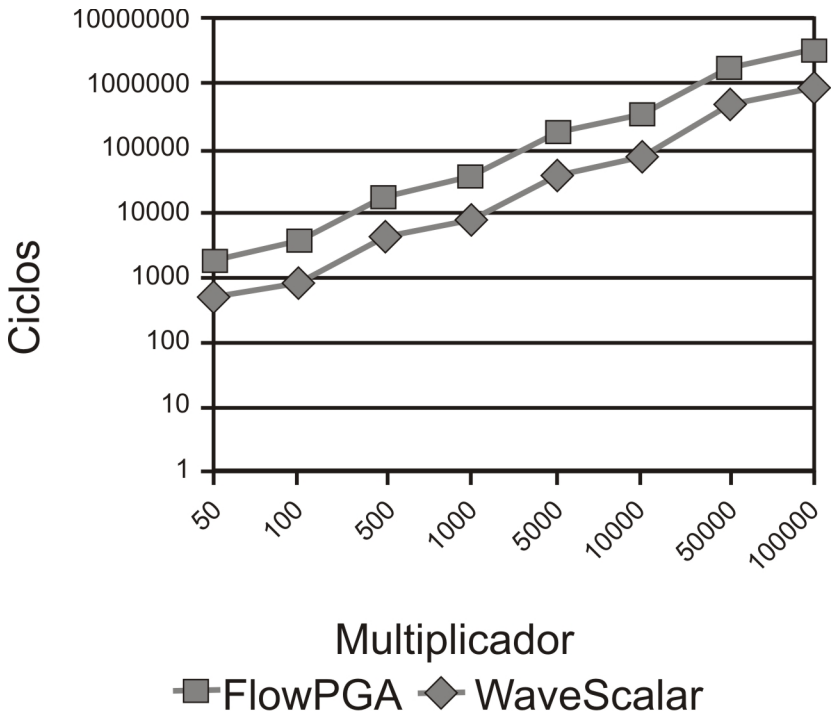

Figura B.8: FlowPGA vs. WaveScalar

Apêndice C

A *Transactional* WaveCache

Por padrão, o WaveScalar segue um estrito mecanismo de ordenação de acessos à memória. *Stores* parciais podem adicionar paralelismo à execução das operações de memória dentro de uma onda, mas a execução paralela entre ondas só é possível se as mesmas não fizerem acessos à memória. A **Transactional WaveCache** [11, 12] é um mecanismo de ordenação de memória alternativo que mantém ordem de execução das operações de memória dentro de uma onda, mas adiciona a habilidade de executar, especulativamente e fora-de-ordem, operações de memória de ondas distintas. Este mecanismo de ordenação é inspirado e conceitos de Memórias Transacionais. Ondas são tratadas como regiões atômicas e executadas como **transações aninhadas**. Se uma onda finalizou a execução de suas operações de memória ela pode fazer o *commit* assim que as ondas anteriores tiverem feito *commit*. Se um perigo é detectado em uma onda especulativa, todas as ondas seguintes (filhas) são abortadas e re-executadas.

Este trabalho foi iniciado pelo presente autor, durante o mestrado, e teve como motivação o fato de que mais paralelismo se torna disponível uma vez que se permite que operações de memória disjuntas executem em paralelo. No entanto, este trabalho é reapresentado neste texto, de forma resumida, pois avanços foram atingidos desde sua apresentação. Tais avanços permitiram a execução de porções de alguns *benchmarks* clássicos. A experiência obtida com a construção da Transactional WaveCache inspirou a construção mecanismo de execução especulativa de acessos à memória para o modelo TALM e sua implementação na Trebuchet [16].

C.1 Transações no WaveScalar

A grande maioria dos trabalhos em Memórias Transacionais sugere um número de alternativas para implementar uma solução. Em seguida, é apresentado o algoritmo que descreve o funcionamento da Transactional WaveCache. A primeira ideia é que cada onda possui um bit F associado, indicando se ela já terminou ou não

suas operações de memória. Esse bit é armazenado no **Mapa de Ondas** (veja a página 21). Além disso, cada *StoreBuffer* possui um atributo extra chamado *lastCommittedWave* que armazena o número da última onda que fez o *commit*. O bit F e o atributo *lastCommittedWave* indicam os estados transacionais, a seguir:

1. A onda X é não especulativa se todas as ondas anteriores receberam o *commit*, isto é, $(X = lastCommittedWave + 1)$. O contexto transacional não precisa ser armazenado para ondas não especulativas;
2. Inicialmente $(SB.lastCommittedWave = -1)$, pois a primeira onda $(Wave.Id = 0)$ é sempre não especulativa;
3. Inicialmente, para todas as ondas W, $W.F = FALSE$;
4. Quando uma onda W termina a execução de todas as suas operações de memória, $W.F = TRUE$;
5. Uma onda W está pendente se

$$(W.Id > lastCommittedWave + 1) \wedge W.F$$

6. Uma onda W pode fazer *commit* quando

$$(W.Id = lastCommittedWave + 1) \wedge W.F$$

Em contraste com mecanismos de Memória Transacional padrão, o tamanho da transação não é um problema: transações que necessitam de mais recursos apenas ficarão paralisadas até que recursos se tornem disponíveis ou até que elas se tornem não especulativas.

C.2 O Contexto Transacional

Em Memórias Transacionais para máquinas de Von Neumann, antes que uma transação seja iniciada é necessário salvar o conteúdo de todos os registradores usados pelo programa. Um histórico de todas as alterações na memória também deve ser mantido. Se um perigo é detectado, esta informação será usada para restaurar a memória e o banco de registradores. No WaveScalar não existe banco de registradores, mas todos os operandos usados por uma transação, que foram produzidos externamente, precisam ser re-enviados, no caso de uma re-execução. Esses operandos formam o conjunto de leitura de uma transação ou onda.

Foi observado que as instruções de `Wave-Advance` controlam como os operandos atingem uma onda. Para manter o conjunto de leitura de cada onda, um passo

natural é modificar estas instruções para que elas enviem uma cópia de seus operandos para o *StoreBuffer*, que por sua vez irá inseri-los em uma nova estrutura, chamada *Wave-Context-Table* (WCT). Um *StoreBuffer* pode ter diversos WCTs: cada um deles armazenando o conjunto de leitura de uma onda especulativa, cada linha contendo um operando de uma instrução `Wave-Advance`. Se o WCT ficar cheio, os operandos de `Wave-Advance` permanecerão nos *buffers* de saída de cada EP, bloqueando o início da onda seguinte.

Além disto, é necessário manter um histórico de memória: isto é implementado com uma tabela chamada *MemOp-History (MOH)*. Para cada operação, a MOH armazena o número da onda, o número *Corrente* da anotação na operação de memória, o tipo de operação, um *backup* de dados para operações de `Store`, além do valor (a ser gravado no caso de um *Store*; ou carregado para operações de *Load*). Cada *StoreBuffer* possui um MOH. Note que um problema é que ondas pertencentes à SBs diferentes podem se relacionar. Todos os SBs deveriam ter uma visão única do MOH para que perigos entre estas ondas pudessem ser detectados. Isto é evitado garantindo que um onda especulativa use o mesmo SB que sua onda não especulativa mais próxima. Esta solução pode limitar o paralelismo, mas, de fato, ondas em uma mesma *thread* geralmente ficam sob a responsabilidade do mesmo SB [42].

C.3 Especulação e Reexecução

Foram usadas as estruturas de dados descritas anteriormente para que o mecanismo original de ordenação de operações de memória no WaveScalar permitisse a execução concorrente de requisições de acesso à memória de ondas diferentes. Para estudar de que forma a especulação pode afetar o desempenho, um limite pode ser imposto no número de ondas especulativas que podem ser executadas ao mesmo tempo. Este limite é chamado de **Janela de Especulação**.

Antes de prosseguir, foi observado o surgimento de problemas quando uma instrução recebe operandos de execuções antigas. No pior caso, operandos antigos podem casar com os novos na *Matching Table (MT)*, sendo consumidos e produzindo resultados incorretos. Para solucionar este problema, foi adicionada o número da execução ($ExeN$) ao rótulo dos operandos. Além disso, a regra de disparo foi alterada para usar o $ExeN$ para casar os operandos. Uma instrução que executa com operandos com rótulo $ExeN = K$ produz operandos também com $ExeN = K$.

O *StoreBuffer* mantém um atributo $lastExeN$ que armazena o número da última execução. Cada onda também possui um atributo $CurrentExeN$: uma requisição de acesso à memória só pode ser aceita por uma onda W se $ExeN \leq W.CurrentExeN$.

Agora é possível descrever o algoritmo de *rollback*. Se um perigo é encontrado em uma onda W os WCTs para todas as ondas Y tais que $Y.Id > W.Id$; todos os

operandos no WCT da onda W são re-enviados, causando a re-execução de todas as ondas seguintes. Note que no momento da re-execução, alguns operandos pertencentes ao conjunto de leitura de W podem não estar presentes no WCT ainda. Isto não é um problema: eles simplesmente ainda não atingiram a instrução de `Wave-Advance`. Quando isto ocorrer, **(i)** os operandos serão inseridos na WCT, **(ii)** terão seus $ExeN$ atualizados e **(iii)** serão enviados para a nova instância da onda W.

Quando os operandos são re-enviados pelos*StoreBuffers* seus $ExeN$ são modificados para um identificador único da nova execução. Mais precisamente, quando uma onda W causa a re-execução, todas as ondas $\geq W$ terão seus $currentExeN$ alterados para $StoreBuffer.lastExeN$. Desta forma, requisições de memória relativas à execuções antigas não serão aceitas pelo sistema de memória.

C.4 Detecção de Perigos

Um perigo pode existir entre duas operações de memória se elas acessam o mesmo endereço de memória. Como a Transactional WaveCache mantém a ordem entre as operações de memória dentro de uma onda, operações em uma mesma ondas nunca causarão um perigo. Quando uma operação de memória pertencente à onda X é executada, um perigo entre a onda X e alguma outra onda Y apenas se $X.Id > Y.id$.

Na arquitetura proposta, a estrutura MOH é a chave para o reconhecimento de perigos. Considere duas operações de memória A e B, e suas respectivas ondas X e Y, tal que $X.Id > Y.Id$. Assuma que não existam *Stores* especulativos entre elas e que A já foi executado. Quando B finalmente executar, os possíveis perigos e suas respectivas soluções são os seguintes:

RAW (Read After Write): Ocorre quando A é um `Load` e B é um `Store`. Todas as ondas $\geq X$ devem ser abortadas e re-executadas. Se B é especulativa, ela é inserida na MOH e o seu campo *backup* é **(i)** copiado do campo valor de A na MOH ou **(ii)** obtida de um `Load`.

WAW (Write After Write): Quando ambas A e B são `Stores`, a onda X não precisa ser re-executada. Se B é especulativa, ela é inserida na MOH e o seu campo *backup* recebe o *backup* de A. De qualquer forma, o `Store` não precisa ir à memória, e o *backup* de A na MOH recebe o conteúdo do campo valor do `Store` B.

WAR (Write After Read): Quando A é um `Store` e B é um `Load`, não é preciso re-executar X, pois o campo *backup* de A pode ser usado como valor de retorno para o `Load`. Se B é especulativa, ela deve ser inserida na MOH.

C.5 Commit e RollBack

O *commit* é feito sempre que uma onda não especulativa termina a sua execução no sistema de memória. O commit de uma onda X torna a onda $X+1$ não especulativa. Sendo assim, o contexto transacional de $X+1$ pode ser apagado. Se $X+1$ completou a sua execução (isto é, $F =$ `TRUE`), ela fará o *commit* e será possível prosseguir para a onda seguinte. Quando uma onda faz o *commit*, a onda seguinte pode estar sob custódia de um *StoreBuffer* diferente. Neste caso, uma mensagem é enviada àquele *SB* para que ele saiba que a onda não especulativa está agora sob sua custódia e que ele pode começar a executar operações de memória. O atributo $lastExeN$ também é enviado e atualizado no *SB* de destino para ser usado no caso de futuras re-execuções.

O *commit* requer a limpeza da MOH. Uma possibilidade seria varrer completamente a estrutura. A solução proposta, no entanto, utiliza um *Search Catalog* para acelerar esta operação. Para cada onda, o *Search Catalogue* aponta para a lista que inclui todas as operações da mesma.

A seguir são apresentados os passos necessários para re-executar um programa, se a onda X for abortada:

1. Apagar as linhas L no *Search Catalog* onde $L.Wave > X$;
2. Restaurar a memória para o estado anterior à execução das ondas $\geq X$ (usando o campo *backup* dos `Stores` na MOH). Durante este processo, estes valores também devem ser apagados na MOH;
3. Limpar os WCTs I onde $I.Wave > X$;
4. Limpar as requisições para ondas $\geq X$ no *StoreBuffer* local. Requisições em SBs remotos serão apagadas tardiamente quando eles recebem requisições para novas execuções;
5. Incrementar o atributo $lastExeN$ no *StoreBuffer* local e copiar este valor para o $currentExeN$ de todas as ondas $\geq X$;
6. Re-enviar todos os operandos no conjunto de leitura de X.

C.6 Experimentos e Resultados

C.6.1 Metodologia

Atualmente, os programas do WaveScalar devem ser compilados usando o compilador `cc` para a máquina Alpha com o sistema operacional Tru64. Depois disto, eles

são traduzidos para a linguagem de montagem do WaveScalar, usando um tradutor binário. O tradutor ainda não suporta o conjunto de instruções completo da máquina Alpha. Sendo assim, a maioria dos programas precisa rodar, parcialmente, no WaveScalar e, parcialmente, na máquina Alpha. Neste trabalho, o simulador arquitetural do WaveScalar (Kahuna) [42] foi estendido para suportar a Transactional WaveCache. A versão corrente ainda não suporta a execução de programas *multi-threaded* e *Decoupled Stores*. O suporte a emulação da máquina Alpha está disponível no simulador, mas antes que o controle seja transferido para o emulador Alpha é necessário parar a especulação e esperar que todas as ondas façam o *commit*. Esta foi a principal modificação introduzida depois da apresentação deste trabalho como dissertação de mestrado do presente autor. Tal alteração permite a execução de *benchmarks* clássicos, conforme exibido nesta seção.

Um conjunto de *benchmarks* do SPEC CPU2000, Mediabench e Mibench foi utilizado para avaliar o desempenho da Transactional WaveCache. Os *benchmarks* não foram executados por completo e, como o número de instruções executadas pode variar de acordo com a janela de especulação, o simulador foi modificado para permitir um corte semântico por ondas, que é feito no *commit*. Desta forma, é possível comparar o estado da memória para cada cenário com o WaveScalar original. Para o SPEC CPU2000, os *benchmarks* mcf, art e equake foram executados para as ondas correspondentes às primeiras duas milhões de instruções. Eles foram compilados usando a otimização O3. Os *benchmarks* G721 e EPIC da Mediabench e o CRC da Mibench foram compilados com a otimização O2 e as ondas correspondentes às 500 mil primeiras instruções foram executadas.

A Tabela C.1 mostra os parâmetros arquiteturais utilizados nas simulações. Todas as aplicações foram executadas no WaveScalar original, sem nenhum mecanismo de desambiguação de memória e também com *Decoupled Stores*. Eles também foram executados com a Transactional WaveCache, com Janelas de Especulação de tamanho 2. Para ter um entendimento mais detalhado do efeito da Janela de Especulação no desempenho, o equake e epic também foram executados para janelas de especulação com tamanhos 3, 5, 10, 20, 30 e com tamanho infinito. As estruturas da Transactional WaveCache (*WaveContext-Tables*, *Search Catalogs*, *MemOp-Histories* e *Execution Maps*) não foram limitados na implementação.

C.6.2 Resultados

A Figura C.1 mostra as acelerações para a Transactional WaveCache (TWC) e *Decouple Stores* para todas as aplicações, quando comparadas ao WaveScalar original sem desambiguação de memória. Aplicações de ponto flutuante (ART e EQUAKE) mostram acelerações significativas, mesmo para janelas de especulação pequenas

Tabela C.1: Parâmetros arquiteturais

Algoritmo de *Placement*	Exp2 (dinâmico)
Cache L1	Mapeamento Direto 1024 linhas de 32 bits Tempo de acesso: 1 ciclo
Cache L2	Associativa de 4 vias 131072 linhas de 128 bits Tempo de acesso: 7 ciclos
Tempo de acesso à memória	100 ciclos
Elementos de Processamento	Tamanho da fila de operandos: 10000000 linhas Tempo de busca nas filas: 0 Execuções por ciclo: 1 Operandos de entrada por instrução por ciclo: 3 Número de instruções: 8
StoreBuffers	4 portas de entrada 4 portas de saída

(1.35 e 1.31, respectivamente).MCF e EPIC realizam várias chamadas de função que são emuladas na máquina Alpha. Como foi necessário desabilitar a especulação para fazer isso, apenas o *overhead* da técnica foi ressaltado. Acelerações de 1.02 para o MCF e de 0.96 para o EPIC foram observados. Isto é causado pela emulação das chamadas de função que não seria necessária se existisse um compilador completo para o WaveScalar. Se acessos à memória ocorrem com frequência, a concorrência na memória será alta e a Transactional WaveCache tenderá a não prover acelerações. Como estão sendo executadas as primeira instruções para cada *benchmark*, se a fase de inicialização de variáveis for muito longa, haverá um longo período onde o mecanismo não ajudará do desempenho final.

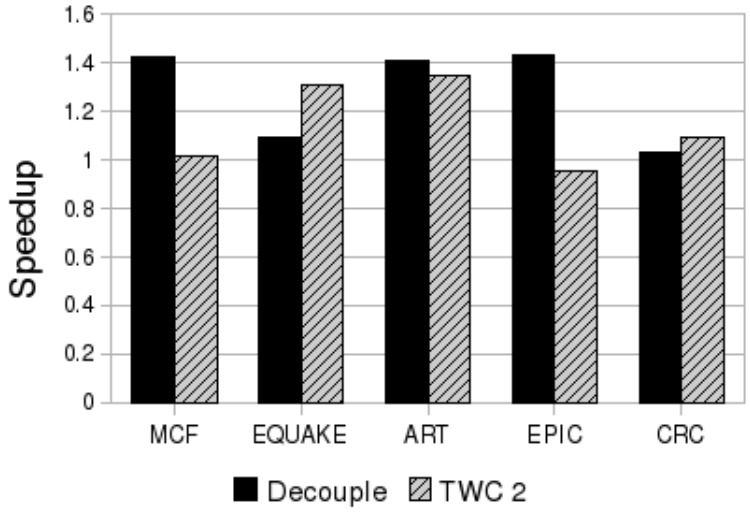

Figura C.1: TWC x Decouple Stores

O *Decoupled Stores* se saiu melhor que a Transactional WaveCache para a maioria

das aplicações. Como ambas as técnicas são ortogonais, uma combinação de ambas é possível e é sugerida como trabalho futuro.

A Figura C.2 mostra as acelerações atingidas para janelas de especulação com tamanho variando de 2 a infinito para o Equake. O aumento na aceleração segue o tamanho da janela de especulação (*Speculation Window*, no gráfico), embora a aceleração máxima (2,24) seja atingida para janelas de especulação com tamanho menor que 20. Isto mostra que as estruturas da Transactional WaveCache podem ser pequenas e ainda proporcionar ganhos de desempenho. O mesmo experimento foi feito para o EPIC. A aceleração constante (0,96) para todos os tamanho de janela de especulação mostra que ela não influencia o desempenho, visto que a especulação é desabilitada, continuamente, devido à chamada de funções emuladas.

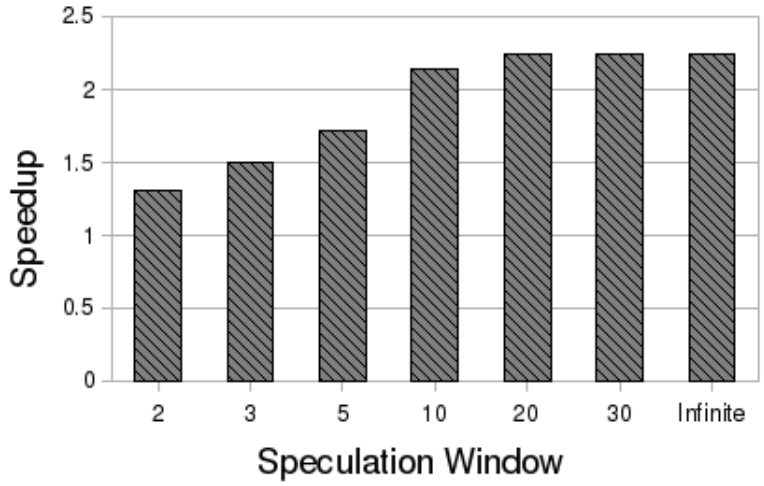

Figura C.2: Equake - Speedup x Especulação

Para todos os *bechmarks*, o Art foi a única aplicação que apresentou perigos RAW e, portanto, re-execuções. Isto era esperado para janelas de especulação pequenas. No entanto, no caso do Equake e Epic, os perigos RAW não foram encontrados nem mesmo para janelas de especulação maiores. O Epic é limitado pelas funções emuladas e o Equake não acessa o mesmo endereço de memória frequentemente, o que explica as grandes acelerações obtidas sem a ocorrência de perigos.

Apêndice D

Código fonte da aplicação Blackscholes

Neste apêndice é exibido o código completo da aplicação Blackscholes usada nos experimentos descritos neste trabalho. A intenção é mostrar o código completo de uma aplicação real, para que o leitor possa verificar o uso do modelo de forma mais completa. Como o código para as versões em *OpenMP*, *TBB* e *Pthreads* já é disponibilizado pelo PARSEC [21], aqui são apenas exibidas as versões para o TALM.

O código que se segue é referente à versão *fork/join* do Blackscholes, ou seja, usando as mesmas técnicas das versões disponibilizadas pelo PARSEC.

```
#BEGINBLOCK
// Copyright (c) 2007 Intel Corp.

// Black-Scholes
// Analytical method for calculating European Options
//
//
// Reference Source: Options, Futures, and Other Derivatives, 3rd Edition, Prentice
// Hall, John C. Hull,

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

//Precision to use for calculations
#define fptype float

#define NUM_RUNS 100
```

```
typedef struct OptionData_ {
        fptype s;          // spot price
        fptype strike;     // strike price
        fptype r;          // risk-free interest rate
        fptype divq;       // dividend rate
        fptype v;          // volatility
        fptype t;          // time to maturity or option expiration in years
                           //     (1yr = 1.0, 6mos = 0.5, 3mos = 0.25, ..., etc)
        char OptionType; // Option type. "P"=PUT, "C"=CALL
        fptype divs;       // dividend vals (not used in this test)
        fptype DGrefval; // DerivaGem Reference Value
} OptionData;

OptionData *data;
fptype *prices;
int numOptions;

int    * otype;
fptype * sptprice;
fptype * strike;
fptype * rate;
fptype * volatility;
fptype * otime;
int numError = 0;
int nThreads;


// Cumulative Normal Distribution Function
// See Hull, Section 11.8, P.243-244
#define inv_sqrt_2xPI 0.39894228040143270286

fptype CNDF ( fptype InputX )
{
    int sign;

    fptype OutputX;
    fptype xInput;
    fptype xNPrimeofX;
    fptype expValues;
    fptype xK2;
    fptype xK2_2, xK2_3;
    fptype xK2_4, xK2_5;
    fptype xLocal, xLocal_1;
    fptype xLocal_2, xLocal_3;

    // Check for negative value of InputX
    if (InputX < 0.0) {
```

```
        InputX = -InputX;
        sign = 1;
    } else
        sign = 0;

    xInput = InputX;

    // Compute NPrimeX term common to both four & six decimal
    // accuracy calcs
    expValues = exp(-0.5f * InputX * InputX);
    xNPrimeofX = expValues;
    xNPrimeofX = xNPrimeofX * inv_sqrt_2xPI;

    xK2 = 0.2316419 * xInput;
    xK2 = 1.0 + xK2;
    xK2 = 1.0 / xK2;
    xK2_2 = xK2 * xK2;
    xK2_3 = xK2_2 * xK2;
    xK2_4 = xK2_3 * xK2;
    xK2_5 = xK2_4 * xK2;

    xLocal_1 = xK2 * 0.319381530;
    xLocal_2 = xK2_2 * (-0.356563782);
    xLocal_3 = xK2_3 * 1.781477937;
    xLocal_2 = xLocal_2 + xLocal_3;
    xLocal_3 = xK2_4 * (-1.821255978);
    xLocal_2 = xLocal_2 + xLocal_3;
    xLocal_3 = xK2_5 * 1.330274429;
    xLocal_2 = xLocal_2 + xLocal_3;

    xLocal_1 = xLocal_2 + xLocal_1;
    xLocal   = xLocal_1 * xNPrimeofX;
    xLocal   = 1.0 - xLocal;

    OutputX = xLocal;

    if (sign) {
        OutputX = 1.0 - OutputX;
    }

    return OutputX;
}

// For debugging
void print_xmm(fptype in, char* s) {
    printf("%s: %f\n", s, in);
}
```

```
fptype BlkSchlsEqEuroNoDiv( fptype sptprice,
                            fptype strike, fptype rate, fptype volatility,
                            fptype time, int otype, float timet )
{
    fptype OptionPrice;

    // local private working variables for the calculation
    fptype xStockPrice;
    fptype xStrikePrice;
    fptype xRiskFreeRate;
    fptype xVolatility;
    fptype xTime;
    fptype xSqrtTime;

    fptype logValues;
    fptype xLogTerm;
    fptype xD1;
    fptype xD2;
    fptype xPowerTerm;
    fptype xDen;
    fptype d1;
    fptype d2;
    fptype FutureValueX;
    fptype NofXd1;
    fptype NofXd2;
    fptype NegNofXd1;
    fptype NegNofXd2;

    xStockPrice = sptprice;
    xStrikePrice = strike;
    xRiskFreeRate = rate;
    xVolatility = volatility;

    xTime = time;
    xSqrtTime = sqrt(xTime);

    logValues = log( sptprice / strike );

    xLogTerm = logValues;


    xPowerTerm = xVolatility * xVolatility;
    xPowerTerm = xPowerTerm * 0.5;

    xD1 = xRiskFreeRate + xPowerTerm;
    xD1 = xD1 * xTime;
```

```
    xD1 = xD1 + xLogTerm;

    xDen = xVolatility * xSqrtTime;
    xD1 = xD1 / xDen;
    xD2 = xD1 - xDen;

    d1 = xD1;
    d2 = xD2;

    NofXd1 = CNDF( d1 );
    NofXd2 = CNDF( d2 );

    FutureValueX = strike * ( exp( -(rate)*(time) ) );
    if (otype == 0) {
        OptionPrice = (sptprice * NofXd1) - (FutureValueX * NofXd2);
    } else {
        NegNofXd1 = (1.0 - NofXd1);
        NegNofXd2 = (1.0 - NofXd2);
        OptionPrice = (FutureValueX * NegNofXd2) - (sptprice * NegNofXd1);
    }

    return OptionPrice;
}


char *inputFile;
char *outputFile;

#ENDBLOCK

int main ()
{

     int a=0,b,d;

     parout int c;


     /***************** INIT AND READ *****************/
     super single input(a) output(b)
     #BEGINSUPER
              FILE *file;
              int i;
              int loopnum;
              fptype * buffer;
              int * buffer2;
              int rv;
```

```
#ifdef PARSEC_VERSION
#define __PARSEC_STRING(x) #x
#define __PARSEC_XSTRING(x) __PARSEC_STRING(x)
      printf("PARSEC Benchmark Suite Version "
           __PARSEC_XSTRING(PARSEC_VERSION)"\n");
      fflush(NULL);
#else
      printf("PARSEC Benchmark Suite\n");
      fflush(NULL);
#endif //PARSEC_VERSION

   if (superargc != 3)
     {
          printf("Usage:\n\tblacksholes <nthreads>
               <inputFile> <outputFile>\n");
          exit(1);
     }
    nThreads = atoi(superargv[0]);
    inputFile = superargv[1];
    outputFile = superargv[2];

    //Read input data from file
    file = fopen(inputFile, "r");
    if(file == NULL) {
      printf("ERROR: Unable to open file `%s'.\n", inputFile);
      exit(1);
    }
    rv = fscanf(file, "%i", &numOptions);
    if(rv != 1) {
      printf("ERROR: Unable to read from file `%s'.\n", inputFile);
      fclose(file);
      exit(1);
    }
    if(nThreads > numOptions) {
      printf("WARNING: Not enough work, reducing number of
          threads to match number of options.\n");
      nThreads = numOptions;
    }

    // alloc spaces for the option data
    data = (OptionData*)malloc(numOptions*sizeof(OptionData));
    prices = (fptype*)malloc(numOptions*sizeof(fptype));
    for ( loopnum = 0; loopnum < numOptions; ++ loopnum )
    {
     rv = fscanf(file, "%f %f %f %f %f %f %c %f %f", &data[loopnum].s,
          &data[loopnum].strike, &data[loopnum].r,
```

```
            &data[loopnum].divq, &data[loopnum].v, &data[loopnum].t,
            &data[loopnum].OptionType, &data[loopnum].divs,
            &data[loopnum].DGrefval);
        if(rv != 9) {
          printf("ERROR: Unable to read from file ‘%s’.\n", inputFile);
          fclose(file);
          exit(1);
        }
      }
      rv = fclose(file);
      if(rv != 0) {
        printf("ERROR: Unable to close file ‘%s’.\n", inputFile);
        exit(1);
      }

      printf("Num of Options: %d\n", numOptions);
      printf("Num of Runs: %d\n", NUM_RUNS);

#define PAD 256
#define LINESIZE 64

      buffer = (fptype *) malloc(5 * numOptions * sizeof(fptype) + PAD);
      sptprice = (fptype *) (((unsigned long long)buffer + PAD)
       & ~(LINESIZE - 1));
      strike = sptprice + numOptions;
      rate = strike + numOptions;
      volatility = rate + numOptions;
      otime = volatility + numOptions;

      buffer2 = (int *) malloc(numOptions * sizeof(fptype) + PAD);
      otype = (int *) (((unsigned long long)buffer2 + PAD)
       & ~(LINESIZE - 1));

      for (i=0; i<numOptions; i++) {
       otype[i]      = (data[i].OptionType == 'P') ? 1 : 0;
       sptprice[i]   = data[i].s;
       strike[i]     = data[i].strike;
       rate[i]       = data[i].r;
       volatility[i] = data[i].v;
       otime[i]      = data[i].t;
      }

      printf("Size of data: %d\n", numOptions *
       (sizeof(OptionData) + sizeof(int)));
#ENDSUPER

/*************** PROCESS ***************/
```

```
super parallel input(b) output(c)
#BEGINSUPER
    int i, j;
    fptype price;
    fptype priceDelta;
    int tid = treb_get_tid();
    int start = tid * (numOptions / nThreads);
    int end = start + (numOptions / nThreads);

    for (j=0; j<NUM_RUNS; j++) {
        for (i=start; i<end; i++) {
            /* Calling main function to calculate option value based on
             * Black & Sholes's equation.
             */
            price = BlkSchlsEqEuroNoDiv( sptprice[i], strike[i],
                                         rate[i], volatility[i], otime[i],
                                         otype[i], 0);
            prices[i] = price;

    #ifdef ERR_CHK
            priceDelta = data[i].DGrefval - price;
            if( fabs(priceDelta) >= 1e-4 ){
                printf("Error on %d. Computed=%.5f, Ref=%.5f, Delta=%.5f\n",
                       i, price, data[i].DGrefval, priceDelta);
                numError ++;
            }
    #endif
        }
    }
#ENDSUPER


/***************** WRITE *****************/
super single input(c::*) output(d)
#BEGINSUPER
        FILE *file;
        int i;
        int rv;
        //Write prices to output file
        file = fopen(outputFile, "w");
        if(file == NULL) {
          printf("ERROR: Unable to open file `%s'.\n", outputFile);
          exit(1);
        }
        rv = fprintf(file, "%i\n", numOptions);
        if(rv < 0) {
          printf("ERROR: Unable to write to file `%s'.\n", outputFile);
```

```
            fclose(file);
            exit(1);
          }
          for(i=0; i<numOptions; i++) {
            rv = fprintf(file, "%.18f\n", prices[i]);
            if(rv < 0) {
           printf("ERROR: Unable to write to file ‘%s’.\n", outputFile);
           fclose(file);
           exit(1);
            }
          }
          rv = fclose(file);
          if(rv != 0) {
            printf("ERROR: Unable to close file ‘%s’.\n", outputFile);
            exit(1);
          }

      #ifdef ERR_CHK
          printf("Num Errors: %d\n", numError);
      #endif
          free(data);
          free(prices);

  #ENDSUPER
  return 0;

}
```

O código que se segue mostra a versão aprimorada do Blackscholes, com técnicas para esconder a latência de operações de entrada e saída.

```
#BEGINBLOCK
// Copyright (c) 2007 Intel Corp.

// Black-Scholes
// Analytical method for calculating European Options
//
//
// Reference Source: Options, Futures, and Other Derivatives, 3rd Edition, Prentice
// Hall, John C. Hull,

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

//Precision to use for calculations
#define fptype float

#define NUM_RUNS 100

typedef struct OptionData_ {
        fptype s;          // spot price
        fptype strike;     // strike price
        fptype r;          // risk-free interest rate
        fptype divq;       // dividend rate
        fptype v;          // volatility
        fptype t;          // time to maturity or option expiration in years
                           //     (1yr = 1.0, 6mos = 0.5, 3mos = 0.25, ..., etc)
        char OptionType; // Option type. "P"=PUT, "C"=CALL
        fptype divs;       // dividend vals (not used in this test)
        fptype DGrefval; // DerivaGem Reference Value
} OptionData;

OptionData *data;
fptype *prices;
int numOptions;

int    * otype;
fptype * sptprice;
fptype * strike;
fptype * rate;
fptype * volatility;
fptype * otime;
int numError = 0;
```

```
int nThreads;

// Cumulative Normal Distribution Function
// See Hull, Section 11.8, P.243-244
#define inv_sqrt_2xPI 0.39894228040143270286

fptype CNDF ( fptype InputX )
{
    int sign;

    fptype OutputX;
    fptype xInput;
    fptype xNPrimeofX;
    fptype expValues;
    fptype xK2;
    fptype xK2_2, xK2_3;
    fptype xK2_4, xK2_5;
    fptype xLocal, xLocal_1;
    fptype xLocal_2, xLocal_3;

    // Check for negative value of InputX
    if (InputX < 0.0) {
        InputX = -InputX;
        sign = 1;
    } else
        sign = 0;

    xInput = InputX;

    // Compute NPrimeX term common to both four & six decimal accuracy calcs
    expValues = exp(-0.5f * InputX * InputX);
    xNPrimeofX = expValues;
    xNPrimeofX = xNPrimeofX * inv_sqrt_2xPI;

    xK2 = 0.2316419 * xInput;
    xK2 = 1.0 + xK2;
    xK2 = 1.0 / xK2;
    xK2_2 = xK2 * xK2;
    xK2_3 = xK2_2 * xK2;
    xK2_4 = xK2_3 * xK2;
    xK2_5 = xK2_4 * xK2;

    xLocal_1 = xK2 * 0.319381530;
    xLocal_2 = xK2_2 * (-0.356563782);
    xLocal_3 = xK2_3 * 1.781477937;
    xLocal_2 = xLocal_2 + xLocal_3;
    xLocal_3 = xK2_4 * (-1.821255978);
```

```
    xLocal_2 = xLocal_2 + xLocal_3;
    xLocal_3 = xK2_5 * 1.330274429;
    xLocal_2 = xLocal_2 + xLocal_3;

    xLocal_1 = xLocal_2 + xLocal_1;
    xLocal  = xLocal_1 * xNPrimeofX;
    xLocal  = 1.0 - xLocal;

    OutputX = xLocal;

    if (sign) {
        OutputX = 1.0 - OutputX;
    }

    return OutputX;
}

// For debugging
void print_xmm(fptype in, char* s) {
    printf("%s: %f\n", s, in);
}

fptype BlkSchlsEqEuroNoDiv( fptype sptprice,
                            fptype strike, fptype rate, fptype volatility,
                            fptype time, int otype, float timet )
{
    fptype OptionPrice;

    // local private working variables for the calculation
    fptype xStockPrice;
    fptype xStrikePrice;
    fptype xRiskFreeRate;
    fptype xVolatility;
    fptype xTime;
    fptype xSqrtTime;

    fptype logValues;
    fptype xLogTerm;
    fptype xD1;
    fptype xD2;
    fptype xPowerTerm;
    fptype xDen;
    fptype d1;
    fptype d2;
    fptype FutureValueX;
    fptype NofXd1;
    fptype NofXd2;
```

```
    fptype NegNofXd1;
    fptype NegNofXd2;

    xStockPrice = sptprice;
    xStrikePrice = strike;
    xRiskFreeRate = rate;
    xVolatility = volatility;

    xTime = time;
    xSqrtTime = sqrt(xTime);

    logValues = log( sptprice / strike );

    xLogTerm = logValues;


    xPowerTerm = xVolatility * xVolatility;
    xPowerTerm = xPowerTerm * 0.5;

    xD1 = xRiskFreeRate + xPowerTerm;
    xD1 = xD1 * xTime;
    xD1 = xD1 + xLogTerm;

    xDen = xVolatility * xSqrtTime;
    xD1 = xD1 / xDen;
    xD2 = xD1 - xDen;

    d1 = xD1;
    d2 = xD2;

    NofXd1 = CNDF( d1 );
    NofXd2 = CNDF( d2 );

    FutureValueX = strike * ( exp( -(rate)*(time) ) );
    if (otype == 0) {
        OptionPrice = (sptprice * NofXd1) - (FutureValueX * NofXd2);
    } else {
        NegNofXd1 = (1.0 - NofXd1);
        NegNofXd2 = (1.0 - NofXd2);
        OptionPrice = (FutureValueX * NegNofXd2) - (sptprice * NegNofXd1);
    }

    return OptionPrice;
}


char *inputFile;
```

```
char *outputFile;
FILE *infile;
FILE *outfile;

#ENDBLOCK

int main ()
{

    int a=0,b,f;

    parout int c,d,e;


    /***************** INIT ****************/
    super single input(a) output(b)
    #BEGINSUPER
            int i;
            int loopnum;
            fptype * buffer;
            int * buffer2;
            int rv;

        #ifdef PARSEC_VERSION
        #define __PARSEC_STRING(x) #x
        #define __PARSEC_XSTRING(x) __PARSEC_STRING(x)
            printf("PARSEC Benchmark Suite Version "
                __PARSEC_XSTRING(PARSEC_VERSION)"\n");
            fflush(NULL);
        #else
            printf("PARSEC Benchmark Suite\n");
            fflush(NULL);
        #endif //PARSEC_VERSION

          if (superargc != 3)
            {
                printf("Usage:\n\tblacksholes <nthreads>
                    <inputFile> <outputFile>\n");
                exit(1);
            }
            nThreads = atoi(superargv[0]);
            inputFile = superargv[1];
            outputFile = superargv[2];

            //open input file
            infile = fopen(inputFile, "r");
            if(infile == NULL) {
```

```
    printf("ERROR: Unable to open file ‘%s’.\n", inputFile);
    exit(1);
  }
  rv = fscanf(infile, "%i", &numOptions);
  if(rv != 1) {
    printf("ERROR: Unable to read from file ‘%s’.\n", inputFile);
    fclose(infile);
    exit(1);
  }
  if(nThreads > numOptions) {
    printf("WARNING: Not enough work, reducing number of threads
        to match number of options.\n");
    nThreads = numOptions;
  }

   //open output file
  outfile = fopen(outputFile, "w");
  if(outfile == NULL) {
    printf("ERROR: Unable to open file ‘%s’.\n", outputFile);
    exit(1);
  }
  rv = fprintf(outfile, "%i\n", numOptions);
  if(rv < 0) {
    printf("ERROR: Unable to write to file ‘%s’.\n", outputFile);
    fclose(outfile);
    exit(1);
  }
   // alloc spaces for the option data
   data = (OptionData*)malloc(numOptions*sizeof(OptionData));
   prices = (fptype*)malloc(numOptions*sizeof(fptype));
#define PAD 256
#define LINESIZE 64

  buffer = (fptype *) malloc(5 * numOptions * sizeof(fptype) + PAD);
  sptprice = (fptype *) (((unsigned long long)buffer + PAD)
       & ~(LINESIZE - 1));
  strike = sptprice + numOptions;
  rate = strike + numOptions;
  volatility = rate + numOptions;
  otime = volatility + numOptions;

  buffer2 = (int *) malloc(numOptions * sizeof(fptype) + PAD);
  otype = (int *) (((unsigned long long)buffer2 + PAD)
       & ~(LINESIZE - 1));

  printf("Num of Options: %d\n", numOptions);
  printf("Num of Runs: %d\n", NUM_RUNS);
```

```
        printf("Size of data: %d\n", numOptions *
              (sizeof(OptionData) + sizeof(int)));

#ENDSUPER

/**************** READ ****************/
super parallel input(starter.b, local.c::(mytid−1)) output(c)
#BEGINSUPER

         int i;
        int loopnum;
        int rv;
        int tid = treb_get_tid();
    int start = tid * (numOptions / nThreads);
        int end = start + (numOptions / nThreads);

        //Read input data from file
        for ( loopnum = start; loopnum < end; ++ loopnum )
        {
              rv = fscanf(infile, "%f %f %f %f %f %f %c %f %f",
                   &data[loopnum].s, &data[loopnum].strike,
                   &data[loopnum].r, &data[loopnum].divq,
                   &data[loopnum].v, &data[loopnum].t,
                    &data[loopnum].OptionType,
                    &data[loopnum].divs, &data[loopnum].DGrefval);
              if(rv != 9) {
                printf("ERROR: Unable to read from file `%s'.\n", inputFile);
                fclose(infile);
                exit(1);
              }
        }


        for (i=start; i<end; i++) {
         otype[i]      = (data[i].OptionType == 'P') ? 1 : 0;
         sptprice[i]   = data[i].s;
         strike[i]     = data[i].strike;
         rate[i]       = data[i].r;
         volatility[i] = data[i].v;
         otime[i]      = data[i].t;
        }
#ENDSUPER

/**************** PROCESS ****************/
super parallel input(c::mytid) output(d)
#BEGINSUPER
```

```
    int i, j;
    fptype price;
    fptype priceDelta;
    int tid = treb_get_tid();
    int start = tid * (numOptions / nThreads);
    int end = start + (numOptions / nThreads);

    for (j=0; j<NUM_RUNS; j++) {
        for (i=start; i<end; i++) {
            /* Calling main function to calculate option value based on
             * Black & Sholes's equation.
             */
            price = BlkSchlsEqEuroNoDiv( sptprice[i], strike[i],
                                         rate[i], volatility[i], otime[i],
                                         otype[i], 0);
            prices[i] = price;

    #ifdef ERR_CHK
            priceDelta = data[i].DGrefval - price;
            if( fabs(priceDelta) >= 1e-4 ){
                printf("Error on %d. Computed=%.5f, Ref=%.5f, Delta=%.5f\n",
                       i, price, data[i].DGrefval, priceDelta);
                numError ++;
            }
    #endif
        }
    }
#ENDSUPER


/***************** WRITE *****************/
super parallel input(d::mytid, local.e::(mytid-1)) output(e)
#BEGINSUPER
        int i;
        int rv;
        int tid = treb_get_tid();
    int start = tid * (numOptions / nThreads);
        int end = start + (numOptions / nThreads);
        //Write prices to output file
        for(i=start; i<end; i++) {
          rv = fprintf(outfile, "%.18f\n", prices[i]);
          if(rv < 0) {
            printf("ERROR: Unable to write to file `%s'.\n", outputFile);
            fclose(outfile);
            exit(1);
          }
        }
```

```
    #ENDSUPER


    /***************** END ****************/
    super single input(e::lasttid) output(f)
    #BEGINSUPER
            int rv = fclose(infile);
            if(rv != 0) {
              printf("ERROR: Unable to close file ‘%s’.\n", inputFile);
              exit(1);
            }
            rv = fclose(outfile);
            if(rv != 0) {
              printf("ERROR: Unable to close file ‘%s’.\n", outputFile);
              exit(1);
            }

        #ifdef ERR_CHK
            printf("Num Errors: %d\n", numError);
        #endif
            free(data);
            free(prices);

    #ENDSUPER
    return 0;

}
```
