UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

GUILHERME DAL PIZZOL

Previsão de Desvios em Arquiteturas Multitarefas Simultâneas

Dissertação apresentada como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Dr. Philippe Olivier Alexandre Navaux
Orientador

Porto Alegre, agosto de 2005

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Dal Pizzol, Guilherme

Previsão de Desvios em Arquiteturas Multitarefas Simultâneas / Guilherme Dal Pizzol. – Porto Alegre: PPGC da UFRGS, 2005.

82 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2005. Orientador: Philippe Olivier Alexandre Navaux.

1. Arquiteturas de computadores. 2. Previsão de desvios. 3. Arquiteturas SMT. I. Navaux, Philippe Olivier Alexandre. II. Título.

“O que é útil? Para que e para quem algo é útil?”
“O que é o inútil? Por que e para quem algo é inútil?”

AGRADECIMENTOS

Em primeiro lugar gostaria de agradecer a todos que de alguma maneira colaboraram nessa longa etapa da minha vida, mesmo que não estejam citados explicitamente nesses parágrafos.

Ao meu pai e minha mãe, por não terem permitido sobre hipótese nenhuma que eu abandonasse o barco no meio da viagem (eles me entendem...).

A minha noiva Léa, que apareceu na minha vida durante o desenrolar do curso e que me apoiou muito nesses últimos meses. Obrigado por entender que nos fins-de-semana eu tinha que trabalhar na dissertação e que muitas vezes não consegui dar a devida atenção.

Também sou grato ao pessoal da TeleNova e TeleHUMANA, por prestar grandes momentos de discussão de assuntos variados (inclusive alguns ligados aos assuntos do mestrado). Além disso, gostaria de agradecer por permitir as minhas intermináveis idas a UFRGS, tanto para disciplinas quanto para reuniões do grupo.

Gostaria de agradecer ao pessoal do APSE: Rafael, Ronaldo, Tatiana, Laurino e todos outros que passaram por aqui nesses anos. Por todo o apoio, discussão, participações em vários SBACs e algumas champagnes em momentos especiais. Um especial agradecimento ao Pilla, por me "co-orientar" com seus conselhos experientes e contínuas revisões nos meus trabalhos.

Finalmente, ao Prof. Navaux, por todos esses anos de orientação (já são mais de 7 anos) e sábios (sempre sábios) conselhos. Sem o apoio dele esse trabalho não teria finalizado.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

BHR	*Branch History Register*
BHT	*Branch History Table*
BTB	*Branch Target Buffer*
CMP	*Chip Multiprocessor*
d-TLB	TLB de dados
i-cache	*Cache* de instruções
i-TLB	TLB de instruções
ILP	*Instruction Level Parallelism*
IPC	Instruções por ciclo
IQ	*Instruction Queue*
ISA	*Instruction Set Architecture*
FIFO	*First In First Out*
MIPS	Milhões de instruções por segundo
MT	Multitarefa
PISA	*Portable Instruction Set Architecture*
PC	*Program Counter*
PHT	*Pattern History Table*
RAS	*Return Address Stack*
RAW	*Read after Write*
RUU	*Register Update Unit*
RISC	*Reduced Instruction Set Computer*
SMP	*Symmetric Multi-Processor*
SMT	*Simultaneous Multithreaded*
TLP	*Thread Level Parallelism*
TLB	*Translation Lookaside Buffer*
WAR	*Write after Read*
WAW	*Write after Write*

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

A exploração do paralelismo no nível de instrução (ILP) em arquiteturas superescalares é limitada fortemente pelas dependências de controle, as quais são ocasionadas pelas instruções de desvio, e pelas dependências de dados. As arquiteturas SMT (*Simultaneous MultiThreaded*) buscam explorar um novo nível de paralelismo, denominado paralelismo no nível de tarefa (TLP), para buscar e executar instruções de diversas tarefas ao mesmo tempo. Com isso, enquanto uma tarefa está bloqueada por dependências de controle e de dados, outras tarefas podem continuar executando, mascarando assim as latências de previsões incorretas e de acessos à memória, usando mais eficientemente as unidades funcionais e demais recursos disponíveis.

Contudo, o projeto dessas arquiteturas continua a esbarrar nos mesmos problemas associados ao uso de técnicas utilizadas para a exploração de ILP, como a previsão de devios. Além disso, essas arquiteturas trazem novos desafios, como a determinação da maneira mais eficiente de distribuição/compartilhamento de recursos entre as tarefas.

Nesse trabalho será apresentada uma topologia para as tabelas de previsão de desvios em arquiteturas multitarefas simultâneas. Além disso, serão desenvolvidas duas análises complementares acerca de previsão de desvios: o impacto da taxa de acertos da previsão de desvios em arquiteturas com *pipelines* profundos e o impacto da taxa de acerto na previsão do alvo de um desvio.

Entre as principais contribuições do trabalho pode-se citar a definição de uma estrutura particionada para as tabelas de previsão em arquiteturas SMT, aliando desempenho a um menor custo de implementação em uma arquitetura real. Além disso, é mostrado que a taxa de acerto da previsão de desvios tem um grande impacto no desempenho das arquiteturas SMT com *pipelines* profundos, bem como nas causas de bloqueio do estágio de busca quando utiliza-se *cache* de instruções bloqueantes.

Palavras-chave: Arquiteturas de computadores, previsão de desvios, arquiteturas SMT.

Branch Prediction on Simultaneous MultiThreaded Architectures

ABSTRACT

The exploitation of ILP (Instruction Level Parallelism) in superscalar architectures is limited by data and control dependencies (caused by branch instructions). Simultaneous MultiThreaded (SMT) architectures can explore another level of parallelism, the so called TLP - Thread-level parallelism - and fetch and execute instructions from different tasks at the same time. While a task is blocked by control or data dependencies, other tasks may execute, masking latencies caused by mispredicted branches and memory accesses, and increasing the occupation of functional units and other available resources.

However, the design of SMT architectures keeps facing the same problems associated with techniques used for ILP exploitation, such as branch prediction. Moreover, SMT architectures also bring new challenges, such as determining the most efficient way to share resources among different threads.

In this work different branch prediction topologies for SMT architectures will be presented. Besides this, two analisys on branch prediction technique will be accomplished: the impact of branch prediction on deeply pipelined machines and the impact of target prediction on both SMT and superscalar architectures.

The main contribution of this work is to provide a clustered branch prediction structure for SMT architectures, which deliver a good performance and is easier to implement than a full distributed topology. Moreover, accurate branch prediction on these architectures is a must to speedup performance on the presence of deep pipelines, as well as to unblock fetch stage when blocking caches are being used.

Keywords: computer architecture, branch prediction, SMT architectures.

1 INTRODUÇÃO

Uma das primeiras formas empregadas para melhorar o desempenho dos computadores foi a técnica de divisão da execução das instruções em pequenos estágios, ao invés de utilizar uma execução monolítica. Com essa técnica, tornou-se possível executar com uma certa sobreposição várias instruções no mesmo ciclo do processador, uma vez que a finalização da execução em um estágio permite que uma nova instrução utilize o mesmo. Esta técnica foi chamada de *pipeline* e, apesar de ter surgido no final da década de 50 com as máquinas STRETCH (BLOCK, 1959) e LARC (ECKERT et al., 1959) , continua sendo utilizada nos processadores comerciais mais conhecidos.

O avanço no desenvolvimento do *software* fez com que os processadores também evoluissem. Novas gerações de *software* necessitavam cada vez mais poder de processamento. Dessa forma, o *pipeline* foi reestruturado e ampliado para o que atualmente é conhecido como arquitetura superescalar (JOHNSON, 1991). Estas arquiteturas possuem a capacidade de executar instruções com paralelismo simultâneo, além daquele com sobreposição parcial.

Os processadores superescalares são representados pela maioria dos processadores disponíveis no mercado, como o **Intel Pentium** (INTEL, 2001) e **AMD Athlon** (ADVANCED MICRO DEVICES, 2000). Para atingir o máximo desempenho, esses processadores exploram o paralelismo no nível de instrução (**ILP - *Instruction Level Parallelism***), utilizando para tal a execução fora de ordem, hierarquia de memória, previsão de desvios, entre outras técnicas. Todas essas técnicas fazem com que esses processadores sejam mais complexos, usem mais transistores e gastem cada vez mais energia e, conseqüentemente, dissipem mais calor.

Os desenvolvedores de *hardware* têm três opções para melhorar o desempenho desses processadores: aumentar a freqüência de funcionamento do mesmo, explorar mais ILP através de novas técnicas e usar *caches* cada vez mais rápidas e próximas ao processador. Técnicas para aumentar a freqüência de *clock* do processador envolvem, desde o uso de novas técnicas de fabricação, até o aumento no número de estágios do *pipeline*. Todas as técnicas esbarram em alguns problemas, como os limites físicos dos processos de fabricação e as quebras do fluxo de instruções nos *pipelines* (desvios, *misses*, etc). A extração de mais ILP envolve técnicas que aumentam o número de instruções executadas por ciclo. O emprego de *caches* maiores e mais próximas aos processadores é outra técnica que visa melhorar o desempenho desses processadores, atenuando a histórica diferença no tempo de ciclo do processador e da memória principal. O problema dessas técnicas é que elas implicam em maiores áreas de silício (maior número de transistores) e uma maior dissipação de energia. Além disso, o aumento de uma delas não provoca aumento proporcional da eficiência do sistema, por exemplo, dobrar o *clock* do um processador não resulta no dobro de desempenho do sistema.

Softwares que fazem uso de múltiplas *threads* ou múltiplos processos que podem ser executados em paralelo são comuns atualmente, principalmente os que rodam em aplicações de servidores. Esse nível de paralelismo é conhecido como paralelismo no nível de *thread* (**TLP - *Thread Level Parallelism***). A exploração do TLP é uma das formas de produzir sistemas com melhor relação custo/benefício.

As primeiras máquinas que tentavam explorar o TLP apareceram em meados de década de 60, os chamados multiprocessadores, como o UNIVAC 1108 e os processadores multitarefas, como por exemplo os processadores de entrada e saída da máquina CDC 6600. Os multiprocessadores consistem em vários processadores acoplados em uma única placa executando diversas tarefas simultaneamente. As máquinas multitarefas, por sua vez, aumentam a taxa de utilização dos recursos já disponíveis nos processadores, compartilhando recursos entre múltiplas tarefas e alternando a execução de instruções provindas de diferentes tarefas a cada novo ciclo de relógio.

Recentemente, no fim da década de 80, uma nova abordagem no projeto de arquiteturas multitarefas passou a executar simultaneamente instruções de diferentes tarefas em um mesmo ciclo de relógio, através do compartilhamento das unidades funcionais. Essas arquiteturas foram chamadas de **SMT (*Simultaneous MultiThreaded***) e combinam características de arquiteturas multitarefas e superescalares.

Dessa forma, as arquiteturas SMT podem tanto se beneficiar da busca, despacho e execução de múltiplas instruções por ciclo das modernas arquiteturas superescalares, assim como da habilidade de esconder longas latências das arquiteturas multitarefas. Por outro lado, se as arquiteturas SMT podem se beneficiar das características das arquiteturas superescalares, elas também acabam herdando limitações inerentes a essas arquiteturas. Um bom exemplo é a limitação de ILP disponível em única tarefa (arquiteturas superescalares) devido às dependências de controle. Para tentar superar essa limitação, uma técnica, que é amplamente utilizada nos processadores superescalares atuais, é a previsão de desvios.

Essas mesmas técnicas de previsão de desvios são utilizadas nas arquiteturas SMT, porém, pouco estudo foi desenvolvido na área de previsão de desvios em arquiteturas SMT. Em uma arquitetura SMT, o comportamento de diferentes previsores de desvios pode ser alterado devido à interferência que uma tarefa pode causar nas outras que estão sendo executadas simultaneamente.

Essa dissertação avaliará diferentes técnicas e implementações de previsores de desvios em arquiteturas SMT. Para tal, será proposto o uso de diferentes topologias de tabelas de previsão de desvios nessas arquiteturas, já que com mais de uma tarefa executando simultaneamente existe a possibilidade de compartilhar ou distribuir recursos a cada uma das tarefas.

O aumento no número de estágios dos *pipelines* utilizados em arquiteturas superescalares e SMT tem se tornado cada vez mais comum, seja com o objetivo de aumento de freqüência ou devido a estruturas em *hardware* que necessitam de mais ciclos para acesso. Nessas condições, a previsão de desvios passa a ter uma maior importância, pois um erro na previsão pode acarretar um grande desperdício de ciclos de relógio. Dessa forma, uma análise complementar acerca da previsão de desvios nessas arquiteturas será conduzida, com o objetivo de apontar diferentes comportamentos em arquiteturas superescalares e SMT. A influência de diferentes taxas de acerto do alvo de um desvio em arquiteturas SMT também será analisada, pois nessas arquiteturas um erro na previsão do alvo do desvio pode acabar subtraindo recursos necessários à execução de outras tarefas.

Essa dissertação está organizada em seis Capítulos. O Capítulo 2, irá tratar sobre

a organização e funcionamento das arquiteturas superescalares, dando enfoque especial à técnica de previsão de desvios. Logo após, no Capítulo 3, as arquiteturas multitarefas simultâneas (SMT) serão apresentadas, juntamente com um simulador desse tipo de arquitetura (*ss-smt*).

O Capítulo 4 contempla a proposta da dissertação, iniciando por uma revisão sobre estudos ligados aos efeitos de múltiplas tarefas nos previsores de desvios e sua ligação com arquiteturas SMT. Logo após, serão apresentados as motivações da dissertação, juntamente com a infra-estrutura utilizada e desenvolvida para a condução dos experimentos.

Os resultados experimentais obtidos ao longo da dissertação, bem como as discussões acerca desses resultados, serão apresentados no Capítulo 5. Finalmente, no Capítulo 6, são apresentadas as conclusões e contribuições dessa dissertação, além da proposta de trabalhos futuros.

2 ARQUITETURAS SUPERESCALARES

Para falar em arquiteturas superescalares é necessário, primeiramente, falar em *pipeline* ou em técnicas de *pipelining*. O *pipeline* foi inicialmente desenvolvido na década de 1950 e se tornou a principal característica dos computadores da década de 1960, tais como o CDC 6600 e o IBM 360/91, que apesar de não serem superescalares - executavam no máximo uma instrução por ciclo - apresentavam características dos processadores superescalares atuais, tais como várias unidades funcionais e escalonamento dinâmico de instruções (SMITH; SOHI, 1995).

Pipeline é uma técnica de implementação na qual múltiplas instruções são parcialmente sobrepostas na execução. O trabalho a ser feito no *pipeline*, por uma instrução, é quebrado em pequenas partes, onde cada uma delas consome uma fração do tempo necessário para executar o trabalho todo. Cada um destes passos é chamado estágio ou segmento do *pipeline*. Os estágios são justapostos para formar uma ”linha de montagem”, onde as instruções entram por um extremo, percorrem os estágios e saem pelo outro extremo, ou seja, a finalização de um estágio por uma instrução permite que uma nova instrução utilize aquele estágio. (HENNESSY; PATTERSON, 2003)

Um exemplo de como ocorre uma execução em uma arquitetura *pipeline* de 5 estágios pode ser visto na Figura 2.1, onde são mostradas as diferentes fases na execução de três instruções. Essa divisão em 5 estágios é clássica e foi implementada em diversos processadores RISC (*Reduced Instruction Set Computer*). Comparativamente a uma arquitetura que não implementa *pipeline*, tem-se um ganho potencial de 8 ciclos durante a execução daquelas três instruções.

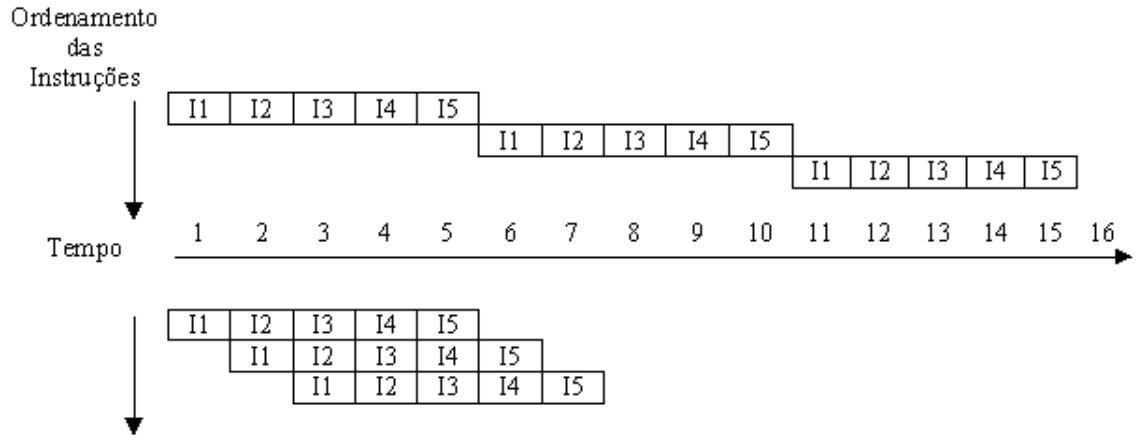

Figura 2.1: Execução de três instruções em modo seqüencial e em um *pipeline* de 5 estágios

O primeiro estágio é o de busca de instruções, o segundo realiza a decodificação das mesmas, o terceiro, por sua vez, é responsável pela execução das instruções. Caso a instrução necessite de acesso à memória, o mesmo é realizado no quarto estágio. Já no último estágio, é feita a atualização dos bancos de registradores.

A execução nesses *pipelines* pode ser quebrada em passos ainda menores, prolongando o *pipeline*, o que resulta na diminuição do tempo de ciclo e no aumento do *throughput*. O esforço no sentido de implementar *pipelines* com estágios cada vez menores deu origem às chamadas arquiteturas **superpipelined** (JOUPPI, 1988). Apesar das arquiteturas *pipeline* permitirem um melhor compartilhamento dos recursos do *hardware* entre diferentes instruções, muitos problemas surgem em função das características intrínsecas dos programas bem como das limitações do *hardware*.

O *pipeline* produz excelentes resultados enquanto nenhuma das instruções que está sendo executada precisa esperar por algum resultado de uma instrução precedente, ou quando não há uma alteração no fluxo de execução de instruções, como aqueles provocados por instruções de desvio. Estas situações, chamadas de conflitos do *pipeline* (*pipeline hazards*), fazem com que as instruções afetadas sejam bloqueadas até que o problema seja resolvido, podendo, ainda, causar o esvaziamento completo do *pipeline* em arquiteturas que são capazes de executar instruções especulativamente. Os principais conflitos que podem ocorrer em um *pipeline* são brevemente abordados abaixo (ROSE; NAVAUX, 2003):

Dependências de controle: as instruções que seguem uma instrução de desvio (*branch*, *jump*), tanto no caminho tomado quanto no não-tomado, são dependentes do controle da instrução de desvio, já que, enquanto a mesma não for resolvida (executada), não é possível saber se elas devem ou não ser executadas. Esse tipo de problema pode ser reduzido, por exemplo, através da previsão de desvios, que tenta prever qual fluxo de instruções deverá ser seguido após uma instrução de desvio. Esta técnica será mais detalhada na Seção 2.1.

Conflito de recursos: quando duas ou mais instruções necessitam de um mesmo recurso (compartilhado), diz-se que um conflito de recursos aconteceu. Por exemplo, se duas instruções necessitam de uma mesma unidade funcional, uma deve esperar o término da execução da outra. Uma forma simples de eliminar esse problema é através da adição de um número maior de recursos. Porém, essa solução pode esbarrar em limites físicos e financeiros.

Dependência de dados: Se o resultado de uma instrução é necessário para que instruções seguintes possam ser executadas e, essas são executadas antes que o resultado necessário esteja disponível, resultados incorretos podem ser gerados (Figura 2.2). Este tipo de dependência é chamado de dependência verdadeira ou RAW (***read-after-write***) (SMITH; SOHI, 1995), a qual pode ter seu efeito diminuído através de técnicas de adiantamento de dados (*data forwarding*): o resultado das instruções que já são conhecidos nos estágios mais avançados do *pipeline* - mas ainda estão esperando pela escrita no banco de registradores alvo - podem ser repassados antecipadamente para instruções que estão esperando esses resultados. Outra alternativa para resolver esse problema é o uso da técnica de reordenação de código, fazendo com que instruções que não serão afetadas por essa instrução sejam executadas logo após a mesma, servindo como um atraso na computação, até que a mesma seja executada. Outras dependências conhecidas como falsas dependências ou WAR (***write-after-read***) e WAW (***write-after-write***) acontecem quando uma

instrução tenta gravar o resultado (dado) após o mesmo ser lido ou escrito, respectivamente, por outra instrução. Essas falsas dependências podem ser eliminadas através de renomeação de registradores (JOHNSON, 1991). Entre outras técnicas que atacam o problema de dependências de dados, pode-se citar a previsão e o reuso de valores. (LIPASTI; SHEN, 1996; SODANI; SOHI, 1997; PILLA et al., 2004)

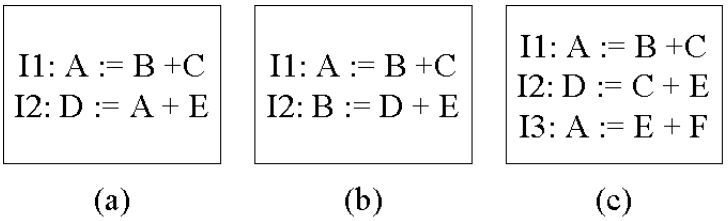

Figura 2.2: Dependências de dados – (a) RAW – (b) WAR – (c) WAW

Para definir-se uma arquitetura superescalar, não basta apenas apresentar o que é um *pipeline* e seus potenciais conflitos, é necessário também definir o que é escalar. O termo escalar é empregado para distinguir a execução de uma única instrução (operando sobre operandos discretos) daquelas instruções (vetoriais) que desencadeiam a ativação paralela de múltiplos elementos de processamento, todos executando a mesma operação. Existe outro modelo de processamento no qual as instruções que serão executadas concorrentemente não precisam possuir o mesmo código de operação, isto é, elas não são vetoriais, embora múltiplas instruções (escalares) possam estar sendo executadas simultaneamente. Por esse motivo, máquinas desse novo modelo de processamento são chamadas de superescalares (FERNANDES; SANTOS; WEBER, 1992).

Estas arquiteturas possuem a capacidade de executar instruções com paralelismo simultâneo, além daquele com sobreposição parcial. Essas máquinas superescalares começaram a surgir no final da década de 1980 e hoje são o padrão tecnológico adotado pela grande maioria dos fabricantes de microprocessadores. De uma forma geral, esses processadores apresentam as seguintes características (SMITH; SOHI, 1995):

- estratégias de busca de múltiplas instruções por ciclo, podendo antever/prever os desvios condicionais, assim como métodos para despachar essas múltiplas instruções;
- métodos para determinação e tratamento das dependências entre os dados dos registradores;
- recursos para execução paralela de múltiplas instruções, incluindo múltiplas unidades funcionais;
- métodos para comunicação de dados através da memória via instruções de *load/store*;
- métodos para recuperação do estado do processador na ordem correta.

É importante salientar que a presença apenas de múltiplas unidades funcionais não é suficiente para que essas máquinas possam explorar o paralelismo no nível de instrução. Além de permitir a execução simultânea de múltiplas instruções, é necessário que exista um mecanismo que, antecipadamente, verifique se existem dependências entre as diferentes instruções a serem executadas. Por exemplo, após a transferência de uma instrução

para uma unidade funcional, o processador não precisa aguardar sua conclusão para iniciar o tratamento das instruções posteriores que não sejam dependentes dos resultados sendo gerados pela instrução em execução. E, se não houver dependência de dados entre as próximas instruções, o algoritmo de despacho pode imediatamente despachá-la para uma outra unidade funcional, desde que esta esteja disponível (SANTOS, 1997).

Para viabilizar esse exame antecipado de instruções (*lookahead*), as arquiteturas superescalares possuem um conjunto de registradores ou *buffers* para armazenar as instruções que podem ser inspecionadas e despachadas para execução em determinado ciclo. Este conjunto de registradores é denominado **janela de instruções** e contém um trecho do programa em execução. A janela de instruções é utilizada, então, para desacoplar os estágios de busca e decodificação de instruções dos estágios de execução das mesmas.

A Figura 2.3 ilustra o funcionamento da janela de instruções e o modelo de execução usado na maioria dos processadores superescalares. O estágio de busca de instruções é o responsável por trazer as instruções da memória. Estas instruções são decodificadas no estágio de decodificação e, a partir desse estágio, são enviadas à janela de instruções para a futura inspeção de dependências entre as mesmas. Feita esta inspeção e dispondo de recursos para a execução, as instruções podem ser enviadas para as múltiplas unidades funcionais presentes. Esse envio pode não respeitar a ordem seqüencial original do programa. Nesse caso, diz-se que o despacho e a execução são fora de ordem. Finalmente, após a execução, as instruções são colocadas de volta na ordem original do programa, quando então elas são retiradas e seus resultados atualizam o estado da arquitetura. Esses estágios serão descritos em maiores detalhes a seguir.

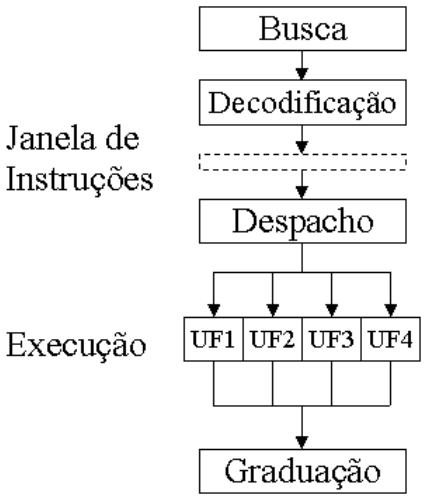

Figura 2.3: *Pipeline* Superescalar com janela de instruções

Busca de instruções

A fase de busca de instruções fornece instruções para o resto do *pipeline* da arquitetura. As instruções são buscadas através da hierarquia de memória – *caches* (atualmente até três níveis) e memória principal – e inseridas em uma fila de busca. Nesta fase também costuma ser realizada a previsão de desvios. Em arquiteturas do estado-da-arte, antes da fase de busca, existe uma fase de pré-busca (*pre-fetching*), que antecipa a busca de instruções de *caches* de níveis inferiores (*caches* nível 2 e 3) ou da memória principal (SANTOS, 2000).

Decodificação de instruções

As instruções que são inseridas na fila de busca são decodificadas posteriormente por este estágio e são inseridas na janela de instruções.

A existência da janela de instruções permite um desacoplamento dos estágios de busca e de decodificação dos demais estágios do *pipeline*. Assim, enquanto existirem entradas disponíveis na janela, esses estágios podem trabalhar independentemente dos demais.

A decodificação das instruções é feita na ordem seqüencial (do programa). Assim ao decodificar uma instrução o estágio de decodificação pode também alocar uma entrada no topo de uma fila especial, responsável por controlar o correto seqüenciamento das instruções nos estágios posteriores do *pipeline*, conhecida por fila de reordenamento (*reorder buffer*). Esta fila é implementada segundo o modelo FIFO (*first in first out*). Dessa maneira, quando uma instrução específica conclui sua execução, seu resultado é também colocado nesta mesma entrada que foi alocada, podendo ser escrito no conjunto de registradores. A fila de reordenamento pode ser preenchida também no estágio de despacho dependendo da lógica de inspeção de instruções escolhida.

Despacho de instruções

O estágio de despacho é responsável pela varredura da janela de instruções à procura de instruções a serem executadas, detectando e controlando as dependências verdadeiras (RAW), bem como eliminando as falsas dependências (WAW e WAR). Para tanto, utiliza-se métodos de renomeação de registradores e técnicas de escalonamento dinâmico das instruções, como o algoritmo de Tomasulo (TOMASULO, 1967; FERNANDES; SANTOS; WEBER, 1992; GONÇALVES, 2000) ou o algoritmo de *scoreboarding*.

O estágio de despacho pode, alternativamente, apenas despachar as instruções para filas de remessa juntas às unidades funcionais, para posterior remessa e execução. Neste caso diz-se que a janela de instruções é distribuída. Cada uma destas filas de reserva é conhecida pelo nome de estação de reserva (*reservation station*). Esta organização é utilizada, por exemplo, no algoritmo de Tomasulo. O estágio de despacho passa a ser dividido em, no mínimo, dois diferentes estágios: despacho das instruções para as estações de reserva e remessa para execução (*issue*).

A organização centralizada da janela de instruções, por sua vez, é implementada na *Register Update Unit - RUU*. Cada uma dessas organizações não será discutida nesse trabalho e pode ser encontrada em (JOHNSON, 1991).

Execução de Instruções

Havendo unidades funcionais disponíveis e estando as instruções prontas sem operandos não-resolvidos, as mesmas podem ser enviadas para execução (pelo estágio de despacho ou remessa, conforme acima). Já nas unidades funcionais, as instruções são executadas em um ou mais ciclos do processador – dependendo do tipo da instrução (inteiro, ponto-flutuante, desvio, memória) – podendo acessar a memória de dados. Essa execução também pode ser quebradas em etapas menores, através do uso de *pipelines* aritméticos.

Graduação da instrução

A fase final do ciclo de uma instrução é a graduação (*commit*), onde o resultado da instrução modifica o estado lógico do processador. A proposta deste estágio é garantir a ordem do modelo de execução seqüencial, ainda que a execução real seja especulativa e

fora de ordem.

Para tal, o resultado da execução de uma instrução é escrito na entrada específica da fila de reordenamento, a qual é utilizada para manter a ordem seqüencial do programa. É importante ressalvar, que dependendo do algoritmo utilizado para inspecionar as instruções nos estágios de despacho e da organização da janela de instruções, a fila de reordenamento pode estar difusa ou sua função pode ser implementada por outras estruturas de controle dos próprios algoritmos, como na RUU (SOHI, 1990).

2.1 Previsão de Desvios

O desempenho do *pipeline* só será máximo se não ocorrer o bloqueio da execução contínua dos diversos estágios de execução. As instruções de desvio provocam uma queda no desempenho desses processadores, já que bloqueiam a operação contínua do *pipeline* por não ser sempre possível conhecer o resultado do comando de desvio (tomado ou não-tomado) durante o estágio de busca.

Devido a grande taxa de ocorrência de instruções de desvio, várias técnicas foram desenvolvidas para reduzir o custo desse tipo de instrução. Entre elas podemos citar: técnicas de previsão de desvios, arquiteturas multifluxos (SANTOS, 1997) e execução predicada.

Segundo Hennessy e Patterson (2003), os desvios e as trocas de fluxos são classificados em quatro tipos:

- desvios condicionais;
- desvios incondicionais;
- chamada a procedimentos;
- retorno de procedimentos.

Um desvio incondicional, uma chamada de procedimento e um retorno de procedimento sempre alteram o fluxo de controle do programa. Quando o endereço alvo destes tipos de desvios faz parte da própria instrução e é conhecido durante a compilação do programa ou durante a carga do mesmo, o processador pode tratar esse tipo de desvio como um fluxo seqüencial de programa, exceto que o contador de programa (PC) é carregado com um novo endereço ao invés de ser incrementado. Por outro lado, quando o endereço alvo está em um registrador da arquitetura, o processador deve aguardar a execução do desvio para carregar o novo valor do PC.

Em um desvio condicional o processador deve fazer algumas avaliações antes de poder determinar o caminho certo de execução. A decisão normalmente deriva de um código de condição e resulta na seleção ou da instrução que está no endereço destino ou na próxima instrução seqüencial. O endereço tipicamente é conhecido durante o tempo de compilação.

Um exemplo de desvio condicional é o construtor ***if-then***, o qual é utilizado em quase todos programas. Analisando a freqüência relativa dos eventos de desvio em alguns *benchmarks* chega-se a conclusão que entre 65 e 80% dos desvios são condicionais e os incondicionais e desvios de procedimentos são em torno de 10 a 20% cada.

Considerando que essas dependências de controle afetam drasticamente o desempenho de processadores *pipeline*, foram desenvolvidas diversas técnicas de previsão de desvios, as quais permitem prever antecipadamente qual das ramificações no fluxo de controle será selecionada quando uma instrução de desvio for executada.

Usando essa previsão, as instruções pertencentes ao fluxo com maior probabilidade de execução podem ser buscadas, decodificadas e executadas antecipadamente. Porém, é necessário que o processador tenha mecanismos para desfazer eventuais computações provenientes de caminhos previstos erroneamente, o que acrescenta complexidade ao *hardware* resultante. Esses mecanismos afetam drasticamente o desempenho dos processadores superescalares, sendo que 10% de erros nestes mecanismos já diminuem sensivelmente a busca de instruções. Levando em conta o nível de implementação desses mecanismos, dois tipos de técnicas de redução do custo de desvios são encontrados: as implementadas em *software* e as em *hardware*.

2.1.1 Redução do custo dos desvios em *software*

Estas técnicas são empregadas durante a compilação do programa, ou seja, parte do problema é transferido para o nível de *software*. O *hardware* retarda a execução do comando de desvio, processando em paralelo uma ou mais instruções subseqüentes, sendo o *software* responsável por reorganizar as instruções do programa, indicando assim quais as instruções que serão executadas em paralelo com o comando de desvio. A seguir serão apresentadas algumas dessas técnicas: *delayed branch*, *branch folding*, *inlining* e o desenrolamento de laços.

Delayed Branch

Essa técnica de redução do custo de desvio consiste em reorganizar as instruções do programa, preservando a equivalência semântica dos programas e minimizando os retardos impostos pela ramificação. O compilador tenta movimentar as instruções do programa, alocando-as após o comando de desvio condicional, já que geralmente existe um retardo na execução desse tipo de desvio. Ao movimentar essas instruções minimiza-se o efeito da execução de instruções que poderiam ser descartadas e também o número de *NOPs* inseridos no código fonte.

Branch Folding

Algumas arquiteturas, no seu conjunto de instruções, incluem instruções específicas para configurar as condições de desvio (*flags* de condição) e trocar o fluxo de controle. Condições de desvio são tipicamente realizadas como o resultado de uma comparação específica entre dois registradores, ou como o resultado de uma operação aritmética. Neste caso, é possível que a instrução que resolva a condição em que o desvio está baseado tenha completado sua execução e o resultado do desvio seja conhecido quando o desvio é encontrado. Este ”desvio resolvido” pode ser removido do fluxo de instruções e substituído pelo seu sucessor lógico. Com a técnica de *branch folding* é alcançado o efeito de um desvio de ciclo zero (SANTOS, 1997).

Inlining

As técnicas de previsão de desvios mais simples apresentam uma reduzida taxa de acerto ao tratar instruções de retorno de funções, já que um procedimento pode ser chamado de diferentes pontos do programa e portanto, a técnica de previsão precisaria armazenar longos padrões de ativações e retornos para aumentar a taxa de acerto. As instruções geradas pelo compilador para a troca de parâmetros e transferir o controle entre o procedimento e o ambiente que o ativou, representa uma parcela não desprezível do tempo de processamento do programa. Este fato motivou o desenvolvimento de técnicas de otimiza-

ção de código como a *inline*, que consiste em substituir as *procedures* dos programas pelo código objeto correspondente nos locais onde as *procedures* são chamadas. O tempo de execução do programa com *procedures* codificadas é mais eficiente, pois as instruções de chamada e retorno se tornam redundantes e são retiradas, eliminado-se as instruções para passagem de parâmetro. A única desvantagem é que o código objeto tem seu tamanho aumentado.

Desenrolamento de laços

Esta técnica reduz o custo das instruções de desvio condicionais existentes no comando ***for***, através da replicação do código do corpo do laço e conseqüente eliminação de testes condicionais. Além disso, o mapeamento em registradores, quando possível, da variável de controle e do número de iterações elimina instruções de *load/store* produzindo uma significativa redução no tempo de processamento da estrutura *for*. Nos processadores superescalares da atualidade, a técnica de desenrolamento de *loops* é duplamente vantajosa: reduz o número de instruções condicionais executadas e aumenta o tamanho dos blocos básicos definidos no interior do novo corpo do comando *for*. Por outro lado, essa mesma técnica pode trazer impactos negativos no uso da *cache* de instruções, devido ao maior código objeto gerado.

2.1.2 Previsão de desvios em *hardware*

Diferentemente das técnicas implementadas por *software*, estas técnicas atuam durante a execução do programa e são implementadas pela unidade de controle do processador. Existem dois tipos de técnicas implementadas por *hardware*: as técnicas estáticas, onde a previsão ocorre baseada em definições feitas em tempo de projeto de um novo processador, e as técnicas dinâmicas, que como o próprio nome diz, realizam dinamicamente as previsões de desvio baseado nas informações coletadas em tempo de execução.

Previsão estática

Na previsão estática, o resultado previsto para um desvio é sempre o mesmo e pode ser computado no nível de *software*, durante a compilação, ou no nível de *hardware*, em tempo de execução. As previsões estáticas podem ser implementadas segundo três variações: o desvio sempre ocorrerá, nunca ocorrerá e o código da operação determina a previsão (SANTOS, 1997).

A primeira técnica explora o fato de que a maioria dos desvios condicionais provocam uma transferência no fluxo de controle, prevendo que a sucessora de um comando de desvio é a instrução contida no endereço alvo. Logo, o estágio de busca de instruções acessa o trecho de código alvo do desvio ao invés do trecho de código seqüencial. A segunda técnica, por sua vez, é o oposto da primeira, ou seja, o caminho tomado é o menos freqüente. Nesse caso, o estágio de busca continua trazendo instruções sequencialmente. É importante ressaltar que os compiladores podem organizar os comandos de teste dos programas para tirar proveito de uma das técnicas.

Na terceira técnica, ao invés de se fixar uma única direção, leva-se em conta o código da instrução de desvio para decidir se o fluxo de controle será transferido para o endereço alvo ou não, já que alguns desvios estão mais associados a um dos fluxos. Essa técnica utiliza resultados de estudos sobre o comportamento dos diversos tipos de desvios, levando em conta a probabilidade da ocorrência do salto no fluxo de controle. Com o objetivo de fixar a direção que será utilizada, o projetista monitora o comportamento

das instruções de desvio durante a simulação de programas representativos (*benchmarks*) e determina o número de vezes que cada tipo de comando provocou uma transferência de controle. Com isso, a previsão para cada tipo de desvios é fixada e armazenada, por exemplo, em uma memória ROM no interior do processador. Durante a execução do programa, o processador decide se o desvio deve ser tomado ou não, dependendo do código da operação.

Previsão dinâmica

Em alguns processadores, a unidade de controle realiza dinamicamente a previsão de desvios. Usualmente, essas técnicas são mais eficientes do que as estáticas. Técnicas dinâmicas armazenam informações coletadas das instruções de desvio em tempo de execução. Quando o desvio for novamente executado, o mecanismo de previsão verifica o que ocorreu no passado mais recente e, baseado nessa informação, prevê qual o resultado que será produzido pela instrução de desvio.

As informações indicando o que ocorreu recentemente com as instruções de desvio ficam armazenadas em pequenas tabelas ou registradores localizados no interior do processador. Devido ao custo, estas tabelas possuem um número limitado de entradas, o que faz com que nem todas instruções de desvio de um programa possam ser armazenados nela ao mesmo tempo. Para fazer a substituição e o gerenciamento dessas tabelas pode-se usar os mesmos algoritmos de substituição utilizados por sistemas operacionais na implementação de memória virtual.

Previsão determinada pela história do desvio

Essa técnica verifica o que ocorreu com as k mais recentes execuções de um desvio e realiza uma previsão do resultado que será produzido pela corrente execução do desvio. Os k mais recentes resultados de cada desvio ficam armazenados numa Tabela de História dos Desvios (BHT - *Branch History Table*) que é atualizada após a conclusão da instrução de desvio. Fisicamente, as entradas contendo a história dos desvios podem ser armazenadas num conjunto de registradores ou então numa memória *cache* no interior do processador.

Quando uma instrução é acessada, ela é pré-decodificada para que seja determinado se aquela é uma instrução de desvio. Se for o caso, os bits menos significativos do endereço do desvio são utilizados para indexar a BHT. Os bits na entrada selecionada fornecem a previsão do desvio. A instrução a ser acessada no próximo ciclo é determinada de acordo com esta indicação.

No estágio de execução, o estado da BHT é comparado com o resultado do desvio para verificar se a previsão estava correta. Neste caso, a execução prossegue normalmente e, caso contrário, as instruções buscadas antecipadamente são descartadas e a busca é redirecionada para o destino correto.

Um esquema bastante simples de previsão consiste em utilizar o resultado da última execução da instrução de desvio. Nesse caso, um bit seria suficiente para armazenar o resultado anterior da instrução de desvio. Se a previsão indicar que o desvio deve ser tomado e se o estágio de execução indicar o contrário, a tabela BHT é atualizada, as instruções nos estágios precedentes são descartadas e o estágio de busca inicia a transferência de instruções pertencentes ao fluxo apropriado. Se a instrução de desvio estiver sendo executada pela primeira vez, utiliza-se uma das técnicas estáticas apresentadas previamente e, em seguida, inclui-se o desvio na BHT. O autômato para esse mecanismo é muito simples e é mostrado na Figura 2.4.

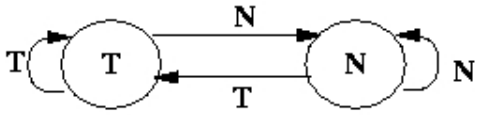

Figura 2.4: Autômato de 1 bit

O número de bits de história é um fator de extrema relevância na escolha do algoritmo de previsão. Acima foi mostrado um autômato para previsão com 1 bit de história. O maior problema dessa técnica é quando se faz necessário prever o destino de desvios de controle do laços, e o laço é executado mais de uma vez. Para n iterações de um *loop*, as primeiras $n-1$ iterações são tomadas e o desvio ao final do *loop* é previsto corretamente.

Entretanto, ao final da última iteração o desvio é não-tomado e a previsão é incorreta, uma vez que durante a última execução o desvio foi tomado. A próxima vez que o *loop* é executado, o desvio de controle é tomado e mais uma vez o algoritmo erra a previsão, pois da última vez o desvio fora não-tomado. Usando-se 1 bit de história é necessário uma previsão errada para que o algoritmo passe a prever a situação inversa. Se a previsão inicial é T e o resultado é não-tomado, o algoritmo passa a prever não-tomado na próxima execução.

Outro esquema que poderia ser utilizado é o mecanismo com 2 bits de história, assim é possível registrar o resultado das duas últimas execuções, e a próxima previsão é modificada apenas se as duas últimas previsões foram incorretas. A Figura 2.5 mostra o autômato utilizado para a previsão com 2 bits.

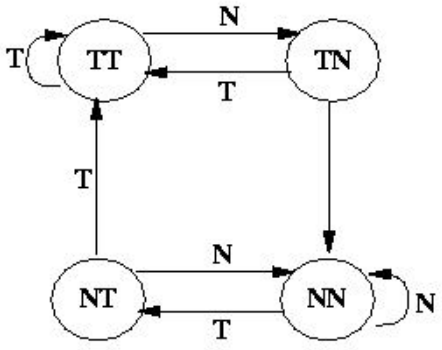

Figura 2.5: Autômato de 2 bits

Nos estados onde os dois bits são iguais (TT e NN), a previsão segue o resultado indicado por ambos. Nos estados onde os dois bits são diferentes, a previsão segue a indicação do bit que registra o estado mais antigo. Estudos realizados mostram que, com 2 bits de previsão, é possível alcançar uma taxa média de acerto individual de 90% (LEE; SMITH, 1984).

Previsão usando contador saturado

Nos experimentos descritos por Smith (SMITH, 1981a), a substituição dos bits de história por contadores (representados em complemento a dois) aumentou as percentagens de

acerto da técnica de previsão. Se o contador não for negativo, a técnica prevê que o desvio será tomado. Caso contrário, a previsão indicará que a instrução adjacente é a sucessora. Após a execução do comando de desvio, o contador será incrementado ou decrementado se o desvio for respectivamente, tomado ou não. A operação incrementar/decrementar é inabilitada quando um dos valores extremos do contador for atingido. Essa técnica produziu previsões mais precisas do que as técnicas baseadas nos bits de história descritas anteriormente. Smith também demonstrou que aumentando o limite de variação dos contadores não implica necessariamente em maiores percentagens de acerto: contadores com maior capacidade normalmente apresentam uma inércia maior para neutralizar o efeito provocado por uma outra instrução de desvio mapeada na mesma entrada.

*Previsão via tabela com alvos dos desvios (*Branch Target Buffer*)*

Uma técnica alternativa para previsão é a que emprega uma tabela contendo os alvos das instruções de desvios. Denominada BTB - *Branch Target Buffer*, essa tabela é uma evolução da tabela que contém a história dos desvios. Como anteriormente, a tabela BTB inclui campos para identificar a instrução de desvio e para armazenar a história das recentes execuções do comando de desvio (ou o contador saturado). Adicionalmente, a BTB inclui um campo contendo informações sobre a instrução sucessora do desvio: geralmente o campo armazena o endereço efetivo da sucessora; em outras implementações, a instrução sucessora também. A Figura 2.6 apresenta o diagrama de uma BTB.

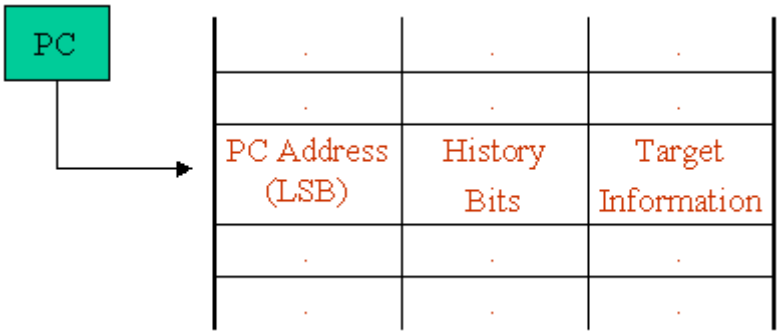

Figura 2.6: Estrutura da BTB

A BTB torna o processador mais eficiente do que aqueles que usam simplesmente uma BHT por causa do potencial oferecido pelas informações sobre a sucessora do desvio. Em paralelo com a busca da próxima instrução, podemos detectar antecipadamente se ela é um desvio e se esse for o caso, qual o endereço da instrução que deve suceder o comando de desvio que está sendo buscado. Através dessa antecipação, o estágio de busca pode ser prontamente redirecionado para o fluxo com maior probabilidade de execução, reduzindo desse modo, a incidência de instruções introduzidas indevidamente nos estágios iniciais do *pipeline*.

A BTB funciona da seguinte maneira, conforme ilustrado na Figura 2.6: o estágio de busca compara o endereço da instrução que está buscando com os endereços que estão na BTB. Se o endereço é encontrado na BTB então uma previsão é feita em função dos bits de história correspondentes. Se essa previsão diz que o desvio será tomado, então o endereço no campo de destino será usado para acessar a próxima instrução. Se o endereço não for encontrado na BTB, o estágio de busca deve continuar com o seu processamento normal. A BTB, nesse caso, selecionará uma nova entrada para conter o endereço dessa

instrução de desvio. Quando o desvio é resolvido, no estágio de execução, a BTB pode ser corrigida com a informação correta sobre o que aconteceu com o desvio, caso a previsão feita anteriormente tenha sido incorreta.

Para que a técnica de previsão de desvios seja realmente eficaz, é necessário que o mecnismo utilizado acerte a maioria das previsões. A taxa de acerto de um mecanimso depende de dois fatores (SANTOS, 1997):

- freqüência com que as informações de previsão são encontradas na tabela; (*hit ratio*)
- freqüência de acerto na previsão de cada desvio.

Logo, a taxa de acerto é dada por:

$$Ta = HitRatio \times FA$$

Aonde FA é a freqüência de acerto na previsão de desvios e $HitRatio$ é a freqüência com que as informações são encontradas na tabela. O primeiro depende em parte do número de bits de previsão, uma vez que com mais bits é possível registrar mais informações sobre o histórico do desvio. O segundo depende da configuração da tabela de previsão, ou seja, do número de entradas.

Previsão dinâmica em dois níveis

A idéia de coletar dinamicamente dois níveis de história de desvios foi proposta inicialmente por Yeh e Patt (1991; 1993) e por Pan, So e Rahmeh (1992). O primeiro nível armazena a história dos últimos k desvios encontrados. O segundo nível armazena o que aconteceu com as últimas j ocorrências de um padrão específico daqueles k desvios. O primeiro nível é denominado *Branch History Register* (BHR) e o segundo nível de *Pattern History Table* (PHT). Quando diversos BHRs são utilizados para manter o histórico em primeiro nível, tem-se uma *Branch History Table - BHT*.

O endereço de um desvio é mapeado para acessar o primeiro nível assim como em uma BTB convencional. Após mapear a entrada correta, o registrador de história (*Branch History Register*) fornece o padrão de bits que irá determinar qual entrada será acessada no segundo nível. Ao acessar o segundo nível, o mecanismo dispõe então do bit de previsão que indicará o caminho a ser seguido pelo estágio de busca para acessar as instruções seguintes, conforme mostra a Figura 2.7.

O segundo nível geralmente contém um contador saturado de 2 bits ou um autômato de 2 bits, conforme descrito anteriormente. A tabela de segundo nível deve conter uma entrada para cada histórico possível em primeiro nível, assim tem-se 2^k possibilidades de diferentes históricos e, conseqüentemente, 2^k entradas na PHT.

Dependendo da implementação de primeiro nível, a história dos últimos k desvios pode significar:

G – Esquemas globais – o primeiro nível guarda a história dos últimos k desvios encontrados, assim, apenas um BHR é utilizado. Isto significa que a previsão será influenciada não só pelo desvio em questão, mas por todos últimos k desvios;

P – Esquemas por endereço – o primeiro nível mantém as últimas k ocorrências de um mesmo desvio, assim, um BHR é associado a cada instrução de desvio. Dessa forma, somente o próprio desvio tem influência no resultado da previsão;

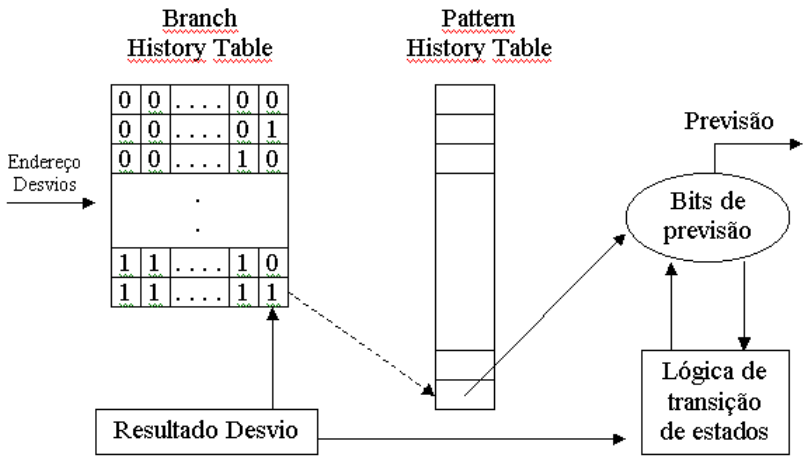

Figura 2.7: Previsão dinâmica utilizando dois níveis

S – Esquemas por conjunto – o primeiro nível guarda as últimas k ocorrências de um conjunto de desvios. Cada BHR está associado a um conjunto de desvios. Esse conjunto pode ser determinado pelo código do desvio, por decisão do compilador ou pelo próprio endereço dos desvios. Assim, somente os desvios de um mesmo conjunto influenciam a previsão naquele conjunto.

A mesma classificação pode ser adotada para as tabelas de segundo nível, ou seja, de que maneira os padrões serão associados à informação de histórico:

g – os últimos k desvios encontrados;

p – as últimas k ocorrências de um mesmo desvio;

s – as últimas k ocorrências de um conjunto de desvios.

Dessa forma, surgem três classes com nove diferentes variações de previsores de dois níveis, conforme resumido na Tabela 2.1.

Após a finalização da execução da instrução de desvio, o resultado (tomado, não-tomado) é deslocado para dentro do registrador de história da entrada correspondente no primeiro nível, modificando o padrão para os desvios que mapearão aquela entrada futuramente – simplesmente aplica-se um deslocamento à esquerda nos bits do histórico. A lógica de transição de estado avalia o resultado do desvio juntamente com a previsão feita anteriormente para este, e fornece o novo bit de previsão que será usado posteriormente.

Uma variação de previsor em dois níveis foi proposto em (MCFARLING, 1993), sendo chamado de **gshare**, o qual é uma variação do esquema de previsão global, conforme descrito anteriormente. O *gshare* aplica a operação XOR sobre o endereço do desvio e o BHR para utilizar como base para acesso às tabelas de segundo nível (PHT). Dessa forma, os estados de execução de um desvio podem ser mais facilmente detectados, pois o mesmo padrão de BHR pode estar associado a mais de um desvio em tempo de execução.

Previsores híbridos e multi-híbridos

Esses previsores apareceram a partir de 1993 nos projetos e atualmente são os que apresentam as melhores taxas de acerto, podendo atingir patamares superiores a 99% em

Tabela 2.1: Variações da previsão dinâmica em dois níveis

Variação	Descrição
GAg	Previsão Global Adaptativa usando uma PHT global
GAs	Previsão Global Adaptativa usando PHTs por conjunto de desvios
GAp	Previsão Global Adaptativa usando PHTs por desvios específicos
PAg	Previsão por desvio específico usando uma PHT global
PAs	Previsão por desvio específico usando PHTs por conjunto de desvios
PAp	Previsão por desvio específico usando PHTs por desvios específicos
SAg	Previsão por conjuntos de desvios usando uma PHT global
SAs	Previsão por conjuntos de desvios usando PHTs por conjunto de desvios
SAp	Previsão por conjuntos de desvios usando PHTs por desvios específicos

alguns *benchmarks*. Previsores híbridos incluem diversas técnicas, todas operando em paralelo, mas somente a técnica com maior probabilidade de acerto é a que fornece o resultado da previsão para a unidade de busca de instruções.

O previsor híbrido proposto por McFarling (1993) é formado por dois previsores simples que são controlados por um mecanismo que seleciona dinamicamente qual das duas previsões será considerada. Esse previsor híbrido usa uma tabela contendo contadores saturados de dois bits. Quando da chegada de um desvio, seu endereço indexa a tabela e dependendo do bit mais significativo do contador saturado, o mecanismo seleciona um dos previsores simples. Após a execução da instrução de desvio, o correspondente contador é incrementado, decrementado ou mantido, dependendo da correção dos dois previsores simples – se ambos estavam corretos ou ambos estavam errados, nenhuma mudança é feita nos contadores. Um previsor de desvios baseado nesse mecanismo foi empregado no microprocessador Alpha 21264 (KESSLER, 1999).

O previsor multi-híbrido proposto por Chang *et al.* (1994) emprega um número maior de componentes, e ao aumentar esse número, os autores verificam que previsões com maior índice de precisão são obtidas. Os componentes utilizam a classificação de tipos de desvios para determinar qual previsor pode determinar a melhor previsão.

Atualmente previsores híbridos continuam sendo simulados e avaliados, como por exemplo o *Cascaded Predictor* (DRIESEN; HOLZLE, 1998), o *Triton Predictor* (KARLINI; STEFANOVIC; FORREST, 2004) , o *Gshare/Perceptron* (JIMENEZ; LIN, 2001) e o *Prophet/Critic Predictor* (FALCON et al., 2004).

3 ARQUITETURAS MULTITAREFAS SIMULTÂNEAS - SMT

Como dito anteriormente, o *software* está cada vez mais utilizando múltiplos processos e tarefas (*threads*), surgindo então, novas possibilidades de extração de paralelismo. Surge, então, a possibilidade de extrair paralelismo *intra-thread* (ILP) e *inter-thread* (TLP). O TLP pode ser explorado por máquinas capazes de executar mais de uma *thread* (processo) simultaneamente. Atualmente, existem arquiteturas capazes de realizar essa tarefa: as máquinas multiprocessadas e as máquinas que executam mais de uma tarefa, simultaneamente (**SMT – *Simultaneous MultiThread***, **CMP – *Chip MultiProcessor***) ou não (**MT – *MultiThread***).

Máquinas multiprocessadas já são utilizadas há muito tempo para extrair maior desempenho das aplicações. Essas máquinas possuem diversos processadores (geralmente superescalares) interligados através de uma ”placa-mãe”. Várias tarefas podem ser executadas em cada um dos processadores, simultaneamente. Essas tarefas podem ser *threads* da mesma aplicação, de diferentes aplicações ou uma tarefa de sistema operacional. Uma vantagem desse tipo de máquina é o conhecimento das técnicas de programação que extraem alto desempenho pelos programadores. Sua principal desvantagem é o alto custo do *hardware*.

Historicamente, a idéia de execução de mais de uma tarefa surgiu dos sistemas operacionais multiprogramados (multitarefas), onde, virtualmente, várias tarefas estão executando simultaneamente. Para o usuário final, o processador está realmente executando mais de uma tarefa ao mesmo tempo. Nesse caso, o paralelismo é virtual e é obtido pelo sistema operacional através da intercalação das tarefas no tempo, ou seja, cada tarefa tem um tempo no qual realmente utiliza o processador. Essa quantidade de tempo é conhecida por *time-slice* (fatia de tempo) e é gerenciada pelo sistema operacional, o qual deve garantir que: a fatia de tempo seja pequena o suficiente para que o usuário não note nenhum tipo de degradação no sistema e, por outro lado, seja grande o suficiente para que o programa utilize o processador, realizando trabalho útil.

Outra técnica utilizada para extrair maior desempenho foi a de integração em *hardware* de muito dos recursos utilizados nesses sistemas operacionais multiprogramados. Assim, surge uma arquitetura intermediária entre multiprocessadores e processadores ”simples”, chamada de arquitetura multitarefas. O *hardware* das mesmas possui muitos recursos duplicados, necessários para manter o contexto das diversas tarefas, e outros compartilhados, tais como *caches* e unidades funcionais.

O *pipeline* desses processadores multitarefas pode, geralmente, conter em um certo estágio instruções de apenas uma tarefa. Dessa forma, as instruções de cada tarefa avançam pelo *pipeline* de forma ”sincronizada”. Na Figura 3.1 esse comportamento pode ser facilmente identificado. A memória princial contém quatro tarefas aguardando execução, porém, a cada ciclo instruções de apenas uma das tarefas podem ser buscadas. O *front-*

end do *pipeline* pode despachar cinco instruções por ciclo para qualquer uma das cinco unidades funcionais. Contudo, essas cinco instruções devem ser de uma mesma tarefa. Logo, cada tarefa ativa está confinada a uma ”fatia de tempo”, nesse caso, um ciclo de relógio. Assim, o *front-end* contém múltiplas tarefas em execução e a lógica de despacho alterna entre as mesmas a cada ciclo ao enviar as instruções para execução.

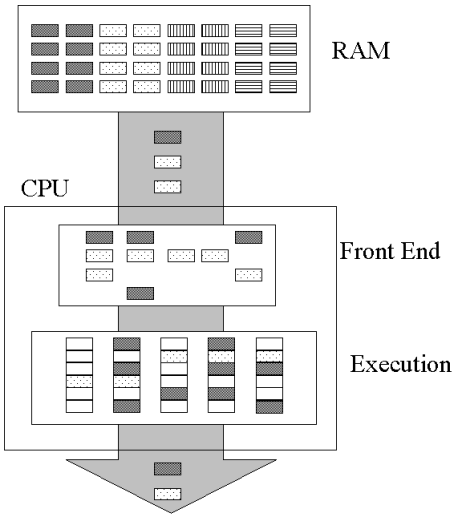

Figura 3.1: Arquitetura Multitarefa

Essas arquiteturas, além de executar o chaveamento de contexto entre as tarefas em tempos definidos, podem chavear na execução de instruções de longa latência. Dessa forma, se uma tarefa requisita um dado que não está presente em *cache* e tem de aguardar muitos ciclos, o processador pode executar instruções de outra tarefa, mantendo o *pipeline* cheio e realizando trabalho em um ciclo que seria desperdiçado. Apesar de ajudar a esconder as altas latências de algumas instruções (como *loads* e *stores*), esse tipo de arquitetura não explora a falta de ILP em uma tarefa individual. Como pode ser visto na Figura 3.1, se o escalonador, em determinado ciclo, encontrar somente duas instruções prontas para executar, três outros *slots* de despacho serão desperdiçados. Esse desperdício tende a ser maior em arquiteturas modernas, onde a largura de despacho pode ser de até oito instruções por ciclo (TENDLER et al., 2001).

Como exemplos desse tipo de arquitetura destacam-se: HEP (SMITH, 1981b), Tera (ALVERSON et al., 1990) e MTA (SNAVELY et al., 1998).

O desperdício atacado por essas arquiteturas (multitarefas) é conhecido como desperdício vertical (*vertical waste*), e é geralmente causado por uma instrução de longa latência. Por sua vez, o desperdício de quando o escalonador não consegue utilizar a largura máxima de despacho é chamado de desperdício horizontal (*horizontal waste*), conforme mostra a Figura 3.2. O desperdício horizontal é causado pela execução superescalar, mais especificamente, quando o ILP disponível é pequeno. Já em uma arquitetura escalar, a largura de despacho é sempre de uma única instrução (não há desperdício horizontal). Além disso, a execução superescalar aumenta o desperdício vertical (TULLSEN; EGGERS; LEVY, 1995). É importante ressalvar que em arquiteturas escalares só existe desperdício vertical.

Uma arquitetura mais agressiva, capaz de atacar ambos problemas era necessária, já que as arquiteturas multitarefas minimizam apenas o desperdício vertical. Com o surgimento de novas tecnologias que perimitiram o uso de um maior número de transistores em

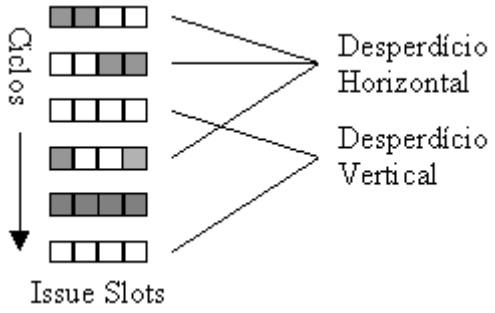

Figura 3.2: Desperdício Vertical x Horizontal

um mesmo *chip*, foi possível implementar uma arquitetura multitarefa capaz de buscar, despachar e executar instruções de múltiplas tarefas em um mesmo ciclo.

Essa arquitetura foi chamada de multitarefas simultâneas e já é utilizada em processadores comerciais, como o Intel Hyper-Threading (MARR et al., 2002) e IBM Power PC4 (TENDLER et al., 2001).

A técnica ou arquitetura multitarefas simultâneas (***Simultaneous MultiThread – SMT***), permite, basicamente, que tarefas independentes enviem instruções para executar em múltiplas unidades funcionais, as quais são compartilhadas.

Essa técnica apareceu pela primeira vez em 1992, quando Hirata (1992) propôs uma arquitetura capaz de despachar instruções de diferentes tarefas para múltiplas unidades funcionais, chamada de *Simultaneous Instruction Issuing*. Já Yamamoto (1994) propôs outra arquitetura capaz de executar instruções provenientes de diferentes fluxos simultaneamente em um mesmo ciclo. Essa arquitetura foi chamada de *Multistreamed Superscalar*. Por sua vez, Tullsen (1995) propôs uma técnica bastante semelhante, a qual foi denominada *Simultaneous Multithreading* (SM ou SMT). Desde então, esse tipo de arquitetura (técnica) passou a ser conhecido pelo acrônimo SMT.

O objetivo dessa arquitetura é aumentar a utilização das múltiplas unidades funcionais que já estão disponíveis em uma arquitetura superescalar, uma vez que essas unidades funcionais podem ficar ociosas em até 80% dos ciclos (TULLSEN; EGGERS; LEVY, 1995). Esse aumento é conseguido através da capacidade da arquitetura de executar instruções de múltiplas tarefas, escondendo a alta latência de certas instruções e indo além do limite do paralelismo existente em uma única tarefa.

Assim, a técnica SMT combina características de uma arquitetura superescalar e de uma arquitetura multitarefa. Ela une a capacidade de despacho de múltiplas instruções das arquiteturas superescalares com a habilidade de esconder as altas latências das arquiteturas multitarefas.

Para contemplar essas características, essas arquiteturas devem ser capazes de armazenar tantos contextos quanto o número máximo de tarefas simultâneas. Para isso,algumas estruturas devem ser replicadas, outras particionadas e também compartilhadas entre os diferentes contextos. De uma forma geral, como mostra a Figura 3.3, a memória (principal e *cache*) é compartilhada, assim como as unidades funcionais. As outras estruturas (filas, *buffers* e o banco de registradores, por exemplo) são ou particionadas ou replicadas.

Resumidamente, a grande diferença entre uma arquitetura SMT e uma multitarefa (Figura 3.1) é a possibilidade de, no mesmo ciclo de relógio, despachar e executar instruções de diferentes fluxos. Já a diferença entre ambas e uma superescalar é a possibilidade de existir ”realmente” mais de uma tarefa executando no processador, conforme mostra a Figura 3.4. Na arquitetura superescalar, caso não existam instruções disponíveis, alguns

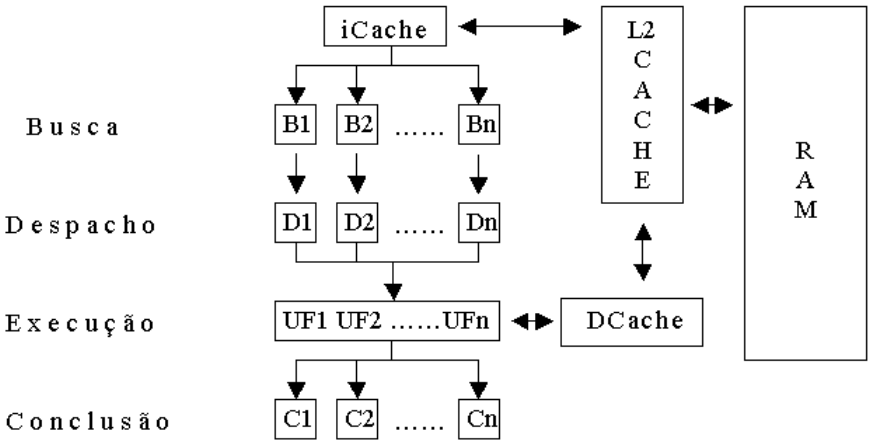

Figura 3.3: *Pipeline* simplificado de uma arquitetura SMT com 4 estágios

slots ficarão sem nenhum uso, enquanto a arquitetura SMT pode preencher esses *slots* vazios com instruções provenientes de outras tarefas. A cada ciclo, o processador SMT seleciona instruções de todas tarefas ativas para execução (no caso, duas tarefas). Outras tarefas podem estar em memória (no caso, quatro tarefas), esperando o escalonador do sistema operacional torná-las ativas. Ele explora o ILP ao selecionar instruções de qualquer tarefa que, potencialmente, pode despachar instruções. Logo após, o processador arranja os recursos de execução dinamicamente entre as instruções, fazendo com que haja uma maior utilização do *hardware* disponível. Se uma tarefa possui muito ILP disponível, o mesmo será utilizado; se várias tarefas têm pouco ILP, elas executarão em paralelo, compensando o baixo ILP. Diz-se então que o TLP está sendo explorado. Dessa maneira, como dito anteriormente, uma arquitetura SMT pode amenizar os problemas de desperdício horizontal e vertical.

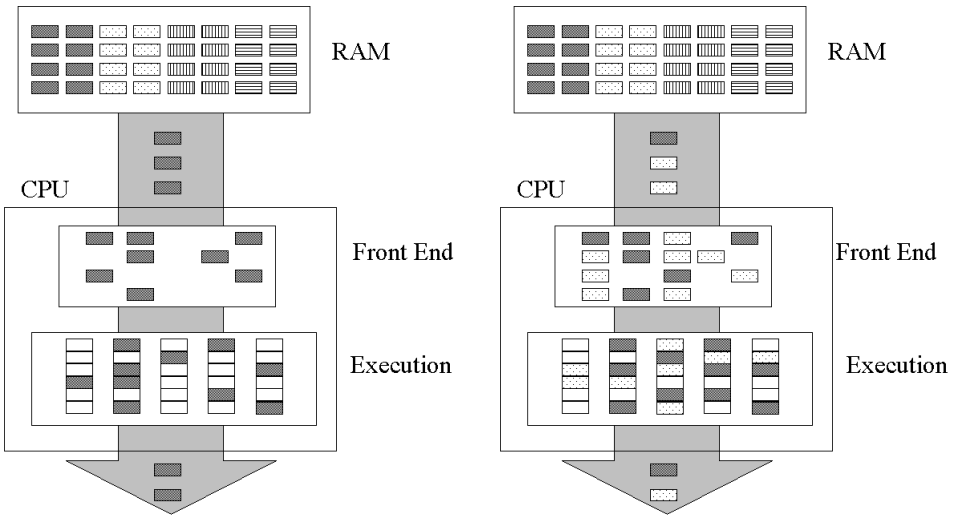

Figura 3.4: Arquitetura Superescalar vs. Multitarefas Simultâneas

Uma arquitetura SMT é uma extensão direta das arquiteturas superescalares. A sua implementação adiciona pouca quantidade de *hardware*. Assim, os projetistas podem manter o foco em uma arquitetura convencional de alto desempenho e adicionar a capa-

cidade de SMT no topo da mesma (EGGERS et al., 1997).

A arquitetura SMT proposta por Tullsen (1995) é uma extensão direta de um processador superescalar típico, similar ao MIPS R10000 (MIPS Technologies Inc., 1995). Esse processador é capaz de buscar até oito instruções da *cache* de instruções. Alguns recursos do processador base foram replicados para suportar o estado de mais de uma tarefa: registradores e contadores de programa, além das estruturas responsáveis pelo esvaziamento do *pipeline*, retirada de instruções, interrupções e retorno de sub-rotinas. Identificadores de tarefa também foram adicionados para a BTB e para as TLBs. Somente dois componentes tiveram de ser redesenhados, o *pipeline* e a unidade de busca.

O *hardware* disponível em arquiteturas superescalares do estado-da-arte já é capaz de despachar instruções de diferentes tarefas para as unidades funcionais, nenhuma alteração se faz necessária para adicionar esse suporte (EGGERS et al., 1997). A renomeação de registradores elimina conflitos inter e intra-tarefa fazendo o mapeamento de registradores específicos de uma tarefa no conjunto de registradores em *hardware*. Feito isso (e quando a instrução não possui outras dependências), o processador pode enviar as instruções sem necessidade de identificar a qual tarefa a mesma pertence.

Essas alterações sempre levam em conta que: o desempenho da arquitetura SMT com apenas uma tarefa deve ser igual ao do processador base e que a transição de uma arquitetura convencional para a mesma deve ser transparente ao usuário final.

Contudo, por ser uma extensão da arquitetura superescalar, a arquitetura SMT acaba herdando alguns dos problemas e deficiências da mesma, como por exemplo, a checagem de dependências entre as instruções, o número de portas necessárias no banco de registradores e a lógica necessária para *forwarding* no *pipeline*.

A seguir será feita uma breve análise do *hardware* da arquitetura proposta por Tullsen, ressaltando as diferenças em relação a uma arquitetura superescalar convencional.

O estágio de busca em um processador convencional tem por função principal buscar instruções da memória para alimentar os estágios de despacho e execução. Para obter desempenho máximo é necessário encontrar um fluxo contínuo de instruções, empregando para tal pré-busca de instruções (CHEN; BAER, 1992), *trace-cache* (ROTENBERG et al., 1997; PATEL; FRIENDLY; PATT, 1999), além da previsão de desvios. Em uma arquitetura SMT existe uma maior pressão sobre esse estágio, o qual necessita fornecer mais instruções para o escalonador, uma vez que o mesmo despacha instruções de diferentes fluxos.

Ao mesmo tempo em que pressiona o estágio de busca, tornando-o um possível gargalo, a arquitetura SMT provê uma maior facilidade para o mesmo encontrar instruções, já que estas podem vir de mais de uma tarefa. Assim, a largura de banda pode ser compartilhada entre as tarefas ativas, especialmente se a largura de banda for de oito ou mais instruções. É sabido que, em média, a cada quatro ou cinco instruções existe um desvio (condicional ou não) podendo quebrar o fluxo contínuo de instruções, logo, nessas arquiteturas a chance de preencher todas entradas das filas de busca é maior que em uma arquitetura que só pode acessar uma única tarefa. Além disso, adicionando mais inteligência a esse estágio é possível selecionar qual das tarefas tem mais chance de extrair um desempenho maior dos estágios subseqüentes do *pipeline* (UNGERER; SIGMUND, 1996).

O estágio de busca proposto por Tullsen continha oito PCs para endereçar no máximo oito tarefas simultâneas. A cada ciclo, uma das tarefas não bloqueadas poderia fornecer instruções. Ao permitir que apenas um PC fornecesse instruções por ciclo esse estágio provia uma busca de grão-fino. Posteriormente, Tullsen, (1996) analisou diferentes

configurações da unidade de busca para obter uma configuração que fosse eficiente (particionamento entre as tarefas), efetiva (qualidade das instruções buscadas) e disponível (eliminação de condições que bloqueiam a busca).

O estudo apontou que o esquema chamado 2.8 atinge um desempenho máximo médio de 10% sobre o que busca 8 instruções de apenas uma tarefa (1.8). Outros esquemas foram avaliados, como o 2.4 e 4.2 (instruções.tarefa). No esquema 2.8 tem-se oito PCs, com a diferença que, em cada ciclo, duas tarefas não bloqueadas são escolhidas pela lógica de busca e 8 instruções de cada tarefa são acessadas.

Devido a menor largura de despacho da arquitetura de Tullsen, o estágio de busca deve escolher um sub-conjunto para o envio ao próximo estágio. Ele preenche as entradas da fila de busca a partir da primeira tarefa até encontrar uma instrução de desvio ou até o fim de uma linha da *cache* e então o resto da largura é preenchido com instruções da outra tarefa.

Outra importante conclusão é relativa a quais tarefas serão selecionadas para prover as instruções, já que o estágio de busca pode escolher de quais tarefas instruções serão buscadas. Nem todas tarefas provem instruções de igual qualidade a cada ciclo. Se o processador poder prever quais tarefas irão produzir menores atrasos, o desempenho pode ser aumentado. Basicamente, dois fatores determinam essa maximização: a probabilidade de uma tarefa seguir pelo caminho errado como resultado de um erro prévio de previsão de desvio (consumindo largura de banda e recursos desnecessariamente); e a quantidade de tempo que as instruções ficarão na fila antes de poderem ser despachadas para execução.

Entre os esquemas analisados estão:

BRCOUNT: dá a preferência às tarefas com menor probabilidade de seguirem um caminho errado (contagem de desvios nos estágios seguintes);

MISSCOUNT: dá a preferência às tarefas com menor probabilidade de ficarem muito tempo na fila de despacho (número de *d-cache misses*);

IQPOSN: dá a menor prioridade para as tarefas que possuem instruções na ponta das filas de despacho para as unidades de inteiros ou de ponto-flutuante;

ICOUNT: dá a maior prioridade para as tarefas que possuem menos instruções nos estágios de decodificação, renomeação e filas de busca.

Tullsen mostrou que o esquema ICOUNT é o que provê a busca de maneira mais eficiente. Essa técnica prioriza as tarefas que estão movendo instruções mais eficientemente pelas filas de busca, evitando que as mesmas encham as filas com instruções dependentes de outras de longa latência. Ela também mantém na fila uma distribuição mais justa de instruções, aumentando a quantidade de paralelismo *inter-thread*. Além disso, ela evita o bloqueio de tarefas, pois as tarefas cujas instruções não estão executando irão, eventualmente, ter menos instruções no *pipeline* e serão escolhidas para a busca.

O último estudo baseou-se em encontrar uma técnica que diminuísse a ocorrência de *misses* na *i-cache*, chamada de ITAG e outra que diminuísse o número de ciclos em que a fila de busca estivesse cheia (IQ), chamada de BIGQ. A técnica ITAG faz uma procura na *i-cache* em um ciclo prévio para definir quais tarefas não estarão bloqueadas. Para implementar essa técnica um estágio a mais deve ser adicionado ao *pipeline*, aumentando o número de ciclos necessários para a recuperação de uma previsão de desvio errada. Já a técnica BIGQ, aumenta o número de instruções na IQ, mantendo a profundidade de procura (o tempo de procura é que determina a complexidade e tamanho da IQ). Essa técnica só apresenta uma melhoria de desempenho quando aplicada sem a técnica ICOUNT.

Um dos maiores impactos de uma arquitetura SMT sobre a arquitetura superescalar de base é o tamanho do conjunto de registradores. Cada registrador lógico de uma tarefa é renomeado para um registrador físico compartilhado através do estágio de renomeação de registradores. Para suportar n tarefas em uma arquitetura com um conjunto de instruções de 32 registradores são necessários $n \times 32$ registradores físicos, além de mais outros para habilitar a renomeação. Por exemplo, para uma arquitetura que suporta até oito tarefas, são necessários no mínimo $8 \times 32 = 256$ registradores. Conseqüentemente, o acesso a esse banco de registradores tende a se tornar lento, afetando o desempenho (tempo de relógio) da arquitetura.

Para sobrepor esse problema, Tullsen propôs a extensão do *pipeline* da arquitetura base em dois estágios, um extra para leitura de registradores e outro para a escrita, conforme mostra a Figura 3.5. Durante o primeiro estágio de leitura, os dados são trazidos para um *buffer* mais próximo das unidades funcionais e a instrução para um *buffer* similar ao mesmo tempo. No próximo ciclo, os dados são mandados para a execução. Os estágios de escrita funcionam dessa mesma maneira.

Figura 3.5: *Pipeline* da Arquitetura SMT de Tullsen

Essa adição de estágios causa algumas modificações no *pipeline*. Ela aumenta a distância entre a busca e a execução, adicionando uma maior penalidade (um ciclo) no caso de previsões erradas de desvio. A escrita de resultados fica postergada em um estágio, necessitando um nível extra de lógica de *bypass*. Ao aumentar a distância entre o despacho e a execução, o período em que uma instrução está sendo executada especulativamente também é aumentado (após a descoberta do erro de especulação).

Finalmente, com a existência desses estágios extras entre a renomeação e a graduação, o tempo mínimo que um registrador físico fica alocado por uma instrução também aumenta. Isso aumenta a pressão sobre a banco de registradores de renomeação, o qual também limita o número de instruções que podem estar executando em um certo ciclo de relógio. Tullsen conclui que o tempo de acesso a esse banco de registradores é o grande fator de limitação no número de tarefas que um processador SMT pode executar simultaneamente.

Com a maior quantidade de instruções buscadas por ciclo, outro aspecto que deve ser analisado é a capacidade do processador de despachar as instruções para execução. Da mesma forma que a busca pode selecionar de quais tarefas buscar instruções, o despacho pode escolher quais deverão ser executadas. Logo, em uma arquitetura SMT, a lógica de despacho tem mais variedade de instruções a escolher. Tullsen analisou quatro alternativas de políticas de despacho para prevenir a perda de ciclos do mesmo.

Em um processador superescalar, a escolha de instruções que não estejam sendo especuladas pode ser mais facilmente atingível escolhendo-se as que estão a mais tempo na fila (início da fila). Já na arquitetura SMT isso já não é possível, pois diferentes posições das filas podem conter tanto instruções que são e que não são especulativas. A primeira alternativa, chamada de *OLDEST_FIRST* é simplesmente a do processador superescalar. A segunda e terceira, chamadas de *OPT_LAST* e *SPEC_LAT*, que, respectivamente, enviam apenas instruções otimistas (aquelas que devem ficar na IQ por um ciclo extra após serem despachadas, até que seja sabido que as mesmas não serão descartadas) e especulativas

após todas outras serem despachadas. A última, chamada *BRANCH_FIRST*, envia desvios tão logo quanto possível para detectar erros na previsão mais cedo. Tullsen chegou a conclusão que o mecanismo mais simples (*OLDEST_FIRST*) seria o mais apropriado (o despacho não era um gargalo), pois todos outros adicionavam lógica de múltiplos passos na procura da IQ, aumentando a complexidade do processador, sem a devida relação custo/benefício.

A fase de execução do *pipeline* não apresenta grandes modificações em relação ao superescalar. Instruções de qualquer tarefa podem ser despachadas para a execução em qualquer unidade funcional desde que seus operandos estejam prontos. Ainda existe a possibilidade de realizar um particionamento das unidades funcionais, assim cada contexto de *hardware* pode estar conectado a exatamente uma unidade funcional de cada tipo. De qualquer forma, o escalonamento das instruções nas unidades funcionais é mais complexo em qualquer implementação de SMT. Uma importante diferença que pode haver nos estágios de execução dessas arquiteturas é a criação de uma unidade funcional específica para o tratamento de instruções de criação, destruição e sincronização de *threads*, além de comunicação entre as mesmas. A graduação das instruções também é particionada por tarefa. Assim, a execução é fora de ordem e a retirada é em ordem e por tarefa.

Para a implementação (simulação) dessa arquitetura Tullsen utilizou um *hardware* que continha:

- Largura de 8 instruções na busca e despacho e graduação de até 12 instruções por ciclo;
- Seis unidades funcionais de inteiros (4 são capazes de executar *load e store*);
- Quatro unidades de ponto-flutuante;
- 32 entradas nas filas de despacho;
- 256 registradores físicos ($8\ contextos \times 32\ registradores$), 200 registradores para renomeação (100 para inteiros e 100 para ponto-flutuante);
- *caches* L1 de dados e instruções de 128 Kbytes, *two-way*, com blocos de 64 bytes. *d-cache* tem quatro bancos de duas portas e a *i-cache* tem oito bancos de uma porta; o tempo de acesso por banco é de dois ciclos;
- L2 unificada de 16 Mbytes, diretamente mapeada, acesso de 12 ciclos em um barramento de 256 bits;
- Latência de acesso à memória principal de 80 ciclos em um barramento de 128 bits;
- Previsão de desvios híbrida com tabelas globais e locais. A tabela global possui 13 bits de história e a local uma tabela de 2048 entradas que indexa uma tabela de previsão de 4096 entradas. A BTB possui 256 entradas, *four-way*, com um identificador adicional de tarefa.

3.1 Simulador *ss-smt*

A ferramenta SimpleScalar (*SimpleScalar Tool Set*) (BURGER; AUSTIN, 1997) foi desenvolvida na Universidade de Wisconsin-Madison e é largamente utilizada em centros de pesquisas de arquiteturas de computadores ao redor do mundo.

Essa ferramenta implementa a arquitetura SimpleScalar, que é muito semelhante à arquitetura MIPS (MIPS Technologies Inc., 1995). A ferramenta possui duas versões, uma *big-endian* e outra *little-endian*, para manter a portabilidade da mesma entre diversos sistemas. A semântica do conjunto de instruções (ISA) é um superconjunto da ISA do MIPS-IV, sendo chamada de PISA (*Portable Instruction Set Architecture*).

As principais diferenças entre as duas são: cada instrução possui tamanho fixo e igual a 64 bits, além de três formatos: registrador, usado para instruções de computação de um modo geral; imediato, que suporta a inclusão de uma constante de 16 bits; e, salto, que suporta a especificação de um endereço alvo de 24 bits para uma instrução de *jump*.

Dentre os simuladores que acompanham a ferramenta, destaca-se o **sim-outorder**, o qual implementa um detalhado processador incluindo simulação de ciclos. Ele suporta execução fora de ordem, baseado na unidade de atualização de registradores (ou RUU - *Register Update Unit*) (SOHI, 1990). Esse esquema usa um *buffer* de reordenamento para renomear os registradores e manipular os resultados de instruções pendentes.

O sistema de memória do processador utiliza uma fila de *load* e *store*. Os valores que devem ser armazenados são dispostos em uma fila, caso essa operação seja especulativa. Além disso, as instruções de carga são despachadas para o sistema de memória quando os endereços de todas as instruções anteriores de armazenamento são conhecidos. A Figura 3.6 apresenta o *pipeline* simplificado do simulador *sim-outorder*. Cada estágio funciona exatamente como em uma arquitetura superescalar.

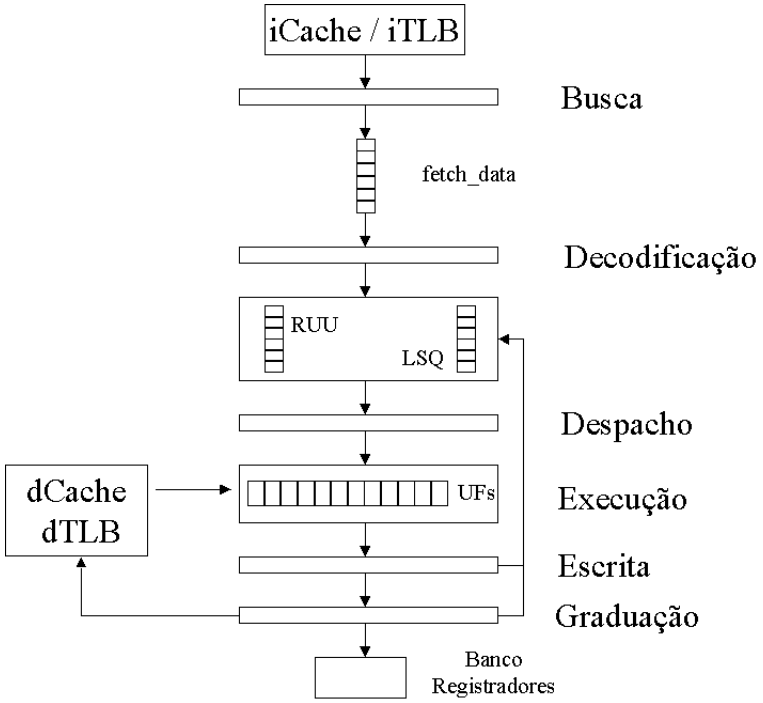

Figura 3.6: *Pipeline sim-outorder*

O estágio de busca de instruções é responsável também pela previsão de desvios, e, se algum desvio é detectado e previsto como tomado, a busca é bloqueada e redirecionada, de acordo com o endereço alvo. Essa previsão pode ser feita através de várias técnicas, entre elas, previsão de dois níveis, previsão combinada, previsão sempre tomada/não-tomada. No caso de ocorrência de falha na *cache* de instruções, a unidade de busca é bloqueada até que essa seja resolvida. Os endereços virtuais das instruções e dos dados são mapeados para endereços reais através das tabelas de tradução específicas, i-TLB e

d-TLB, respectivamente. Ao fim desse estágio, as instruções buscadas são posicionadas em uma fila de instruções. Durante o estágio de despacho, são realizadas a decodificação das instruções e a renomeação de registradores. A cada ciclo, o estágio de despacho encaminha um determinado número de instruções para a fila de escalonamento.

O estágio de delegação é realizado através de consultas a fila de escalonamento: a cada ciclo, as instruções cujos operandos já se encontram disponíveis são encaminhadas para as unidades funcionais especializadas. As instruções podem ser delegadas fora da ordem original do programa. Na execução, as instruções são executadas de acordo com a disponibilidade de cada tipo de unidade funcional, as quais ficam ocupadas pelo tempo de latência da instrução.

A escrita de resultados verifica as instruções que estão prontas para graduação. Quando uma instrução é terminada esse estágio verifica na fila eventuais dependências existentes e caso não haja nenhuma, disponibiliza a instrução para o estágio de graduação. É nesse estágio que as instruções executadas erroneamente devido a uma previsão de desvios incorreta são descartadas.

Os principais parâmetros do simulador são listados abaixo:

- largura de busca, despacho, decodificação e conclusão de instruções;
- tamanho das filas de instruções (fila de busca, despacho, *load/store*, tamanho da RUU);
- número de unidades funcionais (memória, desvios, inteiros e ponto-flutuante);
- configuração de níveis de *cache* e latências associadas à hierarquia de memória;
- configuração de tipos de previsores (híbridos, 2 níveis, perfeito, etc).

Gonçalves (2001; 2000), em sua tese de doutorado, estendeu o *sim-outorder*, desenvolvendo um simulador de arquiteturas multitarefas simultâneas, chamado **ss-smt**. Alguns recursos e estruturas foram replicados e outras foram compartilhadas entre as diferentes tarefas que, agora, são executadas simultaneamente. É importante ressalvar que nesse novo simulador cada tarefa corresponde a uma aplicação independente, ou seja, não existe comunicação entre as tarefas.

O conceito de *slot* foi adicionado ao simulador. Cada *slot* contém estruturas, filas, tabelas e conjunto de registradores necessários para manter o contexto das diferente tarefas em execução.

No nível de *pipeline* (Figura 3.7), as grandes diferenças encontradas no **ss-smt** são:

- o estágio de busca traz um bloco de instruções por ciclo, de apenas uma tarefa. A cada ciclo o estágio altera a tarefa ativa, usando a técnica ICOUNT, descrita na Seção 3. O número de instruções buscadas, como no simulador original, é limitado pela largura do barramento de busca (assim como no resto do *pipeline*);
- os demais estágios operam sobre um *mix* de instruções provenientes das diferentes tarefas, as quais são percorridas em estilo *round-robin*, até atingir o limite da largura do estágio ou até que não haja mais instruções provenientes de estágios anteriores;
- cada *slot* possui um banco de registradores próprio, para armazenar o contexto da tarefa sendo executada.

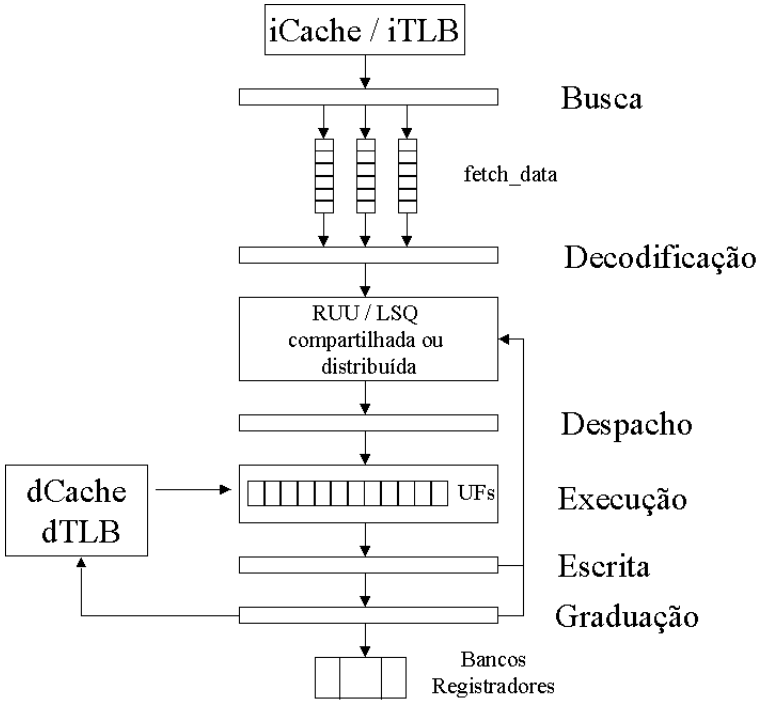

Figura 3.7: *Pipeline ss-smt*

Além dessas diferenças, novas funcionalidades foram adicionadas ao *ss-smt*. Entre elas pode-se citar (GONÇALVES, 2000):

- **Topologias de remessa** – Com a possibilidade de existência de mais de uma tarefa executando simultaneamente em uma arquitetura SMT, faz-se necessário avaliar o impacto da topologia das filas da RUU existentes entre os estágios de decodificação e remessa para as unidades funcionais. Duas topologias podem ser escolhidas (Figura 3.8): **centralizada** e **distribuída** (uma por tarefa). Na topologia distribuída, o estágio de remessa busca instruções das diferentes filas utilizando a técnica *round-robin*, o que não acontece na topologia centralizada, onde todas as instruções estão em uma única fila compartilhada por todas tarefas.

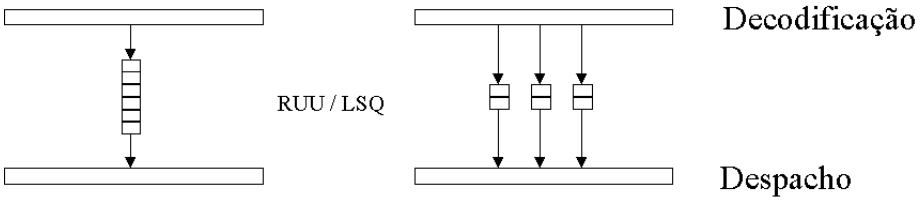

Figura 3.8: Topologias de remessa no *ss-smt*

- **Profundidade de decodificação** – Como dito anteriormente, o novo estágio de busca do simulador SMT traz as instruções da memória para as filas de instruções, as quais são inspecionadas a cada ciclo pelo estágio de decodificação. O número de entradas inspecionadas por fila pode ser configurado pelo usuário, sendo limitado apenas pelo tamanho de cada fila. É importante ressalvar que nem todas instruções inspecionadas serão decodificadas no mesmo ciclo.

- **Topologia de *caches*** – O simulador permite definir diferentes topologias de *caches*, ou seja, pode-se configurar o número de módulos e o número de bancos por módulo, como pode ser visto na Figura 3.9. Cada módulo opera utilizando um barramento independente, permitindo, assim, acesso simultâneo a módulos distintos. Os bancos, por sua vez, são multiplexados em cada módulo, tendo acesso exclusivo. Uma tarefa pode ocupar somente um banco, mas em um banco pode existir mais de uma tarefa. Dessa forma temos as seguintes definições: a **modularidade** da *cache* é o número de módulos independentes, a **separatividade** é o número de bancos multiplexados existentes em cada módulo e a **associatividade** é o número de entradas em cada banco associado a um mesmo endereço de memória. Já o **espaço vetorial de tarefas - evt** é obtido através de $separatividade \times associatividade$, definindo o espaço de memória necessário à co-existência de várias tarefas na *cache* (GONÇALVES, 2000).

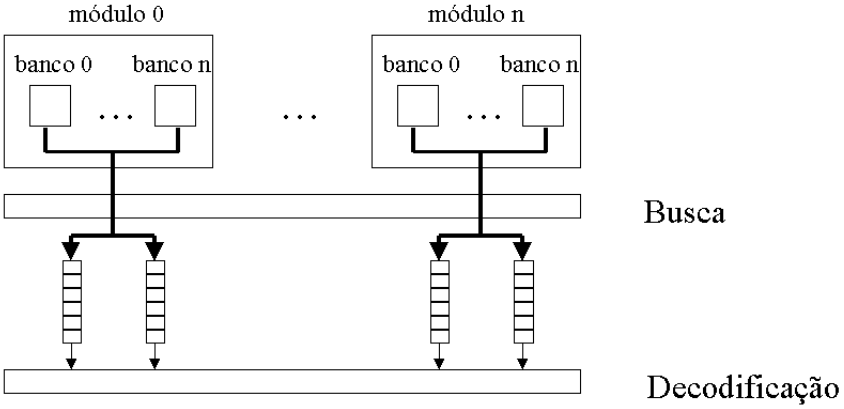

Figura 3.9: Topologias de *cache* no *ss-smt*

- **Previsão de desvios com taxa de acerto variável** – A última novidade introduzida no simulador é a possibilidade de o usuário configurar a taxa de acerto que o previsor de desvio deve obter durante a execução das tarefas. Isso permite a realização de estudos de extrapolação, onde avalia-se o impacto e a validade de um previsor de desvios com uma maior ou menor taxa de acerto. Essa implementação só leva em conta a taxa de acerto na previsão da direção de um desvio (tomado / não-tomado), ou seja, sempre que o previsor definir como tomado o desvio, ele também informará o alvo correto a ser buscado pelo estágio de busca.

4 PREVISÃO DE DESVIOS EM ARQUITETURAS SMT

Muito já foi desenvolvido na área de previsão de desvios, e, atualmente, pode-se atingir até 98 a 99% de acerto na previsão de certos tipos de desvios (MCFARLING, 1993; KESSLER, 1999). Para tal, empregam-se avançadas técnicas que, por sua vez, podem ocupar um espaço significativo na área final do processador.

Essas mesmas técnicas de previsão de desvios são também utilizadas nas arquiteturas SMT, porém, pouco estudo foi desenvolvido na área de previsão de desvios nessas arquiteturas. Em uma arquitetura SMT, o comportamento de diferentes previsores de desvios pode ser alterado devido à interferência que uma tarefa pode causar nas outras que estão sendo executadas simultaneamente. Além disso, um erro de previsão em uma tarefa introduzirá instruções do caminho inválido da mesma, as quais poderão influenciar na execução das demais tarefas, podendo até mesmo consumir recursos que poderiam ser melhor utilizados por outras tarefas, como linhas de *cache* e unidades funcionais.

Além disso, as tabelas de previsão de desvio nessas arquiteturas constituem um ponto de replicação de recursos capaz de aumentar seu desempenho, pois o comportamento das instruções de desvio é algo inerente a uma aplicação específica.

Dessa forma, o trabalho proposto nessa dissertação complementa os estudos já realizados acerca do comportamento/desempenho das arquiteturas SMT em relação à previsão de desvios, incluindo os seguintes aspectos:

- Proposta de diferentes topologias de previsores de desvios;
- Efeito da previsão de desvios em *pipelines* profundos.
- Impacto da variação da taxa de acerto do previsor de alvo de um desvio;

O processo de avaliação dessas arquiteturas SMT e de diferentes técnicas de previsão de desvios utiliza basicamente a técnica de simulação. Dessa forma, o restante do Capítulo irá apresentar, inicialmente na Seção 4.1, uma revisão sobre os trabalhos relacionados a múltiplas tarefas e previsão de desvios. Logo em seguida, na Seção 4.2, será apresentado as modificações realizadas no simulador **ss-smt** (apresentado na Seção 3.1) e uma ferramenta auxiliar para a validação das implementações realizadas no simulador.

A proposta de diferentes topologias está detalhada na Seção 4.3, sendo seguida das verificações do impacto da previsão de desvios em *pipelines* profundos, na Seção 4.4. Finalmente, na Seção 4.5 será comentado o trabalho de análise do impacto da taxa de acerto do alvo de um desvio nas arquiteturas SMT.

4.1 Múltiplas tarefas e a previsão de desvios

Trocas de contexto podem influir no funcionamento do processador, especialmente na previsão de desvios, uma vez que o comportamento das instruções de desvio é algo intrínseco à aplicação sendo executada.

Vários estudos acerca da influência das trocas de contexto foram efetuados. Perleberg e Smith (1993) afirma que é pouco provável que as entradas nas tabelas de previsão relativas a uma tarefa permaneçam após uma troca de contexto. Isso só seria possível se a BTB fosse extremamente grande. Várias simulações foram efetuadas, variando-se o número de instruções executadas de 1000 até 1 milhão entre cada troca de contexto. Em todos os casos simulados houve aumento das taxas de erro na previsão, a não ser no caso onde não havia informações de história associadas aos desvios.

Evers, Chang e Patt (1996) propuseram um previsor multi-híbrido - *Multi-Hybrid* - capaz de minimizar os efeitos das trocas de contexto na taxa de acerto do previsor. O *Multi-Hybrid* é capaz de prever desvios com exatidão superior a previsores híbridos convencionais e, além disso, é menos sensível à troca de contextos, por empregar previsores de diferentes características:

- previsores de dois níveis globais e locais (ver Seção 2.1), capazes de prever com alta taxa de acerto desvios dependentes de outros e das ocorrências anteriores de um mesmo desvio;
- previsores estáticos, capazes de prever com maior exatidão os desvios após um *flush* (esvaziamento) nas tabelas de previsão;
- previsores dinâmicos simples, para fazer a transição entre a previsão estática e a previsão em dois níveis;
- previsor AVG (CHANG; BANERJEE, 1995), responsável pela previsão em *loops*.

O *Multi-Hybrid* foi capaz de reduzir de 12,5 até 20% o número de previsões erradas na presença de trocas de contexto - a cada 256k instruções - em relação a previsores híbridos PAs/gshare e 2bits/gshare.

Finalmente, Co e Skadron (2001) realizaram uma análise mais profunda e atual sobre o real problema dos previsores de desvio em ambientes de *time-sharing*. O trabalho desenvolvido, ao contrário dos anteriores, apresenta um ambiente de simulação capaz de executar mais de uma tarefa (realizando troca de contexto) ao invés de apenas realizar um *flush* nas estruturas de previsão. Além disso, o trabalho simula fatias de tempo utilizadas em sistemas operacionais atuais, como Windows NT, Unix e Linux (de 50 a 200ms). Diferentes tipos e configurações de previsores de desvios foram analisadas, assim como a inclusão de *flush* nas estruturas dos previsores.

Co e Skadron concluíram que o tempo de treinamento de um previsor de desvio é, em média, de 128K instruções, tempo muito menor que a fatia de tempo utilizadas pelos sistemas operacionais atuais [1].

Dessa maneira, apenas fatias de tempo muito pequenas poderiam afetar a previsão de desvio de uma maneira geral. Mesmo a inclusão de *flush* das estruturas não faz diferença no desempenho final do previsor.

Esses estudos não levam em consideração arquiteturas que fornecem mecanismos para troca de contexto a nível de *hardware*, como a arquitetura SEMPRE (GONÇALVES,

[1] 50M de instruções levando em conta um processador de 1GHz e IPC de 1,75

2000), onde a fatia de tempo tende a ser menor, uma vez que existem trocas de contexto sempre que houver uma falha na *cache* ou na previsão de desvios.

Gonçalves, em (2000; 2001), como parte de sua Tese de Doutorado, desenvolveu, no simulador SMT apresentado na Seção 3.1, um mecanismo de previsão com taxa de acerto de direção (tomado/não-tomado) variável.

A partir desse simulador e desse mecanismo de previsão, foi feita uma comparação do desempenho de arquiteturas superescalares e multitarefas simultâneas com relação à taxa de acerto do previsor de desvios. Ambas arquiteturas foram analisadas variando-se, além da taxa de acerto, a quantidade de *hardware* disponível e o número de tarefas simultâneas (no caso da arquitetura SMT).

Gonçalves chegou a duas conclusões com essa análise. Primeiramente, em arquiteturas SMT com pouco *hardware* disponível e, independentemente do número de tarefas, a taxa de acerto da previsão não influencia muito no desempenho final da arquitetura, em termos de instruções por ciclo (IPC). Já quando a quantidade de *hardware* disponível é maior, o aumento do desempenho (IPC) acompanha tanto o aumento do *hardware* quanto da taxa de acerto do previsor.

Em (HILY; SEZNEC, 1996), três técnicas de previsão de desvios comumente utilizadas em arquiteturas superescalares, **gselect, gshare e 2-bit**, foram analisadas em uma arquitetura multitarefa simultânea. Foram testadas aplicações independentes e tarefas paralelas. O impacto da adição de um RAS (*Return Address Stack*) (KAELI; EMMA, 1991) por tarefa também foi analisado.

Hily e Seznec chegaram às seguintes conclusões:

- A adição de um RAS de 12 entradas por tarefa é suficiente para aumentar a taxa de acerto do previsor de desvio. Além disso, o custo de implementação é mínimo. Ao aumentar-se o número de entradas para 32 o ganho sobre 12 entradas é mínimo;

- Em um ambiente multiprogramado, o tamanho das tabelas de previsão (PHT e BTB) deve ser proporcional ao número de tarefas ativas, assim, uma tarefa não interfere na outra, seja para melhor ou pior (interferência construtiva / destrutiva). As tarefas paralelas não foram beneficiadas pelo ambiente compartilhado, na verdade, os benefícios são muito limitados e dependentes do tipo de previsor empregado;

- Mantendo-se pequenas as tabelas de previsão (para os três tipos de previsores estudados), nota-se um pequeno aumento no número de previsões erradas, tanto no ambiente paralelo, quanto no multiprogramado. Esse aumento foi causado, principalmente, pelo aumento de conflitos na BTB.

O trabalho de Hily e Seznec não levou em conta o impacto resultante do aumento ou da diminuição da taxa de acerto dos previsores no IPC, ou seja, não é possível afirmar se o desempenho final da arquitetura multitarefa foi afetado ou não pelos estudos conduzidos.

Além desses trabalhos, pode-se citar a importância do trabalho de Jiménez e Lin (2003), concluindo que as futuras gerações de processadores, as quais disporão de *hardware* para explorar diferentes alternativas para melhorar o desempenho, irão favorecer previsores de desvio simples, ao invés de complexos previsores. Nesse trabalho, é proposto o previsor *gshare.fast*, comparável em desempenho aos melhores previsores atuais: o previsor *perceptron* (JIMENEZ; LIN, 2002), o *Multi-Hybrid* (EVERS, 2000) e o 2Bc-gskew (PIERRE MICHAUD; UHLIG, 1997). A idéia básica do *gshare.fast* é prover um previsor *pipeline* capaz de realizar previsões em um único ciclo, mesmo utilizando *pipelines* profundos.

4.2 Simulador *Golden SMT*

Todos experimentos realizados nesse trabalho utilizaram a técnica de simulação para geração de resultados. A simulação, apesar de conseguir reproduzir e considerar os parâmetros e variáveis de um sistema real, não consegue garantir por si só que as alterações realizadas no simulador em uso estão de acordo com as especificações. É preciso utilizar algum método externo que realize a validação do simulador em uso contra um certo conjunto de normas ou padrões.

Assim, a validação das mudanças realizadas no simulador *ss-smt* foi feita através da utilização da técnica de *Golden Simulator* (SANTOS, 2003). Essa técnica utiliza o modo de execução *fast forward* disponível no simulador *sim-outorder* original. Nesse modo de execução não existe execução especulativa, ou seja, as instruções corretas são executadas uma após a outra.

A versão original do *sim-outorder* é tida como estável e sua execução acreditada (devido à grande difusão do uso da ferramenta). Dessa forma, os resultados obtidos pela execução desse simulador podem ser consideradas corretos (norma/padrão). A execução no modo *fast forward* do *sim-outorder* foi, então, modificada para que, a cada instrução executada, os operandos de origem e destino fossem impressos na saída padrão ou em algum arquivo informado pelo usuário. O conjunto dessas instruções e operandos é visto como um traço do programa que foi executado e pode ser utilizado como fonte de futuras comparações, pois espera-se que a execução do mesmo programa e entradas gere sempre o mesmo comportamento em sua saída.

O simulador *ss-smt* foi, então, modificado para que as instruções sendo graduadas em cada uma das tarefas fossem impressas na saída padrão ou em algum arquivo informado pelo usuário. A comparação das saídas geradas pelo simulador *ss-smt* devem ser semelhantes a do simulador *sim-outorder*.

Para automatizar o processo de comparação desses traços foi desenvolvido um *script* em linguagem *Perl*, chamado *SMT-Comp.pl*. A função básica do *script* é de realizar a leitura das aplicações a serem executadas pelo simulador *ss-smt*, dispará-lo e, para cada uma das aplicações, disparar uma instância do simulador *sim-outorder golden*. Feito isso, o *script* passa a verificar os traços gerados pelos simuladores, e, em caso de alguma diferença na saída de uma das tarefas, a execução de todos simuladores é abortada. Ao abortar a execução, o *script* informa:

- número de instruções graduadas em cada uma das tarefas até o momento;
- tarefa que falhou, juntamente com a instrução que causou a diferença;
- instruções que foram executadas imediatamente antes à falha (o número de instruções é configurável);

A execução sem falhas prova que o simulador em teste executou as mesmas operações que seriam executadas pelo *sim-outorder* quando utilizado com aquela aplicação e seus respectivos dados de entrada.

É importante ressalvar que o *SMT-Comp.pl* não executa a comparação dos resultados de cada uma das aplicações, ou seja, o resultado que a aplicação gera para seu usuário final não é analisada. A simples comparação do resultado gerado por uma aplicação não pode ser utilizado como parâmetro único para validação de mudanças arquiteturais, uma vez que o mesmo resultado de saída poderia ser gerado pela execução de diferentes instruções.

Garantindo que as instruções (e seus operandos) são idênticos, garante-se que a aplicação gerará o mesmo resultado.

A Figura 4.1 ilustra o mecanismo descrito acima. Nota-se que o número de processos executados pelo *SMT-Comp.pl* é dependente da configuração do simulador *ss-smt*. O *SMT-Comp.pl* foi executado com todos *benchmarks* utilizados nessa dissertação, porém a execução limitou-se a quatro tarefas simultâneas, devido à grande quantidade de memória necessária para a execução de todos simuladores simultaneamente.

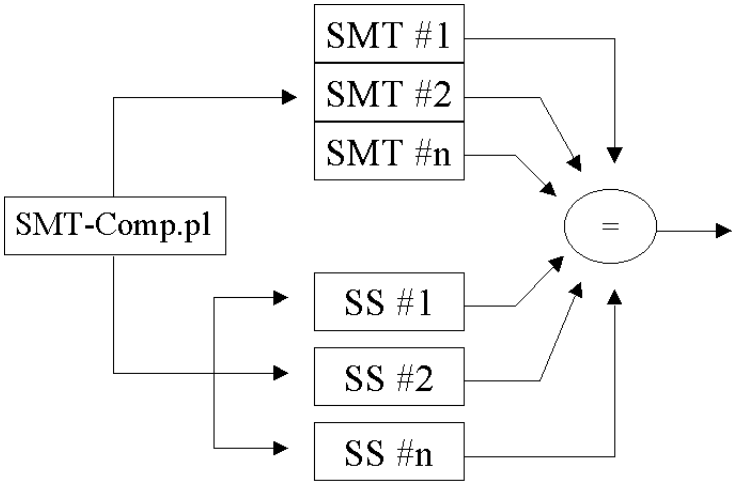

Figura 4.1: Validação do *ss-smt* utilizando o *SMT-Comp.pl*

Além de utilizar o *SMT-Comp.pl* como forma de validação nesse trabalho, um total de mais de 4 trilhões de instruções foram executadas através do simulador *ss-smt*. Foram mais de 1500 simulações executadas ao longo desse trabalho sem nenhum problema aparente – vazamentos de memória ou erros de execução.

4.3 Proposta de topologias para previsão de desvios

A replicação de estruturas em um processador SMT é a primeira alternativa para aumentar seu desempenho. Contudo, essa replicação pode não apresentar uma boa relação custo-benefício, além de eliminar os benefícios do compartilhamento, ou seja, a otimização dos recursos.

O comportamento da previsão de desvios é algo inerente a uma tarefa específica, dessa maneira, as estruturas necessárias para a previsão de desvios são um ponto onde a replicação tende a melhorar o desempenho da arquitetura SMT.

O compartilhamento das tabelas de previsão pode levar à monopolização de suas linhas por tarefas com alto índice de desvios dinâmicos. Por outro lado, uma tarefa com baixo índice de desvios pode sub-utilizar os recursos disponíveis. Dessa forma, propõe-se uma alternativa intermediária entre o compartilhamento e a replicação, a **clusterização**. As estruturas necessárias à previsão de desvios (tabelas e registradores) passam a ser distribuídas em módulos e bancos, de maneira análoga à topologia de *i-cache* proposta em (GONÇALVES et al., 2000). Em uma organização clusterizada (particionada), uma tarefa que necessita de muitos recursos só irá ocupar os recursos das tarefas que estão alocadas em um mesmo *cluster*.

Diferentes módulos utilizam barramentos de leitura e escrita independentes, enquanto, bancos são multiplexados internamente aos módulos. O uso de bancos permite uma sim-

plificação na complexidade do *hardware*, porém limita o uso de dois diferentes bancos em um mesmo ciclo. A Figura 4.2 ilustra o acoplamento dessas estruturas a um *pipeline* SMT.

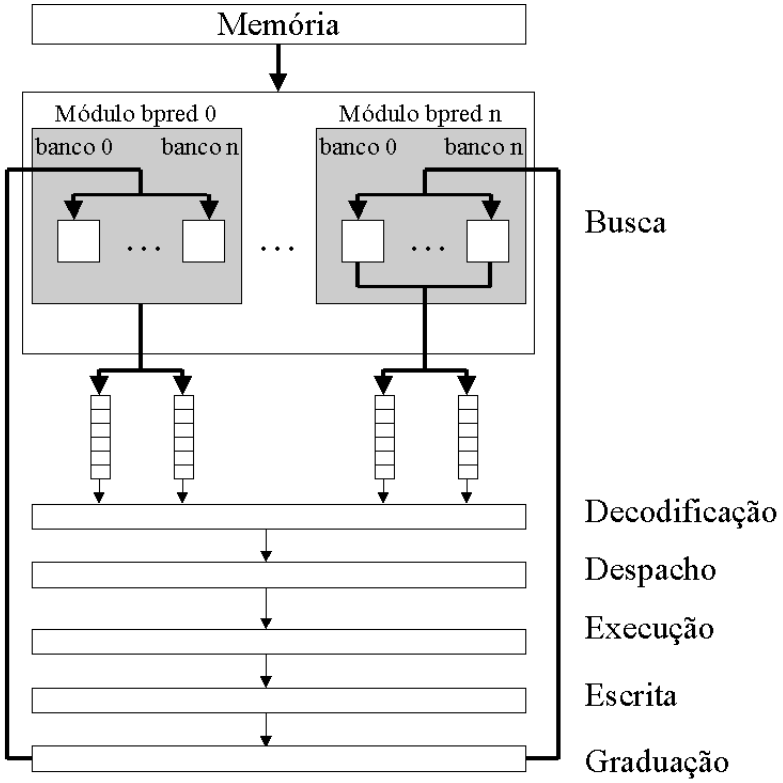

Figura 4.2: Exemplo de topologia para previsão de desvios em arquiteturas SMT

Neste trabalho, três diferentes topologias para previsão de desvios são consideradas: distribuída, compartilhada ou particionada (*clusters*).

A topologia distribuída provê estruturas de previsão individuais a cada tarefa. Além disso, o número de consultas (*lookups*) e atualizações (*updates*) que podem ser realizados em um ciclo não são limitados. Essa topologia é a mais similar àquela encontrada em uma arquitetura CMP, porém sua implementação requer uma grande quantidade de barramentos de acesso às estruturas de previsão de desvios.

A arquitetura compartilhada, por sua vez, é a oposta à distribuída em termos de compartilhamento de recursos. Todas tarefas em execução compartilham as mesmas estruturas de previsão e o número de consultas e atualizações também não é limitado. Sua implementação tende a ser ainda mais complexa, pois requer uma grande tabela capaz de realizar múltiplas consultas e atualizações, além de múltiplos barramentos para transmitir e receber consultas e atualizações.

Topologias particionadas tentam balancear a complexidade do *hardware* e o desempenho da solução. Nem todas as tarefas necessitam acesso às tabelas de previsão em um mesmo ciclo, logo, o compartilhamento dos barramentos e tabelas pode apresentar uma boa relação custo/benefício em uma possível implementação de uma arquitetura SMT. Diferentemente das topologias anteriores, as topologias particionadas impõem limites no número de consultas e atualizações que podem ser feitas em um mesmo ciclo. Dessa forma, pode-se considerar que as topologias distribuída e compartilhada fornecem os limites superiores de desempenho quando a limitação de consultas e atualizações em um mesmo ciclo não está presente.

4.4 Profundidade variável de *pipeline*

Os projetistas de *hardware* têm utilizado freqüentemente a técnica de aumentar a freqüência de funcionamento dos processadores para atingir maiores desempenhos. Essa técnica envolve desde o uso de novos processos e materiais na fabricação dos *chips* até o aumento no número de estágios do *pipeline*. Porém, algumas barreiras impedem que os projetistas utilizem livremente esses métodos, como, por exemplo, limites físicos e as quebras no fluxo de instruções no *pipeline* – instruções de desvio, *misses*, etc.

O aumento da profundidade do *pipeline* em processadores permite a maximização no tempo de ciclo do relógio, o qual é definido pelo maior tempo necessário para a execução dos estágios individuais do *pipeline*. Dessa forma, ao quebrar-se a tarefa sendo executada em um estágio em etapas ainda menores, consegue-se diminuir o tempo de execução original do estágio, aumentando a freqüência de funcionamento do processador.

De uma forma quantitativa, o aumento no número de estágios de um *pipeline* irá aumentar o número de ciclos necessários para a execução de uma instrução. Isso é razoável, pois a instrução deverá passar por um número maior de estágios até a sua finalização. Por outro lado, esse mesmo aumento no número de estágios irá diminuir o tempo de cada ciclo, pois a quantidade de processamento é mantida constante, ou seja, o tempo total de processamento é o mesmo, mas o número de etapas é maior, resultando em um menor tempo por estágio.

O desempenho final de um processador não depende apenas de sua freqüência de operação (*clock* – tempo de ciclo), mas também do número de instruções sendo graduadas em cada um dos ciclos do relógio. O desempenho de um processador seria máximo se nunca ocorressem interrupções no *pipeline*, ou seja, a cada ciclo novas instruções estariam sendo inseridas e retiradas. Porém, a ocorrência dessas interrupções no fluxo contínuo de instruções é algo intrínseco ao modelo de programas utilizados hoje em dia, conforme discutido no Capítulo 2.

Essas interrupções, ou conflitos (*hazards*), têm sua penalidade maximizada ao passo em que o número de estágios do *pipeline* é aumentado, uma vez que esses conflitos não permitem a alimentação constante de instruções ou podem causar até mesmo o esvaziamento completo do *pipeline*.

O erro na previsão de desvios é o mais importante exemplo de conflito que pode ocorrer em um *pipeline*. Uma previsão errada irá causar o esvaziamento de todas instruções posteriores e, conseqüentemente irá diminuir o IPC geral do processador, pois além de descartar todo processamento realizado até o momento, o *pipeline* precisará ser preenchido novamente com instruções do caminho correto. Um *pipeline* profundo irá contribuir com uma maior latência entre o ponto de esvaziamento até o ponto de recuperação do estado correto, conforme pode ser visto na Figura 4.3.

Três situações distintas podem ser vistas na Figura 4.3. Nas situações (a) e (b) tem-se um *pipeline* simples de 4 estágios (B, D, E, G). Já em (c) tem-se 8 estágios, cada estágio de (a) e (b) é duplicado (B1, B2, D1, D2, E1, E2, G1, G2). Em todos os casos a primeira instrução sendo executada é um desvio condicional. Em (a), a previsão realizada está correta e, com apenas oito ciclos o *pipeline* consegue executar cinco instruções, contrastando com (b) que consegue executar apenas duas instruções. No caso (b), o desvio é previsto de maneira errada, sendo necessário o esvaziamento do *pipeline* no ciclo 4. Esse esvaziamento demonstra a penalidade associada de 4 ciclos, pois apenas no ciclo 8 tem-se uma nova graduação de instrução. Ao aumentar a profundidade do *pipeline*, essa penalidade passa a ser de 8 ciclos, conforme pode ser visto em (c). Os exemplos acima demonstraram as penalidades mínimas associadas a erros na previsão de desvios. Essa penalidade pode

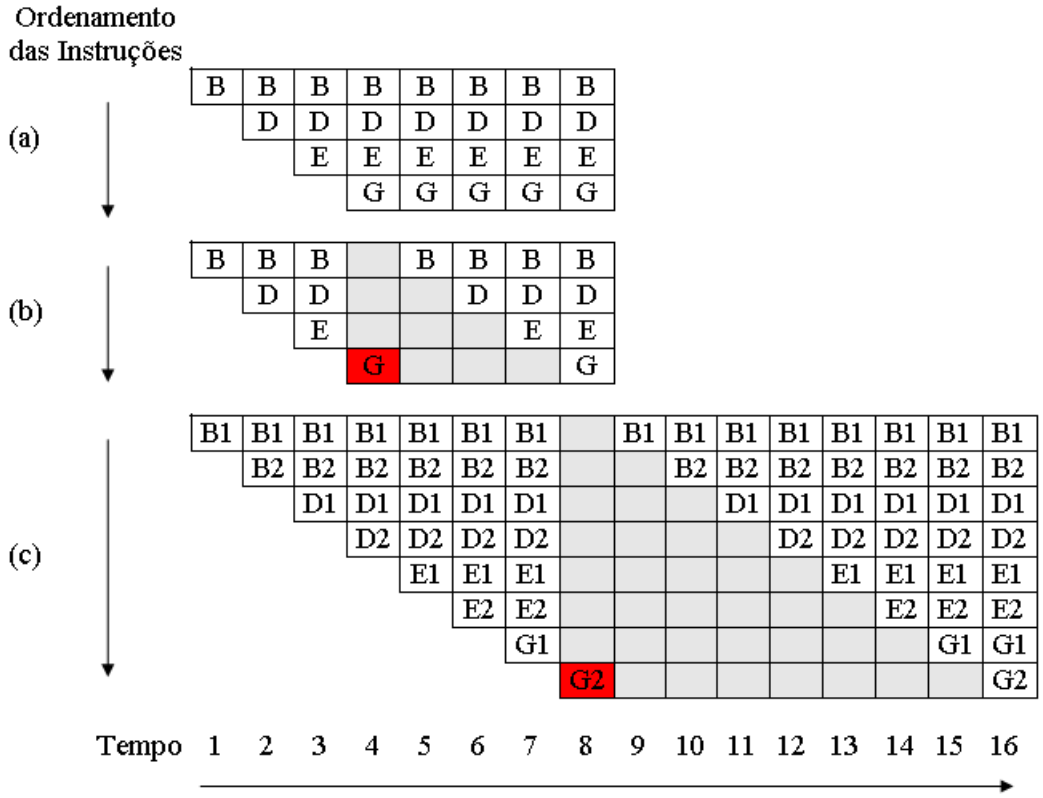

Figura 4.3: Penalidade de erro na previsão de desvios em *pipelines* profundos

ser maior no caso de ocorrência de dependência de dados na execução das instruções.

Resumidamente, pode-se notar que existe uma relação custo-benefício entre a maior vazão e a maior penalidade para conflitos em um *pipeline* profundo. A quantidade de paralelismo existente tende a diminuir a profundidade do *pipeline*, enquanto a não-existência de conflitos tende a aumentar essa profundidade.

Para realizar a avaliação do comportamento de uma arquitetura SMT com *pipelines* profundos, foi adicionado ao simulador **ss-smt** a possibilidade de configuração do número de estágios do *pipeline* conforme a técnica utilizada por Pilla (2004). A idéia básica é adicionar, para cada instrução, em cada um dos estágios originais dos simuladores, o conceito de ”ciclo em que a instrução pode ser processada por um próximo estágio”.

Dessa maneira, por exemplo, para simular um *pipeline* de 9 estágios – busca de 4 estágios, além dos 5 outros estágios originais (decodificação, despacho, execução, escrita e graduação) – adicionaria-se 3 ciclos extras no processamento do estágio original de busca. As instruções que tiveram sua busca iniciada no ciclo 1 recebem a identificação que só podem ser decodificadas após o ciclo 5, uma vez que o estágio de busca modelado deve possuir 4 estágios. Essa mesma analogia é válida para os demais estágios originais do *pipeline*, conforme exemplo na Figura 4.4.

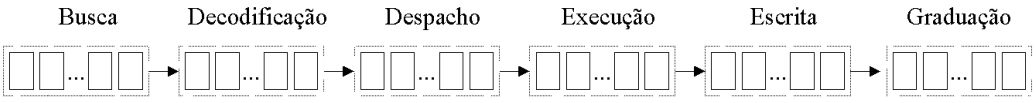

Figura 4.4: Profundidade variável do *pipeline*

4.5 Previsão de desvios com taxa de acerto variável na BTB

Arquiteturas SMT combinam características de arquiteturas multitarefas e superescalares, beneficiando-se da busca/despacho/execução de múltiplas instruções por ciclo das modernas arquiteturas superescalares e da habilidade de esconder longas latências das arquiteturas multitarefas.

As mesmas técnicas de previsão de desvios utilizadas em arquiteturas superescalares são também utilizadas nas arquiteturas SMT e, conforme analisado em outros trabalhos, a ocorrência de erros de previsão acaba reduzindo o desempenho de ambas arquiteturas (mesmo que de maneiras distintas) (PIZZOL; PILLA; NAVAUX, 2001; GONÇALVES et al., 2001). A redução no desempenho é derivada basicamente da execução de instruções em caminhos que serão descartados posteriormente. Essa execução, além de causar o esvaziamento do *pipeline*, reduz o desempenho por outros três motivos principais:

- poluição nas *caches* de níveis inferiores (L1 e L2), já que instruções úteis podem ceder lugar para as instruções do caminho errado, aumentando o número de *misses*;
- contenção nos barramentos de acesso à hierarquia de memória, pois enquanto uma falta no caminho errado está sendo tratada, outra falta no caminho correto pode estar sendo bloqueada;
- utilização de unidades funcionais necessárias para execução de instruções do caminho correto.

Em arquiteturas SMT, os problemas descritos acima podem ser ainda mais evidentes, pois as *caches* serão compartilhadas por mais tarefas, assim como as unidades funcionas. Uma tarefa executando muitas instruções em caminhos inválidos poderá afetar o desempenho de outras, diminuindo mais ainda o desempenho global.

A previsão de desvios pode ser resumida por um simples algoritmo (Figura 4.5): ao encontrar uma instrução de desvio deve-se determinar, primeiramente, se o desvio será tomado ou não-tomado. Caso seja não-tomado, nada mais deve ser feito e a arquitetura pode seguir o fluxo normal de instruções. Já se o desvio for tomado, a previsão deve, então, informar qual é a instrução que deverá ser buscada na seqüência.

A segunda etapa da previsão de desvios é realizada geralmente por uma BTB (*Branch Target Buffer*), com uma estrutura um pouco diferente daquela apresentada na Seção 2.1: os campos de histórico de desvios não ficam armazenados na BTB, apenas os bits menos significativos do endereço do desvio e o endereço do alvo. Essa BTB é indexada pelo endereço do desvio (PC) e, caso seja encontrado em alguma entrada da tabela, o campo de alvo da entrada será retornado. Caso o endereço do desvio não seja encontrado, diz-se que um *miss* ocorreu, e, geralmente uma previsão estática é realizada.

Para permitir a avaliação do impacto da exatidão dos previsores de desvio em arquiteturas superescalares e SMT, Gonçalves (2001) desenvolveu um mecanismo de previsão com taxa de acerto definido pelo usuário em tempo de configuração do simulador. Com esse mecanismo, o usuário pode avaliar o impacto de um previsor no desempenho final da arquitetura antes mesmo de preocupar-se com o desenvolvimento do *hardware* para tal. Além disso, essa abordagem facilita o desenvolvimento de análises de viabilidade e alternativas às técnicas de previsão de desvios. A implementação realizada por Gonçalves atuava apenas na primeira etapa da previsão de desvios, ou seja, sempre que o desvio fosse previsto como tomado, o estágio de busca seria direcionado para o alvo correto, como se existisse uma tabela infinita de endereços. Ao forçar um erro de previsão, o mecanismo

utilizado por Gonçalves apenas direcionava o estágio de busca para o caminho errado, ou seja, não existia a possibilidade de a previsão retornar um endereço distinto ao caminho tomado/não-tomado.

Esse mecanismo foi estendido para também atuar na segunda etapa da previsão de desvios, ou seja, na previsão do alvo para a instrução de desvio corrente. O usuário pode definir, em tempo de configuração, a taxa de acerto do previsor de direção e também a taxa de acerto na BTB. Além disso, o usuário pode definir como serão distribuídos os erros na BTB:

- endereço alvo incorreto (poluição);
- endereço do desvio não encontrado (*miss*)

O ponto de funcionamento do mecanismo é ilustrado na Figura 4.5.

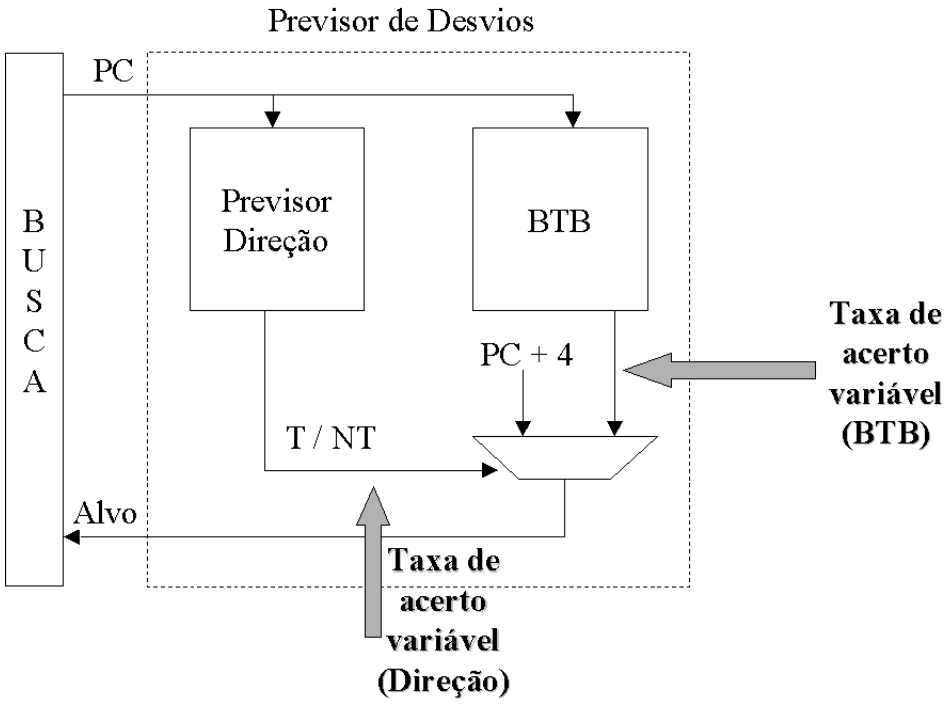

Figura 4.5: Previsão de desvios e a taxa de acerto variável

Ao encontrar um desvio que será tomado, o mecanismo sorteia um número randômico entre 1 e 100, representando taxas de acerto de 1 a 100%. Se o número sorteado for menor que a taxa definida pelo usuário, o mecanismo força o acerto da previsão na BTB, caso contrário, uma previsão errada é forçada. Nesse caso, um novo número randômico é sorteado para definir se o erro na previsão será causado por um *miss* ou por um endereço incorreto, conforme definição do usuário.

A simulação do erro de previsão na BTB através do uso de um endereço incorreto foi implementada tendo por base o endereço alvo de todos os desvios diretos do programa sendo executado. Antes de iniciar a execução do *benchmark*, o simulador varre todo o programa objeto guardando em memória o endereço alvo dos desvios diretos. Desvios indiretos não são utilizados nessa coleta, pois seus alvos só são conhecidos em tempo de execução. A simples utilização de endereços randômicos dentro do espaço de código

válido do programa não pode ser utilizado como base para o mecanismo, pois acarretaria na quebra de blocos básicos na *cache* de instruções.

É importante ressalvar que o mecanismo deve conhecer o endereço correto para o alvo do desvio. Dessa forma, ele é implementado no estágio de decodificação do simulador utilizando para tal a facilidade da previsão perfeita já existente no *sim-outorder*.

5 RESULTADOS

Neste Capítulo serão apresentados os resultados experimentais relativos às simulações acerca de previsão de desvios em arquiteturas SMT, utilizando para tal as ferramentas descritas ao longo desse trabalho. Para controle das simulações foi utilizada a ferramenta **SimMan** (STAEHLER; PIZZOL; NAVAUX, 2003).

Primeiramente, será feita uma breve análise dos *benchmarks* utilizados nas simulações em relação a classes de instruções, com ênfase nas instruções de desvio. Logo após, na Seção 5.2, será analisado o impacto das topologias propostas para previsão de desvio (compartilhada, distribuída e particionada) em arquiteturas SMT.

Na Seção 5.3 será analisado o impacto da previsão de desvios em *pipelines* profundos, comparando o desempenho de arquiteturas superescalares com arquiteturas SMT de quatro e oito tarefas. Finalmente, na Seção 5.4, será verificado o efeito da taxa de acerto de previsão na BTB. De uma maneira geral, será utilizado a métrica IPC (Instruções por ciclo) como medida de desempenho.

5.1 Desvios e o comportamento do SPEC CPU 2000

A técnica de simulação é composta de várias etapas, desde a definição do simulador até a coleta e análise das estatísticas. Dessa forma, uma das primeiras etapas necessárias em uma avaliação de arquiteturas de processadores é a definição e análise dos *benchmarks* que serão utilizados no processo de simulação das técnicas em questão.

Na área de processadores, o conjunto de *benchmarks* mais utilizado em centros de pesquisa é o SPEC CPU 2000 (HENNING, 2000). A utilização de um conjunto padrão de *benchmarks*, aliado a simuladores semelhantes, facilita a comparação entre diferentes trabalhos.

O SPEC CPU 2000 é composto de onze aplicações de inteiros (escritas em linguagem C/C++) e quatorze aplicações de ponto flutuante (escritas em linguagem C, FORTRAN 77 e FORTRAN 90). Nesse trabalho apenas oito dessas aplicações serão analisadas, sendo quatro de inteiros e quatro de ponto flutuante.

Cada *benchmark* teve seiscentos milhões de instruções executadas. As Tabelas 5.1 e 5.2 descrevem os *benchmarks* utilizados, juntamente com o número de instruções distintas de desvio executadas em cada um dos *benchmarks*.

A Figura 5.1 mostra a distribuição de cada tipo de instrução executada nos *benchmarks*. Foram levados em conta cinco tipos de instruções: *load*, *store*, desvios (incluindo *jumps*), cálculos de inteiros e de ponto flutuante. Como pode ser notado, o número médio de instruções de desvio é de 20%, ou seja, a cada instrução de desvio temos outras quatro instruções diversas em média. O único *benchmark* que foge a essa regra é o **mgrid**, com pouco menos de 4% de instruções de desvio. Porém, o número de desvios distintos que

Tabela 5.1: *Benchmarks* de Inteiros

Nome	Desvios	Descrição
175.vpr	2911	Roteamento e colocação de circuitos em FPGAs
176.gcc	33276	Compilador C
197.parser	4048	Processamento de linguagem natural
256.bzip2	1047	Utilitário de compressão de dados

Tabela 5.2: *Benchmarks* de Ponto Flutuante

Nome	Desvios	Descrição
171.swim	898	Modelagem *Shallow water* (F77)
172.mgrid	1690	Solução multi-grid para campos potenciais em 3D (F77)
177.mesa	2206	Biblioteca gráfica 3D (C)
183.equake	678	Modelagem de terremotos: simulação de elementos finitos (C)

foram executados pela aplicação **mgrid** é superior ao de outros quatro *benchmarks*.

Além de analisar a quantidade de instruções de desvio em relação ao número total de instruções executadas e o número de desvios distintos executados, é importante analisar cada tipo de desvio em separado. Para esse análise, os desvios foram classificados em três grupos (não-exclusivos):

- Tomado e Não-Tomado;
- Direto e Indireto;
- Condicional e Incondicional (*jumps*).

Um importante aspecto na distribuição das instruções de desvio, é a relação entre desvios tomados e não-tomados. Um desvio é dito tomado quando a instrução sucessora não é seqüencial, ou seja, existe um ”salto” no código. Durante a fase de busca de instruções em um *pipeline*, consulta-se tabelas de previsão para determinar o provável fluxo que deve ser buscado no próximo ciclo. Caso a previsão aponte o desvio como tomado, um passo extra na previsão faz-se necessário, o acesso à BTB, a qual guarda a previsão mais atualizada para o alvo do desvio.

O número de acessos à BTB tem uma relação direta ao número de desvios tomados em um determinado programa. Uma taxa baixa de acerto na BTB pode acabar trazendo

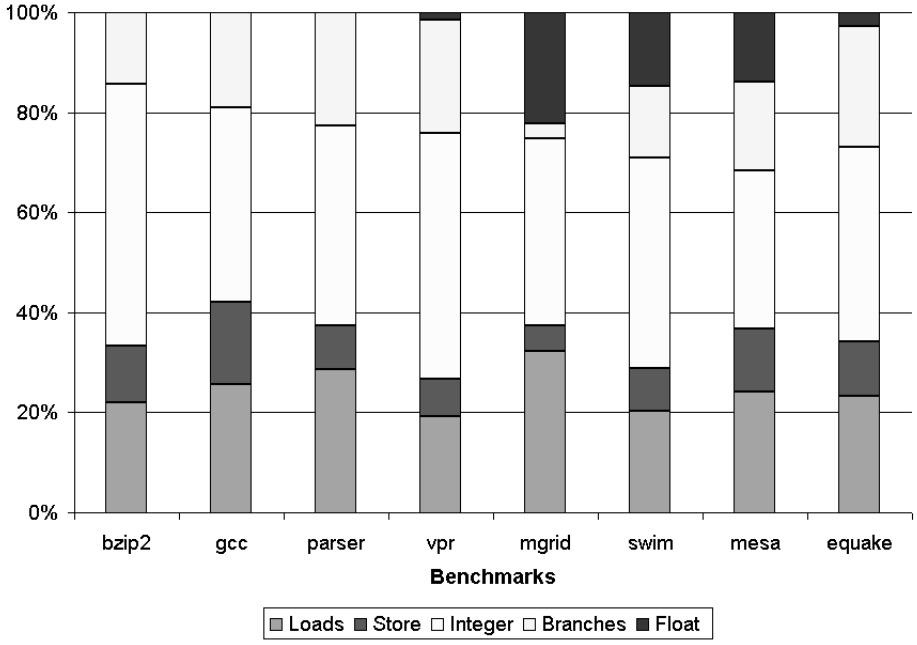

Figura 5.1: Classes de Instruções

instruções de fluxos errados para o *pipeline* (desperdiçando recursos) e poluindo, principalmente, a *cache* de instruções (instruções não necessárias poderão ocupar espaço de outras instruções).

Dessa forma, nota-se na Figura 5.2, que na maioria dos *benchmarks* o número de desvios tomados ultrapassa 60%, chegando a 97% no *benchmark* **mgrid**. A análise do número de desvios tomados e a quantidade de desvios distintos permite estimar o número ótimo de entradas para a BTB.

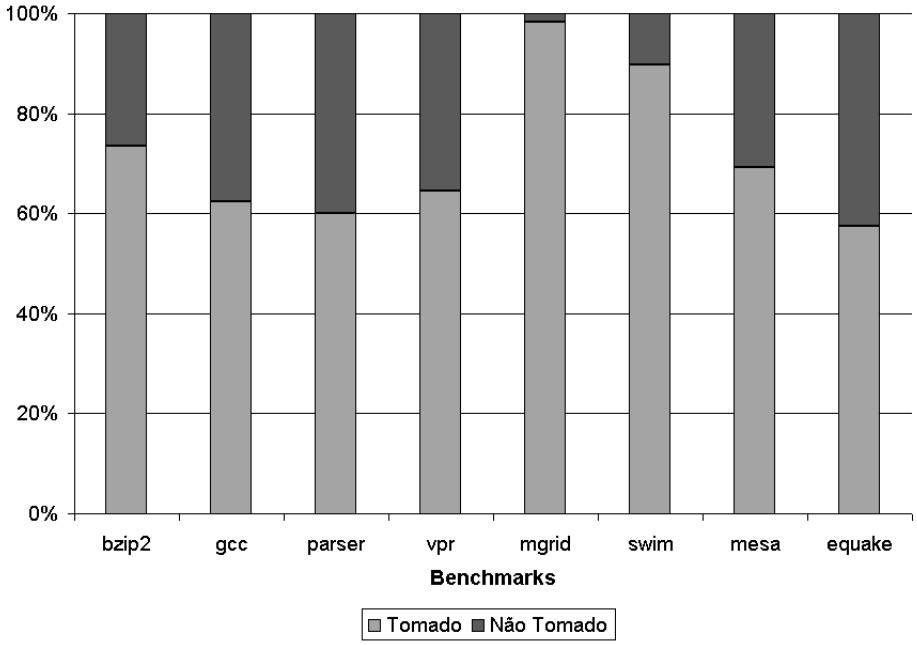

Figura 5.2: Desvios Tomados e Não-Tomados

Na Figura 5.3 a relação entre desvios diretos e indiretos é analisada. Um desvio é dito indireto quando o seu endereço alvo é apontado por um registrador, enquanto um desvio direto tem seu endereço alvo calculado através da soma de um deslocamento (positivo ou negativo) ao PC atual ou através de um endereço imediato na própria instrução. Os desvios diretos são a maioria em todos os benchmarks, chegando a 99% no *benchmark mgrid*. Em algumas arquiteturas é possível que um erro na previsão de um desvio direto seja detectado antes mesmo da sua execução, durante o estágio de decodificação das instruções. Esse erro é conhecido como *misfetch*. Dessa forma, evita-se que em casos de erro na previsão do alvo o mesmo só seja corrigido na finalização da execução da instrução.

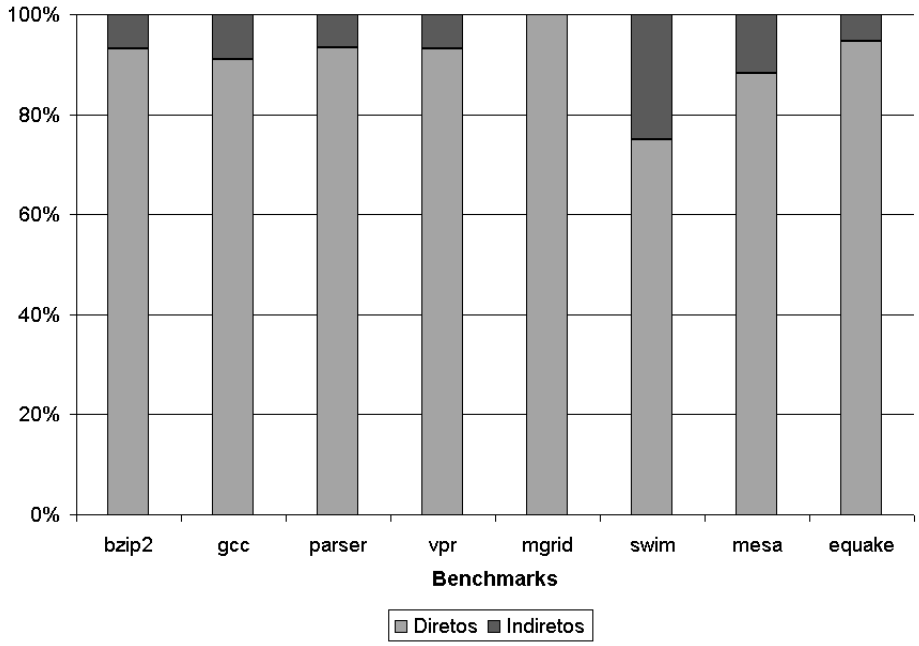

Figura 5.3: Desvios Diretos e Indiretos

Por fim, a relação entre desvios condicionais e incondicionais é mostrada na Figura 5.4. Um desvio incondicional também é conhecido como *jump*. Desvios incondicionais podem ser resolvidos no estágio de decodificação de um *pipeline*, enquanto um desvio condicional deve aguardar a fase de execução para realizar o teste de condição. Dessa forma, a detecção de erros de previsão para desvios incondicionais pode ser feita antecipadamente, esvaziando o *pipeline* e redirecionando o estágio de busca. A grande parte, cerca de 70%, dos desvios nos *benchmarks* analisados é condicional.

5.2 Topologias de previsão de desvios

A avaliação das diferentes topologias de previsão de desvios foi realizada através de simulação, utilizando duas configurações de processadores, SMT-4 e SMT-8. Ambas utilizam como base o simulador *ss-smt* e implementam arquiteturas SMT de quatro e oito *slots*, respectivamente. A Tabela 5.3 apresenta os principais parâmetros de *pipeline* e da hierarquia de memória utilizados nesses processadores. Os parâmetros foram escolhidos de forma a evitar gargalos na arquitetura, os quais poderiam mascarar os efeitos da busca e previsão de desvios no desempenho de arquiteturas SMT.

Para o processador SMT-4, oito diferentes topologias foram analisadas. Já para o

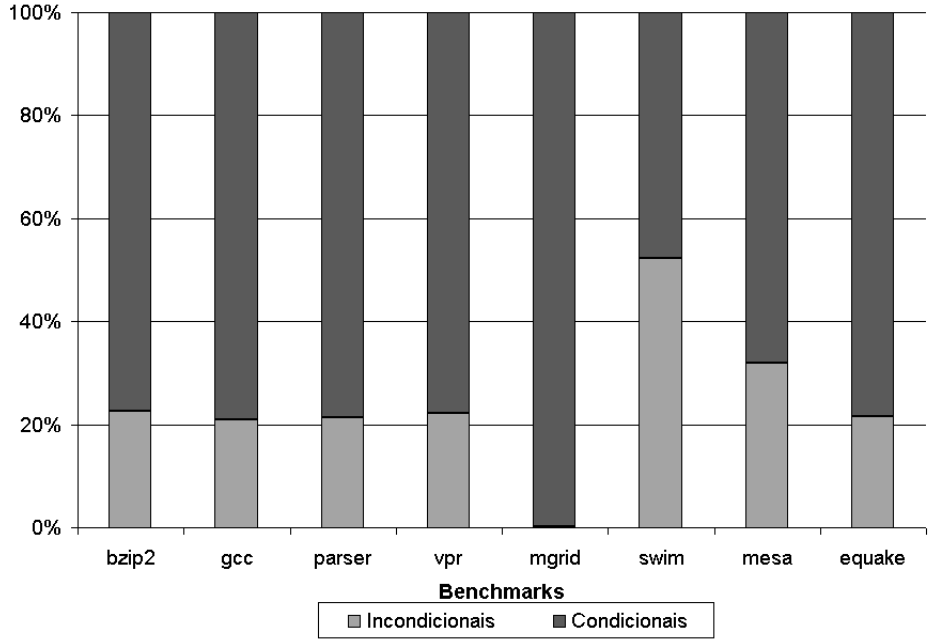

Figura 5.4: Desvios Condicionais e Incondicionais

processador SMT-8 foram doze diferentes topologias:

- **Distribuída**: cada *slot* tem seu próprio previsor;
- **Compartilhada**: todos *slots* compartilham um único previsor;
- **Particionada**: os *slots* são divididos em módulos e bancos, utilizando para tal seis diferente configurações: 1 módulo x 1 banco, 1 módulo x 2 bancos, 1 módulo x 4 bancos, 1 módulo x 8 bancos (SMT-8), 2 módulos x 1 banco, 2 módulos x 2 bancos, 2 módulos x 2 bancos (SMT-8), 4 módulos x 1 banco, 4 módulos x 2 bancos (SMT-8) e 8 módulos x 1 banco (SMT-8).

A topologia distribuída é semelhante à configuração de 8(4) módulos x 1 banco e a compartilhada é semelhante à 1 módulo x 1 banco. A única diferença é que nas topologias distribuída e compartilhada não existe contenção nos barramentos, ou seja, não existe restrições quanto ao número de consultas e atualizações que podem ser realizadas em um mesmo ciclo.

Todas topologias compartilham uma configuração base para a previsão de desvios, contendo um previsor híbrido (dois níveis e bimodal), conforme ilustrado na a Tabela 5.4. Além de variar as topologias de previsão de desvios, as simulações também levaram em conta diferentes topologias para *cache* de instruções (*i-cache*): a quantidade total de memória disponível para o primeiro nível de *i-cache* (Tabela 5.3) foi dividida em módulos (GONÇALVES et al., 2000) para ambos SMT-4 e SMT-8. Utilizando um único módulo de *i-cache*, oito instruções podem ser buscadas em um ciclo. Dobrando o número de módulos, dezesseis instruções podem ser buscadas. Já com quatro módulos de *i-cache*, o número máximo de instruções buscadas em um mesmo ciclo passa a ser 32.

Os mesmo oito *benchmarks* do SPEC2000 foram utilizados nas simulações, porém para o processador SMT-4 foram utilizados quatro combinações (*workloads*), utilizando quatro dos *benchmarks* em cada uma (dois de inteiros e dois de ponto-flutuante). Uma

Tabela 5.3: Principais parâmetros para SMT-4 e SMT-8

Parâmetro	**SMT-4**	**SMT-8**
Largura do *Pipeline*	32	64
RUU (por tarefa)	128	128
LSQ (por tarefa)	64	64
Portas de memória	8	16
latência	1 ciclo / 1 ciclo	
Int add/sub	16	32
latência	1 ciclo / 1 ciclo	
Int mult/div	8	16
latência	3 ciclos / 12 ciclos	
FP add/sub	16	32
latência	2 ciclos / 2 ciclos	
FP mult/div	8	16
latência	4 ciclos / 20 ciclos	
IL1 cache (não-bloqueante)	128K	256K
associatividade	4-way	4-way
Tamanho de linha	32 bytes	32 bytes
DL1 cache (não-bloqueante)	128K	256K
associatividade	8-way	8-way
Tamanho de linha	64 bytes	64 bytes
L2 cache (unificada)	1M	2M
associatividade	8-way	8-way
Tamanho de linha	128 bytes	128 bytes
L1 hit / L2 hit	1 ciclo / 6 ciclos	
Latência memória principal	100 ciclos / 10 ciclos (primeiro bloco / próximos blocos)	

Tabela 5.4: Principais parâmetros do previsor de desvios para SMT-4 e SMT-8

Parâmetro	**Valor**
Branch Target Buffer (BTB)	512 entradas por tarefa
Bimodal Predictor	1024 entradas por tarefa
2-level Predictor	13 bit *gshare*
Meta table	1024 entradas por tarefa

única combinação é utilizada para o processador SMT-8. A Tabela 5.5 sumariza a composição desses *workloads*. Uma breve análise desses *benchmarks* foi apresentada na Seção 5.1.

Todas simulações foram executadas até que um dos *benchmarks* atingisse 600 milhões de instruções. As primeiras 100 milhões de instruções não afetam os resultados, pois são executadas em modo *fast-forward*, ou seja, sua execução apenas altera o estado da memória principal e os conjuntos de registradores. Esse artifício é utilizado para minimizar

os efeitos da inicialização das aplicações.

O desempenho das diferentes topologias foi medido através do IPC (instruções por ciclo) e de *speedups* entre as diversas configurações simuladas. As taxas de acerto na previsão de desvio não serão mostradas explicitamente, pois em arquiteturas SMT o impacto da taxa de acerto no desempenho final (IPC) não é tão direto quanto em arquiteturas superescalares, conforme apresentado em (GONÇALVES et al., 2000).

A análise da topologia distribuída e compartilhada foi a primeira etapa da avaliação do impacto de diferentes topologias na previsão de desvios. Ambas topologias foram simuladas para verificação do comportamento dos *benchmarks* quando não existem limites na quantidade de consultas e atualizações nas tabelas de previsão.

A Figura 5.5 mostra como a métrica IPC (Instruções por ciclo) é afetada por essas topologias no processador SMT-8. A última coluna, *soma*, está apresentada em uma escala diferente das demais, somente para ilustrar o IPC total do processador em cada um dos casos.

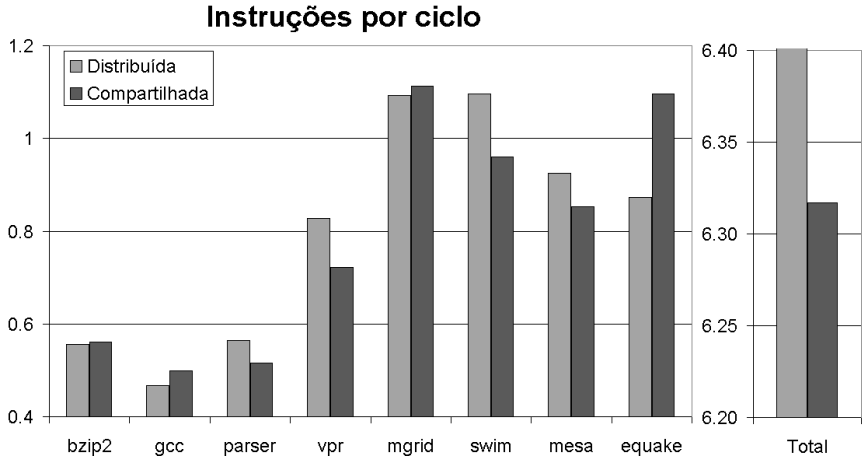

Figura 5.5: Efeitos da topologia distribuída e compartilhada no SMT-8

Embora a replicação de recursos apresente os melhores resultados para o desempenho médio de processadores SMT, alguns *benchmarks*, como o *bzip2*, *gcc*, *equake* e *mgrid*, beneficiam-se de uma topologia compartilhada. Esses *benchmarks* tendem a ocupar todos recursos disponíveis para previsão de desvios, podendo até mesmo subtrair a quantidade de recursos disponíveis para os demais aplicativos. Mesmo apresentando resultados individuais melhores em alguns casos, o desempenho médio da topologia compartilhada é 3% pior que desempenho da topologia distribuída.

Tabela 5.5: *Workloads* para SMT-4 e SMT-8

Processador	**Workloads**	
SMT-4	1	*bzip2, gcc, mgrid, mesa*
	2	*bzip2, gcc, swim, equake*
	3	*parser, vpr, mgrid, mesa*
	4	*parser, vpr, swim, equake*
SMT-8	*bzip2,gcc,parser,vpr,mgrid,mesa,swim,equake*	

Os resultados mais surpreendentes vem do *benchmark equake*, onde a melhoria de desempenho da topologia compartilhada supera em mais de 25% o da distribuída. O número de desvios executado por esse *benchmark* salta de 49 para 56 milhões de instruções, o que confirma que o *equake* pode acabar consumindo os recursos disponíveis para a previsão de desvios. Além disso, a taxa de acerto passa de 88% (distribuída) para aproximadamente 95% (compartilhada). O *benchmark gcc* também apresenta uma melhoria na taxa de acerto da previsão, 85% na topologia compartilhada e apenas 76% na distribuída. Os demais *benchmarks* apresentam resultados levemente superiores ou iguais aos da topologia distribuída.

5.2.1 Resultados do SMT-4

Dando seqüência aos experimentos foram realizadas simulações das diferentes topologias de previsão possíveis no processador SMT-4, utilizando os quatro *workloads* definidos anteriormente. Os resultados mostrados a seguir representam, então, o resultado médio obtido através dessas execuções.

A Figura 5.6 apresenta o IPC total do processador SMT-4 para diferentes topologias de previsão de desvios e 1, 2 e 4 módulos de *i-cache*. Aumentando o número de módulos de *i-cache* o número de instruções que podem ser buscadas também aumenta, gerando uma pressão maior sobre os módulos de previsão de desvios. O melhor desempenho é atingido com uma topologia particionada com 4 módulos, 1 banco e 4 módulos de *i-cache*, com um IPC médio de 3,18 (muito próximo ao da topologia distribuída – 3,19 IPC). Essa topologia é similar à distribuída, porém o número de consultas e atualizações que podem ser feitas em um mesmo ciclo é limitada (uma consulta e uma atualização por módulo).

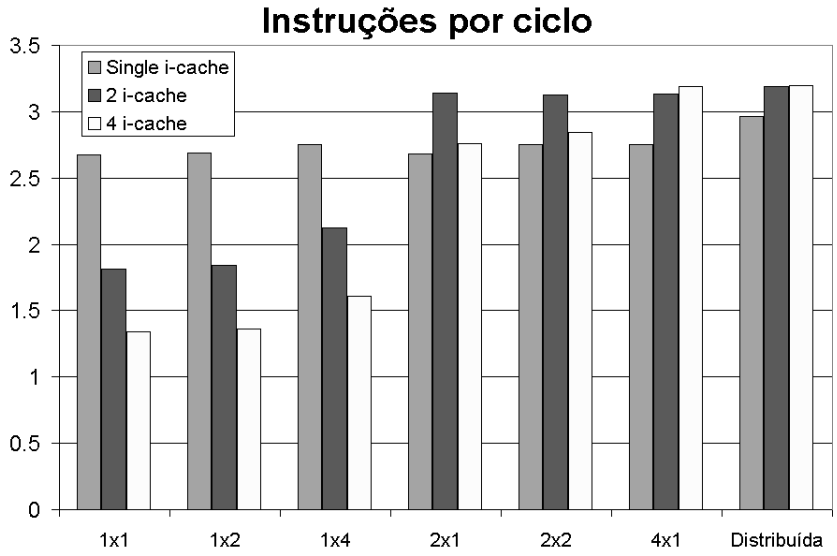

Figura 5.6: IPC total para o SMT-4 com previsores particionados

Um importante padrão no desempenho desse processador pode ser extraído analisando, por exemplo, o comportamento dos resultados com 2 módulos de *i-cache*: quando somente 1 módulo de previsão de desvios está presente, o desempenho é 26% menor que com uma configuração semelhante de previsão de desvios e somente 1 módulo de *i-cache*. Contudo, se o número de módulos de *i-cache* é aumentado para 4, não existe uma melhoria no desempenho do processador. Esse padrão pode ser observado em todas

as simulações desse processador.

A Figura 5.7 mostra de uma maneira mais clara esse padrão. Através da análise do *speedup* sobre a configuração de 1 módulo e 1 banco (1x1) é possível observar que o máximo desempenho em topologias distribuídas é atingido quando existe uma equivalência entre o número de módulos de previsão de desvios e de *i-cache*. O maior *speedup* é atingido com uma configuração de 4 módulos e 1 banco (4x1), com 2,24 sobre a configuração de 1 módulo e 1 banco. Assim, o desempenho aumenta tanto com a distribuição dos recursos – opostamente ao compartilhamento – como com o aumento da largura de busca.

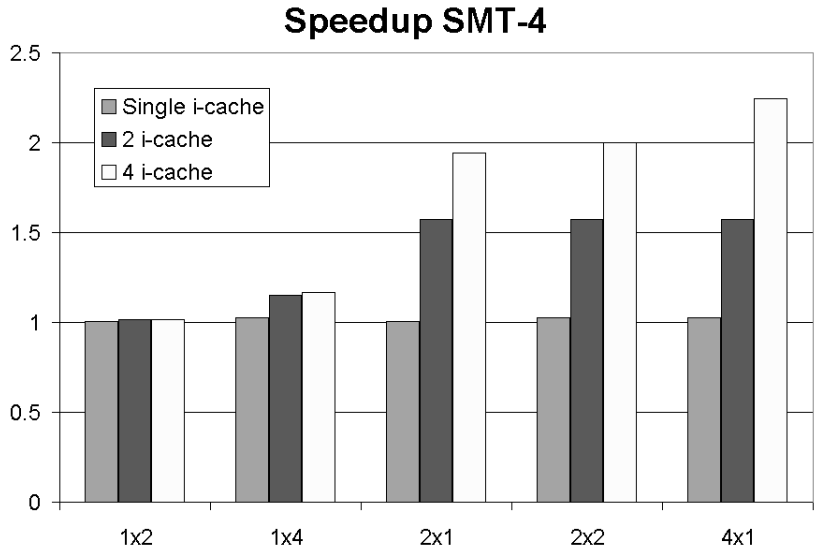

Figura 5.7: *Speedup* sobre 1x1 para SMT-4

5.2.2 Resultados do SMT-8

O conjunto de *benchmarks* utilizados nas simulações de processador SMT-4 nunca inclui todos os *benchmarks* disponíveis ao mesmo tempo, uma vez que somente 4 tarefas podem estar executando simultaneamente. Dessa forma, mesmo tendo resultados significativos com SMT-4, é necessário analisar o comportamento de uma arquitetura capaz de executar os 8 *benchmarks* simultaneamente, a SMT-8.

A Figura 5.8 mostra o IPC total para o processador SMT-8 utilizando diferentes topologias de previsão de desvios e variando o número de módulos de *i-cache*. Um único módulo de *i-cache* acaba limitando o desempenho de toda arquitetura, devido a restrições de largura de busca, atingindo um desempenho máximo de 3,8 IPC. Contudo, existe uma pequena melhoria de 3% quando a configuração passa de 1 módulo e 1 banco para 1 módulo e 8 bancos. Isso demonstra que mesmo quando o número máximo de consultas e atualizações em um mesmo ciclo é mantido constante, a distribuição de recursos continua sendo uma boa opção em processadores SMT.

A Figura 5.9 apresenta o *speedup* para o processador SMT-8 sobre a configuração base de 1 módulo e 1 banco. Ela mostra o mesmo padrão apresentado pelo processador SMT-4, onde o desempenho máximo é atingido através do uso do maior número de módulos de *i-cache*, juntamente com o mesmo número de módulos de previsão de desvios. Os valores totais são superiores nesse processador, pois existe praticamente o dobro de

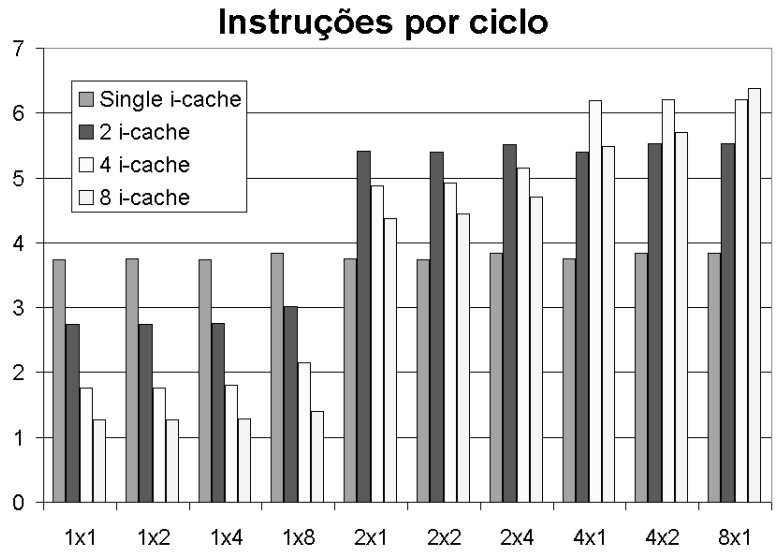

Figura 5.8: IPC total para o SMT-8 com previsores particionados

recursos disponíveis em relação ao SMT-4.

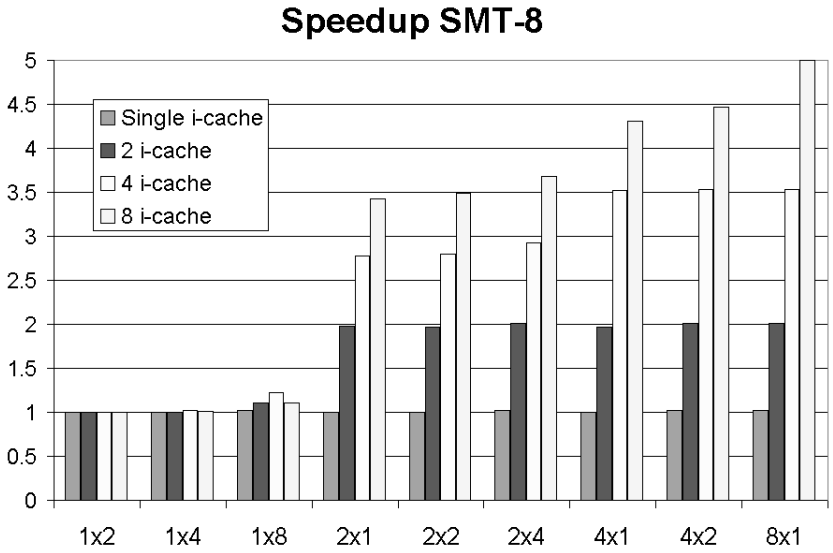

Figura 5.9: *Speedup* sobre 1x1 para SMT-8

A Figura 5.10 apresenta o IPC individual de cada *benchmark* no processador SMT-8 com cinco diferentes topologias e 8 módulos de *i-cache*. As topologias distribuída e compartilhada não limitam o número de consultas e atualizações que podem ser feitas em um mesmo ciclo. Alguns *benchmarks*, como *bzip2* e *mgrid*, apresentam um desempenho semelhante em todas as configurações, enquanto outros *benchmarks*, como *gcc* e *equake* são favorecidos pelo compartilhamento de estruturas. O *benchmark vpr* requer mais acessos às estruturas de previsão do que pode ser obtido com uma topologia de 4 módulos, com um IPC de apenas 34% em relação aos melhores resultados. Além disso, pode ser notado que a maioria dos *benchmarks* apresentam os melhores desempenhos (levando em conta as restrições no número de acessos que ocorrem em processadores reais) com mais

recursos distribuídos (como no caso da topologia 8x1), confirmando os resultados obtidos para o processador SMT-4.

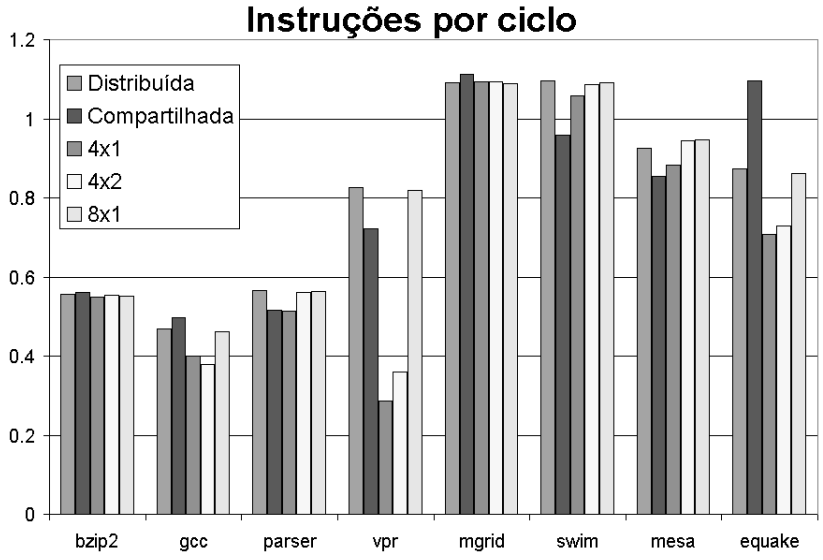

Figura 5.10: IPC para todos *benchmarks* (SMT-8)

5.2.3 Sumário

Os resultados apresentados acima mostram que a melhor relação entre desempenho e quantidade de recursos (número de módulos de previsão de desvios e de *i-cache*) é atingida quando a quantidade de módulos é igual. Dessa forma, não existe um ganho expressivo no desempenho ao aumentar apenas um dos parâmetros isoladamente. O pequeno ganho obtido não justifica o aumento da complexidade no *hardware*. Os resultados do processador SMT-4 também apontam que uma topologia com apenas 2 módulos de previsão de desvios pode prover um bom desempenho, quando comparado com os ganhos de uma configuração muito mais complexa – 4 módulos. O desempenho do processador SMT-4 com 2 módulos de *i-cache* e de previsão de desvios é apenas 3% inferior à melhor configuração particionada nesse processador. Além disso, mesmo quando 4 módulos de *i-cache* são utilizados no SMT-4, a alternativa particionada 2x2 pode ser uma boa escolha, seu desempenho é apenas 11% inferior à topologia 4x1. O mesmo comportamento também pode ser observado no processador SMT-8.

O particionamento das estruturas de previsão de desvios também é interessante do ponto de vista de freqüência de *clock*. Simulando as configurações de BTB utilizadas nesse trabalho com a ferramenta CACTI (SHIVAKUMAR; JOUPPI, 2001) e uma tecnologia de 0.80 μm obtém-se que a configuração de BTB compartilhada é 35% mais lenta que uma configuração sem módulos.

A contenção nos barramentos de previsão de desvios não pode ser ignorada. Em todas as configurações, o desempenho é reduzido quando move-se de uma topologia distribuída para outras particionadas. Essa redução é mínima quando usa-se o mesmo número de módulos de *i-cache* e de previsão de desvios, variando de 1% no processador SMT-8 com 8 módulos (melhor desempenho) à 12% no processador SMT-8 com apenas 1 módulo (pior desempenho).

5.3 Previsão de desvios e a profundidade do *pipeline*

O simulador *ss-smt* foi estendido para permitir a configuração do número de estágios do *pipeline*. Essa modificação também foi realizada no simulador *sim-outorder* original, permitindo, assim, uma comparação entre o comportamento de arquiteturas superescalares e multitarefas simultâneas. Para realizar essa análise, três processadores distintos foram simulados:

- um processador superescalar (SS) capaz de buscar e executar até 8 instruções por ciclo, representando processadores comerciais disponíveis atualmente;
- um processador capaz de executar quatro tarefas independentes (SMT-4);
- um processador capaz de executar oito tarefas independentes (SMT-8);

A configuração de cada um deles está sumarizada na Tabela 5.6. Os processadores SMT simulados tem como base o processador superescalar, ou seja, o SMT-4 (SMT-8) tem quatro (oito) vezes mais recursos que o processador superescalar. Os parâmetros que não estão listados são semelhantes aos encontrados na Tabela 5.3.

Os *benchmarks* utilizados nessa análise são os mesmos encontrados na Tabela 5.5, ressaltando que o processador SS executa os oito *benchmarks* como *workloads* em separado, não havendo nenhum tipo de combinação. A quantidade de instruções executadas é semelhante à análise de topologias de previsão (ver Seção 5.2).

Tabela 5.6: Principais parâmetros para SS, SMT-4 e SMT-8

Parâmetro	**SS**	**SMT-4**	**SMT-8**
Largura do *Pipeline*	8	32	64
RUU (por tarefa)	256	256	256
LSQ (por tarefa)	128	128	128
Portas de memória	2	8	16
Int add/sub	4	16	32
Int mult/div	2	8	16
FP add/sub	4	16	32
FP mult/div	2	8	16
IL1 cache	32K	128K	256K
associatividade	4-way	4-way	4-way
Tamanho de linha	64 bytes	64 bytes	64 bytes
DL1 cache	32K	128K	256K
associatividade	8-way	8-way	8-way
Tamanho de linha	128 bytes	128 bytes	128 bytes
L2 cache (unificada)	256K	1M	2M
associatividade	8-way	8-way	8-way
Tamanho de linha	128 bytes	128 bytes	128 bytes

Para cada um dos processadores foram executadas simulações variando-se a taxa de acerto da previsão de desvios e o número de estágios do *pipeline*. A taxa de acerto da previsão foi variada entre 50 e 100%, com saltos de 10% e incluindo a taxa de 95%.

Essa taxa foi incluída para verificar com um pouco mais de detalhe o comportamento das arquiteturas com taxas de acerto mais próximas de um previsor perfeito. Os estágios do *pipeline* foram variados da seguinte forma:

- 6 estágios, representando a arquitetura base encontrada originalmente na arquitetura do simulador *sim-outorder*;
- 12 estágios, representando alguns processadores mais recentes;
- 18 estágios, representando processadores mais agressivos em termos de profundidade de *pipeline* e freqüência de *clock*.

A primeira etapa da avaliação consistiu na análise do desempenho das arquiteturas em termos de IPC, variando-se a taxa de acerto da previsão de desvios e também o número de estágios do *pipeline*. O comportamento de todas arquiteturas é bastante semelhante, atingindo os melhores desempenhos com seis estágios de *pipeline* e previsão perfeita, conforme mostra a Figura 5.11. O processador superescalar atingiu o desempenho máximo de 2,45 IPC, enquanto os processadores SMT-4 e SMT-8 atingiram 11,9 e 24,0 IPC, respectivamente. O desempenho dos processadores multitarefas superam o do superescalar em pouco mais que a relação de números de tarefas, ilustrando, mais uma vez, o potencial de exploração de paralelismo desse tipo de arquitetura.

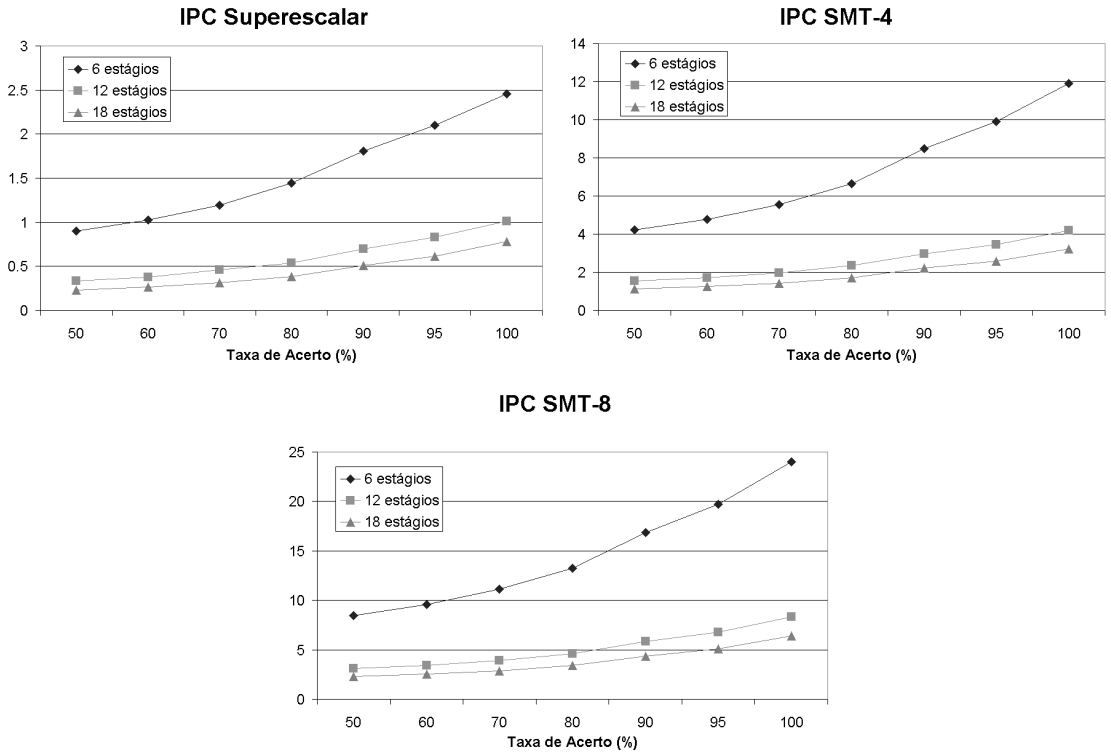

Figura 5.11: IPC variando taxas de acerto e profundidade do *pipeline*

Através dessa mesma figura pode-se notar que o desempenho da arquitetura superescalar com 6 estágios e previsão com taxas de acerto superiores a 95% é semelhante ao da arquitetura multitarefa com 4 tarefas em execução e uma taxa de acerto entre 70 e 80% e 12 estágios de *pipeline*. É importante ressalvar que a arquitetura superescalar é menos complexa que uma arquitetura SMT capaz de executar 4 tarefas e, além disso, taxas de

acerto superiores a 95% já são possíveis atualmente. Essa mesma analogia pode ser feita entre as arquiteturas SMT-4 e SMT-8. O desempenho da arquitetura SMT-8 com uma taxa de acerto de pouco mais de 70% e 12 estágios é equivalente ao da arquitetura SMT-4 com a mesma profundidade de *pipeline* e previsor com mais de 95% de taxa de acerto. Da mesma forma, o desempenho de um mesmo processador com *pipeline* menos profundo pode ser obtido através do uso de um *pipeline* mais profundo aliado a uma taxa de acerto mais alta. Essa alternativa pode ser explorada para que o IPC seja mantido ao aumentar a profundidade de *pipeline*.

É importante lembrar que o desempenho final de um processador não é só composto pela métrica de instruções por ciclo (IPC), mas pela multiplicação desse valor pelo tempo de ciclo. Esse novo valor gera a métrica conhecida por MIPS, ou milhões de instruções por segundo. A diminuição do IPC devido ao maior número de estágios do *pipeline* tende a ser compensada pelo aumento na freqüência de funcionamento do processador. Esse aumento de freqüência deve ser de ordem superior à queda no IPC para que a quantidade de instruções executadas por segundo (MIPS) seja maior. A análise do aumento do número de estágios na freqüência de funcionamento do processador não é o foco desse trabalho.

A partir dos resultados do desempenho das arquiteturas ilustrado na Figura 5.11, foi realizado uma análise da importância do previsor de desvio no ganho de desempenho com diferentes profundidades de *pipeline*. A Tabela 5.7 mostra o *speedup* de cada uma das arquiteturas variando-se a taxa de acerto da previsão de desvios de 90 para 100%. É possível notar que em todos os casos sempre há um aumento significativo no desempenho dos processadores. Esse aumento fica mais evidente em *pipelines* profundos, como mostra a Tabela 5.7. Na arquitetura superescalar esse aumento varia de 36% para 6 estágios e atinge o ganho máximo de 52% com 18 estágios. Na arquitetura SMT os ganhos são menos expressivos, variando de 40% a 45% para SMT-4 e 42% a 46% na SMT-8.

O aumento do IPC nesses casos deve-se principalmente à diminuição do número de ciclos perdidos executando instruções que serão descartadas posteriormente. *Pipelines* profundos tendem a aumentar o número mínimo de ciclos que uma instrução permanece em execução especulativa. Os *speedups* são mais constantes nas arquiteturas SMT devido à capacidade dessas arquiteturas de executar instruções provenientes de tarefas que não estejam sendo afetadas pelas falhas na previsão de desvios, como por exemplo o *benchmark mgrid*.

Tabela 5.7: *Speedups* para Superescalar / SMT-4 / SMT-8

Estágios	IPC 90%	IPC 100%	*Speedup* (%)
6	1,80	2,45	36%
	8,48	11,90	40%
	16,87	24,0	42,1%
12	0,69	1,01	45%
	2,97	4,20	41%
	5,86	8,35	42,2%
18	0,51	0,77	52%
	2,22	3,22	45%
	4,37	6,42	46%

A quantidade de paralelismo existente para ser explorada também pode influenciar o desempenho das arquiteturas com *pipelines* profundos. Para ilustrar esse efeito, foi analisado o desempenho dos três processadores com um previsor de desvios perfeito. Com isso, garante-se que os efeitos da quebra do fluxo de instruções serão minimizados, focando a análise na quantidade de paralelismo que a arquitetura consegue extrair das aplicações disponíveis.

Através da Figura 5.12 pode-se extrair dois comportamentos distintos: as arquiteturas multitarefas tendem a se beneficiar da exploração do paralelismo intra-tarefa, bem como inter-tarefa, atingindo em média 35% (12 estágios) e 27% (18 estágios) do desempenho da arquitetura base de 6 estágios. Por sua vez, a arquitetura superescalar, limitada a exploração do ILP, atinge desempenhos proporcionalmente superiores às arquiteturas SMT, com um desempenho de 41% (12 estágios) e 32% (18 estágios). Dessa forma, nota-se que a quantidade de paralelismo a ser explorado influi reversamente na profundidade do *pipeline*.

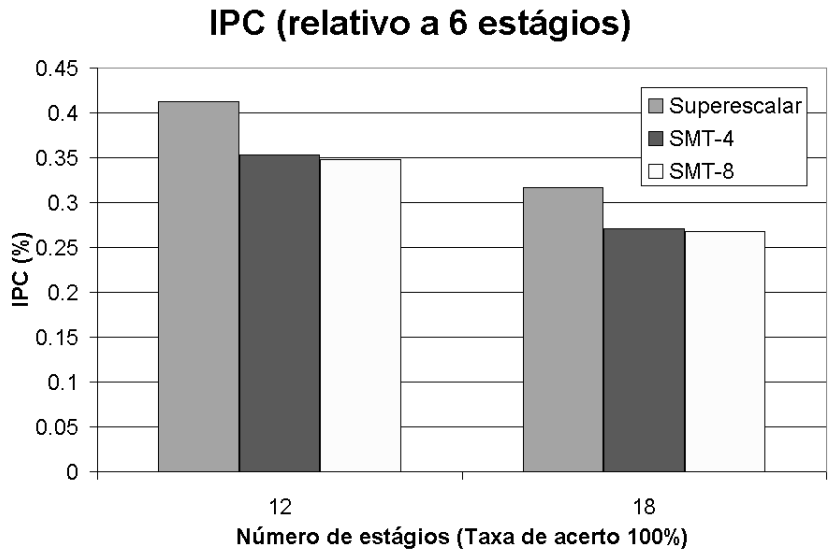

Figura 5.12: Paralelismo e a profundidade do *pipeline*

Essa mesma tendência pode ser extraída da Tabela 5.7. Ao aumentar a taxa de acerto do previsor de desvio, aumenta-se a quantidade de paralelismo que pode ser explorado por ambas arquiteturas. Porém, com 6 estágios o *speedup* da arquitetura superescalar é inferior à das arquiteturas SMT. Com o aumento da profundidade do *pipeline* essa tendência é revertida, ou seja, as arquiteturas SMT tem menores *speedups* em relação à arquitetura superescalar.

Outro ponto importante é a verificação da efetividade das arquiteturas com *pipelines* profundos, ou seja, a relação entre a quantidade de instruções realmente executadas e a quantidade de instruções que foram graduadas. A efetividade de uma arquitetura é dependente da taxa de acerto do previsor de desvio, pois quanto mais erros de previsão maior será o número de instruções executadas e, posteriormente, descartadas. A adição de mais estágios entre os pontos de consulta ao previsor de desvio, geralmente no estágio de busca, e o ponto de atualização do previsor de desvios, geralmente no estágio de escrita ou retirada das instruções, tende a maximizar o número de instruções executadas especulativamente, diminuindo a efetividade da arquitetura.

A Figura 5.13 mostra a efetividade dos processadores SS, SMT-4 e SMT-8 com 6 estágios de *pipeline*. Nessa configuração a arquitetura superescalar é a de menor efetividade em todos os casos. Com um previsor de desvio capaz de acertar 95% de suas previsões a arquitetura superescalar atinge apenas 70% de efetividade, contra, em média, 85% das arquiteturas SMT.

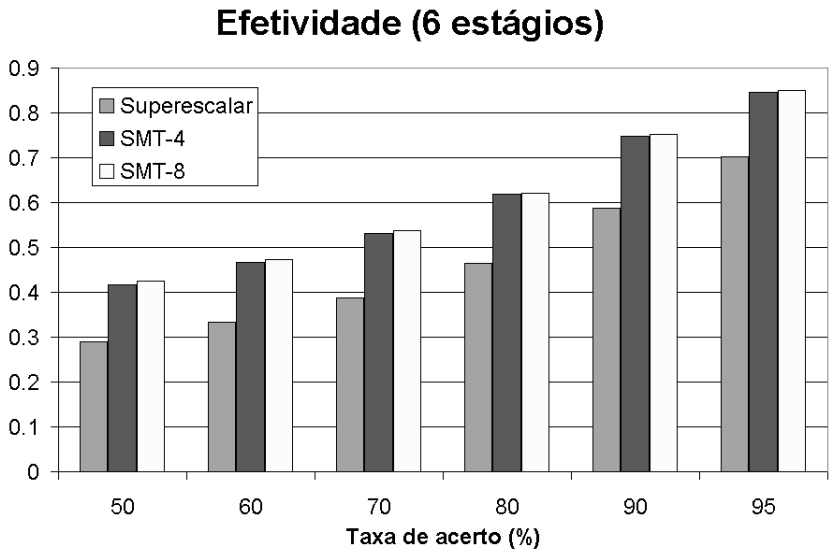

Figura 5.13: Efetividade do *pipeline* de 6 estágios

Ao aumentar a quantidade de estágios no *pipeline* a efetividade da arquitetura superescalar é aumentada e a da arquitetura multitarefa diminuída. Isso pode ser observado na Figura 5.14. A efetividade do processador SMT-8, por exemplo, que era de aproximadamente 85% com 6 estágios passa a ser de 81% com 12 estágios e 79% com 18 estágios. O mesmo comportamento pode ser extraído para o processador SMT-4. A arquitetura superescalar, por sua vez, atinge a efetividade máxima de 80% com 12 estágios, muito próxima dos processadores multitarefas. Com 18 estágios a efetividade cai para 78%. Dessa forma, do ponto de vista de efetividade, a profundidade ótima para o *pipeline* dessa arquitetura estaria entre 12 e 18 estágios.

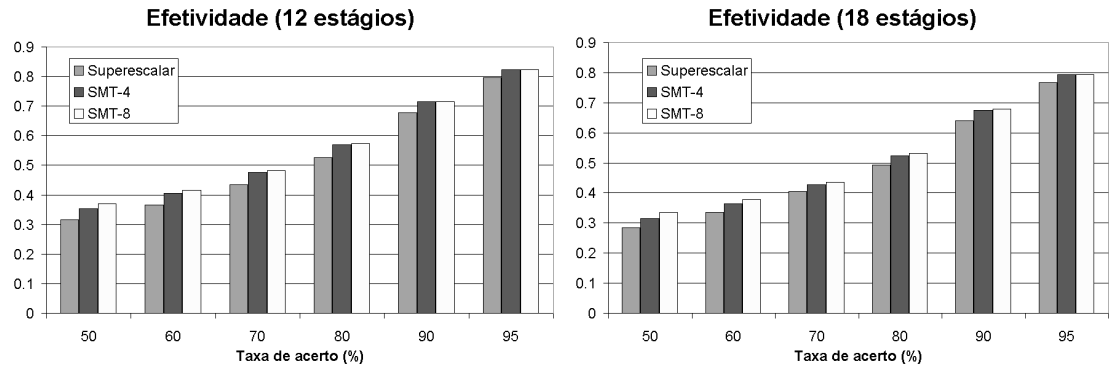

Figura 5.14: Efetividade dos *pipelines* de 12 estágios e 18 estágios

É importante ressalvar que a arquitetura superescalar atinge sempre as menores taxas de efetividade, comprovando a melhor utilização dos recursos disponíveis por parte das

arquiteturas SMT. As arquiteturas SMT conseguem atingir maiores níveis de efetividade devido a sua capacidade de exploração dos dois tipos de paralelismo, ILP e TLP. Quando não existe ILP disponível para ser explorado, a arquitetura SMT passa a explorar mais ativamente o TLP.

5.4 Previsão de desvios com taxa de acerto variável na BTB

A segunda etapa dos mecanismos de previsão é geralmente realizada através de um acesso à tabela BTB, a qual informa, no caso de desvios previstos como tomados, o endereço que deverá ser utilizado para os próximos acessos realizados pelo estágio de busca.

Um erro nessa etapa do mecanismo pode acabar trazendo instruções de caminhos errados para execução no *pipeline*, podendo ocupar recursos necessários à execução das instruções do caminho correto, como por exemplo, entradas nas *caches* de dados e instruções. Em uma arquitetura SMT esse efeito pode ser potencializado, devido a interferência que uma tarefa pode causar nas demais em execução.

Dessa forma, conforme descrito na Seção 4.5, foi desenvolvido um mecanismo capaz de simular taxas de acerto variável no acesso a uma BTB nos simuladores *ss-smt* e *sim-outorder*. Através desse mecanismo foi analisado o impacto de diferentes taxas de acerto na previsão do alvo de desvios em arquiteturas superescalares (processador SS) e SMT (processadores SMT-4 e SMT-8), verificando o impacto no desempenho final e no estágio de busca das mesmas. Nas simulações realizadas o previsor de direção é perfeito, ou seja, apenas a taxa de acerto do previsor de alvo (BTB) foi variada, de 50 a 100% em passos de 10%, incluindo a taxa de acerto de 95%. Na ocorrência de erro na previsão da BTB, 10% delas são causadas por endereço não encontrado (*miss*).

Para a análise do impacto no desempenho final das arquiteturas a métrica de instruções por ciclo (IPC) foi utilizada. Já para o desempenho do estágio de busca, foi utilizado a quantidade de ciclos em que a busca de instruções não pode ser realizada, seja por falhas na *cache*, erros de previsão (redirecionamento da busca) ou pela falta de espaço nas filas de busca (fila de busca cheia).

Os *benchmarks* utilizados são os mesmos das Seções anteriores, assim como a configuração dos processadores utilizados nessa análise é idêntica àquela utilizada na Seção 5.3, com a exceção da quantidade total da *cache* de instruções, a qual teve seu tamanho total diminuído para a base de 8 Kbytes por tarefa. Essa diminuição foi realizada devido a incapacidade de estresse do sistema de memória dos *benchmarks* do SPEC2000 para tamanhos atuais de *caches* (SAIR; CHARNEY, 2000). O processador SMT-8 utiliza na *cache* de instruções, dois módulos com dois bancos, devido ao maior desempenho que pode ser obtido através dessa configuração (GONÇALVES et al., 2000).

Devido ao modelo de erro utilizado no mecanismo de taxa de acerto variável, cada experimento foi repetido diversas vezes e os valores apresentados a seguir representam uma média de todas as execuções. Isso foi necessário devido a utilização de funções randômicas para decisão de quais alvos seriam utilizados no erro ao acesso da BTB. Cada simulação utilizou uma semente de randomização distinta. Sementes idênticas geram padrões de acesso iguais, podendo ser utilizadas para garantir a repetibilidade dos resultados.

5.4.1 Processador Superescalar

Inicialmente, foi analisado o comportamento de cada um dos *benchmarks* individualmente no processador superescalar. O IPC de cada um deles ao variar a taxa de acerto

na BTB pode ser visto na Figura 5.15. O único *benchmark* que atinge mais de 1 IPC é o *mgrid*, o qual não é muito afetado pela previsão de desvios (*speedup* de apenas 3%), pois é o que menos possui instruções de desvio, em média uma em cada 30 instruções executadas. Seu desempenho foi limitado basicamente pela quantidade de recursos (unidades funcionais). Os demais *benchmarks* tem um aumento de desempenho médio de 22%, variando entre 18% para o *vpr* e 35% para o *swim*.

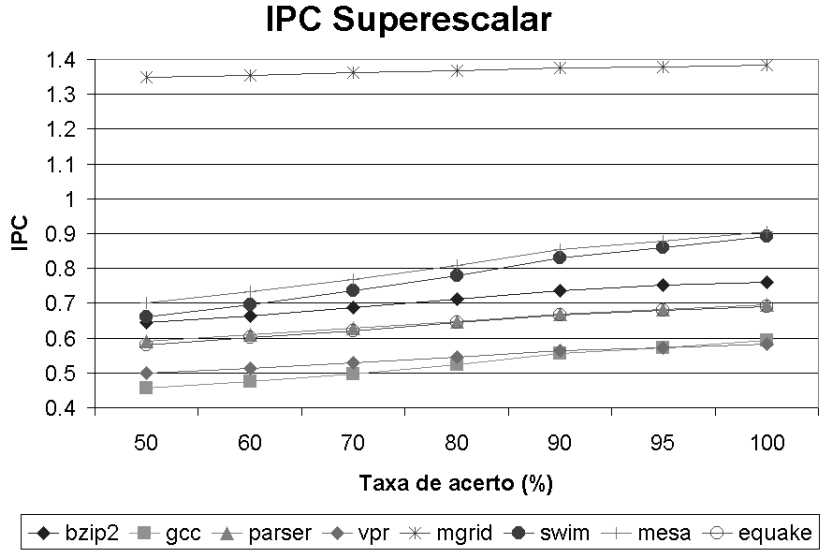

Figura 5.15: IPC dos *benchmarks* SPEC2000 no processador superescalar variando a taxa de acerto na BTB

Para verificar as razões da melhoria no desempenho dos *benchmarks* com o aumento da taxa de acerto foi analisado a quantidade de ciclos em que o estágio de busca estava bloqueado. A Figura 5.16 mostra para cada *benchmark* a quantidade de ciclos em relação ao tempo total de simulação em que o estágio de busca estava bloqueado, não podendo fornecer instruções para os demais estágios do *pipeline*.

O *benchmark swim*, por exemplo, teve o número de ciclos sem busca diminuído de 26% com uma taxa de acerto de 50% para apenas 6% com o uso do previsor perfeito. Essa queda refletiu diretamente no desempenho do *benchmark*, passando de 0,66 para 0,89 IPC, ou seja, *speedup* de aproximadamente 35%.

A Tabela 5.8 mostra a contribuição percentual de cada uma das causas de ciclos sem busca para três taxas de acerto: 50, 80 e 100% (previsão perfeita). As colunas da tabela com valores em negrito indicam a causa que possui a maior representatividade na quantidade de ciclos sem busca. Cada coluna tem o seguinte significado:

Fc: fila de instruções cheia;

Cb: *cache* de instruções estava bloqueada, tratando *miss*;

Mp: estágio de busca bloqueado devido à ocorrência de *misprediction*;

Mf: estágio de busca bloqueado devido à ocorrência de *misfetch*.

É possível notar que a taxa de acerto no alvo do desvio impacta diretamente em todas as causas de bloqueio da busca. Para os *benchmarks bzip2* e *mgrid*, a principal causa de

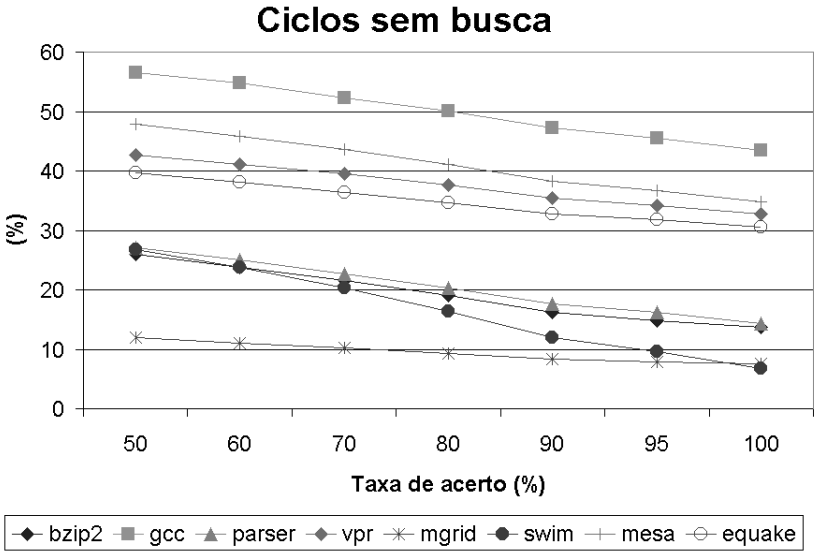

Figura 5.16: Ciclos sem busca para os *benchmarks* SPEC2000 no processador superescalar variando a taxa de acerto na BTB

Tabela 5.8: Principais causas de parada do estágio de busca

Benchmarks	Taxa de Acerto											
	50				80				100			
	Fc	Cb	Mp	Mf	Fc	Cb	Mp	Mf	Fc	Cb	Mp	Mf
bzip2	**11**	3,7	0,9	9,9	**12,1**	2,1	0,04	4,3	**13,2**	0,3	0	0
gcc	3,5	**45**	1	7	3,8	**42,4**	0,04	3,2	4,2	**39,1**	0	0
parser	2,3	**12,8**	1,3	10	3,1	**12**	0,05	4,5	3,7	**10,6**	0	0
vpr	0,3	**32,4**	1,1	9,4	0,6	**31,9**	0,04	4	1	**31,7**	0	0
mgrid	**6,8**	0,09	0	5	**7,1**	0,09	0	2	**7,4**	0,1	0	0
swim	0	**13,6**	3,5	9,5	0	**10,3**	1,6	4,4	0	**6,8**	0	0
mesa	0	**37,9**	1,6	8,2	0	**36,6**	0,7	3,7	0	**34,8**	0	0
equake	0	**32,6**	0,7	7,5	0	**31,1**	0,4	4	0	**30,6**	0	0

bloqueio é a ocorrência de fila de instruções cheia. Com o aumento da taxa de acerto a ocorrência de fila de instruções cheia também aumenta, pois mais instruções estarão sendo trazidas para execução e, caso as unidades de execução não sejam suficientes, as instruções não poderão ser despachadas. Note que a ocorrência de fila cheia não pode ser considerada como um problema nesses casos, uma vez que mostra que a busca está sendo mais efetiva do que a execução das instruções.

O comportamento dos demais *benchmarks* é semelhante do ponto de vista de ocorrência de fila cheia, porém, a maior contribuição para os ciclos sem busca provem da ocorrência de *cache* de instruções bloqueada. Contudo, com o aumento da taxa de acerto do previsor do alvo, a ocorrência de bloqueio na *cache* de instruções diminui, pois, geralmente na ocorrência de erro de previsão pelo menos um acesso extra à *cache* de instruções é necessário. É importante notar que esse acesso extra pode não acontecer nos casos em que o alvo do desvio (certo ou errado) já se encontra na *cache*.

As duas outras causas de bloqueio (*misfetch* e *misprediction*) do estágio de busca também tem seu efeito diminuído com o aumento da taxa de acerto do previsor, ressalvando que com o previsor perfeito essas duas causas são nulas. A porcentagem de bloqueios do estágio de busca devido a essas duas causas tem uma diminuição de, em média, mais de 50% ao aumentar a taxa de acerto do previsor de 50 para 80%. A maior contribuição do *misfetch* em todos os casos é proveniente do mecanismo sendo utilizado, juntamente com a maior quantidade de desvios diretos encontrados nos *benchmarks*. Sem erros na direção, o estágio de decodificação sempre será capaz de detectar para desvios diretos os erros gerados pelo mecanismo. Isso pode ser facilmente comprovado verificando o comportamento do *benchmark mgrid*, o qual contém mais de 99% de desvios diretos e raramente sofre bloqueios por *misprediction*.

Finalmente, a Figura 5.17 valida o mecanismo de escolha da causa do erro no acesso à BTB, ilustrando a diferença causada pela ocorrência de *miss* na BTB e pelo erro no alvo do desvio. Nesse experimento foi fixado a taxa de acerto de 50% na BTB e foi variado entre 10, 30, 50, 80 e 100% a porcentagem de erros causados por *misses*. Como esperado, a ocorrência de ciclos sem busca devido à faltas na *cache* de instruções diminui ao aumentar a ocorrência de *misses*. A arquitetura simulada sempre segue o caminho não-tomado nesses casos, diminuindo a chance de ocorrer uma falha no acesso à *cache* de instruções.

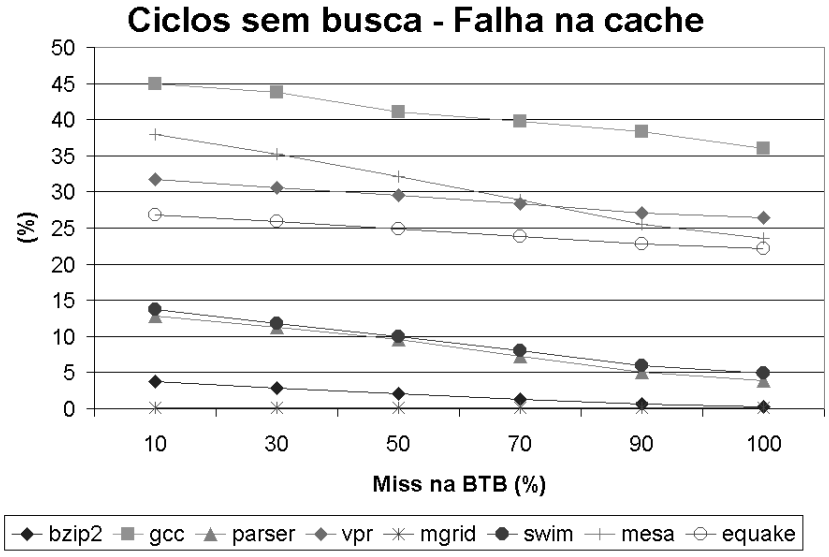

Figura 5.17: Ciclos sem busca devido a falhas na *i-cache* variando a taxa de endereços não encontrados na BTB

5.4.2 Processadores SMT

Experimentos semelhantes àqueles executados para o processador superescalar foram realizados para os processadores SMT, com o objetivo de verificar se o impacto na variação da taxa de acerto do alvo do previsor é semelhante nessas arquiteturas.

Verificando o IPC do processador SMT-4 na Figura 5.18, nota-se que, com o aumento da taxa de acerto, existe um respectivo aumento no desempenho. O IPC para esse processador é de aproximadamente 2,7 com uma taxa de acerto de 50%. Com a previsão perfeita o IPC passa a ser de 3,3, ou seja, um aumento de 22%. Ao verificar o número de ciclos sem busca para esse processador foi possível notar que ele era menor que 1%. O

número de ciclos sem busca para esse processador é mínimo pois sua implementação é baseada no uso de *caches* não-bloqueantes e, dessa forma, para que em determinado ciclo não haja busca é necessário que as 4 tarefas em execução estejam bloqueadas, seja por falhas na *cache*, fila de busca cheia ou bloqueio devido a *misprediction* ou *misfetch*. Um processador SMT tem um potencial maior que um processador superescalar no uso de tais *caches*, pois é capaz de fornecer instruções provenientes de fluxo diferentes daquelas que estão causando falhas, bastando selecionar outra tarefa para acessar a *cache*.

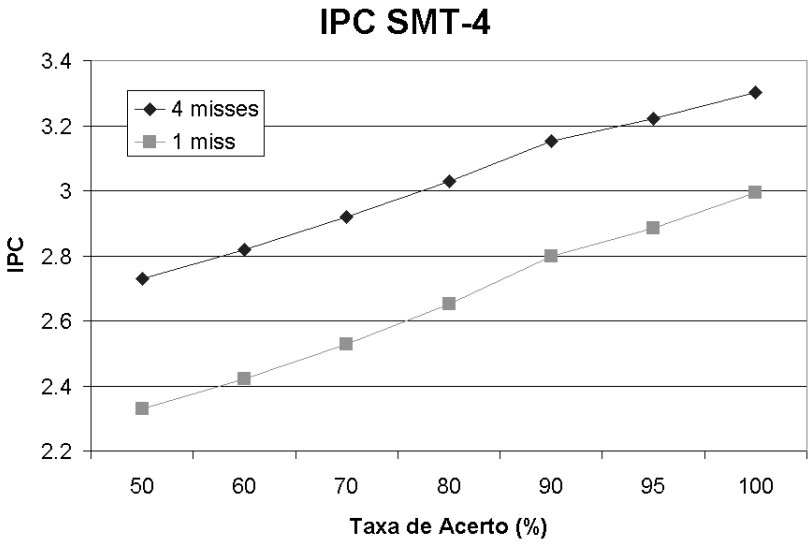

Figura 5.18: IPC do processador SMT-4 variando a taxa de acerto na BTB

Dessa forma, o simulador *ss-smt* foi modificado para não utilizar *caches* não-bloqueantes. Com isso, espera-se que o impacto da falha na *cache* de instruções por uma das tarefas acabe influindo no desempenho das demais. O IPC do processador SMT-4 sem uso de *caches* não-bloqueantes pode ser visto também na Figura 5.18, na seqüência *1 miss*. Como esperado o desempenho é diminuído, com um IPC de 2,35 para uma taxa de 50% de acerto e de 3 IPC para o previsor perfeito, resultando em um *speedup* de 27%.

A diferença no desempenho com o uso de *caches* não-bloqueantes (seqüência 4 *misses*) e no uso de *caches* bloqueantes diminui com o aumento na taxa de acerto do previsor do alvo do desvio. Para a taxa de acerto de 50% a diferença entre os IPCs é de 14%, enquanto com o previsor perfeito essa diferença cai para apenas 10%. Essa diminuição é conseqüência da diminuição no número de falhas na *i-cache*. Com menos falhas, o bloqueio de tarefas também diminui, aumentando o número de ciclos em que o estágio de busca fornece instruções para os demais estágios.

O mesmo comportamento é obtido no processador SMT-8, reforçando o comportamento obtido para o processador SMT-4, apenas ressaltando que o desempenho é superior devido a maior quantidade de recursos disponíveis nesse processador.

Com uma taxa de acerto de 50%, o processador SMT-8 atinge um desempenho de aproximadamente 5,5 IPC com o uso de *caches* não-bloqueantes e 4,8 para *caches* bloqueantes. Com o previsor perfeito o desempenho passa para 6,6 e 6,1, respectivamente. O *speedup* é de 20% para *caches* não-bloqueantes e 27% para *caches* bloqueantes. A diferença de desempenho entre ambas também diminui com o aumento da taxa de acerto, passando de 14% para 8%, resultado semelhante ao processador SMT-4.

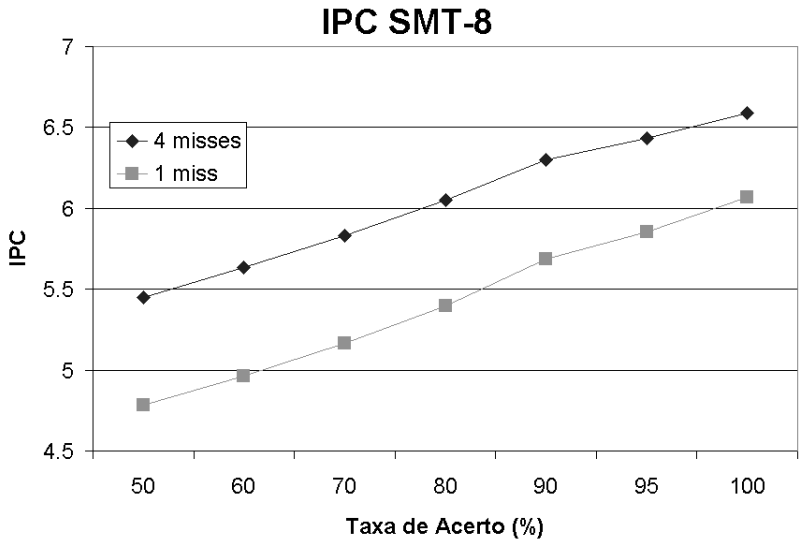

Figura 5.19: IPC do processador SMT-8 variando a taxa de acerto na BTB

O número de ciclos sem busca para o processador SMT-8 e *caches* bloqueantes é mostrado na Figura 5.20. O número de ciclos sem busca para *caches* não-bloqueantes não é mostrado por também ser inferior a 1%, mesmo com uma taxa de acerto de 50%.

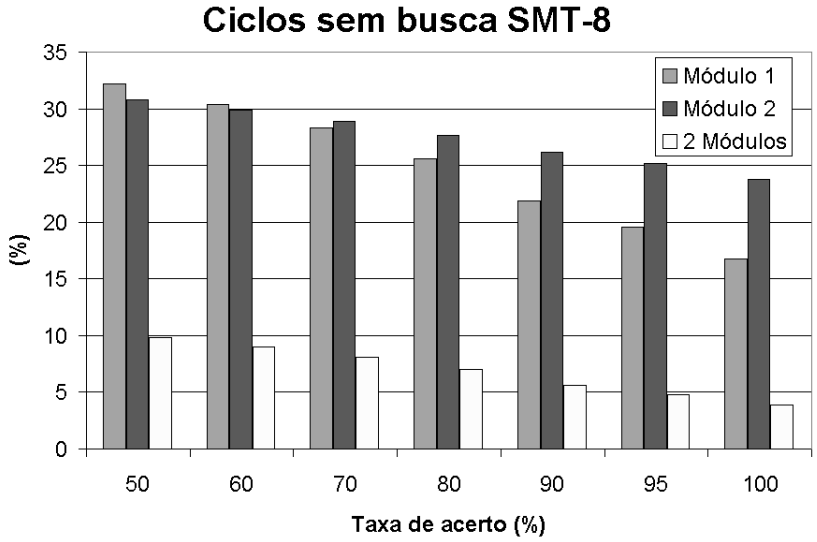

Figura 5.20: Ciclos sem busca para o processador SMT-8 variando a taxa de acerto na BTB

Conforme comentado anteriormente, no processador SMT-8 foi utilizado dois módulos de *i-cache*, devido a maior potencialidade de ganho no desempenho. Dessa forma, a Figura 5.20 mostra a ocorrência de ciclos sem busca em cada um desses sub-estágios e também a quantidade de ciclos sem busca nos dois sub-estágios simultaneamente.

O número de ciclos sem busca diminui com o aumento da taxa de acerto, assim como na arquitetura superescalar. Note-se porém que, para que não haja busca efetiva, é necessário que em um mesmo ciclo os dois sub-estágios estejam bloqueados. Dessa forma,

existe uma grande diferença entre o número de ciclos em que os dois sub-estágios estão bloqueados simultaneamente para o número de ciclos em que cada um deles está bloqueado. Para uma taxa de 50% de acerto tem-se em média 30% de ciclos sem busca em cada um dos sub-estágios e apenas 10% para os dois simultaneamente. Com o uso de um previsor perfeito o número de ciclos sem busca cai para apenas 4%.

O comportamento de cada tarefa analisada em separado, nos processadores SMT, é semelhante ao da tarefa quando executada no processador superescalar, ou seja, com o aumento da taxa de acerto do previsor de alvo, os indicadores de ciclos bloqueados por falhas na *i-cache*, *mispredictions* e *misfetches* diminuem seu impacto.

6 CONCLUSÕES E TRABALHOS FUTUROS

As arquiteturas multitarefas simultâneas (SMT), apesar de poderem ser consideradas uma expansão das arquiteturas superescalares, trazem novos desafios para os projetistas de *hardware*, pois é necessário que o desempenho de uma única tarefa não seja afetado e que o desempenho de múltiplas tarefas simultaneamente seja condizente com o esforço despendido.

Essa alternativa vem sendo utilizada comercialmente por fabricantes de microprocessadores, conforme comentado no Capítulo 3. Contudo, nos últimos meses alguns fabricantes de *hardware* anunciaram novos processadores capazes de explorar TLP através do uso de outro tipo de arquitetura, o Multiprocessador em um *chip* (CMP). O uso de CMP em detrimento dos processadores SMT pode ilustrar a maior dificuldade no projeto dos últimos.

A simples replicação de processadores, como no caso do CMP, não resolve o problema da baixa utilização dos recursos disponíveis, ao contrário da arquitetura SMT. Dessa forma, a procura de soluções que possam simplificar o projeto de processadores SMT é fundamental para o sucesso dessa arquitetura.

Assim, essa dissertação realizou uma análise de diferentes aspectos e da importância da previsão de desvios em arquiteturas SMT, identificando possíveis simplificações para o projeto dessas arquiteturas. Primeiramente, o efeito de diferentes estratégias de compartilhamento/distribuição das estruturas de previsão de desvios foi analisado. **Mostrou-se que os melhores resultados são obtidos quando os recursos estão distribuídos de acordo com o número de tarefas em execução simultânea. Entretanto, essa abordagem tende a ser muito onerosa em termos de recursos de *hardware***. Dessa forma, foi proposto o particionamento das estruturas de previsão (registradores e tabelas) em módulos e bancos.

O particionamento das estruturas de previsão de desvios mostrou ser capaz de aliar um bom desempenho a um menor custo de implementação. Para uma arquitetura SMT capaz de executar até quatro tarefas simultâneas, o particionamento das estruturas de previsão pode atingir até 97% do desempenho máximo atingido por uma configuração totalmente distribuída. Além disso, o particionamento dessas estruturas pode diminuir o tempo de acesso necessário, bem como a complexidade do *hardware*.

Os melhores resultados com essa técnica de particionamento são obtidos quando existe um casamento entre a configuração da *cache* de instruções e do previsor de desvio, ou seja, o número de módulos de *i-cache* é idêntico ao número de módulos utilizados para a previsão de desvios. A utilização de uma quantidade menor de módulos para a previsão de desvios também mostrou-se uma boa alternativa. Por exemplo, uma arquitetura SMT com oito tarefas utilizando oito módulos de *i-cache* e apenas quatro módulos de previsão de desvios pode atingir cerca de 91% do desempenho máximo da arquitetura.

A existência de múltiplas tarefas em execução causa uma maior pressão sobre as estruturas do *pipeline*, em especial os bancos de registradores e a janela de instruções. Dessa forma, sua implementação deve ser baseada em memórias que possuem múltiplas portas de leitura e de escrita. O tempo necessário para o acesso a essas estruturas passa a gerar, então, a necessidade de utilização de *pipelines* mais profundos.

Embora a implementação de arquiteturas SMT necessite de *pipelines* mais profundos, elas tendem a se beneficiar de *pipelines* mais curtos, devido à sua capacidade de explorar uma maior quantidade de paralelismo, quando comparado com uma arquitetura superescalar. Além disso, a ocorrência de falhas na previsão de desvios constitui um dos principais fatores de degradação de desempenho quando o *pipeline* é aprofundado. **Através dos experimentos conduzidos, mostrou-se que, para uma arquitetura SMT capaz de executar 8 tarefas simultâneas, o aumento da taxa de acerto na previsão de 90 para 100% aumenta o desempenho da arquitetura em mais de 46%.**

A importância da taxa de acerto no acesso à BTB também foi analisado nessa dissertação e, para arquiteturas superescalares, mostrou-se que o impacto sobre o estágio de busca é significante. **Um aumento na taxa de acerto da previsão do alvo dos desvios pode reduzir em mais de 100% o número de ciclos sem busca para alguns *benchmarks*, impactando em mais de 40% no IPC da arquitetura.**

Em arquiteturas SMT esse impacto não é tão evidente devido à inteligência extra adicionada ao estágio de busca dessas arquiteturas, o qual tem a capacidade de selecionar tarefas que não estejam bloqueadas. Para aproveitar esse potencial, as arquiteturas devem fazer o uso de *caches* não-bloqueantes. A implementação dessas *caches* permite que o número de ciclos em que o estágio de busca não tem nenhuma tarefa capaz de fornecer instruções seja praticamente nulo.

Com a utilização de *caches* bloqueantes, as quais tem uma implementação em *hardware* mais simples, a taxa de acerto na BTB passa a ter uma importância maior, uma vez que consegue diminuir a quantidade de falhas no acesso à *cache*. Dessa forma, consegue aproximar o desempenho das arquiteturas SMT com *caches* bloqueantes àquelas que utilizam *caches* não-bloqueantes. Para o processador SMT-8, com uma taxa de acerto de apenas 50% a distância entre os desempenhos é de 14%. Com um previsor perfeito, essa diferença passa a ser de apenas 8%.

Como trabalhos futuros, podem ser citados:

- Adição de detalhamento ao simulador que permita a análise de tempo de ciclo de arquiteturas SMT;
- Impacto de atualizações especulativas nas tabelas de previsão de desvios, as quais geram uma maior pressão nos barramentos de escrita;
- Utilização de diferentes técnicas de escalonamento nos estágios de busca e graduação de instruções, como por exemplo, escolha de tarefas com maior número de desvios em execução;
- Utilização de *benchmarks* representativos de aplicações comerciais e multimídia, os quais podem ter comportamento diferente do SPEC CPU 2000.

REFERÊNCIAS

ADVANCED MICRO DEVICES. **AMD Athlon Processor Architecture**. Sunnyvale: [s.n.], 2000. Disponível em: <http://www.amd.com/products/cpg/athlon/pdf/architecture_wp.pdf>. Acesso em: maio 2001.

ALVERSON, R.; CALLAHAN, D.; CUMMINGS, D.; KOBLENZ, B.; PORTERFIELD, A.; SMITH, B. The Tera Computer System. In: INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, 1990. **Proceedings...** [S.l.: s.n.], 1990. p.1–6.

ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 19., 1992. **Proceedings...** New York: ACM, 1992.

ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 23., 1996, Philadelphia. **Proceedings...** New York: ACM, 1996.

ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 24., 1997. **Proceedings...** New York: ACM, 1997.

BLOCK, E. The Engineering Design of the STRETCH Computer. In: EASTERN JOINT COMPUTER CONFERENCE, 1959. **Proceedings...** New York: Spartan Books, 1959. p.48–58.

BURGER, D. C.; AUSTIN, T. M. **The SimpleScalar Tool Set, Version 2.0**. Madison: University of Wisconsin, 1997. (CS-TR-1997-1342).

CHANG, P.; BANERJEE, U. Profile-Guided Multi-Heuristic Branch Prediction. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, 1995. **Proceedings...** [S.l.: s.n.], 1995.

CHANG, P.-Y.; HAO, E.; YEH, T.-Y.; PATT, Y. Branch Classification: A new mechanism for improving branch predictor performance. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 27., 1994, San Jose. **Proceedings...** Los Alamitos: IEEE Computer Society, 1994. p.22–31.

CHEN, T.-F.; BAER, J.-L. Reducing memory latency via non-blocking and prefetching caches. In: INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEM, 5., 1992, Boston. **Proceedings...** New York: ACM Press, 1992. p.51–61.

CO, M.; SKADRON, K. The Effects of Context Switching on Branch Predictor Performance. In: IEEE INTERNATIONAL SYMPOSIUM ON PERFORMANCE ANALYSIS OF SYSTEMS AND SOFTWARE, 2001, Tuscon AZ. **Proceedings...** [S.l.: s.n.], 2001.

DRIESEN, K.; HOLZLE, U. The Cascaded Predictor: economical and adaptive branch target prediction. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 25., 1998, Barcelona. **Proceedings...** New York: ACM, 1998. p.249–258.

ECKERT, J. P.; CHU, J. C.; TONIK, A. B.; SCHMITT, W. F. Design of UNIVAC-LARC system. In: EASTERN JOINT COMPUTER CONFERENCE, 1959. **Proceedings...** New York: Spartan Books, 1959. p.59–74.

EGGERS, S. J.; EMER, J. S.; LEVY, H. M.; LO, J. L.; STAMM, R. L.; TULLSEN, D. M. Simultaneous Multithreading: A platform for next-generation processors. **IEEE Micro**, [S.l.], v.17, n.5, p.12–19, Sept/Oct 1997.

EVERS, M. **Improving branch prediction by understanding branch behavior**. 2000. Tese (Doutorado em Ciência da Computação) — University of Michigan, Department of Computer Science and Engineering.

EVERS, M.; CHANG, P.-Y.; PATT, Y. N. Using Hybrid Branch Predictors to Improve Branch Prediction Accuracy in the Presence of Context Switches. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 23., 1996, Philadelphia. **Proceedings...** New York: ACM, 1996. p.3–11.

FALCON, A.; STARK, J.; RAMIREZ, A.; LAI, K.; VALERO, M. Prophet/Critic Hybrid Branch Prediction. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 31., 2004, München, Germany. **Proceedings...** New York: ACM, 2004. p.250–263.

FERNANDES, E. S. T.; SANTOS, A. D.; WEBER, T. S. **Arquiteturas Super Escalares**: detecção e exploração do paralelismo de baixo nível. Porto Alegre: Instituto de Informática da UFRGS, 1992.

GONÇALVES, R. A. L. **Arquiteturas multi-tarefas simultâneas**: SEMPRE: arquitetura SMT com capacidade de execução e escalonamento de processos. 2000. 134p. Tese (Doutorado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

GONÇALVES, R. A. L.; AYGUADÉ, E.; VALERO, M.; NAVAUX, P. O. A. A Simulator for SMT Architectures: evaluating instruction cache topologies. In: SYMPOSIUM ON COMPUTER ARCHITECTURES AND HIGH PERFORMANCE COMPUTING, 12., 2000, São Pedro. **Proceedings...** São Paulo: SBC, 2000. p.279–286.

GONÇALVES, R. A. L.; PILLA, M. L.; PIZZOL, G. D.; SANTOS, T. G. S. dos; SANTOS, R. R. dos; NAVAUX, P. O. A. Evaluating the Effects of Branch Prediction Accuracy on the Performance of SMT Architectures. In: EUROMICRO WORKSHOP ON PARALLEL AND DISTRIBUTED PROCESSING, 9., 2001, Mantova. **Proceedings...** Los Alamitos: IEEE Computer Society, 2001. p.335–362.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture**: a quantitative approach. 3rd ed. San Francisco: Morgan Kaufmann, 2003.

HENNING, J. L. SPEC CPU200: measuring CPU performance in the new millenium. **IEEE Computer**, Los Alamitos, v.33, n.7, p.28–35, July 2000.

HILY, S.; SEZNEC, A. **Branch Prediction and Simultaneous Multithreading**. [S.l.]: IRISA, 1996. (RR-2843).

HIRATA, H.; KIMURA, K.; NAGAMINE, S.; MOCHIZUKI, Y.; NISHIMURA, A.; NAKASE, Y.; NISHIZAWA, T. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 19., 1992. **Proceedings...** New York: ACM, 1992. p.136–145.

INTEL. **Inside the NetBurst Micro-Architecture of the Intel Pentium 4 Processor**. Disponível em: <http://download.intel.com/pentium4/download/netburst.pdf>. Acesso em: jan. 2001.

JIMENEZ, D. A.; LIN, C. Dynamic Branch Prediction with Perceptrons. In: INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURES, 7., 2001, Monterey. **Proceedings...** Los Alamitos: IEEE Computer Society, 2001. p.197–206.

JIMENEZ, D. A.; LIN, C. Neural methods for dynamic branch prediction. **ACM Transactions on Computer Systems**, New York, v.20, n.4, p.369–397, Nov. 2002.

JIMéNEZ, D. A. Reconsidering Complex Branch Predictors. In: INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURES, 9., 2003, Anaheim, California. **Proceedings...** Los Alamitos: IEEE Computer Society, 2003. p.43–52.

JOHNSON, M. **Superscalar Microprocessor Design**. Englewood Cliffs: Prentice Hall, 1991.

JOUPPI, N. Superscalar versus Superpipelined Machines. **Computer Architecture News**, [S.l.], June 1988.

KAELI, D. R.; EMMA, P. G. ranch History Table Prediction of Moving Target Branches Due to Subroutine Returns. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 18., 1991. **Proceedings...** New York: ACM, 1991. p.34–42.

KARLINI, J.; STEFANOVIC, D.; FORREST, S. **The Triton Predictor**. [S.l.]: University of New Mexico, 2004. (TR-CS-2005-28).

KESSLER, R. E. The Alpha 21264 Microprocessor. **IEEE Micro**, Los Alamitos, v.19, n.2, March/April 1999.

LEE, J. K.; SMITH, A. Branch Prediction Strategies and Branch Target Buffer Design. **IEEE Computer**, Los Alamitos, v.17, n.1, p.6–22, 1984.

LIPASTI, M. H.; SHEN, J. P. Exceeding the Dataflow Limit via Value Prediction. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 29., 1996, Paris. **Proceedings...** Los Alamitos: IEEE Computer Society, 1996. p.226–237.

MARR, D.; BINNS, F.; HILL, D. L.; HINTON, G.; KOUFATY, D. A.; MILLER, J. A.; UPTON, M. Hyper–Threading Technology Architecture and Microarchitecture. **Intel Technology Journal**, [S.l.], v.6, n.1, Q1 2002.

MCFARLING, S. **Combining Branch Predictors**. Palo Alto: Western Labs, 1993. (DEC WRL TN–36).

MIPS Technologies Inc. **MIPS R10000 Microprocessor User's Manual**. Mountain View: [s.n.], 1995.

PAN, S. T.; SO, K.; RAHMEH, J. T. Improving the accuracy of dynamic branch prediction using branch correlation. In: ANNUAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, 5., 1992, San Jose. **Proceedings...** New York: ACM, 1992. p.76–84.

PATEL, S.; FRIENDLY, D.; PATT, Y. Evaluation of design options for the trace cache fetch mechanism. **IEEE Transactions on Computers**, New York, v.48, n.2, p.435–446, Feb. 1999.

PERLEBERG, C.; SMITH, A. Branch Target Buffer Design and Optimization. **IEEE Transactions on Computers**, New York, v.42, n.4, p.396–412, Apr. 1993.

PIERRE MICHAUD, A. S.; UHLIG, R. Trading conflict and capacity aliasing in conditional branch predictors. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 24., 1997. **Proceedings...** New York: ACM, 1997. p.292–303.

PILLA, M. L. **RST**: reuse through speculation on traces. 2004. Tese (Doutorado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

PILLA, M. L.; NAVAUX, P. O. A.; CHILDERS, B. R.; COSTA, A. T. da; FRANçA, F. M. G. Value Predictors for Reuse through Speculation on Traces. In: SYMPOSIUM ON COMPUTER ARCHITECTURES AND HIGH PERFORMANCE COMPUTING, 16., 2004, Foz do Iguaçu. **Proceedings...** Porto Alegre: SBC, 2004. p.48–55.

PIZZOL, G. D.; PILLA, M. L.; NAVAUX, P. O. A. Branch Prediction x Performance: an analysis on superscalar processors. In: SYMPOSIUM ON COMPUTER ARCHITECTURES AND HIGH PERFORMANCE COMPUTING, 13., 2001, Pirenópolis. **Proceedings...** Brasília: SBC, 2001. p.56–61.

ROSE, C. F. D.; NAVAUX, P. O. A. **Arquiteturas Paralelas**. Porto Alegre: Sagra Luzzatto, 2003. 152p.

ROTENBERG, E.; JACOBSON, Q.; SAZEIDES, Y.; SMITH, J. Trace processors. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 30., 1997. **Proceedings...** Los Alamitos: IEEE Computer Society, 1997. p.138–148.

SAIR, S.; CHARNEY, M. **Memory Behavior of the SPEC2000 Benchmark Suite**. [S.l.]: IBM T.J. Watson Research Center, 2000. (RC-21852).

SANTOS, R. R. dos. **Um Mecanismo de Busca Especulativa de Múltiplos Fluxos de Instruções**. 1997. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

SANTOS, R. R. dos. **DCE**: the Dynamic Conditional Execution in a multipath control independent architecture. 2003. Tese (Doutorado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

SANTOS, T. G. S. dos. **Análise do Comportamento de Mecanismos de Pré-busca em Memórias Hierárquicas de Microprocessadores RISC Superescalares**. 2000. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

SHIVAKUMAR, P.; JOUPPI, N. P. **CACTI 3.0**: an integrated cache timing, power, and area model. Palo Alto: Western Labs, 2001. (WRL RR 2001/2).

SMITH, B. Architecture and applications of the HEP multiprocessor computer system. In: REAL TIME SIGNAL PROCESSING, 4., 1981, New York. **Proceedings...** [S.l.]: SPIE Press, 1981. p.241–248.

SMITH, J. E. A study of branch prediction strategies. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 8., 1981, Minneapolis, MN, USA. **Proceedings...** New York: ACM, 1981. p.135–148.

SMITH, J. E.; SOHI, G. S. The Microarchitecture of Superscalar Processors. **Proceeding of the IEEE**, New York, v.83, p.1609–1624, Dec. 1995.

SNAVELY, A.; CARTER, L.; BOISSEAU, J.; MAJUMDAR, A.; GATLIN, K. S.; MITCHELL, N.; FEO, J.; KOBLENZ, B. Multi-processor Performance on the Tera MTA. In: ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 1998, Orlando, FL. **Proceedings...** [S.l.: s.n.], 1998. p.1–8.

SODANI, A.; SOHI, G. S. Dynamic Instruction Reuse. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 24., 1997. **Proceedings...** New York: ACM, 1997.

SOHI, G. Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers. **IEEE Transactions on Computers**, New York, v.39, n.3, p.349–359, March 1990.

STAEHLER, W. T.; PIZZOL, G. D.; NAVAUX, P. O. A. Integração Extended SimMan Tool & CCS - Simulação de Arquiteturas Superescalares em Clusters. In: WORKSHOP EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO, 4., 2003, São Paulo. **Anais...** São Paulo: USP, 2003. p.56–63.

TENDLER, J.; DODSON, S.; FIELDS, S.; LE, H.; SINHAROY, B. **POWER4 system microarchitecture**. [S.l.]: IBM Server Group, 2001.

TOMASULO, R. M. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. **IBM Journal of Research and Development**, Armonk, v.11, n.1, Jan. 1967.

TULLSEN, D. M.; EGGERS, S. J.; EMER, J. S.; LEVY, H. M.; LO, J. L.; STAMM, R. L. Exploiting Choice: instruction fetch and issue on an implementable simultaneous multithreading processor. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 23., 1996, Philadelphia. **Proceedings...** New York: ACM, 1996. p.191–202.

TULLSEN, D. M.; EGGERS, S.; LEVY, H. M. Simultaneous Multithreading: maximizing on-chip parallelism. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 22., 1995, Santa Margherita Ligure. **Proceedings...** New York: ACM, 1995. p.392–403.

UNGERER, T.; SIGMUND, U. Evaluating A Multithreaded Superscalar Microprocessor Versus a Multiprocessor Chip. In: WORKSHOP ON PARALLEL SYSTEMS AND ALGORITHMS, 4., 1996, Forschungszentrum Jülich, Germany. **Proceedings...** [S.l.]: World Scientific Publishing, 1996. p.147–159.

YAMAMOTO, W.; SERRANO, M. J.; TALCOTT, A. R.; WOOD, R. C.; NEMIROVSKY, M. Performance estimation of multistreamed, superscalar processors. In: HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES, 27., 1994. **Proceedings...** New York: ACM, 1994. p.195–204.

YEH, T.-Y.; PATT, Y. N. Two-Level Adaptive Training Branch Prediction. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 24., 1991, Alburquerque. **Proceedings...** Los Alamitos: IEEE Computer Society, 1991. p.51–61.

YEH, T.-Y.; PATT, Y. N. A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 20., 1993, San Diego. **Proceedings...** New York: ACM, 1993. p.257–266.