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 desprendido 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 demostram 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

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação

À medida que aumenta a capacidade de processamento por unidade de tempo dos microprocessadores (seja pelo aumento da freqüência 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

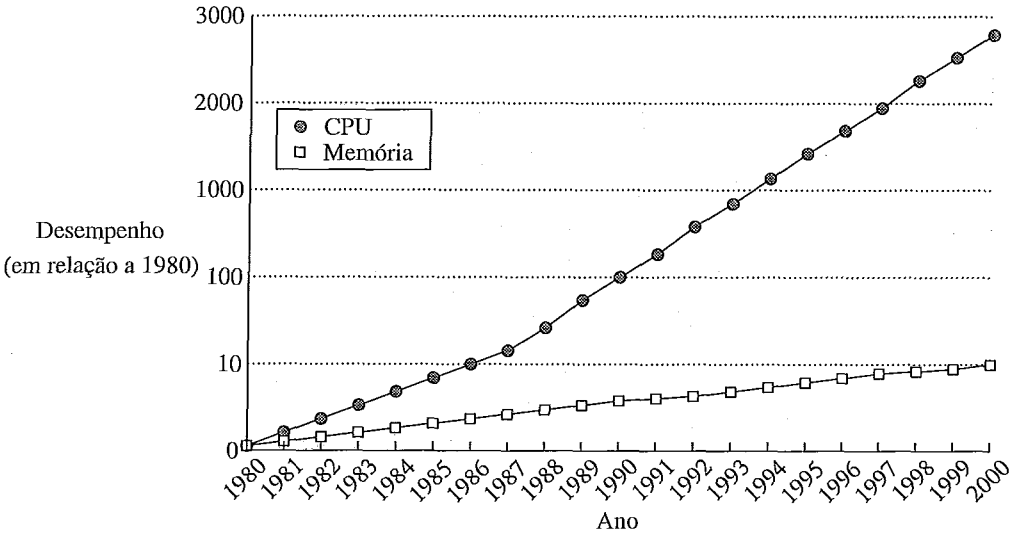

Figura 1.1: Evolução do desempenho de memórias semicondutoras primárias e de processadores ao longo dos anos (tomando como base o desempenho de ambos em 1980).

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*

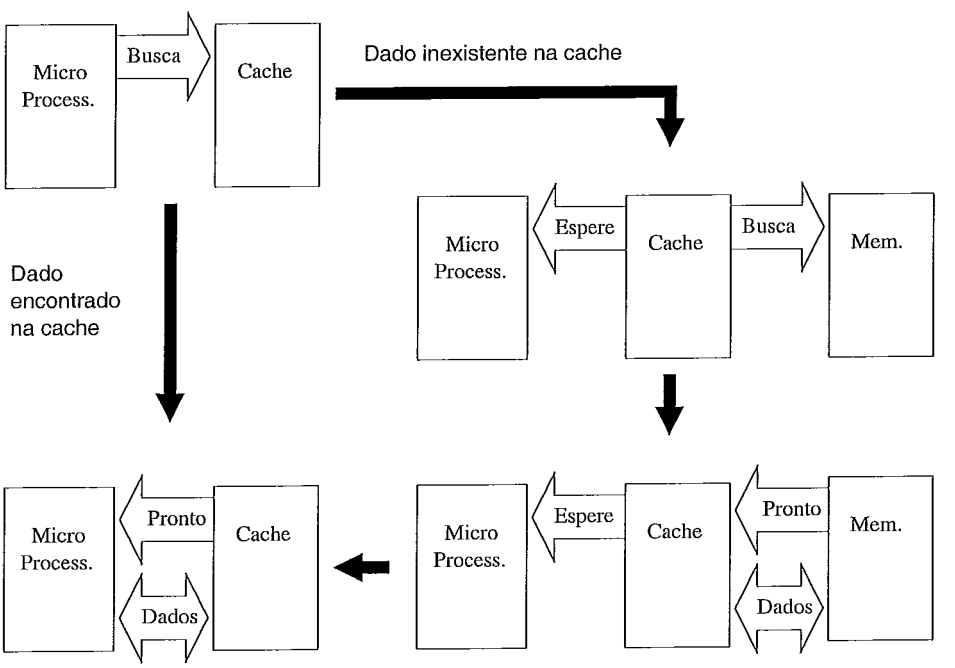

Figura 1.2: Esquema simplificado de uma *Cache*: apenas quando um operando não é encontrado na *Cache*, ele é buscado na memória.

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

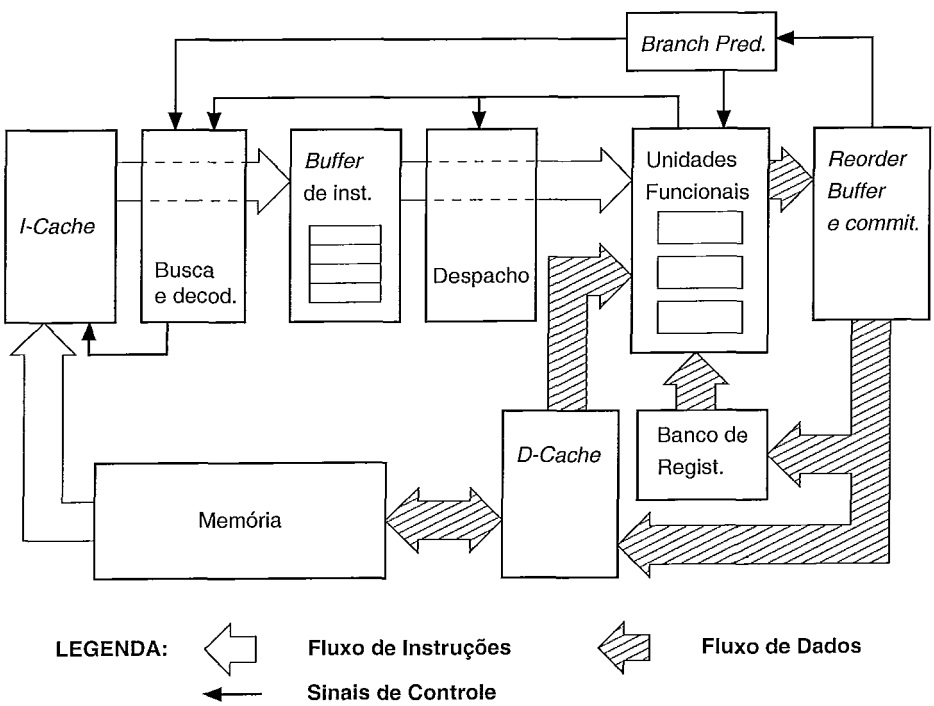

Figura 1.3: Diagrama de um processador Super Escalar.

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.

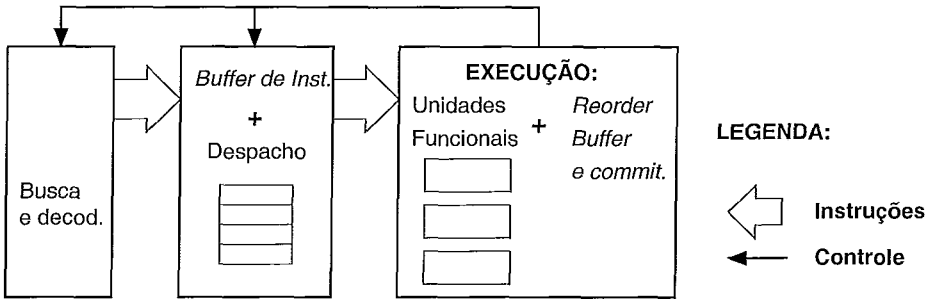

Figura 1.4: Separação do mecanismo de Execução do de Busca e Decodificação.

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.

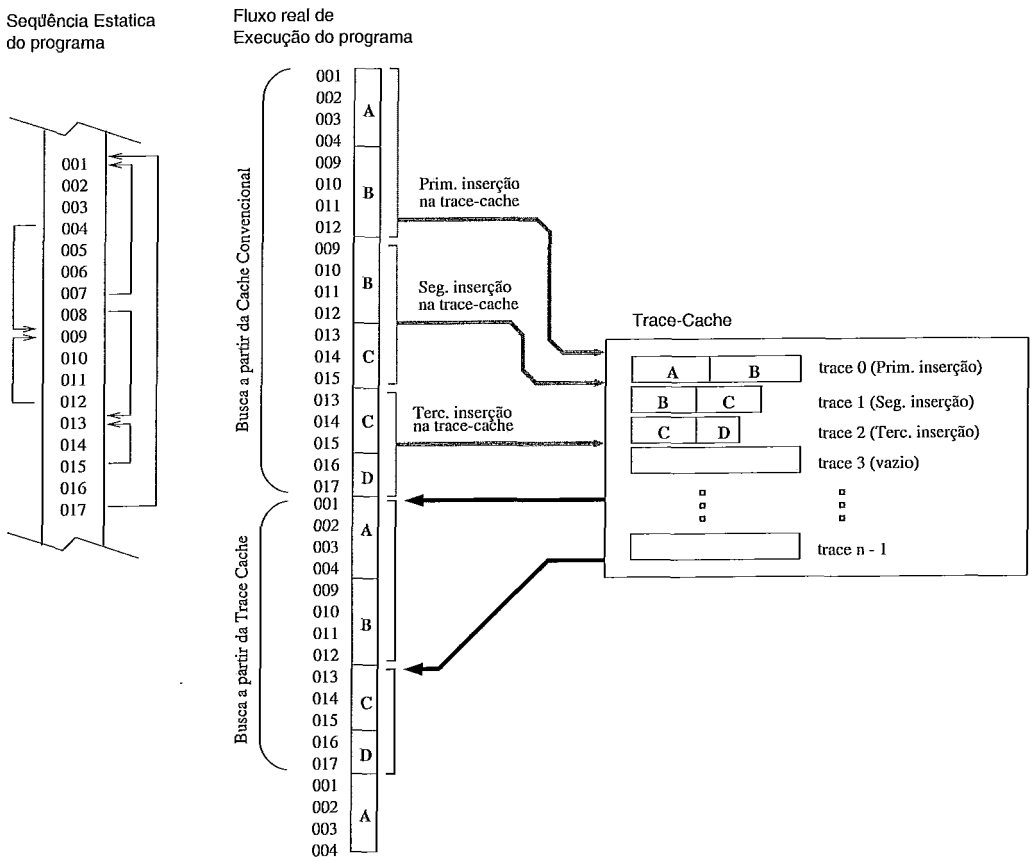

Figura 1.5: Esquema Simplificado de Operação de uma *Trace Cache*.

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).

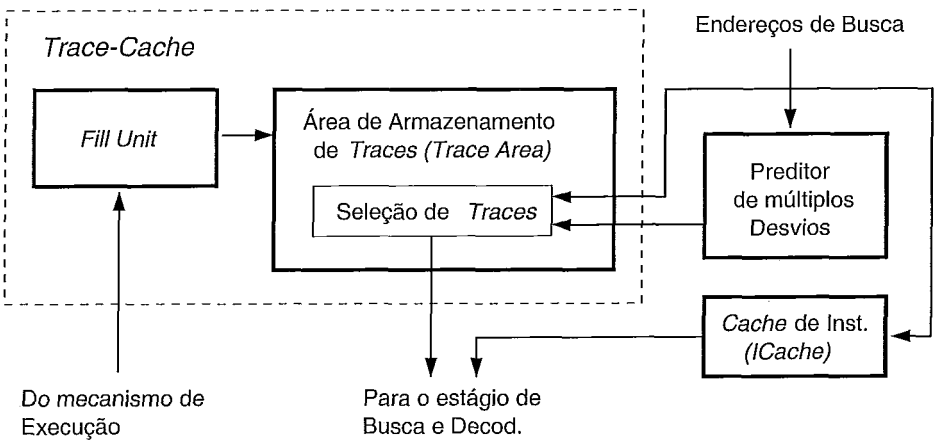

Figura 1.6: Elementos básicos de uma *Trace Cache* e sua inserção no mecanismo de busca e decod. do processador.

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 outra[1].

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 freqüência 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

[1]Neste 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 implementação[2], 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]).

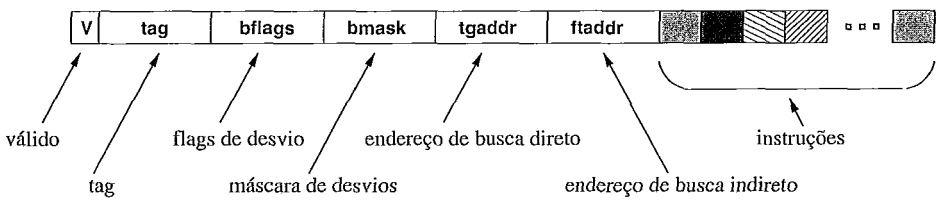

Figura 1.7: Composição de um *trace*.

A descrição dos campos da Figura 1.7 é:

valido 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*.

[2]A 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

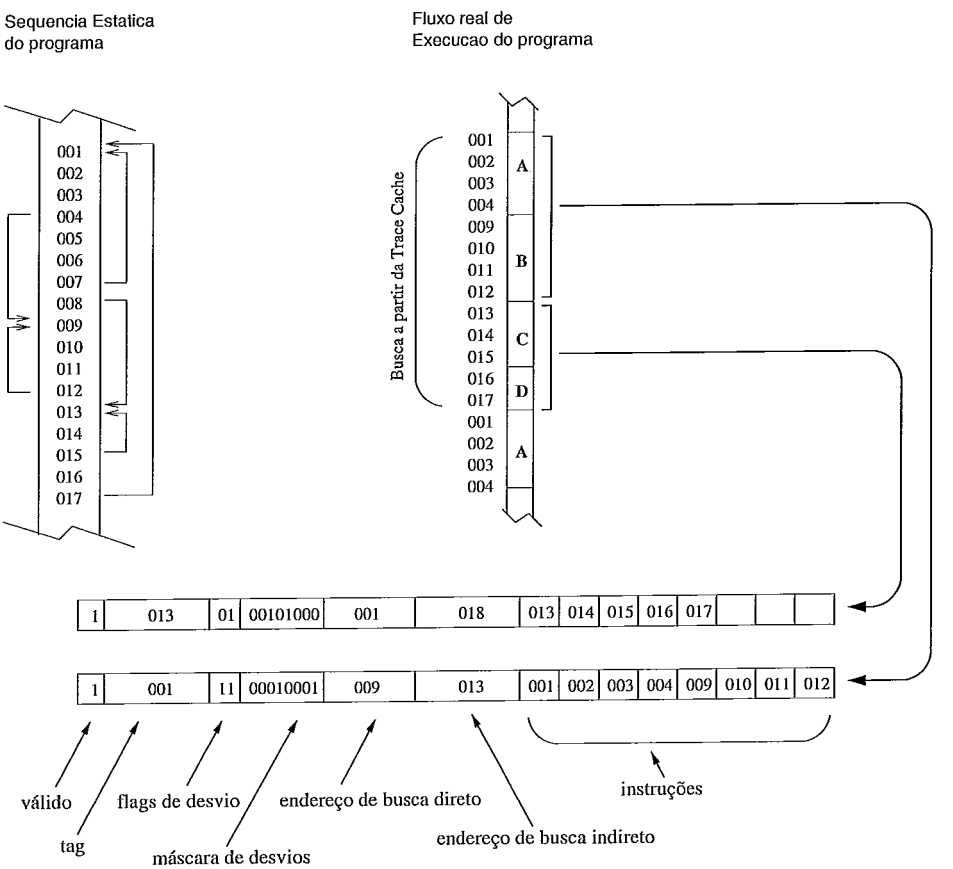

Figura 1.8: Exemplo de composição de *traces* tomando, como exemplo, os *traces* da Figura 1.5.

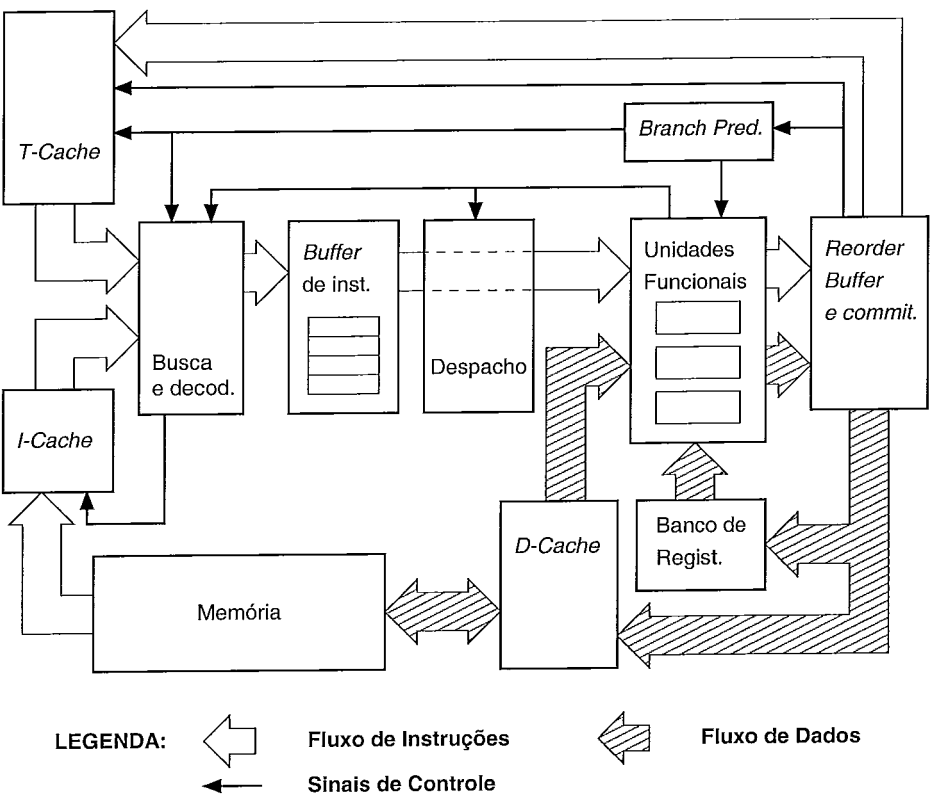

Figura 1.9: *Pipeline* de um processador Super Escalar com *Trace Cache*.

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 adeqüadamente 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 partilhado 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 processo[3] 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.

[3]A 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

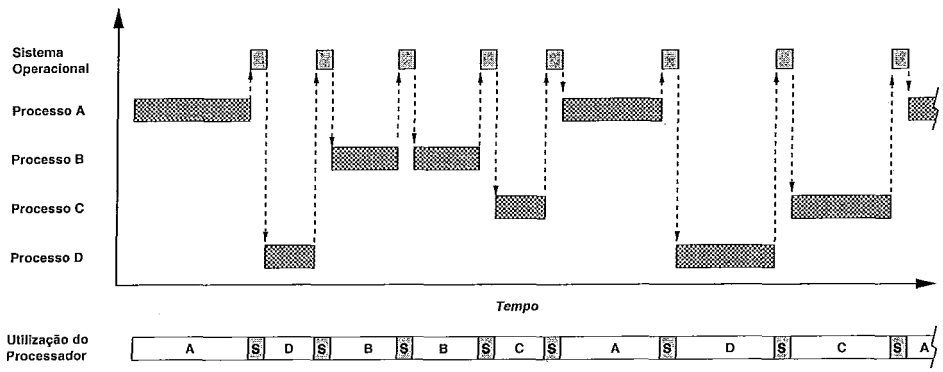

Figura 1.10: Exemplo simplificado de Revezamento de processos através de Multiprogramação.

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.

[1] Quando 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 atendesse 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 peso[4] que indica por quanto tem-

[2] "Normais" são os processos não classificados como processos de tempo real.

[3] Um jiffie corresponde, normalmente, a 10ms.

[4] No 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 lenta[5], 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.

[5]Interrupções 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 pares[6], 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.

[6]Nem 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

- Código das bibliotecas utilizadas pelo Compilador para a arquitetura PISA.

A arquitetura PISA é uma arquitetura derivada do conjunto de instruções MIPS-IV, criada justamente para simulação. A semântica do conjunto de instruções PISA é um superconjunto do conjunto da MIPS IV com os seguintes acréscimos e/ou diferenças dignas de nota:

- Não há *delay-slots*.
- *Loads* e *Stores* suportam dois modos de endereçamento além daqueles disponíveis no conjunto de instruções MIPS-IV: indexado (registrador + registrador) e auto-incremento/decremento.
- Há uma instrução para extração de raiz quadrada (em ponto flutuante de precisão simples ou dupla).
- O tamanho das instruções foi estendido para 64 *bits* (dos quais, 32 *bits* para anotações).

3.1.2 Recursos oferecidos pelo Simulador

Apesar do conjunto de instruções derivado do MIPS-IV, o **SimpleScalar** não modela uma arquitetura real. Os programas que ele simula são gerados pelo seu próprio compilador GCC e só é oferecido o suporte para as *syscalls* do sistema operacional **Ultrix** [1].

Uma vantagem oferecida pelo **SimpleScalar** e, em certa medida, decorrente de não haver o compromisso de modelar uma arquitetura real, é sua grande flexibilidade de configuração. É possível modelar processadores com (ou sem) *Caches*, *TLB*, preditores de variados tipos, com variáveis quantidades e modalidades de unidades funcionais, etc. Além do mais, de acordo

[1] O sistema operacional hipotético suportado pelo **SimpleScalar** é denominado, pelos seus criadores, **SSTrix**, uma fusão dos nomes **SS**, de **SimpleScalar** e **Trix**, de **Ultrix**

com o tipo de simulação pretendida, pode-se utilizar um dos vários simuladores oferecidos pelo *Tool Set* **SimpleScalar**; há desde simuladores rápidos e pouco detalhados em seus resultados até aqueles mais detalhados (que, por isso, processam menos instruções por unidade de tempo) passando por simuladores de *Caches*.

Os simuladores oferecidos pelo **SimpleScalar** são:

sim-fast um simples simulador funcional, o mais rápido e menos sofisticado do **SimpleScalar**; não simula memórias *Cache*, não verifica o alinhamento de instruções nem tem suporte para o depurador **Dlite** (que faz parte do **SimpleScalar**).

sim-safe similar a **sim-fast**, mas checa o alinhamento de instruções e permissões de acesso a determinados segmentos de memória.

sim-cache um simulador funcional de memórias *Cache*.

sim-cheetah tal como **sim-cache**, um simulador funcional de memórias *Cache*; porém, capaz de gerar resultados para múltiplas configurações de *Cache* numa única simulação.

sim-profile um simulador funcional que efetua a classificação e contagem (por classe) das instruções simuladas permitindo criar um perfil (*profile*) da execução.

sim-outorder o mais detalhado dos simuladores do **SimpleScalar**. Simula o despacho e execução de múltiplas instruções por ciclo, mesmo fora de ordem. Também permite a simulação de *Cache* (tal como *sim-cache*) e a geração do perfil da execução (tal como **sim-profile**).

sim-eio simulador funcional, cuja principal função é capturar toda a interação entre programa simulado + simulador e sistema operacional residente; estas interações são registradas de maneira a permitir a re-execução do programa (por qualquer um dos demais simuladores) re-

produzindo os eventos externos originais (ou seja, os resultados das chamadas de sistema).

Neste trabalho, adotou-se o simulador **sim-outorder** em virtude de o maior detalhamento no funcionamento do simulador oferecido pelo mesmo. Pelo fato da *Trace Cache* ser um recurso intimamente ligado aos estágios de busca e de conclusão (*commit*) de instruções de um processador Super Escalar, esta foi uma escolha natural. Afinal, dos simuladores oferecidos pelo **SimpleScalar**, **sim-outorder** é o único a modelar, com detalhes, a busca e o despacho de instruções. Também foram levadas em conta a generalidade deste simulador em sua capacidade de simulação e a capacidade de interferência no funcionamento da *I-Cache* simulada.

3.1.3 Funcionamento Interno

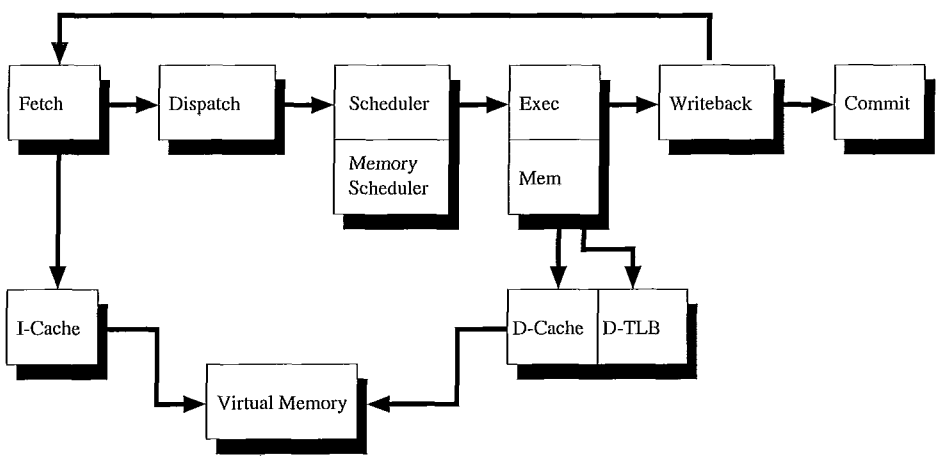

Figura 3.1: *Pipeline* da arquitetura PISA implementada pelo **SimpleScalar**.

O funcionamento interno do **SimpleScalar** compreende duas visões:

- Organização da Arquitetura simulada.
- Operação do simulador.

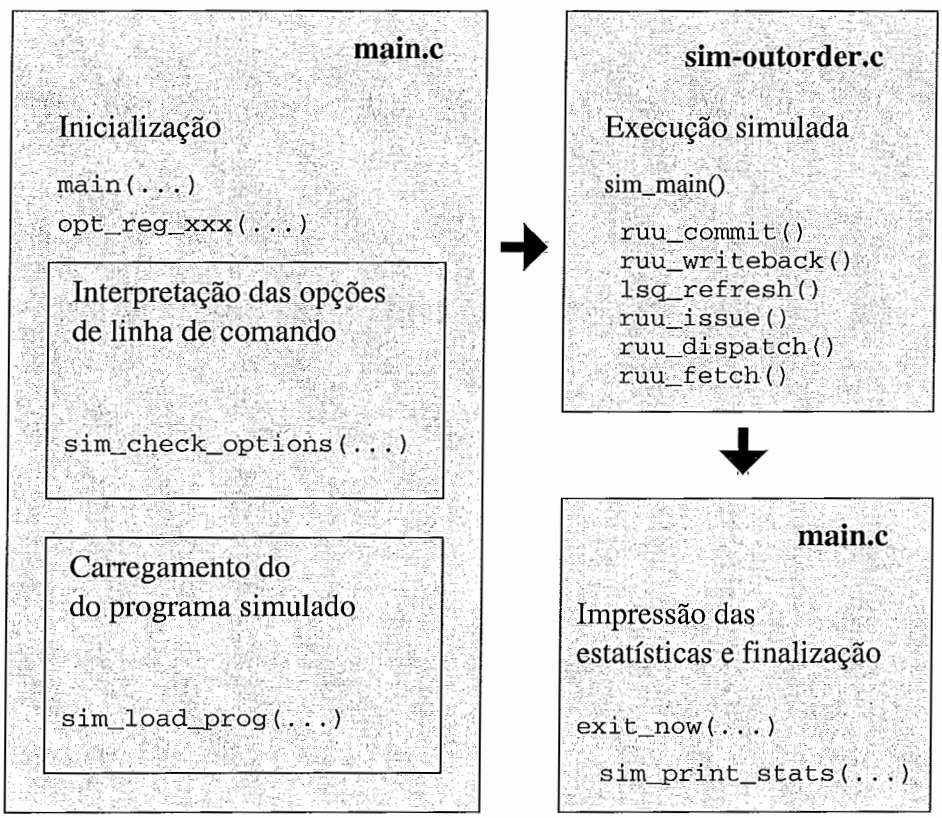

Figura 3.2: Organização simplificada do simulador sim-outorder.

Organização da Arquitetura simulada

A Figura 3.1 mostra uma visão geral do *pipeline* implementado no simulador **sim-outorder** do **SimpleScalar**; as setas da ilustração não representam, exatamente, o fluxo de dados no *pipeline*, mas sim o fluxo de informações de controle[2]. O funcionamento do *pipeline* é típico do processador Super Escalar (Seção 1.2.1, página 4); a função de cada unidade mostrada na Figura 3.1 é denotada pelo seu nome; cada uma delas gastará, pelo menos, um ciclo para desempenhar sua função. Uma breve explicação das funções de cada unidade é dada a seguir:

Fetch: Esta unidade é responsável pela busca de instruções para o *pipeline*.

Dispatch: É responsável pelo decodificação de instruções e renomeação de registradores.

Scheduler+Memory Scheduler: São as unidades que escalonam as instruções para execução conforme a disponibilidade de recursos.

Exec+Mem: São as unidades responsáveis pela execução das instruções já decodificadas e selecionadas para execução.

Writeback: É responsável pela eliminação de dependências nas instruções que estão esperando por resultados de instruções anteriormente despachadas.

Commit: É a unidade que atualiza o estado preciso[3] do processador, retirando instruções do *reorder buffer*.

I-Cache: É a *Cache* de instruções.

D-Cache: É a *Cache* de dados.

[2]Nota-se, por exemplo, que há uma seta direcionada da *I-Cache* para a memória. Embora a memória forneça instruções para a *I-Cache*, é a *I-Cache* quem controla a operação da memória.

D-TLB: É a TLB (*Translation Lookaside Buffer*) para acesso às posições de memória de dados.

Virtual Memory: É a memória Virtual. Na verdade, não deve ser considerada como parte do processador; encontra-se na ilustração para facilitar o entendimento da arquitetura do **SimpleScalar**.

Uma diferença importante, não aparente na ilustração, é o fato da implementação proposta utilizar uma estrutura chamada RUU (*Register Update Unit*) [25]. Numa RUU, são fundidas as ações de busca, despacho, escalonamento e retirada de instruções. Apesar de tudo, conceitualmente, continua valendo a seqüência de operações mostrada na Figura 3.1.

Operação do simulador

A operação do simulador **sim-outorder** do **SimpleScalar**, em linhas gerais, é mostrada na Figura 3.2. Para os outros simuladores, deve-se substituir o despacho, escalonamento, execução, atualização dos registradores (*writeback*) e retirada de instruções por um único estágio de execução já que, para eles, a simulação é meramente funcional.

A operação do simulador começa com sua inicialização. Antes da simulação propriamente dita, o programa carrega, num banco de dados próprio, o elenco de opções de linha de comando que reconhece para, depois verificar quais foram passadas pelo usuário (através da função *sim_check_options*). O passo seguinte é o carregamento do programa a ser simulado, após o qual o simulador gera uma imagem do programa simulado.

Desde que o carregamento do programa tenha sido efetuado corretamente, o próximo passo é a simulação propriamente dita, através da chamada à função *sim_main*(). Cada estágio do *pipeline* mostrado na Figura 3.1 é

[3]Por estado preciso, entende-se o estado dos registradores do processador após a execução de instruções já validadas, isto é, que não podem ser canceladas em virtude dos efeitos produzidos pela execução especulativa.

implementado, dentro de *sim_main*(), através de uma das funções mostradas no bloco **sim-outorder.c**, na Figura 3.2. A ordem de chamada das funções é inversa a ordem de aparição dos estágios do *pipeline*. Isto é feito para garantir que cada estágio enxergue o estado do processador virtual como se as operações ocorressem sincronamente. Neste ponto cabem algumas observações:

1. Inicialmente, dentro de *sim_main*(), mas antes da primeira chamada à *ruu_fetch*(), ocorre uma simples execução funcional[4] para o preenchimento do *pipeline*. É possível, para o usuário, especificar como opção de linha de comando, quantas instruções devem ser executadas desta maneira.

2. Ao contrário do sugerido pelo *pipeline* mostrado, a execução das instruções ocorre na função *ruu_dispatch*().

3. As demais funções simulam o funcionamento dos outros estágios do *pipeline* através, principalmente, da atualização das estatísticas de simulação e da contabilização das latências de cada operação.

4. Quando alguma instrução buscada por *ruu_fetch*() é um *load* ou *store*, essa instrução é colocada numa fila (*load-store queue*). Fica a cargo da função *lsq_refresh*() verificar quais dessas instruções já cumpriram sua latência e retirá-las da *load-store queue*.

5. As instruções buscadas por *ruu_fetch*() são colocadas numa estrutura de dados denominada, no código, **RUU_station**. Esta estrutura é uma fila circular na qual cada elemento da fila contém, dentre outras informações, o código da instrução, seu resultado, seu endereço, o endereço da próxima instrução e outros dados para controle da simulação.

[4]Executam-se as instruções apenas para atualizar o estado do processador sem levar em conta a latência de cada instrução; supõe-se que a execução de cada instrução seja atômica

Ao término da simulação (ou antecipadamente, no caso de alguma anormalidade) as estatísticas contabilizadas durante a simulação são exibidas (ou redirecionadas para algum arquivo, conforme indicado pelo usuário) e o programa encerra sua atividade.

3.2 SPEC CINT95

SPEC95 é o nome pelo qual é conhecido um conjunto de aplicativos para avaliação(*benchmark*) de UCPs, criado e mantido pela organização SPEC (Standard Performance Evaluation Corporation). O conjunto de aplicativos SPEC95 é composto de dois subconjuntos:

SPEC CINT95 um conjunto de 8 aplicativos exigentes em termos de desempenho com instruções com operandos inteiros ou de ponto fixo (não ponto flutuante).

SPEC CFP95 um conjunto de 10 aplicativos projetados para exigir desempenho com instruções de ponto flutuante.

Devido à maior ocorrência de desvios na execução dos aplicativos SPEC CINT95 (em comparação com os SPEC CFP95), estes aplicativos têm estado entre os preferidos para avaliação do desempenho de *Trace Caches* e outros mecanismos.

Outra divisão do conjunto de aplicativos ocorre em função dos argumentos e arquivos de entrada. Há três subconjuntos de entradas e argumentos: **test**, cuja finalidade é óbvia; **train**, cujos resultados podem, sob certas condições, serem utilizados para compilação otimizada dos demais aplicativos e, por fim, **reference**, o conjunto de entradas utilizado para cálculo dos índices de desempenho SPEC. Neste trabalho, foram utilizados os 8 aplicativos do conjunto SPEC CINT95, com entradas do subconjunto **train** ou do conjunto **text**. Os aplicativos são brevemente descritos na Tabela 3.1 a seguir (conforme [9]):

Aplicativo	Tarefa desempenhada pelo Aplicativo
GO	Joga Go contra si mesmo.
GCC	Compila arquivo fonte C pré-processado e gera código assembly SPARC otimizado.
COMPRESS	Compacta grandes arquivos de texto usando codificação adaptativa Limpel-Ziv.
LI	Interpretador Lisp.
IJPEG	Realiza compressão de imagens JPEG com vários parâmetros.
VORTEX	Constrói e manipula três bancos de dados interrelacionados.
M88KSIM	Simula o processador Motorola 88100.
PERL	Realiza manipulações numéricas e de texto (fatoração de números primos e geração de anagramas).

Tabela 3.1: Descrição dos aplicativos SPEC CINT95 utilizados nesta trabalho.

3.3 Modificações no SimpleScalar

Para realizar as avaliações desejadas, foram feitas várias modificações no **SimpleScalar** original. Estas modificações tiveram, principalmente, os seguintes objetivos:

- Simulação de *Trace Cache*.
- Simulação de multiprogramação.

3.3.1 Simulação de *Trace Cache*

Para implementar *Trace Cache* no **SimpleScalar**, preferiu-se adotar uma interface semelhante àquela usada em *Caches* convencionais. Antes de usar o recurso, o simulador precisa criá-lo, alocando espaço para suas estruturas (no caso das *Caches* convencionais, o simulador utiliza a função *cache_create()*, para *Trace Cache* criou-se a função *tcache_create()*). Nesta criação, é necessário especificar, como principais parâmetros, seu número de conjuntos, o número de instruções por *trace*, o número de *traces* por conjunto, o número de predições por *trace* (determinando o número máximo de blocos básicos no *trace*), a latência de acesso e sua política de substituição. Caso a criação seja bem sucedida, a função encarregada desta tarefa retornará um ponteiro para a *Trace Cache* criada.

Diferente das *Caches* convencionais, não existe uma única função para acessar a *Trace Cache* já que a busca de *traces* e sua inserção na *Trace Cache* ocorrem em fases diferentes da execução. De fato, um *miss* numa *Trace Cache*, pela própria natureza da *Trace Cache* não provoca a busca do bloco desejado nos níveis subseqüentes da hierarquia de memória e sua inserção na *Cache*, substituindo outro bloco.

Busca de instruções com *Trace Cache*

A Figura 3.3 ilustra a operação de busca de instruções no **SimpleScalar**, após a implementação da *Trace Cache*. A primeira opção de busca é a *Trace*

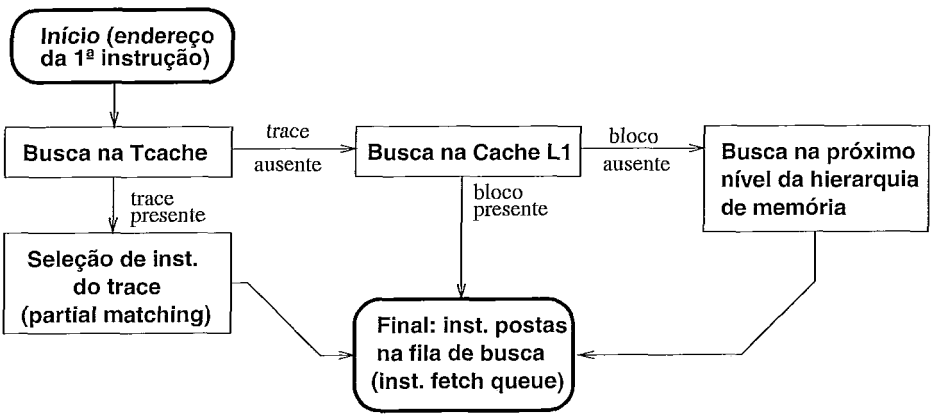

Figura 3.3: Busca de instruções no **SimpleScalar** com *Trace Cache*.

Cache. A *Trace Cache* é indexada pelo endereço da primeira instrução do *trace*. É possível que uma mesma instrução pertença a *traces* diferentes, desde que o bloco básico ao qual ela pertença faça parte de *traces* distintos. E, para diferenciar os *traces*, basta que comecem com instruções distintas.

Caso o *trace* desejado seja encontrado, será selecionado, do *trace*, a maior seqüência de blocos básicos (começando da primeira instrução) que esteja de acordo com as predições efetuadas (*partial matching*). As alternativas seriam utilizar o *trace* apenas se houvesse total concordância com as predições, ou despachar integralmente o *trace*, ignorando as predições. Em [17], foi mostrado que a primeira alternativa é menos eficiente que a coincidência parcial; a segunda alternativa permite melhor desempenho, mas depende de um mecanismo eficiente para recuperação do estado preciso do processador, após uma predição incorreta (tal como em [12]).

Caso o *trace* não seja encontrado, a busca continua na *Cache* L1[5], como ocorreria normalmente. E, se necessário, segue para a *Cache* L2, Memória

[5]Os trabalhos de Rotenberg *et al* [19] e Patel *et al* [17] avaliam *Trace Cache* em conjunto com *Cache* L1; vê-se a *Trace Cache* como substituta ou complemento da *Cache* L1. Segundo Rotenberg, a *Trace Cache* é superior no atendimento ao mecanismo de execução do processador; mas, se um determinado *trace*/bloco não está na *Trace Cache*, é procurado na *Cache* L1.

Principal, etc. Obtido o *trace* (total ou parcialmente), suas instruções são postas na fila de decodificação e despacho (no **SimpleScalar**, chama-se **instruction fetch queue**).

Formação de *traces* e sua inserção na *Trace Cache*

A formação de *traces* é limitada pela quantidade máxima de instruções num *trace* e pela quantidade máxima de blocos básicos. Uma vez atingido um destes limites, a inserção de novas instruções é paralisada. Alguns tipos de instruções provocam o aborto da formação de um *trace* (a formação é cancelada e as instruções já inseridas no *trace* são descartadas). Essenciamente, as chamadas de sistema (*syscalls*) e *traps*, as instruções de retorno de procedimento e as de desvio indireto provocam o aborto da formação do *trace*. Este comportamento se deve à dificuldade de se prever acuradamente o destino dos desvios provocados por estas instruções (previsões incorretas levam a necessidade de desfazer as operações realizadas pela seqüência de instruções, sendo uma penalidade à execução).

3.3.2 Simulação de Multiprogramação

Para simular Multiprogramação no **SimpleScalar**, algumas exigências se impunham:

- Dotar o **SimpleScalar** da capacidade de carregar, e simular, vários programas simultaneamente.
- Possibilitar o compartilhamento da *Trace Cache* e da *Cache* de instruções L1, as quais são comuns a todos os processos simulados.

Duas alternativas foram consideradas:

1. Um único processo simulador carrega todos os programas da sessão de multiprogramação e os simula, controlando seus acessos à *Trace Cache* e *Cache* de instruções L1.

2. A *Trace Cache* e a *Cache* de instruções L1 são implementadas em uma região de memória compartilhada. Cria-se um processo simulador para cada processo simulado. O acesso dos processos simuladores à região de memória compartilhada é controlado por um processo coordenador.

A alternativa 1 tem, como desvantagem, a necessidade de estabelecimento no próprio **SimpleScalar** de um sistema de gerenciamento de memória virtual mais complexo. A complexidade vem da necessidade de carregar vários programas e copiar código e dados de cada um deles em porções diferentes da memória. Outra tarefa complicada seria coordenar o acesso à *Trace Cache* e à *Cache* devido à grande quantidade de porções de memória a alocar e gerenciar.

A alternativa 2 foi adotada por ser mais facilmente implementável (e também depurável). Uma de suas vantagens é, justamente, utilizar o recurso da memória compartilhada (*shared memory*) que, atualmente, é padrão em sistemas tipo Unix, além de ser uma modalidade conhecida e confiável de comunicação entre processos (*System V IPC*, conforme utilizado em Linux [10]).

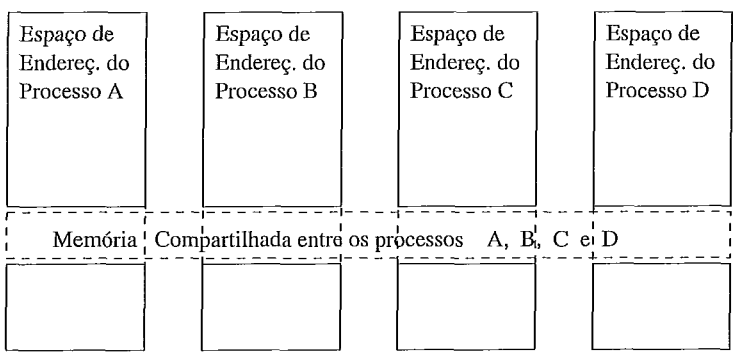

Figura 3.4: Compartilhamento da *Trace Cache* e *Cache* entre processos simuladores/simulados, através de memória compartilhada.

A idéia é simples: A *Trace Cache* e a *Cache* de instruções L1 são implementadas numa região de memória compartilhada por vários processos

(Figura 3.4). Há várias instâncias do **SimpleScalar**; cada instância é um processo simulador encarregado de simular um processo da sessão de multiprogramação. Cada processo simulador tem acesso à *Trace Cache* e *Cache* de instruções L1 implementados em memória compartilhada e se comporta sem se importar com os demais. Um dos processos não simula nenhum programa, sua única tarefa é coordenar o acesso à região de memória compartilhada para que dois processos não possam utilizá-la ao mesmo tempo. Pode-se considerar a permissão de acesso à *Trace Cache* e *Cache* de instruções L1 como a posse de um processador simulado; enquanto não podem acessá-las, os outros processos simuladores são obrigados a suspender a simulação. Tal como ocorre num sistema multiprogramado real, o processador só atende um processo de cada vez; os demais ficam à espera.

Implementação da Simulação de Multiprogramação

Tanto a *Trace Cache* quanto a *Cache* convencional foram implementadas em memória compartilhada (*shared memory*); porém, isso não resolve, automaticamente, o problema do uso compartilhado da *Trace Cache*. É necessário que os processos simulados acessem a *Trace Cache* na ordem definida pelo processo escalonador. Para sincronizar este acesso, utiliza-se outra modalidade de comunicação entre processos, os semáforos.

A Figura 3.5 ilustra como ocorre a sincronização de acesso à memória compartilhada. Usa-se um conjunto de $n + 1$ semáforos, onde n é o número de processos simulados. Cada um dos processos precisa de uma autorização para acessar a *Trace Cache*; esta autorização é dada pelo processo coordenador ao incrementar o semáforo do processo simulador. Enquanto não tiver esta autorização, o processo simulador permanece bloqueado (**Sleeping**) esperando por sua liberação. Com a autorização, o processo simulador simula um determinado número de instruções (que formam a sua *Fatia de tempo* – *time-slice* – simulada) e, após, cede voluntariamente a posse do processador simulado, primeiro incrementando o semáforo do processo coordenador e, depois, decrementando o seu semáforo. O processo coordenador precisa

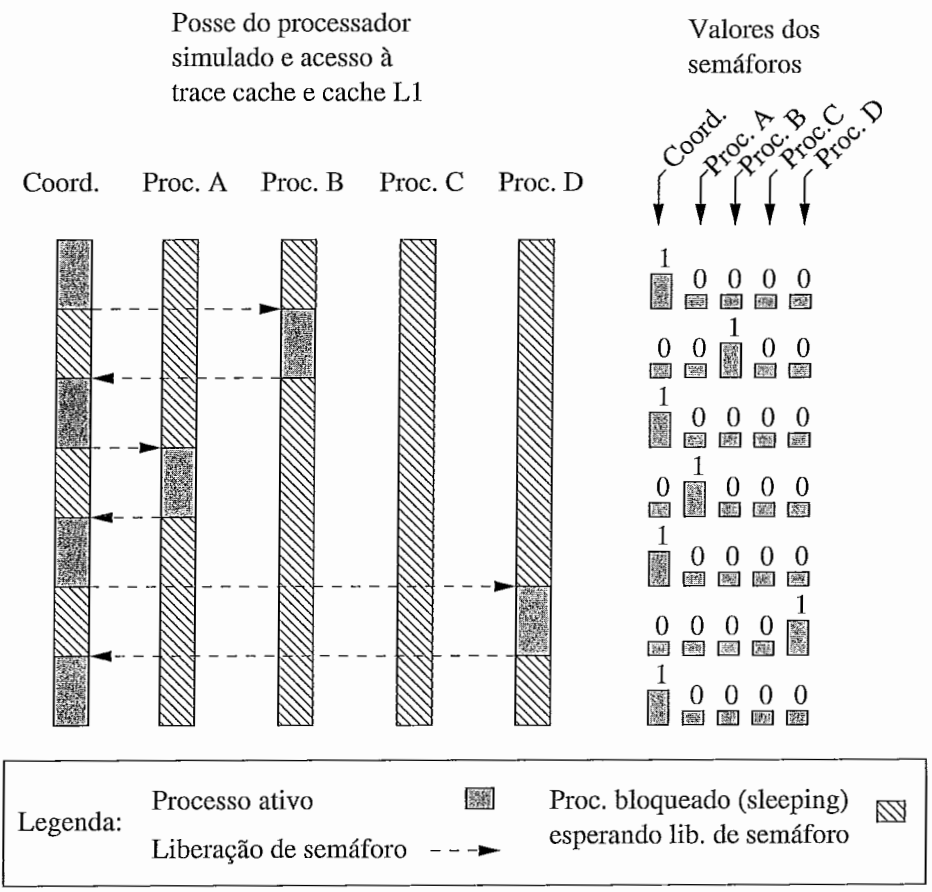

Figura 3.5: Exemplo de sincronização dos processos simuladores no acesso à *Trace Cache* e *Cache* de instruções L1.

da liberação, dada pelo processo simulado, para saber se é hora dele intervir e determinar qual o novo processo que assumirá o processador simulado. Chegou a ser feita a tentativa de sincronização através da troca de sinais [6]; porém, o envio e recebimento de sinais são implementados, em sistemas tipo Unix, como operações não atômicas (tampouco pode ser garantido que os sinais são entregues aos processos destino na mesma ordem em que foram enviados). Daí a escolha de semáforos para sincronização, embora a interface para envio de sinais seja mais simples que a de manipulação de semáforos.

Deve-se notar que um processo simulado não cede, diretamente, o processador simulado a outro. Ele, apenas, devolve o processador simulado ao processo coordenador que, então, decide qual o próximo processo simulado a ser liberado.

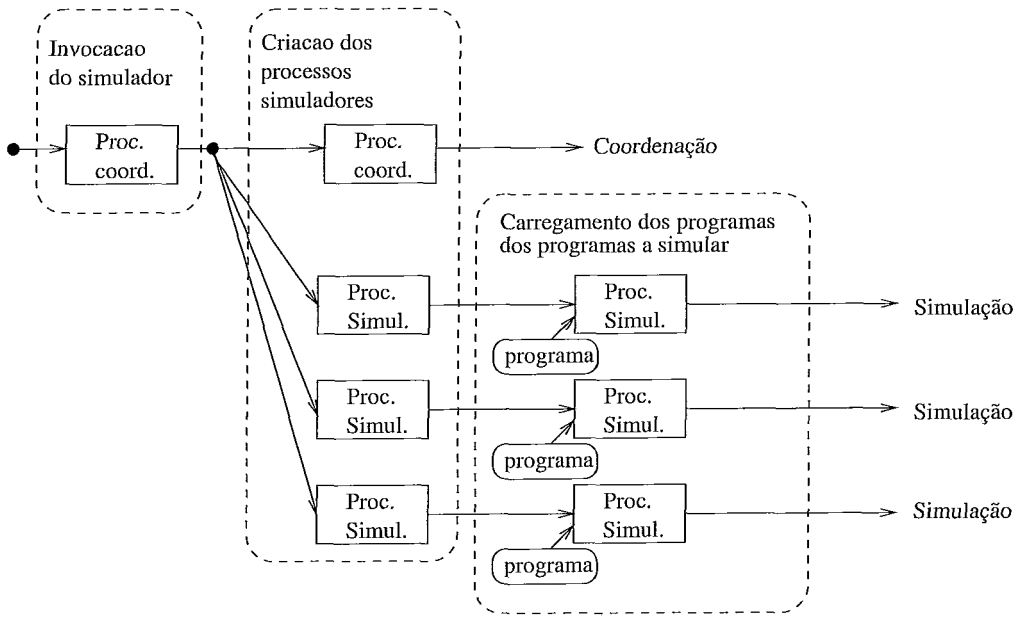

Figura 3.6: Seqüência de operações até o início da simulação (pelos processos simuladores) e coordenação(pelo processo coordenador).

[6]Os sinais referidos são usados para sinalizar assincronamente a ocorrências de certos eventos em sistemas tipo Unix. A lista de sinais disponíveis, num sistema tipo Unix, pode ser exibida com a linha de comando **kill -l**.

Na prática, o processo coordenador é aquele iniciado diretamente pelo usuário. Ele (o coordenador) cria vários cópias de si mesmo através de chamadas a *fork*() (Figura 3.6). Cada um dos processos filho é um processo simulador que efetuará o carregamento de seu próprio programa simulado.

Escalonamento de Processos

O escalonamento de processos simulados é feito pelo processo coordenador, de acordo com o algoritmo de escalonamento utilizado pelo *kernel* Linux 2.0.33 [21, 22, 4, 6] (devidamente adaptado para um número reduzido de processos). Foram simulados os seguintes eventos capazes de interferir no escalonamento de processos:

- Interrupções do relógio.
- Interrupções devido a operações de Entrada/Saída envolvendo arquivos.
- Chamada do escalonador após o atendimento de chamadas de sistema.

As interrupções do relógio ocorrem a cada 10ms e forçam a atualização (o decremento) do peso do processo simulado correntemente, encarregado de registrar por quanto tempo o processo poderá usar o processador (Seção 2.2.2).

As interrupções provocadas por operações de Entrada/Saída acontecem em decorrência de uma instrução de chamada de sistema anterior. Quando um processo simulado realiza uma chamada de sistema envolvendo operações sobre arquivos (*open*, *read*, *write*, etc), ele é bloqueado. Em função do valor atribuído para penalidade destas operações e do valor do *clock* do processador simulado, calcula-se quantos ciclos seriam necessários para o atendimento da requisição de Entrada/Saída. Decorridos estes ciclos de simulação, um outro processo simulado terá sua execução interrompida, o processo que realizou a chamada de sistema volta a estar apto a rodar e o escalonador reavaliará os processos. Se não houver nenhum processo apto a rodar (por estarem bloqueados), a contagem de ciclos avança até que algum dos bloqueados possa ser despertado.

É importante notar que os processos simulados desconhecem o escalonamento real de processos, realizado pela plataforma onde o simulador está sendo executado. Na verdade, do ponto de vista de cada processo simulado, ele está sendo executado isoladamente e tem processador, *Cache* de instruções L1 e *Trace Cache* inteiramente à sua disposição. Porém, cada processo simulado está sendo executado num processador simulado e o processo coordenador faz o papel de sistema operacional simulado, ao decidir quem poderá assumir o processador simulado. Embora o processo coordenador e os processos simuladores estejam sujeitos ao escalonamento real, efetuado pelo sistema operacional da máquina onde estão sendo executados, este escalonamento não é visível aos processos simulados. De fato, os processos simulados não têm acesso direto aos recursos reais do sistema: entre eles e o sistema real existem os simuladores.

A maioria das chamadas de sistema em Linux ao seu final, invocam o escalonador. Do elenco de chamadas de Sistema do **SimpleScalar**, verificou-se quais provocariam a ação do escalonador de Linux e modificou-se o simulador para que reproduzisse este comportamento. Algumas destas chamadas de sistema penalizam o processo, como explicado anteriormente. De acordo com a penalidade e o tipo de intervenção do escalonador, as chamadas de sistema foram classificadas em 4 tipos:

Criação de arquivos quando se atribuiu a mesma penalidade usada para a *syscall* ***open***.

Leitura de arquivos quando se atribuiu a mesma penalidade usada para a *syscall* ***read***.

Sem penalidade quando a *syscall* não impunha nenhuma penalidade ao processo, mas provocava a ação do escalonador, forçando uma reavaliação dos processos para fins de escalonamento.

Sem escalonamento quando nem o processo era penalizado, nem o escalonador era invocado.

As maiores penalidades foram atribuídas à categoria de **Criação de arquivos** que, incluiu chamadas de sistema de abertura e fechamento de arquivos. Já a categoria **Leitura de arquivos** incluiu a maior parte das chamadas de sistema. A distribuição das chamadas de sistema entre categorias ocorreu segudo a observação direta do código fonte do *kernel* Linux 2.0.33 e da prática diária com este sistema. Comumente, as operações de abertura e fechamento de arquivo são mais demoradas; acessos subsequentes a um arquivo já aberto, costumam ser mais rápidas. Uma mesma penalidade foi atribuída às *syscalls* que manuseiam o sistema de arquivos. Algumas chamadas de sistema, porém, acessam tão somente estruturas mantidas em memória pelo Sistema e, devido a sua rápida execução, não mereceram a atribuição de penalidade[7] (a lista das chamadas de sistema por categoria está disponível no Apêndice A).

Neste trabalho, atribuiu-se o valor de 500 μs para a penalidade das chamadas de sistema da categoria **Criação de arquivos** e 200 μs para a penalidade das chamadas de sistema da categoria **Leitura de arquivos**. Tal como a classificação adotada opara as chamadas de sistema, estes valores são discutíveis; porém, perfeitamente plausíveis.

3.3.3 Outras modificações

Relacionadas à simulação de *Trace Cache* e de multiprogramação, bem como a necessidade de facilitar a simulação e obtenção de seus resultados, foram feitas modificações adicionais. Elas acrescentaram as seguintes características ao **SimpleScalar**:

Exibição parcial de estatísticas de simulação. Periodicamente, conforme o número de acessos à *Trace Cache*, o número de instruções executadas ou

[7]Em sistemas Linux, o comando **strace** permite listar as chamadas de sistema efetuadas por programas e o momento em que cada uma foi efetuada. Um programa adequadamente preparado, executado sob controle de **strace**, pode revelar quanto tempo o sistema gasta para atender cada *syscall*.

o número de ciclos simulados, um conjunto selecionado de estatísticas parciais da simulação é exibido.

Relatório de multiprogramação. A cada troca de contexto de processador simulado, estatísticas chave para o presente trabalho são armazenadas num arquivo de relatório de multiprogramação (*multiprogramming report*). Atualmente, as estatísticas do relatório incluem identificação do processo que perdeu o processador, quantidade de *time-slices*, acessos à *Trace Cache*, etc.

Exibição seletiva das estatísticas parciais de simulação. É possível exibir apenas as estatísticas da *Trace Cache* e da *Cache* L1, etc. (esta característica economiza espaço em disco e reduz o tempo da simulação).

Redirecionamento da entrada padrão à partir de arquivos, para cada um dos processos simulados. Originalmente, o redirecionamento de um arquivo para a entrada padrão do simulador (e, portanto, do programa simulado) era feita através do metacaracter <; isto, porém não funcionaria com multiprogramação já que os processos encarregados da simulação não receberiam os dados da entrada padrão após o *fork* que os criasse.

Busca integral de *traces*, só permite a busca de instruções e sua colocação na fila de instruções buscadas se, naquele ciclo, a fila começou vazia.

Inserção otimizada de *traces*, permite, ao invés de aborto prematuro de um *trace*, que este seja inserido na *Trace Cache* até a última instrução válida coletada. As inserção destes tipos de *traces* na *Trace Cache* está condicionada à existência de, pelo menos, uma instrução de desvio no *trace* e ao número de instruções já inseridas (*traces* "longos" serão inseridos).

No Apêndice C, é exibida a tela de apresentação do **SimpleScalar** após as modificações efetuadas neste trabalho. Esta tela de apresentação per-

mite uma melhor apreciação das características presentes no **SimpleScalar** modificado.

Capítulo 4

Avaliações e Resultados

Neste capítulo, são descritas e discutidas as configurações utilizadas nas avaliações do comportamento de *Trace Caches* sob Multiprogramação, bem como os resultados destas avaliações.

4.1 Configuração do Processador Simulado

Na Tabela 4.1, pode ser vista a configuração adotada para o processador simulado pelo **SimpleScalar**. O processador simulado possui *pipeline* de 5 estágios: Busca, Decodificação + Escalonamento, Execução, *Writeback* e Conclusão (*Commit*), conforme a Figura 3.1 (Seção 3.1.3). Cada estágio gasta, pelo menos, um ciclo de máquina para executar sua tarefa.

Conforme mencionado na Seção 2.2.1, o objetivo ao estabelecer em configurar uma máquina com as características especificadas na Tabela 4.1 foi fazer com que o "gargalo" de desempenho do processador fosse o estágio de busca. Adotou-se, para os demais parâmetros do processador simulado, valores que caracterizassem uma postura de projeto agressiva. A maior parte destas características foram empregadas por Patel *et al* [17].

Deve-se recordar que o modelo de *Trace Cache* adotado possui, entre outras, as seguintes características (Seção 2.2.1):

- Instruções são coletadas para formar *traces* após o estágio de Conclusão

Característica	**Valor**
Freqüência de Operação	800 MHz
Largura de Busca	16 insts./ciclo
Tamanho da Fila de Despacho	2048 insts.
Largura de Despacho	16 insts./ciclo (fora de ordem)
Preditor de desvios	GAg de 18 bits (128 KB)
Tamanho da Fila de oper. de memória (*Load-Store Queue size*)	64 insts.
Cache L1	Dividida (*split-cache*) 4KB (*I-Cache*) + 64KB (*D-Cache*)
Latência de acesso à *Cache* L1	1 ciclo
Cache IL1 (*I-Cache*)	8 blocos × 128 *bytes* × 4 vias
Cache DL1 (*D-Cache*)	128 blocos × 128 *bytes* × 4 vias
Número de unidades funcionais	16 grupos de 4 unidades: 1 × Inteiros + 1 × Ponto Flut. + 1 × Memória + 1 × de Desvios
Tamanho da Pilha de End. de Retorno (*Return Address Stack size*)	16384 endereços
Cache L2	Unificada (1 MB): 2048 blocos × 128 *bytes* × 4 vias
Latência de acesso à *Cache* L2	8 ciclos
Política de substituição de blocos de *Cache* (L1 e L2)	LRU
Latência de acesso à memória	50 ciclos
Trace Cache	1024 *traces* × 16 insts. × 8 vias até 3 predições de desvio por *trace*
Política de substituição de *traces*	LRU
Latência de acesso à *Trace Cache*	1 ciclo

Tabela 4.1: Configuração do Processador Simulado.

(*Commit*).

- As instruções buscadas, provêm, em primeiro lugar da *Trace Cache* e, se não for encontrado o *trace* desejado, da *Cache* L1.
- Utiliza-se coincidência parcial (*partial matching*).
- Apenas um *trace* é inserido na *Trace Cache* por ciclo.
- Instruções capazes de provocar o cancelamento (aborto) prematuro do *trace* sob formação, cancelam-no, a não ser que o número de instruções já inserido seja menor que 8 ou que nenhuma delas seja uma instrução de desvio.
- O início da busca de instruções, a partir da *Trace Cache* ou *Cache* convencional, num detrminado ciclo, só ocorre se a fila de instruções buscadas estiver vazia ao início do ciclo (Seção 2.2.1).

4.2 Avaliações

Foram realizadas 4 baterias de avaliações variando-se o nível de multiprogramação desde 1 (Monoprogramação, com cada processo tendo o processador simulado para seu uso exclusivo) até 6, passando pelos níveis de multiprogramação 2 e 4. Para os níveis de multiprogramação 2, 4 e 6, a escolha dos conjuntos de programas a serem simulados foi limitada, já que seria custoso, dentro do escopo deste trabalho, simular todas combinações de programas (seriam 28 combinações distintas dos 8 programas utilizados, tomados 2 a 2, 70 combinações, se tomados 4 a 4, e 28, tomados 6 a 6).

Os aplicativos foram agrupados, para formar as combinações, com base no número de ciclos de execução simulados por cada programa, no tempo de simulação e nos resultados obtidos em sessões prévias. Com nível de multiprogramação igual a 2, os programas foram agrupados de acordo com o número de ciclos de execução simulados sob monoprogramação (exceto M88KSIM e PERL, que foram simulados posteriormente). Ordenando-se os

aplicativos de acordo com o número de ciclos, definimos pares de aplicativos para estas sessões; supunha-se que, deste modo, o nível de multiprogramação permaneria igual a 2 durante o maior parte do período de simulação. Para os níveis de multiprogramação 4 e 6, as combinações de aplicativos foram formadas com base nos resultados obtidos com nível de multiprogramação 2 e no tempo previsto para simulação (foi dada prioridade às sessões com menor tempo de simulação).

As combinações utilizadas para formar as sessões de multiprogramação simuladas estão listadas na Tabela 4.2. A tabela também lista as sessões em que os aplicativos tiveram sua execução simulada isoladamente, sob mono-programação.

Nível de Multiprog.	**Sessões**
1	GO
	LI
	IJPEG
	GCC
	COMPRESS
	VORTEX
	M88KSIM
	PERL
2	GO & LI
	GCC & VORTEX
	LI & COMPRESS
	GO & IJPEG
	M88KSIM & PERL
4	GO, LI, IJPEG & GCC
	LI, IJPEG, VORTEX & COMPRESS
	LI, IJPEG, GCC & COMPRESS
	GO, PERL, M88KSIM & VORTEX
6	GO, LI, IJPEG, GCC, VORTEX & COMPRESS
	GO, LI, IJPEG, PERL, M88KSIM & COMPRESS

Tabela 4.2: Sessões de Multiprogramação Simuladas.

Aplicativo	Núm. de insts.	Arq. de Entrada
GO	548.177.449	2stone9.in (train)
LI	183.304.160	train.lsp (train)
IJPEG	1.464.811.063	vigo.ppm (train)
GCC	1.276.486.373	amptjp.i (train)
COMPRESS	35.818.648	test.in (train)
VORTEX	2.520.154.411	vortex.raw (train)
M88KSIM	490.514.701	ctl.raw (test)
PERL	10.519.230	primes (test)

Tabela 4.3: Número de instruções concluídas e arquivos de entrada para os aplicativos SPEC CINT95.

Os aplicativos utilizados foram retirados do conjunto SPEC CINT95. Todos os aplicativos foram executados integralmente. Os arquivos de entrada (com a identificação do subconjunto ao qual pertencem) bem como o número de instruções concluídas (*commited*) para cada um dos aplicativos são dados na Tabela 4.3.

4.3 Sessões de multiprogramação

As Tabelas 4.4 e 4.5 listam as características relevantes das sessões de multiprogramação. Nestas tabelas, são mostrados a quantidade de trocas de contexto[1] efetuadas em cada sessão, o somatório de ciclos de execução simulados por todos os processos (sem levar em conta as penalidades devido às chamadas de sistema) e o número total de ciclos de execução simulados (este, levando em conta as penalidades das chamadas de sistema).

Deve-se notar, que, mesmo sob monoprogramação foram computadas penalidades devido às chamadas de sistema. Nesta situação, uma penalidade

[1]Na verdade, são contadas as intervenções do escalonador; uma intervenção não provoca, necessariamente, a troca do processo que ocupa o processador.

devido a uma chamada de sistema, simplesmente, faz com que a contagem de ciclos avance o valor correspondente à penalidade imposta.

Nível de Multiprogramação 1	**Trocas de contexto**
GO	95
LI	26
IJPEG	2.520
GCC	280
COMPRESS	163
VORTEX	4.882
M88KSIM	111
PERL	37
Nível de Multiprogramação 2	**Trocas de contexto**
GO & LI	218
GCC & VORTEX	9.341
LI & COMPRESS	245
GO & IJPEG	4.017
M88KSIM & PERL	207
Nível de Multiprogramação 4	**Trocas de contexto**
GO, LI, IJPEG & GCC	5.599
LI, IJPEG, VORTEX & COMPRESS	14.490
LI, IJPEG, GCC & COMPRESS	5.664
GO, PERL, M88KSIM & VORTEX	9.622
Nível de Multiprogramação 6	**Trocas de contexto**
GO, LI, IJPEG, GCC, VORTEX & COMPRESS	15.764
GO, LI, IJPEG, PERL, M88KSIM & COMPRESS	5.939

Tabela 4.4: Quantidade de Trocas de Contexto nas Sessões de Multiprogramação Simuladas.

Os dados exibidos nas Tabela 4.5 mostram que, para alguns aplicativos, a multiprogramação implementada neste trabalho acrescenta poucos ciclos à quantidade de ciclos efetivamente utilizados pelo conjunto de aplicativos em cada sessão de multiprogramação. Para outros, porém, o acréscimo é bastante elevado. Numa execução real, estes ciclos acrescentados pela multiprogramação deveriam ser levados em conta para o cálculo do IPC do processador.

Nível de Multiprogramação 1	Sem penal.	Com penal.	Acrésc.
GO	217.094.068	223.734.001	3,06%
LI	50.172.080	53.292.061	6,22%
IJPEG	239.791.478	438.269.013	82,77%
GCC	509.565.748	528.365.537	3,69%
COMPRESS	8.639.042	22.398.882	159,28%
VORTEX	492.976.250	839.611.973	70,31%
M88KSIM	134.305.644	143.185.553	6,61%
PERL	3.950.012	8.349.977	111,39%
Nível de Multiprogramação 2	**Sem penal.**	**Com penal.**	**Acrésc.**
GO & LI	519.801.430	522.174.001	0,46%
GCC & VORTEX	1.532.851.107	1.550.921.102	1,18%
LI & COMPRESS	112.144.750	123.124.166	9,79%
GO & IJPEG	811.825.237	905.245.249	11,51%
M88KSIM & PERL	178.244.167	184.900.118	3,73%
Nível de Multiprogramação 4	**Sem penal.**	**Com penal.**	**Acrésc.**
GO, LI, IJPEG & GCC	2.180.433.120	2.197.612.408	0,79%
LI, IJPEG, VORTEX & COMPRESS	2.408.354.323	242.1445.367	0,54%
LI, IJPEG, GCC & COMPRESS	1.240.352.195	1.257.718.659	1,40%
GO, PERL, M88KSIM & VORTEX	1.834.314.361	1.837.391.545	0,17%
Nível de Multiprogramação 6	**Sem penal.**	**Com penal.**	**Acrésc.**
GO, LI, IJPEG, GCC, VORTEX & COMPRESS	7.230.430.107	7.234.750.061	0,06%
GO, LI, IJPEG, PERL, M88KSIM & COMPRESS	3.110.040.893	3.121.528.977	0,37%

Tabela 4.5: Quantidade de ciclos de execução simulados nas Sessões de Multiprogramação, com e sem as penalidades das chamadas de sistema.

Neste trabalho, porém, eles não são levados em conta; calculamos o IPC considerando, apenas, os ciclos efetivamente gastos pelo processador na execução do aplicativo. Entretanto, os ciclos adicionais ainda interferem na avaliação já que provocam o bloqueio de um processo e tentativa de escalonamento de outros.

A Tabela 4.4 mostra o número de trocas de contexto efetuado pelos aplicativos sob monoprogramação. Combinando-se os dados da Tabela 4.4 e os número de instruções executadas, da Tabela 4.3, obtem-se o Gráfico da Figura 4.1. Não por acaso, os aplicativos mais penalizados pelo acréscimo de ciclos devido às chamadas de sistema são aqueles que apresentam menor número médio de instruções entre chamadas de sistema. A grande freqüência das chamadas de sistema responde pela grande penalidade relativa, em ciclos. Devido a estes resultados, podemos afirmar que IJPEG, VORTEX, COMPRESS e PERL são aplicativos com o desempenho limitado por operações de Entrada/saída (*I/O Bound*), enquanto GO, LI, GCC e M88KSIM são aplicativos limitados pelo tempo de CPU (*CPU Bound*).

Convém notar que a quantidade total de ciclos que penaliza os aplicativos depende, não só de código do mesmo mas, também, da freqüência assumida para o processador simulado e dos valores atribuídos para as penalidades devido às chamadas de sistema.

4.4 Desempenho da *Trace Cache* sob Multiprogramação

As Tabelas 4.6 e 4.7, mostram os valores de IPC obtidos para cada um dos aplicativos conforme o nível de multiprogramação varia de 1 até 6.

Conforme esperado, o desempenho obtido cai conforme o nível de multiprogramação aumenta. Porém, esta queda não depende, apenas, do nível de multiprogramação mas, também, do conjunto de aplicativos considerado. Além do mais, a queda de desempenho não é monotônica em relação ao nível de multiprogramação (COMPRESS, por exemplo, obtém um desempenho

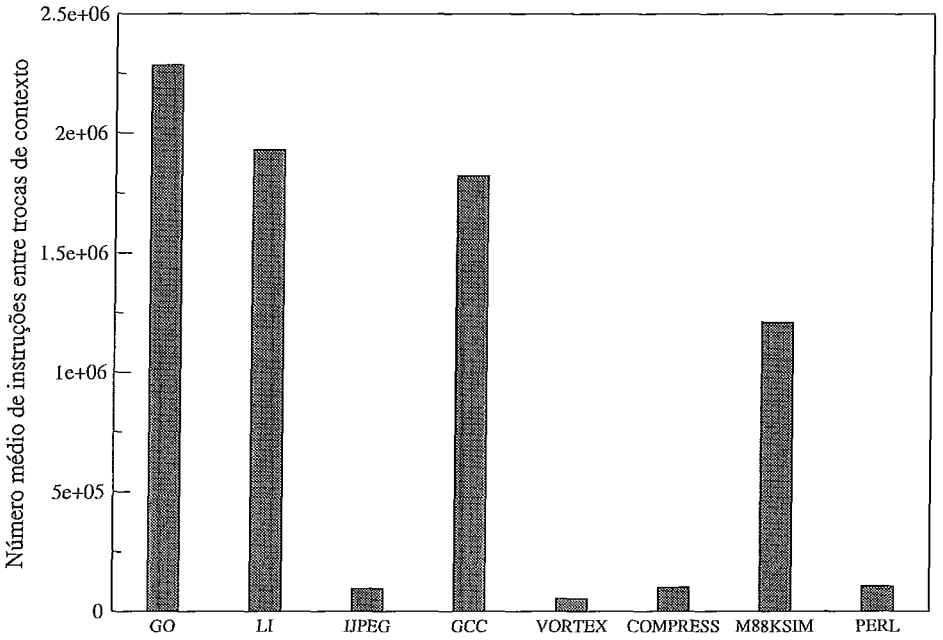

Figura 4.1: Número médio de Instruções entre chamadas de Sistema para os aplicativos.

melhor sob nível de multiprogramação 4, com LI, IJPEG e VORTEX, do que, apenas com LI). Portanto, estes resultados comprovam ser um equívoco desconsiderar o efeito da multiprogramação sobre a eficiência de *Trace Caches*. Outro equívoco desmascarado pelos resultados é considerar, apenas, o nível de multiprogramação para estimar a eficácia de *Trace Caches* já que a queda de desempenho depende do aplicativo considerado.

Analisando-se, conjuntamente, as Tabelas 4.3, 4.6, 4.7 e a Figura 4.1, é possível chegar a algumas conclusões acerca do desempenho dos aplicativos, sob multiprogramação.

Quando postos a rodar dois a dois, a maior queda no desempenho é observada no aplicativos *I/O Bound* ou com menor número de instruções em relação ao outro participante da sessão de multiprogramação. Assim, com GO & LI, GO experimenta uma queda pequena no seu IPC, comparando-se com a monoprogramação, ao contrário de LI. No caso de GCC e VORTEX, VORTEX experimenta a maior queda em seu desempenho pelo fato de ser

I/O Bound, apesar de GCC executar um menor número de instruções. Com LI e COMPRESS, COMPRESS é *I/O Bound* e, também, apresenta um menor número de instruções executadas do que LI. GO e IJPEG apresentam quedas de desempenho próximas (38% para GO, 48% para IJPEG); a leve vantagem de GO deve ser creditada ao fato dele ser *CPU Bound*. Esta regra se aplica, também à sessão formada por M88KSIM e PERL.

Com 4 e 6 processos numa sessão a interação é mais complexa; porém, ainda valem algumas observações: Na sessão 4A, o ordenamento dos aplicativos, segundo o número de instruções executadas se repete ao considerarmos as quedas relativas de desempenho de cada um dos aplicativos, em relação à monoprogramação. Na sessão 4C, esta característica se repete. Nas sessões 4B, 4D, 6A e 6B, esta tendência falha por 1 ou 2 aplicativos fora desse ordenamento.

Este comportamento revela, além da dependência do desempenho, da *Trace Cache*, em relação aos aplicativos, uma dependência em relação ao número de instruções executadas numa sessão. Considerando-se que, à medida, que os aplicativos vão terminando sua execução, o nível de multiprogramação da sessão cai, este comportamento é compreensível. Aplicativos com menor número de instruções a executar, potencialmente, terminam sua execução mais cedo; assim, eles participam dos trechos iniciais da sessão, quando têm que partilhar o processador com mais processos e deixam o processador livre para os aplicativos com maior número de instruções a executar. Com menos processos para compartilhar a *Trace Cache*, os aplicativos conseguiriam melhor desempenho nos trechos finais da sessão. Não se deve esquecer, contudo, da interferência do sistema operacional. Os processos não partilham igualmente os processadores; o número e o tipo de chamadas de sistema realizado por cada processo introduz perturbações no comportamento descrito.

Aplicativos	Sessões					
	1	2A	2B	2C	2D	2E
GO	2,5251	2,1696	-	-	1,5614	-
LI	3,6535	0,6862	-	3,4313	-	-
IJPEG	6.1087	-	-	-	3,1793	-
GCC	2,5051	-	2,4759	-	-	-
COMPRESS	4,1461	-	-	0,6100	-	-
VORTEX	5,1121	-	2,4773	-	-	-
M88KSIM	3,6522	-	-	-	-	3,5765
PERL	2,6631	-	-	-	-	0,2560

Legenda: 2A: GO & LI 2B: GCC & VORTEX
2C: LI & COMPRESS 2D: GO & IJPEG
2E: M88KSIM & PERL

Tabela 4.6: IPC sob níveis de multiprogramação 1 e 2

Aplicativos	Sessões					
	4A	4B	4C	4D	6A	6B
GO	1,0181	-	-	0,6347	0,3565	1,1294
LI	0,8806	2,2375	1,9094	-	0,1375	0,2780
IJPEG	3,5305	1,8434	4,7623	-	1,1073	2,2634
GCC	1,2528	-	1,5811	-	1,1119	-
COMPRESS	-	0,0446	1,2168	-	0,0294	0,0538
VORTEX	-	3,4548	-	3,2382	3,7505	-
M88KSIM	-	-	-	2,6567	-	0,9303
PERL	-	-	-	1,3636	-	0,0843

Legenda: 4A: GO, LI, IJPEG & GCC
4B: LI, IJPEG, VORTEX & COMPRESS
4C: LI, IJPEG, GCC & COMPRESS
4D: GO, PERL, M88KSIM & VORTEX
6A: GO, LI, IJPEG, GCC, VORTEX & COMPRESS
6B: GO, LI, IJPEG, PERL, M88KSIM & COMPRESS

Tabela 4.7: IPC sob níveis de multiprogramação 4 e 6

4.4.1 Taxa de *miss* da *Trace Cache* sob multiprogramação

Além da medida indireta da eficiência da *Trace Cache* através do desempenho na execução de programas, convém analisar outra medida de sua eficiência: sua taxa de *miss*. As Tabelas 4.8 e 4.9 mostram os valores obtidos conforme muda o nível de multiprogramação.

Um fato notável é que, sob multiprogramação, as taxas de *miss* da *Trace Cache* quase não mudaram em relação à monoprogramação, mantendo sua variação dentro dos limites de $\pm$ 4%, exceto por IJPEG e PERL, em algumas das sessões. Este é um comportamento contrário ao do IPC, que variou drasticamente conforme a quantidade de processos na sessão. Para entendermos este comportamento, devemos lembrar que, ao contrário das *Caches* convencionais, um *miss* na *Trace Cache* não provoca a imediata substituição de um *trace* (no caso, das *Caches* convencionais, um bloco). A substituição dos blocos de uma *Cache* contribui para a correção do seu conteúdo e, devido aos princípios de localidade temporal e de referência, os próximos acessos, logo após um *miss*, provavelmente provocarão *hits*.

Numa *Trace Cache*, porém *misses* não contribuem, diretamente, para a atualização da *Trace Cache*; portanto, as taxas de *miss* permanecem altas. Para *Trace Cache*, a taxa de *miss* dependerá da configuração da *Trace Cache* e do aplicativo sendo executado, através das taxas de formação e substituição *traces*. A Tabela 4.10 mostra as quantidades de substituições e de inserções de *traces*, bem como a razão entre substituições e inserções para as sessões de multiprogramação simuladas. Percebe-se, claramente, que o número de substituições é bastante inferior ao número de inserções; esta diferença indica que muitos *traces* são reinseridos (estas inserções não são computadas como substituições). Estes *traces* não sofrem substituição por serem muito reutilizados (a política de substituição de *traces* é *Less Recently Used* o que coloca os *traces* recentemente usados no fim da fila de substituição).

Aplicativos	Sessões					
	1	2A	2B	2C	2D	2E
GO	0,3092	0,3073	-	-	0,3098	-
LI	0,3813	0,3937	-	0,3812	-	-
IJPEG	0,2830	-	-	-	0,3058	-
GCC	0,3861	-	0,3845	-	-	-
COMPRESS	0,4685	-	-	0,4684	-	-
VORTEX	0,2974	-	0,3103	-	-	-
M88KSIM	0,2234	-	-	-	-	0,2254
PERL	0,4597	-	-	-	-	0,4597

Legenda: 2A: GO & LI 2B: GCC & VORTEX
2C: LI & COMPRESS 2D: GO & IJPEG
2E: M88KSIM & PERL

Tabela 4.8: Taxa de *miss* de *Trace Cache* sob níveis de multiprogramação 1 e 2

Aplicativos	Sessões					
	4A	4B	4C	4D	6A	6B
GO	0,3122	-	-	0,3108	0,3086	0,3108
LI	0,3785	0,3749	0,3800	-	0,3814	0,3783
IJPEG	0,3156	0,2724	0,2851	-	0,3238	0,3061
GCC	0,3877	-	0,3854	-	0,3863	-
COMPRESS	-	0,4880	0,4692	-	0,4723	0,4969
VORTEX	-	0,3041	-	0,3100	0,3069	-
M88KSIM	-	-	-	0,2229	-	0,2616
PERL	-	-	-	0,5348	-	0,5012

Legenda: 4A: GO, LI, IJPEG & GCC
4B: LI, IJPEG, VORTEX & COMPRESS
4C: LI, IJPEG, GCC & COMPRESS
4D: GO, PERL, M88KSIM & VORTEX
6A: GO, LI, IJPEG, GCC, VORTEX & COMPRESS
6B: GO, LI, IJPEG, PERL, M88KSIM & COMPRESS

Tabela 4.9: Taxa de *miss* de *Trace Cache* sob níveis de multiprogramação 4 e 6

Sessões	Número de inserções	Número de substit.	Substit./ inserções
GO	26.947.936	2.432.227	0,0903
LI	8.490.803	29	$3{,}41 \cdot 10^{-7}$
IJPEG	58.055.330	22.217	0,0004
GCC	70.337.179	2.384.862	0,0339
COMPRESS	1.579.513	0	0
VORTEX	104.678.678	127.768	0,0012
M88KSIM	29.111.694	50	$1{,}72 \cdot 10^{-7}$
PERL	668.148	73	0,0001
GO & LI	35.539.492	2.437.193	0,0686
GCC & VORTEX	17.6245.024	2.630.877	0,0149
LI & COMPRESS	10.070.296	54	$5{,}36 \cdot 10^{-6}$
GO & IJPEG	8.4923.543	2.470.706	0,0290
M88KSIM & PERL	29.589.488	652	$2{,}20 \cdot 10^{-5}$
GO, LI, IJPEG & GCC	163.888.334	4.931.332	0.0301
LI, IJPEG, VORTEX & COMPRESS	173.425.695	190.534	0.0011
LI, IJPEG, GCC & COMPRESS	138.449.349	2.423.936	0.0175
GO, PERL, M88KSIM & VORTEX	162.568.448	2.696.793	0.0166
GO, LI, IJPEG, GCC, VORTEX & COMPRESS	163.888.334	4931332	0.0188
GO, LI, IJPEG, PERL, M88KSIM & COMPRESS	271.176.416	5.088.473	0.0200

Tabela 4.10: Quantidades de Inserções e substituições de *traces* nas Sessões de Multiprogramação Simuladas.

4.4.2 IPC com *Trace Cache* e multiprogramação

Visto que a taxa de *miss* varia pouco, em função do nível de multiprogramação, surge, naturalmente, a qüestão da origem da queda de desempenho da *Trace Cache*. Deve-se atribuir a responsabilidade pela queda de desempenho à interferência dos demais aplicativos; esta influência pode ser avaliada pelo número de substituições de *traces* de outros processos feitos por cada aplicativo. As Tabelas 4.11 e 4.12 mostram o número de substituições interprocessos, para cada aplicativo, em cada sessão de multiprogramação. De modo geral, quanto mais *traces* de outros processos um aplicativo desaloja, maior a sua influência sobre os demais e menor sua suscetibilidade à influência dos demais processos. A título de exemplo, a Figura 4.2, à esquerda, mostra a contribuição do IPC de cada aplicativo na seção 6B (GO,LI, IJPEG, M88KSIM, PERL e COMPRESS), numa das sessões. À direita da Figura 4.2, usou-se a contribuição relativa de cada aplicativo como fator de correção para o IPC dos aplicativos sob monoprogramação [2]; os valores assim obtidos foram usados para formação do gráfico à direita. Constata-se que há uma semelhança nos gráficos.

A Figura 4.2 pode levar a crer na existência de uma fórmula simples para se descobrir o IPC de cada aplicativo numa sessão multiprogramada, a partir do IPC monoprogramado. Longe de estabelecer uma regra prática, os gráficos mostram que há uma correlação entre o número de substituições de *traces* interprocessos e o desempenho dos aplicativos sob multiprogramação. A fórmula utilizada, inclusive, falha drasticamente em vários casos; por exemplo, para descobrir o IPC de PERL na sessão 4D (GO, PERL, M88KSIM e VORTEX). Nesta sessão, PERL colabora com 3,02% do total de substituições interprocessos e seu IPC sob monoprogramação foi 2,6631; sendo assim, seu IPC na sessão 4D deveria ser de $0,302 \cdot 2,6631 = 0.08$. Comparando-se este valor com o da Tabela 4.7, constata-se um erro da estimativa. Apesar disso o número de substituições interprocessos efetuadas

[2]Multiplicou-se o IPC de cada aplicativo sob monoprogramação, pela contribuição relativa de cada um para o total de substituições entre processos

por cada aplicativo mantém sua utilidade para indicar tendências no IPC de cada aplicativo.

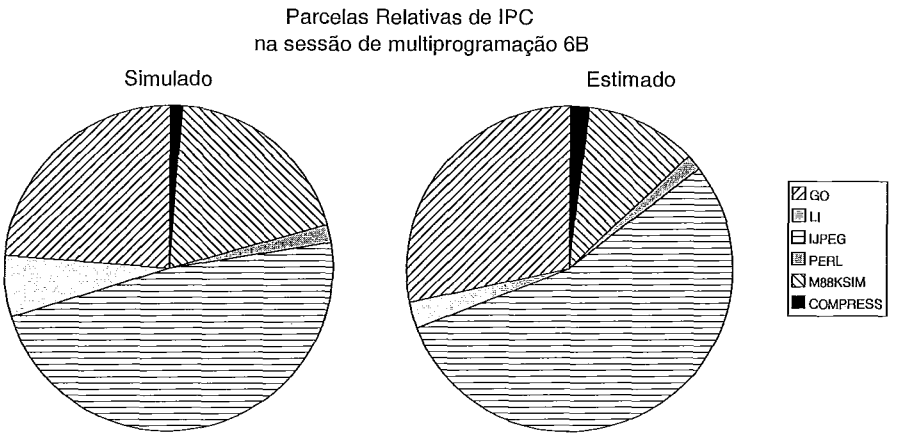

Figura 4.2: Comparação entre IPCs simulados e estimados a partir da monoprogramação e das substituições de *traces* interprocessos.

Apesar do número reduzido, em relação ao total de substituições, as substituições interprocessos têm papel fundamental no desempenho da *Trace Cache*, sob multiprogramação. Deve-se levar em conta que os aplicativos SPEC CINT95 possuem elevada freqüência de desvios; por isso, eles são mais suscetíveis à eficiência das *Trace Caches*. A substituição de um *trace* por outro processo pode ter um efeito devastador no desempenho do aplicativo já que, devido às freqüentes trocas de contexto, o aplicativo pode não ter tempo de estabelecer seu *working set* na *Trace Cache* antes de perder a posse do processador.

4.5 Evolução do desempenho sob Multiprogramação

As tabelas e gráficos já vistos só levaram em conta o desempenho dos aplicativos computados ao final do seu tempo de execução. Neste trabalho, porém,

Aplicativos	Sessões				
	2A	2B	2C	2D	2E
GO	3.235	-	-	15.061	-
LI	2.828	-	4	-	-
IJPEG	-	-	-	19.040	-
GCC	-	80.420	-	-	-
COMPRESS	-	-	2	-	-
VORTEX	-	87.414	-	-	-
M88KSIM	-	-	-	-	362
PERL	-	-	-	-	25

Legenda: 2A: GO & LI 2B: GCC & VORTEX
2C: LI & COMPRESS 2D: GO & IJPEG
2E: M88KSIM & PERL

Tabela 4.11: Quantidade de substituições de *traces* interprocessos sob nível de multiprogramação 2

Aplicativos	Sessões					
	4A	4B	4C	4D	6A	6B
GO	58.532	-	-	89.635	25.666	29.860
LI	2.383	553	1.067	-	3.007	1.916
IJPEG	54.960	13.234	7.296	-	62.109	23.937
GCC	11.974	-	10.032	-	72.713	-
COMPRESS	-	627	256	-	1.133	1.224
VORTEX	-	13.436	-	77.344	24.233	-
M88KSIM	-	-	-	5156	-	8.085
PERL	-	-	-	1.357	-	1.634

Legenda: 4A: GO, LI, IJPEG & GCC
4B: LI, IJPEG, VORTEX & COMPRESS
4C: LI, IJPEG, GCC & COMPRESS
4D: GO, PERL, M88KSIM & VORTEX
6A: GO, LI, IJPEG, GCC, VORTEX & COMPRESS
6B: GO, LI, IJPEG, PERL, M88KSIM & COMPRESS

Tabela 4.12: Quantidade de substituições de *traces* interprocessos sob níveis de multiprogramação 4 e 6

torna-se necessário analisar o comportamento transiente dos aplicativos. Os aplicativos tomados para nossos testes possuem características diferentes, tempos de execução diferentes e, conseqüentemente, instantes de início de execução e término diferentes entre si. Numa sessão de multiprogramação com, por exemplo, 6 processos, o término da execução de um deles faz com que o nível de multiprogramação caia, temporariamente para 5 e, conforme os programas prossigam, o nível de multiprogramação cairá para 4, 3, 2 e 1 até o término do último aplicativo.

Os gráficos das Figuras 4.3 e 4.4 mostram os valores de IPC para os aplicativos entre trechos delimitados pelo início da sessão e o término da execução de cada processo. Os valores de IPC são exibidos em função de número de ciclos globais de execução, incluindo as penalidades devido às chamadas de sistema. O IPC, porém, tal como os valores previamente apresentados, foi calculado com o número de ciclos gastos por cada processo, sem levar em conta os ciclos em que o processo permaneceu bloqueado ou esperando escalonamento. Por motivos de espaço não são mostrados os gráficos de todas as sessões (todos eles podem ser vistos no Apêndice B). Nas Tabelas 4.13 e 4.14 são mostrados os valores de IPC obtidos em cada trecho. Percebe-se que muitos dos processos apresentam, em certos trechos, elevados valores de IPC, tais como LI na Figura 4.4; pelo fato de, no trecho considerado, o nível de multiprogramação ser efetivamente menor. Ainda no exemplo de LI, o máximo valor de IPC obtido é maior que com monoprogramação (na Tabela 4.6). Isto é possível porque, enquanto sob monoprogramação considerou-se toda o período de execução dos programas, aqui considera-se, apenas, um parte deste período onde as condições de execução são diferentes.

Os gráficos das Figuras 4.3 e 4.4 demonstram que, sob multiprogramação, não se deve considerar apenas as estatísticas finais de execução dos programas. Para programas interativos, por exemplo, pode haver considerável diferença entre o desempenho medido e o desempenho percebido pelo usuário, conforme o período de execução considerado. Neste caso, também há a influência preponderante do sistema operacional, responsável pelo escalona-

mento dos processos. Constata-se, a partir dos valores das Tabelas 4.6 e 4.7, que o desempenho dos aplicativos depende não só do nível de multiprogramação mas, também, do conjunto de aplicativos participante da sessão de multiprogramação.

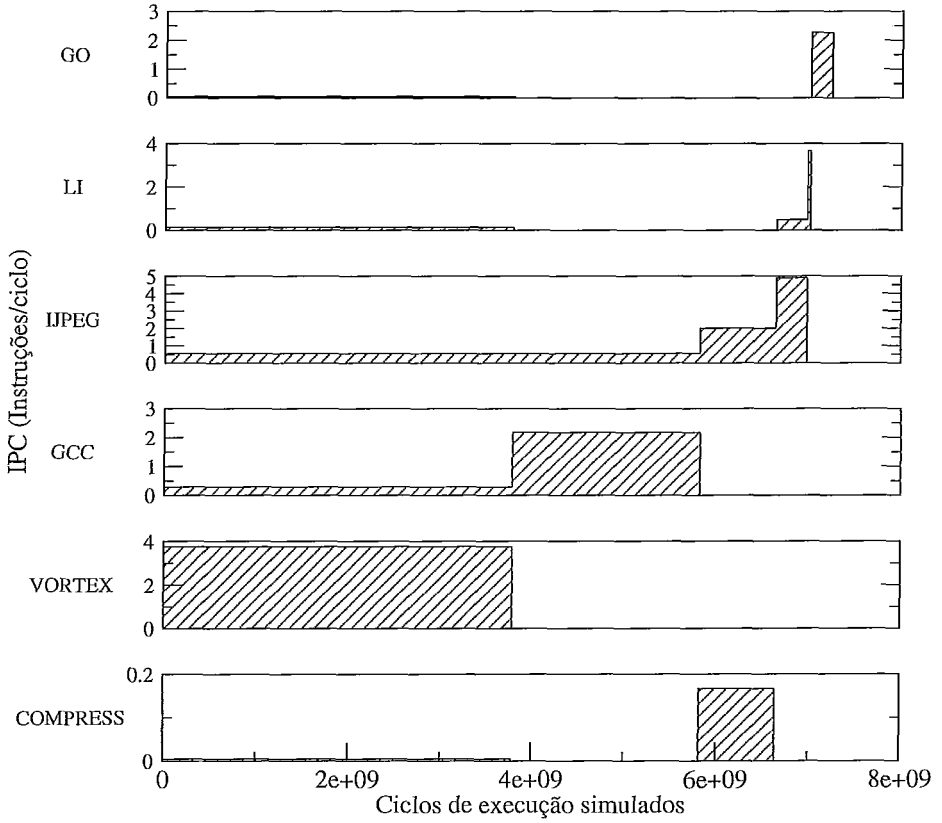

Figura 4.3: Evolução do IPC dos aplicativos da sessão 6A (GO, LI, IJPEG, GCC, VORTEX e COMPRESS) por trechos.

4.6 Avaliação geral da *Trace Cache*

Pode-se ter uma idéia geral do desempenho da *Trace Cache* sob multiprogramação calculando-se o IPC geral para todos os aplicativos da sessão. Este IPC será o somatório dos ciclos gastos pelos processos dividido pelo número de ciclos da sessão. Para as sessões de multiprogramação, pode-se usar o

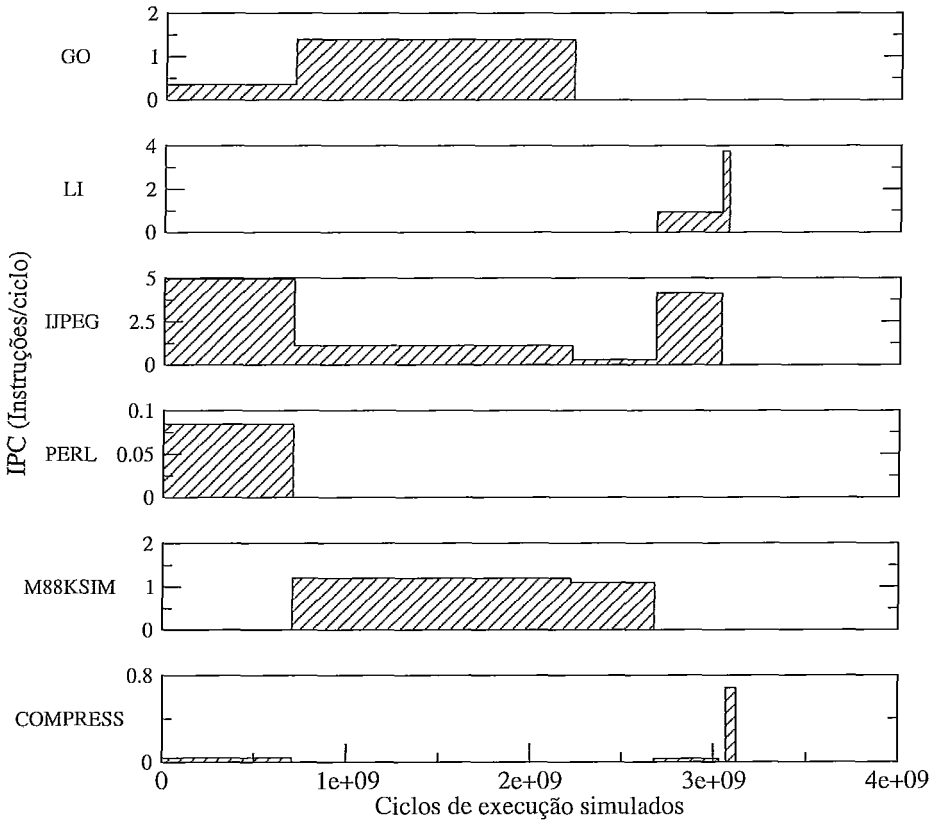

Figura 4.4: Evolução do IPC dos aplicativos da sessão 6B (GO, LI, IJPEG, PERL, M88KSIM e COMPRESS) por trechos.

Aplicativo	**Ciclos (milhões)**					
	706	2.220	2.678	3.030	3.068	3.122
GO	0.0522	0.0000	0.0000	0.0000	0.0000	2.2588
LI	0.1399	0.0000	0.0000	0.4862	3.6715	-
IJPEG	0.5551	0.5536	2.0046	4.8913	-	-
GCC	0.2911	2.1749	-	-	-	-
COMPRESS	0,0050	0,0000	0,1665	-	-	-
VORTEX	3,7505	-	-	-	-	-

Tabela 4.13: IPC por trechos para os aplicativos da sessão 6A (GO, LI, IJPEG, GCC, VORTEX & COMPRESS).

Aplicativo	Ciclos (milhões)					
	3.780	5.816	6.637	6.973	7.001	7.235
GO	0,3508	1,3867	-	-	-	-
LI	0,0018	0,0000	0,0000	0,9311	3,7446	-
IJPEG	4,9449	1,1089	0,2921	4,1373	-	-
M88KSIM	0,0110	1,1943	1,0900	-	-	-
PERL	0,0843	-	-	-	-	-
COMPRESS	0,0406	0,0000	0,0000	0,0315	0,0000	0,6842

Tabela 4.14: IPC por trechos para os aplicativos da sessão 6B (GO, LI, IJPEG, M88KSIM, PERL & COMPRESS).

somatório dos ciclos gastos por cada processo (desconsiderando as penalidades devido ao atendimento das chamadas de sistema) ou, então o total de ciclos (desta vez, considerando as penalidades). Neste trabalho, todos os valores de IPC calculados utilizaram a quantidade de ciclos dos processos, sem levar em conta as penalidades; esta metodologia, portanto será mantida nesta avaliação. Este cálculo também pode ser efetuado considerando todos os aplicativos sob monoprogramação, como se eles tivessem sido executados um imediatamente após o outro. A comparação entre estes valores de IPC dá uma idéia geral do efeito da multiprogramação sobre *Trace Caches*. A Tabela 4.15 mostra os valores obtidos sob multiprogramação e para as sessões de multiprogramação deste trabalho.

Os resultados da Tabela 4.15 mostram o efeito final da multiprogramação sobre a *Trace Cache*. Se, para cada nível de multiprogramação calcularmos o desempenho relativo como a razão entre o IPC sob multiprogramação e o IPC para monoprogramação e tirarmos a média dos valores obtidos, obteremos o gráfico da Figura 4.5. Este gráfico mostra o efeito médio provocado pela multiprogramação, no desempenho dos aplicativos, usando *Trace Cache*. Conforme visto, com o nível de multiprogramação 2, a queda de desempenho média foi de 40% aproximadamente, aumentando para cerca de 50% com nível de multiprogramação 2 e, por fim, 80%, sob nível de multiprogramação 6. Contudo, deve-se evitar o equívoco de considerar, apenas o nível de mul-

Aplicativos	**Regime**	IPC
GO & LI	monoprog.	2,7369
	multiprog.	1,4072
GCC & VORTEX	monoprog.	3,7870
	multiprog.	2,4768
LI & COMPRESS	monoprog.	3,7259
	multiprog.	1,9539
GO & IJPEG	monoprog.	4,4059
	multiprog.	2,4796
M88KSIM & PERL	monoprog.	3,6240
	multiprog.	2,8109
GO, LI, IJPEG & GCC	monoprog.	3,4160
	multiprog.	1,5927
LI, IJPEG, VORTEX & COMPRESS	monoprog.	5,3110
	multiprog.	1,7456
LI, IJPEG, GCC & COMPRESS	monoprog.	3,6631
	multiprog.	2,3868
GO, PERL, M88KSIM & VORTEX	monoprog.	4,2075
	multiprog.	1,9459
GO, LI, IJPEG, GCC, VORTEX & COMPRESS	monoprog.	3,9709
	multiprog.	0,8338
GO, LI, IJPEG, PERL, M88KSIM & COMPRESS	monoprog.	4,1794
	multiprog.	0.8788

Tabela 4.15: IPC conjunto dos aplicativos, sob monoprogramação e sob nível de multiprogramação 6.

tiprogramação como parâmetro para avaliação de desempenho. A Figura 4.5 apenas sumariza os resultados deste trabalho; com outros *benchmarks*, outra configuração de *Trace Cache* ou outros algoritmos de escalonamento de processos, os resultados sejam diferentes. Especial atenção deve ser prestada ao fato de o IPC dos aplicativos variar conforme o período avaliado, como se tornou claro com as gráficos das Figuras 4.3 e 4.4.

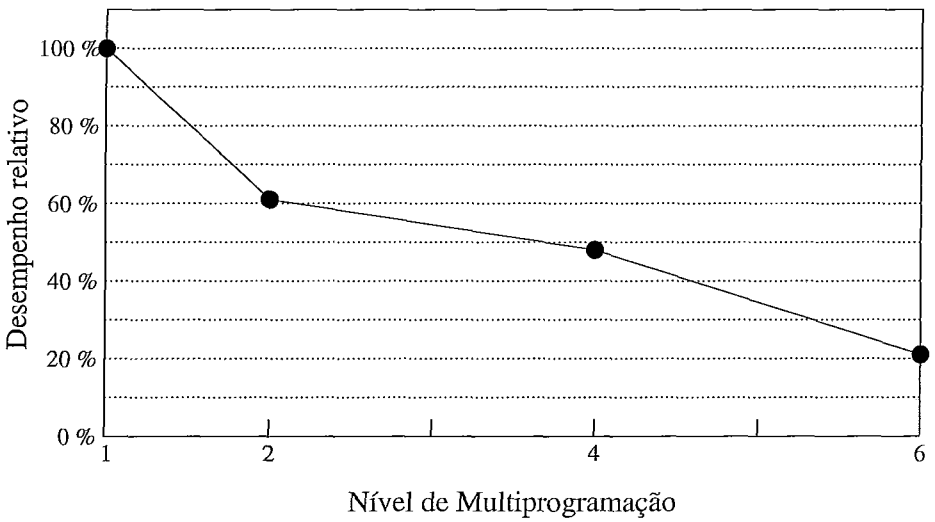

Figura 4.5: Desempenho relativo médio em relação à monoprogramação em função do nível de multiprogramação.

Capítulo 5

Conclusões e Trabalhos Futuros

5.1 Sumário

Este trabalho descreveu o comportamento de uma *Trace Cache* em sessões de multiprogramação, com níveis de multiprogramação variando de 1 até 6, passando por 2 e 4. Conforme esperado, a existência de vários processos se revezando no acesso à *Trace Cache* influiu negativamente no desempenho de cada processo, em relação ao caso em que cada um deles possui acesso exclusivo à *Trace Cache*.

Para avaliação do comportamento da *Trace Cache* numa sessão multiprogramada, foram utilizados o IPC para cada processo e a taxa de *miss* da *Trace Cache*. O melhor parâmetro para avaliação pretendida foi o IPC, que revelou dramáticas quedas de desempenho nos aplicativos, conforme variava o nível de multiprogramação e o conjunto de aplicativos nas sessões de multiprogramação.

Desempenho dos aplicativos

Este trabalho revelou que a queda no desempenho, nas condições aqui apresentadas dependente, não só do nível de multiprogramação, mas também, do conjunto de aplicações que compartilham o acesso à *Trace Cache*. As características, dos aplicativos, que mais influenciaram seus desempenhos

foram o número de instruções executadas e a quantidade de chamadas de sistema efetuadas. A maior ou menor influência destas características depende, não só de cada aplicativo, isoladamente considerado, mas, também, das características dos outros aplicativos participando da mesma sessão de multiprogramação. De modo geral, quanto maior o número de instruções e menor o número de chamadas de sistema realizadas, menos suscetível à influência dos demais processos participantes da sessão um aplicativo será.

A influência de um processo sobre os demais se dá através da substituição de *traces* de outros processos. Quanto mais substituições interprocessos um aplicativo realiza, em relação ao total de todos os outros processos, menor a queda de desempenho experimentada por ele.

Comprovou-se ser que a avaliação de *Trace Caches* sem levar em conta multiprogramação é inadequada. Ao final das avaliações, foi revelada uma queda de desempenho média de cerca de 40 % para o nível de multiprogramação igual a 2, 50% para o nível de multiprogramação 4 e 80 % para o nível de multiprogramação 6. Apesar destes números sumarizarem os resultados, não devem ser usados como um parâmetro para estimar o desempenho em situações muito diferentes das exploradas neste trabalho. A análise dos resultados mostrou que o desempenho dos aplicativos varia conforme o período considerado, em função do número de processos ainda participando da sessão de multiprogramação.

Taxa de *miss*

Descobriu-se que a taxa de *miss* da *Trace Cache* varia pouco em função de níveis modestos de multiprogramação, como aqueles usados neste trabalho. Este fato é conseqüência do baixo número de substituições de *traces* verificados neste trabalho. Também, a taxa de *miss* permanece alta (entre 22 e 54 %) para todos aplicativos em virtude do próprio funcionamento da *Trace Cache*; *misses* não provocam substituições imediatas de *traces*.

Os resultados também sugerem que há pouca relação entre a taxa de *miss* e o desempenho dos aplicativos sob multiprogramação. Contudo, há

necessidade de estudos mais aprofundados.

Limitações e contribuições

Este trabalho apresentou algumas limitações, tais como não levar em conta a execução do código do sistema operacional, bem como simular, no máximo, 6 processos acessando simultaneamente a *Trace Cache*. Apesar de tudo, os resultados indicam, claramente, o equívoco cometido ao não se levar em conta o efeito da multiprogramação sobre a eficiência dos elementos da hierarquia de memória.

Outras contribuições relevantes deste trabalho foram a implementação de um método de simulação de multiprogramação através da alocação das estruturas de dados comuns aos processos simulados em memória compartilhada e a modelagem simplificada do escalonamento de processos de um sistema operacional real.

5.2 Trabalhos Futuros

A partir do trabalho realizado e dos resultados obtidos, revelam-se, naturalmente, possibilidades de estender este trabalho, seja através do aumento do seu alcance, seja através de novos trabalhos. Algumas sugestões, relacionadas à simulação pura e simples são:

- Realizar mais avaliações, aumentando o número de processos simulados.
- Implementar o compartilhamento de outras estruturas do processador, tais como preditores, TLB's, etc.
- Implementar um modelo mais detalhado de acesso à Entrada/Saída (*I/O*) de modo a tornar mais realistas as penalidades impostas à estas operações.

Em relação à *Trace Cache*, o óbvio seria estudar de que maneiras se poderia diminuir sua vulnerabilidade à multiprogramação. Deve-se consi-

derar, também que, independente de qualquer consideração sobre multiprogramação, a eficiência das *Trace Caches* precisa melhorar; altas taxas de *miss* são um indicativo de baixa eficiência que podem diminuir a eficácia de medidas que tentem tornar a *Trace Cache* menos influenciável pela multiprogramação.

A partir da descoberta de que a queda de desempenho da *Trace Cache* sob monoprogramação está associada a substituições de *traces* interprocessos, os esforços devem se concentrar neste problema. Provavelmente, a solução envolve a modificação do algoritmo de escalonamento de processos, de modo a favorecer a permanência de um determinado processo por maiores períodos de tempo, de tal modo que a eficiência da *Trace Cache* seja menos afetada pelo alternância de processos.

Outro possível trabalho seria explorar a eficiência de medidas tais como o aproveitamento de *traces*, que de outro modo seriam abortados [1], desde que sejam satisfeitos certos critérios de eficiência. Esta otimização foi introduzida neste trabalho devido ao fato de, para a configuração de *Trace Cache* utilizada neste trabalho, ter se revelado vantajosa. Contudo, não há evidências de que isto sempre seja benéfico.

[1] Devido a ocorrência de instruções que não podem fazer parte de um *trace*: desvios indiretos, chamadas de sistema, etc.

Bibliografia

[1] Agarwal, A., Hennessy, J., Horowitz, M. "Cache Performance of Operating System and Multiprogramming Workloads". *ACM Transactions on Computer Systems*, v. 6, n. 4, pp. 393–431, November 1988.

[2] "Product Brief: Élan SC520 Microcontroller". Disponível em `http://www.amd.com/epd/processors/4.32bitcont/14.lan5xxfam/24.lansc520/brief/`.

[3] Anderson, T. E., Levy, H. M., Bershad, B. N., Lazowska, E. D. "The Interaction of Architecture and Operating System Design". In *ASPLOS, International Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 108–120, Santa Clara, CA (USA), 1991.

[4] Beck, M., Bohme, H., Dziadzka, M., Kunitz, U., Magnus, R., Verworner, D. *Linux Kernel Internals.* Addison-Wesley Longman Limited, Edinburgh Gate, Harlow, Essex, England, third ed., 1996.

[5] Black, B., Rychlik, B., Shen, J. P. "The Block-based Trace Cache". In *Proceedings of the 26th Annual Intl. Symposium on Computer Architecture*, pp. 196–207. IEEE Computer Society Press, May 1999.

[6] Bowman, I. "Conceptual Architecture of the Linux Kernel". Disponível em `http://www.grad.math.uwaterloo.ca/~itbowmann/CS746G/a1/`, January 1998.

[7] Burger, D., Austin, T. M. *The SimpleScalar Tool Set, Version 2.0.* Technical report, University of Wisconsin-Madison Computer Sciences Department, June 1997.

[8] Clark, D. W., Emer, J. S. "Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement". *ACM Transactions on Computer Systems*, v. 3, n. 1, pp. 31–62, February 1985.

[9] Dixit, K., Reilly, J. "SPEC CPU95 Q&A". Disponível em `http://www.spec.org/cpu95/qanda.html`, September 1999.

[10] Goldt, Sven, van der Meer, Sven, Burkett, Scott, Welsh, Matt. *The Linux Programmer's Guide.* The Linux Documentation Project, http://www.tldp.org, 0.4 ed., March 1995.

[11] Hennessy, J., Patterson, D. *Computer Architecture: A Quantitative Approach, 2nd ed.* Morgan Kaufmann Publishers Inc., Palo Alto, CA 94303, 1995.

[12] Hwu, W. W., Patt, Y. N. "Checkpoint Repair for Out-of-order Execution Machines". In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pp. 18–26, 1987.

[13] Kisuki, T., Corporaal, H., Knijnenburg, P. *The Effect of Process Switches on Branch Prediction Accuracy.* Technical report, Department of Computer Science, Leiden University, January 1999.

[14] Mogul, J. C., Borg, A. *The Effect of Context Switches on Cache Performance.* Technical report, Digital Equipment Corporation, Western Research Laboratory, 100 Hamilton Avenue Palo Alto, California 94301 USA, December 1990. Published in Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 75–84, Santa Clara, CA, USA, April 1991.

[15] Patel, S., Evers, M., Patt, Y. "Improving Trace Cache Effectiveness with Branch Promotion and Trace Packing". In *Proceedings of the 25th*

Annual International Symposium on Computer Architecture (ISCA-98), v. 26,3, *ACM Computer Architecture News*, pp. 262–271, New York, June 27–July 1 1998. ACM Press.

[16] Patel, S. J., Friendly, D. H., Patt, Y. N. *Critical Issues Regarding the Trace Cache Fetch Mechanism.* Technical Report CSE-TR-335-97, Computer Science and Engineering, University of Michigan, May 7 1997. Thu, 15 Oct 1998 15:18:24 GMT.

[17] Patel, S. J., Friendly, D. H., Patt, Y. N. "Evaluation of Design Options for the Trace Cache Fetch Mechanism". *IEEE Transactions on Computers*, v. 48, n. 2, pp. 193–204, 1999.

[18] Peleg, A., Weiser, U. *Dynamic flow instruction cache memory organized around traces segments independents of virtual line, U.S. Patent Number 5,381,533.* Technical report, US Patent & Trademark Office, January 1995.

[19] Rotenberg, E., Bennett, S., Smith, J. E. "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching". In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pp. 24–34, Paris, France, December 2–4, 1996. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[20] Rotenberg, E., Jacobson, Q., Sazeides, Y., Smith, J. "Trace Processors". In *International Symposium on Microarchitecture*, pp. 138–148, 1997.

[21] Rusling, D. A. *The Linux Kernel.* The Linux Documentation Project, 3 Foxglove Close, Wokingham, Berkshire RG41 3NF, UK, January 1999.

[22] Schlapbach, A. "Linux Process Scheduling, version 0.1". Disponível em `http://iamexwiwww.unibe.ch/studenten/schlpbch/linuxScheduling/LinuxScheduling.html`, May 2000.

[23] Smith, A. J. "Cache Memories". *ACM Computing Surveys*, v. 14, n. 3, pp. 473–530, September 1982.

[24] Smith, J., Sohi, G. "The Microarchitecture of Superscalar Processors". In *Proceedings of the IEEE*, v. 83, December 1995.

[25] Sohi, G. S. "Instruction Issue Logic for High-performance, Interruptible, Multiple Functional Unit, Pipelined Processors". *IEEE Transactions on Computers*, v. 39, n. 3, pp. 349–359, March 1990.

[26] Torrellas, J., Xia, C., Daigle, R. L. "Optimizing the Instruction Cache Performance of the Operating System". *IEEE Transactions on Computers*, v. 47, n. 12, pp. 1363–1381, 1998.

[27] Vahalia, U. *Unix Internals: The New Frontiers.* Prentice-Hall, Upper Saddle River, New Jersey, 1996.

[28] Wilkes, M. "Slave Memories and Dynamic Storage Allocation". *IEEE Transactions on Electronic Computers*, v. EC-14, pp. 270–271, 1965.

Apêndice A

Penalidades das Chamadas de Sistema

Neste trabalho, foram atribuídas penalidades às chamadas de sistema realizados pelos aplicativos simulados pelo **SimpleScalar**. O valor das penalidades não foram os mesmos para todas as chamadas de sistema (na verdade, algumas das chamadas de sistema não penalizavam o aplicativo). Na Tabela A.1, a seguir, é dada a lista de chamadas de sistema suportadas pelo **SimpleScalar** e o tipo de penalidade a ela atribuída.

Há 4 tipos tipos de penalidade: **Criação de arquivo** (que, neste trabalho, recebeu o valor de 500 μs), penalidade de **Leitura de arquivo** (que, neste trabalho recebeu o valor de 200 μs), **Sem penalidade**, que apenas provocava a ação do escalonador e **Sem escalonamento** que, além de não provocar nenhuma penalização, também não solicitavam a ação do escalonador.

A classificação do tipo de penalidade foi derivada da observação direta do código fonte do Kernel Linux 2.0.33.

Tipo de penalidade	Chamadas de Sistema
Criação de arquivo	open, creat, close
Leitura de arquivo	read, write, unlink, chdir, chmod, chown, lseek, access, stat, lstat, pipe, ioctl, fstat, dup2, select, writev, utimes, getrlimit, setrlimit
Sem penalidade	brk, getuid, getpid, dup, getgid, getpagesize, setitimer getdtablesize, gettimeofday
Sem escalonamento	getrusage, fcntl, sigvec, sigblock, sigsetmask

Tabela A.1: Tipos de penalidades atribuídas às chamadas de sistema do **SimpleScalar**.

Apêndice B

Valores de IPC por trechos

Neste apêndide, são mostrados gráficos e tabelas com os valores de IPC em cada trecho das sessões de multiprogramação. Cada trecho é um intervalo que começa com o início da sessão ou com o fim da execução de um aplicativo e termina com póximo término de execução de outro aplicativo. As sessões são nomeadas, de acordo com os aplicativos envolvidos na sessão de multiprogramação, segundo a Tabela B.1

Nome	Sessões
2A	GO & LI
2B	GCC & VORTEX
2C	LI & COMPRESS
2D	GO & IJPEG
2E	M88KSIM & PERL
4A	GO, LI, IJPEG & GCC
4B	LI, IJPEG, VORTEX & COMPRESS
4C	LI, IJPEG, GCC & COMPRESS
4D	GO, PERL, M88KSIM & VORTEX
6A	GO, LI, IJPEG, GCC, VORTEX & COMPRESS
6B	GO, LI, IJPEG, PERL, M88KSIM & COMPRESS

Tabela B.1: Nomes das Sessões de Multiprogramação.

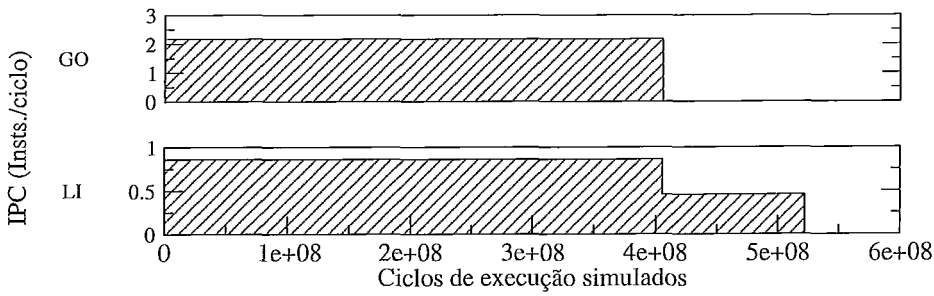

Figura B.1: Evolução do IPC dos aplicativos da sessão 2A por trechos.

Aplicativo	Ciclos (milhões)	
	406	522
GO	2,1696	-
LI	0,8600	0,4563

Tabela B.2: IPC por trechos para os aplicativos da sessão 2A.

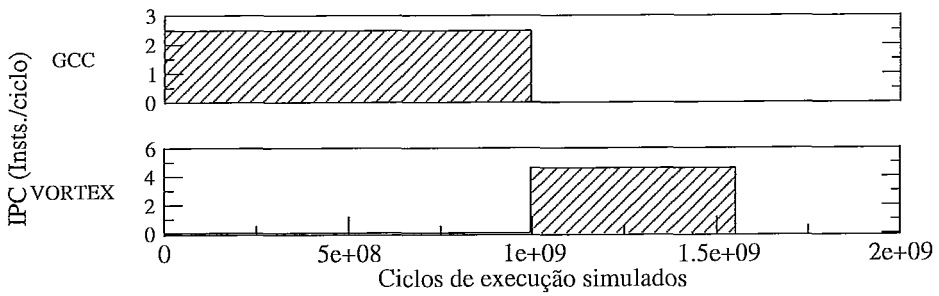

Figura B.2: IPC dos aplicativos da sessão 2B por trechos.

Aplicativo	Ciclos (milhões)	
	995	1551
GCC	2,4759	-
VORTEX	0,0480	4,5938

Tabela B.3: IPC por trechos para os aplicativos da sessão 2B.

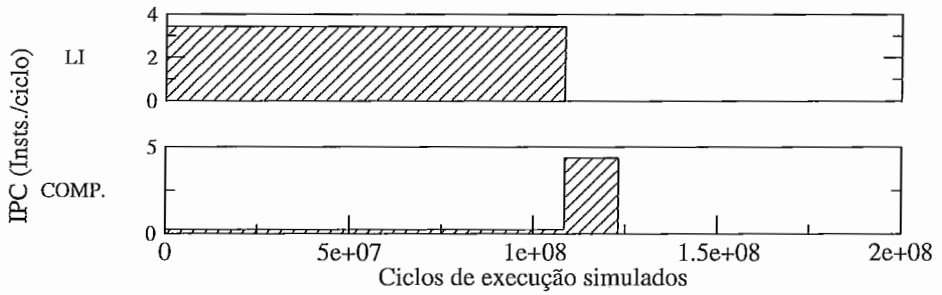

Figura B.3: Evolução do IPC dos aplicativos da sessão 2C por trechos.

Aplicativo	**Ciclos (milhões)**	
	108	123
LI	3,4313	-
COMPRESS	0,2358	4,3804

Tabela B.4: IPC por trechos para os aplicativos da sessão 2C.

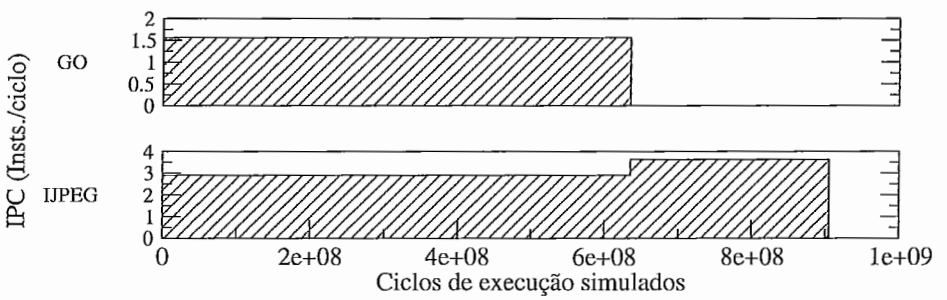

Figura B.4: Evolução do IPC dos aplicativos da sessão 2D por trechos.

Aplicativo	**Ciclos (milhões)**	
	636	905
GO	1,5614	-
IJPEG	2,9017	3,6233

Tabela B.5: IPC por trechos para os aplicativos da sessão 2D.

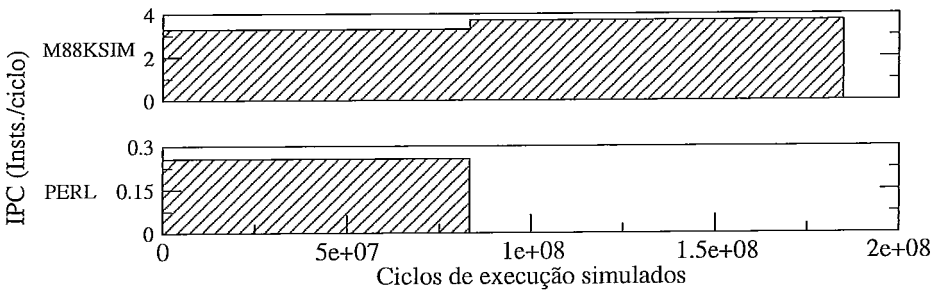

Figura B.5: Evolução do IPC dos aplicativos da sessão 2E (M88KSIM & PERL) por trechos.

Aplicativo	**Ciclos (milhões)**	
	83	185
M88KSIM	3,2858	3,7009
PERL	0,2560	-

Tabela B.6: IPC por trechos para os aplicativos da sessão 2E.

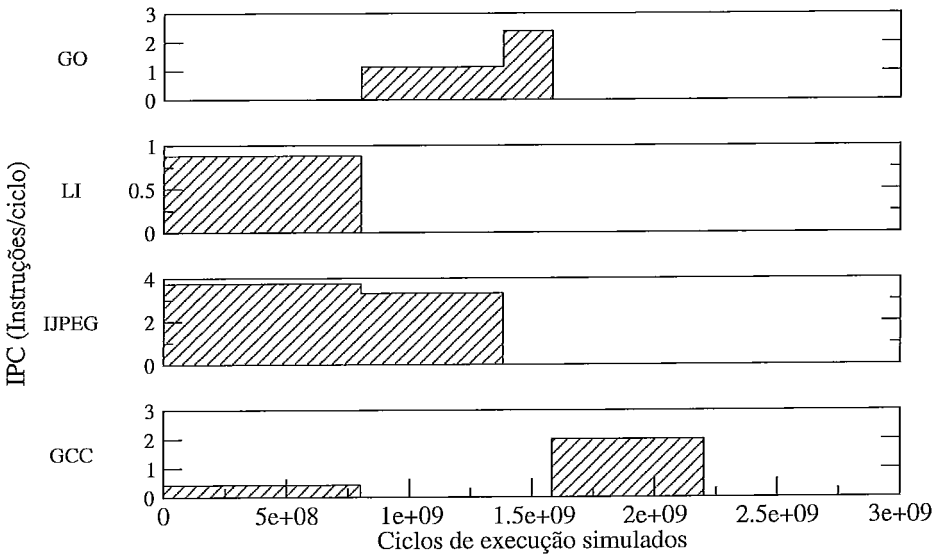

Figura B.6: Evolução do IPC dos aplicativos da sessão 4A por trechos.

Aplicativo	**Ciclos (milhões)**			
	802	1379	1582	2198
GO	0,01046	1,1419	2,3884	-
LI	0,8805	-	-	-
IJPEG	3,7516	3.3079	-	-
GCC	0,4219	0,0000	0,0000	2,0028

Tabela B.7: IPC por trechos para os aplicativos da sessão 4A.

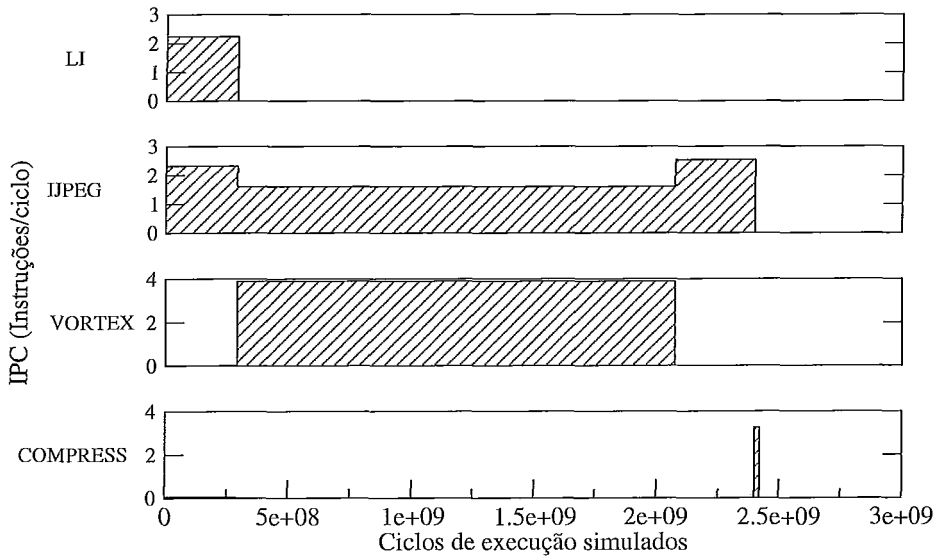

Figura B.7: Evolução do IPC dos aplicativos da sessão 4B por trechos.

Aplicativo	Ciclos (milhões)			
	289	2073	2399	2421
LI	2,2375	-	-	-
IJPEG	2,3219	1,5995	2,5366	-
VORTEX	0,0131	3,8902	-	-
COMPRESS	0,0636	0.0000	0.0000	3,2448

Tabela B.8: IPC por trechos para os aplicativos da sessão 4B.

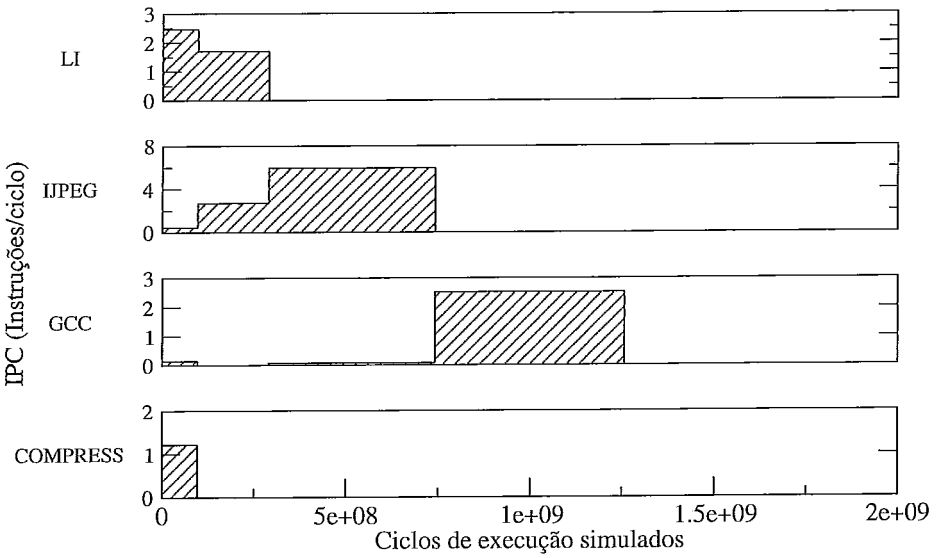

Figura B.8: Evolução do IPC dos aplicativos da sessão 4C por trechos.

Aplicativo	**Ciclos (milhões)**			
	97	290	741	1258
LI	2,4642	1,7032	-	-
IJPEG	0,4461	2,7038	5,9504	-
GCC	0,1460	0,0000	0,0768	2,5126
COMPRESS	1.216818	-	-	-

Tabela B.9: IPC por trechos para os aplicativos da sessão 4C.

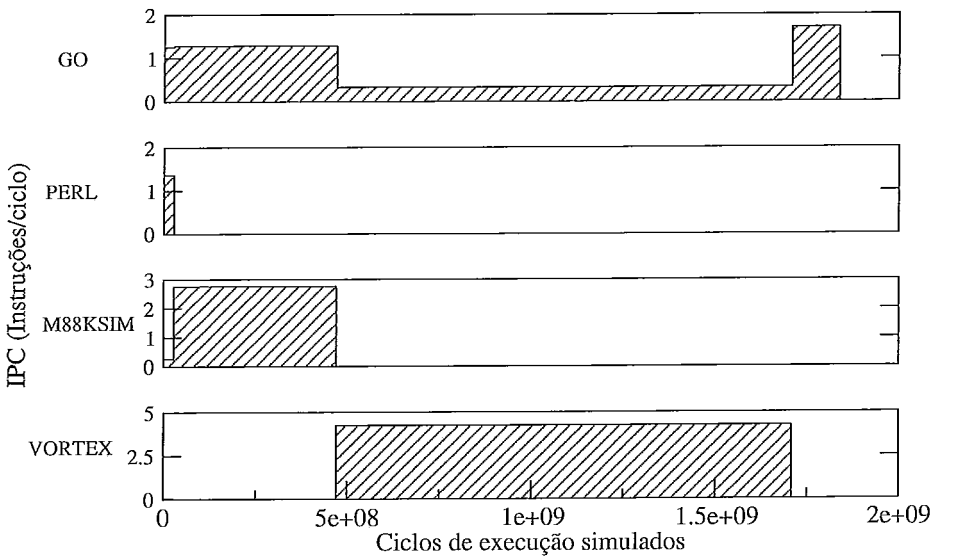

Figura B.9: Evolução do IPC dos aplicativos da sessão 4D por trechos.

Aplicativo	**Ciclos (milhões)**			
	29	470	1708	1837
GO	1,2557	1,2821	0,3312	1,6980
PERL	1,3636	-	-	-
M88KSIM	0,2573	2.7613	-	-
VORTEX	0,0588	0,0133	4.2407	-

Tabela B.10: IPC por trechos para os aplicativos da sessão 4D.

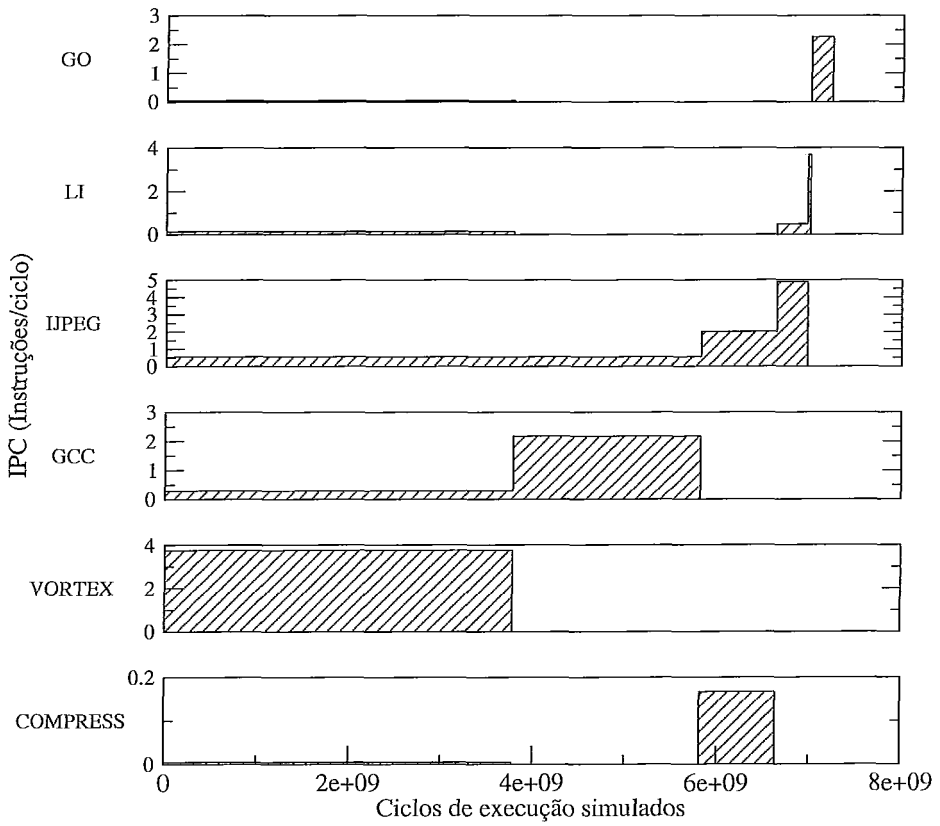

Figura B.10: Evolução do IPC dos aplicativos da sessão 6A (GO, LI, IJPEG, GCC, VORTEX e COMPRESS) por trechos.

Aplicativo	**Ciclos (milhões)**					
	706	2220	2678	3030	3068	3122
GO	0,0522	0,0000	0,0000	0,0000	0,0000	2,2588
LI	0,1399	0,0000	0,0000	0,4862	3,6715	-
IJPEG	0,5551	0,5536	2,0046	4,8913	-	-
GCC	0,2911	2,1749	-	-	-	-
COMPRESS	0,0050	0,0000	0,1665	-	-	-
VORTEX	3,7505	-	-	-	-	-

Tabela B.11: IPC por trechos para os aplicativos da sessão 6A (GO, LI, IJPEG, GCC, VORTEX & COMPRESS).

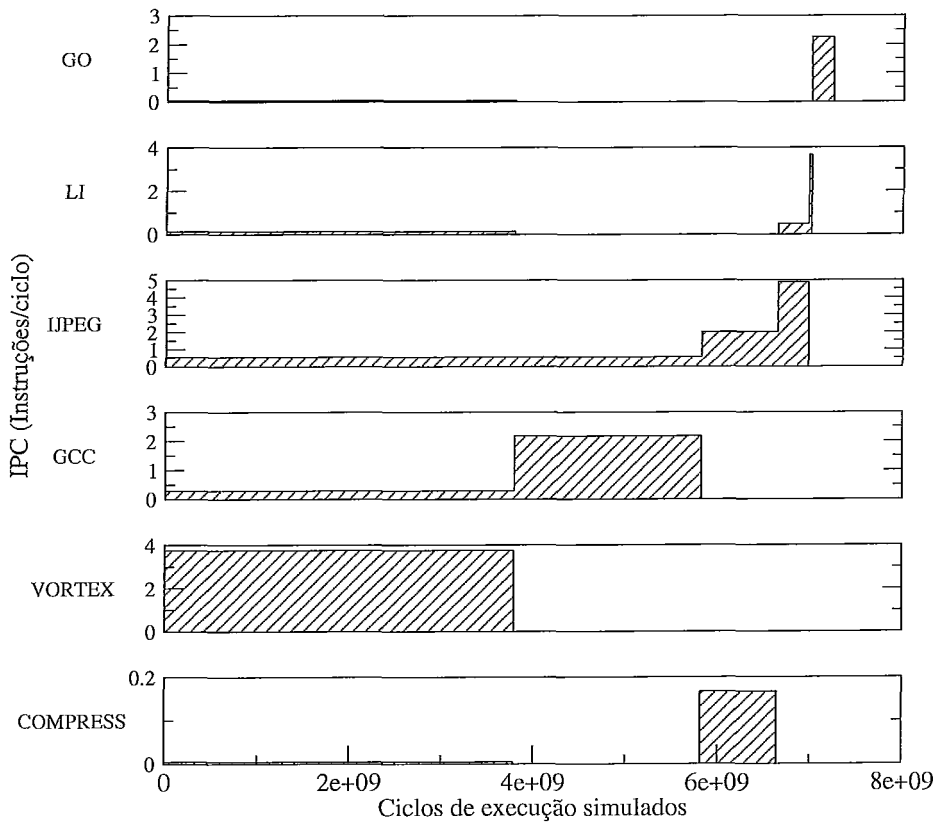

Figura B.11: Evolução do IPC dos aplicativos da sessão 6A (GO, LI, IJPEG, GCC, VORTEX e COMPRESS) por trechos.

Aplicativo	**Ciclos (milhões)**					
	706	2220	2678	3030	3068	3122
GO	0,0522	0,0000	0,0000	0,0000	0,0000	2,2588
LI	0,1399	0,0000	0,0000	0,4862	3,6715	-
IJPEG	0,5551	0,5536	2,0046	4,8913	-	-
GCC	0,2911	2,1749	-	-	-	-
COMPRESS	0,0050	0,0000	0,1665	-	-	-
VORTEX	3,7505	-	-	-	-	-

Tabela B.12: IPC por trechos para os aplicativos da sessão 6B (GO, LI, IJPEG, GCC, VORTEX & COMPRESS).

Apêndice C

Modificações no SimpleScalar

A seguir, exibe-se a tela de apresentação do simulador **sim-outorder** do **SimpleScalar**, após as modificações efetuadas para realização deste trabalho. Estas informações podem ser obtidas invocando-se o simulador **sim-outorder** sem argumentos.

```
sim-outorder: SimpleScalar/PISA Tool Set version 3.0 of November, 2000.
Copyright (c) 1994-2000 by Todd M. Austin.  All Rights Reserved.
This version of SimpleScalar is licensed for academic non-commercial use only.
This version contains Multiprogramming support by Alvaro Ferreira
All/Only additions or modifications by Alvaro Ferreira are GPL covered.

Usage: ./sim-outorder {-options} executable {arguments}

sim-outorder: This simulator implements a very detailed out-of-order issue
superscalar processor with a two-level memory system and speculative
execution support.  This simulator is a performance simulator, tracking the
latency of all pipeline operations.

#
# -option        <args>          #    <default> # description
#
-config          <string>        #       <null> # load configuration from a file
-dumpconfig      <string>        #       <null> # dump configuration to a file
-h               <true|false>    #        false # print help message
-v               <true|false>    #        false # verbose operation
-d               <true|false>    #        false # enable debug message
-multi           <int>           #            0 # multiprogramming level
-i               <true|false>    #        false # start in Dlite debugger
-seed            <int>           #            1 # random number generator seed (0 for t
imer seed)
```

```
-q              <true|false>     #        false # initialize and terminate immediately
-chkpt          <string>         #       <null> # restore EIO trace execution from <fna
me>
-redir:sim      <string>         #       <null> # redirect simulator output to file (no
n-interactive only)
-redir:prog     <string>         #       <null> # redirect simulated program output to
file
-nice           <int>            #            0 # simulator scheduling priority
-max:inst       <uint>           #            0 # maximum number of inst's to execute
-fastfwd        <int>            #            0 # number of insts skipped before timing
 starts
-ptrace         <string list...> #       <null> # generate pipetrace, i.e., <fname|stdo
ut|stderr> <range>
-fetch:ifqsize  <int>            #            4 # instruction fetch queue size (in inst
s)
-fetch:mplat    <int>            #            3 # extra branch mis-prediction latency
-fetch:speed    <int>            #            1 # speed of front-end of machine relativ
e to execution core
-bpred          <string>         #        bimod # branch predictor type {nottaken|taken
|perfect|bimod|2lev|comb}
-bpred:bimod    <int>            # 2048 # bimodal predictor config (<table size>)
-bpred:2lev     <int list...>    # 1 1024 8 0 # 2-level predictor config (<l1size> <l2s
ize> <hist_size> <xor>)
-bpred:comb     <int>            # 1024 # combining predictor config (<meta_table_size>
)
-bpred:ras      <int>            #            8 # return address stack size (0 for no r
eturn stack)
-bpred:btb      <int list...>    # 512 4 # BTB config (<num_sets> <associativity>)
-bpred:spec_update <string>        #       <null> # speculative predictors update in {I
D|WB} (default non-spec)
-decode:width   <int>            #            4 # instruction decode B/W (insts/cycle)
-issue:width    <int>            #            4 # instruction issue B/W (insts/cycle)
-issue:inorder  <true|false>     #        false # run pipeline with in-order issue
-issue:wrongpath <true|false>    #         true # issue instructions down wrong executi
on paths
-commit:width   <int>            #            4 # instruction commit B/W (insts/cycle)
-ruu:size       <int>            #           16 # register update unit (RUU) size
-lsq:size       <int>            #            8 # load/store queue (LSQ) size
-cache:dl1      <string>         # dl1:128:32:4:l # l1 data cache config, i.e., {<confi
g>|none}
-cache:dl1lat   <int>            #            1 # l1 data cache hit latency (in cycles)
-cache:dl2      <string>         # ul2:1024:64:4:l # l2 data cache config, i.e., {<conf
ig>|none}
-cache:dl2lat   <int>            #            6 # l2 data cache hit latency (in cycles)
-cache:il1      <string>         # il1:512:32:1:l # l1 inst cache config, i.e., {<confi
g>|dl1|dl2|none}
-cache:il1lat   <int>            #            1 # l1 instruction cache hit latency (in
```

```
cycles)
-cache:il2       <string>         #          dl2 # l2 instruction cache config, i.e., {<
config>|dl2|none}
-cache:il2lat    <int>            #            6 # l2 instruction cache hit latency (in
cycles)
-cache:flush     <true|false>     #        false # flush caches on system calls
-cache:icompress <true|false>     #        false # convert 64-bit inst addresses to 32-b
it inst equivalents
-mem:lat         <int list...>    # 18 2 # memory access latency (<first_chunk> <inter_c
hunk>)
-mem:width       <int>            #            8 # memory access bus width (in bytes)
-tlb:itlb        <string>         # itlb:16:4096:4:l # instruction TLB config, i.e., {<c
onfig>|none}
-tlb:dtlb        <string>         # dtlb:32:4096:4:l # data TLB config, i.e., {<config>|
none}
-tlb:lat         <int>            #           30 # inst/data TLB miss latency (in cycles
)
-res:ialu        <int>            #            4 # total number of integer ALU's availab
le
-res:imult       <int>            #            1 # total number of integer multiplier/di
viders available
-res:memport     <int>            #            2 # total number of memory system ports a
vailable (to CPU)
-res:fpalu       <int>            #            4 # total number of floating point ALU's
available
-res:fpmult      <int>            #            1 # total number of floating point multip
lier/dividers available
-pcstat          <string list...> #       <null> # profile stat(s) against text addr's (
mult uses ok)
-bugcompat       <true|false>     #        false # operate in backward-compatible bugs m
ode (for testing only)
```

Pipetrace range arguments are formatted as follows:

{{@|#}<start>}:{{@|#|+}<end>}

Both ends of the range are optional, if neither are specified, the entire execution is traced. Ranges that start with a '@' designate an address range to be traced, those that start with an '#' designate a cycle count range. All other range values represent an instruction count range. The second argument, if specified with a '+', indicates a value relative to the first argument, e.g., 1000:+100 == 1000:1100. Program symbols may be used in all contexts.

Examples:
```
-ptrace FOO.trc #0:#1000
-ptrace BAR.trc @2000:
-ptrace BLAH.trc :1500
```

```
                -ptrace UXXE.trc :
                -ptrace FOOBAR.trc @main:+278

    Branch predictor configuration examples for 2-level predictor:
      Configurations:   N, M, W, X
        N   # entries in first level (# of shift register(s))
        W   width of shift register(s)
        M   # entries in 2nd level (# of counters, or other FSM)
        X   (yes-1/no-0) xor history and address for 2nd level index
      Sample predictors:
        GAg     : 1, W, 2^W, 0
        GAp     : 1, W, M (M > 2^W), 0
        PAg     : N, W, 2^W, 0
        PAp     : N, W, M (M == 2^(N+W)), 0
        gshare  : 1, W, 2^W, 1
    Predictor 'comb' combines a bimodal and a 2-level predictor.

    The cache config parameter <config> has the following format:

      <name>:<nsets>:<bsize>:<assoc>:<repl>

      <name>   - name of the cache being defined
      <nsets>  - number of sets in the cache
      <bsize>  - block size of the cache
      <assoc>  - associativity of the cache
      <repl>   - block replacement strategy, 'l'-LRU, 'f'-FIFO, 'r'-random

      Examples:   -cache:dl1 dl1:4096:32:1:l
                  -dtlb dtlb:128:4096:32:r

    Cache levels can be unified by pointing a level of the instruction cache
    hierarchy at the data cache hiearchy using the "dl1" and "dl2" cache
    configuration arguments.  Most sensible combinations are supported, e.g.,

      A unified l2 cache (il2 is pointed at dl2):
        -cache:il1 il1:128:64:1:l -cache:il2 dl2
        -cache:dl1 dl1:256:32:1:l -cache:dl2 ul2:1024:64:2:l

      Or, a fully unified cache hierarchy (il1 pointed at dl1):
        -cache:il1 dl1
        -cache:dl1 ul1:256:32:1:l -cache:dl2 ul2:1024:64:2:l


sim-outorder: SimpleScalar/PISA Tool Set version 3.0 of November, 2000.
Copyright (c) 1994-2000 by Todd M. Austin.  All Rights Reserved.
This version of SimpleScalar is licensed for academic non-commercial use only.
This version contains Multiprogramming support by Alvaro Ferreira
All/Only additions or modifications by Alvaro Ferreira are GPL covered.
```

```
Usage: ./sim-outorder {-options} executable {arguments}

sim-outorder: This simulator implements a very detailed out-of-order issue
superscalar processor with a two-level memory system and speculative
execution support.  This simulator is a performance simulator, tracking the
latency of all pipeline operations.

#
# -option        <args>           #    <default> # description
#
-config          <string>         #       <null> # load configuration from a file
-dumpconfig      <string>         #       <null> # dump configuration to a file
-h               <true|false>     #        false # print help message
-v               <true|false>     #        false # verbose operation
-d               <true|false>     #        false # enable debug message
-multi           <int>            #            0 # multiprogramming level
-i               <true|false>     #        false # start in Dlite debugger
-seed            <int>            #            1 # random number generator seed (0 for t
imer seed)
-q               <true|false>     #        false # initialize and terminate immediately
-chkpt           <string>         #       <null> # restore EIO trace execution from <fna
me>
-redir:sim       <string>         #       <null> # redirect simulator output to file (no
n-interactive only)
-redir:prog      <string>         #       <null> # redirect simulated program output to
file
-nice            <int>            #            0 # simulator scheduling priority
-max:inst        <uint>           #            0 # maximum number of inst's to execute
-fastfwd         <int>            #            0 # number of insts skipped before timing
 starts
-ptrace          <string list...> #       <null> # generate pipetrace, i.e., <fname|stdo
ut|stderr> <range>
-fetch:ifqsize   <int>            #            4 # instruction fetch queue size (in inst
s)
-fetch:mplat     <int>            #            3 # extra branch mis-prediction latency
-fetch:speed     <int>            #            1 # speed of front-end of machine relativ
e to execution core
-bpred           <string>         #        bimod # branch predictor type {nottaken|taken
|perfect|bimod|2lev|comb}
-bpred:bimod     <int>            # 2048 # bimodal predictor config (<table size>)
-bpred:2lev      <int list...>    # 1 1024 8 0 # 2-level predictor config (<l1size> <l2s
ize> <hist_size> <xor>)
-bpred:comb      <int>            # 1024 # combining predictor config (<meta_table_size>
)
-bpred:ras       <int>            #            8 # return address stack size (0 for no r
eturn stack)
```

```
-bpred:btb       <int list...>    # 512 4 # BTB config (<num_sets> <associativity>)
-bpred:spec_update <string>         #       <null> # speculative predictors update in {I
D|WB} (default non-spec)
-decode:width    <int>            #            4 # instruction decode B/W (insts/cycle)
-issue:width     <int>            #            4 # instruction issue B/W (insts/cycle)
-issue:inorder   <true|false>     #        false # run pipeline with in-order issue
-issue:wrongpath <true|false>     #         true # issue instructions down wrong executi
on paths
-commit:width    <int>            #            4 # instruction commit B/W (insts/cycle)
-ruu:size        <int>            #           16 # register update unit (RUU) size
-lsq:size        <int>            #            8 # load/store queue (LSQ) size
-cache:dl1       <string>         # dl1:128:32:4:l # l1 data cache config, i.e., {<confi
g>|none}
-cache:dl1lat    <int>            #            1 # l1 data cache hit latency (in cycles)
-cache:dl2       <string>         # ul2:1024:64:4:l # l2 data cache config, i.e., {<conf
ig>|none}
-cache:dl2lat    <int>            #            6 # l2 data cache hit latency (in cycles)
-cache:il1       <string>         # il1:512:32:1:l # l1 inst cache config, i.e., {<confi
g>|dl1|dl2|none}
-cache:il1lat    <int>            #            1 # l1 instruction cache hit latency (in
cycles)
-cache:il2       <string>         #          dl2 # l2 instruction cache config, i.e., {<
config>|dl2|none}
-cache:il2lat    <int>            #            6 # l2 instruction cache hit latency (in
cycles)
-cache:flush     <true|false>     #        false # flush caches on system calls
-cache:icompress <true|false>     #        false # convert 64-bit inst addresses to 32-b
it inst equivalents
-mem:lat         <int list...>    # 18 2 # memory access latency (<first_chunk> <inter_c
hunk>)
-mem:width       <int>            #            8 # memory access bus width (in bytes)
-tlb:itlb        <string>         # itlb:16:4096:4:l # instruction TLB config, i.e., {<c
onfig>|none}
-tlb:dtlb        <string>         # dtlb:32:4096:4:l # data TLB config, i.e., {<config>|
none}
-tlb:lat         <int>            #           30 # inst/data TLB miss latency (in cycles
)
-res:ialu        <int>            #            4 # total number of integer ALU's availab
le
-res:imult       <int>            #            1 # total number of integer multiplier/di
viders available
-res:memport     <int>            #            2 # total number of memory system ports a
vailable (to CPU)
-res:fpalu       <int>            #            4 # total number of floating point ALU's
available
-res:fpmult      <int>            #            1 # total number of floating point multip
lier/dividers available
```

```
-pcstat          <string list...> #       <null> # profile stat(s) against text addr's (
mult uses ok)
-bugcompat       <true|false>     #        false # operate in backward-compatible bugs m
ode (for testing only)

  Pipetrace range arguments are formatted as follows:

    {{@|#}<start>}:{{@|#|+}<end>}

  Both ends of the range are optional, if neither are specified, the entire
  execution is traced.  Ranges that start with a '@' designate an address
  range to be traced, those that start with an '#' designate a cycle count
  range.  All other range values represent an instruction count range.  The
  second argument, if specified with a '+', indicates a value relative
  to the first argument, e.g., 1000:+100 == 1000:1100.  Program symbols may
  be used in all contexts.

    Examples:   -ptrace FOO.trc #0:#1000
                -ptrace BAR.trc @2000:
                -ptrace BLAH.trc :1500
                -ptrace UXXE.trc :
                -ptrace FOOBAR.trc @main:+278

  Branch predictor configuration examples for 2-level predictor:
    Configurations:   N, M, W, X
      N   # entries in first level (# of shift register(s))
      W   width of shift register(s)
      M   # entries in 2nd level (# of counters, or other FSM)
      X   (yes-1/no-0) xor history and address for 2nd level index
    Sample predictors:
      GAg     : 1, W, 2^W, 0
      GAp     : 1, W, M (M > 2^W), 0
      PAg     : N, W, 2^W, 0
      PAp     : N, W, M (M == 2^(N+W)), 0
      gshare  : 1, W, 2^W, 1
  Predictor 'comb' combines a bimodal and a 2-level predictor.

  The cache config parameter <config> has the following format:

    <name>:<nsets>:<bsize>:<assoc>:<repl>

    <name>    - name of the cache being defined
    <nsets>   - number of sets in the cache
    <bsize>   - block size of the cache
    <assoc>   - associativity of the cache
    <repl>    - block replacement strategy, 'l'-LRU, 'f'-FIFO, 'r'-random
```

```
  Examples:   -cache:dl1 dl1:4096:32:1:l
              -dtlb dtlb:128:4096:32:r
```

Cache levels can be unified by pointing a level of the instruction cache hierarchy at the data cache hieachy using the "dl1" and "dl2" cache configuration arguments. Most sensible combinations are supported, e.g.,

```
  A unified l2 cache (il2 is pointed at dl2):
    -cache:il1 il1:128:64:1:l -cache:il2 dl2
    -cache:dl1 dl1:256:32:1:l -cache:dl2 ul2:1024:64:2:l

  Or, a fully unified cache hierarchy (il1 pointed at dl1):
    -cache:il1 dl1
    -cache:dl1 ul1:256:32:1:l -cache:dl2 ul2:1024:64:2:l
```