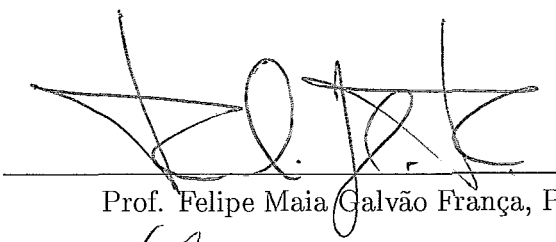
GANHOS POTENCIAIS DA MIGRAÇÃO DINÂMICA DE TRACES PARA INSTRUÇÕES DO TIPO CISC EM MICROARQUITETURAS PIPELINED

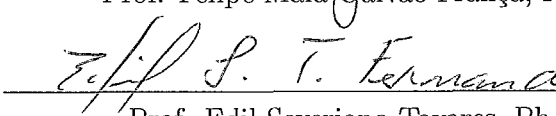
Pedro Henrique Rausch Bello

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

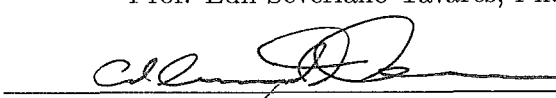
Aprovada por:



Prof. Felipe Maia Galvão França, Ph.D.

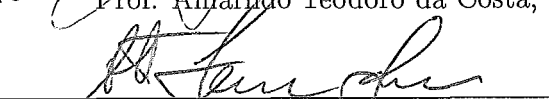


Prof. Edil Severiano Tavares, Ph.D.



Prof. Carlo Emmanoel Tolla de Oliveira, Ph.D.

Prof. Amarildo Teodoro da Costa, D.Sc.



Prof. Alberto Ferreira de Souza, Ph.D.

RIO DE JANEIRO, RJ - BRASIL

MARÇO DE 2004

BELLO, PEDRO HENRIQUE RAUSCH

Ganhos Potenciais da Migração Dinâmica de Traces para Instruções do Tipo CISC em Microarquiteturas Pipelined [Rio de Janeiro] 2004

XII, 45 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2004)

Tese – Universidade Federal do Rio de Janeiro, COPPE

1 - Migração Dinâmica de Primitivas

2 - Microarquiteturas pipelined

I. COPPE/UFRJ II. Título (série)

Ao meu avô Paulo, que gostaria de presenciar este momento.

Agradecimentos

Em primeiro lugar, agradeço a minha mulher Silvia pelo apoio dado em todos os momentos deste trabalho, e por continuar sendo a pessoa com quem me casei e com quem desejo passar o resto da minha vida.

Ao meu filho Felipe, que não reclamou ao ser privado da companhia do pai nos últimos meses.

A toda a família, em especial a minha mãe Diva, que sempre me mostrou a importância do estudo e do aprimoramento intelectual, e pela revisão do texto final.

Ao amigo Diego Carvalho, pelas discussões frutíferas e pelas infrutíferas também, e pelos planos de dominação do mundo. Se pudéssemos escolher nossos irmãos, você seria um deles.

A Bárbara Hansen por ser quem é, minha amiga mais querida.

Ao orientador e desorientador Felipe Maia Galvão França, pelas inúmeras lições, apoio durante todo o mestrado, pelo trabalho de psicanalista, e por nunca ter perdido a paciência com minha desorganização.

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

GANHOS POTENCIAIS DA MIGRAÇÃO DINÂMICA DE TRACES PARA INSTRUÇÕES DO TIPO CISC EM MICROARQUITETURAS PIPELINED

Pedro Henrique Rausch Bello

Março/2004

Orientador: Felipe Maia Galvão França

Programa: Engenharia de Sistemas e Computação

O presente trabalho explora o conceito de *Migração Dinâmica de Traces*, um novo mecanismo que consiste em transformar sequências dinâmicas (traços) de instruções simples em instruções complexas durante a execução de um programa por um processador superescalar pipelined. A seleção de traces foi baseada no mecanismo de reuso *DTM*. Para medir o potencial de ganho com a migração dinâmica de traces são investigadas duas alternativas principais: (i) Uma alternativa "otimista", onde as novas instruções complexas são executadas em unidades funcionais especiais, no mesmo tempo de execução de uma instrução base do processador alvo; (ii) Uma alternativa "pessimista", onde as instruções complexas são executadas nas mesmas unidades funcionais, mas com o tempo de execução seqüencial das instruções originais do processador alvo. Foram realizados experimentos utilizando um subconjunto dos programas do *SPECINT2000* obtendo-se um *speedup* de 1,19 para a alternativa "otimista", e de 0,94 para a alternativa "pessimista".

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

POTENTIAL GAINS OF DYNAMIC TRACE MIGRATION TO CISC TYPE INSTRUCTIONS IN PIPELINED MICROARCHITECTURES

Pedro Henrique Rausch Bello

March/2004

Advisor: Felipe Maia Galvão França

Department: Computing and Systems Engineering

The present work explores the concept of *Dynamic Trace Migration*, a novel mechanism designed to transform dynamic sequences of simple instructions (traces) into complex ones during execution of a program in a pipelined superscalar processor. The selection of traces worth of migration is based on the reuse mechanism called *DTM*. In order to measure the potential gain from the mechanism two alternatives are investigated: (i) an optimistic approach, where new complex instructions are executed in a special unit in the same time as the one taken by an original instruction; (ii) a pessimistic approach, where the complex instructions still are executed in a special unit, except that their execution time is the same as if the instructions were executed sequentially. Experiments with a subset of the SPECINT2000 benchmark suite show a speedup of 1.19 for the optimistic approach, and of 0,94 for the pessimistic approach.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A demanda por processadores mais poderosos cresce continuamente, fruto da ubiqüidade dos sistemas de computadores em nossa sociedade. Existem duas formas de se aumentar a capacidade de processamento de um processador:

- Mudanças na tecnologia utilizada, resultando em frequências de *clock* cada vez mais altas, reduzindo o tempo de execução das instruções;
- Mudanças na arquitetura dos processadores, introduzindo técnicas que permitem a execução de mais instruções em um mesmo ciclo de *clock*.

A busca por ciclos de *clock* cada vez menores contribuiu para a disseminação da idéia de que arquiteturas *RISC - Reduced Instruction Set Computer*, ao manter a estrutura interna do processador conceitualmente o mais simples possível seriam as únicas capazes de oferecer escalabilidade com o aumento da frequência de *clock*. As arquiteturas *CISC - Complex Instruction Set Computer* estariam destinadas ao fracasso.

A idéia de que arquiteturas RISC são inerentemente melhores por causa de sua aparente simplicidade permeou a indústria a ponto de se tornar uma verdade incontestável. Mesmo os competidores aceitam este fato sem discussão. Somente recentemente [17] começaram a aparecer críticas ao modelo, e a "Penalidade *RISC*" veio à tona: devido à menor localidade do código, arquiteturas *RISC* tendem a sofrer

com invalidações constantes nas memórias *cache* de instruções e de dados, tornando o acesso à memória mais custoso do que para arquiteturas *CISC*.

O trabalho desenvolvido aqui visa determinar o potencial de ganho de performance em arquiteturas RISC superescalares onde, baseado em um mecanismo para memorização dinânica de traços, instruções do tipo CISC são identificadas e utilizadas durante o tempo de execução de um programa. As instruções complexas geradas são executadas em uma unidade funcional especial de acordo com duas alternativas de execução propostas.

1.1 Histórico

A migração vertical de primitivas surgiu pela primeira vez em sistemas microprogramados [14] como uma maneira de aumentar a performance destes. A idéia inicial era transformar as primitivas selecionadas (instruções, sequências de instruções ou até mesmo funções inteiras) em uma representação de mais baixo nível dentro de uma arquitetura específica.

Trabalhos mais recentes neste campo são voltados para processadores embarcados [1] [3] [23], onde restrições de custo impõe limitações ao tamanho do código executado, ou máquinas virtuais de baixo nível [11] [13]. Nesses casos, parte das primitivas é sintetizada no hardware do processador, ou existe algum tipo de co-processador reconfigurável acoplado ao processador principal capaz de se adequar à execução de diferentes programas.

1.1.1 Classificação dos esquemas de migração de primitivas

A migração vertical de primitivas envolve três etapas distintas: *etapa de análise*, onde são identificados e selecionados os possíveis candidatos à migração; *etapa de síntese*, onde a migração é realizada e *etapa de carga*, onde as primitivas migradas são carregadas para posterior utilização [10].

Os esquemas de migração de primitivas podem ser classificados de acordo com a Tabela 1.1 [19]:

		Síntese	
		Estático	Dinâmico
Carga	Estático	S/S	—
	Dinâmico	S/D	D/D

Tabela 1.1: Classificação dos esquemas de migração vertical de primitivas

- Esquema S/S: As primitivas migradas são escolhidas estaticamente através de análise do comportamento dinâmico dos candidatos, e carregadas no processador antes do princípio da execução do programa propriamente dito [11] [13] [1].

- Esquema S/D: As primitivas migradas são escolhidas estaticamente através de análise do comportamento dinâmico dos candidatos, porém somente são carregadas em tempo de execução, à medida em que se fazem necessárias [8] [23].

- Esquema D/D: As primitivas são identificadas e sintetizadas durante a execução do programa, ficando disponíveis imediatamente para utilização se necessário [15].

1.2 Motivação e Objetivos

Novos métodos para aumentar a performance de processadores superescalares são necessários para trazer inovações ao campo da microarquitetura. A motivação para o trabalho atual é a avaliação do potencial de uma nova classe de mecanimos baseada na migração de instruções simples do tipo RISC para traços representando instruções complexas do tipo CISC como alternativa para aumentar a performance de um procesador superescalar substrato.

O trabalho atual pretende atingir os seguintes objetivos:

- Demonstrar a validade da migração dinâmica de traços como forma de aumentar o desempenho de arquiteturas superescalares;
- Propor um esquema D/D de migração utilizando o mecanismo de reuso *DTM - Dynamic Trace Memoization*, descrito no Capítulo 3, para seleção de traços;
- Avaliar o potencial do mecanismo através de duas alternativas de utilização das instruções *CISC* geradas, uma pessimista e outra otimista;
- Avaliar se o mecanismo tem efeito mitigador sobre a "Penalidade RISC´´ descrita nas seções anteriores;

O mecanismo proposto foi validado através de simulações, e foi constatado um significativo ganho de performance para a alternativa otimista. Também foi observado que a alternativa otimista pode mitigar a "Penalidade *RISC*", através da diminuição da quantidade de acessos à memória cache de instruções.

Os objetivos propostos foram atingidos. Foi proposto um mecanismo de migração utilizando como critério de seleção o mecanismo DTM, e este mecanismo foi validado segundo alternativas otimista e pessimista.

O resultado das simulações indica um *speedup* (aceleração) de 19% (média harmônica entre os resultados obtidos para cada um dos programas executados) para a alternativa otimista, e uma perda de performance de 5% para a alternativa pessimista. Estes resultados serão discutidos no Capítulo 5.

A perda de performance observada na alternativa pessimista é devido ao fato de que a alternativa pessimista reduz o processador superescalar substrato à execução sequencial das instruções simples. Foram então realizados experimentos com um processador base escalar e mesmo a alternativa pessimista apresentou ganhos de performance da ordem de 15%

1.3 Estrutura do Trabalho

Este trabalho segue-se apresentado na seguinte estrutura. No segundo capítulo são apresentados trabalhos relacionados à migração dinâmica de primitivas em geral, e o estado da arte na área. No terceiro capítulo é feita uma introdução ao mecanismo de reuso *DTM*, responsável pela seleção dos traços que serão migrados para instruções complexas. No quarto capítulo é apresentado o mecanismo de migração de traços, e as duas alternativas propostas para a avaliação do seu potencial. No quinto capítulo é apresentada a base experimental e no sexto capítulo é feita uma análise dos resultados obtidos. O sétimo capítulo apresenta as conclusões atingidas e trabalhos futuros derivados desta pesquisa.

Capítulo 2

Trabalhos Relacionados

Neste capítulo apresentamos alguns trabalhos relacionados à migração dinâmica de primitivas em geral.

2.1 O coprocessador para migração dinâmica vertical

O coprocessador para migração dinâmica vertical [15] implementa um esquema de migração do tipo D/D em uma arquitetura microprogramada. O mecanismo funciona como um cache, armazenando o microcódigo gerado durante a execução de instruções dinâmicas e reutilizando-o caso a instrução volte a ser executada em seguida.

O mecanismo está fora do caminho crítico do processador, atuando em paralelo à decodificação e execução das instruções. A etapa de geração de microcódigo é substituída por um mecanismo de interconexão entre o código de máquina e o microcódigo das instruções. Durante a execução do programa, toda vez que ocorre a busca de uma instrução é verificada sua presença em uma memória de interconexão, que armazena os parâmetros de execução do microcódigo da instrução, eliminando os estágios de busca e decodificação, acelerando a execução dos programas

A estrutura do coprocessador é constituída pelas seguintes unidades:

- Memória de Interconexão (I.M.) — armazena as interconexões relativas a uma

janela de código do programa na memória principal. Esta janela se movimenta ao longo da execução do programa, e é fisicamente dividida em blocos;

- Unidade de gerenciamento da memória de interconexão (I.M.M.U) – Atualiza a I.M. e gerencia o mapeamento entre páginas da memória principal e blocos da I.M.;

- Unidade de controle do coprocessador (C.C.U.) — Contém o microcódigo responsável pelo gerenciamento do coprocessador e uma versão modificada do microcódigo do conjunto de instruções do processador.

Durante o estágio de busca, as instruções são procuradas tanto na memória principal quanto na I.M. Quando a entrada é encontrada na I.M., a informação armazenada nesta é reutilizada e o processador pode passar para a busca da próxima instrução. Caso a instrução não se encontre na I.M., o coprocessador seleciona uma janela na I.M. e escreve o padrão de interconexão na I.M. quando este é disponibilizado pelo estágio de decodificação. Em seguida a instrução é executada normalmente e o fluxo de execução prossegue.

No trabalho original foi analisado o desempenho do mecanismo para estruturas de laço, e constatado que o mecanismo tinha um bom desempenho quando o tamanho do laço é menor do que o tamanho da I.M.

2.2 MLM – Multi-Lingual Machine

A MLM, é uma arquitetura microprogramável onde a programação da unidade de controle pode ser alterada de acordo com o código a ser executado no momento, oferecendo suporte especializado para linguagens de alto nível. Pode ser caracterizada como um esquema de migração S/D, uma vez que os microprogramas alternativos são gerados estaticamente, podendo contudo ser trocados durante a execução de um programa.

A MLM foi utilizada para a implementação de uma versão microprogramada para uma máquina virtual EDISON [21], com a comunicação entre o processador base e a máquina virtual implementada através de uma mistura de firmware e software. Esta implementação demonstrou a viabilidade de se modificar a estrutura e funcionamento de uma arquitetura de forma a suportar primitivas complexas não implementadas pelo processador base.

2.3 DISC – Dynamic Instruction Set Computer

O DISC [23] foi criado para suportar modificações no conjunto de instruções sob demanda da aplicação. É uma esquema de migração S/D, onde as primitivas são selecionadas estaticamente e migradas para um circuito implementado em uma FPGA.

O sistema apresenta um número de módulos de propósito geral, como instruções lógico-aritméticas e comparadores, que podem ser misturados livremente com módulos sintetizados sob medida para cada aplicação.

Os módulos utilizados são realocáveis graças à implementação de um espaço linear para sua alocação. A grade bidimensional de células configuráveis da FPGA é organizada como um vetor de linhas: a localização de um módulo é dada por sua posição vertical no vetor e seu tamanho é dado pelo número de linhas que ele ocupa.

Para gerenciar a comunicação entre módulos e entre o processador e a memória foi criado um controlador global, implementado na própria FPGA mas sem possibilidade de relocação. Este controlador é responsável pela execução de instruções de desvio e de acesso à memória. Existe ainda um outro processador externo, implementado em outra FPGA, responsável pela carga de novos módulos sob demanda.

2.4 PRISM

A arquitetura PRISM [22] é um esquema de migração S/S constituído por um compilador C, capaz de gerar as estruturas necessárias para reconfigurar as FPGAs onde são executadas as instruções migradas.

O compilador implementa um subconjunto quase completo da linguagem C. O primeiro passo para implementar uma seqüência de instruções em hardware é montar o grafo de fluxo de dados decorrente da execução da seqüência de instruções sendo migrada. Em seguida, os tempos de execução de cada nó do grafo gerado são calculados, e é gerada uma máquina de estado responsável pelo gerenciamento da execução. O resultado é armazenado em uma *netlist* para ser carregado em uma das FPGAs responsáveis pela execução de instruções migradas.

2.5 Colapso de Instruções Dependentes

O trabalho desenvolvido em [11] consiste em uma máquina virtual de baixo nível que implementa o conjunto de instruções x86 através de um conunto de micro-operações RISC, e através de heurísticas colapsa micro-operações RISC dependentes em uma só, visando reduzir a complexidade no estágio de escalonamento. A instrução colapsada continua executando no tempo de execução das duas micro-operações originais. O tempo de execução maior permite que a implementação do escalonador seja pipelined de forma simples, permitindo o aumento da profundidade do pipeline.

Outras arquiteturas recentes como o AMD Opteron [6] e o Intel Pentium M [7] vão na direção deste trabalho, substituindo em alguns estágios do pipeline instruções simples por instruções complexas com mesmo valor semântico, com o intuito de reduzir o tráfego de instruções pelo pipeline, aumentando a performance e economizando energia.

Capítulo 3

Memorização de Traços – O mecanismo DTM

Conforme descrito no Capítulo 1, a migração dinâmica de primitivas ocorre em duas etapas:

- Seleção das primitivas a serem migradas;
- Utilização das primitivas migradas na execução do código.

O sucesso da migração dinâmica de traços depende em grande parte do mecanismo utilizado para realizar a primeira etapa do processo. Foi escolhido o mecanismo DTM para esta fase, por motivos explanados nas próximas seções.

3.1 O mecanismo DTM

O mecanismo intitulado *Dynamic Trace Memoization* - (DTM) desenvolvido em [5] [4] foi escolhido para realizar a etapa de seleção de primitivas a serem migradas.

O DTM utiliza duas tabelas de memorização para estender a seleção e o reuso de traços (seqüências) de instruções além das fronteiras de blocos básicos encontrados durante a execução de um programa.

Para entender melhor o funcionamento do mecanismo, faz-se necessária uma apresentação da terminologia utilizada:

- Uma instrução faz parte do *Domínio de validade* do DTM quando ela pertence a umas das seguintes classes: Operação Lógico-Aritmética, Instrução de desvio condicional ou incondicional e cálculo de endereço de uma instrução de memória. Somente instruções pertencentes ao domínio de validade do DTM podem ser inseridas na *Memo_ Table_ G* , e conseqüentemente, participar em um traço;

- O *contexto de entrada* de um traço é o conjunto de registradores e respectivos valores utilizados no interior do traço e produzidos por instruções externas a ele;

- O *contexto de saída* de um traço é o conjunto de registradores e respectivos valores produzidos por instruções pertencentes ao traço;

- *Instância de uma instrução dinâmica* é a instrução propriamente dita junto com seus operandos de entrada e seu resultado.

O uso de duas tabelas de memorização permite ao DTM atuar em dois níveis de reuso: o reuso de instruções e o reuso de traços de instruções dinâmicas. Instruções individuais são armazenadas na *Memo_ Table_ G* enquanto a *Memo_ Table_ T* armazena os traços identificados pelo mecanismo.

3.1.1 Estrutura da *Memo_ Table_ G*

A tabela *Memo_ Table_ G* é constituída de entradas com o formato apresentado na Figura 3.1.

O campo *pc* armazenam o endereço da instrução; os campos *sv1* e *sv2* armazenam o conteúdo dos registradores de entrada da instrução. O campo *restarg* armazena o resultado da operação lógico-aritmética ou o endereço alvo da instrução de desvio. Os campos *jmp* e *brc* identificam a instrução como um desvio incondicional ou

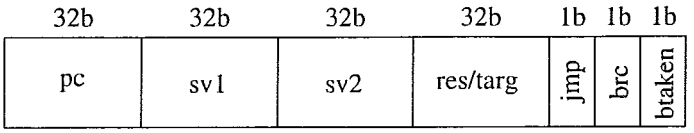


Figura 3.1: Uma entrada na *Memo_ Table_ G*. Os valores acima dos campos indicam o número de bits destes

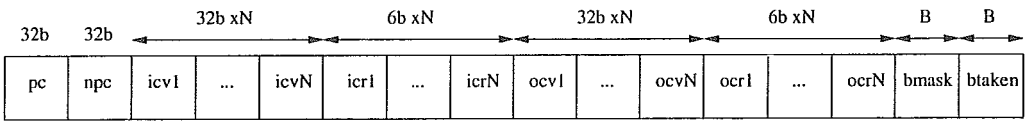


Figura 3.2: Uma entrada na *Memo_ Table_ T*. Os valores acima dos campos indicam o número de bits destes

condicional, respectivamente, e o campo *btaken* indica se o desvio foi tomado ou não.

3.1.2 Estrutura da *Memo_ Table_ T*

A tabela *Memo_ Table_ T* é constituída de entradas com o formato apresentado na Figura 3.2.

Os campos *pc* e *npc* indicam o endereço da instrução que inicia o traço (o endereço do traço) e o próximo endereço a ser buscado, após o trace. Os campos *icv1* a *icvN* armazenam os valores dos registradores que fazem parte do contexto de entrada do traço, enquanto que os campos *icr1* a *icrN* indicam quais são estes registradores. Os campos *ocv1* a *ocvN* e *ocr1*a *ocrN* fazem o mesmo pelo contexto de saída. Os campos *bmask* e *btaken* estão relacionados à presença de instruções de desvio no trace, e à tomada ou não destes desvios, respectivamente.

É importante ressaltar que o mecanismo DTM permite que um trace contenha múltiplas instruções de desvio, permitindo a construção de traços além das fronteiras dos blocos básicos encontrados durante a execução do programa.

3.1.3 Construção de traços

O DTM analisa todas as instruções dinâmicas, avaliando se elas pertencem ao domínio de validade. Se a instrução é inválida (não pertence ao domínio), o mecanismo a marca como *não-redundante* e não a insere na *Memo_ Table_ G* . Caso contrário, o mecanismo compara o endereço da instrução e os valores correntes dos operandos de entrada com as entradas da *Memo_ Table_ G* . Se não for encontrada nenhuma entrada com valores iguais, o mecanismo marca a instrução como não-redundante e cria uma nova instância na *Memo_ Table_ G* . Por outro lado, se for verificada a igualdade com alguma entrada da *Memo_ Table_ G* , a instrução é marcada como *redundante* e não é inserida na *Memo_ Table_ G* .

Se uma instrução estiver marcada como não-redundante, o mecanismo termina o trace em construção e o insere na *Memo_ Table_ T* . Caso contrário, o mecanismo atualiza o trace em construção ou inicia a construção de um novo trace.

Na Figura 3.2, N representa o tamanho máximo dos contextos de entrada e saída de um trace, e B representa o número máximo de instruções de desvio em um trace. O mecanismo DTM termina a construção de um trace quando um dos seguintes eventos ocorre:

- Uma instrução não-redundante é encontrada;
- o número de registradores em um dos contextos atingiu o limite N;
- o número máximo de instruções de desvio B foi atingido.

3.1.4 Reuso de instruções e traços

Para cada instrução dinâmica pertencente ao domínio de validade, o DTM faz buscas simultâneas em *Memo_ Table_ G* e *Memo_ Table_ T* . Se for encontrada uma instância válida em *Memo_ Table_ T* , o trace é redundante e é reutilizado. Caso contrário,

Entrada encontrada na tabela		ação tomada pelo mecanismo
Memo_ Table_ G	*Memo_ Table_ T*	
Não	Não	Instrução marcada como não redundante e inserida na *Memo_ Table_ G*
Sim	Não	Instrução é marcada como redundante e reusada a partir da *Memo_ Table_ G*
X	Sim	Trace é reusado a partir da *Memo_ Table_ T*

Tabela 3.1: Seleção de candidatos a reuso

Operação	Estágios envolvidos	Descrição
(1)	DS1,DS2, DS3	Inserção de entradas na *Memo_ Table_ G*
(2)	DS1	Inicia a seleção de entradas na *Memo_ Table_ G*
(3)	DS2	Identifica instâncias de instruções redundantes
(4)	DS3	Atualização do trace em construção

Tabela 3.2: Operações relacionadas a construção de traços

se foi encontrada uma entrada na *Memo_ Table_ G* , a instrução é marcada como redundante e seu resultado é reusado. A tabela 3.1 resume o processo.

3.2 Incorporando o DTM em uma arquitetura superescalar

Para incorporar o mecanismo DTM a uma arquitetura superescalar utilizamos três estágios, atuantes em paralelo aos estágios de Busca, Decodificação e Despacho, e Entrega do processador. Estes estágios serão chamados de *DS1*, *DS2* e *DS3*, respectivamente.

Podemos ainda dividir a atuação desses estágios de acordo com a tarefa sendo realizada pelo DTM no momento: construção de traços ou reuso de traços redundantes. As operações realizadas pelos estágios do mecanismo estão sintetizadas nas tabelas 3.2 e 3.3.

Operação	Estágios envolvidos	Descrição
(2)	DS1	Inicia a seleção de entradas na *Memo_ Table_ G*
(3)	DS2	Identifica instâncias de instruções redundantes
(5)	DS1	Inicia a seleção de entradas redundantes na *Memo_ Table_ T*
(6)	DS2	Identifica traços redundantes
(7)	DS2	Decide se uma instrução ou trace podem ser reusados

Tabela 3.3: Operações relacionadas ao reuso de traços e instruções

3.2.1 Atuação dos estágios do mecanismo durante a construção de traços

Os estágios do mecanismo DTM realizam as seguintes operações durante a construção de traços de instruções redundantes:

- Operação (1) – Paralelamente ao estágio de Busca, para cada instrução pertencente ao domínio de validade do DTM o estágio DS1 insere uma nova entrada na *Memo_ Table_ G* , preenchendo o campo pc e enviando uma referência a esta entrada ao estágio DS2. Durante a decodificação o estágio DS2 anexa à instrução a entrada selecionada pelo estágio DS1. Quando a instrução chega ao estágio de entrega, o estágio DS3 preenche os campos sv1, sv2 e res/targ com os operandos de entrada e saída da instrução; se for uma instrução de desvio, os campos jmp, brc e btaken serão preeenchidos corretamente. Se a instrução corrente foi executada especulativamente e o estágio de entrega a descarta, o estágio DS3 fará o mesmo com a entrada na *Memo_ Table_ G* , ou seja, o DTM só armazena instâncias de instruções que ocorrem no caminho correto de execução;

- Operação (2) – Paralelamente ao estágio de busca, o estágio DS1 realiza uma busca na *Memo_ Table_ G* por instâncias de instruções que possuam o mesmo endereço. Estas instâncias pré-selecionadas serão as únicas comparadas com os

valores dos operandos de entrada na instrução durante o estágio de despacho;

- Operação (3) – Quando a instrução chega ao estágio de decodificação, este faz a leitura dos operandos fonte no conjunto de registradores e os passa ao estágio DS2, onde é feita a comparação com as entradas pré-selecionadas na operação (2). Se a instância é identificada como redundante a sua entrada na *Memo_ Table_ G* é liberada e a instrução é marcada como redundante. Instruções redundantes não são despachadas, mas são entregues normalmente durante o estágio de entrega. Convém ressaltar que a instrução só é marcada como redundante se os operandos fonte desta estiverem armazenados no conjunto de registradores;

- Operação (4) – quando a instrução chega ao estágio de entrega, este passa ao estágio DS3 os registradores de entrada e saída da instrução, e seus respectivos valores. Dependendo da classificação da instrução, redundante ou não redundante, o estágio DS3 decide pela criação de um novo traço, pela atualização do traço em construção ou pela finalização do traço em construção.

3.2.2 Atuação dos estágios do mecanismo durante o reuso de instruções e traços

Os estágios do mecanismo DTM realizam as seguintes operações durante o reuso de instruções e traços redundantes:

- Operações (2) e (3) – Estas operações são comuns para construção e reuso de traços e instruções;

- Operação (5) – Durante o estágio de Busca, para cada instrução dinâmica o estágio DS1 faz uma busca na *Memo_ Table_ T* por traços iniciados pelo mesmo pc; estas entradas serão comparadas com os valores atuais dos operandos de entrada do trace durante o estágio de decodificação.

- Operação (6) – Durante a decodificação da instrução, o estágio DS2 envia uma requisição ao conjunto de registradores para obtenção dos valores atuais dos registradores presentes no contexto de entrada dos traços pré-selecionados; estes valores são comparados com os valores armazenados nos traços, e em caso de uma comparação válida, o traço é redundante e poderá ser reutilizado.

- Operação (7) – O resultado das operações (3) e (6) pode indicar uma instrução ou traço redundante. Neste caso, o estágio DS2 toma as providências necessárias para efetivar o reuso;

O Mecanismo DTM é utilizado como critério para seleção de traços porque seleciona traços com bom número de dependências verdadeiras conforme relatado na Seção anterior. Foi verificado ainda [4] que os traços selecionados pelo DTM apresentam alta taxa de reuso, o que indica que aparecem várias vezes no decorrer da execução de um programa. Como não estamos interessados no reuso de traços, não será utilizada a parte do mecanismo responsável pela reutilização dos traços migrados.

Como devemos armazenar informações sobre a semântica dos traços migrados para instruções complexas, precisamos fazer algumas alterações na estrutura do mecanismo de reuso. Estas alterações são apresentadas no capítulo seguinte.

Capítulo 4

Migração de Traços

O objetivo da migração de traços é transformar instruções simples RISC em instruções complexas CISC (traços), obtendo ganho de performance no processo. As instruções CISC geradas têm que possuir a mesma semântica que as instruções simples constituintes do traço. O processo está representado na Figura 4.1.

Durante a execução de um programa ocorrem inúmeros traços de instruções dinâmicas. Como não dispomos de uma tabela com espaço infinito, precisamos utilizar algum critério para determinar os traços que serão migrados para instruções do tipo CISC.

Devemos ressaltar que a seleção dinâmica de traços oferece vantagem sobre a estática pois não sabemos *a priori* se traços selecionados estaticamente serão utilizados com freqüencia durante a execução de um programa. Conforme descrito em [4], os traços selecionados dinamicamente pelo mecanismo DTM tem alto percentual de reuso, o que indica que podem ocorrem várias vezes durante a execução de um programa (nem todos os traços selecionados pelo mecanismo apresentam esta

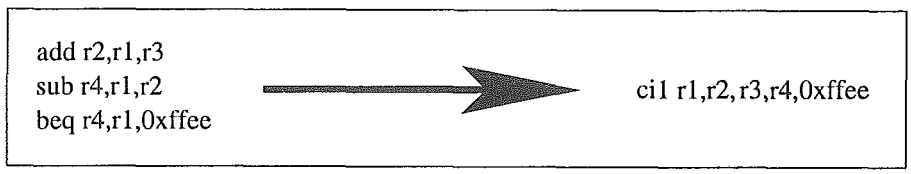


Figura 4.1: Migração de instruções RISC para uma instrução CISC

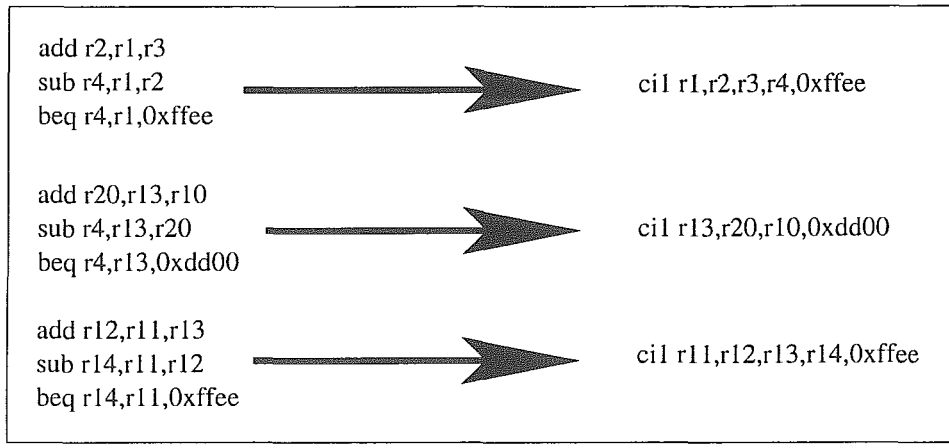


Figura 4.2: Traços diferentes sendo migrados para a mesma instrução CISC

característica). Traços escolhidos pelo mecanismo DTM são então bons candidatos para serem migrados para instruções do tipo CISC.

Outro fator importante é que diferentes traços, ocorrendo em momentos distintos da execução e operando em subconjuntos diferentes de registradores, podem possuir mesmo valor semântico, ou seja, podem ser representados por uma mesma instrução CISC. O mecanismo deve reconhecer esses traços (distintos para o mecanismo DTM) como a mesma instrução dinâmica CISC (Figura 4.2) ocorrendo em mais de uma localização no código do programa.

Dois traços possuem mesmo valor semântico quando contêm a mesma seqüência de instruções, e a mesma estrutura de interdependência entre estas instruções. Ressaltamos que a presença de imediatos ou endereços alvo de desvios não interfere no o valor semântico dos traços.

Os traços representados na Figura 4.3 apresentam mesma seqüência de instruções – *ADD*, *SUB* e *ADD* – e mesma estrutura de interdependência entre estas instruções. Podemos observar que a estrutura de interdependência não leva em consideração quais são os registradores no contexto de entrada do traço, uma vez que estes não estão atrelados à semântica deste.

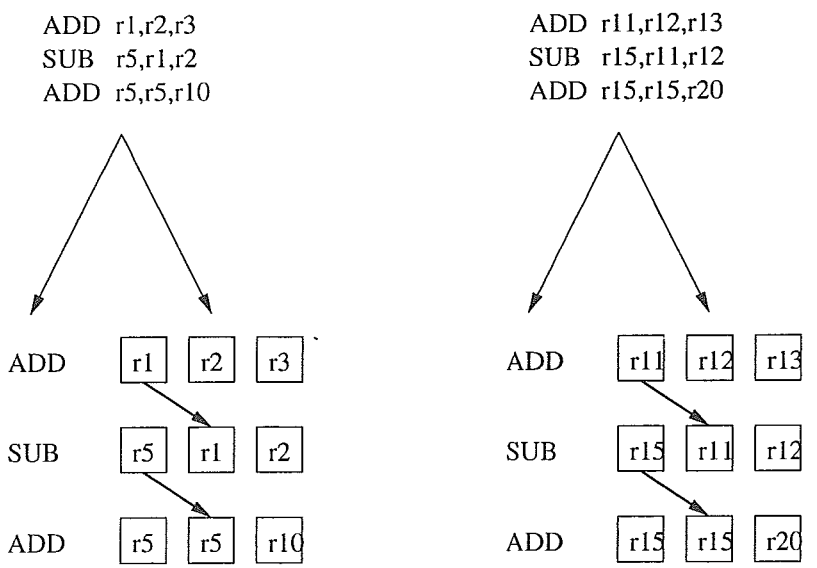


Figura 4.3: Identificando traços com mesmo valor semântico

4.1 Alterações no mecanismo DTM

Antes de descrever as alterações necessárias no DTM para o suporte à migração dinâmica de traços, é importante chamar a atenção para a mudança de foco no mecanismo: não estamos mais interessados em reutilizar instâncias de instruções e traços, e sim identificar quais os traços candidatos à migração com maior potencial de ocorrerem novamente durante a execução do programa. Como todos os traços migrados serão executados pelo processador, torna-se necessário armazenar informações sobre a semântica das operações contidas em cada um dos traços migrados.

Para implementar o mecanismo proposto, substituímos a Tabela *Memo_Table_T* por duas tabelas, chamadas de *tsMT - Trace Semantics Memo Table* - e *tlMT - Trace Lookup Memo Table*. A Tabela *tsMT* armazena informações sobre a semântica dos traços migrados, independente de sua localização no código do programa. A Tabela *tlMT* mapeia instâncias de instruções complexas no código à entrada na *tsMT* com semântica correspondente.

4.1.1 Estrutura da Tabela *tsMT*

As entradas da Tabela *tsMT* possuem a estrutura representada na Figura 4.4. O campo *tsize* armazena o número de instruções simples que fazem parte da instrução complexa correspondente. Uma instrução complexa pode ser composta por até N instruções simples. Os campos *ops1* a *opsN* armazenam os *opcodes* das instruções constituintes da instrução complexa. Os campos *it1_0* a *itN_1* armazenam o índice da instrução dentro do traço responsável pela produção do valor daquele operando, ou o valor zero se o operando for produzido por uma instrução não pertencente ao traço. O registrador *r0* é um caso especial, uma vez que seu valor é fixo e não pode ser alterado por nenhuma instrução simples ou complexa).

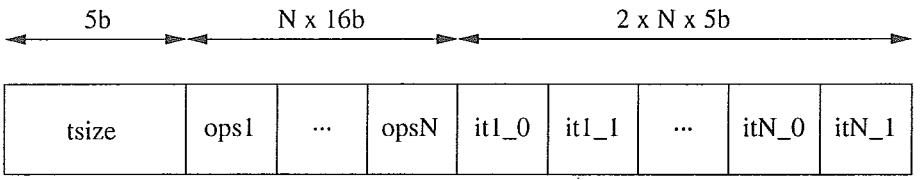


Figura 4.4: Uma entrada da Tabela *tsMT*

Dois traços possuem mesma semântica quando todos os campos de suas respectivas entradas na *tsMT* são iguais.

4.1.2 Estrutura da Tabela *tlMT*

As entradas da tabela *tlMT* possuem a estrutura representada na Figura 4.5. O campo *PC* armazena o endereço de início da instrução complexa, que é igual ao endereço da primeira instrução dinâmica presente no traço correspondente. Os campos *rin1_0* a *rinN_1* armazenam os endereços dos registradores correspondentes ao contexto de entrada das instruções simples que fazem parte do traço, ou zero se a entrada correspondente é produzida por uma instrução contida no traço. Os campos *rout1* a *routN* armazenam os registradores que fazem parte do contexto de saída do traço, ou zero se o registrador correspondente é sobrescrito por uma operação

posterior contida no mesmo traço. Os campos *imm1* a *immN* armazenam valores de imediatos contidos nas instruções dos traços. O campo *target* armazena o destino de uma instrução de desvio incondicional, que termina o traço. O campo *tsidx* aponta para a entrada na *tsMT* que contém a semântica deste traço.

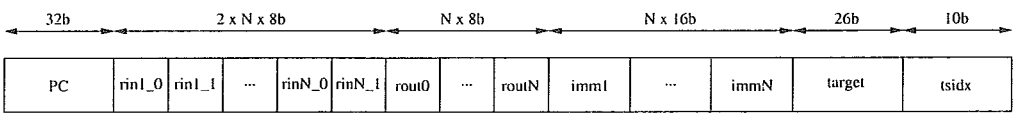


Figura 4.5: Uma entrada da tabela *tlMT*

4.1.3 Outras estruturas necessárias ao mecanismo

O mecanismo apresenta ainda um *buffer* onde ocorre a construção de traços a partir de instruções simples redundantes. O *buffer* é composto de uma entrada da Tabela *tsMT* e uma entrada da Tabela *tlMT*, além de dois campos de máscaras que indicam qual a última instrução do traço responsável pela produção do valor de cada um dos registradores arquiteturais. Essas estruturas também registram a composição dos contexto de entrada e saída do traço em construção.

Outra estrutura de importância no funcionamento do mecanismo é uma nova unidade funcional chamada de *Trace Unit*, responsável pela execução das instruções complexas.A diferença entre as duas alternativas de execução propostas se dá na forma como a *Trace Unit* executa as instruções complexas: na alternativa "otimista" os traços são executados como se todas as instruções simples constituintes pudessem ser executadas em paralelo, ou seja, um traço sem dependências verdadeiras em sua composição. Esta unidade poderia ser baseada em uma extensão da unidade lógico-aritmética descrita em [16]; na alternativa "pessimista", consideramos sempre o pior caso: existem dependências verdadeira entre todas as instruções simples do traço, de forma que este é executado sequencialmente pela unidade funcional.

4.1.4 Alterações nos critérios de formação de traços

Os critérios de formação de traços de instruções dinâmicas utulizados pelo mecanismo DTM precisam ser alterados para suportar a criação de instruções complexas. Os novos critérios são:

- No mecanismo DTM o tamanho dos traços é limitado pelo tamanho do contexto de entrada. Para a migração de primitivas, um parâmetro mais razoável é o tamanho dos traços migrados, pois informações sobre as instruções dinâmicas simples contidas nos traços devem ser armazenadas nas tabelas *tsMT* e *tlMT*.

- O mecanismo DTM prevê traços com número variável de instruções de desvio em sua constituição. Como instruções complexas são executadas atomicamente, elas devem ser encerradas no primeiro desvio encontrado. Caso contrário, a instrução complexa formada não representaria todos os fluxos de execução possíveis.

O traço representado na Figura 4.6 só seria válido se o desvio armazenado na posição (1) sempre fosse avaliado como não tomado, caso contrário deveríamos invalidar esta instrução complexa e redirecionar o estágio de busca para a instrução simples correspondente ao início do traço. Este comportamento não é vantajoso em um mecanismo que irá avaliar a instrução complexa toda vez que ela aparecer no fluxo de execução do programa. Ao terminar o traço sempre que uma instrução de desvio é encontrada, o problema descrito não ocorre.

4.2 A construção de instruções complexas

O mecanismo realiza operações idênticas às descritas no Capítulo anterior para a memorização de traços. Os mecanismos diferem entre si apenas na operação (4),

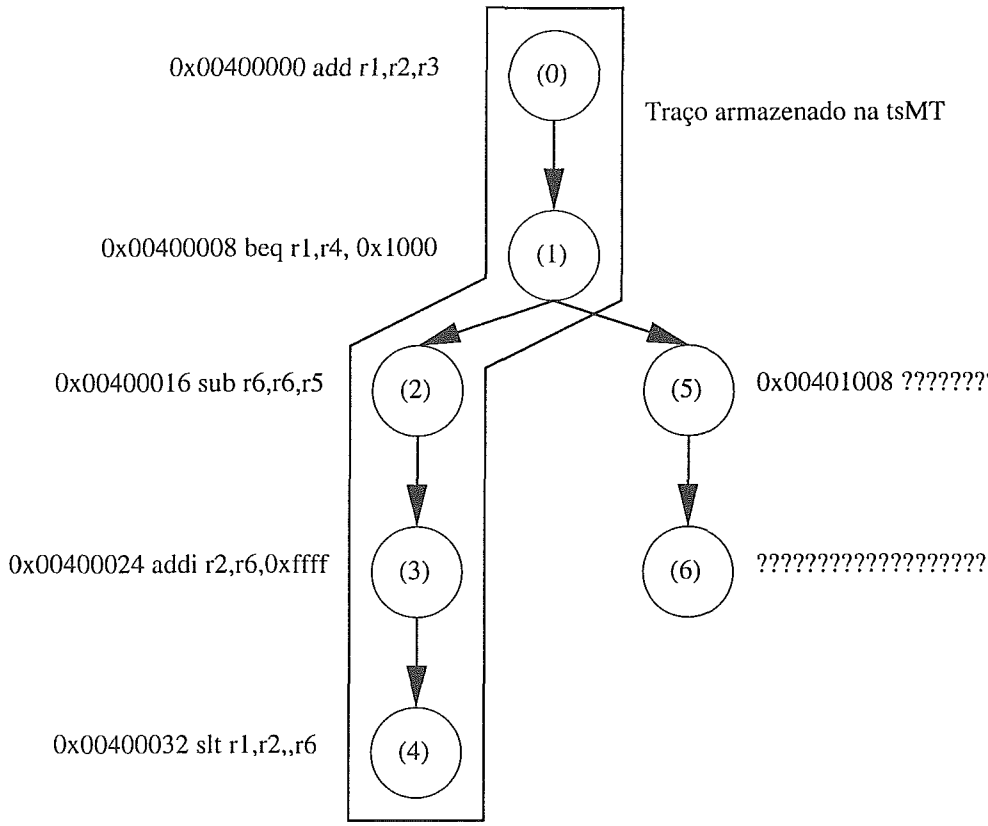


Figura 4.6: Traço com semântica incompleta devido à instrução de desvio em seu interior

atualização do traço em construção.

No mecanismo de migração de traços, toda vez que uma instrução simples chega ao estágio de entrega, ela é passada ao estágio DS3 que toma uma decisão de acordo com os critérios de construção de traços descritos na seção anterior.

Caso o mecanismo decida por adicionar a instrução ao traço em construção, o mecanismo segue as seguintes etapas:

- (1) verifica-se no campo de máscaras de saída se os valores dos registradores de entrada são produzidos por alguma instrução presente no traço em construção. Se o valor é produzido pelo traço, a entrada correspondente na estrutura de interligação é preenchida com o índice da instrução que produz o valor. Caso contrário, é consultado o campo de máscaras de entrada para determinar se

o registrador já faz parte do contexto de entrada do traço. Se o resultado for negativo, a respectiva entrada na *tlMT* é preenchida com seu endereço e a respectiva entrada no campo de máscaras de entrada é atualizada;

- (2) Campos com valor imediato são inseridos diretamente na entrada da *tlMT*;
- (3) O opcode da instrução é armazenado no campo correspondente na entrada da *tsMT* ;
- (4) O campo do contexto de saída correspondente ao registrador de saída da instrução é atualizado com o índice da instrução dentro do traço. Esta informação é utilizada na etapa (1) para a atualização da estrutura de inter-dependência entre as instruções do traço.

A Figura 4.7 demonstra o processo de construção de um traço.

Caso o mecanismo decida por encerrar a construção do traço, ele procede da seguinte forma: O mecanismo faz uma busca na *tsMT* por uma entrada igual à armazenada no *buffer*. Se é encontrada uma entrada igual, o mecanismo descarta a entrada armazenada no *buffer* e insere somente a entrada do *buffer* na *tsMT* , preenchendo o campo *tsidx* com o endereço da entrada encontrada na *tlMT*. Caso contrário, a entrada é inserida na *tsMT* e na *tlMT* .

Quando o mecanismo precisa eliminar uma entrada da *tsMT*, para armazenar uma nova instrução complexa, ele faz uma busca na *tlMT* por entradas que fazem referência à entrada a ser eliminada na *tsMT* e as invalida. Quando uma entrada da *tlMT* deve ser eliminada para dar lugar a outra, o mecanismo busca outras entradas na *tlMT* que fazem referência à mesma entrada na *tsMT* , utilizando o campo *tsidx*. Se esta entrada for a única a fazer referência, ambas as entradas são invalidadas. Caso contrário, somente a entrada da tabela *tlMT* será eliminada.

O mecanismo como implementado seleciona traços de instruções simples e os ar-

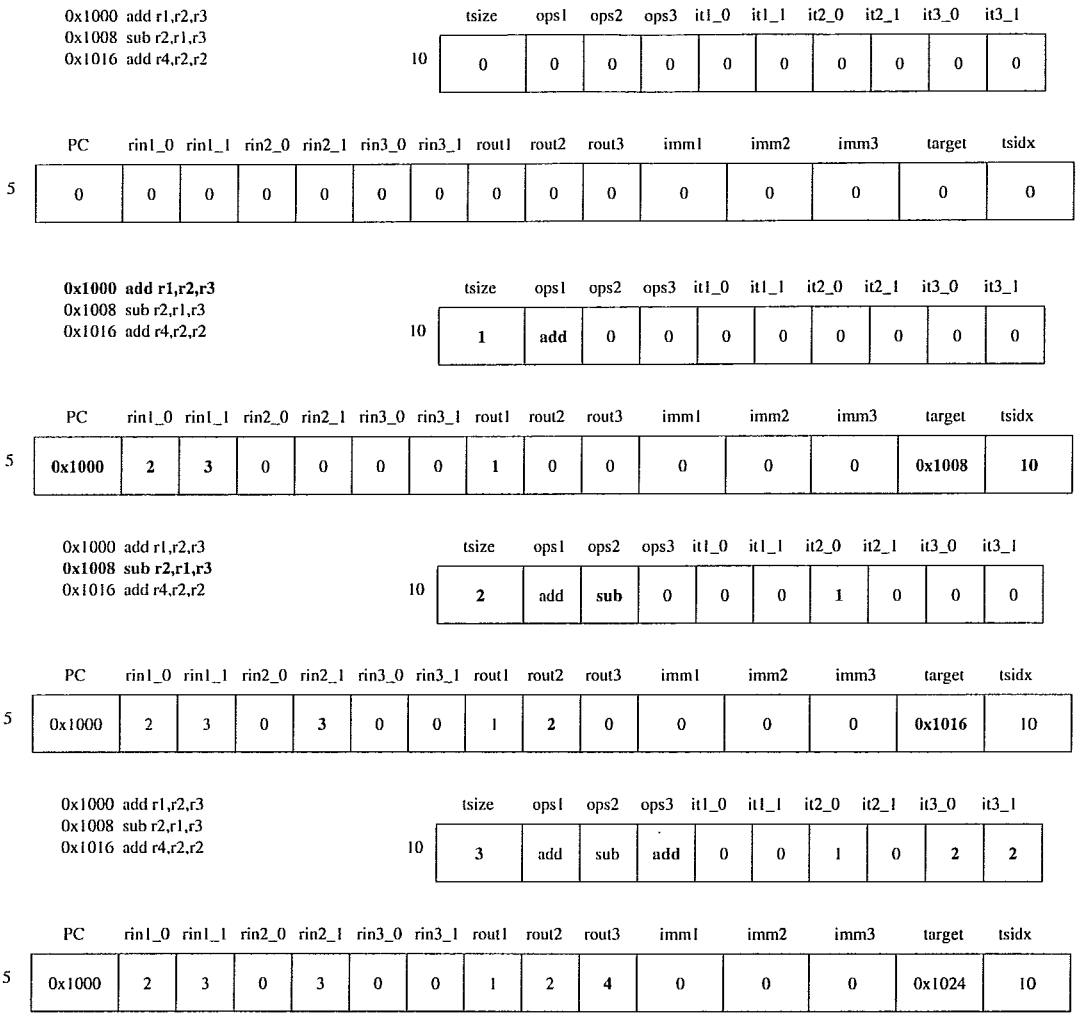


Figura 4.7: Construção de um traço a partir de instruções simples

mazena como instruções complexas. A próxima seção descreve como esse mecanismo pode ser adicionado a uma arquitetura superescalar pipelined.

4.3 Adicionando o mecanismo a uma arquitetura superescalar pipelined

O mecanismo de migração de traços pode ser adicionado a um processador superescalar pipelined substrato da maneira descrita no capítulo anterior. Os estágios utilizados são os mesmos DS1, DS2 e DS3. Porém, o mecanismo de migração de traços não reutiliza resultados computados durante a execução de traços anteriores.

Estes traços são executados como uma única instrução dentro uma nova unidade funcional, a *Trace Unit* .

Tendo implementado o mecanismo em uma arquitetura superescalar pipelined, precisamos avaliar de que forma as instruções complexas criadas podem ser utilizadas pelo processador e o impacto provocado por sua utilização na execução de programas. Para avaliar o potencial do mecanismo foram consideradas duas alternativas principais: uma "otimista", onde todas as instruções complexas são executadas pela *Trace Unit* no mesmo tempo de execução de uma instrução simples; e uma "pessimista", onde todas as instruções complexas são executadas pela *Trace Unit* como uma sequência de instruções simples sem possibilidade de paralelismo.

Outras dimensões, tais como o impacto do mecanismo sobre o número de acessos à memória cache de instruções, o funcionamento do mecanismo quando utilizado com um cache de instruções perfeito ou com um preditor de desvios perfeito, tamanho dos traços formados, quantos desses traços contém dependências verdadeiras e o tamanho do contexto de entrada dos traços migrados serão analisados nos capítulos seguintes.

Capítulo 5

Base Experimental

5.1 Ambiente de simulação

Para efetuar os experimentos de avaliação do mecanismo proposto foram desenvolvidos dois simuladores – *sim-dpm-opt* para a alternativa otimista e *sim-dpm-pes* para a alternativa pessimista – a partir do simulador *sim-outorder* do *SimpleScalar Tool Suite v3.0.d* [2]. Este simulador implementa a arquitetura descrita em [20], e sofreu alterações para representar a arquitetura descrita no Capítulo 4. Foram adicionadas ao processador as tabelas *Memo_Table_G* , *tsMT* e *tlMT* , e as unidades funcionais *Trace Unit* . Para executar as simulações foram utilizados processadores *AMD Athlon XP 2000+* e *Intel Xeon 2.8GHz* sob sistema operacional linux *Fedora Core 1.*

5.2 Programas de teste

Para a realização dos experimentos foram utilizados 5 programas da suíte *SpecINT2000* [9]. Os programas gzip e bzip2 foram rodados respectivamente com 5 e 3 entradas. O programa vpr é dividido em seus dois modos de execução: place e route. A Tabela 5.1 apresenta os programas utilizados com as respectivas entradas, e o número de instruções executadas em cada caso. Em todos os casos a execução dos programas foi completada.

Foram utilizadas as entradas *lgred* descritas em [12] com os parâmetros padrão,

programa	entradas	instruções executadas	instruções entregues
gzip	lgred.graphic	1.526.624.998	1.335.138.131
gzip	lgred.log	514.842.262	480.479.342
gzip	lgred.program	2.163.441.243	1.915.902.087
gzip	lgred.random	1.170.595.971	1.045.629.187
gzip	lgred.source	1.290.928.773	1.138.564.087
vpr	lgred.net place	1.986.451.923	1.694.114.115
vpr	lgred.net route	823.702.609	709.592.955
mcf	lgred.in	820.330.745	713.186.689
parser	2.1.dict lgred.in	3.476.449.238	3.039.760.397
bzip2	lgred.graphic	2.576.767.748	2.429.214.737
bzip2	lgred.program	2.072.590.046	1.914.755.384
bzip2	lgred.source	1.703.142.892	1.547.472.494

Tabela 5.1: Programas e entradas utilizados na simulação

em virtude do tempo de execução proibitivo das entradas de referência do *SpecINT2000*. Os executáveis utilizados foram compilados com o *gcc 2.7.3.2* pelo grupo MIRV [18], com as chaves de compilação *-02 -funroll-loops*.

A Tabela 5.2 fornece informações adicionais referentes ao perfil das instruções executadas em cada um dos programas do *SpecINT2000* utilizados.

5.3 Parâmetros arquiteturais do processador simulado

A Tabela 5.3 contém os parâmetros de configuração do processador substrato utilizado na simulação.

5.4 Parâmetros do mecanismo de migração de traços

A Tabela 5.4 contém os parâmetros de configuração do mecanismo de migração de traços. O tamanho das tabelas *Memo_Table_G* e *tsMT* foi escolhido de acordo com o observado em [4], apresentando melhor relação custo-benefício. Foi utilizado

programa	Load/Store	Desvio	ILA Inteiro	ILA PF	Outros
gzip – graphic	30,47	18,43	51,10	00,00	00,00
gzip – log	30,26	16,59	53,15	00,00	00,00
gzip – program	24,00	21,32	54,68	00,00	00,00
gzip – random	29,95	19,15	50,90	00,00	00,00
gzip – source	27,12	19,11	53,77	00,00	00,00
vpr – place	32,43	16,46	44,83	06,82	00,00
vpr – route	40,36	14,36	40,37	04,90	00,00
mcf	35,90	25,07	39,04	00,00	00,00
parser	37,87	20,44	41,69	00,00	00,00
bzip2 – graphic	32,55	12,62	54,83	00,00	00,00
bzip2 – program	31,27	13,90	54,83	00,00	00,00
bzip2 – source	29,80	14,75	55,45	00,00	00,00

Tabela 5.2: Perfil dos tipos de instrução executadas em cada programa

Estágio de busca	4 instruções por ciclo. Um desvio tomado por ciclo. Pode ultrapassar as fronteiras de uma linha de cache no *mesmo* ciclo.
Cache de instruções	16 KB, associativo, 2 por conjunto, 32 bytes por linha, latência de 6 ciclos para acessos sem sucesso ao cache L1 e latência de 20 ciclos para acessos sem sucesso ao cache L2
Cache de dados	16 KB, associativo, 4 por conjunto, 32 bytes por linha, latência de 6 ciclos para acessos sem sucesso ao cache L1 e latência de 20 ciclos para acessos sem sucesso ao cache L2. Não bloqueante.
Preditor de desvios	Bimodal, 2048 entradas, pode predizer vários desvios por ciclo.
Mecanismo de execução	Execução fora de ordem, com suporte para execução especulativa. Buffer de emissão (RUU) com 16 entradas e fila de *load/store* com 8 entradas. Loads são executados após todos os endereços de stores prévios serem conhecidos. Loads são servidos por stores acessando o mesmo endereço se ambos estiverem na fila de load/store.
Conjunto de registradores	32 registradores inteiros e 32 registradores de ponto flutuante. Registradores hi, lo e fcc
Unidades funcionais	4 ULA para inteiros, com latência 1; 1 unidade MULT/DIV para inteiros, com latência 3 para mult e 20 para div; 4 ULA para ponto flutuante, com latência 2; 1 unidade MULT/DIV para ponto flutuante, com latência 4 para mult, 12 para div e 24 para sqrt; 2 unidades de load/store com latência 1.

Tabela 5.3: Parâmetros de configuração do processador substrato

Tamanho da *Memo_Table_G*	4096 entradas
Tamanho da *tsMT*	512 entradas
Tamanho da *tlMT*	1024 entradas
Tamanho máximo dos traços	16 instruções
Domínio de validade	Instruções de ULA de inteiros; Desvios condicionais e incondicionais; Instruções de acesso à memória.
Política de atualização das tabelas	LRU

Tabela 5.4: Parâmetros de configuração do mecanismo de migração de traços

um tamanho maior para a tabela *tlMT* a fim de verificar melhor a quantidade de instruções migradas com mesma semântica.

Capítulo 6

Resultados

6.1 Definições

Nas seções a seguir, utilizaremos as seguintes definições:

ipc – Número médio de instruções entregues por ciclo de clock;

speedup – Aceleração de desempenho;

Média Harmônica $= n\left(\sum_{i=1}^{n}(1/x_i)\right)^{-1}$, onde x_i é um elemento a ser considerado para a média.

$speedup = ipc_{mec}/ipc_{base}$

onde ipc_{mec} considera a arquitetura base com o mecanismo de migração de traços e ipc_{base} considera a arquitetura base sem o mecanismo de migração.

6.2 Efeito do mecanismo de migração sobre a Arquitetura Base

Na Figura 6.1 podemos observar o efeito da adição do mecanismo de migração de traços à arquitetura substrato. Para a alternativa otimista, podemos observar um expressivo ganho de performance. Os programas do *SpecINT2000* avaliados apresentaram média harmônica do *speedup* igual a 1,19, com 1,15 para o programa bzip2 com entrada graphic e 1,35 para o programa gzip com entrada program, mesmo com todas as instruções sendo executadas pelo processador. É interessante observar que ambos os extremos são programas que realizam tarefas de compressão de arquivos.

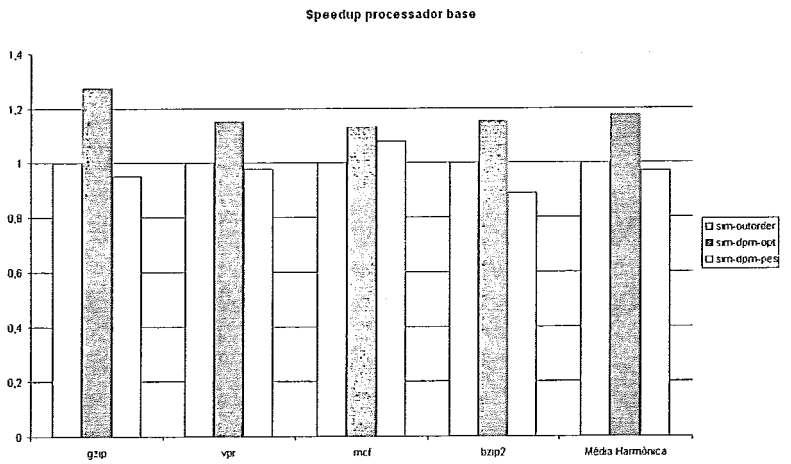


Figura 6.1: Speedup decorrente da adição do mecanismo de migração de traços à arquitetura base

Para a alternativa pessimista, observamos um *speedup* menor do que 1, mais exatamente 0,94. Isto ocorre porque a alternativa pessimista atua "eliminando" a superescalaridade da máquina, fazendo com que instruções que fazem parte de um mesmo traço mas não possuem dependências verdadeiras sejam executadas sequencialmente.

6.3 Influência da cache de instruções e do preditor de desvios no funcionamento do mecanismo de migração de traços

Para avaliar o impacto do cache de instruções e do preditor de desvios sobre o mecanismo de migração de traços, foram realizadas simulações utilizando caches de instruções e preditores de desvios perfeitos.

No caso em que utilizamos uma cache de instruções perfeita, obtivemos média harmônica de 1,18 e 0,92 para as alternativas otimista e pessimista, respectivamente. Podemos então concluir que o funcionamento do mecanismo não é afetado por alterações na estrutura da memória cache de instruções.

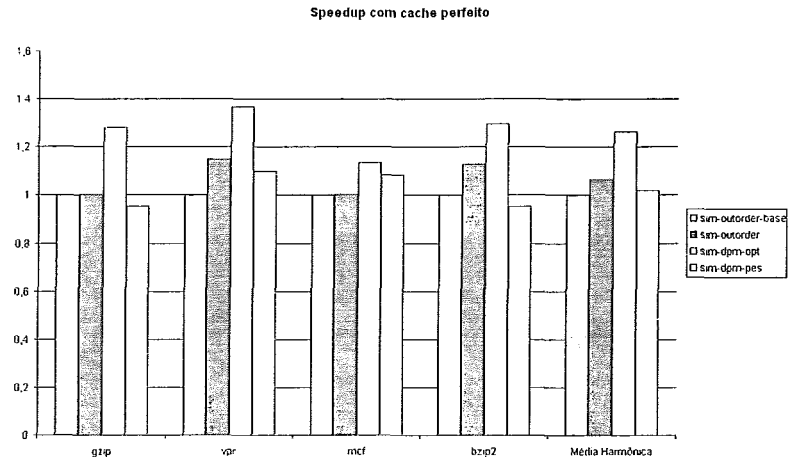


Figura 6.2: efeito de uma cache de instruções perfeita no funcionamento do mecanismo de migração de traços

Podemos observar ainda que o mecanismo proposto produz ganhos expressivos de performance mesmo quando em uma situação ideal eliminamos a penalidade RISC imposta pela cache de dados do processador.

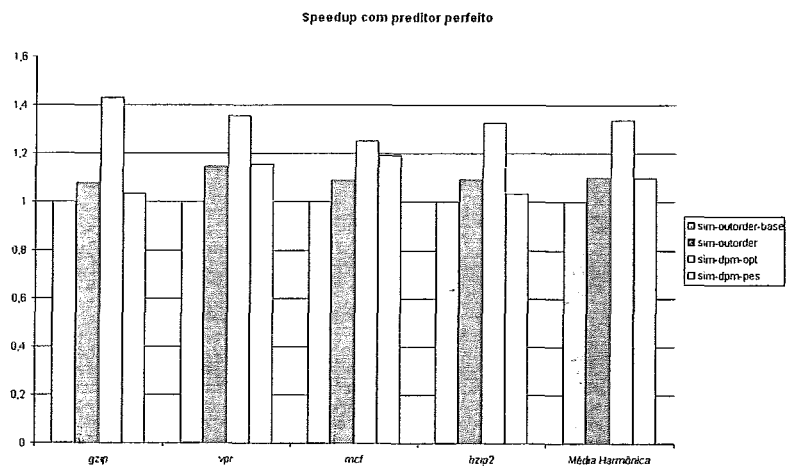


Figura 6.3: Efeito de um preditor de desvios perfeito no funcionamento do mecanismo de migração de traços

Podemos observar que a mudança no preditor de desvios influi pouco no desempenho do mecanismo de migração. Houve uma pequena melhora, com as médias harmônicas em 1,20 e 0,98 para as alternativas otimista e pessimista, respectiva-

mente. Esta melhora é proveniente da não execução do programa parser nesta configuração, e não de alguma característica do mecanismo.

6.4 Economia no acesso ao cache de instruções

Quando um traço é utilizado a partir do mecanismo de migração, deixamos de fazer acessos à memória cache para as instruções subsequentes à inicial. Espera-se então uma expressiva economia no número de acessos à memória cache de instruções.

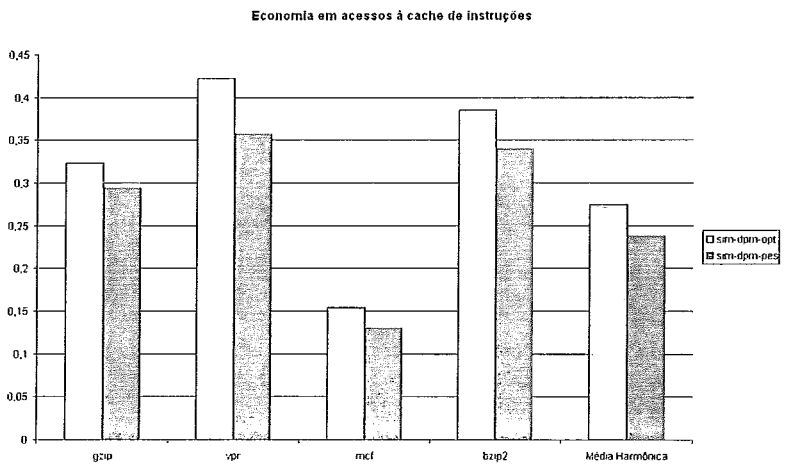


Figura 6.4: Economia de acessos à cache de instruções

Podemos constatar que o mecanismo elimina em até 50% o número de acessos à cache de instruções, com média harmônica de 30% para ambas as alternativas. Este pode ser o motivo porque o funcionamento da memória cache de instruções tem pouca influência no potencial de aceleração do mecanismo.

6.5 Perfil dos traços migrados e reutilizados

Analisamos agora o perfil qualitativo dos traços migrados sob diversos aspectos. Inicialmente, podemos observar na Figura 6.5 que 96% dos traços migrados possuem até 5 instruções simples em sua composição. Este dado se repete de maneira uniforme em todos os programas utilizados durante o teste.

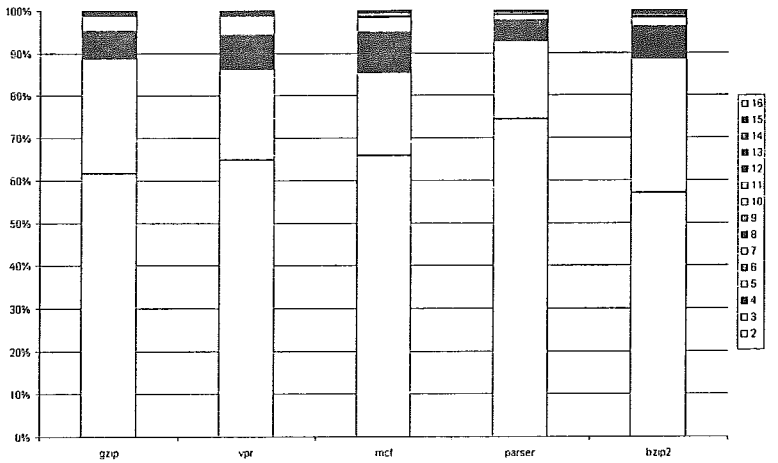


Figura 6.5: Distribuição percentual do número de instruções simples contidas nos traços migrados

Na Figura 6.6, podemos obsevar que o mecanismo proposto cria novas instruções complexas pois seleciona traços com bom número de dependências verdadeiras, com 67% no gzip no melhor caso e 45% no parser no pior caso, com média harmônica de 56%.

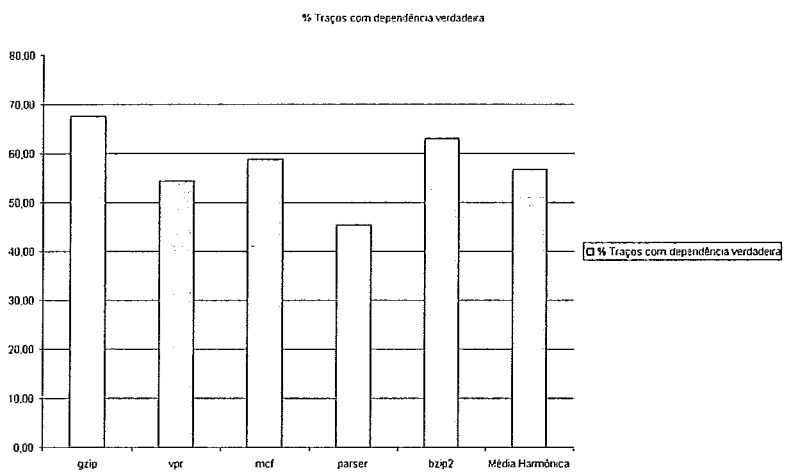


Figura 6.6: Percentual de traços migrados com dependências verdadeiras

Podemos observar ainda na Figura 6.7 que quase a totalidade dos traços migrados e reutilizados durante a execução do programa contém até 5 instruções em

sua composição. Este dado pode ser utilizado para limitar o tamanho dos traços construídos economizando espaço na estrutura das tabelas de memorização.

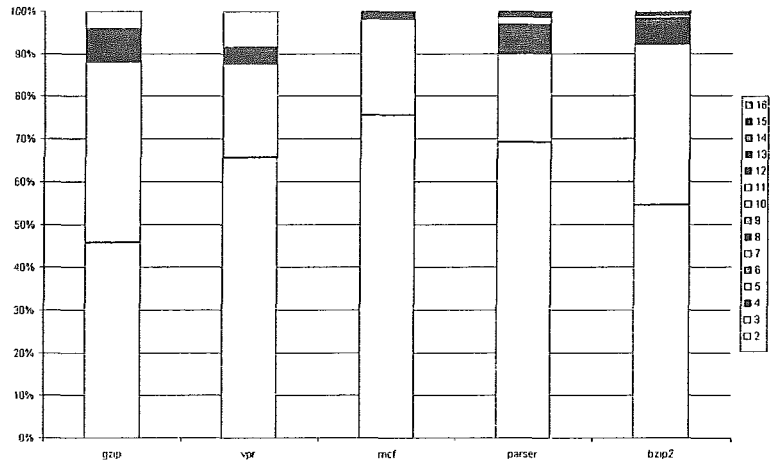


Figura 6.7: Percentual de reuso de traços de acordo com o tamanho

6.6 Pressão no banco de registradores

Para avaliar a pressão que o mecanismo exerce sobre o banco de registradores durante os estágios de leitura de operandos e escrita de resultados, se torna necessária uma avaliação do número de operandos no contexto de entrada de cada traço migrado.

Conforme mostrado na Figura 6.8, quase a totalidade dos traços migrados possui até 5 operandos distintos no seu contexto de entrada. Este dados serve de base para que possamos avaliar a pressão de leitura e escrita no banco de registradores a cada ciclo de clock. Os valores obtidos podem ser encontrados nas Figuras 6.9 e 6.10. Para o estágio de leitura, observamos até 8 registradores distintos por ciclo de clock. Para o estagio de escrita, observamos a necessidade de 4 portas de escrita no banco de registradores. Ambos os valores são factíveis em microarquiteturas atuais.

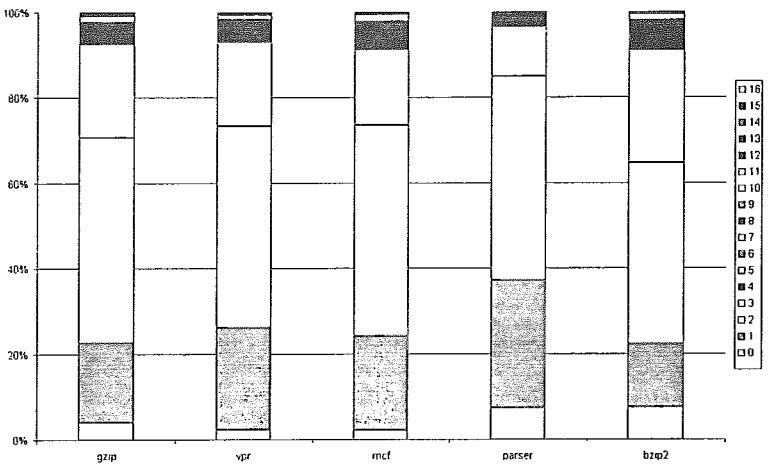


Figura 6.8: Distribuição percentual do número de entradas requeridas por traço

6.7 Ganhos do mecanismo em um processador escalar

Para avaliar se a perda de performance da alternativa pessimista é decorrente da perda da superescalaridade, foram realizados experimentos com base em um processador escalar pipelined. Os resultados podem ser observados na Figura 6.11. Foi obtida média harmônica de 15% de speedup na alternativa pessimista, o que indica que realmente a penalidade observada é decorrente da perda de escalaridade nesta alternativa.

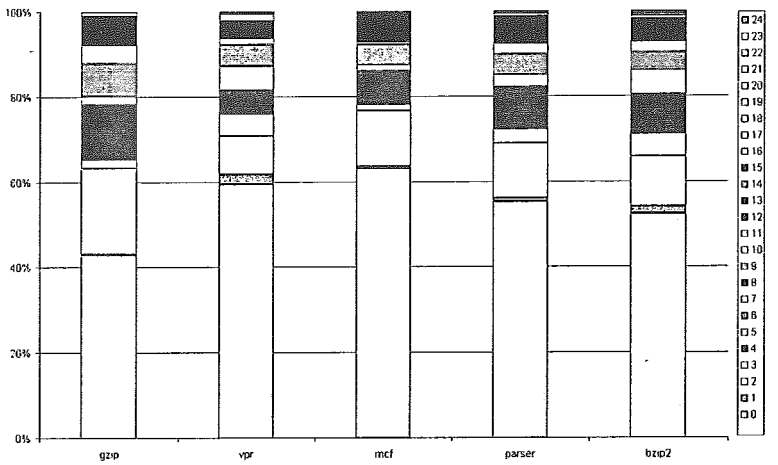


Figura 6.9: Pressão sobre o banco de registradores durante o estágio de leitura

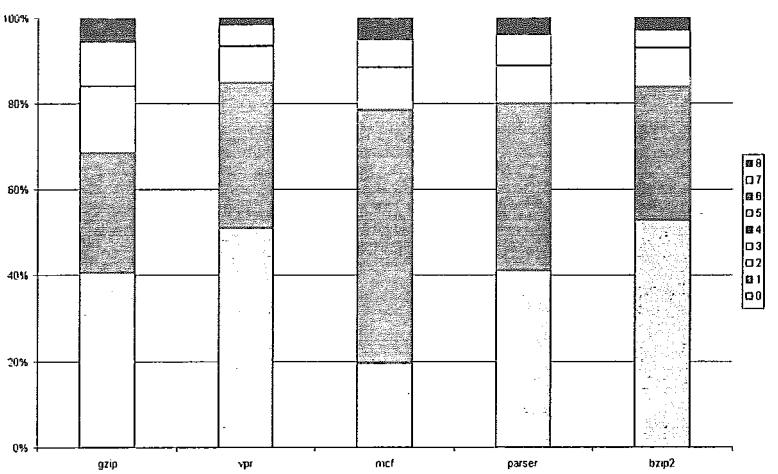


Figura 6.10: Pressão sobre o banco de registradores durante o estágio de escrita

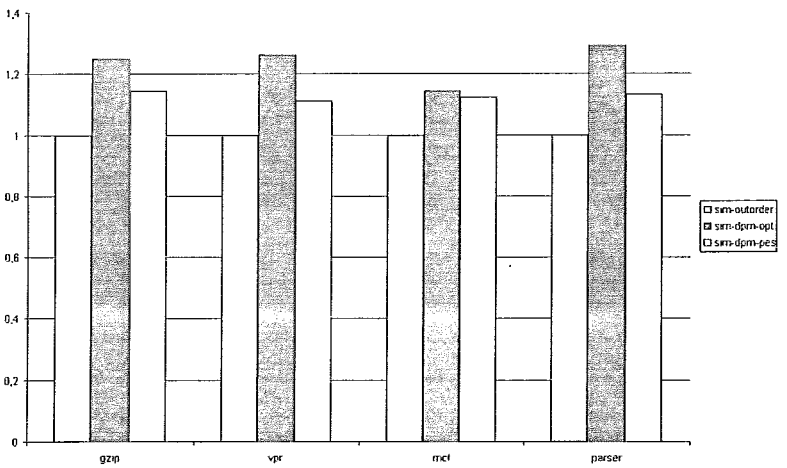


Figura 6.11: Speedup em um processador escalar

Capítulo 7

Conclusões

Neste trabalho foi idealizada uma nova classe de mecanismos com base na migração de traços para instruções complexas com o intuito de melhorar a performance de um processador superescalar substrato.

Foi realizada uma exploração inicial do potencial da migração de traços para instruções complexas do tipo CISC utilizando o mecanismo DTM para realizar a seleção de candidatos a migração. Foram analisadas duas alternativas representando os dois extremos do espectro de pesquisa.

No lado positivo, observamos um potencial de aceleração de 19% na média, resultado bastante encorajador. No extremo oposto, observamos uma desaceleração de 5% decorrente da perda de superescalaridade imposta por esta alternativa. Foi observado que em um processador escalar, mesmo a alternativa pessimista apresenta ganhos da ordem de 15%

Observamos ainda que o mecanismo é bastante tolerante a alterações no funcionamento do preditor de desvios e da cache de instruções, e podemos concluir que seu ganho é independente destes mecanismos. Podemos ainda observar que o mecanismo apresenta ganhos subtanciais mesmo quando é eliminada a penalidade RISC devido à misses na memória cache de instruções.

7.1 Trabalhos Futuros

Vários aspectos da migração da traces não foram avaliados nesta investigação preliminar. Uma lista de futuros tópicos de pesquisa incluiria:

- **elaboração de uma alternativa mais realista.** Neste trabalho foram analisados os dois extremos de funcionamento de um mecanismo de migração de traços. É preciso desenvolver uma alternativa mais realista, que leve em consideração a quantidade de dependências verdadeiras nos traços migrados para explorar ao máximo o paralelismo contido neles.

- **Tamanho das tabelas e políticas alternativas de troca.** Nesta exploração inicial, não foram realizados experimentos visando avaliar o impacto do tamanho das tabelas de memorização ou a política de atualização destas tabelas. Este aspecto é especialmente importante visto que a política atual de eliminação de entradas nas tabelas é bastante custosa para ser implementada em hardware.

- **Construção de traces com mais de uma instrução de desvio.** Na proposta atual instruções de desvio sempre terminam um trace. O mecanismo poderia ser estendido a fim de que guardássemos, após uma instrução de desvio, os dois fluxos de execução possíveis, formando uma instrução complexa em árvore;

- **Construção de traces de traces.** Outra extensão possível para o mecanismo é a construção de traces a partir de traces migrados recursivamente, podendo chegar ao ponto de em determinados programas executarmos apenas uma instrução complexa.

Referências Bibliográficas

[1] Peter M. Athanas and Harvey F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *Computer*, 26(3):11–18, 1993.

[2] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *Computer Architecture News*, pages 13–25, Junho 1997.

[3] Nathan Clark, Hongtao Zhong, and Scott Mahlke. Processor acceleration through automated instruction set customization. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 129. IEEE Computer Society, 2003.

[4] A. T. Costa. *Explorando dinamicamente o reuso de traces em nível de aruitetura de processador.* PhD thesis, Departamento de Engenharia de Sistemas e Computação, COPPE/UFRJ, Abril 2001.

[5] Amarildo T. da Costa, Felipe M. G. França, and Eliseu M. C. Filho. The dynamic trace memoization reuse technique. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 92–99, Philadelphia, Outubro 2000.

[6] C. N. Keltcher et al. the amd opteron processor for multiprocessor servers. *IEEE Micro*, pages 66–76, Mar–Abr 2003.

[7] Simcha Gochamn et al. The intel pentium m processor: Microarchitecture and performance. *Intel Technology Journal*, 7(2), 2003.

[8] F. M G Franca, N. Q. Vasconcelos, and E. S T Fernandes. Design and realization of mlm: a multilingual machine. In *Proceedings of the 19th annual workshop on Microprogramming*, pages 129–137. ACM Press, 1986.

[9] John L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer magazine*, Julho 2000.

[10] B. Holtkamp and P. Wagner. An algorithm for selection of migration candidates. In *17th Annual Microprogramming Workshop*, pages 140–146, 1984.

[11] Shiliang Hu and James E. Smith. Using dynamic binary translation to fuse depedent instructions. In *2004 International Symposium on Code Generation and Optimization*, 2004. Trabalho submetido para conferência.

[12] AJ KleinOsowski and David J. Lilja. Minnespec: A new spec benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1, Junho 2002.

[13] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[14] E. Luque and A. Ripoll. Microprogramming: a tool for vertical migration. *Microprocessing and microprogramming*, 8:219–228, 1981.

[15] E. Luque, J. Sorribes, and A. Ripoll. Coprocessor for real-time dynamic vertical migration. *Microprocessing and microprogramming*, 8:219–228, 1981.

[16] Nadeem Malik, Richard J. Eickemeyer, and Statis Vassiliadis. Interlock collapsing alu for increased instruction-level parallelism. In *Proceedings of the 25th*

Annual ACM/IEEE International Symposium and Workshopon Microarchitecture, setembro 1992.

[17] Tom Pittman. The risc penalty. *IEEE Micro*, pages 5,76–80, Dezembro 1995.

[18] Matthew Postiff, David Greene, Charles Lefurgy, Dave Helder, and Trevor Mudge. The mirv simplescalar/pisa compiler. Technical Report CSE-TR-421-00, University of Michigan EECS Department, Abril 2000.

[19] H. Shin and M. Malek. Identification of microprogrammable loops for problem oriented architecture synthesis. In *16th Annual Microprogramming Workshop*, pages 1–6, 1983.

[20] Gurindar S. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Trans. Comput.*, 39(3):349–359, 1990.

[21] Nelson Q. Vasconcelos. Uma máquina básica edison. Master's thesis, Programa de Engenharia de Sistema e Computação, COPPE – UFRJ, Nov 1984.

[22] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, and S. Ghosh. Prism-ii compiler and architecture. In Duncan A. Buell and Kenneth L. Pocek, editors, *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 9–16, Los Alamitos, CA, 1993. IEEE Computer Society Press.

[23] M. Wirthlin and B. Hutchings. A dynamic instruction set computer. In Peter Athanas and Kenneth L. Pocek, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 99–107, Los Alamitos, CA, 1995. IEEE Computer Society Press.