OTIMIZANDO LAÇOS EM COMPUTAÇÃO POR FLUXO DE DADOS

Leandro Santiago de Araújo

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

Orientadores: Felipe Maia Galvão França
Leandro Augusto Justen Marzulo

Rio de Janeiro
Março de 2016

OTIMIZANDO LAÇOS EM COMPUTAÇÃO POR FLUXO DE DADOS

Leandro Santiago de Araújo

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

Examinada por:

Prof. Felipe Maia Galvão França, Ph.D.

Prof. Leandro Augusto Justen Marzulo, D.Sc.

Prof. Claudio Luis de Amorim, Ph.D.

Prof. Cristiana Barbosa Bentes, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
MARÇO DE 2016

Araújo, Leandro Santiago de

Otimizando Laços em Computação por Fluxo de Dados/Leandro Santiago de Araújo. – Rio de Janeiro: UFRJ/COPPE, 2016.

XIII, 77 p.: il.; 29, 7cm.

Orientadores: Felipe Maia Galvão França

Leandro Augusto Justen Marzulo

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2016.

Referências Bibliográficas: p. 75 – 77.

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.

"A menos que modifiquemos a nossa maneira de pensar, não seremos capazes de resolver os problemas causados pela forma como nos acostumamos a ver o mundo."

Albert Einstein

Agradecimentos

Aos membros da banca examinadora, por disponibilizarem seu tempo para avaliar este trabalho.

Agradeço aos meus orientadores Felipe França e Leandro Augusto Justen Marzulo pela motivação e o acompanhamento deste trabalho e por se disponibilizarem em me atender para tirar dúvidas, até mesmo pela internet. Obrigado pelo apoio e amizade.

Ao Programa de Engenharia de Sistemas e Computação (PESC/COPPE/UFRJ) pela oportunidade de aprender e ser capaz de realizar esse trabalho. Agradeço aos professores, que se dedicaram em fornecer o melhor conteúdo ensinando a apreciar o conhecimento aprendido. Aos funcionários da secretaria por serem atenciosos e prestativos.

À Universidade do Estado do Rio de Janeiro, pela oportunidade de aprender e conseguir uma boa formação. Agradeço aos professores, que se dedicaram em oferecer a melhor experiência possível para meu aprendizado.

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

Agradeço aos amigos que fiz na UFRJ: Alexandre, Alexsander, Brunno, Jean, João Paulo, Leonardo, Luciano, Luiz Fernando, Paulo, Rafael, Raul, Renan e Ygor. Obrigado a todos pela amizade e compartilhar os momentos durante essa trajetória.

À minha mãe, por sempre estar ao meu lado me dando forças para seguir em frente e por ter me ensinado o grande valor do estudo, educação e respeito. Obrigado pelo seu amor e por sempre acreditar na minha capacidade.

À minha namorada Isis, pela sua compreensão em todos os momentos de ausência e por sempre me dá forças para alcançar meus objetivos. Obrigado pelo seu amor, carinho e por sempre querer o que é melhor para mim. Te amo!

À Deus, por me conceder oportunidades e me guiar para o caminho certo. Obrigado por tudo.

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

OTIMIZANDO LAÇOS EM COMPUTAÇÃO POR FLUXO DE DADOS

Leandro Santiago de Araújo

Março/2016

Orientadores: Felipe Maia Galvão França
Leandro Augusto Justen Marzulo

Programa: Engenharia de Sistemas e Computação

O modelo de *dataflow* dinâmico permite que as iterações distintas de um *loop* executem simultaneamente, impulsionando a exploração de paralelismo no modelo. Para evitar que operandos produzidos em iterações futuras sejam utilizados em iterações passadas, cada operando possui um rótulo com um número de instância, que é incrementado a cada iteração do *loop*. A execução de uma instrução é disparada somente quando todos seus operandos de entrada com os mesmos rótulos se tornam disponíveis. Entretanto, este mecanismo de rótulo tradicional impõe a geração de instruções de controle desnecessárias para manipular os rótulos e garantir o casamento correto de operandos que não são processados dentro de um *loop*. Para tratar desse problema, este trabalho apresenta um conjunto de técnicas de otimização de *loop* no *dataflow*. O *Stack-Tagged Dataflow* é um mecanismo que utiliza pilha de inteiro no rótulo de cada operando para reduzir o controle de *overhead* em *dataflow*. Contudo, este mecanismo de pilha pode obter *overheads* significativos em *loops* aninhados. O *Tag Resetting* pode ser usado para zerar o rótulo de um operando quando ele alcançar um ponto seguro de execução, permitindo a redução de um nível da profundidade de pilha. Finalmente, o *Loop Skipping* contribui ainda mais com a redução do *overhead* da comparação de pilha em *loops*, quando o número de iterações de um *loop* pode ser previsto pelo compilador. Os resultados dos experimentos mostram que todas as técnicas contribuem com a redução do *overhead* em *loops* no *dataflow*. Os experimentos mostram que a execução de tarefas com granularidade mais fina se torna viável com a aplicação dessas técnicas. Além do mais, uma abordagem de compilação híbrida pode ser usada para melhorar o desempenho de cada técnica.

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

OPTIMIZING LOOPS IN COMPUTING FOR DATAFLOW

Leandro Santiago de Araújo

March/2016

Advisors: Felipe Maia Galvão França
Leandro Augusto Justen Marzulo

Department: Systems Engineering and Computer Science

The dynamic dataflow model allows different iterations of a loop run simultaneously, boosting parallelism exploitation in the model. To avoid produced operands in future iterations to be used in previous iterations, each operand has a tag with an instance number, which is incremented as it goes through the loop. Instruction execution is triggered only when all input operands with the same tag become available. However, this traditional tagging mechanism requires the generation of unnecessary control instructions to manipulate tags and guarantee that operands that are not used inside the loop match properly. To address this problem, this work presents a set of dataflow loop optimization techniques. The Stack-Tagged Dataflow is a mechanism that uses stacks of tags in each operand to reduce control overheads in dataflow. Nevertheless, this stack mechanism can achieve significant overheads in nested loops. The Tag Resetting can be used to reset the tag of an operand when it reaches a safe point of execution, allowing to reduce 1-level at the stack depth. Finally, Loop Skipping further contributes to reduce stack comparison overhead in loops, when the number of iterations can be predicted by the compiler. Experimental results show that all techniques contribute to reduce the overhead in dataflow loops. Experiments show that running tasks with finer granularity becomes feasible with the application of these techniques. Furthermore, a hybrid compiling approach can be used to improve the performance of each technique.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A evolução dos processadores de único núcleo se tornou impraticável devido a complexidade necessária para se obter desempenho. A quantidade de energia para suportar tal complexidade limita a frequência de relógio. Ainda que seja possível o aumento da quantidade de transistores por unidade de área, o aumento de instruções por ciclo se torna muito custoso. Isso acontece, devido a lógica adicional necessária para se encontrar instruções paralelas dinamicamente. [1]

As indústrias adotaram os processadores *multicore* como alternativa para o desenvolvimento de arquitetura de computadores. O aumento da densidade de transistores, garantido pela lei de Moore, passou a ser aproveitado para implementar múltiplos núcleos, normalmente mais simples. O foco para a exploração de paralelismo na aplicação passa a ser em nível de *threads*, que são distribuídas nos recursos de processamento disponíveis pela arquitetura. Assim, é possível obter ganhos de desempenhos mais significativos e tornar mais eficiente o consumo de energia [2].

A mudança na arquitetura também afetou o modelo de programação que era adotado. Alguns tipos de aplicações importantes expõem uma quantidade significativa de paralelismo, e podem se beneficiar de forma trivial nessa nova abordagem. Entretanto, outras aplicações são escritas inteiramente de forma sequencial dificultando sua paralelização. O resultado disso, é a necessidade de reescrever tais aplicações buscando encontrar porções de códigos que possam executar concorrentemente, com o objetivo de reduzir o tempo de execução.

A paralelização de aplicações não é uma tarefa trivial e exige um conhecimento mais profundo dos programadores. Para encontrar partes independentes do código com potencial ganho de desempenho é necessário aplicar técnicas de *profiling*. Em alguns problemas é necessário sua reformulação com base num modelo de programação paralela para que seja possível expor paralelismo. Além disso, em determinados momentos há a necessidade de compartilhar dados e sincronizar o fluxo de processamento. O programador é responsável por estabelecer a comunicação entre as *threads* respeitando os acessos corretos aos dados compartilhados por meio de *locks*,

barreiras e outras técnicas de sincronização. Também deve-se prever e tratar as possibilidades de ocorrência de *deadlocks* e manter o equilíbrio no balanceamento de carga para usufruir de maneira efetiva dos recursos disponíveis. Outra questão, é a definição de granularidade de tarefas que serão executadas em paralelo. Entre as possibilidades de escolha há um *tradeoff* entre complexidade e desempenho. As tarefas com granularidade grossa são fáceis de implementar e manter o controle, mas tendem a serializar a execução. Em contrapartida, as tarefas com granularidade fina expõem maior paralelismo, porém dificultam o gerenciamento de sincronização.

No modelo *dataflow*, a execução de um programa é guiada pelo fluxo de dados. As instruções são processadas a medida que seus operandos de entrada se tornam disponíveis e seus operandos de saída são enviados as instruções de destino. Dessa forma, é simples determinar as dependências de instruções e expor naturalmente o paralelismo da aplicação. As instruções independentes podem executar em paralelo, contanto que hajam recursos computacionais disponíveis.

Na década de 90, as pesquisas do modelo *dataflow* retornaram. Os estudos da época relataram que uma abordagem híbrida entre o modelo *dataflow* e o modelo de Von Neumann seria promissora para a formulação das arquiteturas de computadores futuras [3]. Blocos de instruções poderiam ser definidos em linguagens imperativas e processados conforme as regras *dataflow*.

Com base nessas ideias, foi desenvolvido o TALM (*TALM is an Architecture and Language for Multi-threading*) [4, 5], um modelo de execução baseado nos princípios *dataflow* para facilitar a exploração de paralelismo em *threads*. Nesse modelo, é possível definir blocos de código com linguagem imperativa, chamados de super-instruções, e utilizá-los em conjunto com instruções nativas conectadas em um grafo de fluxo de dados. As arestas do grafo expressam as dependências entre as instruções. A paralelização de uma aplicação no TALM consiste na definição de super-instruções e na descrição de suas dependências. Assim a execução do programa segue o fluxo de dados de acordo com as dependências de dados, expondo o paralelismo de forma natural.

A principal vantagem do TALM é oferecer um modelo de programação que permite definir a granularidade de tarefas que serão guiadas pelo fluxo de dados. A Trebuchet é uma máquina virtual desenvolvida com base no TALM, no qual é possível emular instruções no modelo *dataflow* e processar as super-instruções criadas pelo programador. Apesar das instruções serem interpretadas, os resultados obtidos em [4] mostram que são nas super-instruções que se concentram a maior parte do tempo de execução. Esta solução obtém melhor desempenho do que as tecnologias usadas no mercado, como OpenMP, Pthreads e *Intel Thread Building Blocks* (TBB), para aplicações que exigem técnicas complexas de paralelização.

Os *loops* são a maior fonte de paralelismo de uma aplicação. Algumas técnicas

de paralelismo em nível de instruções e em nível de *threads* otimizam os *loops* permitindo que iterações independentes sejam processadas em paralelo. A Trebuchet foi implementanda seguindo a abordagem de *dataflow* dinâmico. Nessa abordagem, os operandos possuem um rótulo indicando o número de instância, que é incrementado na transição entre as iterações. Assim, as iterações dos *loops* em *dataflow* podem ser executadas fora de ordem e uma instrução só poderá prosseguir seu processamento se todos seus operandos de entradas com o mesmo rótulo estiverem disponíveis.

Esse mecanismo possui um efeito indesejável - os operandos, que são produzidos antes de um *loop* e processados por instruções que o sucede, são forçados a passarem pelo *loop* para que seus rótulos sejam atualizados corretamente, de acordo com os operandos produzidos pela última iteração. Com isso, será possível ocorrer o casamento de rótulos com sucesso, permitindo que os operandos prossigam com o processamento após o término do *loop*. Essa restrição pode resultar em significantes *overheads*, intensificados com o aumento do número de iterações.

Para tratar desse problema, este trabalho apresenta um conjunto de técnicas de otimização em *loops* no *dataflow*. O *Stack-Tagged Dataflow* é um mecanismo no qual o rótulo de cada operando passa a ser uma pilha de inteiros. Ao entrar num processamento de *loop*, o operando é enviado a uma instrução de **Push** que empilhará um novo rótulo em sua pilha. No final de um *loop*, o operando é encaminhado a uma instrução de **Pop** responsável por desempilhar e descartar o topo da pilha. Na transição entre as iterações, o topo da pilha de cada operando é incrementado. Com essa nova abordagem, as variáveis que não são usadas no *loop* são enviadas diretamente para suas instruções de destino (depois do laço), evitando a execução desnecessária de instruções impostas pela a abordagem original.

Em *loops* aninhados, o processo de casamento de rótulos de pilha pode aumentar o *overhead* significativamente, afinal a pilha deve ser comparada por completo. O *Tag Resetting* pode reduzir esse custo eliminando um nível de profundidade de pilha de forma simples e elegante, zerando o rótulo dos operandos que alcançam algum ponto seguro de execução.

Por último, com o *Loop Skipping* é possível reduzir ainda mais o *overhead* da comparação de pilha, quando o compilador é capaz de prever o número de iterações de um *loop*. Neste caso, é possível eliminar as instruções de **Push** e **Pop**, incrementando apenas o número de iterações previsto nos rótulos das variáveis que passam por fora do *loop*. Dessa forma, essas variáveis terão seus rótulos comparados corretamente com os operandos produzidos pela última iteração do *loop*.

Para avaliar as técnicas propostas, elas foram implementadas no conjunto de ferramentas do TALM e foram conduzidos 6 experimentos. Além dessas técnicas, foi desenvolvida uma versão híbrida da Trebuchet que permite alternar a rotina de casamento (pilha ou inteiro) em tempo de execução. Também foi implementado

uma otimização do casamento de pilha. Os resultados obtidos mostram que todas as técnicas contribuem com a redução do *overhead* em *loops* no *dataflow*, tornando propício a execução de tarefas com granularidade mais fina que também é desejável para escalabilidade.

As contribuições deste trabalho são apresentadas a seguir:

- Criação das técnicas *Stack-Tagged Dataflow*, *Tag Resetting* e *Loop Skipping*.
- Implementação das técnicas na Trebuchet, uma implementação do TALM para arquiteturas *multicore*.
- Otimização da técnica *Stack-Tagged Dataflow* na Trebuchet.
- Inclusão de novas instruções na linguagem de montagem do TALM correspondentes as técnicas de otimização.
- Inclusão da lógica para geração das novas instruções no Couillard, um compilador para a linguagem de alto nível do TALM.
- Execução de experimentos para avaliar o custo de cada técnica.
- Execução de experimentos para avaliar o comportamento de cada técnica em aplicações artificiais, mostrando como a redução de controle de laços pode obter desempenho em relação a abordagem original do *dataflow* dinâmico.

O restante deste trabalho está dividido da seguinte forma: *(i)* o Capítulo 2 apresenta de forma geral o modelo *dataflow*; *(ii)* o Capítulo 3 apresenta o modelo TALM; *(iii)* os detalhes do conjunto de ferramentas para a implementação do TALM são apresentados no Capítulo 4; *(iv)* as técnicas de otimização de *loops* em *dataflow* são descritas no Capítulo 5; *(v)* os experimentos e as análises dos resultados são discutidos no Capítulo 6; *(vi)* e a conclusão e os trabalhos futuros são discutidos no Capítulo 7.

Capítulo 2

Modelo *Dataflow*

Os processadores *multicore* surgiram como alternativa para o aumento de desempenho. Uma tarefa complexa é dividida em trechos que possam executar em paralelo nos processadores simples. Assim, o tempo do processamento total de uma tarefa pode ser reduzido, conforme mais trechos possam ser alocados nos recursos disponíveis.

A abordagem de *multicore* possui alguns problemas: (i) o aumento no número de núcleos não garante melhor desempenho, (ii) a comunicação entre muitos processadores pode ocasionar degradação considerável no desempenho, (iii) o programador precisa conhecer características específicas do *hardware* para usufruir o potencial máximo fornecido pelos processadores, e (iv) certos problemas são essencialmente dependentes dos dados dificultando o desenvolvimento de uma solução paralela. Por fim, a limitação da exploração de paralelismo também é dada pela arquitetura predominante nos computadores atuais baseada no modelo de Von Neumann.

Na década de 70, Dennis [6] apresentou o modelo *dataflow* como uma alternativa de arquitetura de computador. O principal objetivo da proposta era eliminar os entraves do modelo de Von Neumann. Neste capítulo serão discutidas as principais ideias do modelo *dataflow*.

2.1 O Modelo *Dataflow*

Alguns desafios de arquitetura de computadores do futuro são discutidos em [7]. Os desafios mencionados são: como aumentar o desempenho com o custo mínimo de *hardware* e como facilitar a programação de abstrações via *software* em arquiteturas complexas. A proposta de construir o computador com modelos baseado em linguagens naturais é desejável para garantir que uma arquitetura seja programável. Nesse tipo de abordagem, o sistema de computador se comportaria como um interpretador de *hardware* para uma linguagem base específica, e os programas só poderiam ser expressos com a linguagem estabelecida pelo *hardware*. Esse tipo de abordagem não

teve sucesso, pois as linguagens convencionais sofrem de limitações das máquinas convencionais (memória global) e não são aptas para expressar concorrência.

O modelo de Von Neumann é composto por três componentes chaves: um programa armazenado na memória, uma memória global para armazenamento de dados e um contador de programa que guia a execução do programa armazenado.[8] O contador de programa indica qual instrução deve ser executada no ciclo corrente e é atualizado para informar as instruções seguintes. Esse processo é conhecido como fluxo de controle sendo caracterizado pela execução sequencial das instruções. O modelo *dataflow* surgiu com a proposta de explorar paralelismo de forma natural. Nesse modelo, a execução de instruções é guiada pelo fluxo de dados. Uma instrução necessita de operandos de entrada para produzir um resultado, que será usado por outras instruções no futuro. A ordem de execução é dada pela disponibilidade dos operandos necessários para o processamento das instruções.

Os modelos de computadores baseados nos princípios de fluxo de dados se tornaram interessantes como uma alternativa de arquitetura de computador, por derivar da noção de execução convencional de programas sequenciais e ser compatível com os conceitos modernos de estrutura de programa (facilitando a definição de programação da arquitetura). Além disso, ele oferece uma solução eficiente para exploração de concorrência em alta escala.[7]

O modelo *dataflow* também evita dois tipos de gargalos existentes: atualização de memória global e o contador de programa global. Esses dois gargalos são tratados, respectivamente, com uso de memória local e a execução de instruções a partir de seus operandos disponíveis.[3]

Os programas em máquinas *dataflow* são representados como um grafo de fluxo de dados. Assim, pode se considerar que os computadores *dataflow* são uma forma de arquitetura baseada em linguagem natural e o grafo de fluxo de dados é sua linguagem base. Na próxima seção será definido o grafo *dataflow*.

2.2 O Grafo *Dataflow*

Programas no modelo de fluxo de dados podem ser representados como um grafo direcionado onde os vértices representam as instruções e as arestas estabelecem as dependências entre elas. Um vértice pode ter arestas de entrada que apontam para o mesmo, indicando que a instrução necessita dos operandos para poder computar, e pode ter arestas de saída indicando os destinos dos operandos produzidos pela instrução. No início de um programa, nós de ativação especiais colocam os dados iniciais em certos vértices específicos disparando a execução do programa. Quando um vértice recebe um dado, esse dado é disponibilizado em uma de suas portas de entrada. O nó é ativado ao receber todos os operandos de entrada e, em seguida, os

dados são consumidos no processamento resultando em um novo dado. O resultado é colocado em uma de suas portas de saída e encaminhado para outros vértices. Os operandos são transportados de um nó para o outro dentro de pacotes de dados chamado de *tokens*. Assim, uma aresta direcionada $A \rightarrow B$ indica que a instrução B depende de um operando de saída da instrução A. A Figura 2.1 mostra um exemplo do grafo de fluxo de dados.

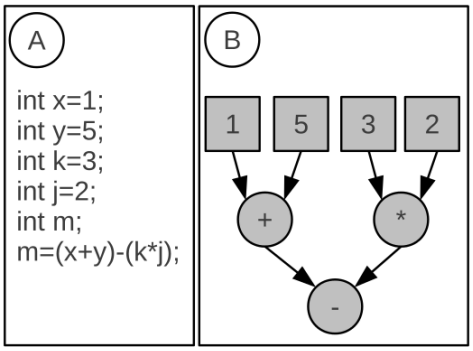

Figura 2.1: Exemplo de um programa em *dataflow*. O quadro A mostra um trecho de código em linguagem de alto nível e o quadro B apresenta o grafo *dataflow* correspondente ao código. Os operandos são trafegados entre as instruções (nós) pelas arestas (dependências). [4]

A representação de grafo permite observar que várias instruções podem ser ativadas simultaneamente, ou seja, muitas instruções podem ser processadas em paralelo. Outro detalhe importante é que cada operação é funcional - um dado nunca é modificado e o resultado de uma execução sempre cria um novo dado. Essa característica torna os efeitos colaterais inexistentes, garantindo que um conjunto de operandos de entrada de um programa sempre produzirá o mesmo conjunto de operandos de saída.

Há necessidade de linguagens de programação de alto nível para expressar programas em máquinas *dataflow*, pois a representação em grafos não é apropriada como linguagem de programação. Ela é propícia a erros e de difícil manipulação. No entanto, a existência de uma memória global na arquitetura possibilita a ocorrência de efeitos colaterais, dificultando a elaboração de tais linguagens. [9]

2.3 Tipos de Modelo de Execução *Dataflow*

O modelo de execução de fluxo de dados mais adotado no campo de pesquisa é o modelo *dataflow* baseado em *token*. Esse modelo é baseado no fluxo de *tokens* de dados como descrito na seção 2.2.

Na literatura, existe uma alternativa de modelo de execução conhecida como modelo de estruturas. Esse modelo contém os mesmos nós e arestas do modelo

baseado em *tokens*, mas cada vértice pode criar uma ou mais estruturas de dados em suas arestas de saída. As estruturas de dados conseguem manter um *array* de valores preservando o mesmo processo de execução do modelo baseado em *tokens* com as seguintes vantagens: acesso aleatório aos dados e sensibilidade ao histórico de execução do programa. [3]

A principal diferença entre os dois modelos é a forma de exibição dos dados. No modelo baseando em *token*, os *tokens* de dados são executados em sequência pelos nós - que são projetados como processadores de *streams* com armazenamento de dados de forma eficiente e sequencial. Já no modelo de estruturas, os nós processam sobre as estruturas sem o conceito de *stream* de estruturas necessitando recorrer ao suporte de linguagens e compiladores eficientes. Ele também requer que todos os dados gerados sejam armazenados para preservar a capacidade de examinar o histórico do programa. O termo *i-structures* foi inventado para estruturas incompletas - estruturas que precisam de um *hardware* especial para adiar a busca de elementos não disponíveis [10].

O modelo em estruturas parece ser mais vantajoso pela flexibilidade de escolha no armazenamento de dados e a preservação do histórico de execução, porém não consegue armazenar dados eficientemente. Como no modelo baseado em *tokens* é possível representar os *streams* por infinitos objetos dando a vantagem de acesso aleatório no *stream* e preservando dados antigos, ele é o modelo de execução mais adotado nas arquiteturas *dataflow*.

2.4 Arquiteturas *Dataflow*

A construção de uma máquina *dataflow* é um grande desafio, embora a teoria do modelo *dataflow* seja simples e eficiente. A maior dificuldade é atender os requisitos do modelo. Primeiro, o modelo assume que as arestas são representadas por filas com capacidade ilimitada, o que é impossível de ser obtido em uma memória física real. É necessário aplicar técnicas complexas para armazenamento de *tokens* para melhor atender esse requisito. Segundo, o modelo assume que qualquer quantidade de instruções pode ser executada em paralelo, o que também é inviável pela quantidade finita de processadores.

Com essas suposições do modelo nota-se que é impossível de construir um *hardware* que reflita exatamente o que foi proposto. Há necessidade de fazer algumas mudanças sutis que podem gerar casos com possibilidade de *deadlock*, o que não é previsto no modelo *dataflow* puro [3].

Diversas arquiteturas *dataflow* [9] foram criadas e expandidas com o objetivo de desenvolver uma proposta para um processador geral usando uma linguagem *dataflow* generalizada. Nesta seção serão apresentadas algumas arquiteturas que

buscaram resolver o problema de obter uma implementação viável do modelo de fluxo de dados.

2.4.1 Abordagens de arquitetura

As propostas iniciais do modelo *dataflow* consideravam os *tokens* de dados como elementos passivos que aguardariam nas arestas até serem consumidos. No entanto, rapidamente mudou-se o funcionamento do modelo e o controle de execução passou a ser orientado por dados. A implementação dessa característica do modelo de fluxo de dados pode seguir dois caminhos. [3]

A primeira abordagem é conhecida como abordagem orientado a dados (*data-driven approach*). Nessa abordagem, a execução depende da disponibilidade do dado. Um nó fica inativo enquanto os dados estão sendo trafegados nas suas arestas de entrada. Um dispositivo global é responsável de ativar os nós quando os dados de entrada correspondentes chegam. O processo é constituído de duas fases:

1. O nó é ativado quando os dados de entradas estão disponíveis.
2. Os dados são consumidos e os *tokens* resultantes são colocados nas arestas de saída.

A segunda abordagem é a abordagem orientada a demanda (*demand-driven approach*). Nela, um nó é ativado somente quando recebe uma requisição de dado por uma de suas arestas de saída. Uma vez que recebe os dados de entradas, ele processa e cria os dados solicitados colocando-os nas arestas de saída por onde vieram as requisições. Com isso, o nó demanda dados para todas arestas de entrada relevantes. A execução do programa inicia quando o ambiente do grafo demanda alguma saída do grafo. O processo é composto por quatro fases:

1. O ambiente do nó faz a requisição do dado.
2. O nó é ativado e faz requisição dos dados de entrada ao seu ambiente.
3. O ambiente responde com os dados solicitados.
4. O nó executa e coloca os dados gerados em suas arestas de saída.

Cada abordagem tem suas vantagens. A abordagem orientada a dados tem a vantagem de não ter o *overhead* de propagação de requisições de dados que sobem no grafo *dataflow* a partir de alguma saída. Já na abordagem orientada a demanda, tem a vantagem de eliminar certos tipos de nós que não são necessários. Como exemplo, podemos citar que o nó de desvio apresentado na Figura 2.2 seria desnecessário pois somente uma saída seria ativada - *True* ou *False*. A saída ativada enviaria uma

requisição de dados subindo o grafo, conforme a abordagem apresentada. Assim, bastaria ter um nó com uma única porta de saída. Esse exemplo mostra como a programação em *dataflow* pode ser afetada pela escolha da implementação física e do modelo de execução.

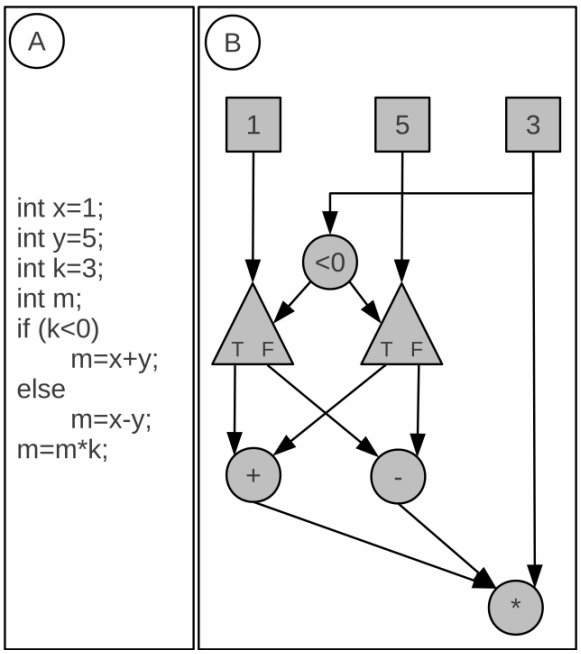

Figura 2.2: Exemplo de desvios em *dataflow*. O quadro A mostra um trecho de código em linguagem de alto nível e o quadro B apresenta o grafo *dataflow* correspondente ao trecho. Os triângulos representam as instruções de desvios. [4]

2.4.2 Arquitetura Estática

A arquitetura estática foi proposta por Denis e Misunas [11]. Ao invés de utilizar o mecanismo de filas das arestas para armazenar os operandos de entrada, cada aresta comporta um único *token* de dado. Um nó é ativado conforme a seguinte regra: um *token* tem que estar disponível em cada aresta de entrada e não pode existir *tokens* nas arestas de saída.

A implementação é feita com arestas de reconhecimento inseridas no grafo *dataflow*. Essas arestas têm direção oposta a cada aresta original do grafo e carregam um *token* de reconhecimento. Antes de um nó começar a sua execução, ele deve receber todos os *tokens* de reconhecimento correspondentes. O *hardware* é simples, já que não precisa gerenciar filas de *tokens*. O grafo é armazenado como um conjunto de *templates*, onde cada *template* representa um vértice armazenando informações como: *opcode*, espaço de memória dos dados de entrada, lista de endereços de destino e uma *flag* informando se o nó está ativo. O nó ativo tem seu endereço do *template* armazenado numa fila de instruções. A unidade de busca remove cada *template* da

fila e envia um pacote de operação para a unidade de operação adequada. O resultado produzido é enviado para a unidade de atualização que o coloca na aresta alvo correta com o endereço de destino do *template*. A unidade de atualização verifica quais *templates* estão ativos e os coloca na fila de instruções fechando o ciclo do processo.

Essa arquitetura possui as seguintes características: simplicidade e rapidez em detectar se um nó está ativo ou inativo, a memória pode ser alocada as arestas em tempo de compilação e uma aresta só pode carregar no máximo um *token*. Um problema nessa arquitetura é o aumento de tráfego provocado pelo rápido crescimento das arestas de reconhecimento. Outro problema é dado pela espera de todos os *tokens* de reconhecimento de um nó antes dele começar sua execução. A arquitetura estática limita a execução de *loops* - em cada iteração é necessário aguardar o término de todas as operações da iteração anterior antes de prosseguir com a execução da iteração corrente.

Algumas implementações de máquina *dataflow* com arquitetura estática são apresentadas em [9, 10, 12].

2.4.3 Arquitetura Dinâmica (*Tagged-token)*

A arquitetura dinâmica, que também é conhecida como modelo dinâmico, foi proposta por Arvid e Culler [13]. Ela expõe paralelismo adicional permitindo múltiplas invocações de subgrafos. Com essa característica, operações com *loop* são beneficiadas - cada iteração pode ser executada fora de ordem sem aguardar o término de todas as operações da iteração anterior.

Na implementação, uma cópia do grafo é armazenada na memória e cada *token* possui um rótulo (*tag*) utilizado para distinguir sua instância. O rótulo é um identificador da invocação (ou iteração). Ao contrário do modelo estático, cada aresta do modelo dinâmico pode conter uma grande quantidade de *tokens*. Um nó é ativado quando todos os dados de entrada possuem o mesmo rótulo da iteração que o vértice se encontra. As instâncias de um nó podem ser executadas fora de ordem e os rótulos garantem que não haja conflito - um dado gerado de uma iteração futura ser usado numa iteração do passado. Os rótulos são gerados pelo sistema.

A Figura 2.3 mostra um exemplo de grafo *dataflow* com operação de *loop*. Na operação de *loop*, o rótulo de cada nó envolvido começa com o identificador da iteração igual a zero. No início de cada iteração, um operador de controle especial incrementa o rótulo de cada operando.

O *hardware* da arquitetura baseada no modelo dinâmico é mais complexo que o modelo estático. Unidades adicionais são necessárias para processar diversos *tokens* em uma aresta e executar a correlação dos rótulos. Também precisa-se de mais

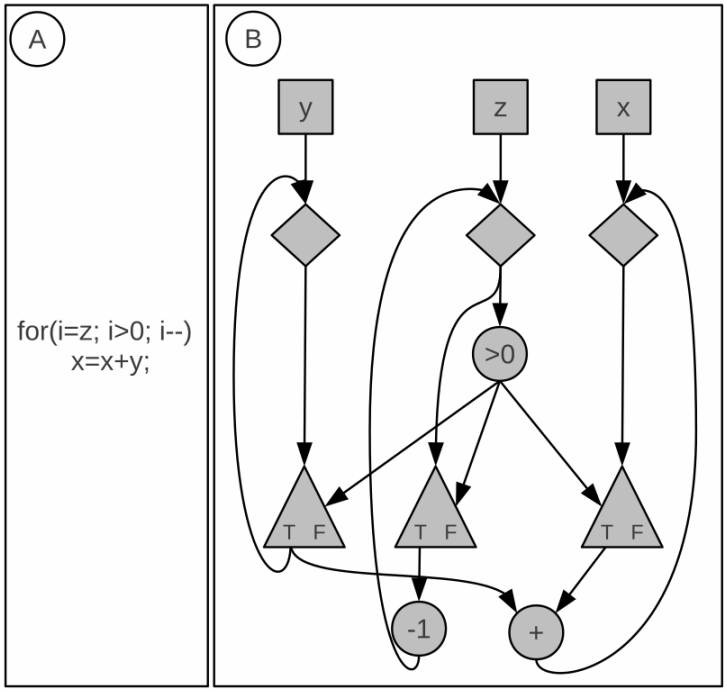

Figura 2.3: Exemplo de laços em *dataflow*. O quadro A mostra um trecho de código em linguagem de alto nível e o quadro B apresenta o grafo *dataflow* correspondente ao trecho. Os losangos representam as instruções que incrementam os rótulos de iteração dos operandos. [4]

memória para armazenar os *tokens* extras acumulados nas arestas. A vantagem desse modelo é a possibilidade de sobreposição de execução de tarefas podendo até mesmo executar diferente iterações de *loops* ao mesmo tempo. A principal desvantagem do modelo é o *overhead* extra obtido pelo casamento de rótulos, ao invés de uma *flag* indicando se o nó está ativo. Por ter execução fora de ordem, o armazenamento de dados com memória associativa não é tão benéfico tornando lento os acessos a memória. [3]

Algumas implementações de máquina *dataflow* com arquitetura dinâmica são apresentadas em [9, 10, 12, 14–18].

2.5 *Dataflow* Híbrido

No início dos anos 90, as pesquisas na área de *dataflow* se intensificaram novamente. O principal tema abordado era a questão do nível de granularidade das aplicações. Um dos primeiros trabalhos realizados reconheceu que as técnicas de *dataflow* e Von Neumann não eram mutuamente exclusivas e que seus conceitos poderiam se relacionar dando continuidade a possíveis arquiteturas de computadores. O *dataflow* com granularidade fina poderia ser visto como uma arquitetura *multithread*. Algumas pesquisas observaram como se comporta o paralelismo com as mudanças do nível de

granularidade. O resultado dessas mudanças levou a exploração da área conhecida como *dataflow* híbrido. [3]

Em [19] foi explorado o desempenho de diferentes níveis de granularidade de aplicações em máquinas *dataflow*. A Figura 2.4 mostra os resultados obtidos nas pesquisas a partir de um grafo genérico obtido com o resumo dos cenários de teste. De acordo com a Figura 2.4, o melhor desempenho de paralelismo não é alcançado com o nível de granularidade fina (tradicional *dataflow*) e nem com o nível de granularidade grossa (execução sequencial) do *dataflow*. A abordagem de granularidade média é a que oferece o melhor desempenho. Daí, surge alguma formas de *dataflow* híbrido: *dataflow* com extensões de Von Neumann ou vice-versa. Entretanto, o trabalho não identifica qual é o melhor nível médio de granularidade e qual combinação híbrida oferece o melhor desempenho. [3]

Não há como definir um consenso universal da melhor forma híbrida que pode ser alcançada em termos de arquitetura de *hardware*. Algumas abordagens híbridas são arquiteturas de Von Neumann com poucas adições de *dataflow* e outras são essencialmente arquiteturas *dataflow* com alguns comportamentos de Von Neumann. A comunidade de programação *dataflow* se dividiu em dois grupos relacionados a abordagem que obteria o melhor ganho de desempenho: o grupo que defendia a geração de grafos com grãos finos e outro grupo que defendia a abordagem de granularidade grossa. [3]

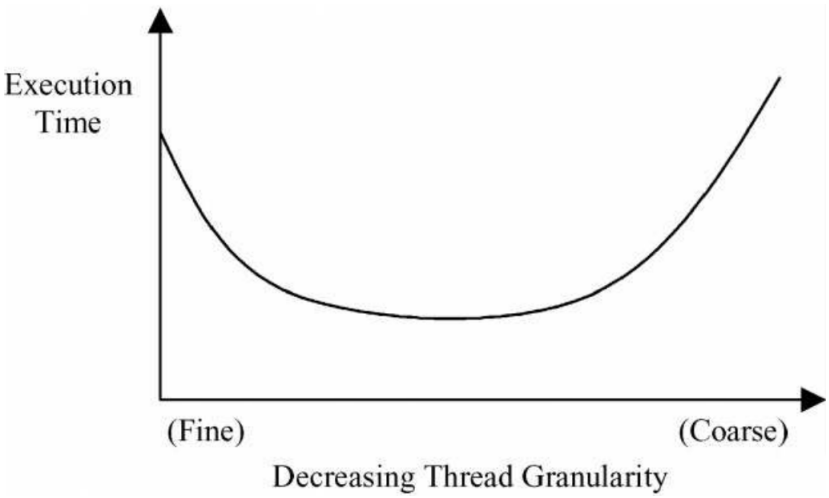

Figura 2.4: Curva de otimização de granularidade em *dataflow*. O gráfico mostra que o menor tempo de execução deve ser alcançado por uma abordagem de granularidade média. [3]

O primeiro grupo propôs analisar o grafo com grãos finos e identificar subgrafos com baixo nível de paralelismo no qual deveria ser executado em sequência. Os nós dos subgrafos identificados são agrupados em segmentos. Quando o primeiro nó do segmento é executado, os demais vértices do segmento prosseguem com sua

execução imediatamente após o fim do vértice anterior. Dessa maneira a execução ainda é feita com grãos finos, mas o custo de processar o casamento dos *tokens* para os nós subsequentes dos segmentos é evitado. Essa abordagem foi denominada de *threaded dataflow*. A vantagem desse método é a eliminação do *overhead* pelas partes executadas sequencialmente e a economia de tempo e recursos. [3]

O segundo grupo defende a suspensão de grãos finos na execução do *dataflow*. Ao invés de agrupar nós associados, os subgrafos são compilados em processos sequenciais de Von Neumann engrossando o grão dos nós. O processo é executado em um ambiente *multithread* e escalonado de acordo com a regra de execução do *dataflow*. A diferença é que cada nó contém, por exemplo, uma função inteira expressa em linguagem sequencial. Essa abordagem ficou conhecida como *large-grain dataflow*. [3]

Capítulo 3

TALM

Várias técnicas foram propostas para melhor explorar o potencial paralelismo nos processadores *multicore*. Dentre as técnicas, se tem o algoritmo de Tomasulo [20], onde a execução das instruções é feita fora de ordem seguindo a ideia do modelo de fluxo de dados. A abordagem de Tomasulo é limitada, pois explora paralelismo em nível de instrução e o despacho das instruções ainda é feito sequencialmente.

Visando aproveitar a oportunidade de melhorar a exploração de paralelismo com o modelo de fluxo de dados de forma compatível com as arquiteturas em vigor, em [4, 5] foi criado o TALM (*TALM is an Architecture and Language for Multi-threading*): um modelo de execução que oferece execução de *threads* com disparo guiado por fluxo de dados em máquinas de Von Neumann. O modelo é flexível podendo ser implementado em diversos tipos de plataforma (como CPU e GPU). Os programas podem ser paralelizados com diferentes níveis de granularidade. O modelo também disponibiliza um conjunto de instruções e uma arquitetura base, que dão suporte ao ambiente de execução do modelo *dataflow*.

Neste capítulo serão apresentados a arquitetura e o conjunto de instruções do TALM.

3.1 A Arquitetura TALM

A arquitetura TALM é composta por um conjunto de Elementos de Processamento (EPs) homogêneos interconectados por uma rede, conforme apresentado na Figura 3.1.

Os Elementos de Processamento interpretam as instruções e as executam de acordo com as regras de disparo do modelo *dataflow*. Cada EP possui um *buffer* de comunicação para armazenar as mensagens recebidas de outros EPs através da rede. As mensagens contém os operandos produzidos pelas outras instruções processadas em EPs diferentes. Um mensagem também pode conter marcadores necessários pelos

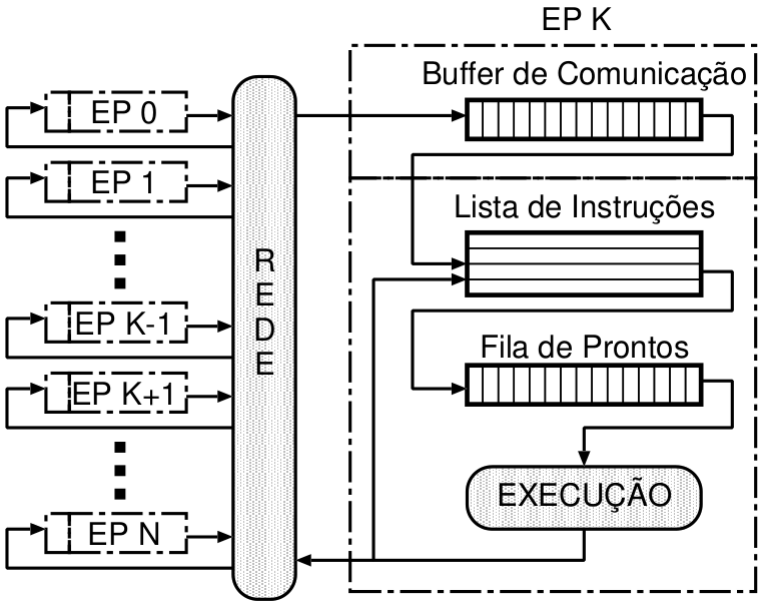

Figura 3.1: Arquitetura do TALM. [4]

algoritmos distribuídos implementados no sistema, como é o caso do algoritmo de detecção de terminação global [4].

Para uma aplicação ser executada no TALM, primeiro suas instruções devem ser mapeadas nos diferentes EPs disponíveis. Cada instrução possui uma lista de operandos correspondentes as suas instâncias dinâmicas. O controle dos operandos de uma instrução em diferentes instâncias é feito através de um rótulo pertecente a cada operando. O rótulo identifica a instância do operando e é utilizado no processo de casamento entre as operações evitando que instâncias futuras interajam com as instâncias passadas, conforme discutido na Seção 2.4.3. A instrução e os operandos resultantes do casamento são enviados para uma fila de prontos. A partir daí, os EPs removem as instruções contidas na fila de prontos e as enviam para a execução.

Por não existir contador de programa em máquinas *dataflow*, o término da execução do programa é determinado por um um algoritmo distribuído de detecção de terminação global adaptado a implementação do modelo.

3.2 Conjunto de Instruções

O TALM permite que o programador defina a granularidade de uma aplicação a ser executada guiada pelo fluxo de dados. A granularidade é definida através de super-instruções que agrupam instruções básicas que serão processadas seguindo o modelo de Von Neumann. Além de paralelizar as instruções básicas é possível paralelizar o bloco de instruções definido na super-instrução. Nesta seção serão descritas a arquitetura, o formato do conjunto de instruções e as instruções básicas fornecidas

pelo TALM.

3.2.1 Formato das Instruções

A Figura 3.2 apresenta o formato de uma instrução do TALM. Esse formato permite que instruções tenham números distintos de operandos de entrada e saída.

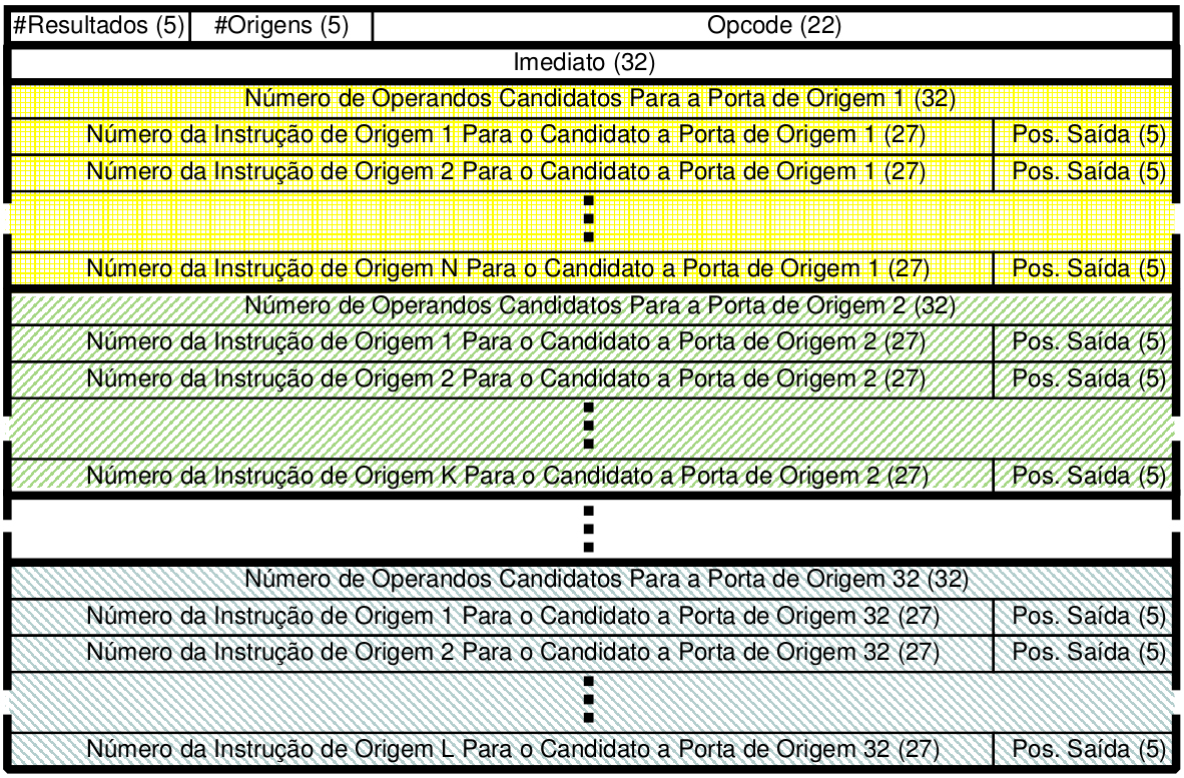

Figura 3.2: O formato de uma instrução no TALM. [4]

Os primeiros 32 *bits* são obrigatórios para todas as instruções. Os demais *bits* são ajustados quando necessário. Dos 32 *bits* iniciais, 22 *bits* são reservados para a definição do *opcode* e 5 *bits* para quantidade de operandos de entrada e saída. Dessa forma, cada instrução suporta um total de 32 operandos de entradas e 32 operandos de saída. Os 32 *bits* seguintes são opcionais e destinados ao operando com valor imediato necessário em instruções que operam com valor imediato.

No grafo *dataflow* uma instrução pode depender de um operando de entrada produzido por diferentes instruções. Um operando de saída pode ser encaminhado para diferentes instruções segundo alguma condição. O TALM suporta a execução de instruções com operandos vindo de origens diferentes, representando os operandos de entrada em uma lista de origens. Cada porta de entrada do operando possui uma lista de origens. Durante a execução, um único caminho será escolhido como origem de um operando. Esse caminho será dito disponível quando sua porta de entrada for preenchida por algum elemento de sua lista de entrada.

A representação da lista das origens é feita da seguinte forma. Os primeiros 32 *bits* da lista especifica a quantidade de possíveis origens para o operando. Para cada origem são usados 32 *bits*, divididos em dois campos de 27 e 5 *bits*, respectivamente. Os 27 *bits* iniciais informam o endereço da instrução de origem e os 5 *bits* restantes definem a porta de saída da instrução, no qual o operando pode ser enviado. Assim, os valores $\langle i|p \rangle$ nesses dois campos indicam que um operando pode ser recebido pela p-ésima porta de saída da i-ésima instrução.

3.2.2 Tipo das Instruções

O TALM fornece um conjunto de instruções nativas. Nesse conjunto estão incluídas as seguintes instruções: lógicas e aritméticas (como add, sub, and e or; mais as suas variações com o operando imediato); instruções responsáveis pelo controle do desvio condicional; instruções responsáveis pelo controle de execução de laços; instruções responsáveis pelo controle das chamadas de função e instruções que definem as super-instruções.

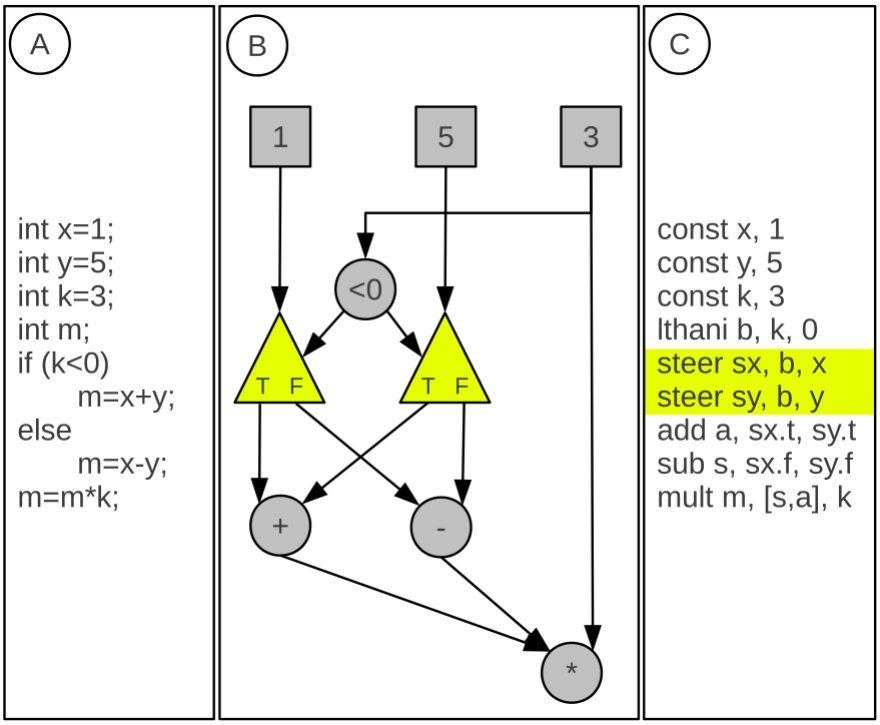

Figura 3.3: Exemplo com as instruções **steer**. O quadro A apresenta um trecho de código em linguagem de alto nível. O quadro B mostra o grafo *dataflow* associado ao trecho com as instruções **steer** representadas por triângulos. O quadro C mostra o código da linguagem de montagem do TALM [4]

Como não existe um contador de programa no modelo *dataflow*, as instruções de desvio condicional e de laços são implementadas, respectivamente, como desvios no fluxo de dados e incrementos nos rótulos de interações dos operandos.

O desvio condicional é implementado pela instrução **steer**, que recebe um operando O e um booleano seletor S. O operando O é encaminhado para um dos possíveis caminhos no grafo de fluxo de dados de acordo com o valor de S. Um exemplo do uso da instrução **steer** pode ser visto na Figura 3.3, onde os triângulos representam as instruções **steer**.

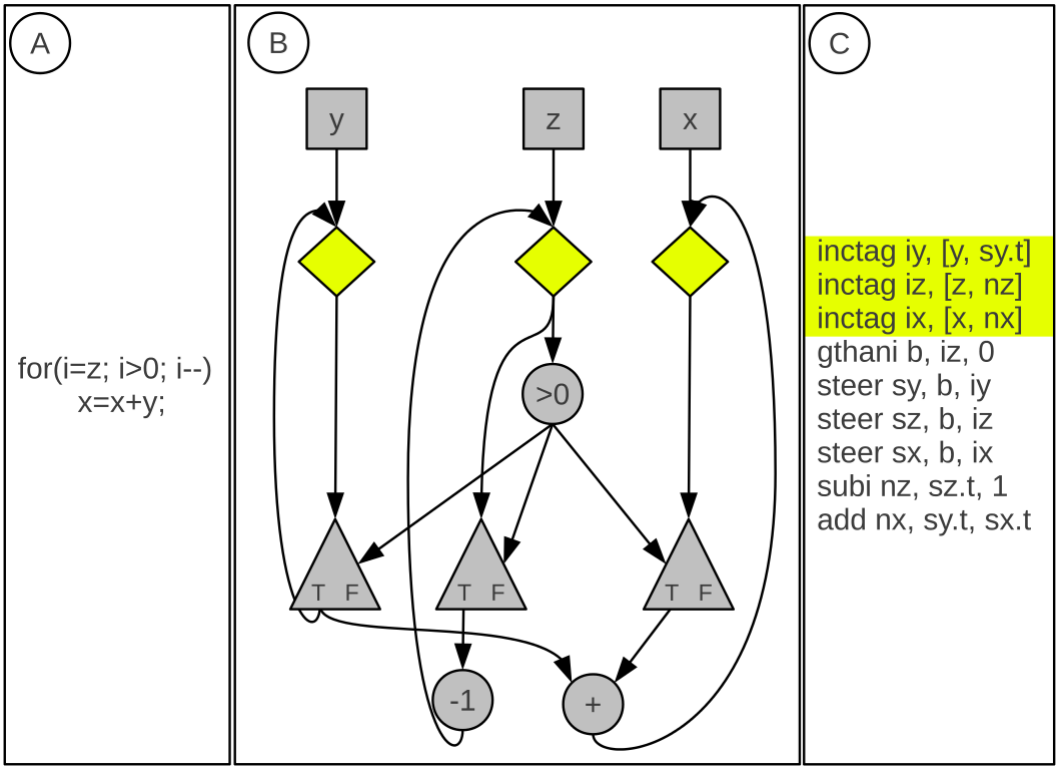

Figura 3.4: Exemplo de laços com as instruções **inctag**. O quadro A apresenta um trecho de código em linguagem de alto nível. O quadro B mostra o grafo *dataflow* associado ao trecho com as instruções **inctag** representadas por losangos. O quadro C mostra o código da linguagem de montagem do TALM. [4]

Para os laços, o TALM adota uma abordagem do modelo de *dataflow* dinâmico. As instruções de um laço são processadas em uma iteração sem precisar esperar que todas as instruções das iterações anteriores finalizem. Para resolver o conflito das iterações, um rótulo é usado para identificar a iteração dos operandos. Quando um operando passa para a próxima iteração, seu rótulo é incrementado e ele se torna válido na nova iteração. A instrução **inctag** é responsável por incrementar o rótulo de um operando. As instruções começarão a executar quando seus operandos de entradas estiverem disponíveis com o mesmo rótulo. A Figura 3.4 mostra como se usa a instrução **inctag** (os losangos representam a instrução **inctag**).

As super-instruções permitem que trechos de código imperativo sejam definidos para executar tarefas mais complexas sem aumentar a complexidade do grafo de fluxo de dados. Elas possibilitam que o programador defina os blocos de instruções que serão executados na arquitetura *dataflow*. Nessa situação, o grafo terá o papel

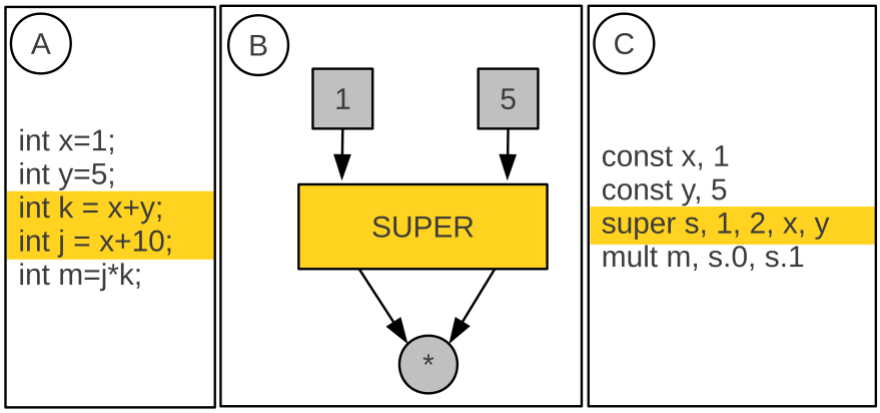

Figura 3.5: Exemplo de super-instrução no TALM. O quadro A apresenta um trecho de código em linguagem de alto nível. O quadro B mostra o grafo *dataflow* associado ao trecho com a instrução super representada por um retângulo maior. O quadro C mostra o código da linguagem de montagem do TALM. [4]

de estabelecer as relações entre as instruções de granularidade grossa para extrair paralelismo com o modelo *dataflow*. A instrução **super** e sua variação **superi** são usadas para definir a sintaxe de uma super-instrução. Na Figura 3.5 é apresentado como é utilizado as instruções **super** e **superi**.

3.2.3 Resumo do conjunto de instruções

Existem mais algumas instruções básicas no modelo. A Tabela 3.1 mostra um resumo das instruções suportadas pela linguagem de montagem do TALM.

Mnemônico	Função	#In	#Out	Imm	Tipo
add	$+$	2	1	não	int
sub	$-$	2	1	não	int
div	$/$	2	1	não	int
mult	$\times$	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	$\leq$	2	1	não	int
geq	$\geq$	2	1	não	int
addi	$+$	2	1	sim	int
subi	$-$	2	1	sim	int
divi	$/$	2	1	sim	int
multi	$\times$	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	$\leq$	2	1	sim	int
geqi	$\geq$	2	1	sim	int
fadd	$+$	2	1	não	float
fsub	$-$	2	1	não	float
fdiv	$/$	2	1	não	float
fmult	$\times$	2	1	não	float

Tabela 3.1: Resumo do conjunto de instruções do TALM. [4]

Capítulo 4

Implementação do TALM

O TALM provê um modelo abstrato de como implementar uma arquitetura *dataflow* genérica, conforme apresentado no Capítulo 3. O modelo define a arquitetura do conjunto de instruções e disponibiliza um conjunto de instruções básicas. A rede definida no modelo pode ser implementada por diferentes recursos: uma rede de computador, um *cluster*, uma área compartilhada ou qualquer outro meio de interconexão desejável. Os elementos de processamento seguem a mesma convenção: pode ser implementado como um processador real, um computador integrante de um sistema distribuído, processos distintos e assim por diante.

Em [4] foi desenvolvida uma implementação do TALM para avaliar o funcionamento do modelo abordando sua eficiência. Um conjunto de ferramentas foi desenvolvido com o intuito de mostrar a facilidade e flexibilidade de se programar com base no modelo. Os resultados dos experimentos com tais ferramentas mostram a capacidade de conseguir explorar técnicas avançadas de programação paralela, tal como pipeline e ocultação da latência de entrada e saída. Além disso, as avaliações mostraram que o TALM alcança melhor desempenho em comparação com os modelos populares de programação paralela como: OpenMP, Pthreads e *Intel Thread Building Blocks* (TBB). E ainda facilita a exposição do ambiente de desenvolvimento em programação paralela de forma eficiente. Os detalhes da avaliação de desempenho e compatibilidade do modelo são discutidos em [4].

Neste capítulo serão apresentados os seguintes tópicos: a linguagem de alto nível THLL, incluindo exemplos de seu uso; o compilador desenvolvido para criar instruções básicas definidas no TALM a partir de arquivo com código fonte expresso em THLL; e a Trebuchet - uma implementação do modelo de execução do *dataflow* baseada num ambiente *multithreaded*.

4.1 THLL

A THLL (*TALM High Level Language*) é uma linguagem de alto nível que estende da linguagem C desenvolvida para facilitar a descrição dos programas destinados ao TALM. Ela possui diretivas para criar super-instruções - trechos que executarão no modelo de Von Neumann - como um nó do grafo *dataflow* definindo seus operandos de entrada e saída, e diretivas para orquestrar a comunicação das tarefas que serão processadas em paralelo. Um programa escrito nessa linguagem é composto basicamente de porções de blocos: trechos de código que recebem argumentos de entrada, trechos que executam uma tarefa e trechos que produzem argumentos de saída. As dependências entre os dados são expressas num modelo com a relação produtor/consumidor através dos argumentos das super-instruções.

O controle do programa pode ser escrito normalmente usando a linguagem C, possibilitando utilizar a sintaxe para operações de *loops* e desvios condicionais. Após desenvolver um programa com a THLL, ele deve ser submetido ao compilador Couillard. O Couillard é responsável por compilar o arquivo fonte e gerar o *assembly* com as instruções básicas fornecidas pelo TALM e o grafo de fluxo de dados do programa. [4]

```
#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 4.1: Exemplo de um programa implementado com a THLL. [4]

A Figura 4.1 apresenta um programa simples implementado com a THLL. A aplicação preenche um vetor de 10000 números inteiros através de um arquivo passado por linha de comando, calcula a soma dos números no vetor e exibe o resultado. O somatório é feito em paralelo, enquanto as etapas de entrada e saída de dados são feitas sequencialmente.

4.2 Couillard

O Couillard é um compilador C que gera as instruções básicas fornecidas pelo TALM a partir de um programa escrito em THLL, conforme apresentado na Seção 4.1. Para tratar o código expresso dentro de uma super-instrução, o compilador produz o código correspondente em um arquivo C. Em seguida os trechos de código são compilados com o compilador padrão da linguagem C (como o gcc) em forma de uma biblioteca de ligação dinâmica. O compilador também gera as dependências das super-instruções definidas pelo programador. Todas as instruções produzidas compõem o grafo de fluxo de dados do programa. Por fim, a Trebuchet, que será apresentada na Seção 4.3, receberá como entrada o grafo de fluxo de dados gerado pelo compilador para controlar sua execução guiada pelas regras *dataflow*.

Além de gerar a linguagem de montagem do TALM, o compilador gera o grafo *dataflow* no formato dot. O arquivo com esse formato pode ser plotado através de uma ferramenta de visualização de grafo chamada Graphviz [21]. Assim, é possível verificar visualmente se o programa foi escrito corretamente através do grafo gerado.

O compilador é composto de duas implementações: o *front-end* - analisador léxico e sintático - e o *back-end* - gerador de código binário. O *front-end* foi desenvolvido em Python com o auxílio da biblioteca PLY (*Python Lex-Yacc*) [22], que oferece uma implementação base para facilitar a construção de um analisador léxico e sintático. A gramática da linguagem suportada pelo compilador é um subconjunto do ANSI-C estendida para reconhecer as diretivas de super-instruções da THLL. A etapa de compilação não se baseia completamente na gramática. Os códigos expressos dentro de super-instruções são exportados em arquivos externos a serem compilados num compilador C tradicional (como o gcc). Após o processamento do *front-end*, é gerada uma AST (*Auxiliary Syntax Tree*) que será processada para produzir o código de representação em forma de grafo de fluxo de dados. [4]

O *back-end* também foi desenvolvido em Python e possui as seguintes atribuições: gerar o código de montagem do TALM, gerar o código das super-instruções e gerar a representação gráfica do grafo no formato dot para ser usado com a ferramenta Graphviz. Após obter a AST gerada pelo *front-end*, o Couillard monta o grafo de fluxo de dados e gera três arquivos de saída:

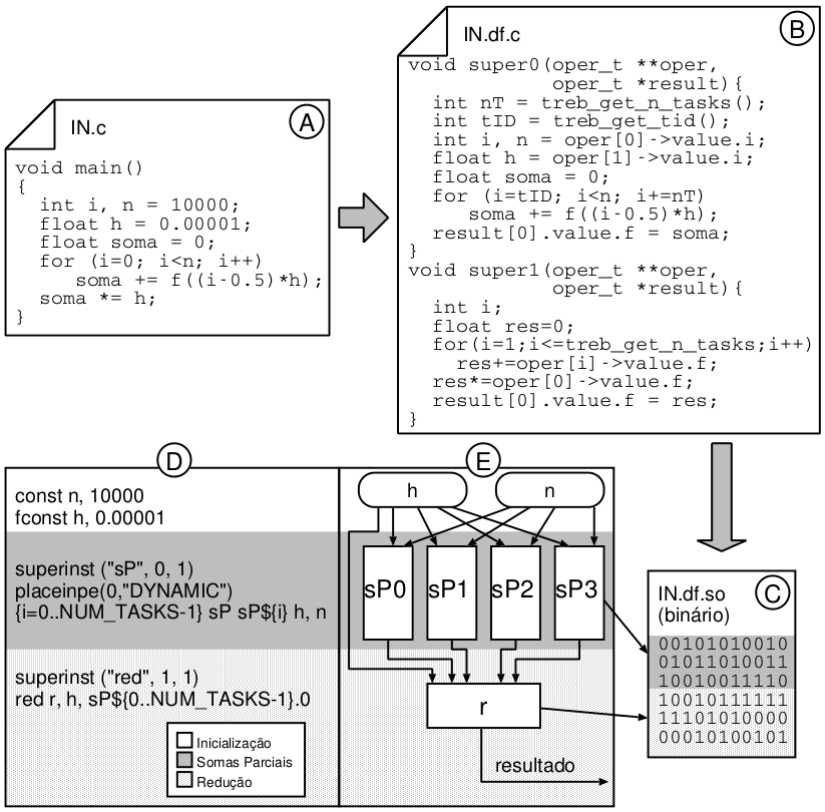

Figura 4.2: Exemplo de arquivos gerados para Trebuchet. O quadro A mostra o código original. O quadro B apresenta o código que descreve o comportamento das super-instruções. O quadro C apresenta o código binário criado pela compilação de B. O quadro D mostra o código da linguagem de montagem do TALM. O quadro E mostra o grafo *dataflow* associado ao código original. [4]

1. O arquivo com a extensão .dot que está no formato da notação do Graphviz que pode ser usado para fins acadêmicos.

2. O arquivo com a extensão .fl que possui a linguagem de montagem correspondente ao grafo *dataflow* representante do programa compilado. A partir desse arquivo é gerado um arquivo com código binário pelo montador com a extensão .flb (que será carregado pela Trebuchet).

3. O arquivo com o término .lib.c que contém as descrições das super-instruções em formas de função, onde todas as variáveis de entrada e saída definidas pelo programador são geradas e iniciadas automaticamente. As instruções do arquivo estão em linguagem C. Esse arquivo é compilado por um compilador tradicional como uma biblioteca de ligação dinâmica.

A Figura 4.2 apresenta um exemplo dos arquivos gerados pelo compilador a serem utilizados na Trebuchet.

4.3 Trebuchet

A Trebuchet é uma máquina virtual que implementa o modelo TALM. Ela executa sobre máquinas *multicore* com seus elementos de processamento sendo *threads* mapeadas em processadores reais. Cada elemento de processamento é associado a um único núcleo do processador, ao invés de ter um grupo de candidatos. O TALM é baseado em um modelo de troca de mensagens para a comunicação entres os EPs. Assim, quando as instruções estão alocadas no mesmo EP, a comunicação para troca de operandos é feita com o acesso direto as estruturas dos operandos. Já em instruções localizadas em EPs distintos, a troca de operandos é feita por meio das rotinas envia() e recebe(). Como a implementação atual da Trebuchet é voltada para processadores *multicore*, o envio e recebimento de operandos é feito através de memória compartilhada. Logo, o custo obtido na comunicação é dado pelas falhas da *cache* mais o custo de obter *locks* necessários para o controle de acesso às regiões compartilhadas. [4]

É importante minimizar o tempo de acesso para a troca de operandos em instruções dependentes quando se usa memória compartilhada. Essas instruções são escalonadas no mesmo EP ou são colocadas em EPs que compartilham a mesma *cache* com o uso de *schedule affinity*. A estrutura de uma instrução na lista de instruções é apresentada na Figura 4.3.

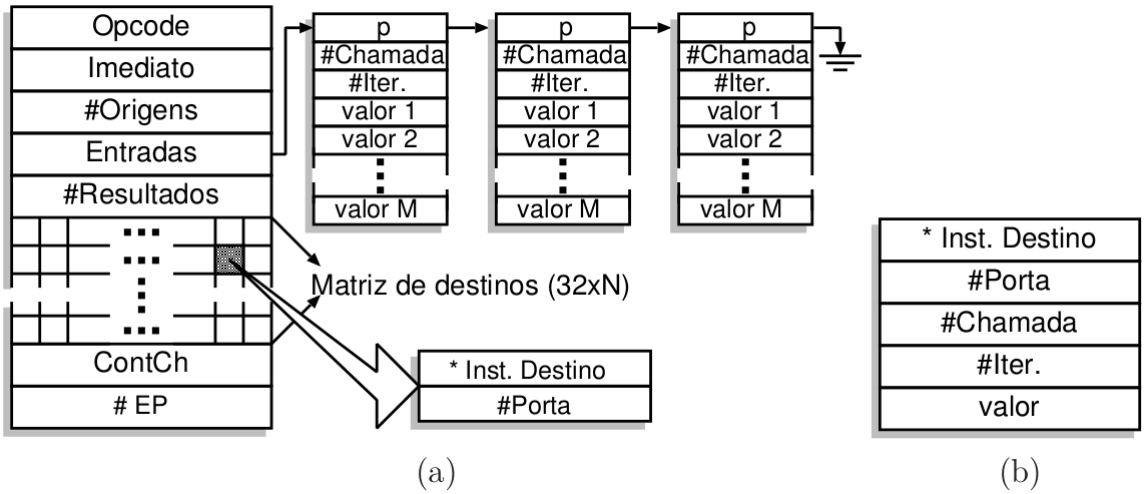

Figura 4.3: Estrutura da Trebuchet. Em (a) mostra a estrutura para armazenamento de operandos e instruções e em (b) mostra a estrutura de uma mensagem com operando. [4]

4.4 Usando a Trebuchet

As seções anteriores apresentaram as ferramentas que apoiam a implementação do modelo TALM. Para uma aplicação sequencial ser submetida ao TALM, é necessário definir o grafo de fluxo de dados por meio da THLL. Em seguida, o arquivo deve ser submetido ao Couillard para produzir a linguagem de montagem de fluxo de dados do TALM e assim poder executar na Trebuchet. Quando um programa é executado na Trebuchet, suas instruções são alocadas aos elementos de processamento. As instruções que não possuem operandos de entrada iniciam a execução do programa seguindo a regra do modelo *dataflow*. E as instruções mapeadas em EPs diferentes executarão em paralelo conforme a disponibilidade dos núcleos do processador real.

As super-instruções criadas pela linguagem THLL são compiladas pelo compilador tradicional e executadas sequencialmente como um programa em C. O disparo da execução desses blocos é feito pela Trebuchet com base nas regras de execução do *dataflow*. A Figura 4.4 apresenta os passos necessários para paralelizar e executar uma aplicação sequencial com a Trebuchet em conjunto com as ferramentas apresentadas. Primeiro é definido o programa sequencial com os blocos de super-instruções. O Couillard é usado para extrair o código das super-instruções definidas pelo programador e transformar em funções, que receberão operandos de entradas e produzirão operandos de saída. Após isso, o arquivo gerado com os blocos transformados é compilado para uma biblioteca dinâmica que é disponibilizada para o interpretador da Trebuchet. Além disso, o compilador gera o arquivo com a linguagem de montagem do grafo de fluxo de dados do programa que pode conter instruções simples e as super-instruções. Por fim, é gerado o binário do arquivo de montagem junto com o arquivo que define as alocações das instruções. [4]

A Trebuchet necessita das seguintes entradas para processar um programa: o código binário correspondente o arquivo de montagem, a biblioteca dinâmica contendo o que será executado pelas super-instruções e um arquivo de alocação de instruções.

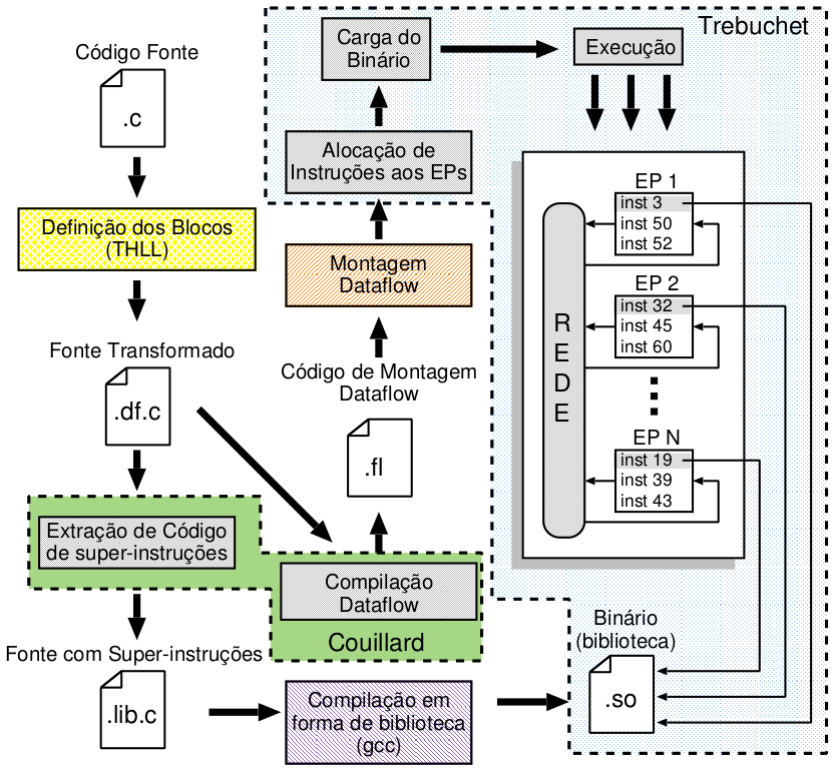

Figura 4.4: Fluxo de trabalho da Trebuchet. [4]

Capítulo 5

Técnicas de Otimização

O Capítulo 4 apresentou um conjunto de ferramentas para a implementação da arquitetura TALM. Essas ferramentas permitem que o programador codifique um programa definindo o nível de granularidade das tarefas a serem executadas em paralelo guiadas pelas dependências dos dados.

A Seção 2.5 discute a área de *dataflow* híbrido mostrando um resultado importante: a abordagem de granularidade média obtém o melhor desempenho na exploração de paralelismo das aplicações. Entretanto, a definição de granularidade média não é trivial. O melhor nível médio de granularidade depende da aplicação.

No modelo de *dataflow* dinâmico, os *loops* são as principais fontes de paralelismo na execução de um programa. No entanto, esse modelo tem um efeito indesejável: cria instruções de controle desnecessárias para os operandos produzidos antes de um *loop* e utilizados somente após o *loop*, para evitar o conflito no casamento dos rótulos desses operandos com os rótulos dos operandos produzidos na última iteração do *loop*. Esse efeito pode resultar em *overheads* significativos com muitos operandos extras sendo processados, especialmente em *loops* com muitas iterações. Outro problema decorrente é a limitação da exploração de granularidade fina nas aplicações destinadas ao TALM, para evitar tal *overhead*.

Para tratar desse problema e possibilitar afinar o grão de tarefas de forma que alcance o nível de granularidade ideal, esse trabalho propõe um conjunto de técnicas de otimização de laços em *dataflow*. Neste capítulo seram abordados o problema com laços e as técnicas de otimização com suas modificações no TALM.

5.1 O problema do *loop*

Na Seção 3.2.2 foram apresentadas as instruções nativas fornecidas pelo TALM. Para usar laços, é necessário usar as instruções de controle **Steer** e **IncTag**. A instrução **Steer** envia um operando O para uma das duas possíveis portas de saída de acordo com um booleano seletor S. A instrução **IncTag** incrementa o rótulo de

um operando para diferenciar suas instâncias e garantir que somente operandos com o mesmo rótulo poderão ser correlacionados (casados).

Quando um operando produzido antes de um *loop* é processado somente por instruções que executam após o término do *loop*, ele é forçado a passar por dentro do *loop* para ter seu rótulo atualizado de acordo com os operandos produzidos na última iteração do laço e assim poder casar corretamente com esses operandos. Esse operando é chamado de *skip variable*.

A Figura 5.1 e a Figura 5.2 exemplificam como o uso de um rótulo global por programa pode elevar o *overhead* na execução de laços.

A Figura 5.1 mostra como um *loop* simples é implementado no TALM. Cada operando precisa de uma instrução **IncTag** e uma instrução **Steer** para direcionar seu destino dentro do *loop*. Se a condição do *loop* não for válida, os operandos são direcionados para fora do *loop* pela instrução **Steer**. As *skip variables* (operandos que não precisam ser processados no *loop*) precisam passar através do laço, e são obrigatoriamente direcionadas as instruções de controle **Steer** e **IncTag**. No exemplo apresentado na Figura 5.1, os operandos a e b foram produzidos antes do *loop* e serão usados após ele na operação que calcula o valor de m.

A Figura 5.2 mostra como *loops* aninhados é implementado no TALM. Note que com a introdução de laços internos, o controle pode ser muito custoso. No exemplo, os operandos a e x foram produzidos antes dos *loops*. O operando a é utilizado no processamento do valor de m que se encontra após os laços aninhados, e o operando x é atualizado somente no *loop* externo. Isto significa que instruções de controles extras serão inseridas no *loop* interno para ajustar o rótulo de x e cada instrução de controle será executada 15 vezes (3×5). Por outro lado, a variável a, que é uma *skip variable* para ambos os *loops*, também vai precisar de instruções de controle extras. Neste caso, as instruções de controle extras no *loop* externo serão executadas 5 vezes, enquanto as do *loop* interno serão executadas 15 vezes.

Quanto maior a quantidade de *skip variables* e maior a profundidade do aninhamento de laços, maior será o desperdício de processamento e, consequentemente, maior o *overhead*. Uma forma de argumentar sobre a solução do problema de *skip variables* é mover as instruções que produzem aqueles valores para algum local depois do *loop*. Essa seria uma possível solução, mas não cobriria todos os casos. Por exemplo, as instruções que produzem múltiplos resultados, que é o caso da maioria das super-instruções, podem produzir operandos que serão consumidos no laço, enquanto outros valores serão usados após o laço. Neste caso, onde somente alguns resultados são considerados *skip variables*, não é simples mover as instruções para algum lugar depois do *loop*.

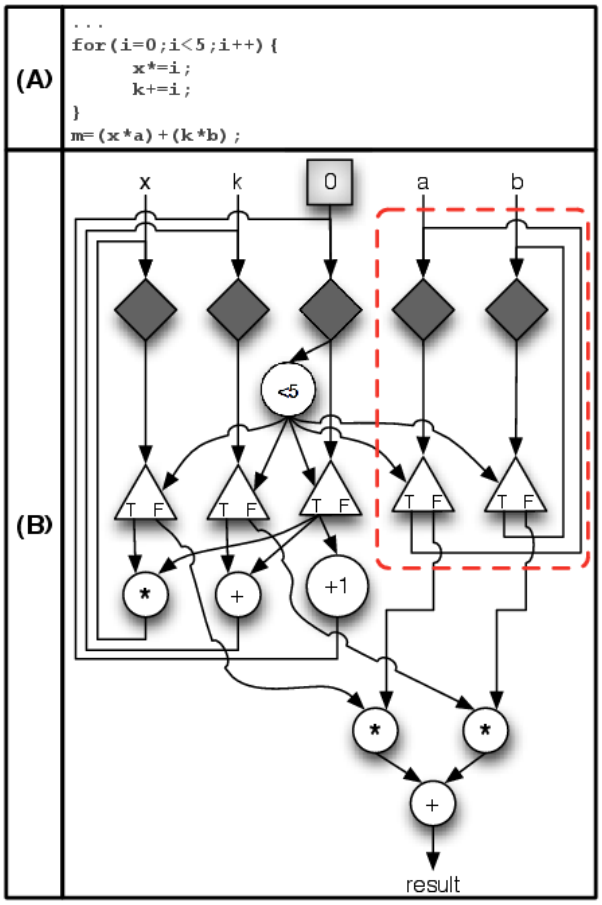

Figura 5.1: Exemplo de problema com *loop* simples em *dataflow*. O quadro A mostra um trecho de código com *loop* simples em linguagem de alto nível. O quadro B mostra o grafo *dataflow* associado ao trecho. Os losangos representam instruções **IncTag** e os triângulos representam as instruções **Steer**.

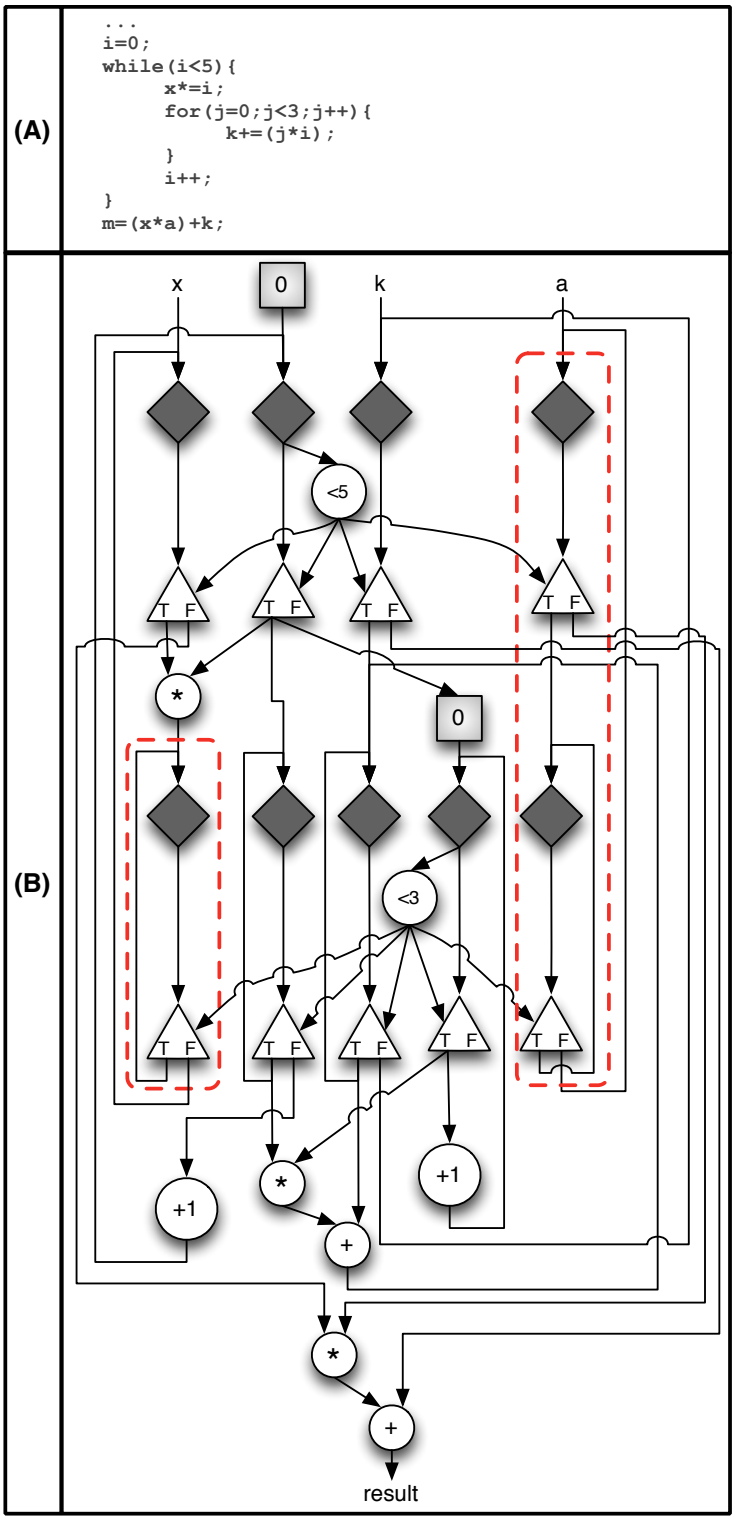

Figura 5.2: Exemplo de problemas com *loops* aninhados em *dataflow*. O quadro A mostra um trecho de código com *loops* aninhados em linguagem de alto nível. O quadro B mostra o grafo *dataflow* associado ao trecho. Os losangos representam instruções **IncTag** e os triângulos representam as instruções **Steer**.

5.2 *Stack-Tagged Dataflow*

5.2.1 Abordagem

Stack-Tagged Dataflow [23] é um mecanismo de rotulação que utiliza pilhas de inteiros para armazenar rótulos de diferentes contextos de blocos de instruções, ao invés de simples valores escalares. As pilhas de rótulos foram inspiradas nas estruturas de pilhas usadas para armazenar informações de contextos relacionadas a chamadas de funções em programas executados em máquinas de Von Neumann, onde existe uma implementação de *hardware* eficiente para manipulação de pilha com uso de registradores como *stack pointer*. A ideia chave dessa técnica é permitir a criação de escopos individuais para os rótulos em cada nível de *loop* em aninhamento de laços. Cada operando possui uma pilha de inteiros no lugar de um simples inteiro. Cada elemento da pilha representa o rótulo de iteração relacionado a diferentes *loops* aninhados. Esta técnica facilita a geração de código *dataflow* no compilador em linguagens imperativas. A compilação de laços no *dataflow* tradicional é uma tarefa árdua, pois o compilador precisa encontrar todos os operandos que deverão ser forçados a passar através do laço, mesmo que eles não possuam dependências no laço. Esses operandos são as *skip variables*.

Nesta técnica, cada operando enviado para um *loop* passa primeiro por uma instrução de **Push**, inserida antes de sua instrução **IncTag** correspondente. As instruções **Push** inserem um novo rótulo zerado no topo da pilha para ele ser usado durante o processamento do laço, enquanto as instruções **IncTag** incrementarão somente os rótulos que estão no topo da pilha de cada operando. Além disso, uma instrução **Pop** deve ser adicionada na porta de saída F (*False*) de cada instrução **Steer** responsável pelo controle do fluxo de operandos no *loop*. As instruções **Pop** desempilharão e discartarão o topo da pilha, restaurando a pilha de rótulos com seu valor original. Ao fim da execução do laço, os operandos são direcionados para uma instrução de **Pop**. Assim, as *skip variables* poderão casar corretamente com os operandos produzidos na última iteração do laço sem executar desnecessariamente as instruções **IncTag** e **Steer**.

A Figura 5.3 mostra como a pilha de rótulos pode ser usada para reduzir o *overhead* no controle de um *loop* simples. Note como as instruções de **Push** e **Pop** evitam as instruções **IncTag** e **Steer** desnecessárias para os operandos a e b. No exemplo há 3 instruções **Push** e 2 **Pop**. Isso ocorre, porque um dos operandos utilizados no *loop* não é processado após seu término. Portanto, ele pode ser descartado não havendo necessidade de ser enviado para uma instrução **Pop**.

A Figura 5.4 mostra como a pilha de rótulos pode reduzir o *overhead* de controle em *loops* aninhados. Note como as instruções de **Push** e **Pop** evitam as instruções **IncTag** e **Steer** desnecessárias para ambos os *loops* interno e externo para o ope-

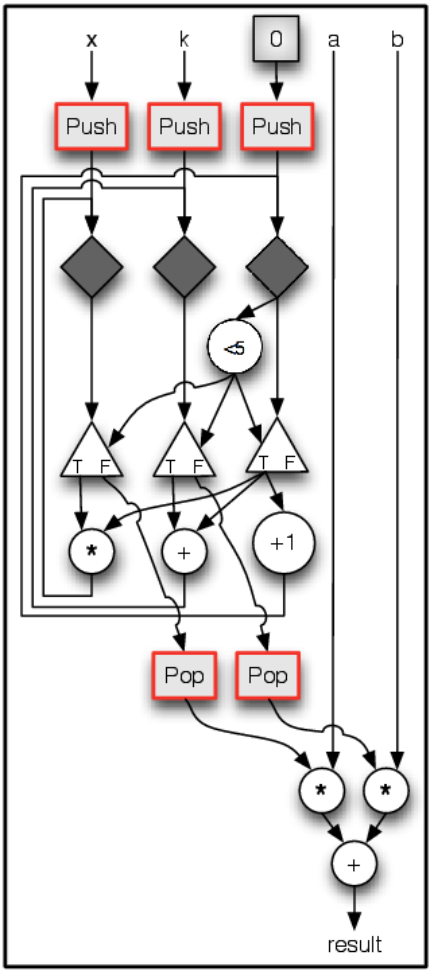

Figura 5.3: Exemplo de redução de *overhead* em *loop* simples com rótulos de pilha. A figura mostra o grafo referente a Figura 5.1 com instruções de pilha. Note como é evitado o uso desnecessário de instruções **Steer** e **IncTag** para os operandos a e b.

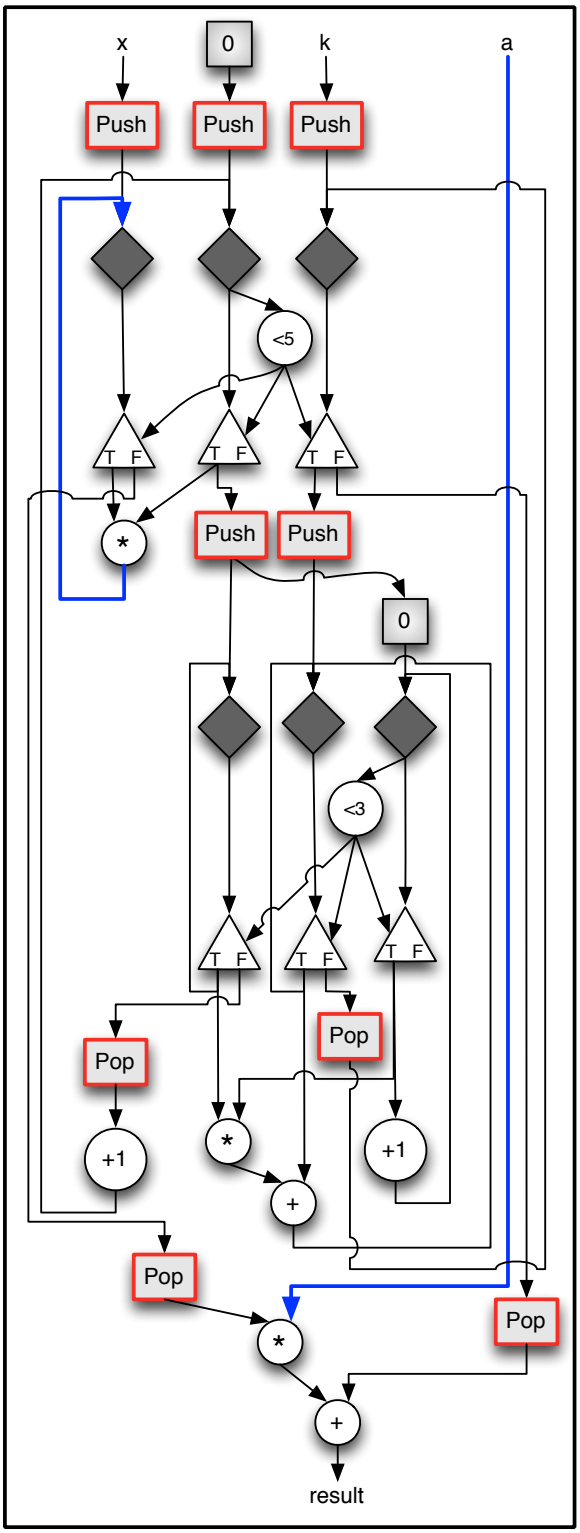

Figura 5.4: Exemplo de redução de *overhead* em *loops* aninhados com rótulos de pilha. A figura mostra o grafo referente a Figura 5.2 com instruções de pilha. Note que são evitadas instruções de **Steer** e **IncTag** para os *loops* internos e externos do operando a. O mesmo acontece com o operando x, que não precisa das instruções no *loop* interno.

rando a. O mesmo pode ser observado para operando x, que não é necessário no *loop* interno.

5.2.2 Vantagens e Desvantagens

Assim como foi discutido na Seção 5.2.1, a principal vantagem da pilha nos rótulos é permitir escopos individuais de rótulos para instruções de laços. Cada inteiro armazenado na pilha corresponde ao rótulo de iteração de diferentes *loops* aninhados.

O uso de pilha nos rótulos tem um outro efeito positivo: reduzir a chance de ocasionar *overflow* no rótulo. Esse caso ocorre, quando o tamanho do rótulo não é grande o suficiente para armazenar o número total de iterações executadas no programa. No *dataflow* tradicional, os rótulos são inteiros sem sinal incrementados a cada iteração. Para os programas que possuem múltiplos *loops* não aninhados, o valor máximo do rótulo será dado pela soma do número de iteração de todos os *loops*. Em programas com *loops* aninhados, o valor máximo do rótulo será dado pelo produto do número de iteração de cada *loop*. Já que essa técnica separa os escopos de cada laço, o limite de iteração de um *loop* será determinado unicamente pelo tamanho do elemento da pilha.

Em máquinas *dataflow* reais (*hardware*), tal como o WaveScalar [8], o *overflow* do rótulo é resolvido pela limitação da ocorrência de iterações ativas no *loop*, de acordo com a capacidade do rótulo. Isso é feito usando a técnica *k-loop bounding* para implementar janela deslizante. Por exemplo, uma máquina com rótulo de tamanho de 4-bits permite execução concorrente de 16 iterações de um laço. Conforme as iterações vão sendo concluídas, novas serão autorizadas a iniciarem sua execução. Neste caso, o *overflow* do rótulo não será um problema, já que somente 16 iterações poderão executar depois que a iteração 0 terminar e as novas execuções também compartilharão o rótulo 0. Além disso, as máquinas *dataflow* reais têm diversas limitações rígidas no armazenamento do operando. Portanto, não há como permitir muitas iterações se não houver espaço de armazenamento suficiente para guardar os operandos não consumidos. [17]

A implementação de janela deslizante não foi implementada na máquina virtual Trebuchet, por causa do custo extra de interpretação e a possível redução na exploração de paralelismo. Além do mais, detectar a terminação de uma iteração do *loop* em *dataflow* pode ser custoso e precisar de sincronização de instruções, assim como acontece com a instrução canônica *Wave-Advance* no WaveScalar. Quando parte de uma iteração do *loop* executa mais rápido que as demais, pode ocorrer explosão de paralelismo degradando o desempenho da aplicação e aumentando drasticamente o uso de memória. Neste caso, o mecanismo *k-loop bounding* pode ser incluído diretamente no grafo, mas esse não foi o foco desse trabalho.

O *Stack-Tagged Dataflow* também tem suas desvantagens. Primeiro, o processo de casamento de operandos é potencialmente mais custoso. Antes, a comparação era feita com dois inteiros e agora é necessário comparar todos os elementos das pilhas. Essa restrição tem que ser cumprida para garantir que os contextos dos operandos casados sejam iguais, respeitando até mesmo os contextos dos *loops* aninhados. Por exemplo, considere um programa com dois loops aninhados e que (i, j) é o rótulo de pilha de um operando executando na i-ésima iteração do *loop* externo e j-ésima iteração do *loop* interno. Se somente o topo da pilha for usado para o casamento de operandos, um operando rotulado com $(1, 5)$ poderá casar com outro operando rotulado com $(2, 5)$ e produzir resultados errados. Segundo, o acréscimo das instruções **Push** e **Pop** no início e fim de cada laço pode aumentar o caminho crítico de execução do programa, se o *loop* é o próprio caminho crítico. Esse efeito é maximizado em *loops* aninhados, pois as instruções **Push** e **Pop** são executadas em cada instância do *loop* interno. Por exemplo, na Figura 5.4 as 3 instruções **Push** e 3 instruções **Pop** no *loop* externo executarão somente uma vez, enquanto as 2 instruções **Push** e **Pop** no *loop* interno executarão 5 vezes, visto que o *loop* externo tem 5 iterações.

Para reduzir a complexidade da comparação entre pilhas, a primeira solução verifica se ambas pilhas tem o mesmo tamanho. Se o tamanho for diferente, então a desigualdade já é retornada. Caso contrário, os elementos das pilhas são comparados até que se encontre alguma diferença (falha) ou que se chegue ao final da pilha (casado). Além disso, para evitar custo de realocação de memória, o número de elementos alocados na pilha é fixo e definido em tempo de compilação do ambiente de execução da Trebuchet. Isso limita o número de *loops* aninhados, mas não é um problema, pois as aplicações dificilmente possuem mais do que 4 *loops* aninhados.

Stack-Tagged Dataflow tem o potencial de reduzir o *overhead* no controle de *loops*, impulsionando a exploração de paralelismo. Os operandos não necessários em um *loop* serão diretamente enviados aos seus destinos e poderão disparar a execução de suas instruções com menos atraso.

5.2.3 Modificações no TALM

Para incluir o suporte ao *Stack-Tagged Dataflow* no conjunto de ferramentas da Trebuchet foram feitas as seguintes mudanças:

- Fazer o compilador Couillard usar instruções **Push** e **Pop** na geração do arquivo com a linguagem de montagem do TALM.
- Incluir as instruções **Push** e **Pop** no montador do TALM.

- Implementar a lógica de execução do **Push** e **Pop** na máquina virtual Trebuchet.
- Modificar o processo de casamento de operandos na Trebuchet para verificar todos os elementos da pilha.

A sintaxe da instrução `push` é a seguinte:

```
push <nome>, <operando>
```

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

- `push`: é o mnemônico da instrução.
- `nome`: é o identificador da instância da instrução usado para referenciar o operando de saída.
- `operando`: indica o operando de entrada no qual o rótulo de iteração será empilhado com valor zerado.

A sintaxe da instrução `pop` é a seguinte:

```
pop <nome>, <operando>
```

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

- `pop`: é o mnemônico da instrução.
- `nome`: é o identificador da instância da instrução usado para referenciar o operando de saída.
- `operando`: indica o operando de entrada no qual o rótulo de iteração terá o topo da pilha descartado e restaurado para o valor do escopo anterior da execução do laço.

5.2.4 Otimização

A pilha de rótulos implementada para cada operando possui um tamanho fixo. Para otimizar o casamento de pilha, foi criada uma macro que injeta a lógica de comparação de todos os elementos da pilha em uma única operação condicional, ao invés de utilizar um laço para comparar todos os elementos inseridos conforme explicado na Seção 5.2.2. Inicialmente, todas as posições da pilha são inicializadas com o rótulo zero para que possa ocorrer o casamento correto se a pilha conter menos elementos que sua capacidade máxima. A instrução **Pop** foi modificada para zerar o rótulo do topo antes de desempilhar para manter a consistência do casamento.

5.2.5 Trebuchet Híbrida

A Trebuchet híbrida é uma versão da máquina virtual onde é decidido em tempo de execução qual o processo de casamento que será aplicado nos operandos. Inicialmente, o casamento ocorre com os números inteiros. A partir da execução da primeira instrução de **Push**, o processo de casamento é modificado para o casamento de pilha e só mudará para casamento com inteiros ao final de um *loop* externo numa instrução de **Pop**. Isso é feito, com o acréscimo de um ponteiro indicando a função do casamento em vigor no rótulo de cada operando durante a execução. Além do novo ponteiro, cada operando possui os dois tipos de rótulo (inteiro e pilha). No ínicio da execução, um contador é inicializado com zero. Cada instrução **Push** incrementa o contador e cada instrução **Pop** o decrementa. Em cada execução da instrução de **Push**, o ponteiro de função é modificado para função de casamento de pilha. Ao decrementar e zerar o contador, o ponteiro de função é modificado para função de casamento de inteiro. A instrução **IncTag** mantém a mesma lógica do casamento de pilha, incrementando sempre o topo da pilha.

5.3 *Tag Resetting*

5.3.1 Abordagem

Assim como foi explicado nas seções anteriores, os *loops* aninhados podem aumentar potencialmente o *overhead* do casamento de operandos. Por isso, as soluções que minimizam a profundidade da pilha são desejáveis. *Tag Resetting* fornece um meio de resetar o rótulo sempre que o operando alcançar um local seguro. Um ponto seguro para resetar é a saída de um *loop* externo, no qual define uma nova fase de computação.

O *Tag Resetting* garante redução de um nível de profundidade da pilha de maneira simples e elegante. A técnica não precisa de suporte do ambiente em tempo de execução, mas precisa ter a implementação da instrução para zerar o rótulo. Ao fim da execução do *loop* externo, uma instrução **zTag** deve ser adicionada na porta de saída F (*False*) de cada instrução **Steer** responsável pelo controle do fluxo de operandos no *loop*. O rótulo do operando receberá o valor 0 e, assim, poderá casar com as *skip variables*.

A Figura 5.5 mostra como as instruções **zTag** podem ser usadas para resetar o rótulo de *loops* externos. Note como as instruções **zTag** evitam as instruções de pilha, **Steer** e **IncTag** desnecessárias em ambos *loops* interno e externo. Visto que o operando x é necessário dentro do *loop* externo, o *Tag Resetting* não pode ser usado para eliminar as instruções **IncTag** e **Steer** relacionados ao x, pois não se trata de um ponto seguro.

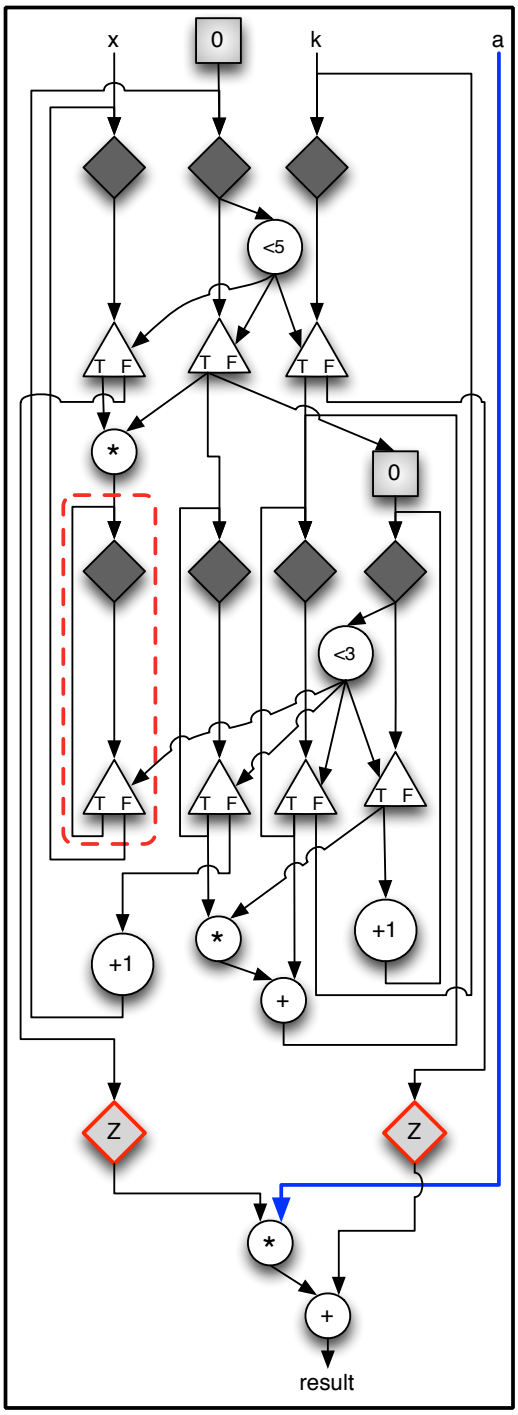

Figura 5.5: Exemplo de redução de *overhead* em *loops* aninhados com o *Tag Resetting*. A figura mostra o grafo referente a Figura 5.2 com instruções **zTag** representadas por losangos vermelhos marcados com Z. Note que são evitadas as instruções de **Push** e **Pop** e o operando a pode ser enviado diretamente ao seu destino. O operando x é obrigado a passar pelo *loop* interno.

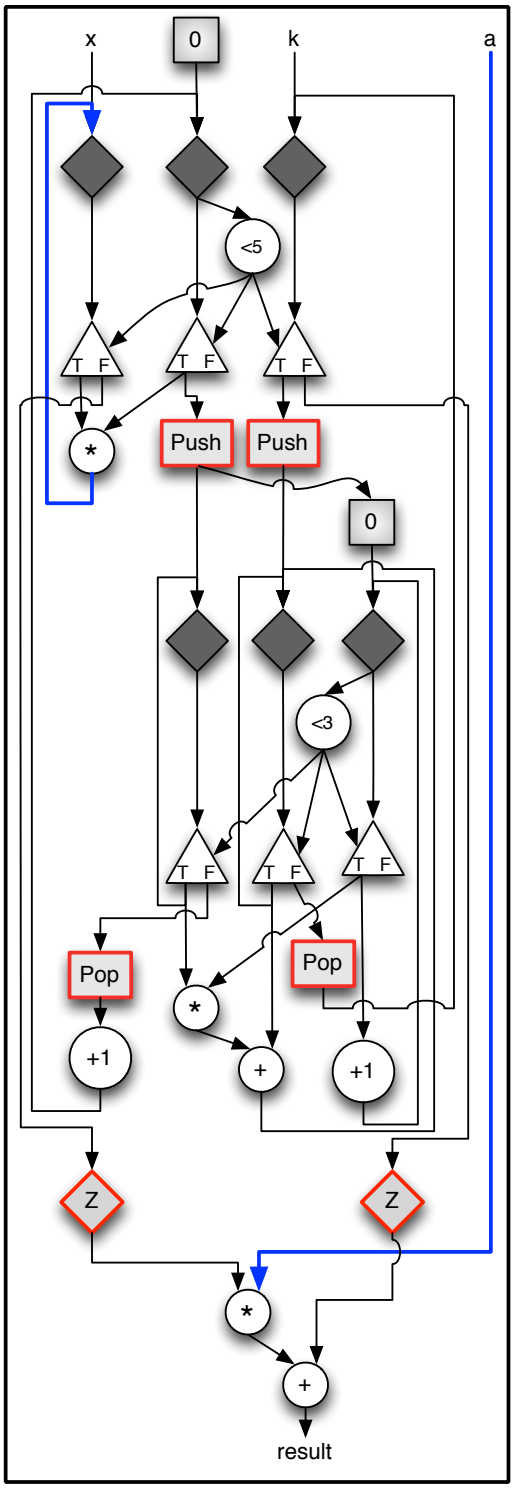

Figura 5.6: Exemplo de redução de *overhead* em *loops* aninhados com o *Tag Resetting* combinado com a pilha de rótulos. A figura mostra o grafo referente a Figura 5.2 com instruções de pilha e **zTag** (representadas por losangos vermelhos marcados com Z). Note que são evitadas as instruções de **Push** e **Pop** no *loop* externo. O operando a pode ser enviado diretamente ao seu destino. O operando x não precisa das instruções no *loop* interno.

A Figura 5.6 mostra como as instruções **Push** e **Pop** podem ser usadas para evitar instruções **IncTag** e **Steer** para o operando x, enquanto as instruções **zTag** fazem o mesmo para o operando a.

5.3.2 Vantagens e Desvantagens

A principal vantagem do *Tag Resetting* é reduzir um nível da pilha de rótulos evitando o uso de duas instruções por operando: um **Push** e um **Pop**. Quanto menor for a pilha no rótulo, mais rápido será o casamento de operandos e melhor o desempenho total da aplicação. Além do mais, a técnica também reduz a chance de ocorrer *overflow* no operando, apesar de não ser tão eficiente como a pilha de rótulos que pode reduzir o *overflow* em *loops* aninhados.

O *Tag Resetting* é aplicado somente nos *loops* externos. Os rótulos não podem ser resetados dentro de *loops* aninhados para evitar o conflito no casamento de operandos. Um *loop* externo executa múltiplas vezes o *loop* interno e por isso é necessário manter um rótulo > 0 que identifique sua instância. Neste caso, as instruções **Push** e **Pop** podem ser utilizadas nos *loops* internos. É importante mencionar que resetar o rótulo dentro de uma função não afetará o chamador, desde que haja uma estrutura de rótulo separadas para identificar as diferentes chamadas da mesma função. Portanto, o *Tag Resetting* pode ser empregado múltiplas vezes: uma para cada chamada de função. Mais informações sobre rótulos de funções na Trebuchet pode ser encontradas em [24].

Outra desvantagem do *Tag Resetting* é a adição de instruções no *loop* que, assim como o *Stack-Tagged Dataflow*, pode aumentar o caminho crítico da aplicação.

5.3.3 Modificações no TALM

Para incluir o suporte ao *Tag Resetting* no conjunto de ferramentas da Trebuchet foram feitas as seguintes mudanças:

- Fazer o compilador Couillard usar a instrução **zTag** na geração do arquivo com a linguagem de montagem do TALM.
- Adicionar a instrução **zTag** no montador do TALM.
- Implementar a lógica de execução do **zTag** na máquina virtual Trebuchet.

A sintaxe da instrução `ztag` é a seguinte:

```
ztag <nome>, <operando>
```

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

- `ztag`: é o mnemônico da instrução.
- `nome`: é o identificador da instância da instrução usado para referenciar o operando de saída.
- `operando`: indica o operando de entrada no qual o rótulo de iteração será zerado.

5.4 *Loop Skipping*

5.4.1 Abordagem

É possível evitar ainda mais o *overhead* na comparação de pilha em *loops*, quando o número de iterações pode ser determinado pelo compilador. Neste caso, as instruções **Push** e **Pop** podem ser eliminadas e os rótulos das *skip variables* podem ser incrementados pelo o número de iterações obtido em tempo de compilação. Desta forma, os rótulos das *skip variables* casarão com os operandos produzidos por um *loop*.

Este trabalho vai abordar alguns casos onde é possível determinar o número de iterações de um *loop*. Apenas os *loops* com o comando `for` serão considerados, mas alguns casos é possível empregar o mesmo raciocíno com os *loops* com o comando `while`. Além disso, a técnica vai ser utilizada apenas para *loops* com o seguinte padrão:

```
for ( x=f ; x RELATIONAL_OP l ; x STEP_OP [s])
```

Onde:

- **x**: é a variável de controle do *loop*;
- **f**: (inteiro) é o primeiro valor de **x**;
- **RELATIONAL_OP**: é o operador relacional ($<$, $<=$, $>$ ou $>=$ neste trabalho);
- **l**: (inteiro) é o último valor de **x**;
- **STEP_OP**: é o operador que modifica x em cada passo do *loop* ($++$, $--$, $+=$ ou $-=$ neste trabalho);
- **s**: (inteiro > 0) é o passo a ser aplicado (no caso $+=$ ou $-=$ é usado);

Se x, f, l e s não são modificados dentro do *loop* e o controle da execução está dentro do *loop*, ou seja, não possui comandos *break*, *continue* ou *return*, então o número de iterações pode ser calculado com uma fórmula simples. Por exemplo, considere o seguinte *loop*:

```
for ( x=f ; x < l ; x += s)
```

Neste caso, o número de iterações ($N_{\text{iterations}}$) será dado por $N_{\text{iterations}} = \lfloor (max((l - f), 0) + s - 1)/s \rfloor$. Essa fórmula varia conforme a relação dos operadores relacionais e dos passos utilizados no laço. Para eliminar as instruções de **Steer** e **IncTag** nas *skip variables*, tudo que é necessário a ser feito é incrementar o rótulo delas por $N_{\text{iterations}} + 1$, já que haverá uma iteração extra para determinar se o *loop* terminou (o corpo do *loop* não será executado).

Se f, l e s são constantes, o compilador pode aplicar a fórmula mencionada anteriormente e usar os resultado nas instruções **IncTagI**. A instrução **IncTagI** é responsável por incrementar o rótulo de um operando por um imediato. Desse modo, as *skip variables* são enviadas para as instruções **IncTagI**, que incrementarão seus rótulos por $N_{\text{iterations}} + 1$ ($N_{\text{iterations}} + 1$ é o imediato do **IncTagI**).

A Figura 5.7 mostra como as instruções **IncTagI** podem ser usadas para colocar os rótulos nas *skip variables* adequadamente. Note como as instruções **IncTag** e **Steer** para os operandos a e x podem ser evitados no *loop* interno. O *Loop Skipping* não é usado no *loop* externo porque os *loops* com comando `while` não são suportados nessa implementação. A Figura 5.8 mostra como a combinação das instruções **Push** e **Pop** com **IncTagI** podem ser usadas. A Figura 5.9 mostra como a combinação das instruções **IncTagI** e **zTag** podem evitar instruções de pilhas e fazer as correções nos rótulos adequadamente.

Se f, l e s são variáveis, o compilador vai gerar o código para calcular a fórmula em tempo de execução e enviar os valores produzidos para as instruções **IncTagN**. A instrução **IncTagN** é responsável por incrementar o rótulo de um operando por um operando de entrada. Assim, as *skip variables* são enviadas para as instruções **IncTagN**, que incrementarão seus rótulos por um operando que terá o resultado da fórmula executada.

É necessário estabelecer algumas condições para determinar o valor correto da fórmula empregada em tempo de execução. A primeira restrição é que as variáveis f, l e s sejam produzidas antes do *loop*. A segunda condição restringe a utilização da técnica para *loops* aninhados: todas as variáveis f, l e s correspondentes a cada *loop* devem ser produzidas antes do laço mais externo em que se pode aplicar a técnica. Essa última restrição tem que ser estabelecida pois se uma *skip variable* precisa ultrapassar dois *loops* aninhados e as variáveis f, l e s do *loop* interno são definidas dentro do *loop* externo, então torna-se inviável prever de maneira simples

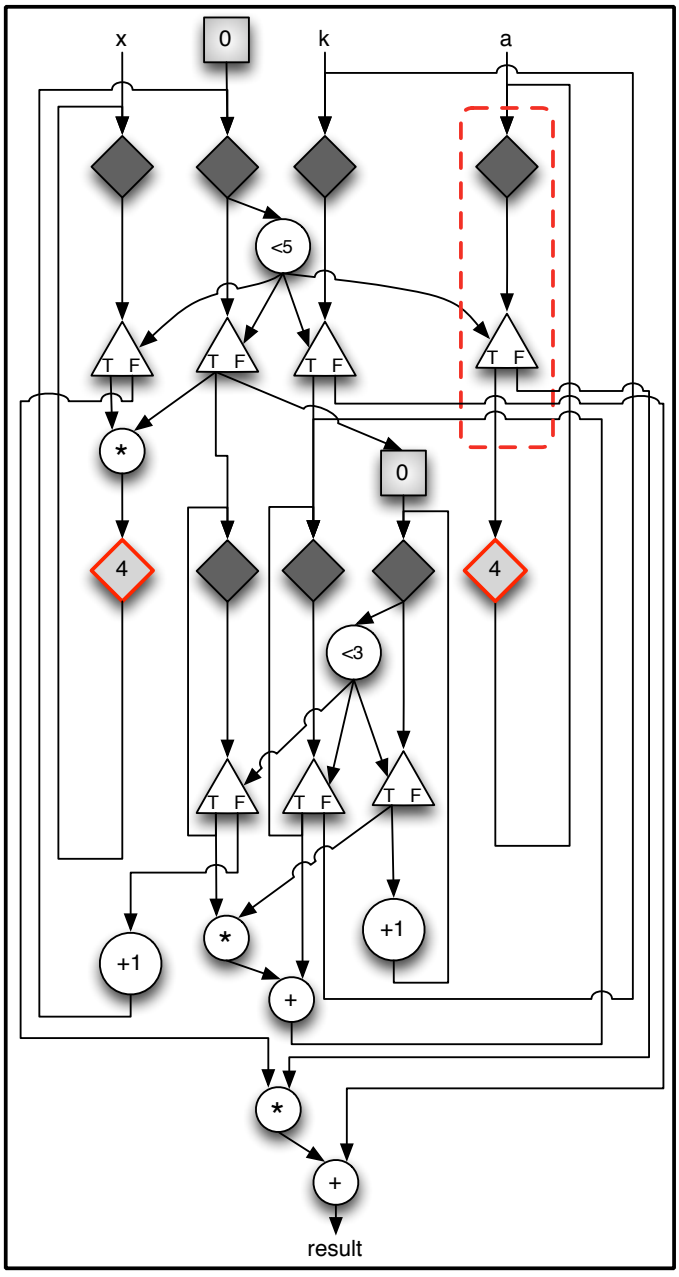

Figura 5.7: Exemplo de redução de *overhead* em *loops* aninhados com o *Loop Skipping*. A figura mostra o grafo referente a Figura 5.2 com instruções **IncTagI** representadas por losangos vermelhos contendo o imediato. Note que são evitadas as instruções de **Steer** e **IncTag** no *loop* interno para os operandos a e x. O *Loop Skipping* não foi usado no *loop* externo porque o comando com `while` não é suportado nessa implementação.

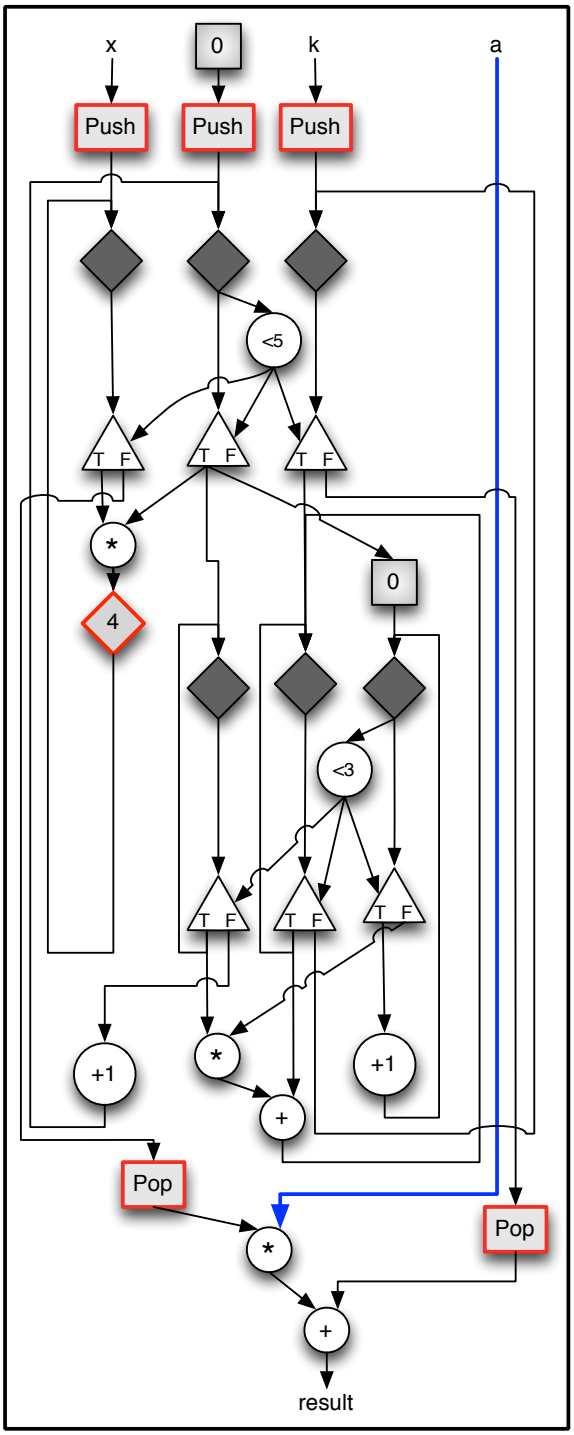

Figura 5.8: Exemplo de redução de *overhead* em *loops* aninhados com o *Loop Skipping* combinado com a pilha de rótulos. A figura mostra o grafo referente a Figura 5.2 com instruções de pilha e **IncTagI** (representadas por losangos vermelhos contendo o imediato). Note que são evitadas as instruções de **Push** e **Pop** no *loop* interno. O operando a pode ser enviado diretamente ao seu destino. O operando x precisa somente de uma instrução **IncTagI** no *loop* interno.

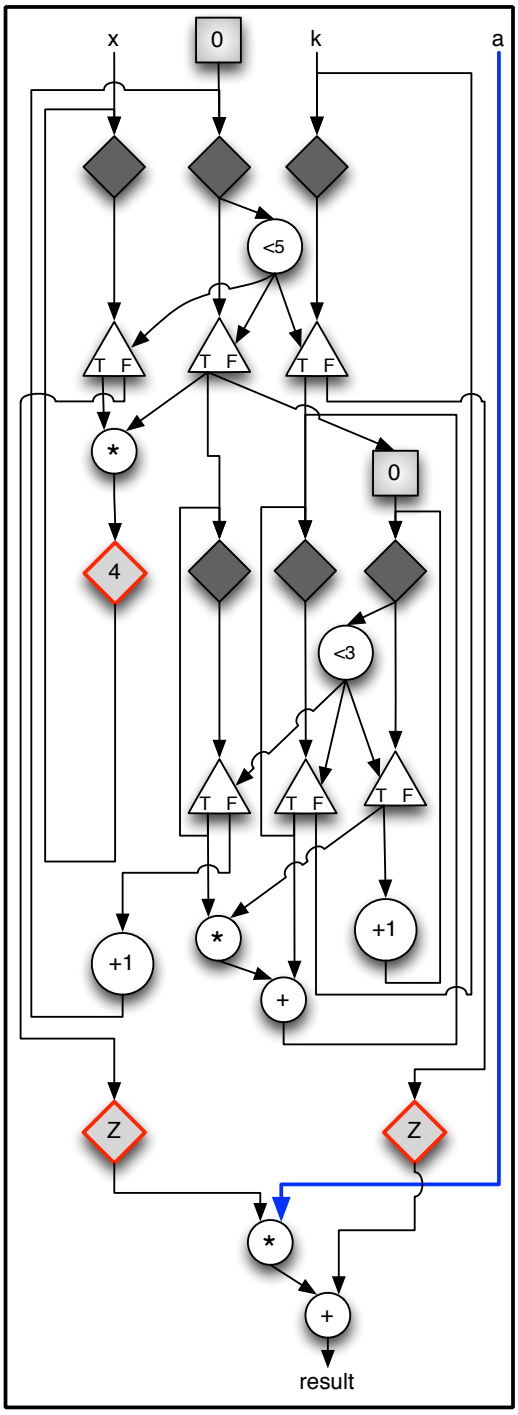

Figura 5.9: Exemplo de redução de *overhead* em *loops* aninhados com o *Loop Skipping* combinado com *Tag Resetting*. A figura mostra o grafo referente a Figura 5.2 com instruções **IncTagI** (representadas por losangos vermelhos marcados com *Z*) e **zTag** (representadas por losangos vermelhos contendo o imediato). Note que são evitadas as instruções de **Push** e **Pop**. O operando *a* pode ser enviado diretamente ao seu destino. O operando *x* passa pelas instruções **IncTagI** e **zTag** para sair dos *loops*.

a quantidade de iterações processadas por esses laços, já que a cada iteração do *loop* externo poderá definir uma nova quantidade de iterações para o *loop* interno, sendo necessário primeiro executar os dois laços para obter o total de iterações e depois fazer o *Loop Skipping*, serializando a aplicação.

Considere um cenário com dois *loops* aninhados, onde n é a quantidade de iterações do primeiro *loop*, m é a quantidade de iteração do segundo *loop* e o primeiro *loop* é o externo. Para prever as iterações do *loop* externo deve ser aplicada a seguinte fórmula:

$$N_{\text{iterations}} = n + 1 + n \times (m + 1)$$

Em todo o *loop* é necessário uma iteração extra para que a condição dele possa ser invalidada e o fluxo de execução sair do *loop*. Em casos que existam mais de dois *loops* aninhados, a fórmula apresentada terá que ser aplicada diversas vezes começando do *loop* mais interno com seu pai até chegar no *loop* mais externo.

Para atender aos casos de *loops* aninhados, deve-se usar as instruções **PredTag** e **JoinTag**. Essas instruções foram criadas para encapsular o cálculo do número de iterações com um grão mais grosso, minimizando custos de interpretação, caso os cálculos fossem feitos por instruções de grão fino, geradas pelo compilador. A instrução **PredTag** é responsável por definir o número de iteração de um *loop* sem levar em consideração se ele tem filhos. O **PredTag** recebe as variáveis f, l e s, e calcula o número de iteração de acordo o tipo de operador relacional e o operador de passo do laço, da mesma forma que o compilador executa para instrução **IncTagI**. A instrução **JoinTag** é responsável por executar a fórmula apresentada para determinar a iteração do *loop* pai nos casos de aninhamento de laços. Quando é determinado o aninhamento de laços, os operandos produzidos por duas instruções **PredTag** são enviadas para o **JoinTag** que obtém o número de iterações dos *loops* aninhados. O resultado do **JoinTag** é enviado como o operando de entrada para instrução **IncTagN**.

5.4.2 Vantagens e Desvantagens

A principal vantagem do *Loop Skipping* é evitar o aumento de profundidade da pilha sempre que possível, através da predição dos rótulos das *skip variables*. No caso que não for possível prever as iterações, uma solução genérica e compreensiva deverá ser usada, tal como o mecanismo *Stack-Tagged Dataflow*. Outra vantagem da técnica, é que ela não adiciona instruções extras nos *loops*, podendo reduzir o caminho crítico.

Por outro lado, *Loop Skipping* não previne a possibilidade *overflow* do rótulo, pois os rótulos não são resetados e nem empilhados em uma pilha.

5.4.3 Modificações no TALM

Para incluir o suporte ao *Loop Skipping* no conjunto de ferramentas da Trebuchet foram feitas as seguintes mudanças:

- Fazer o compilador Couillard identificar *loops* que seguem o padrão explicado nessa Seção. Isto inclui *loops* aninhados para ambos os casos apresentados.
- Adicionar as instruções **IncTagI**, **IncTagN**, **PredTag** e **JoinTag** na geração do arquivo com a linguagem de montagem do TALM.
- Adicionar as instruções **IncTagI**, **IncTagN**, **PredTag** e **JoinTag** no montador do TALM.
- Implementar a lógica de execução do **IncTagI**, **IncTagN**, **PredTag** e **JoinTag** na máquina virtual Trebuchet.

A sintaxe da instrução `inctagi` é a seguinte:

```
inctagi <nome>, <operando>, <imediato>
```

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

- `inctagi`: é o mnemônico da instrução.
- `nome`: é o identificador da instância da instrução usado para referenciar o operando de saída.
- `operando`: indica o operando de entrada no qual o rótulo será incrementado.
- `imediato`: valor inteiro a ser incrementado no rótulo do operando.

A sintaxe da instrução `inctagn` é a seguinte:

```
inctagn <nome>, <op1>, <op2>
```

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

- `inctagn`: é o mnemônico da instrução.
- `nome`: é o identificador da instância da instrução usado para referenciar o operando de saída.
- `op1`: indica o operando de entrada no qual o rótulo será incrementado.
- `op2`: operando a ser incrementado no rótulo do operando de entrada.

A sintaxe da instrução `predtag` é a seguinte:

```
predtag <nome>, <op1>, <op2>, <op3>, <imediato>
```

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

- `predtag`: é o mnemônico da instrução.
- `nome`: é o identificador da instância da instrução usado para referenciar o operando de saída.
- `op1`: indica o valor inicial da variável de controle do laço.
- `op2`: indica o valor limite da variável de controle do laço.
- `op3`: indica o valor de incremento em cada iteração do laço.
- `imediato`: valor inteiro indicando o tipo de operador relacional e de passo a ser utilizado para a predição.

A sintaxe da instrução `jointag` é a seguinte:

```
jointag <nome>, <op1>, <op2>
```

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

- `jointag`: é o mnemônico da instrução.
- `nome`: é o identificador da instância da instrução usado para referenciar o operando de saída.
- `op1`: indica o número de iterações do laço pai.
- `op2`: indica o número de iterações do laço filho.

Capítulo 6

Experimentos e Resultados

Para avaliar o potencial das técnicas de otimização de *loop* em *dataflow* propostas foram conduzidos seis experimentos. A máquina hospedeira usada para todos os experimentos neste capítulo possui um processador Intel®Core™i7-3770 (3GHz) com 32GB de memória DDR3 1600MHz, rodando o sistema operacional Linux.

Nesse capítulo serão apresentados e discutidos os experimentos abordando uma análise dos resultados e indicando em quais cenários a adoção de cada técnica seria mais apropriada.

6.1 Experimento 1

O primeiro experimento consiste em verificar o *overhead* no casamento do *Stack-Tagged Dataflow*. A lógica de comparação de ambas versões (original e com pilha) foi copiada da máquina virtual Trebuchet para um programa externo permitindo medir somente o tempo da comparação. Além disso, também foi copiada a versão otimizada da comparação de pilha apresentada na Seção 5.2.4.

Nas comparações com pilha, a aplicação permite a seleção da profundidade de pilha para simular o casamento dentro de *loops* aninhados. Quanto mais profundo for um *loop*, mais elementos serão empilhados e, portanto, é esperado que se apresente maiores *overheads*. Note que o programa compara o custo do casamento em caso de sucesso resultando no pior caso do casamento (mais custoso), visto que os conflitos são detectados antes de alcançar o fim das pilhas (comparação de *loops* para quando o primeiro par de elementos diferentes são encontrados). No casamento de pilha otimizado, o custo de conflito é igual ao custo do pior caso (sucesso), já que todos os elementos da pilha sempre são comparados.

A Figura 6.1 mostra os resultados do primeiro experimento exibindo as desacelerações (*slowdowns*) dos casamentos do *Stack-Tagged Dataflow* quando comparado com a solução original. Cada cenário processa 100 mil casamentos e foi executado 20 vezes, com desvio padrão desprezível. Embora as desacelerações sejam signifi-

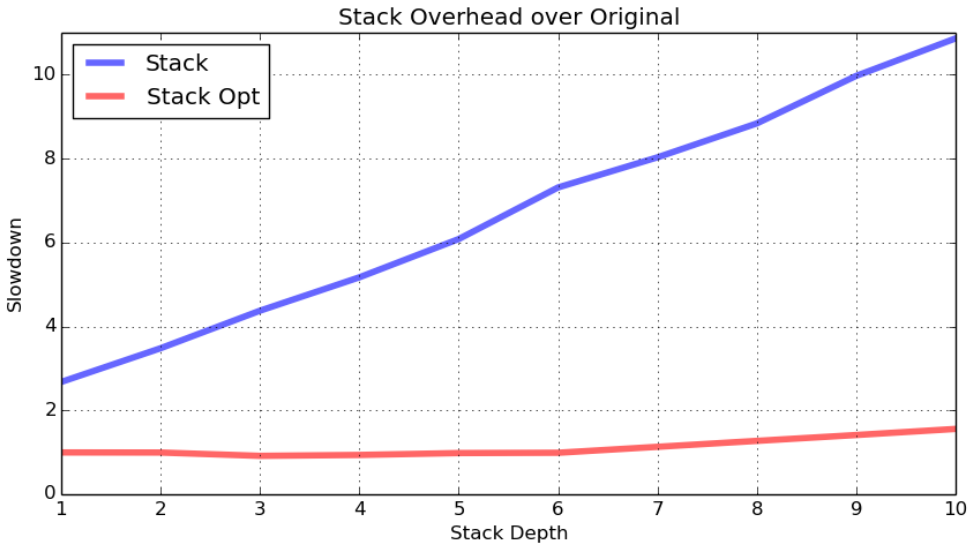

Figura 6.1: *Overheads* nos casamentos com pilha de rótulos. O eixo x mostra a profundidade da pilha em cada cenário e o eixo y mostra a desaceleração obtida pelas versões com pilha em relação com a versão original.

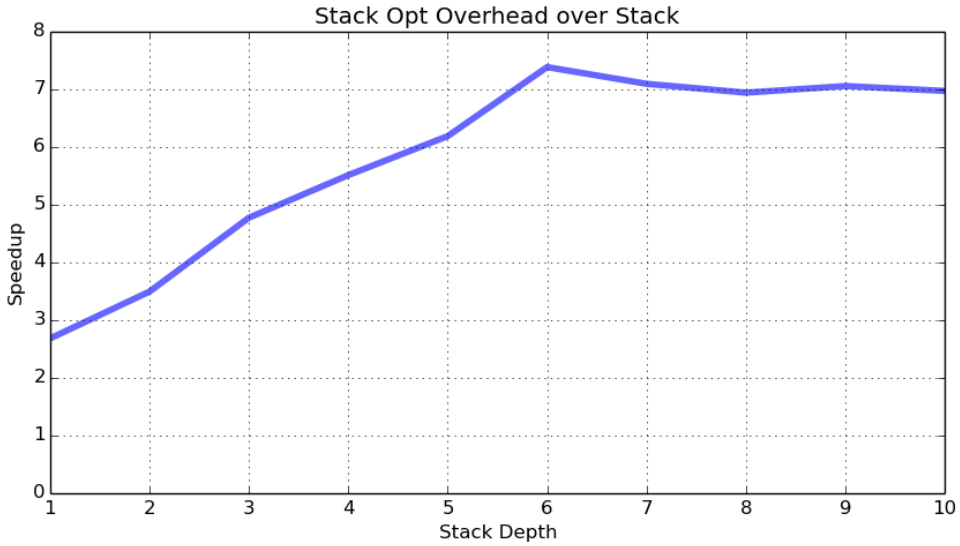

Figura 6.2: Melhoria do casamento otimizado com pilha de rótulos. O eixo x mostra a profundidade da pilha em cada cenário e o eixo y mostra o *speedup* obtido pela versão otimizada em relação com a versão de pilha.

cantemente altas no casamento de pilha sem otimização, a operação de casamento normalmente corresponde a menos de $0,5\%$ do tempo total de execução de uma aplicação com granularidade grossa.

A Figura 6.2 mostra o *speedup* obtido pelo casamento otimizado em relação o casamento com pilha. O casamento de pilha otimizado alcança *speedup* de 7. Essa otimização impulsiona o paralelismo da aplicação permitindo que super-instruções com granularidade mais fina possam executar de forma eficiente no modelo *dataflow*.

6.2 Experimento 2

O segundo experimento tem o objetivo de avaliar o *overhead* das instruções apresentadas nas técnicas de otimização (**Push**, **Pop**, **zTag**, **IncTagI**, **IncTagN**, **PredTag** e **JoinTag**). Uma aplicação sintética foi desenvolvida para executar a lógica de cada uma das instruções 1 bilhão de vezes e essa aplicação foi rodada 20 vezes. Como as execuções mantém o mesmo rótulo nas instruções, o uso de *cache* é mais eficiente que um cenário real, já que em 1 bilhão de acessos ocorrerá somente 1 *cache miss*. Entretanto, ao comparar o tempo de execução das instruções se obterá a mesma taxa de tempo de um cenário real.

Os resultados mostram que as instruções de **Push** e **Pop** têm o mesmo custo. As instruções das demais técnicas foram desenvolvidas para minimizar o custo das instruções de pilha em programas onde é possível prever a quantidade de iteração de laço. A Tabela 6.1 mostra os *speedups* das instruções que evitam o uso de pilha em relação ao uso de instruções **Push** e **Pop**. Para as instruções que processam o rótulo do operando também foi executado a lógica abordando as situações em que o rótulo é uma pilha, mostrando a redução de custo nos casos que as técnicas são aplicadas em *loops* internos. As instruções **JoinTag** e **PredTag** auxiliam a instrução **IncTagN** e, portanto, seu processamento não envolve o rótulo do operando possuindo o mesmo custo em ambas versões da máquina virtual.

Versão	Instrução	*Speedup*
Sem Pilha	**zTag**	4.4
Com Pilha	**zTag**	3.4
Sem Pilha	**IncTagI**	5.1
Com Pilha	**IncTagI**	3.1
Sem Pilha	**IncTagN**	4.4
Com Pilha	**IncTagN**	3.4
-	**JoinTag**	5.0
-	**PredTag**	0.7

Tabela 6.1: *Speedups* das instruções usadas nas técnicas de *Loop Skipping* e *Tag Resetting* em relação as instruções de pilha.

A Tabela 6.2 mostra o *speedup* da técnica *Loop Skipping* com **IncTagN** nos cenários de um *loop* simples e dois *loops* aninhados em relação as instruções de pilha. Em todos os cenários o custo foi maior do que a execução de instruções de *Push* e *Pop*. No caso de um *loop* simples a instrução *PredTag* é executada para determinar a quantidade de iteração em tempo de execução e, em seguida, o resultado é processado pela instrução **IncTagN**. Já no caso de dois *loops* aninhados, são executadas duas instruções **PredTag** para determinar as iterações dos dois laços e os resultados são enviados para a instrução **JoinTag** que encaminhará, em seguida, seu resultado para a instrução **IncTagN**. Note que o uso do *Loop Skipping* com **IncTagN** apesar de ser mais custoso do que o *Stack-Tagged Dataflow* ainda pode obter um custo total menor ao ser empregado em cenários com muito processamento de instruções de pilha, como por exemplo, em *loops aninhados* onde o *loop* externo possui mais iterações do que seu *loop* interno. Outro ponto em relação ao *Loop Skipping* com **IncTagN** é que quanto maior a profundidade dos *loops* aninhados, maior será o custo da técnica.

Cenário	Versão	Instruções	*Speedup*
Loop simples	Sem Pilha	**IncTagN** + **PredTag**	0.75
Loop simples	Com Pilha	**IncTagN** + **PredTag**	0.65
Dois *Loops* aninhados	Sem Pilha	**IncTagN** + 2× **PredTag** + **JoinTag**	0.39
Dois *Loops* aninhados	Com Pilha	**IncTagN** + 2× **PredTag** + **JoinTag**	0.37

Tabela 6.2: *Speedups* do uso de *Loop Skipping* com **IncTagN** em relação ao uso de pilha.

6.3 Experimento 3

O terceiro experimento consiste numa aplicação sintética com *loop* simples contendo uma super-instrução em seu corpo, com o objetivo de avaliar como as técnicas apresentadas podem contribuir para reduzir o *overhead* da versão original da Trebuchet que utiliza o casamento de inteiros. A aplicação recebe como parâmetros de entrada: o número de iterações do *loop*, o peso de computação da super-instrução e o número de variáveis (operandos) criadas para serem passadas por fora do *loop* na versão com rótulo de pilha. O número de iterações e as *skip variables* estão relacionadas com o número de instruções evitadas pelas técnicas, enquanto o peso de computação amortizará seu *overhead*. O experimento foi executado 10 vezes com desvio padrão irrelevante em um único elemento de processamento (execução serial), pois o principal objetivo é determinar o *overhead* de cada técnica.

As figuras 6.3, 6.4, 6.5, 6.6, 6.7 e 6.8 mostram os resultados do terceiro experimento. Em cada cenário, o peso de computação (*Computation Weight*) varia, o

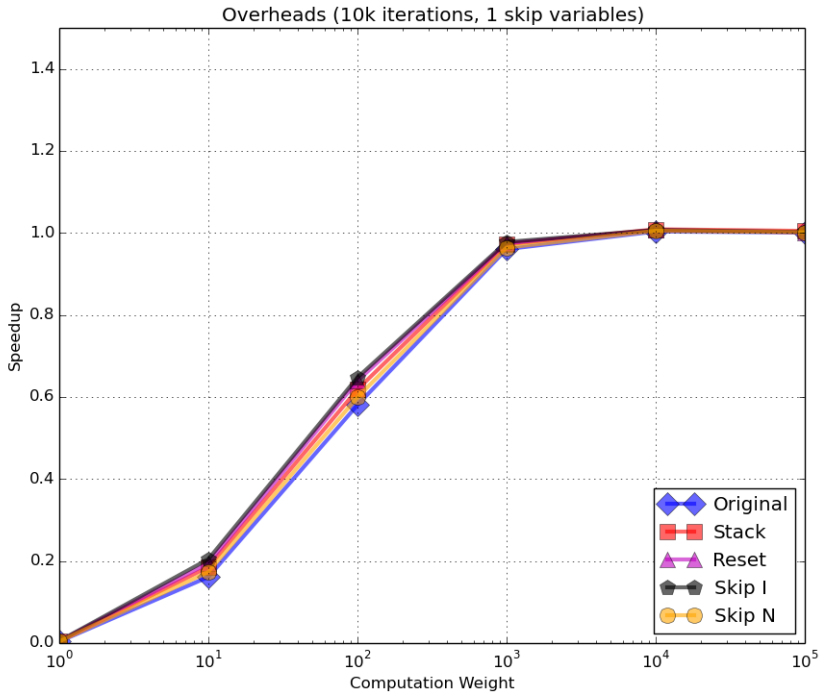

Figura 6.3: *Overheads* das versões *dataflow* para uma *skip variable*. O eixo x mostra o peso de computação e o eixo y mostra o *speedup* obtido em relação a um versão da aplicação C sequencial sem *dataflow*.

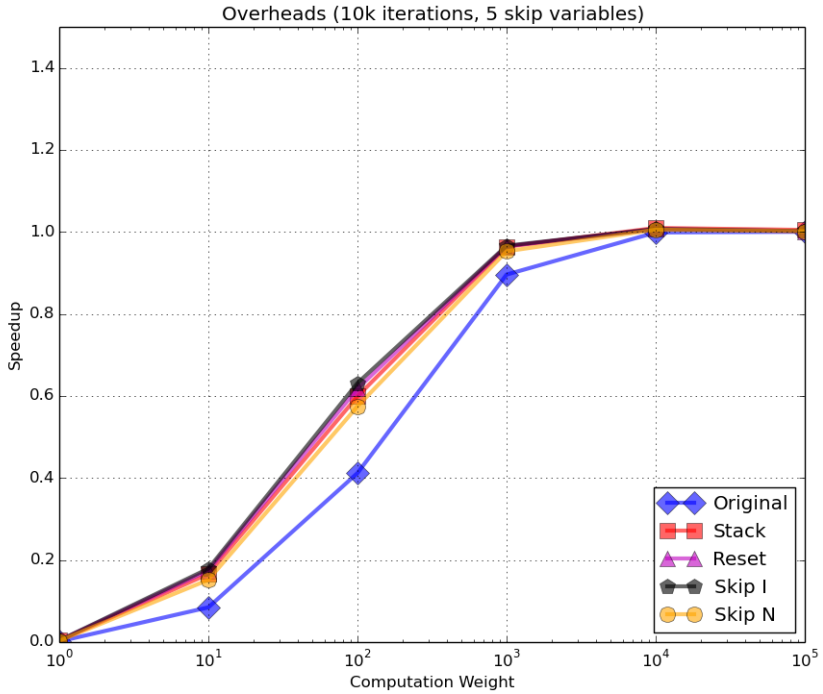

Figura 6.4: *Overheads* das versões *dataflow* para cinco *skip variables*. O eixo x mostra o peso de computação e o eixo y mostra o *speedup* obtido em relação a um versão da aplicação C sequencial sem *dataflow*.

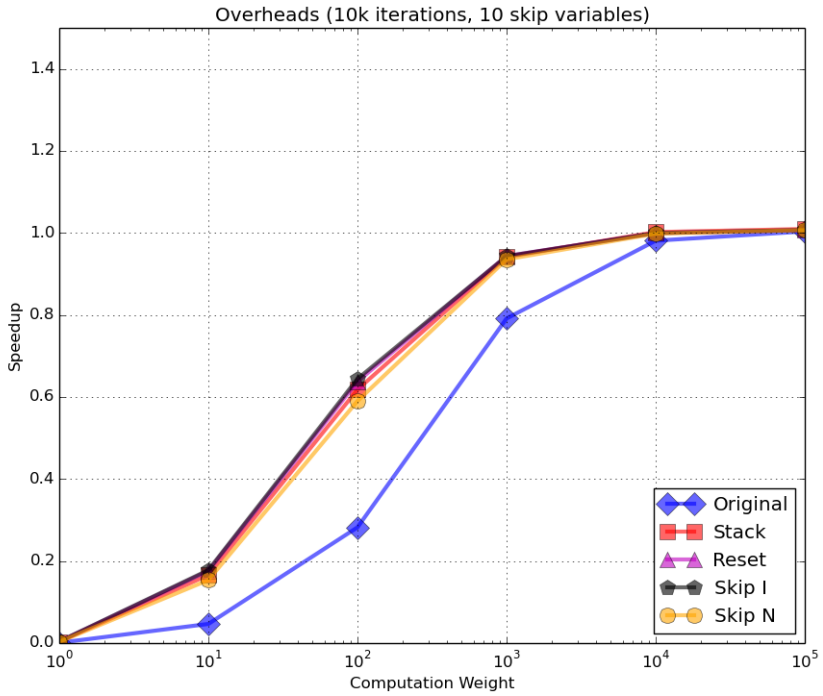

Figura 6.5: *Overheads* das versões *dataflow* para dez *skip variables*. O eixo x mostra o peso de computação e o eixo y mostra o *speedup* obtido em relação a um versão da aplicação C sequencial sem *dataflow*.

número de iterações é fixado em 10 mil e as *skip variables* são fixadas em 1, 5 e 10. Nas figuras 6.3, 6.4 e 6.5, os gráficos mostram os *speedups* relacionados com uma versão C sem *dataflow* para: a Trebuchet original (*dataflow* sem otimizações), *Stack-Tagged Dataflow*, *Tag Resetting* e as duas variações do *Loop Skipping*. A versão C sem *dataflow* foi escolhida como *baseline* para mostrar que todas as versões *dataflow* apresentam custo de interpretação significante para tarefas com grão fino e custo de interpretação nulo para tarefas com grão grosso. Observe que o aumento do peso de computação em tarefas com grão grosso anula o *overhead* pois a quantidade de variáveis na aplicação se mantém. Se a quantidade de variáveis também aumentasse seria necessário um peso de computação maior para poder amortizar completamente o custo de interpretação. Para as tarefas de granularidade média, observa-se que as técnicas de otimizações de *loops* melhoram o desempenho, pois o *overhead* é reduzido em relação ao custo da versão original. Com isso, verifica-se que essas otimizações no *loop* são essenciais na redução de *overheads* permintindo a execução de tarefas com grão mais fino, que são desejadas para favorecer a escalabilidade. Os resultados mostram que todas as técnicas de otimização de *loops* se igualam no desempenho sendo melhores do que a versão original. Esse comportamento é esperado, pois todas as técnicas processam as *skip variables* com baixo custo em um único *loop*. Além do mais, para os pesos de computação acima de 1000 o custo de interpretação se torna

irrelevante.

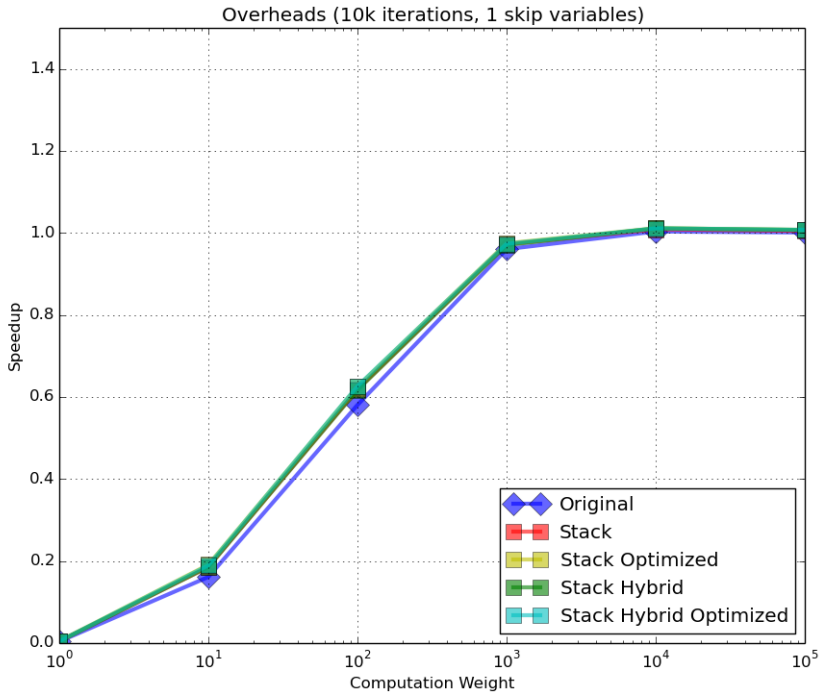

Figura 6.6: *Overheads* das versões *dataflow* com pilha para uma *skip variable*. O eixo x mostra o peso de computação e o eixo y mostra o *speedup* obtido em relação a um versão da aplicação C sequencial sem *dataflow*.

Nas figuras 6.6, 6.7 e 6.8, os gráficos mostram os *speedups* relacionados com uma versão C sem *dataflow* para: a Trebuchet original (*dataflow* sem otimizações), *Stack-Tagged Dataflow* sem otimização, *Stack-Tagged Dataflow* com otimização, a Trebuchet híbrida sem otimização e a Trebuchet híbrida com otimização do *Stack-Tagged Dataflow*. Os resultados mostram que as otimizações do *Stack-Tagged Dataflow* conseguem reduzir o custo de interpretação em tarefas com granularidade fina e obtêm ganho em relação ao *Stack-Tagged Dataflow* sem otimização, além de ser melhor do que a versão original. Também como as demais versões, para os pesos de computação acima de 1000 o custo de interpretação se torna irrelevante.

6.4 Experimento 4

O quarto experimento também consiste numa aplicação sintética com *loop* simples contendo uma super-instrução em seu corpo, mas com o objetivo de verificar os efeitos do número de iterações e *skip variables* no desempenho do *Stack-Tagged Dataflow*. A aplicação recebe como parâmetros de entrada: o número de iterações do *loop*, o peso de computação (*Computation Weight*) da super-instrução e o número

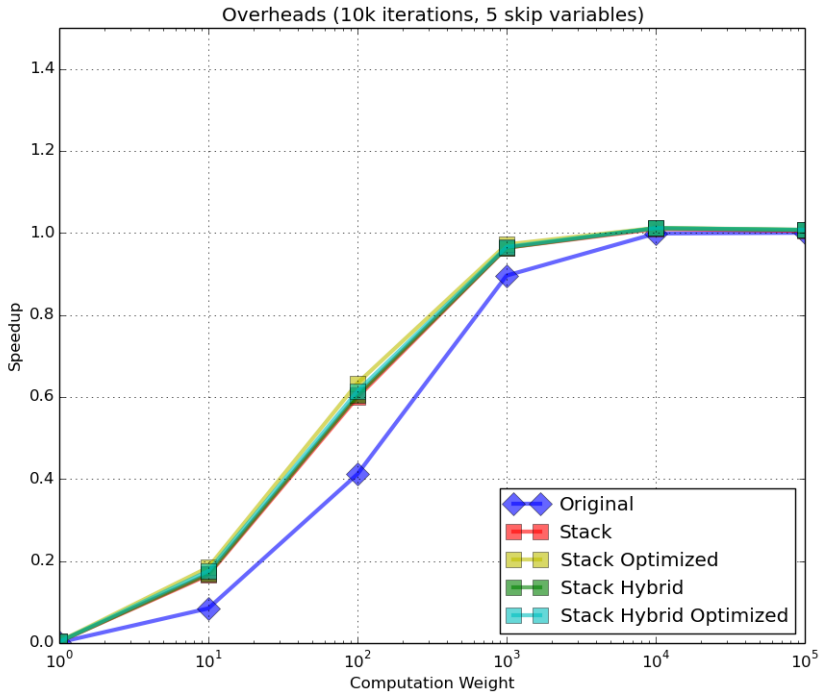

Figura 6.7: *Overheads* das versões *dataflow* com pilha para cinco *skip variables*. O eixo x mostra o peso de computação e o eixo y mostra o *speedup* obtido em relação a um versão da aplicação C sequencial sem *dataflow*.

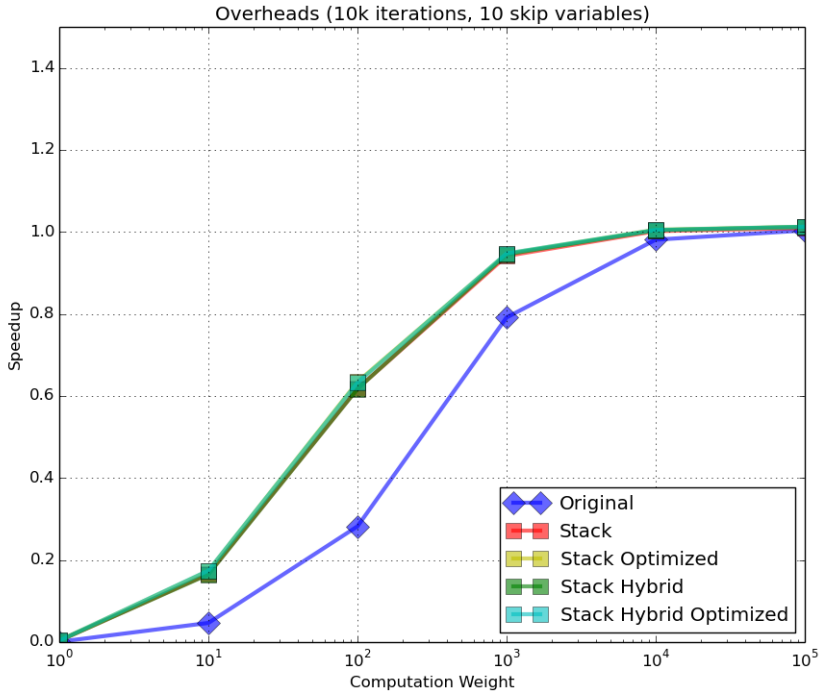

Figura 6.8: *Overheads* das versões *dataflow* com pilha para dez *skip variables*. O eixo x mostra o peso de computação e o eixo y mostra o *speedup* obtido em relação a um versão da aplicação C sequencial sem *dataflow*.

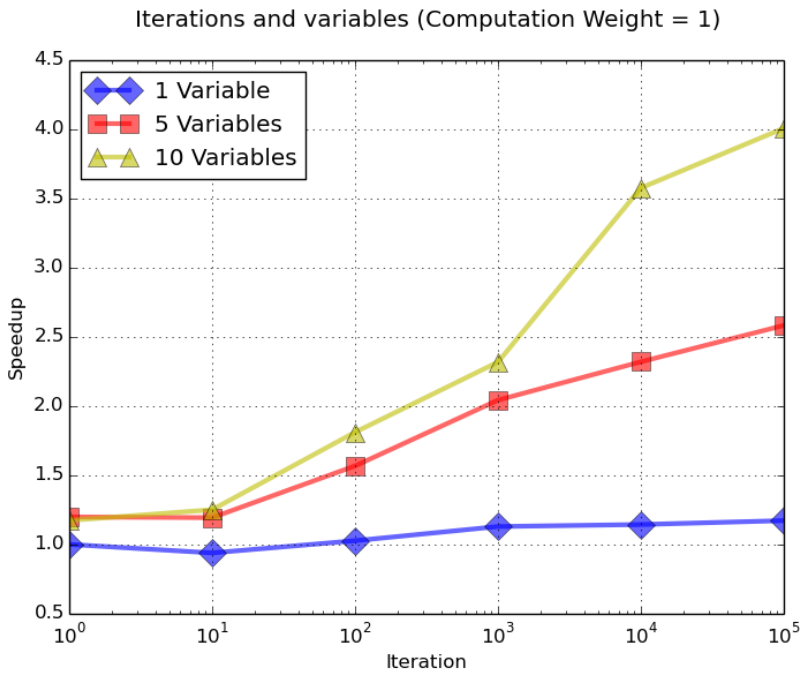

Figura 6.9: Desempenho do *Stack-Tagged Dataflow* sem otimização em um *loop* simples para peso de computação 1. O eixo x mostra a variação dos números de iterações e o eixo y o *speedup* alcançado em relação a versão original. Cada linha mostra o resultado dos cenários com quantidade diferentes de *skip variables*(1, 5 e 10).

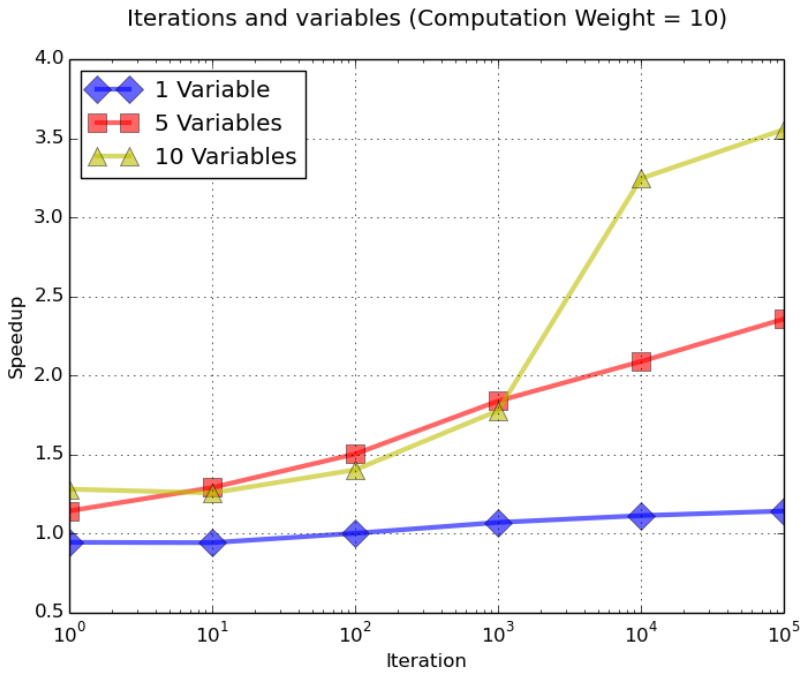

Figura 6.10: Desempenho do *Stack-Tagged Dataflow* sem otimização em um *loop* simples para peso de computação 10. O eixo x mostra a variação dos números de iterações e o eixo y o *speedup* alcançado em relação a versão original. Cada linha mostra o resultado dos cenários com quantidade diferentes de *skip variables*(1, 5 e 10).

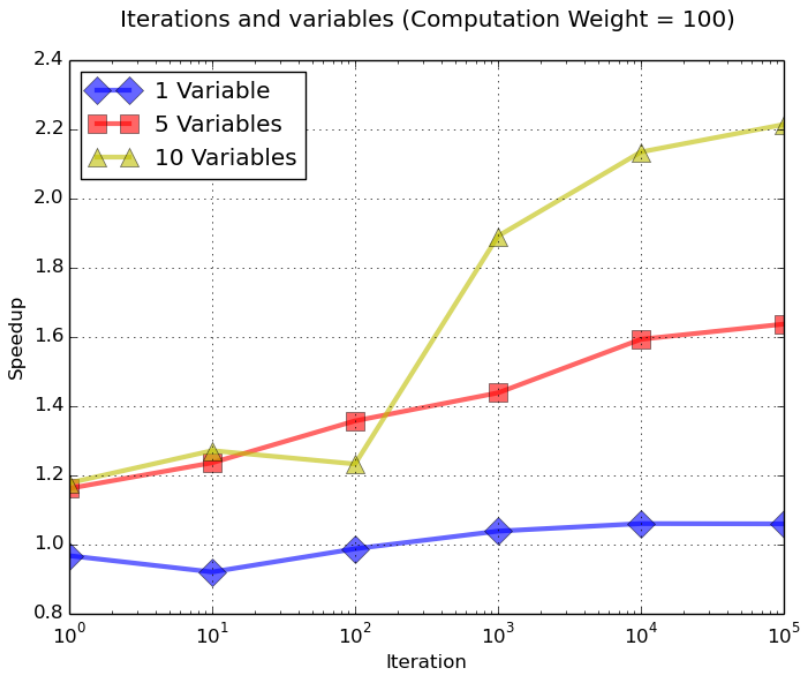

Figura 6.11: Desempenho do *Stack-Tagged Dataflow* sem otimização em um *loop* simples para peso de computação 100. O eixo x mostra a variação dos números de iterações e o eixo y o *speedup* alcançado em relação a versão original. Cada linha mostra o resultado dos cenários com quantidade diferentes de *skip variables*(1, 5 e 10).

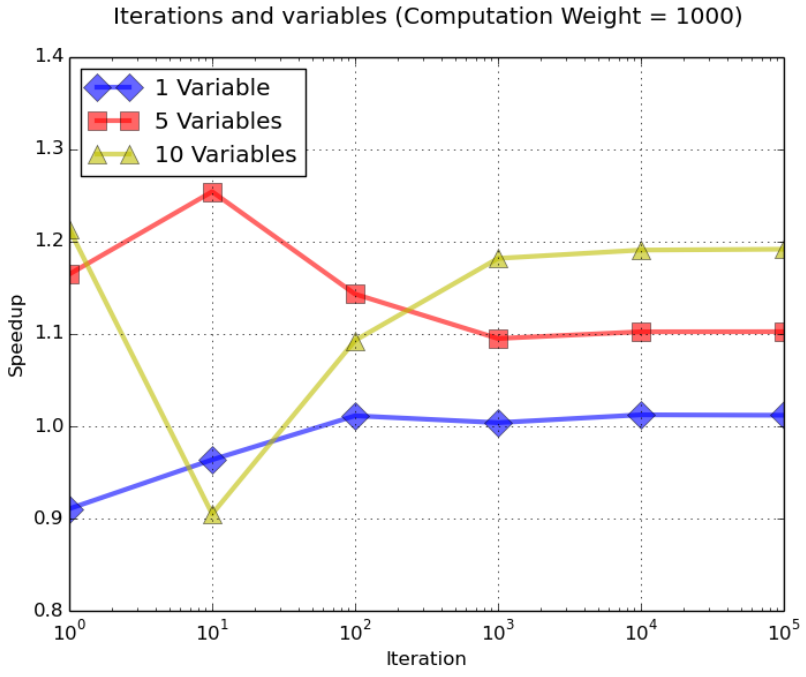

Figura 6.12: Desempenho do *Stack-Tagged Dataflow* sem otimização em um *loop* simples para peso de computação 1000. O eixo x mostra a variação dos números de iterações e o eixo y o *speedup* alcançado em relação a versão original. Cada linha mostra o resultado dos cenários com quantidade diferentes de *skip variables*(1, 5 e 10).

de variáveis (operandos) criadas para serem passadas por fora do *loop* na versão com rótulo de pilha. O número de iterações e as *skip variables* estão relacionadas com o número de instruções evitadas pela técnica e o peso de computação amortiza seu *overhead*. O experimento foi executado 20 vezes com desvio padrão irrelevante em um único elemento de processamento, visto que o principal objetivo é determinar o *overhead* da técnica.

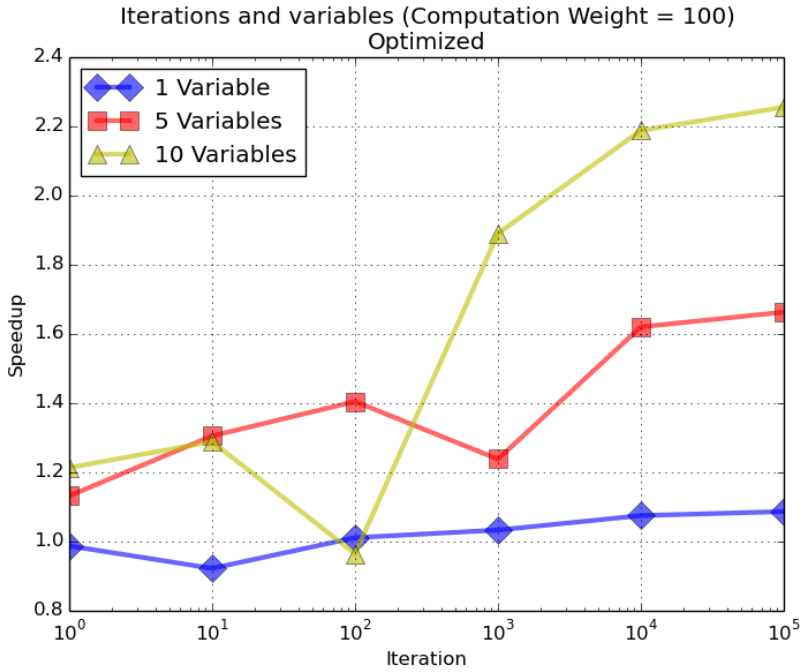

Figura 6.13: Desempenho do *Stack-Tagged Dataflow* otimizado em um *loop* simples. O eixo x mostra a variação dos números de iterações e o eixo y o *speedup* alcançado em relação a versão original. Cada linha mostra o resultado dos cenários com quantidade diferentes de *skip variables*(1, 5 e 10) com o peso fixo em 100.

As figuras 6.9, 6.10, 6.11, 6.12, 6.13, 6.14 e 6.15 mostram os *speedups* relacionados a Trebuchet original. Em cada cenário, o peso de computação foi fixado e o número de iterações foi variando. As figuras 6.9, 6.10, 6.11 e 6.12 apresentam os resultados de todos os cenários do *Stack-Tagged Dataflow* sem otimização. Os *speedups* aumentam com o número de iterações e *skip variables* conforme mais instruções **IncTag** e **Steer** são evitadas. As tarefas com grão médio evidenciam o maior ganho da técnica, enquanto com as tarefas de grãos grossos mostram que a diferença entre as versões é amortizada. As figuras 6.13, 6.14 e 6.15 mostram os resultados das variações do *Stack-Tagged Dataflow* no cenário em que o peso de computação é fixado em 100. Os *speedups* das variações também aumentam com o número de iterações e *skip variables* conforme mais instruções **IncTag** e **Steer** são evitadas. Isso permite maior liberdade de escolha em decidir qual versão da técnica deve ser adotada, visto que todas conseguem minimizar o custo da versão original.

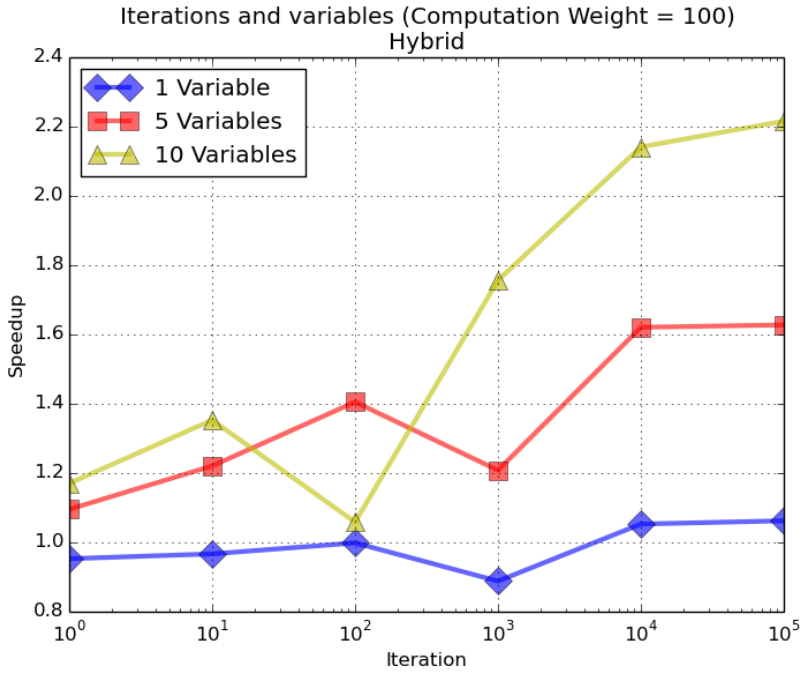

Figura 6.14: Desempenho do *Stack-Tagged Dataflow* híbrido em um *loop* simples. O eixo x mostra a variação dos números de iterações e o eixo y o *speedup* alcançado em relação a versão original. Cada linha mostra o resultado dos cenários com quantidade diferentes de *skip variables*(1, 5 e 10) com o peso fixo em 100.

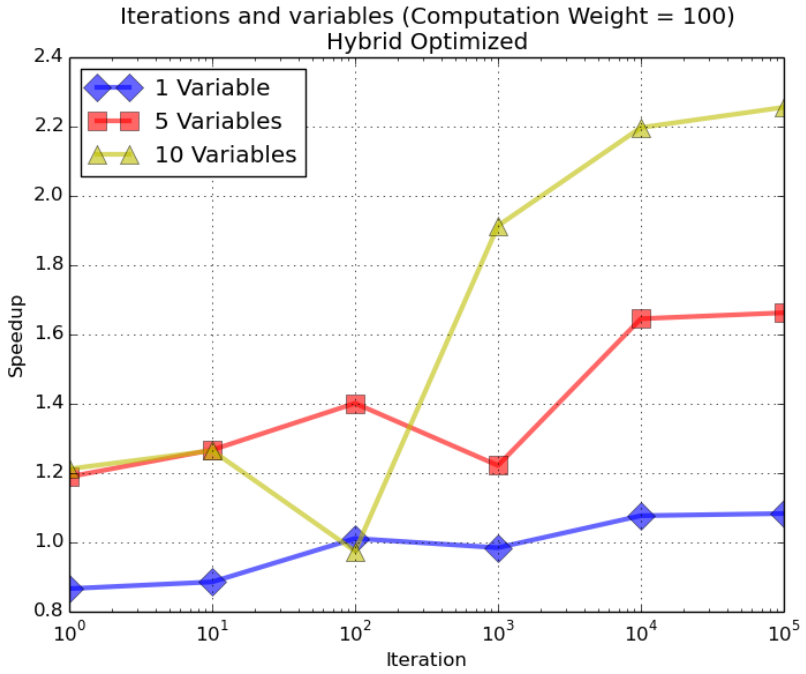

Figura 6.15: Desempenho do *Stack-Tagged Dataflow* híbrido e otimizado em um *loop* simples. O eixo x mostra a variação dos números de iterações e o eixo y o *speedup* alcançado em relação a versão original. Cada linha mostra o resultado dos cenários com quantidade diferentes de *skip variables*(1, 5 e 10) com o peso fixo em 100.

6.5 Experimento 5

O objetivo do quinto experimento é analisar o custo das instruções de **Push** e **Pop** em *loops* aninhados. Para isso, uma aplicação sintética com dois *loops* aninhados foi desenvolvida recebendo como parâmetros: o número de *skip variables* em ambos *loops*, o número de iterações do *loop* interno e o peso de computação do *loop* interno. O número de iteração do *loop* externo foi fixado em 5000 e cada cenário foi executado 20 vezes com desvio padrão irrelevante. Observe, neste experimento, que o número de instruções **Push** e **Pop** são fixos pois o número de iterações do *loop* externo também é fixo. Por outro lado, o número de instruções **IncTag** e **Steer** executadas na versão original aumenta conforme o aumento do número de iterações do *loop* interno.

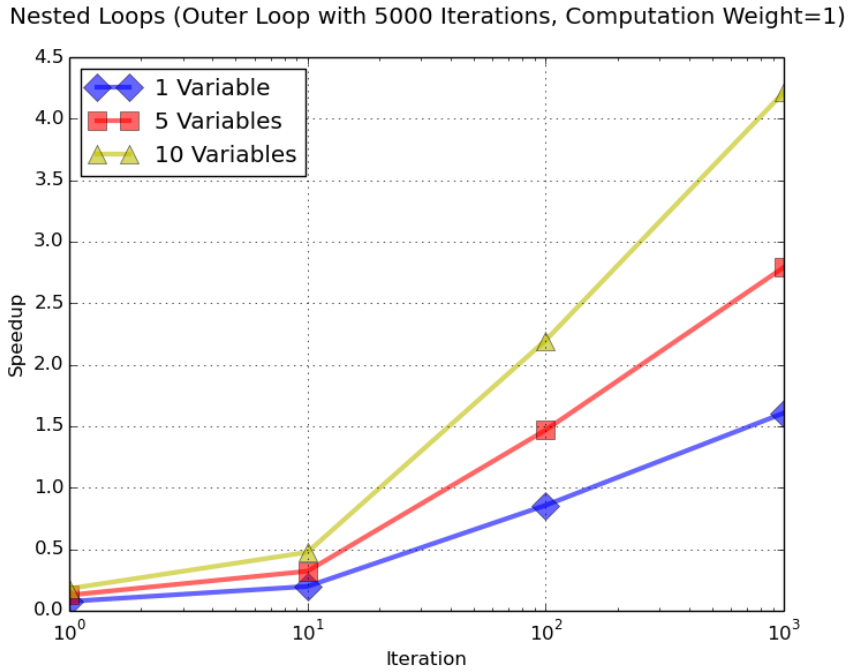

Figura 6.16: Desempenho do *Stack-Tagged Dataflow* sem otimização em *loops* aninhados para peso de computação 1. O eixo x mostra a variação dos números de iterações do *loop* interno e o eixo y o *speedup* alcançado em relação a versão original. Cada linha mostra o resultado dos cenários com quantidade diferentes de *skip variables*(1, 5 e 10).

As figuras 6.16, 6.17, 6.18, 6.19, 6.20 e 6.21 mostram os *speedups* obtidos sobre a Trebuchet original deste experimento. As figuras 6.16, 6.17 e 6.18 apresentam os resultados de todos os cenários do *Stack-Tagged Dataflow* sem otimização e as figuras 6.19, 6.20 e 6.21 mostram os resultados das variações do *Stack-Tagged Dataflow* no cenário em que o peso de computação do *loop* interno é fixado em 100. Em *loops* aninhados, quando o *loop* interno possui poucas iterações, a versão original tem o melhor desempenho já que o custo das instruções **Push** e **Pop** não é amortizado

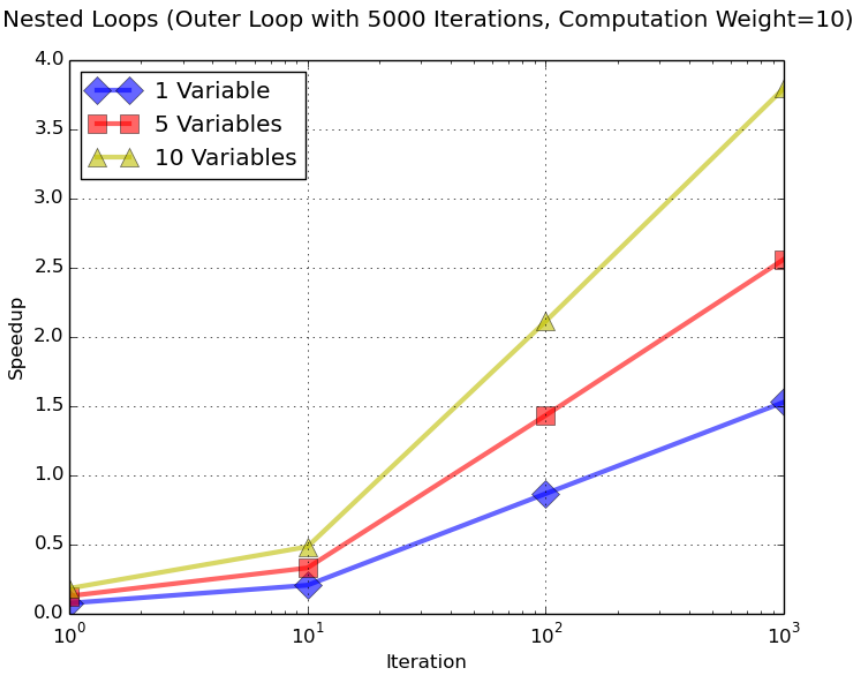

Figura 6.17: Desempenho do *Stack-Tagged Dataflow* sem otimização em *loops* aninhados para peso de computação 10. O eixo x mostra a variação dos números de iterações do *loop* interno e o eixo y o *speedup* alcançado em relação a versão original. Cada linha mostra o resultado dos cenários com quantidade diferentes de *skip variables*(1, 5 e 10).

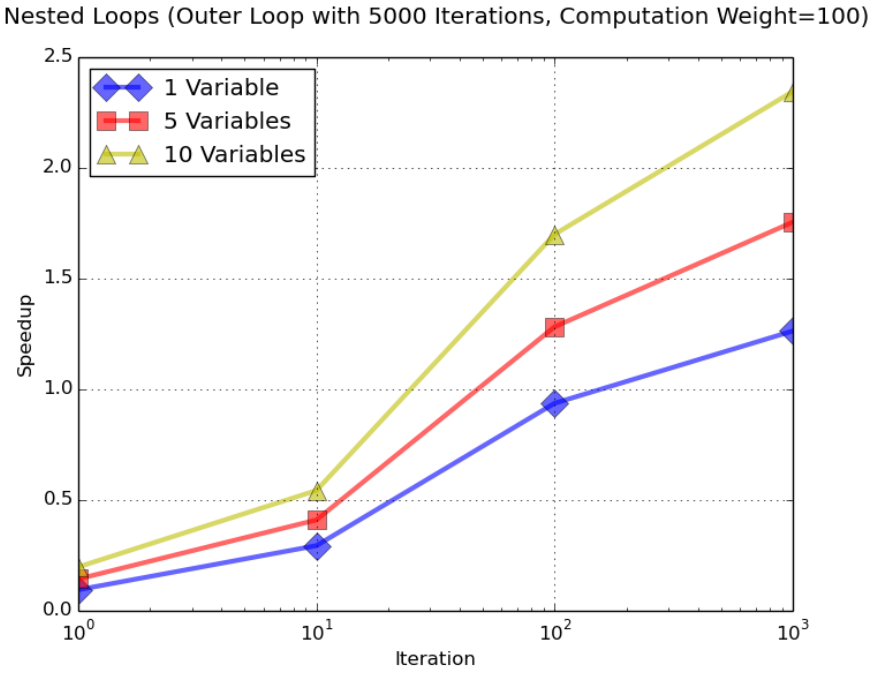

Figura 6.18: Desempenho do *Stack-Tagged Dataflow* sem otimização em *loops* aninhados para peso de computação 100. O eixo x mostra a variação dos números de iterações do *loop* interno e o eixo y o *speedup* alcançado em relação a versão original. Cada linha mostra o resultado dos cenários com quantidade diferentes de *skip variables*(1, 5 e 10).

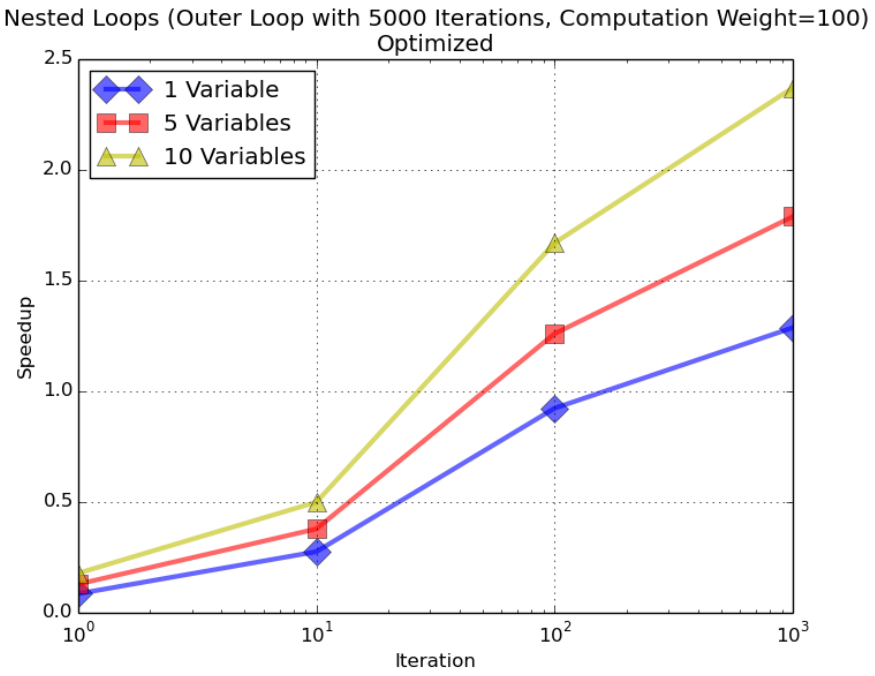

Figura 6.19: Desempenho do *Stack-Tagged Dataflow* otimizado em *loops* aninhados. O eixo x mostra a variação dos números de iterações do *loop* interno e o eixo y o *speedup* alcançado em relação a versão original. Cada linha mostra o resultado dos cenários com quantidade diferentes de *skip variables*(1, 5 e 10) com o peso fixo em 100.

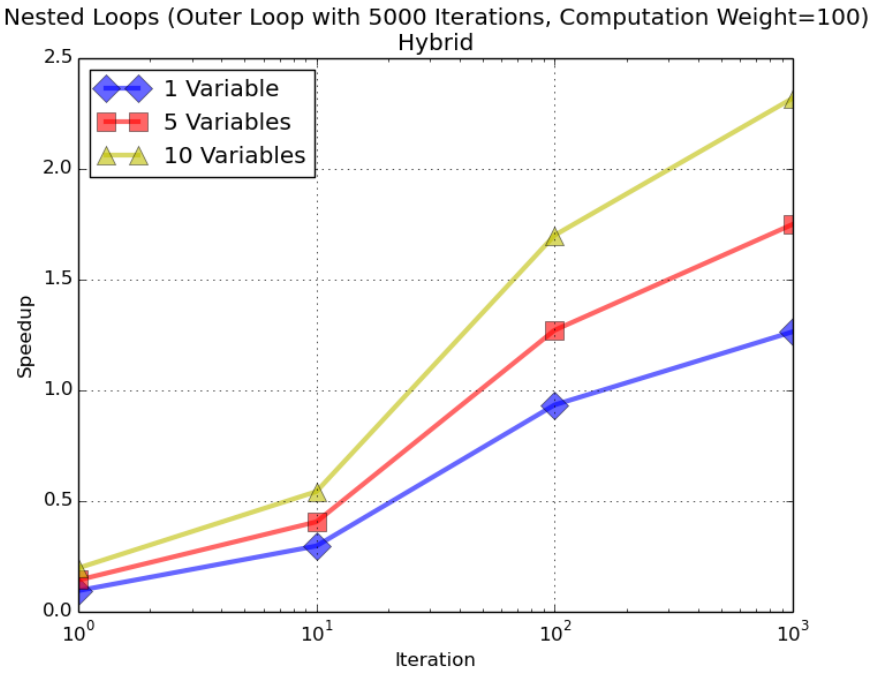

Figura 6.20: Desempenho do *Stack-Tagged Dataflow* híbrido em *loops* aninhados. O eixo x mostra a variação dos números de iterações do *loop* interno e o eixo y o *speedup* alcançado em relação a versão original. Cada linha mostra o resultado dos cenários com quantidade diferentes de *skip variables*(1, 5 e 10) com o peso fixo em 100.

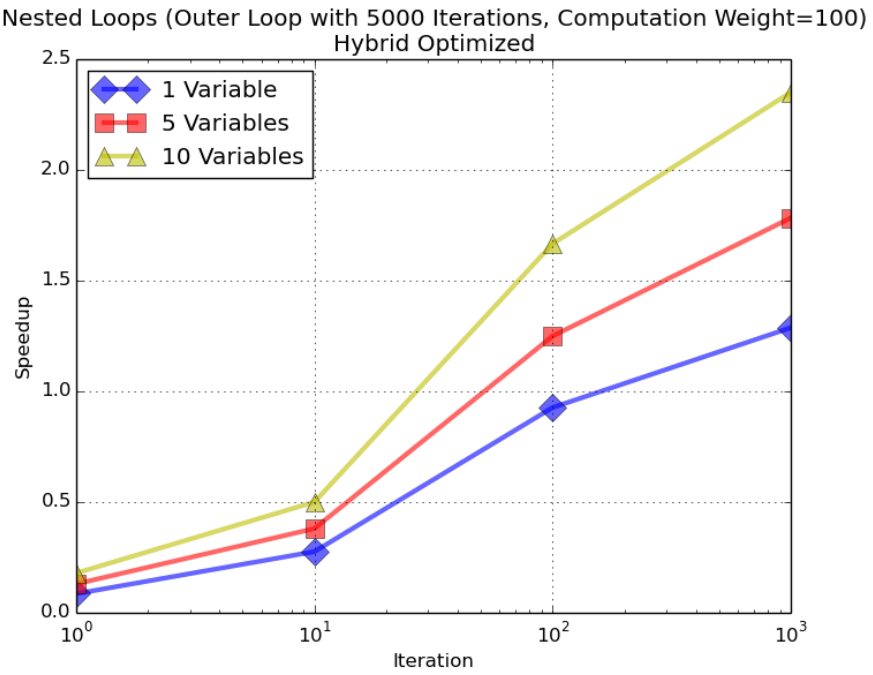

Figura 6.21: Desempenho do *Stack-Tagged Dataflow* híbrido e otimizado em *loops* aninhados. O eixo x mostra a variação dos números de iterações do *loop* interno e o eixo y o *speedup* alcançado em relação a versão original. Cada linha mostra o resultado dos cenários com quantidade diferentes de *skip variables*(1, 5 e 10) com o peso fixo em 100.

na versão com pilha. Entretanto, o *Stack-Tagged Dataflow* alcança melhor desempenho superando a versão original quando o número de iterações do *loop* interno cresce. As versões otimizadas da pilha obtêm resultados semelhantes a pilha sem otimização, tornando-as alternativas viáveis para implementação da técnica. Uma sugestão de melhoria é permitir que o compilador use instruções de **Push** e **Pop** em *loops* aninhados apenas quando o número de iterações do *loop* interno alcançar um limiar desejado com base no número de iterações do *loop* externo.

6.6 Experimento 6

O objetivo do sexto experimento é avaliar como o *Tag Resetting* e o *Loop Skipping* podem contribuir para reduzir o *overhead* do *Stack-Tagged Dataflow* em *loops* aninhados. Para isso, foi implementado uma aplicação sintética com dois *loops* aninhados com o comando `for`. Os parâmetros de entrada são o número de iterações do *loop* interno e o peso de computação do *loop* interno. A aplicação tem 10 *skip variables* fixas: sendo 5 do *loop* externo e as outras 5 do *loop* interno (essas variáveis entram no *loop* externo). O número de iterações do *loop* externo foi fixado em 5000, enquanto o *loop* interno executa de 1 a 1000 iterações. Cada cenário foi executado

10 vezes com desvio padrão irrelevante.

	***Loop* Interno**							***Loop* Externo**						
	Push	**Pop**	**IncTagI**	**IncTagN**	**PredTag**	**JoinTag**	**zTag**	**Push**	**Pop**	**IncTagI**	**IncTagN**	**PredTag**	**JoinTag**	**zTag**
Stack Tag	3	2	-	-	-	-	-	8	6	-	-	-	-	-
Loop Skipping I	-	-	6	-	-	-	-	-	-	5	-	-	-	-
Loop Skipping N	-	-	-	8	-	-	-	-	-	-	5	2	1	-
Tag Resetting	3	2	-	-	-	-	-	-	-	-	-	-	-	6

Tabela 6.3: Número de instruções de controle para o sexto experimento.

Para melhor analisar o custo das instruções de controle relacionadas com cada técnica, foram contadas cada uma das instruções observando o código de montagem de cada versão. A Tabela 6.3 mostra o número de instruções **Push**, **Pop**, **IncTagI**, **IncTagN**, **PredTag**, **JoinTag** e **zTag** encontradas nos *loops* interno e externo. Note que cada instrução de controle do *loop* interno será executada 5 mil vezes, pois este é o número de iterações fixado no *loop* externo.

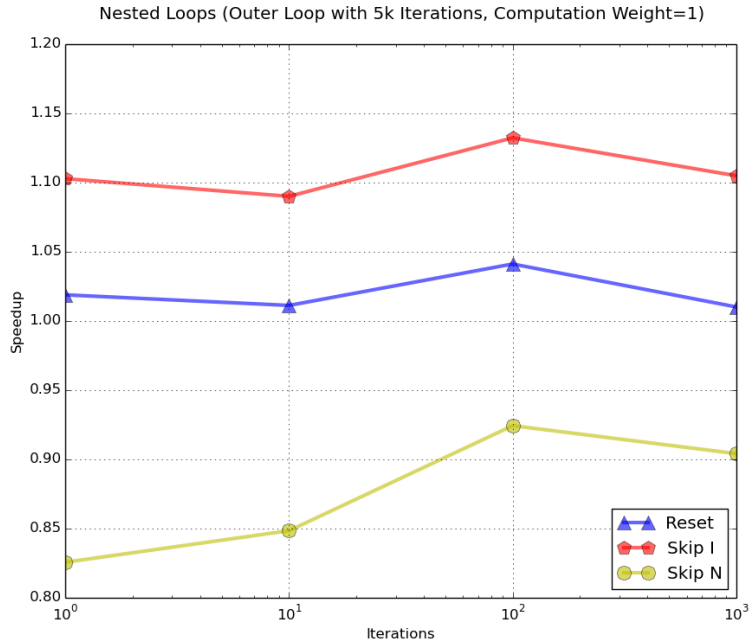

Figura 6.22: Desempenho do *Tag Resetting* e *Loop Skipping* em *loops* aninhados com peso de computação 1. O eixo x mostra a variação dos números de iterações do *loop* interno e o eixo y o *speedup* alcançado em relação a versão com a técnica *Stack-Tagged Dataflow*.

As figuras 6.22, 6.23 e 6.24 mostram os *speedups* do *Tag Resetting* e as duas variações do *Loop Skipping* sobre o *Stack-Tagged Dataflow* sem otimização (*baseline*).

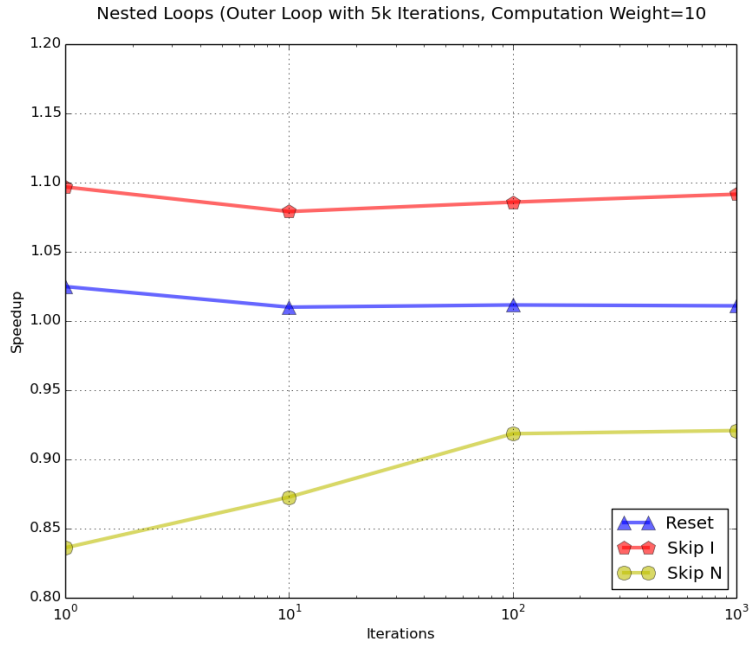

Figura 6.23: Desempenho do *Tag Resetting* e *Loop Skipping* em *loops* aninhados com peso de computação 10. O eixo x mostra a variação dos números de iterações do *loop* interno e o eixo y o *speedup* alcançado em relação a versão com a técnica *Stack-Tagged Dataflow*.

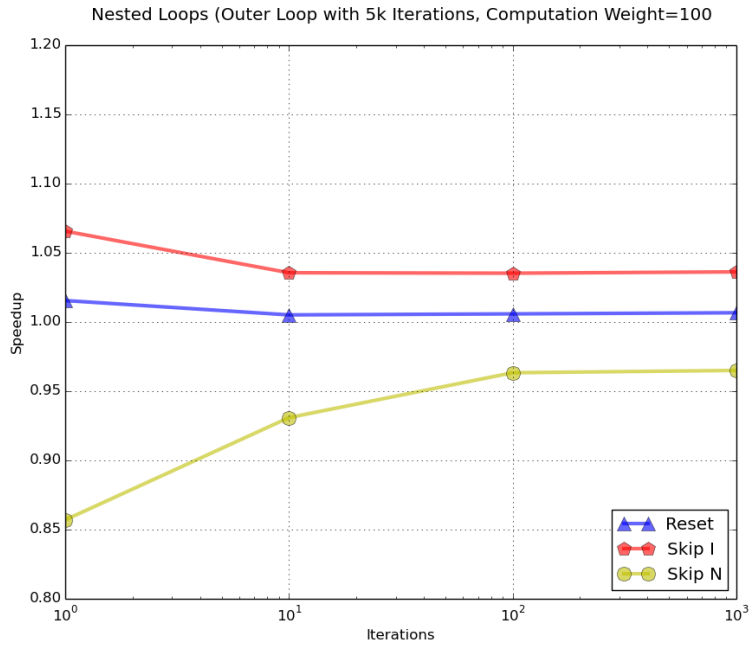

Figura 6.24: Desempenho do *Tag Resetting* e *Loop Skipping* em *loops* aninhados com peso de computação 100. O eixo x mostra a variação dos números de iterações do *loop* interno e o eixo y o *speedup* alcançado em relação a versão com a técnica *Stack-Tagged Dataflow*.

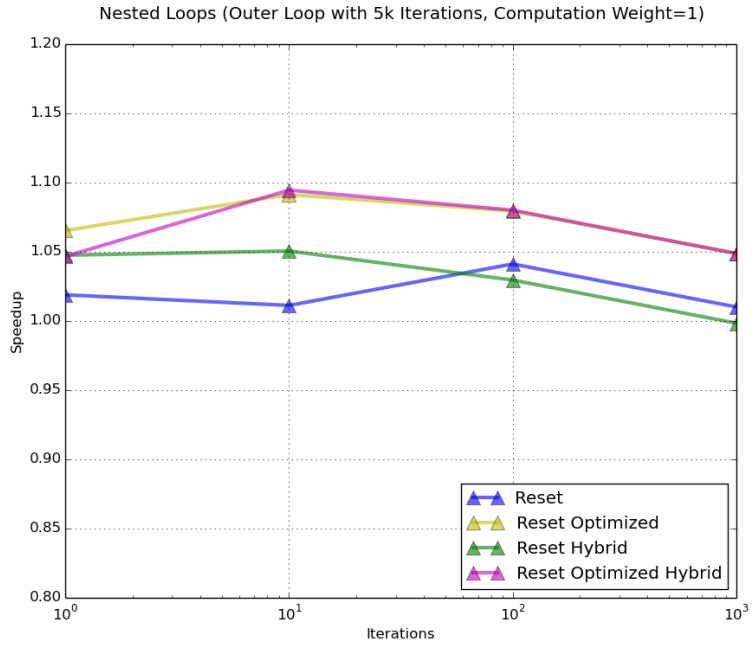

Figura 6.25: Desempenho do *Tag Resetting* com as otimizações do *Stack-Tagged Dataflow* em *loops* aninhados com peso de computação 1. O eixo x mostra a variação dos números de iterações do *loop* interno e o eixo y o *speedup* alcançado em relação a versão com a técnica *Stack-Tagged Dataflow*.

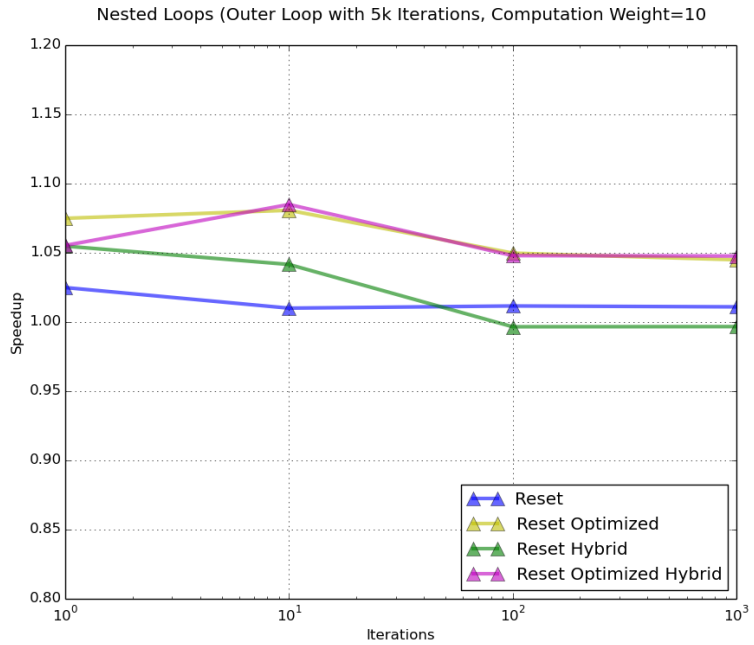

Figura 6.26: Desempenho do *Tag Resetting* com as otimizações do *Stack-Tagged Dataflow* em *loops* aninhados com peso de computação 10. O eixo x mostra a variação dos números de iterações do *loop* interno e o eixo y o *speedup* alcançado em relação a versão com a técnica *Stack-Tagged Dataflow*.

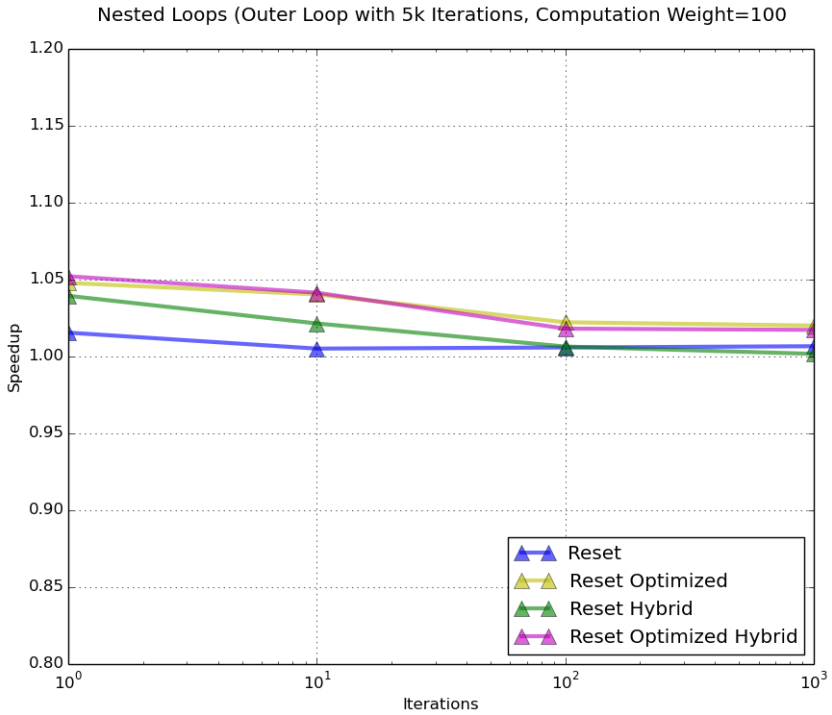

Figura 6.27: Desempenho do *Tag Resetting* com as otimizações do *Stack-Tagged Dataflow* em *loops* aninhados com peso de computação 100. O eixo x mostra a variação dos números de iterações do *loop* interno e o eixo y o *speedup* alcançado em relação a versão com a técnica *Stack-Tagged Dataflow*.

Cada linha mostra o resultado dos cenários com as versões: *Tag Resetting*, *Loop Skipping* com **IncTagI** e *Loop Skipping* com **IncTagN**. O *Tag Resetting* não apresentou melhoria no desempenho ($\approx 2\%$). Esse fato era esperado, pois esta técnica modifica somente o controle de *loops* externos, removendo as instruções de **Push** e substituindo as instruções de **Pop** pelas instruções **zTag**.

O *Loop Skipping* com **IncTagI** forneceu uma melhoria de $\approx 10\%$ sobre o *Stack-Tagged Dataflow*. Como nesta aplicação o maior *overhead* vem das instruções de **Push** e **Pop** no *loop* interno, o *Loop Skipping* remove tais instruções de ambos *loops* e adiciona as instruções **IncTagI** para todas as *skip variables*.

O *Loop Skipping* com **IncTagN** piorou o desempenho sobre o *Stack-Tagged Dataflow* em $\approx 14\%$. Apesar da técnica executar os cálculos das instruções *PredTag* e *JoinTag* antes de entrar no *loop* externo, o operando resultante, que contém o número de iterações do *loop* interno obtido em tempo de execução, precisa ser enviado para as instruções de controle **IncTag** e **Steer** para entrar no *loop* externo. Assim, seu rótulo será atualizado e poderá executar a instrução **IncTagN** nas *skip variables* do *loop* interno. Conforme o aumento do peso de computação, a contribuição de cada técnica se torna menos relevante já que o custo de computação do *loop* interno aumenta. Para o peso de computação 1, o *Tag Resetting* e *Loop Skipping* com **IncTagI** forneceu *speedups* de aproximadamente 10% e 2% respectivamente, e o *Loop Skipping* com **IncTagN** piorou em aproximadamente 14%. Para o peso de computação 100, o *Tag Resetting* e *Loop Skipping* com **IncTagI** forneceu *speedups* de aproximadamente 5% e 0.9% respectivamente, e o *Loop Skipping* com **IncTagN** piorou em aproximadamente 7%. Visto que o principal objetivo é permitir a execução de tarefas com grãos mais finos, os resultados mostraram um bom potencial com exceção do *Loop Skipping* com **IncTagN**.

Como a aplicação do experimento com o *Tag Resetting* também utiliza instruções de pilha no *loop* interno, essa técnica foi processada nas diferentes variações do *Stack-Tagged Dataflow*. As figuras 6.25, 6.26 e 6.27 mostram os *speedups* do *Tag Resetting* com as otimizações de pilha sobre o *Stack-Tagged Dataflow* sem otimização (*baseline*). Cada linha mostra o resultado dos cenários com as versões: *Tag Resetting* sem otimização do casamento de pilha, *Tag Resetting* com otimização do casamento de pilha, *Tag Resetting* na Trebuchet híbrida sem otimização do casamento de pilha e *Tag Resetting* na Trebuchet híbrida com otimização do casamento de pilha. Para o peso de computação 1, a melhor configuração do *Tag Resetting* foi com otimização do casamento do *Stack-Tagged Dataflow* obtendo *speedup* de aproximadamente 7%. Para o peso de computação 100, a melhor configuração do *Tag Resetting* foi com otimização do casamento do *Stack-Tagged Dataflow* obtendo *speedup* de aproximadamente 4%.

A Tabela 6.3 mostra o percentual de desempenho obtido em cada cenário em

	Peso 1	**Peso** 10	**Peso** 100
Tag Resetting	$1,98\%$	$1,41\%$	$0,82\%$
Tag Resetting Otimizado	$6,64\%$	$5,87\%$	$3,15\%$
Tag Resetting Híbrido	3%	$2,13\%$	$1,68\%$
Tag Resetting Híbrido Otimizado	$6,29\%$	$5,55\%$	$3,1\%$
Loop Skipping I	$9,69\%$	$8,12\%$	$4,12\%$
Loop Skipping N	$-14,44\%$	$-12,9\%$	-7.88%

Tabela 6.4: Percentual de desempenho para o sexto experimento.

relação o *Stack-Tagged Dataflow* sem otimização. As otimizações do *Stack-Tagged Dataflow* contribuiu com a técnica *Tag Resetting* melhorando seu desempenho. Tais otimizações também mostram um bom potencial na execução de tarefas com granularidade fina.

Capítulo 7

Conclusões e Trabalhos Futuros

Este trabalho apresentou um conjunto de técnicas que permite otimizações de *loops* numa computação baseada no *dataflow* dinâmico: *Stack-Tagged Dataflow*, *Tag Resetting* e *Loop Skipping*.

Stack-Tagged Dataflow é um mecanismo de rotulação que usa pilha, ao invés de um simples inteiro, para reduzir o *overhead* de controle no *dataflow* evitando instruções **Steer** e **IncTag** desnecessárias para os operandos que são produzidos antes de *loops* e usados somente depois dele (chamadas de *skip variables*). A pilha de rótulos é manipulada pelas instruções **Push** e **Pop** e cada posição da pilha armazena o contexto de um *loop*, representando sua iteração na execução corrente da aplicação. Antes de um operando ser enviado para um *loop*, ele é encaminhado a uma instrução de **Push**, que incluirá um novo rótulo no topo da pilha com o valor zero. Ao fim do *loop*, um operando é enviado a uma instrução de **Pop** responsável por desempilhar e descartar o topo da pilha restaurando a pilha de rótulos com seu valor original. As *skip variables* são enviadas diretamente as suas instruções de destino passando por fora do *loop*. Os resultados mostram que a técnica *Stack-Tagged Dataflow* é viável e proporciona ganho de desempenho conforme o aumento do número de iterações de laços e *skip variables*. Em *loops* aninhados, a técnica apresenta degradação no desempenho quando a quantidade de iterações de um *loop* interno é menor em relação ao *loop* externo. Nesse cenário, poderia-se evitar tal degradação se o compilador fosse capaz de escolher quando utilizar ou não as instruções de **Push** e **Pop**. Para melhorar o desempenho dessa técnica, o casamento de pilha foi otimizado com o uso de pilhas de tamanho fixo nos operandos, sendo possível realizar a comparação de todos os elementos em uma única operação condicional. Também foi desenvolvido uma mecanismo de casamento híbrido na máquina virtual permitindo alternar entre o processo de casamento de pilha e inteiros. Tais otimizações contribuíram com a redução de *overhead* da técnica.

Tag Resetting é uma técnica que zera os rótulos dos operandos quando eles alcançam um ponto seguro na aplicação. O ponto seguro é ao final de um *loop*

externo. Assim, esse mecanismo pode reduzir o *overhead* do *Stack-Tagged Dataflow* em *loops* externos, evitando o uso de instruções **Push** e **Pop** nos operandos que passam por dentro do *loop*. Em aplicações com dois *loops* aninhados foi possível obter *speedups* de 2% na versão sem otimização do *Stack-Tagged Dataflow* e 7% na versão com otimização. Acreditamos que os ganhos dessa técnica podem ser mais significativos para aplicações com aninhamento de laços mais profundo.

Loop Skipping é uma técnica aplicada quando o compilador é capaz de determinar a quantidade de iterações de um *loop*. As *skip variables* são enviadas para instruções **IncTagI** ou **IncTagN** que são responsáveis por incrementar seus rótulos pelo número de iterações previsto em tempo de compilação ou tempo de execução, respectivamente. Desse modo, é possível reduzir o *overhead* do *Stack-Tagged Dataflow* evitando o uso de instruções **Push** e **Pop** nos operandos que passam por dentro do *loop*. Em uma aplicação sintética com dois *loops* aninhados, foi possível obter *speedups* de 5% a 10%, mostrando que esta técnica pode ser eficiente com o uso de instruções **IncTagI**. Já com o uso de instruções **IncTagN**, seu benefício é obtido em aplicações com um *loop* simples. Em aplicações com *loops* aninhados, acreditamos que está técnica se beneficiará no cenário em que poucas *skip variables* serão necessárias em *loops* internos com um peso de computação ajustado para esconder o *overhead* dos controles impostos pelo método. A investigação sobre as melhores condições de usar o **IncTagN** será tratada em um trabalho futuro.

O próximo passo para essa pesquisa é avaliar as técnicas tratadas neste trabalho em um conjunto de *bechmarks* reais. Isto permitiria obter informações do nível de profundidade de aninhamento, números de iterações, pesos de computação e testar a solução com múltiplas *threads* para avaliar o efeito de cada técnica no caminho crítico das aplicações. Além disso, é desejável que compilador abranja mais situações para a aplicação do *Loop Skipping*, tais como *loops* com o comando `while` e `do while`.

Referências Bibliográficas

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

[2] BORKAR, S., CHIEN, A. A. "The Future of Microprocessors", *Commun. ACM*, v. 54, n. 5, pp. 67–77, maio 2011. ISSN: 0001-0782. doi: 10.1145/1941487.1941507. Disponível em: <http://doi.acm.org/10.1145/1941487.1941507>.

[3] JOHNSTON, W. M., HANNA, J. R. P., MILLAR, R. J. "Advances in Dataflow Programming Languages", *ACM Comput. Surv.*, v. 36, n. 1, pp. 1–34, mar. 2004. ISSN: 0360-0300. doi: 10.1145/1013208.1013209. Disponível em: <http://doi.acm.org/10.1145/1013208.1013209>.

[4] MARZULO, L. A. J. *Explorando Linhas de Execução Paralelas com Programação Orientada por Fluxo de Dados*. Tese de Doutorado, COPPE - UFRJ, out. 2011.

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

[6] DENNIS, J. B., FOSSEEN, J. B. *Introduction to Data Flow Schemas*. Technical Memorandum Memo-81-1, Laboratory for Computer Science, Massachusetts Institute of Technology, September 1973.

[7] DENNIS, J. B. "The Varieties of Data Flow Computers". In: *Distributed Computing Systems (ICDCS), 1979 IEEE 1st International Conference on*, out. 1979.

[8] 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. IEEE Comput. Soc, 2003. ISBN: 0-7695-2043-X. doi: 10.1109/MICRO.2003.1253203.

[9] LEE, B., HURSON, A. R. "Issues in dataflow computing", *ADV. IN COMPUT*, v. 37, pp. 285–333, 1993.

[10] VEEN, A. H. "Dataflow Machine Architecture", *ACM Comput. Surv.*, v. 18, n. 4, pp. 365–396, dez. 1986. ISSN: 0360-0300. doi: 10.1145/27633.28055. Disponível em: <http://doi.acm.org/10.1145/27633.28055>.

[11] DENNIS, J. B., MISUNAS, D. P. "A Preliminary Architecture for a Basic Dataflow Processor", *SIGARCH Comput. Archit. News*, v. 3, n. 4, pp. 126–132, dez. 1974. ISSN: 0163-5964. doi: 10.1145/641675.642111. Disponível em: <http://doi.acm.org/10.1145/641675.642111>.

[12] ARVIND, CULLER, D. E. "Annual Review of Computer Science Vol. 1, 1986". Annual Reviews Inc., cap. Dataflow Architectures, pp. 225–253, Palo Alto, CA, USA, 1986. ISBN: 0-8243-3201-6. Disponível em: <http://dl.acm.org/citation.cfm?id=17814.17824>.

[13] ARVIND, CULLER, D. E. *Tagged-Token Dataflow Architecture*. Technical Memorandum Memo-229, Laboratory for Computer Science, Massachusetts Institute of Technology, July 1983.

[14] MAGNA, P. *Proposta e simulação de uma arquitetura a fluxo de dados de segunda geração*. Tese de Doutorado, Instituto de Física de São Carlos - USP, mar. 1997.

[15] GRAFE, V. G., DAVIDSON, G. S., HOCH, J. E., et al. "The Epsilon Dataflow Processor", *SIGARCH Comput. Archit. News*, v. 17, n. 3, pp. 36–45, abr. 1989. ISSN: 0163-5964. doi: 10.1145/74926.74930. Disponível em: <http://doi.acm.org/10.1145/74926.74930>.

[16] PAPADOPOULOS, G., CULLER, D. "Monsoon: an explicit token-store architecture". In: *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*, pp. 82–91, May 1990. doi: 10.1109/ISCA.1990.134511.

[17] SWANSON, S., SCHWERIN, A., MERCALDI, M., et al. "The WaveScalar Architecture", *ACM Trans. Comput. Syst.*, v. 25, n. 2, pp. 4:1–4:54, maio 2007. ISSN: 0734-2071. doi: 10.1145/1233307.1233308. Disponível em: <http://doi.acm.org/10.1145/1233307.1233308>.

[18] GURD, J. R., KIRKHAM, C. C., WATSON, I. "The Manchester Prototype Dataflow Computer", *Commun. ACM*, v. 28, n. 1, pp. 34–52, jan. 1985. ISSN: 0001-0782. doi: 10.1145/2465.2468. Disponível em: <http://doi.acm.org/10.1145/2465.2468>.

[19] STERLING, T., KUEHN, J., THISTLE, M., et al. "Studies on optimal task granularity and random mapping", *Advanced Topics in Dataflow Computing and Multithreading*, pp. 349–365, 1995.

[20] TOMASULO, R. M. "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", *IBM J. Res. Dev.*, v. 11, n. 1, pp. 25–33, jan. 1967. ISSN: 0018-8646. doi: 10.1147/rd.111.0025. Disponível em: <`http://dx.doi.org/10.1147/rd.111.0025`>.

[21] "Graphviz web-site". `http://www.graphviz.org`.

[22] BEAZLEY, D. "PLY - Python Lex-Yacc". `http://www.dabeaz.com/ply/`.

[23] SANTIAGO, L., MARZULO, L., GOLDSTEIN, B., et al. "Stack-Tagged Dataflow". In: *Computer Architecture and High Performance Computing Workshop (SBAC-PADW), 2014 International Symposium on*, pp. 78–83, Oct 2014. doi: 10.1109/SBAC-PADW.2014.21.

[24] 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.