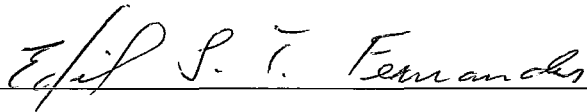


COMPORTAMENTO DA *TRACE CACHE* NUM AMBIENTE
MULTIPROGRAMADO

Álvaro da Silva Ferreira

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.

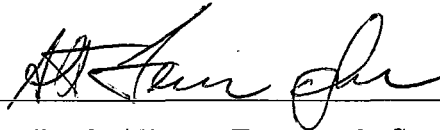
Aprovada por:



Prof. Edil Severiano Tavares Fernandes, Ph.D.



Prof. Valmir Carneiro Barbosa, Ph.D.



Prof. Alberto Ferreira de Souza, Ph.D.

RIO DE JANEIRO, RJ - BRASIL

MARÇO DE 2003

FERREIRA, ÁLVARO DA SILVA

Comportamento da *Trace Cache* num Ambiente Multiprogramado

xii, 101 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2003)

Tese – Universidade Federal do Rio de Janeiro, COPPE

1 - Trace Cache

2 - Memória Cache

3 - Multiprogramação

I. COPPE/UFRJ II. Título (série)

Aos meus pais, Antônio Carlos N. Ferreira e
Cecília da S. Ferreira.

Às minhas avós, Vicentina da Silva e
Lígia Nascimento Ferreira.

À Josilene, que fez tudo valer a pena.

Agradecimentos

Agradeço:

A Deus, por tudo.

Aos meus pais, pela educação, o amor e o apoio incondicionais.

A todos os meus parentes, por sua ajuda, ao longo de toda a minha vida.

A Josilene Beltrame, minha fiel companheira, por seu carinho e compreensão.

Aos meus orientadores, prof Edil Severiano T. Fernandes e prof. Eliseu Monteiro Chaves Filho, pelo tema de trabalho e pelo grande esforço despendido para sua conclusão.

Ao prof. Alberto Ferreira de Souza, pela *workstation* Alpha.

A Christian Daros de Freitas, pelo *loader* ELF para **SimpleScalar**.

A Ralf Baechle, pela ajuda inicial, com o *kernel* Linux para MIPS.

Ao prof. Vítor Santos Costa, pela orientação inicial com o *kernel* Linux.

A Alessandra e Valentim, pelo grande ajuda nos momentos finais.

A Paulo César, pela execução das avaliações no CBPF.

À equipe de suporte de Laboratório da Universidade Estácio de Sá - Campus Terra Encantada, pelo empréstimo dos computadores.

A Marluce, pela preocupação e o empréstimo de sua tese.

A Fagundes, Cremildo, Magno, Marcilene, Moema, Nivaldo, Paulo Sérgio e Valentim, pela acolhida em Laranjeiras.

Ao CNPq, pelo suporte financeiro para conclusão deste trabalho.

Ao corpo técnico docente, administrativo e técnico da COPPE pelo constante suporte ao meu trabalho.

Aos grandes amigos que fiz durante todo o período de mestrado: Aninha, Drica, Kelvin, Paulo Henrique, Marluce e tantos outros não citados por falta de espaço.

A todos aqueles não citados, que direta ou indiretamente, contribuíram neste trabalho.

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

COMPORTAMENTO DA *TRACE CACHE* NUM AMBIENTE
MULTIPROGRAMADO

Álvaro da Silva Ferreira

Março/2003

Orientadores: Eliseu Monteiro Chaves Filho
Edil Severiano Tavares Fernandes

Programa: Engenharia de Sistemas e Computação

Esta tese avalia o comportamento de uma *Trace Cache* num ambiente multiprogramado. Adicionou-se, a um simulador *execution-driven*, a capacidade de simular *Trace Cache* e sessões de multiprogramação, com compartilhamento da *Trace Cache* e da *Cache L1* entre processos.

Foi avaliada a eficiência de uma *Trace Cache* sob variados níveis de multiprogramação, com diferentes conjuntos de programas, num processador Super Escalar com alta taxa de despacho de instruções. A comparação entre o desempenho dos programas, sob multiprogramação e sob monoprogramação, revelou que *Trace Caches* têm sua eficiência negativamente afetada pela multiprogramação. Os resultados deste trabalho identificam a origem das quedas de desempenho e demonstram o equívoco de ignorar a influência da multiprogramação em avaliações de *Trace Cache*.

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

BEHAVIOR OF TRACE CACHE ON A MULTIPROGRAMMED
ENVIRONMENT

Álvaro da Silva Ferreira

March/2003

Advisors: Eliseu Monteiro Chaves Filho
Edil Severiano Tavares Fernandes

Department: Systems Engineering and Computer Sciences

This thesis evaluates the behavior of a Trace Cache on a multiprogrammed environment. The capability of Trace Cache simulation and multiprogramming session simulation, with trace cache and L1 Cache sharing among processes, was added to a execution-driven simulator.

The efficiency of a Trace Cache was evaluated on several multiprogramming levels, with different workloads, on a wide dispatch superscalar processor. The comparison between the program's performance on multiprogramming and monoprogramming has shown that the efficiency of trace caches is adversely affected by multiprogramming. This work identifies the origins of performance losses and shows the mistake of ignoring the multiprogramming influence on Trace Cache evaluations.

Conteúdo

1	Introdução	1
1.1	Motivação	1
1.2	<i>Trace Caches</i>	4
1.2.1	Busca e Decodificação de instruções em Processadores Super Escalares	4
1.2.2	<i>Trace Caches</i>	6
1.2.3	Operação de <i>Trace Caches</i>	7
1.2.4	Organização de <i>Trace Caches</i>	9
1.2.5	Inserção da <i>Trace Cache</i> no Processador Super Escalar	12
1.3	Multiprogramação	15
1.3.1	A Técnica da multiprogramação	16
1.3.2	Multiprogramação e <i>Caches</i>	16
1.4	Trabalhos relacionados	17
2	Plano de Trabalho	19
2.1	Estratégia empregada	19
2.2	Simulação	20
2.2.1	Simulação de <i>Trace Caches</i>	21
2.2.2	Simulação de Multiprogramação	24
2.3	Avaliações Efetuadas	26
2.3.1	Validade das avaliações	28
3	Características da Plataforma de Trabalho	29
3.1	SimpleScalar	29

3.1.1	Visão geral	29
3.1.2	Recursos oferecidos pelo Simulador	30
3.1.3	Funcionamento Interno	32
3.2	SPEC CINT95	37
3.3	Modificações no SimpleScalar	39
3.3.1	Simulação de <i>Trace Cache</i>	39
3.3.2	Simulação de Multiprogramação	41
3.3.3	Outras modificações	48
4	Avaliações e Resultados	51
4.1	Configuração do Processador Simulado	51
4.2	Avaliações	53
4.3	Sessões de multiprogramação	55
4.4	Desempenho da <i>Trace Cache</i> sob Multiprogramação	58
4.4.1	Taxa de <i>miss</i> da <i>Trace Cache</i> sob multiprogramação	62
4.4.2	IPC com <i>Trace Cache</i> e multiprogramação	65
4.5	Evolução do desempenho sob Multiprogramação	66
4.6	Avaliação geral da <i>Trace Cache</i>	69
5	Conclusões e Trabalhos Futuros	74
5.1	Sumário	74
5.2	Trabalhos Futuros	76
A	Penalidades das Chamadas de Sistema	82
B	Valores de IPC por trechos	84
C	Modificações no SimpleScalar	94

Lista de Figuras

1.1	Evolução do desempenho de memórias semicondutoras primárias e de processadores ao longo dos anos.	2
1.2	Esquema simplificado de uma <i>Cache</i>	3
1.3	Diagrama de um processador Super Escalar.	4
1.4	Separação do mecanismo de Execução do de Busca e Decodificação.	5
1.5	Esquema Simplificado de Operação de uma <i>Trace Cache</i>	8
1.6	Elementos básicos de uma <i>Trace Cache</i> e sua inserção no mecanismo de busca e decod. do processador.	9
1.7	Composição de um <i>trace</i>	11
1.8	Exemplo de composição de <i>traces</i> tomando, como exemplo, os <i>traces</i> da Figura 1.5.	13
1.9	<i>Pipeline</i> de um processador Super Escalar com <i>Trace Cache</i>	14
1.10	Exemplo simplificado de Revezamento de processos através de Multiprogramação.	16
3.1	<i>Pipeline</i> da arquitetura PISA implementada pelo SimpleScalar	32
3.2	Organização simplificada do simulador sim-outorder.	33
3.3	Busca de instruções no SimpleScalar com <i>Trace Cache</i>	40
3.4	Compartilhamento da <i>Trace Cache</i> e <i>Cache</i> entre processos simuladores/simulados, através de memória compartilhada.	42
3.5	Exemplo de sincronização dos processos simuladores no acesso à <i>Trace Cache</i> e <i>Cache</i> de instruções L1.	44

3.6	Seqüência de operações até o início da simulação (pelos processos simuladores) e coordenação(pelo processo coordenador).	45
4.1	Número médio de Instruções entre chamadas de Sistema para os aplicativos.	59
4.2	Comparação entre IPCs simulados e estimados a partir da monoprogramação e das substituições de <i>traces</i> interprocessos.	66
4.3	Evolução do IPC dos aplicativos da sessão 6A (GO, LI, IJPEG, GCC, VORTEX e COMPRESS) por trechos.	69
4.4	Evolução do IPC dos aplicativos da sessão 6B (GO, LI, IJPEG, PERL, M88KSIM e COMPRESS) por trechos.	70
4.5	Desempenho relativo médio em relação à monoprogramação em função do nível de multiprogramação.	73
B.1	Evolução do IPC dos aplicativos da sessão 2A por trechos. . .	85
B.2	IPC dos aplicativos da sessão 2B por trechos.	85
B.3	Evolução do IPC dos aplicativos da sessão 2C por trechos. . .	86
B.4	Evolução do IPC dos aplicativos da sessão 2D por trechos. . .	86
B.5	Evolução do IPC dos aplicativos da sessão 2E (M88KSIM & PERL) por trechos.	87
B.6	Evolução do IPC dos aplicativos da sessão 4A por trechos. . .	88
B.7	Evolução do IPC dos aplicativos da sessão 4B por trechos. . .	89
B.8	Evolução do IPC dos aplicativos da sessão 4C por trechos. . .	90
B.9	Evolução do IPC dos aplicativos da sessão 4D por trechos. . .	91
B.10	Evolução do IPC dos aplicativos da sessão 6A (GO, LI, IJPEG, GCC, VORTEX e COMPRESS) por trechos.	92
B.11	Evolução do IPC dos aplicativos da sessão 6A (GO, LI, IJPEG, GCC, VORTEX e COMPRESS) por trechos.	93

Lista de Tabelas

3.1	Descrição dos aplicativos SPEC CINT95 utilizados nesta trabalho.	38
4.1	Configuração do Processador Simulado.	52
4.2	Sessões de Multiprogramação Simuladas.	54
4.3	Número de instruções concluídas e arquivos de entrada para os aplicativos SPEC CINT95.	55
4.4	Quantidade de Trocas de Contexto nas Sessões de Multiprogramação Simuladas.	56
4.5	Quantidade de ciclos de execução simulados nas Sessões de Multiprogramação, com e sem as penalidades das chamadas de sistema.	57
4.6	IPC sob níveis de multiprogramação 1 e 2	61
4.7	IPC sob níveis de multiprogramação 4 e 6	61
4.8	Taxa de <i>miss</i> de <i>Trace Cache</i> sob níveis de multiprogramação 1 e 2	63
4.9	Taxa de <i>miss</i> de <i>Trace Cache</i> sob níveis de multiprogramação 4 e 6	63
4.10	Quantidades de Inserções e substituições de <i>traces</i> nas Sessões de Multiprogramação Simuladas.	64
4.11	Quantidade de substituições de <i>traces</i> interprocessos sob nível de multiprogramação 2	67
4.12	Quantidade de substituições de <i>traces</i> interprocessos sob níveis de multiprogramação 4 e 6	67

4.13 IPC por trechos para os aplicativos da sessão 6A (GO, LI, IJPEG, GCC, VORTEX & COMPRESS).	70
4.14 IPC por trechos para os aplicativos da sessão 6B (GO, LI, IJPEG, M88KSIM, PERL & COMPRESS).	71
4.15 IPC conjunto dos aplicativos, sob monoprogramação e sob nível de multiprogramação 6.	72
A.1 Tipos de penalidades atribuídas às chamadas de sistema do SimpleScalar .	83
B.1 Nomes das Sessões de Multiprogramação.	84
B.2 IPC por trechos para os aplicativos da sessão 2A.	85
B.3 IPC por trechos para os aplicativos da sessão 2B.	85
B.4 IPC por trechos para os aplicativos da sessão 2C.	86
B.5 IPC por trechos para os aplicativos da sessão 2D.	86
B.6 IPC por trechos para os aplicativos da sessão 2E.	87
B.7 IPC por trechos para os aplicativos da sessão 4A.	88
B.8 IPC por trechos para os aplicativos da sessão 4B.	89
B.9 IPC por trechos para os aplicativos da sessão 4C.	90
B.10 IPC por trechos para os aplicativos da sessão 4D.	91
B.11 IPC por trechos para os aplicativos da sessão 6A (GO, LI, IJPEG, GCC, VORTEX & COMPRESS).	92
B.12 IPC por trechos para os aplicativos da sessão 6B (GO, LI, IJPEG, GCC, VORTEX & COMPRESS).	93

Capítulo 1

Introdução

1.1 Motivação

À medida que aumenta a capacidade de processamento por unidade de tempo dos microprocessadores (seja pelo aumento da frequência de operação, seja pelo aumento do número de unidades funcionais), torna-se mais crítica a influência da largura de busca de instruções (em instruções/ciclo) da memória principal. Infelizmente, o tempo de acesso das memórias semicondutoras atualmente utilizadas como armazenamento primário está, pelo menos, uma ordem de magnitude abaixo do tempo mínimo de ciclo dos microprocessadores atuais e esta diferença tende a aumentar, conforme a Figura 1.1 (extraída de [11]). A adoção de memórias com velocidades compatíveis com o processador que a utiliza seria a solução óbvia não fosse pelo seu alto custo por *bit* armazenado em comparação com as memórias mais lentas. Uma estratégia já há muito conhecida e empregada consiste na introdução de um ou mais níveis de armazenamento conhecidos como memórias *Cache* [28, 23]. A idéia da memória *Cache* é “filtrar” o acesso aos outros níveis de armazenamento de maneira que as operações de memória busquem seus operandos na *Cache* e, apenas se não os encontrarem, sejam acessados os níveis menos hierárquicos de armazenamento (Figura 1.2).

Embora eficaz, a organização de *Caches* convencionais ainda não atinge

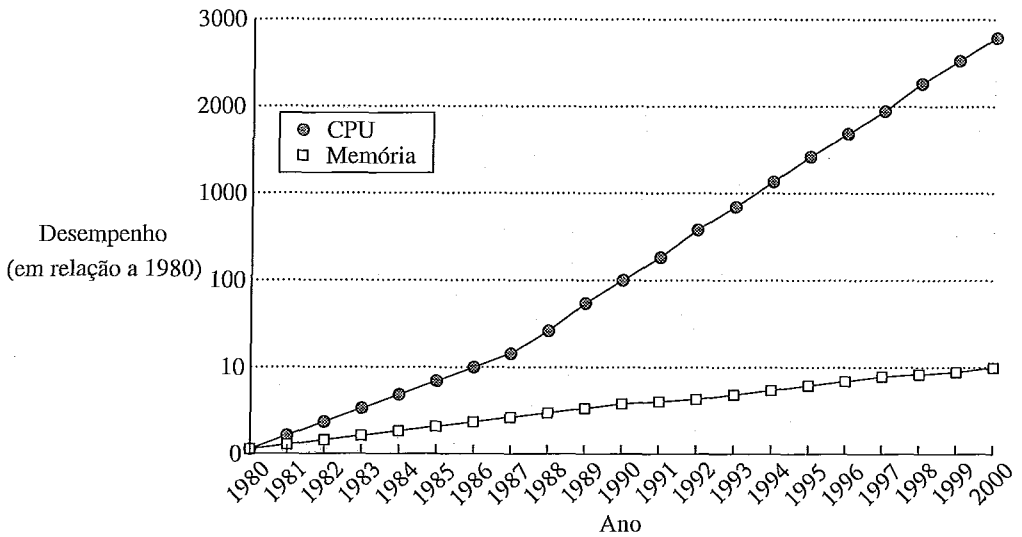


Figura 1.1: Evolução do desempenho de memórias semicondutoras primárias e de processadores ao longo dos anos (tomando como base o desempenho de ambos em 1980).

o máximo de eficiência possível. Para superar o desempenho das *Caches* convencionais no armazenamento e suprimento de instruções para um processador Super Escalar, foi criado o mecanismo *Trace Cache*. Ao invés de armazenar instruções localizadas em posições contíguas de memória (como ocorre com *Caches* convencionais), uma *Trace Cache* armazena seqüências de instruções, na ordem em que elas foram previamente executadas pelo processador. Esta característica da *Trace Cache* evita a queda de desempenho quando o processador busca um grupo de instruções localizado em mais de um bloco da *Cache* convencional (nas arquiteturas convencionais só pode ser buscado um bloco da *Cache* por ciclo).

Comprovadamente, *Trace Caches* superam o desempenho das *Caches* convencionais sem adicionar grande complexidade ao projeto dos processadores. Contudo, todas as avaliações do desempenho de *Trace Caches* até o momento foram realizadas com *benchmarks* sob monoprogramação (uma única aplicação de cada vez). Tal como anteriormente revelado para *Caches* convencionais [1, 14], é natural supor que a avaliação das *Trace Caches*

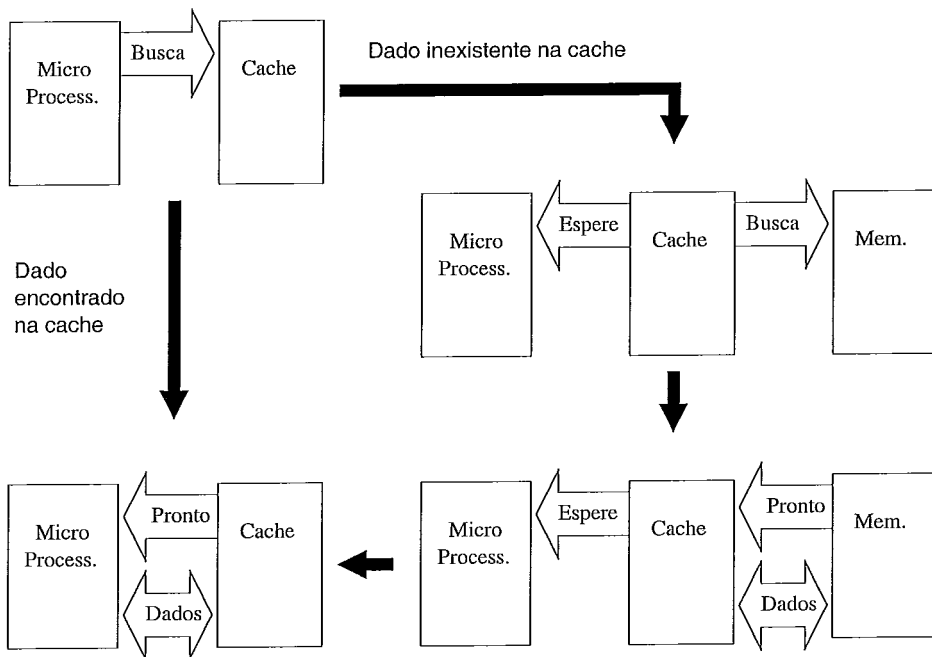


Figura 1.2: Esquema simplificado de uma *Cache*: apenas quando um operando não é encontrado na *Cache*, ele é buscado na memória.

apresente resultados diferentes conforme se passa de um ambiente mono-programado para um multiprogramado. Este trabalho, portanto, procura mostrar alguns efeitos da sessão de multiprogramação no desempenho da *Trace Cache*.

1.2 Trace Caches

1.2.1 Busca e Decodificação de instruções em Processadores Super Escalares

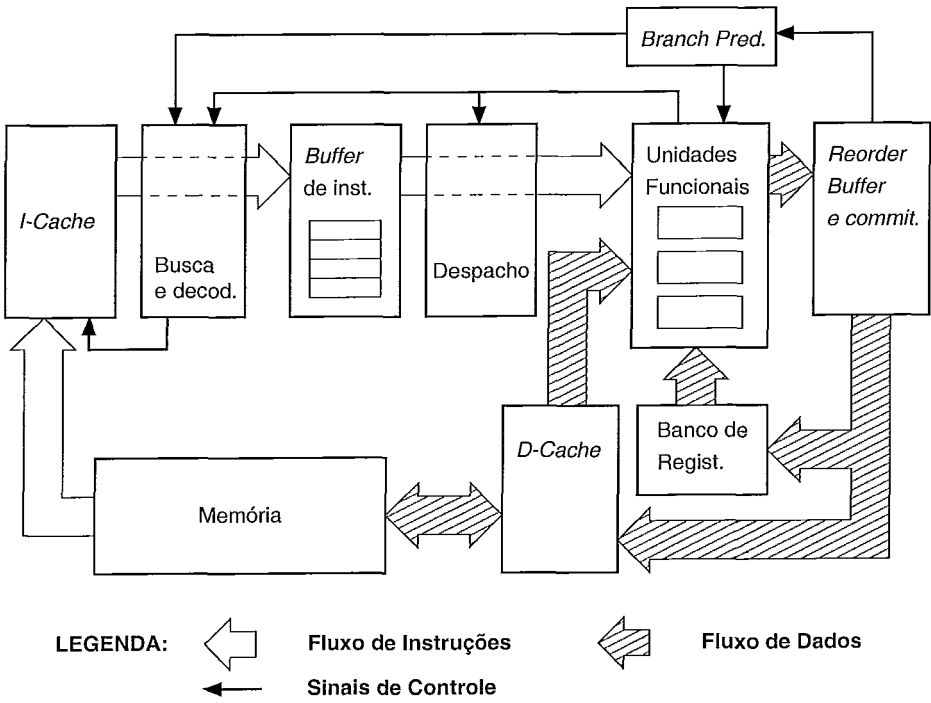


Figura 1.3: Diagrama de um processador Super Escalar.

Processadores Super Escalares são aqueles capazes de iniciar a execução de mais de uma instrução por ciclo de *clock* [24] através do uso de múltiplas unidades funcionais.

Na Figura 1.3, vemos o diagrama simplificado de um processador Super

Escalar convencional; ele contém elementos que, embora dispensáveis, pela definição de processador Super Escalar, são, tipicamente, encontrados nos projetos reais.

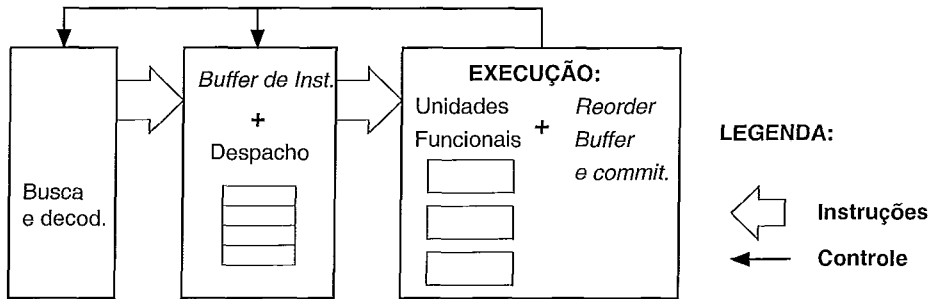


Figura 1.4: Separação do mecanismo de Execução do de Busca e Decodificação.

Conforme descrito por Rotenberg *et al* [19], é possível dividir um processador Super Escalar em dois mecanismos principais: um mecanismo de busca e decodificação de instruções e um outro de execução (ou consumo) de instruções. Atuando como fronteira entre estes dois mecanismos, existe um *buffer* de instruções onde o mecanismo de busca coloca instruções para serem executadas pelo mecanismo de execução (a Figura 1.4, derivada da Figura 1.3 mas exibindo apenas o fluxo de instruções, ilustra esta afirmação). O *buffer* pode ser implementado como fila, *reservation stations*, etc; o fato importante é que seja dada, ao mecanismo de execução, a possibilidade de executar várias instruções em paralelo, satisfeitas as restrições impostas pelas dependências entre estas instruções e a disponibilidade de recursos no processador.

Para manter uma alta taxa de execução de instruções, é necessário que o número de unidades funcionais no mecanismo de execução seja alto e que estas unidades funcionais apresentem uma alta taxa de ocupação. Para isto, além de ser importante ter grandes *Buffers* de instruções, é fundamental que o processador consiga buscar e decodificar instruções numa taxa compatível com a taxa de execução pretendida.

Caches convencionais falham no suprimento de instruções para processadores Super Escalar quando se exigem altas taxas para este suprimento. As limitações se revelam, fundamentalmente, na ocorrência de desvios.

Se um desvio é previsto como tomado e a instrução alvo do desvio não está armazenada no mesmo bloco da instrução de desvio, será preciso um novo ciclo para buscar o bloco da instrução alvo já que *Caches* convencionais entregam um bloco por ciclo ao processador. Ainda que a *Cache* possa entregar mais de um bloco por ciclo (incluindo a instrução de desvio e a instrução alvo), resta o problema de descartar as instruções desnecessárias e alinhar as instruções úteis para que sejam entregues ao processador. Esta não é uma tarefa trivial, exigindo um circuito especializado, aumentando a complexidade da *Cache* e, potencialmente, aumentando a latência do mecanismo de busca/decodificação. Embora seja possível imaginar circuitos razoáveis para o caso de uma única predição de desvio, é difícil conciliar baixa complexidade e baixa latência para buscar/decodificar uma quantidade maior de blocos básicos (que é a tendência esperada para os futuros processadores Super Escalares de alto desempenho). Além do mais, o problema do desalinhamento não se manifesta apenas quando ocorre uma instrução de desvio prevista como tomada; também aparece quando um bloco básico se estende através de mais de um bloco de *Cache*.

Em suma, *Caches* convencionais revelam suas limitações em três aspectos:

- Desvios
- Desalinhamento de instruções em blocos de *Caches*
- Latência do mecanismo de busca/decodificação

1.2.2 *Trace Caches*

Enquanto *Caches* convencionais armazenam, em seus blocos, dados de localização contígua na memória, *Trace Caches* armazenam *traces*, seqüências de instruções temporalmente contíguas; isto é, instruções adjacentes na seqüência (ou fluxo) de execução de instruções. *Trace Caches* revelam suas virtudes,

comparadas às *Caches* convencionais, quando o processador necessita de uma grande vazão de instruções para execução (como acontece em processadores Super Escalares de alto desempenho). Neste caso, as *Caches* convencionais falham em prover a vazão necessária.

Conforme visto anteriormente, a extensão de um bloco básico através de mais de um bloco de *Cache* (seja devido à ocorrência de desvios ou pelo fato de o bloco básico ser muito grande) limita o desempenho das *Caches* convencionais. O fato de *Trace Caches* armazenarem *traces* faz com que elas sejam naturalmente imunes a estes problemas. Num único *trace*, é possível armazenar vários blocos básicos tendo, como restrição, apenas, o número máximo de instruções por *trace* e o número máximo de previsões simultâneas fornecidas pelo preditor de desvios. Desta forma, numa arquitetura contendo a estrutura *Trace Cache*, a ocorrência de desvios não afeta o suprimento de instruções já que estes e as instruções posteriores farão parte do mesmo *trace* e serão entregues, num mesmo ciclo, ao *buffer* de instruções. Pelo mesmo motivo, desaparecerá o problema de desalinhamento de instruções. Quanto à latência do mecanismo de busca/decodificação, foi demonstrado, em [19], que *Trace Caches*, potencialmente, propiciam baixa latência de busca, em virtude de não exigirem circuitos tão complexos quanto as alternativas então sugeridas para resolver o problema do fornecimento de instruções a processadores Super Escalares com alta largura de busca.

1.2.3 Operação de *Trace Caches*

A Figura 1.5 ilustra, de maneira simplificada, o funcionamento conceitual de uma *Trace Cache* (nesse exemplo, cada *trace* contém até dois blocos básicos). Na figura, à esquerda, é mostrada a disposição estática da seqüência do programa; esta é a disposição importante para uma *Cache* convencional: instruções armazenadas contiguamente na memória, muito provavelmente, ocupam um mesmo bloco de *Cache*. No centro, verticalmente, vê-se uma possível seqüência temporal de instruções (o *Fluxo Real de Execução do programa*). À direita, a *Trace Cache* exemplo mostra três *traces* válidos (*traces*

0, 1 e 2), preenchidos a partir do fluxo de instruções. As instruções do fluxo são provenientes, no início, da *Cache* convencional (de onde só podem ser buscadas à taxa de, no máximo, um bloco básico por vez); conforme a *Trace Cache* é alimentada com *traces* provenientes do fluxo de instruções (setas cinzentas, na figura), torna-se possível que uma seqüência de blocos básicos seja proveniente da *Trace Cache* e não da *Cache* convencional (na figura, setas negras, provenientes da *Trace Cache*). A vantagem é que a *Trace Cache* pode fornecer vários blocos básicos num único ciclo, ao contrário da *Cache* convencional.

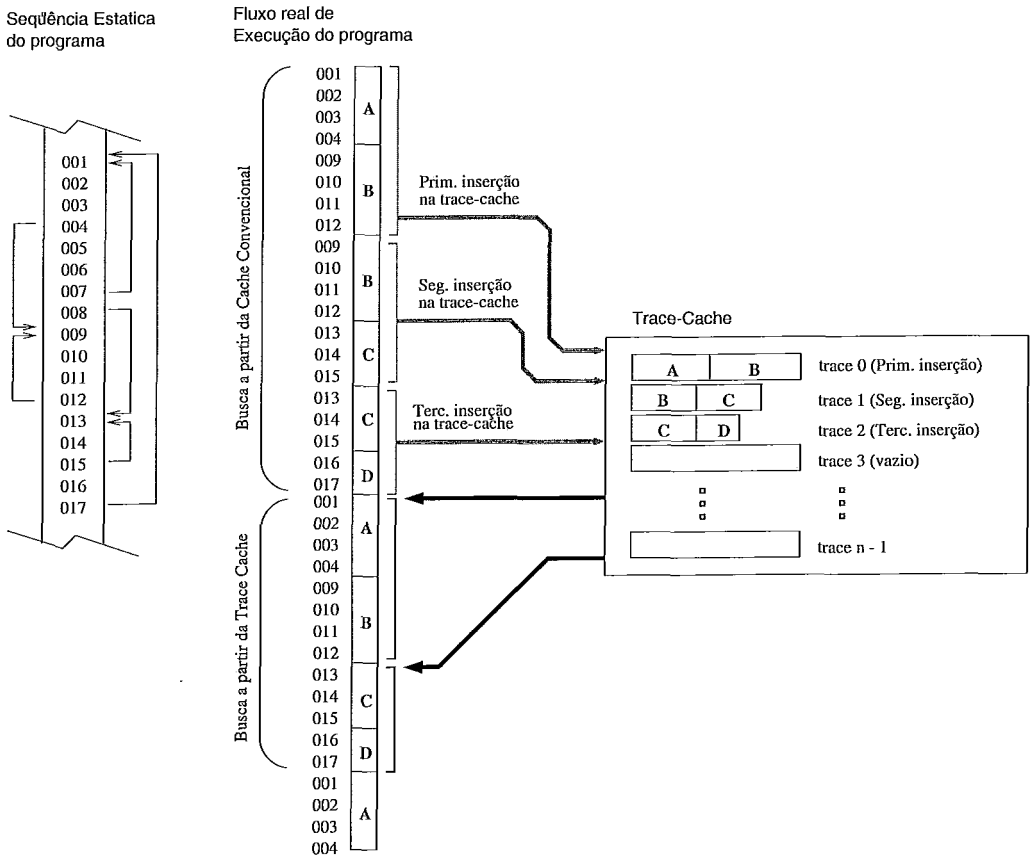


Figura 1.5: Esquema Simplificado de Operação de uma *Trace Cache*.

1.2.4 Organização de *Trace Caches*

Trace Caches são constituídas de dois elementos principais (Figura 1.6): *Fill Unit* e a Área de armazenamento de *traces* (*Trace Area*), em alguns artigos chamada, simplesmente, de *Trace Cache* (embora, rigorosamente falando, seja apenas uma parte do mecanismo de *Trace Cache*). Além destes dois elementos, existe, como acessório a uma *Trace Cache*, um preditor de múltiplos desvios. A introdução de uma *Trace Cache* também exigirá, do mecanismo de busca do processador, um circuito para selecionar, a cada ciclo, entre o *trace* fornecido pela *Trace Cache* ou, caso o *trace* adequado não esteja disponível, o bloco fornecido pela *I-Cache* (*Cache* de instruções).

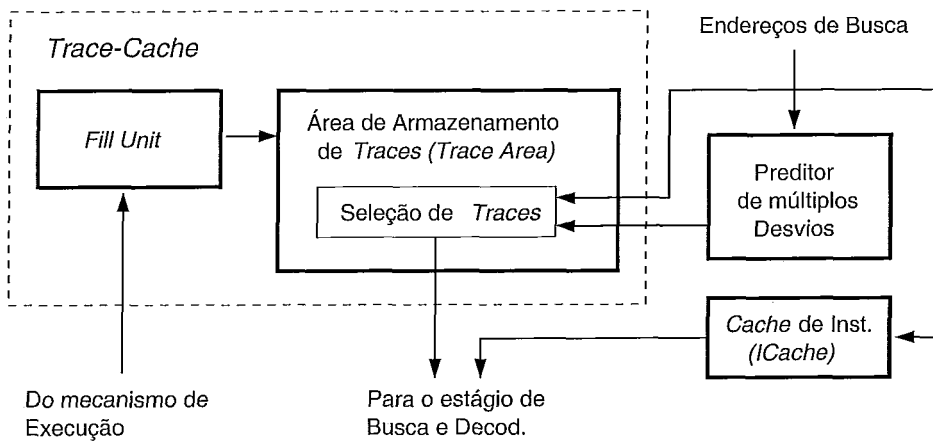


Figura 1.6: Elementos básicos de uma *Trace Cache* e sua inserção no mecanismo de busca e decod. do processador.

Fill Unit

A *Fill Unit* é a unidade responsável pela inserção de novos *traces* na *Trace Cache*. Os *traces* são formados pela concatenação de blocos de instruções executadas pelo processador. Para formação dos *traces* há duas opções: *traces* formados a partir, apenas, de instruções efetivamente executadas pelo processador (isto é, que sofreram *commit*) ou *traces* formados a partir de instruções simplesmente escalonadas para execução (ou seja, incluindo os

blocos executados especulativamente). Conforme Patel *et al* [17], há pouca diferença no desempenho do processador com uma opção ou outra¹.

Formação de *traces*

A formação de *traces* é simples. Conforme as instruções sofram *commit* (ou, opcionalmente, conforme sejam escalonadas para despacho), elas são coletadas pela *Fill Unit* e, na ordem de aparição no fluxo dinâmico de instruções (que pode ser diferente da ordem de conclusão da execução) preenchem um *Buffer* que armazena os *traces* em formação. O *trace* é limitado em relação ao número máximo de instruções que pode conter, bem como em relação ao número máximo de blocos básicos. A limitação com respeito aos blocos básicos é, na verdade, uma limitação quanto ao número máximo de predições de desvios efetuadas por ciclo. A relação entre o número máximo de instruções e o número máximo de desvios depende da frequência de desvios do *workload* desejado para o processador.

Juntamente com a seqüência dinâmica de instruções, deve ser armazenada, no *trace*, o resultado (ou, opcionalmente, as predições) dos desvios do *trace* (se tomados ou não tomados) e o próximo endereço a ser buscado após a entrega do *trace* ao mecanismo de busca do processador.

As instruções formadoras do *trace* recebem tratamento diferente conforme seu tipo. Instruções lógico-aritméticas e quaisquer outras que forcem a busca seqüencial da próxima instrução recebem tratamento normal; elas são incluídas no *trace* até que seja atingido o limite de instruções no *trace*. Instruções de desvio em que, caso a predição seja desvio tomado, o endereço alvo é conhecido previamente, avançam a contagem de desvios no *trace* e são inseridas até que seja atingido o limite de instruções ou o limite de desvios no *trace*. *Traps*, desvios indiretos, instruções de retorno de subrotinas e outras que possuem um número indefinido de alvos (o alvo deverá ser avaliado no momento da execução) abortam a formação do *trace*. O aborto do *trace*, é

¹Neste trabalho, foi escolhida a primeira opção em virtude da facilidade de implementação.

o cancelamento de sua formação, com o descarte das instruções já inseridas no *trace*. Esta é a única alternativa já que o preditor não consegue tratar eficazmente este caso, pois está preparado apenas para indicar se o desvio foi tomado ou não. Chamadas de procedimentos e desvios indiretos poderiam receber um tratamento distinto das demais já que não exigem predição; porém, por motivos de facilidade de implementação², em [19] estas instruções foram tratadas como instruções de desvio, avançando a contagem de desvios, mas cuja predição é totalmente resolvida (ou seja, 100% de acerto).

Área de armazenamento de *traces*

Esta unidade contém, realmente, os *traces* de instruções. Tal como no caso dos blocos de *Caches*, cada *trace* armazenado não é formado, simplesmente, por uma seqüência de instruções. Deve haver mais informações armazenadas com o *trace* para permitir sua recuperação e classificação de maneira adequada. A Figura 1.7 mostra as informações básicas necessárias num *trace* (adaptadas de [19]).

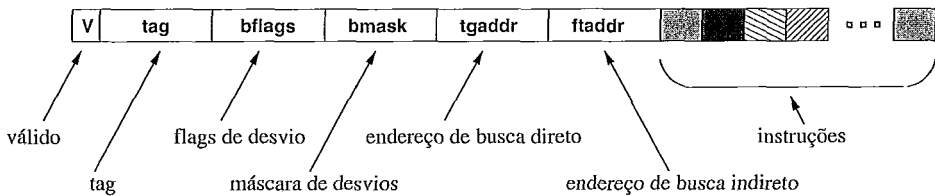


Figura 1.7: Composição de um *trace*.

A descrição dos campos da Figura 1.7 é:

válido é um *flag* que indica se o *trace* é válido ou não.

tag é o endereço da primeira instrução do *trace*.

flags de desvio indicam o resultado dos desvios (tomados ou não tomados) dentro do *trace*.

²A simplificação na implementação foi tratar este caso como um dos anteriormente citados, ao invés de dar-lhe um tratamento distinto.

máscara de desvios indica quais instruções do *trace* são instruções de desvio (indiretamente, informa a quantidade de desvios no *trace*).

endereço de busca direto é o próximo endereço de busca após o uso do *trace*, caso seu último desvio seja previsto como não tomado (de acordo com os flags de desvio).

endereço de busca indireto será usado como próximo endereço de busca caso o último desvio do *trace* seja previsto como não tomado.

instruções formam a carga útil, a seqüência de instruções que forma, realmente, o *trace*.

Na Figura 1.8, pode-se ver como seria a composição de dois *traces* tomando, como exemplo, aqueles da Figura 1.5.

Tal como para as instruções em *Caches* convencionais, os *traces* são agrupados em conjuntos para fim de busca, permitindo, às *Trace Caches*, diferentes graus de associatividade. Também as *Trace Caches* aceitam diferentes políticas de substituição; porém, ao contrário das *Caches* convencionais, a substituição de um *trace* não ocorre concomitantemente com a sua busca (quando um bloco não era encontrado na *Cache*, ele era buscado do nível seguinte da hierarquia de memória e substituía algum bloco pré-existente). Na *Trace Cache*, a substituição de blocos é feita exclusivamente pela *Fill Unit* e ocorre de acordo com a seqüência de execução de instruções e o preenchimento do novo *trace* a ser inserido na *Trace Cache*.

1.2.5 Inserção da *Trace Cache* no Processador Super Escalar

A Figura 1.9 mostra a inserção da *Trace Cache* no diagrama de um processador Super Escalar. Em relação à Figura 1.3, há o acréscimo da *Trace Cache* (identificada como *T-Cache*), uma via de instruções entre o estágio de despacho e a *T-cache* e as conexões entre esta última, o *reorder buffer* e o preditor de desvios. A mudança importante em relação ao funcionamento

Sequencia Estatica do programa

Fluxo real de Execucao do programa

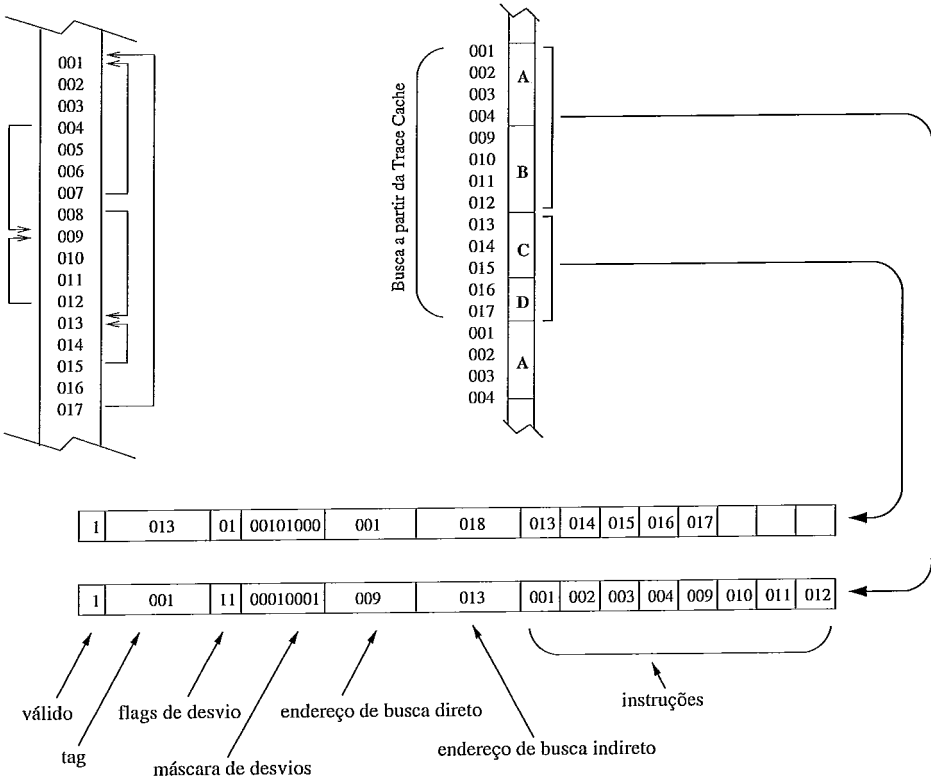


Figura 1.8: Exemplo de composição de *traces* tomando, como exemplo, os *traces* da Figura 1.5.

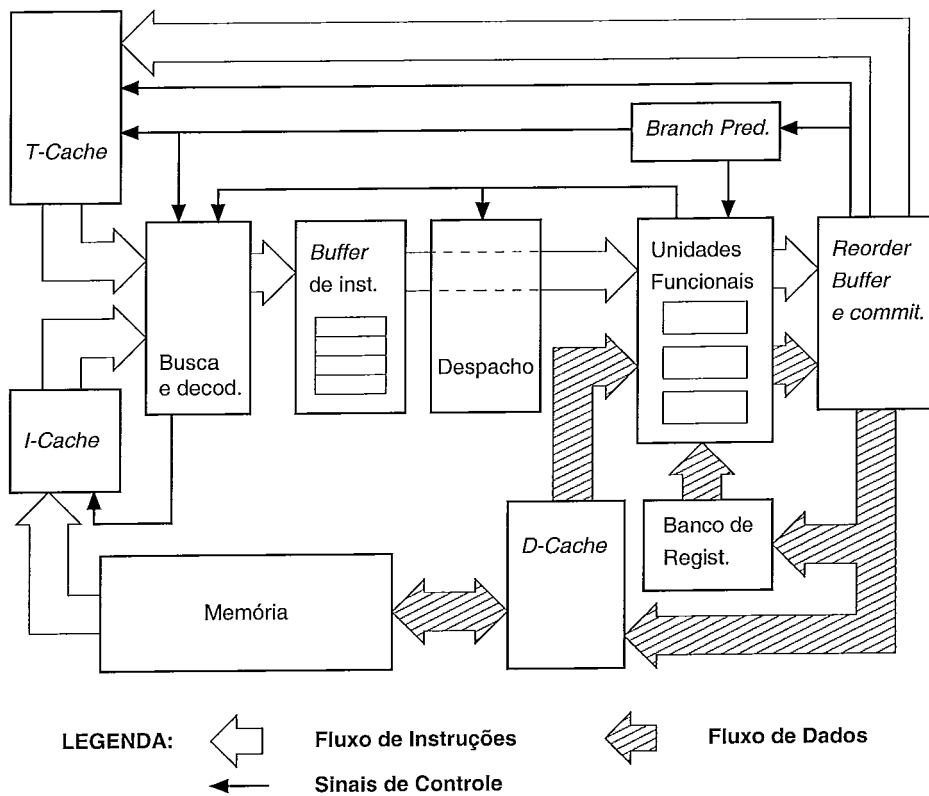


Figura 1.9: Pipeline de um processador Super Escalar com Trace Cache.

normal de um processador escalar é que, agora, as instruções para o estágio de busca e decodificação podem provir da *I-Cache* ou da *T-Cache*. De resto, há, apenas, o funcionamento da *Trace Cache* como novidade.

Conforme as instruções sofram *commit* (ou sejam despachadas para execução), elas são mandadas para a *Trace Cache*, mais especificamente para a *Fill Unit*. Para inserir adequadamente os *traces* na *Trace Cache*, a *Fill Unit* precisa, além das instruções, das predições de desvio e das informações sobre o *commit* das instruções.

1.3 Multiprogramação

Multiprogramação consiste em permitir a execução concorrente de várias tarefas (chamadas de *processos*), cada uma com seu espaço de endereçamento e fluxo de execução próprios, através do revezamento no uso do processador. Comumente, os usuários terão a ilusão de que vários programas estão sendo executados simultaneamente. Além do mais, cada processo “imaginará” ser a única aplicação sendo executada na máquina.

A característica chave da multiprogramação é o fato de o uso do processador ser compartilhado por vários programas em execução (inclusive o sistema operacional) através de revezamento. A cada processo, é concedido um período máximo de tempo (da ordem de dezenas de milissegundos) ao fim do qual o controle do processador é reassumido pelo sistema operacional que, então, escolherá outro processo³ que irá usar o processador.

Neste trabalho, empregaremos a multiprogramação como empregada em sistemas tipo Unix [27]. Adicionalmente, a maior parte dos trabalhos similares a este foi conduzida em plataformas semelhantes o que facilita comparações entre o resultado deste trabalho e os resultados de seus precedentes.

³A mistura dos termos *processo* e *programa* não quer dizer que eles sejam sinônimos. O termo *programa* foi usado porque esta é a entidade com a qual maior parte dos usuários está familiarizada. Na verdade, um único programa sendo executado pode contar com vários processos cooperando para a execução de alguma tarefa.

1.3.1 A Técnica da multiprogramação

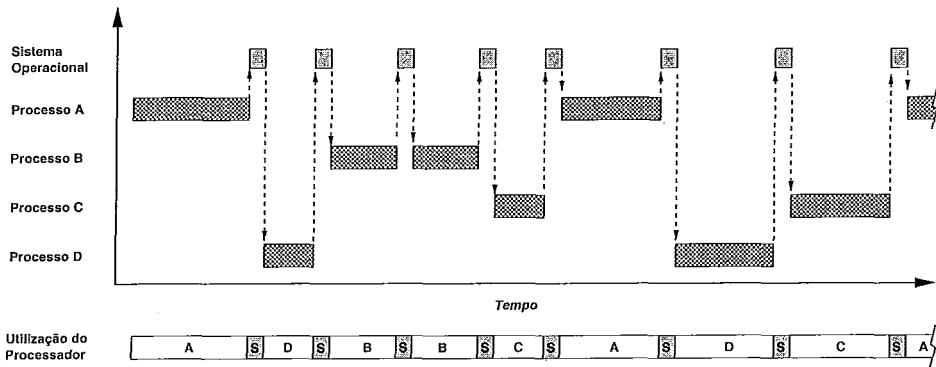


Figura 1.10: Exemplo simplificado de Revezamento de processos através de Multiprogramação.

A Figura 1.10 ilustra a idéia de multiprogramação; nesta figura, quatro aplicações (cada uma delas representada por um processo) revezam-se no uso do processador. Após cada processo, o Sistema Operacional assume o controle do processador para executar tarefas do sistema (inclusive a escolha do novo processo ao qual será cedido o processador).

1.3.2 Multiprogramação e Caches

Neste trabalho, multiprogramação é importante devido à sua influência no desempenho de programas em comparação com a situação em que estes programas são executados como única aplicação controlando o processador. Esta última situação é aquela normalmente encontrada em trabalhos anteriores sobre *Trace Cache*.

Do ponto de vista de um processo isolado, a multiprogramação pode trazer desvantagens. Sem multiprogramação, um processo tem o processador totalmente à sua disposição; com multiprogramação, ele tem que dividir o tempo do processador com outros processos. Um processo pode ser prejudicado, também, pela menor eficiência de outros recursos tais como *Cache*, *TLB's*, etc.

Um dos recursos cujo desempenho é afetado pela multiprogramação é a memória *Cache*, conforme já revelado em trabalhos anteriores [1, 14]. A interferência ocorre através da atividade do Sistema Operacional (agindo em prol do próprio processo e/ou do Sistema como um todo) e, também, através da atividade dos demais processos, também utilizando a *Cache*.

Neste trabalho, será considerada, apenas, a influência da alternância de processos sobre a *Trace Cache*. Tal como mostrado por trabalhos anteriores (em [1, 14], para *Caches* convencionais), é esperado que esta influência seja relevante.

Para *Caches* convencionais, a influência da multiprogramação se faz sentir cada vez que um processo reassume o processador; quando isso ocorre, aquele processo, apesar de já ter utilizado o processador, estará sujeito a alguns *misses* decorrentes de invalidações de blocos da *Cache* causadas pelos outros processos. Embora estes sejam *misses de conflito*, para o processo, isoladamente, será como se ele incorresse, a cada fatia de tempo, em novos *misses* compulsórios (afinal, ele “imagina” que é a única aplicação utilizando o processador). Por outro lado, ao colocar seu *working set* na *Cache*, um processo desaloja dados utilizados pelos outros processos. Estes outros processos, ao reassumirem o processador, enfrentarão os mesmos tipos de *misses*.

É importante notar que as desvantagens citadas ocorrem, apenas, quando se considera o ponto de vista de uma única aplicação. Na verdade, para o sistema como um todo, a multiprogramação é favorável já que evita a ociosidade do processador (quando um processo espera pelo fim de alguma operação de *I/O*, por exemplo) e facilita a cooperação entre processos.

1.4 Trabalhos relacionados

Embora já existisse uma patente descrevendo o mecanismo da *Trace Cache* [18], ele foi, pela primeira vez, avaliado por Rotenberg *et al* [19], descrevendo-o como um mecanismo de baixa latência para o suprimento de instruções

a processadores Super Escalares de alta largura de despacho. Patel *et al* realizaram um série de trabalhos analisando questões importantes no projeto de *Trace Caches* [16], explorando as opções de projeto para *Trace Caches* [17], e propondo aperfeiçoamentos [15].

Outros trabalhos sobre *Trace Caches* se concentram em seu papel em *trace processors* [20], (onde se faz uma análise do desempenho da *Trace Cache* mostrando suas taxas de *miss* para diferentes configurações). Black *et al* [5] atacam o problema da eficiência da *Trace Cache* no armazenamento de *traces* propondo uma *Trace Cache* organizada em torno de blocos básicos.

Alguns trabalhos anteriores exploraram a interação entre o sistema operacional e vários recursos do processador. Anderson *et al* [3] analisaram tendências no projeto de processadores e sistemas operacionais. Torrelas *et al* [26] analisaram o efeito da execução de aplicativos fazendo uso intensivo do sistema operacional e descobriram que para aplicativos não otimizados, 40% das referências e mais de 80 % dos *misses* são de responsabilidade do sistema operacional.

Analisando o efeito da alternância de processos, Agarwal *et al* [1] analisaram a eficiência de *Caches* convencionais para código de sistema e de usuário, sobre variados níveis de multiprogramação encontrando aumentos de até 75% na taxa de *miss* de estado estacionário.

Mogul e Borg [14], avaliaram o efeito das trocas de contexto sobre a taxa de CPI do processador, assumindo que a queda de desempenho fosse devida, unicamente à *misses* da *Cache*. Eles encontraram *overheads* de até 7,8% devido às trocas de contexto.

Não são conhecidos trabalhos explorando a influência da multiprogramação sobre o desempenho de *Trace Caches*; nem mesmo a interação entre *Trace Cache* e sistema operacional. Contudo, há estudos sobre a alternância de processos sobre a precisão de preditores de desvio [13], os quais são fundamentais para o bom desempenho de *Trace Caches*.

Capítulo 2

Plano de Trabalho

2.1 Estratégia empregada

Este trabalho se propôs avaliar o efeito de multiprogramação no desempenho de uma arquitetura com *Trace Cache*. A maneira escolhida para se atingir esse objetivo foi:

1. Utilizar uma plataforma de avaliação na qual se modelou:
 - (a) Um processador Simulado.
 - (b) Uma *Trace Cache* integrada ao processador simulado.
 - (c) A capacidade de executar, simultaneamente, vários aplicativos de *Benchmark* no processador simulado.
 - (d) Um mecanismo para controlar o acesso dos aplicativos do *Benchmark* à *Trace Cache* modelada.
2. Simular uma sessão de multiprogramação usando os aplicativos de *Benchmark* sobre a plataforma de avaliação e extrair, desta simulação, os dados desejados.

2.2 Simulação

A simulação tem a vantagem de facilitar a variação das condições hipotéticas de trabalho tanto do processador quanto da *Trace Cache*. A alternativa à simulação seria extrair as informações de desempenho da *Trace Cache* diretamente do *hardware* da plataforma empregada, no que podemos chamar de instrumentação direta. Esta alternativa já foi empregada em avaliações semelhantes (vide, por exemplo, [8]) mas, no caso presente, apresenta, como dificuldades, a inexistência de processadores comerciais com *Trace Caches*¹ e a impossibilidade de extrair todas as informações significativas das plataformas mais atuais. Como vantagens da instrumentação direta, podemos citar a rapidez na obtenção dos resultados e a fidelidade dos mesmos. Como desvantagens principais, podemos citar as limitações quanto às configurações e condições de teste que a plataforma poderia assumir.

A respeito da simulação, ainda existe a possibilidade de escolher entre duas modalidades: *trace-driven* ou *execution-driven*. Na simulação *trace-driven* o simulador não executa, realmente, as instruções; ele apenas segue os “rastros” da execução real de um programa, realizada por uma máquina real. Nesta modalidade de simulação, portanto, é necessário obter, previamente, um *trace* da execução real de um programa. Este *trace* contém a seqüência de endereços de instruções executadas, bem como os endereços das posições de memória acessadas pelo programa para leitura ou escrita de dados.

Na simulação *execution-driven*, simula-se, com mais detalhes, a arquitetura interna do processador e, principalmente, fica a cargo do simulador realizar os mesmos cálculos que o processador real faria e tomar decisões acerca de desvios, por exemplo.

¹Quando do início deste trabalho, o microcontrolador Élan SC520 da AMD [2] já empregava *Trace Cache*; porém, seu uso de *Trace Cache* é voltado para depuração e não para aceleração do desempenho “*on the fly*” Atualmente, o processador Pentium 4, da Intel, possui *Trace Cache* de uma maneira mais próxima à descrita neste trabalho.

Implementação da simulação

Neste trabalho, realizou-se simulação *execution-driven* do processador, oferecida pelo simulador **SimpleScalar** 3.0c [7]. A escolha da modalidade de simulação *execution-driven* foi feita devido à necessidade de modelar não só o acesso às *Trace Caches* mas, também, seu funcionamento interno.

Para realizar a simulação pretendida, foi necessário efetuar modificações no código do **SimpleScalar**. Estas modificações podem ser classificadas em duas categorias:

1. Modificações para simulação de *Trace Caches*.
2. Modificações para simular a Multiprogramação.

No Capítulo 3, exploram-se com mais detalhes as modificações efetuadas.

2.2.1 Simulação de *Trace Caches*

A configuração de *Trace Cache* adotada neste trabalho foi inspirada no trabalho original sobre *Trace Caches*, por Rotenberg *et al* [19], e no trabalho de Patel *et al* [17]. Algumas das características da configuração, contudo, foram adotadas tendo em vista a facilidade de implementação no **SimpleScalar**. Deve-se notar que não é objetivo deste trabalho realizar uma avaliação de *Trace Caches*, mas sim, avaliar o efeito da multiprogramação sobre ela. Resumindo, as principais características adotadas na implementação de *Trace Caches* foram:

- Instruções são coletadas para formar *traces* após o *commit*.
- Instruções que invocam *syscalls* provocam o aborto (ou invalidação) de um *trace* sob formação ou, opcionalmente, o término do *trace* na instrução anterior.
- São utilizadas as mesmas opções de predição nativas do simulador exceto pela ausência do preditor perfeito (decidimos não implementá-lo em conjunto com *Trace Caches*).

- O tamanho máximo do *trace* é dado pelo número máximo de instruções na fila de busca do processador simulado.
- As instruções são buscadas, primeiro, na *Trace Cache* e, depois, na *Cache* L1 (apenas se não forem encontradas na *Trace Cache*). Se não estiverem na *Cache* L1, elas são procuradas na *Cache* L2 e, se não encontradas, são buscadas na memória.
- Utiliza-se coincidência parcial (*partial matching*) na busca de *traces* (se as predições efetuadas não coincidem com o caminho no *trace*, este será aproveitado até onde ocorrer a discordância da predição).
- A identificação dos *traces* (**TAG**) inseridos na *Trace Cache* é dada pelo endereço da primeira instrução do *trace*.
- Apenas um *trace* é inserido na *Trace Cache* por ciclo.

Além da configuração da *Trace Cache*, é de interesse a configuração do processador simulado nas avaliações efetuadas já que este interfere profundamente no desempenho obtido (afinal, a *Trace Cache* só modifica, diretamente, a busca de instruções). A configuração adotada para o processador simulado foi tal que atendessem às seguintes exigências:

- O processador deve ser capaz de processar, pelo menos, 16 instruções nas suas unidades funcionais, simultaneamente.
- Até 16 instruções devem ser despachadas por ciclo, atendidas suas necessidades quanto à disponibilidade dos operandos.
- Os demais elementos do processador (e os subsistemas acoplados a ele, tais como memória, *Cache* de dados, etc) devem ser modelados tomando-se configurações “agressivas”, de modo a não limitar excessivamente o desempenho do processador.

Com estas exigências, o estágio de busca será o “gargalo” no desempenho do processador. Com estas limitações no estágio de busca e recursos abundantes nos outros estágios, a responsabilidade pelo desempenho do processador

pode ser atribuída, diretamente, ao estágio de busca e, por conseguinte, à eficiência da *Trace Cache*.

A configuração adotada possui dois pontos notáveis:

1. Quando é encontrada uma instrução que provocaria o aborto do *trace* sob formação, segundo o modelo descrito originalmente [19], verificam-se o número de instruções e o número de desvios já presentes no *trace*. Caso haja 8 ou mais instruções ou, pelo menos, um desvio, ao invés de se abortar a formação do *trace*, este será aproveitado até a última instrução previamente inserida.
2. A busca de instruções, a partir da *Trace Cache* ou *Cache* convencional, num determinado ciclo, só ocorre caso a fila de instruções buscadas esteja vazia ao início daquele ciclo.

O objetivo do item 1 acima é aproveitar, parcialmente, *traces* que, de outro modo, seriam desperdiçados. Porém, destes *traces*, selecionam-se aqueles mais vantajosos para inserção (ou seja, *traces* longos ou que compreendem mais de um bloco básico) os quais não seriam facilmente obtidos a partir da *Cache* convencional. A escolha destes *traces* mais vantajosos, pelo fato de diminuir a taxa de inserção de novos *traces*, também diminui a probabilidade de se desalojar um *trace* útil já presente na *Trace Cache*. A quantidade mínima de instruções, 8, foi escolhida em função da configuração empregada nas avaliações; não é garantido que estas medidas ou que este valor sejam eficazes para toda e qualquer configuração de *Trace Cache*.

A providência descrita no item 2 acima ocorre pelo fato de o **SimpleScalar**, originalmente, retirar instruções da fila, uma a uma, conforme as decodifica e despacha. A sobra de instruções na fila de busca, porém, penaliza a *Trace Cache* que, mais freqüentemente, é capaz de preencher totalmente a fila de busca. Já as *Caches* convencionais, conforme o funcionamento adotado no **SimpleScalar**, fornecem, no máximo, a exata quantidade de instruções necessárias para preenchimento da fila de busca.

2.2.2 Simulação de Multiprogramação

Para simular a multiprogramação, implementou-se o acesso compartilhado à *Cache* de instruções L1 e à *Trace Cache*. Os demais recursos do processador simulado (*Cache* L2, preditores, etc) permaneceram privados para cada um dos processos de *benchmark* simulados.

Para cada um dos processos participantes da sessão de multiprogramação simulada (ou seja, cada processo simulado), criou-se um processo simulador, constituído do **SimpleScalar** “rodando” aquele processo simulado. Cada um dos processos simulados acessava sua própria *Trace Cache* e *Cache* de instruções L1; porém, estas foram implementadas numa região de memória compartilhada entre os processos simuladores. O compartilhamento de memória entre os processos simuladores ocorreu através de uma modalidade de comunicação interprocessos para sistemas tipo Unix (*System V IPC* [10]) conhecida como *shared memory*.

Um processo especial (processo coordenador) foi responsável, durante a simulação, por criar os processos simuladores e coordenar o acesso à *Trace Cache* e *Cache* L1, de modo a reproduzir o comportamento do escalonador de processos de um Sistema Operacional tipo Unix. Neste trabalho, o algoritmo de escalonamento foi derivado daquele utilizado pelo Sistema Operacional Linux 2.0.33 [21, 22, 4, 6], para processos “normais”².

O algoritmo pode ser sumarizado nos seguintes pontos:

- A unidade básica de tempo para as atividades do kernel Linux (e, portanto, do escalonador de processos) é o *jiffie*³. A passagem de um período de tempo igual a 1 *jiffie* é marcada pela ocorrência de uma interrupção do relógio.
- Cada processo possui uma prioridade, expressa por um número inteiro. Cada processo também possui um peso⁴ que indica por quanto tem-

²“Normais” são os processos não classificados como processos de tempo real.

³Um *jiffie* corresponde, normalmente, a 10ms.

⁴No código fonte de Linux esta variável é identificada pelo campo *counter* da estrutura

po (expresso em quantidade de *jiffies*) aquele processo pode ocupar o processador antes que seu peso seja recalculado.

- Cada processo, inicialmente, tem permissão para ser executado por um certo intervalo de tempo (por *default*, 20 *jiffies*).
- Para a maioria das chamadas de sistema (*syscalls*) e para cada interrupção lenta⁵, o escalonador (*scheduler*) de processos do sistema escolhe o processo que terá a posse do processador até a próxima vez que o escalonador for invocado.
- Os processos aptos a rodar são postos numa fila única de execução. Quando um processo solicita uma operação que não pode ser atendida imediatamente (tipicamente, operações de Entrada/Saída) este processo deixa de ser “apto a rodar” e é posto numa fila de espera “dormindo” até a ocorrência do evento que o tirará da fila de espera.
- O escalonador escolhe o próximo processo para qual cederá o processador com base no peso do processo. O processo (apto a rodar) momentaneamente com o maior peso (isto é, que tenha ocupado o processador por menos tempo, assumindo-se que todos os processos iniciem com mesmo peso) será escolhido. Se o último processo escolhido “empata”, em peso, com outros de mesma prioridade, ele permanecerá com o processador.
- Quando todos os processos aptos a rodar têm seu peso reduzido a zero, os pesos serão recalculados (para todo os processos) , atribuindo-se um valor inicial igual ao de suas prioridades.

task_struct do processo; o nome peso, portanto, não corresponde à tradução literal de *counter*, mas sim à função da variável. No código do escalonador, há uma variável *weight*, cujo valor é calculado a partir de *counter* o que justifica o nome *peso* usado neste trabalho.

⁵Interrupções lentas, no contexto do Sistema Operacional Linux, são as interrupções cujo tratamento é demorado, pelos padrões do sistema, e que, por isso, provocarão a ação do escalonador.

Em Linux, a maioria das chamadas de sistema provoca ação do escalonador. Dentre estas, aquelas relacionadas com operações de *Entrada/Saída*, freqüentemente, provocam a transferência do processo que originou a chamada para uma fila de espera. Neste trabalho, os equivalentes, em Linux, às chamadas de sistema oferecidas pelo **SimpleScalar** tiveram seus códigos fonte inspecionados para determinar quais não deveriam provocar a ação do escalonador simulado. Uma simplificação foi efetuada no tratamento de operações de *Entrada/Saída*: assumiu-se uma penalidade, em ciclos, para a operação de abertura de arquivos e outra penalidade para as subseqüentes operações de acesso a estes arquivos (leitura, escrita, reposicionamento, etc). As penalidades são expressas em microssegundos, transformados, pelo simulador, em quantidade de ciclos de máquina. Quando um processo simulado é penalizado, ele perde o processador e passa a esperar um evento que ocorrerá quando os ciclos correspondentes à penalidade já decorreram. Nesta ocasião, o escalonador “acorda” o processo e o inclui na lista de execução.

A necessidade de simular o escalonamento de processos de Linux, bem como a existência de penalidades, ambos dependentes de valores expressos em unidades reais de tempo, levou à necessidade de se introduzir um *clock* equivalente como parâmetro de simulação. Originalmente, o **SimpleScalar** utiliza, apenas, ciclos de máquina como padrão de contagem de tempo. Após as modificações introduzidas, há conversões entre valores de tempo e números de ciclos de máquina em função do valor de *clock* assumido para o processador.

2.3 Avaliações Efetuadas

Para avaliar o efeito da multiprogramação sobre *Trace Caches*, foram escolhidos os seguintes parâmetros:

- Taxa de *miss* de *traces* da *Trace Cache* (*miss rate*, a razão entre o número de *hits* de *traces* e o número de acessos à *Trace Cache*).
- Taxa de conclusão de instruções, em instruções por ciclo (IPC).

A cada ação do escalonador de processos simulado, tomaram-se medidas das estatísticas parciais de simulação que permitissem avaliar os parâmetros acima. Estes valores parciais propiciaram uma visão da evolução destes valores ao longo da simulação.

O IPC medido é uma valor que depende não só do desempenho da *Trace Cache* mas, também dos outros recursos simulados pela plataforma (*Caches*, preditores, unidades funcionais, etc). Adotou-se, para a configuração da plataforma, parâmetros que depositassem, principalmente sobre o estágio de busca, a responsabilidade pelo IPC obtido (Seção 2.2.1), tendo em vista uma taxa máxima de busca de 16 instruções por ciclo. Desta forma, o desempenho dos aplicativos passou a ser uma medida direta da eficiência da *Trace Cache*. O máximo IPC possível, igual a 16, foi escolhido por ser um valor recorrente em trabalhos anteriores sobre *Trace Cache* (inclusive no trabalho original sobre *Trace Cache* [19]).

Os aplicativos postos a simular foram escolhidas do conjunto de programas de *benchmark* para operações com números inteiros SPECINT95. O uso de aplicativos para números inteiros se justifica pelo fato de esta classe de aplicativos exigir mais do mecanismo de busca quanto ao adequado tratamento de instruções de desvio.

As avaliações foram efetuadas para uma configuração específica de *Trace Cache* e repetidas para níveis de multiprogramação iguais a 1, 2, 4 e 6. Com nível de multiprogramação 1, simulou-se o funcionamento da *Trace Cache* com cada programa de *benchmarking* isoladamente. Em seguida, aumentou-se progressivamente o nível de multiprogramação agrupando-se processos simulados aos pares⁶, depois de quatro em quatro e, por fim, 6 processos numa única sessão. As simulações foram efetuadas considerando-se que os processadores simulados possuíam *clock* igual a 800 MHz. A descrição dos testes efetuados, das configurações e dos resultados encontra-se no Capítulo 4.

⁶Nem todas as combinações entre processos foram experimentadas.

2.3.1 Validade das avaliações

Deve-se levar em conta algumas limitações importantes deste trabalho:

1. Não foi levado em conta o efeito da execução do código do sistema operacional. Trabalhos anteriores obtiveram, com um sistema operacional tipo Unix, taxas de *miss*, para *Caches L1*, variando entre 52% e 64% [1] da taxa de *miss* total.
2. Não foram levados em conta outros processos que não aqueles do conjunto de aplicativos SPEC.
3. Só foi simulado o acesso, por vários processos, à *Cache* de instruções L1 e à *Trace Cache*. Não foi simulado o acesso a preditores de desvio, RAS (*Return Address Stack*) ou BTB (*Branch Target Buffer*) compartilhados entre processos.
4. Foram atribuídas penalidades fixas para as operações de Entrada/Saída. Na prática, não ocorre assim; as penalidades deveriam variar amplamente em função da solicitação e do momento em que é realizada a operação de Entrada/Saída.

Apesar destas limitações, este trabalho continua mantendo sua validade já que:

- Não foram publicados trabalhos similares para *Trace Caches*.
- As demais avaliações de *Trace Caches* foram tão ou mais limitadas no tocante à influência do sistema operacional.
- É objetivo do sistema operacional tornar tão grande quanto possível o tempo de CPU dos processos em relação ao tempo de CPU do próprio sistema.
- Os aplicativos SPEC gastam relativamente pouco tempo de execução nas chamadas de sistema (o objetivo do conjunto de aplicativos SPEC 95 não é avaliar o desempenho do sistema operacional [9]).

Capítulo 3

Características da Plataforma de Trabalho

3.1 SimpleScalar

3.1.1 Visão geral

O simulador utilizado ao longo deste trabalho foi baseado na versão 3.0c do “SimpleScalar Tool Set”, uma versão evoluída da edição 2.0 [7] (doravante, o simulador e o conjunto de ferramentas associados serão chamados, apenas, **SimpleScalar**). As ferramentas distribuídas com o **SimpleScalar** compreendem:

- Conjunto de Simuladores para duas arquiteturas: PISA (Portable ISA) e Alpha.
- Compilador GCC-2.6.3 para a arquitetura PISA
- Tradutor de Fortran para Linguagem C
- Arquivos binários SPEC95 para a arquitetura PISA
- Arquivos para teste dos simuladores (ambas arquiteturas)
- Utilitários GNU Binutils-2.5.2 para a arquitetura PISA

