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

Arquiteturas Multi-Tarefas Simultâneas: SEMPRE – Arquitetura SMT com Capacidade de Execução e Escalonamento de Processos

por

RONALDO AUGUSTO DE LARA GONÇALVES

Tese submetida à avaliação,
como requisito parcial para a obtenção do grau de Doutor
em Ciência da Computação

Prof. Philippe Olivier Alexandre Navaux
Orientador

Porto Alegre, outubro de 2000

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Gonçalves, Ronaldo Augusto de Lara
Arquiteturas Multi-Tarefas Simultâneas: SEMPRE - Arquitetura SMT com Capacidade de Execução e Escalonamento de Processos / por Ronaldo Augusto de Lara Gonçalves. – Porto Alegre : PPGC do II/UFRGS, 2000.
136p.:il.

Tese (Doutorado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, RS-BR, 2000. Orientador: Navaux, Philippe O. A.

1. Arquiteturas SMT. 2. Arquiteturas Superescalares. 3. Arquiteturas de Processadores. 4. Avaliação de Desempenho. 5. Simulação. I. Navaux, Philippe Olivier Alexandre. II. Título.

Agradecimentos

Durante todo o curso de doutorado, várias pessoas me apoiaram e eu gostaria de agradecê-las. Primeiramente, ao meu orientador Prof. Philippe Navaux, pela orientação segura e confiança irrestrita, sem as quais eu jamais teria terminado este trabalho. Também, ao pessoal do Departamento de Informática da Universidade Estadual de Maringá, que permitiu o meu afastamento e assumiu a manutenção das atividades que a mim seriam atribuídas. Aos colegas do grupo APSE, com os quais eu pude discutir temas importantes e que me ajudaram a definir de forma mais precisa a presente tese de doutorado. Aos professores Mateo Valero e Eduard Ayguadé, da Universidade Politécnica da Catalúnia, pela ajuda no desenvolvimento do simulador básico. Aos colegas de sala, pelas conversas descontraídas que muito me ajudaram nos momentos de trabalho árduo. Ao pessoal dos laboratórios da UFRGS, pelo apoio na manutenção da minha área de trabalho. Aos professores e funcionários da UFRGS e da UEM, pela assessoria nos momentos de necessidade. A CAPES e ao CNPq, pelo suporte financeiro concedido e finalmente, a todas as pessoas que direta ou indiretamente contribuíram para a conclusão do presente trabalho.

Dedico esta tese aos meus pais
Agostinho e Maria,
à minha querida esposa Maria,
e aos meus filhos amados Carla e Caio.

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

O avanço tecnológico no projeto de microprocessadores, nos recentes anos, tem seguido duas tendências principais. A primeira tenta aumentar a freqüência do relógio dos mesmos usando componentes digitais e técnicas VLSI mais eficientes. A segunda tenta explorar paralelismo no nível de instrução através da reorganização dos seus componentes internos. Dentro desta segunda abordagem estão as arquiteturas multi-tarefas simultâneas, que são capazes de extrair o paralelismo existente entre e dentro de diferentes tarefas das aplicações, executando instruções de vários fluxos simultaneamente e maximizando assim a utilização do *hardware*.

Apesar do alto custo da implementação em *hardware*, acredita-se no potencial destas arquiteturas para o futuro próximo, pois é previsto que em breve haverá a disponibilidade de bilhões de transistores para o desenvolvimento de circuitos integrados. Assim, a questão principal a ser encarada talvez seja: como prover instruções paralelas para uma arquitetura deste tipo? Sabe-se que a maioria das aplicações é seqüencial pois os problemas nem sempre possuem uma solução paralela e quando a solução existe os programadores nem sempre têm habilidade para ver a solução paralela. Pensando nestas questões a arquitetura SEMPRE foi projetada.

Esta arquitetura executa múltiplos processos, ao invés de múltiplas tarefas, aproveitando assim o paralelismo existente entre diferentes aplicações. Este paralelismo é mais expressivo do que aquele que existe entre tarefas dentro de uma mesma aplicação devido a não existência de sincronismo ou comunicação entre elas. Portanto, a arquitetura SEMPRE aproveita a grande quantidade de processos existentes nas estações de trabalho compartilhadas e servidores de rede. Além disso, esta arquitetura provê suporte de *hardware* para o escalonamento de processos e instruções especiais para o sistema operacional gerenciar processos com mínimo esforço. Assim, os tempos perdidos com o escalonamento de processos e as trocas de contextos são insignificantes nesta arquitetura, provendo ainda maior desempenho durante a execução das aplicações. Outra característica inovadora desta arquitetura é a existência de um mecanismo de pré-busca de processos que, trabalhando em cooperação com o escalonamento de processos, permite reduzir faltas na *cache* de instruções. Também, devido a essa rápida troca de contexto, a arquitetura permite a definição de uma fatia de tempo (*fatia de tempo*) menor do que aquela praticada pelo sistema operacional, provendo maior dinâmica na execução das aplicações.

A arquitetura SEMPRE foi analisada e avaliada usando modelagem analítica e simulação dirigida por execução de programas do SPEC95. A modelagem mostrou que o escalonamento por *hardware* reduz os efeitos colaterais causados pela presença de processos na *cache* de instruções e a simulação comprovou que as diferentes características desta arquitetura podem, juntas, prover ganho de desempenho razoável sobre outras arquiteturas multi-tarefas simultâneas equivalentes, com um pequeno acréscimo de *hardware*, melhor aproveitando as fatias de tempo atribuídas aos processos.

Palavras-Chaves: Arquiteturas SMT, Arquiteturas Superescalares, Arquiteturas de Processadores, Avaliação de Desempenho

TITLE: "SIMULTANEOUS MULTI-THREADED ARCHITECTURES: SEMPRE - SMT ARCHITECTURE WITH PROCESS EXECUTION AND SCHEDULING"

Abstract

The technological advance in microprocessor design in the recent years has followed two main trends. The first one tries to increase the microprocessor clock frequency using more efficient digital components and VLSI techniques. The second one tries to exploit parallelism at the instruction level through the reorganization of its internal components. In the second approach there are the simultaneous multithreaded architectures, which are able to extract inter and intra-thread parallelism from applications, executing instructions of many flows simultaneously, maximizing the utilization of the hardware.

Besides high cost of hardware implementation, it is believed there is potential for these architectures in a near future, because it is forecasted that soon there will be the billions of transistors available for the development of integrated circuits. Thus, the major question to be faced is: how to provide parallel instructions to the architecture of this type? It is known that the most of applications is sequential because the problems can not be always parallelized, and when it is possible, the programmers many times don't have ability to see the parallel solution. Thinking on this questions we have designed the SEMPRE architecture.

This architecture executes multiple processes, instead of multiple threads, taking advantage of the existent parallelism among different applications. That parallelism is more expressive than among threads inside the same application, due to lack of both synchronism and communication. Therefore, the SEMPRE architecture makes a good use of the great amount of existent processes in shared workstations and network servers. Besides, this architecture provides hardware support to process scheduling and special instructions to help the operating system to manage processes with minimal effort. Thus, the lost time with both process scheduling and context switching are insignificant in this architecture, providing much more performance during the execution of applications. Another novel feature of this architecture is the existence of a process pre-fetching mechanism that, working in cooperation with the process scheduling, reduces instruction cache misses. Also, due that fast context switching, the architecture allows to define a time slice smaller than that used by operating system, providing more dynamics in applications' execution.

The SEMPRE architecture was both analyzed and evaluated using analytical modeling and execution driven simulation of SPEC95 benchmarks. The analytical model showed that scheduling by *hardware* reduces the side effects caused by presence of processes in the instruction cache, and the simulation showed that the different features of this architecture can provide reasonable performance over other equivalent simultaneous multithreaded architectures, with minimal increase of *hardware*, taking more advantage of process time-slice.

Keywords: SMT Architecture, Superscalar Architecture, Processor Architecture, Performance Evaluation.

1 Introdução

Antigamente, o *hardware* dos processadores convencionais era pouco utilizado durante a execução das instruções. Posteriormente, com a divisão da execução das instruções em estágios, onde a finalização de um estágio por uma instrução permite que uma nova instrução utilize aquele estágio, tornou-se possível executar com sobreposição parcial várias instruções no mesmo ciclo do processador. Esta técnica é chamada de *pipeline* e talvez tenha sido o maior passo na evolução arquitetural dos processadores.

Mas com o decorrer de muito pouco tempo, as aplicações ficam maiores e mais complexas, exigindo constantes alterações arquiteturais que permitam aos processadores ficarem mais rápidos. Aqueles construídos com arquiteturas de *pipeline* convencional já não atendem mais as necessidades de desempenho. Muita pesquisa surgiu nesta área e o *pipeline* foi reestruturado e ampliado para o que atualmente é conhecido como arquitetura superescalar. Estas arquiteturas possuem a capacidade de executar instruções com paralelismo simultâneo, além daquele com sobreposição parcial. Os processadores de última geração têm sido projetados como arquiteturas superescalares, tais como o *Pentium* [AND 95] e [SAI 93], o Power e *PowerPC* [CHD 94], [DIE 95] e [YOU 96], MIPS R10000 [MIP 95], UltraSparc [ULT 96] e Alpha 21264 [KES 99].

Estas arquiteturas possuem mecanismos eficientes de previsão de desvios, escalonamento dinâmico de instruções, renomeação de registradores, execução fora-de-ordem e outras técnicas elaboradas que proporcionam uma máxima extração do paralelismo no nível de instrução existente nos códigos convencionais (mono-tarefa). Mesmo com todo este *hardware* sofisticado, o desempenho destas arquiteturas ainda é fortemente limitado por um fator degradante e inevitável: as dependências verdadeiras, também conhecidas como dependências do fluxo de dados, inerente às aplicações mono-tarefa.

Em uma visão mais ampla, sabe-se que o paralelismo no nível de instrução é limitado basicamente por 3 fatores: dependência de controle, dependência de dados verdadeira (fluxo de dados) e dependência de recursos de *hardware* (falsas dependências, conflitos de portas, barramento etc.). Dentre estes 3 fatores, a dependência causada pelo fluxo de dados não consegue ser resolvida por nenhuma arquitetura atual. Em uma atitude de extremo exagero, as dependências de recursos e de controles poderiam ser eliminadas se a arquitetura fosse projetada com recursos de *hardware* infinitos e se todos os caminhos dos desvios, tomados e não tomados, fossem executados simultaneamente. Obviamente, isto não é viável e nem ao menos possível, mas mesmo assim, as dependências de controles podem ser quase eliminadas usando boas técnicas de previsão de desvios, e as dependências de recursos podem ser satisfatoriamente reduzidas com um bom balanceamento da arquitetura e uso de renomeação de registradores. Mas as dependências verdadeiras sempre continuarão, e por isso têm sido alvo em recentes pesquisas.

Jouppi e Wall [JOU 89] mediram um paralelismo variando de 1.6 até 3.2 instruções por ciclo (*IPC*). Butler, Yeh e Patt [BUT 91] concluíram que se o *hardware* estiver corretamente balanceado, o paralelismo pode atingir de 2 até 5.8 *IPC*. Tran e Wu [TRA 92] não conseguiram obter paralelismo maior que 2 *IPC*, para aplicações reais. Um dos

trabalhos mais completos e com resultados mais otimistas foi desenvolvido por Wall [WAL 93] através de simulações sobre 18 diferentes programas compilados para código MIPS, com 375 modelos diferentes de análise do paralelismo disponível. Para a maioria dos programas simulados por Wall, o paralelismo médio variou de 4 a 10 *IPC*, usando técnicas conhecidas de extração de paralelismo, mas não-triviais.

Preocupados com o futuro, onde os limites do paralelismo poderão ser causados somente pelo fluxo de dados, Lipasti e Shen [LIP 96] começaram a desenvolver trabalhos que envolvem a especulação de dados. Eles foram motivados a estes estudos após concluírem que os processadores superescalares modernos, apesar de toda a tecnologia envolvida, são capazes de executar somente entre 0,5 e 1,5 instrução por ciclo.

A teoria de Park [PAR 91] é mais pessimista: "O desempenho além do limite de 2,5 *IPC* para a maioria das aplicações não-científicas e de propósitos gerais não pode ser alcançado com o uso de um único fluxo de instruções". Obviamente, isto sugere o uso de arquiteturas multi-tarefas, que tentam obter alto grau de *IPC* escalonando instruções provenientes de diferentes tarefas e possuem a habilidade de esconder as latências de acesso à memória e as latências das dependências entre instruções.

A forma como esses fluxos são executados dentro do processador pode seguir duas abordagens básicas: concorrentes ou simultâneas. A primeira abordagem, chamada de Multi-Tarefas, permite intercalar a execução de diferentes tarefas durante a execução, mas em determinado momento, somente uma tarefa pode estar ocupando as unidades funcionais [TSA 96]. A segunda abordagem aproveita os recursos do processador de forma mais eficiente, executando simultaneamente instruções de diferentes tarefas [TUL 95]. Este tipo de arquitetura é conhecido como SMT (*Simultaneous MultiThreaded*), ou Multi-Tarefas Simultâneas.

Apesar de ser uma técnica promissora, o desenvolvimento destas arquiteturas também tem sido limitado por outros fatores característicos das aplicações multi-tarefas, tais como: 1) seqüencialidade das aplicações, pois a maioria das aplicações convencionais não é multi-tarefas, 2) fluxo de mensagens, característico das aplicações multi-tarefas e que reduz o desempenho devido à necessidade de gerenciamento de pontos de sincronização, 3) interferência do sistema operacional, que causa a troca de contexto e não tira vantagem da fatia de tempo de cada aplicação e 4) degradação do desempenho da *cache* de instruções causada pelos conflitos entre as tarefas.

Visando reduzir estes problemas, este trabalho apresenta o projeto de uma arquitetura SMT chamada SEMPRE, que visa a execução de múltiplos processos ao invés de múltiplas tarefas, além de possuir a habilidade adicional de executar diretamente pelo *hardware* muitas das operações (onerosas em consumo de tempo de CPU) normalmente executadas pelo sistema operacional, tais como escalonamento e gerenciamento de processos. Com isso, a arquitetura SEMPRE é capaz de aproveitar o grande número de aplicações não-comunicantes que são executadas nas estações de trabalho compartilhadas e servidores de rede, e simplificar o trabalho do sistema operacional, proporcionando maior desempenho final do sistema. Devido ao escalonamento de processos feito pelo *hardware*, a arquitetura SEMPRE é capaz de melhor aproveitar as fatias de tempo dos processos e também reduzir as faltas na *cache* de instruções usando um mecanismo de pré-busca orientado pelo escalonamento.

Para o desenvolvimento do presente projeto foi necessário um estudo sobre os principais conceitos e técnicas associados aos processadores superescalares e multi-tarefas simultâneas, incluindo estudo de casos, cujo material coletado está organizado nos capítulos 2 e 3. Este material serviu de apoio durante a definição das características e funcionalidades da arquitetura SEMPRE, apresentada no capitulo 4. No capítulo 4 também são descritas as principais simplificações algorítmicas que levaram a implementação de um simulador.

No capítulo 5 é apresentada uma análise de comportamento da unidade de busca desta arquitetura, através do uso de modelagem analítica, a qual mostrou que a mesma pode prover alto desempenho sob determinadas configurações. No capítulo 6 é apresentado o simulador que foi desenvolvido a partir da ferramenta *SimpleScalar*, bem como os resultados de diversas simulações e análises de desempenhos que mostram a eficiência e a importância de tal arquitetura. Nos capítulos seguintes tem-se as conclusões, os trabalhos futuros, as publicações e as referências utilizadas.

2 Arquiteturas Superescalares

Pode-se facilmente notar que as arquiteturas superescalares surgiram com o aumento do *hardware* existente nas arquiteturas *pipeline*, e da mesma forma, foi adicionado mais *hardware* nas superescalares para que estas se tornassem multi-tarefas simultâneas. A figura 2.1 resume esta evolução em função do aumento de *hardware*.

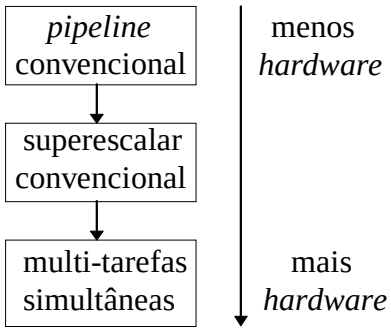


FIGURA 2.1 - Evolução da SMT x Aumento do *Hardware*

Muitos aspectos funcionais da mais simples arquitetura também estão presentes na mais complexa, incluindo limitações e problemas. Assim, um estudo em todos estes níveis arquiteturais, abordando suas principais características, se fez necessário. Este estudo é apresentado nas próximas seções a fim de permitir melhor situar o presente trabalho e facilitar o seu entendimento e avaliação.

2.1 *Pipeline*

Pipeline foi inicialmente desenvolvido na década de 1950 e se tornou a principal característica dos computadores de larga escala da década de 1960, tais como os CDC 6600 e IBM 360/91, que apesar de não serem superescalares, pois executavam no máximo uma instrução por ciclo, apresentavam características dos processadores superescalares atuais, tais como várias unidades funcionais e escalonamento dinâmico de instruções [SMI 95].

Pipeline [HEN 94] 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 "tubulação" onde as instruções entram por um extremo, percorrem os estágios e saem pelo outro extremo. Um exemplo de como ocorre uma execução em uma arquitetura *pipeline* de 5 estágios pode ser visto na figura 2.2, onde são mostradas as diferentes fases na execução de três instruções, em diferentes ciclos de um referencial de tempo comum.

Todos os estágios do *pipeline* devem estar prontos para prosseguir no mesmo tempo. Assim, a freqüência com que as instruções deixam o *pipeline* não pode exceder a freqüência com que elas entram no *pipeline*. O tempo necessário para mover uma instrução por um estágio é, idealmente, de um ciclo. O tempo do ciclo é determinado pelo tempo do estágio mais lento, pois todos os estágios devem prosseguir sincronizados na mesma freqüência [DWY 92].

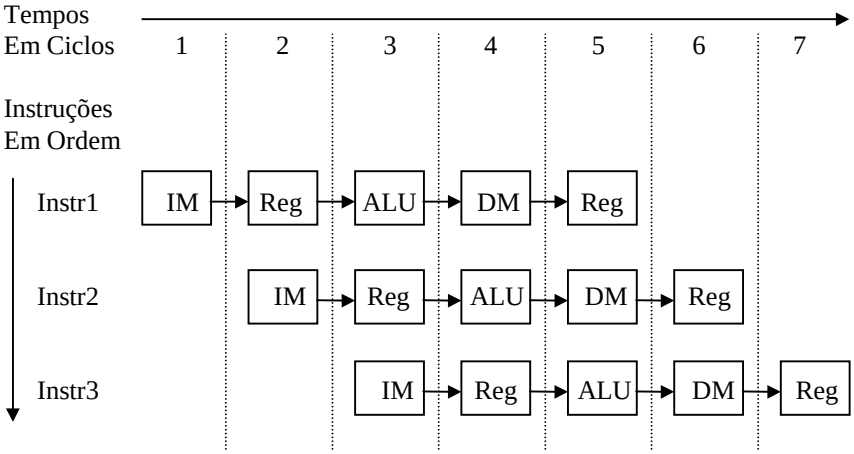


FIGURA 2.2 - Execução no *Pipeline* de 5 Estágios

Muitas vezes o *pipeline* é representado de forma muito simples, como na figura 2.2, o que não permite visualizar seus componentes internos. A figura 2.3 [HEN 94] mostra uma visão mais detalhada e real do *pipeline* anterior, destacando seus 5 estágios (busca, decodificação, execução, acesso à memória e escrita dos resultados), os registradores intermediários, para conter os dados em transição e permitir que os estágios sejam compartilhados entre diferentes instruções, além multiplexadores, deslocadores e adicionadores.

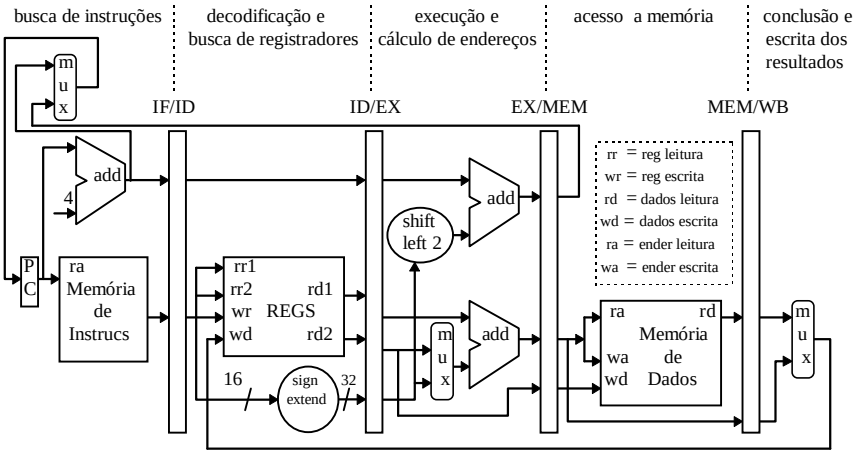


FIGURA 2.3 - *Pipeline* de 5 Estágios Detalhado

Quando a execução das instruções é quebrada em passos ainda menores, prolongando o *pipeline* e fornecendo granulosidade fina na execução, surgem as arquiteturas

superpipelining [RYA 92]. 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*. Os principais problemas deste tipo são discutidos na próxima subseção.

2.1.1 Azares do *Pipeline*

O *pipeline* trabalha bem enquanto nenhuma das instruções precisa esperar algum resultado de uma instrução precedente, ou quando não há uma alteração no fluxo de instruções, como um desvio ou interrupção, por exemplo [RYA 92]. Estas situações, chamadas de azares do *pipeline* (*pipeline hazards*), podem ocorrer em um ou mais estágios, fazendo com que as instruções afetadas sejam bloqueadas até que o problema seja resolvido. Existem vários tipos de azares do *pipeline* [GOL 94], cinco dos quais descritos a seguir, que precisam ser controlados para evitar erros na computação.

1) Azar do Desvio: as instruções subseqüentes a uma instrução de desvio (*branch*), no caminho não-tomado, são dependentes de controle da instrução de desvio, pois enquanto esta não for resolvida não é possível saber se elas devem ou não ser executadas. Esta situação está sempre presente no *pipeline,* e convive juntamente com os demais azares.

2) Azar da Leitura: As instruções seguintes a uma instrução de leitura de dado em memória (*load*) podem precisar deste dado que pode ainda não estar pronto no momento da execução destas instruções, podendo gerar resultados incorretos.

3) Azar de Escrita: As instruções de leitura em memória (*load*), seguintes a uma instrução de escrita em memória (*store*), podem querer ler o dado armazenado na memória antes que ele esteja pronto, podendo gerar resultados incorretos.

4) Azar de Geração de Endereços: As instruções de leitura em memória podem acessar posições de memória cujos endereços de base são computados por instruções anteriores. Estes endereços podem ainda não estar prontos quando as instruções de leitura forem executadas, podendo gerar resultados incorretos.

5) Azar da Dependência Verdadeira: Se o resultado de uma instrução é requisitado por instruções seguintes no código, que são executadas antes que o referido resultado esteja pronto, resultados incorretos podem ser gerados. Este tipo de dependência é chamado de dependência verdadeira ou RAW (*read-after-write*) [SMI 95]. Outras dependências conhecidas como falsas dependências ou WAR (*write-after-read*) e WAW (*write-after-write*), podem ser eliminadas através de renomeação de registradores [JOH 91].

De uma forma geral, instruções anteriores geram azares (instrução azarenta) em instruções posteriores do código (instrução azarada), pois estas segundas são afetadas pelas primeiras. De uma forma geral, os azares podem ser evitados e/ou controlados através de diferentes abordagens que podem ser utilizadas em conjunto, tais como [GOL 94]:

1) reordenação do código a ser executado: através de uma reestruturação de código, uma ou mais instruções não azaradas podem ser colocadas imediatamente após a instrução azarenta, servindo de atraso na computação até que a instrução azarenta seja concluída. Mas esta alternativa é bastante limitada pela falta de paralelismo nas imediações da instrução azarenta. Quase sempre são colocadas instruções de NOPs que reduzem o desempenho do *pipeline* por não realizarem trabalho útil.

2) parar o *pipeline*: causar uma parada (*stall*) no *pipeline*, quando uma instrução azarenta for decodificada, até que ela se complete. Esta alternativa é conhecida como *interlock*.

3) usar previsão: prever o caminho a ser seguido após um desvio para tentar executar as instruções seguintes mais prováveis. Outras formas de previsões, não muito comuns, também podem ser feitas. A previsão de leitura em memória pode buscar o dado antecipadamente, evitando azares de leitura. A previsão de escrita em memória pode armazenar o dado antecipadamente evitando azares de escrita. As previsões de leitura e escrita acabam executando estas instruções antecipadamente e se comportam como reordenação de código.

4) usar caminhos-desvios (*bypass*): que são caminhos de dados que interligam estágios do *pipeline*, permitindo que o dado gerado por uma instrução ainda não concluída seja antecipadamente enviado para outra que a requisita. Isto é possível nos estágios finais do *pipeline*, quando o dado já foi de fato calculado mas ainda restam estágios no *pipeline* tais como o estágio de conclusão e/ou escrita dos resultados.

As arquiteturas *pipeline* distribuem os recursos do *hardware* entre diferentes instruções, mas não aproveitam o paralelismo inerente das aplicações [GON 97b]. Visando melhorar o desempenho destas arquiteturas, uma nova abordagem é adotada para permitir a execução paralela das instruções, além da execução com sobreposição parcial já existente. Esta nova abordagem deu origem aos processadores superescalares, onde o *pipeline* é replicado e cada unidade funcional do processador possui seu próprio *pipeline* e pode operar independentemente das outras unidades [RYA 92]. Estas arquiteturas são discutidas a seguir.

2.2 Arquiteturas Superescalares

Os microprocessadores superescalares começaram a aparecer no final da década de 1980, e estão sendo hoje projetados e produzidos, por todos os fabricantes de produtos de alto-desempenho, como padrão tecnológico. De uma forma geral, um processador superescalar apresenta as seguintes características [SMI 95]:

- estratégias de busca de instruções, para buscar várias instruções simultaneamente, muitas vezes prevendo os desvios condicionais.
- métodos para determinação e tratamento das dependências entre os dados dos registradores.
- métodos para o despacho de múltiplas instruções.

- recursos para a execução paralela de múltiplas instruções, incluindo: múltiplas unidades funcionais e memória hierárquica para permitir múltiplos acessos.
- métodos para comunicação dos dados através da memória via instruções de leitura e escrita em memória.
- métodos para a recuperação do estado do processador na ordem correta

A figura 2.4 ilustra o modelo de execução paralelo usado na maioria dos processadores superescalares. As instruções começam no programa estático. O processo de busca de instruções, incluindo previsão de desvios, é usado para formar um fluxo de instruções dinâmicas. Estas instruções são inspecionadas para que muitas dependências artificiais sejam removidas.

As instruções são então despachadas para a janela de execução. As instruções da janela de execução não são representadas em ordem seqüencial, mas são parcialmente ordenadas pelas suas dependências de dados verdadeiras. As instruções são remetidas (*issued*) da janela em uma ordem determinada pelas dependências verdadeiras e pela disponibilidade de recursos de *hardware*. 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.

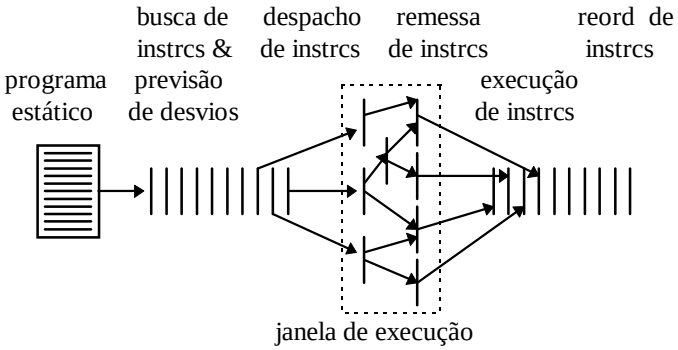


FIGURA 2.4 - Modelo de Execução Superescalar

A figura 2.5 ilustra a organização do *hardware* de um processador superescalar típico, onde os principais componentes podem ser vistos. As maiores fases desta micro-arquitetura são: busca, decodificação, despacho, remessa, execução e conclusão da instrução. A previsão de desvios pode ser feita durante a busca, durante a decodificação ou de forma combinada nos dois estágios como no PowerPC 620 [DIE 95].

Nesta figura, podem ser vistas as principais estruturas manipuladas: as *caches* de dados e de instruções, para armazenar dados e código, respectivamente; a fila de busca, onde as instruções buscadas da *cache* são colocadas para posterior decodificação; a fila de instruções, onde as instruções já decodificadas são colocadas para depois serem despachadas para os filas de remessa junto às unidades funcionais, para posterior execução; a fila de reordenação, utilizada normalmente para controlar a seqüência

correta das instruções, e o conjunto de registradores, muitas vezes divididos em vários bancos.

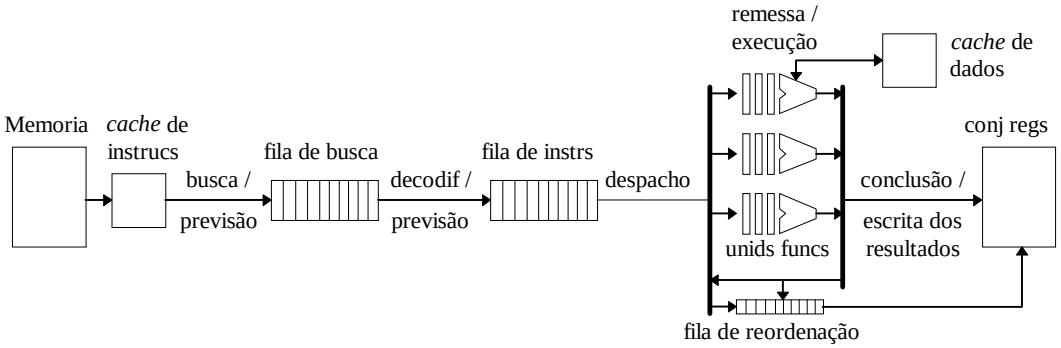


FIGURA 2.5 - Arquitetura Típica de Um Processador Superescalar

Muitas arquiteturas apresentam outras estruturas e estágios, tais como tabelas de renomeação ou estágio de pré-busca. Outra diversidade que existe entre elas está no tipo do conjunto de instruções utilizado. Enquanto algumas utilizam instruções RISC (MIPS, SPARC e PowerPC) outras utilizam instruções CISC (Pentium e AMD). Deve-se também ter em mente que, subjacente a esta organização, existe uma implementação de *pipeline* onde os estágios específicos podem ou não estar alinhados com as maiores fases da execução superescalar. Em função da complexidade e da importância de cada fase, elas são discutidas a seguir.

2.2.1 Busca de Instruções

A fase de busca de instruções fornece instruções para o resto do *pipeline* da arquitetura. As instruções são buscadas na *cache* de instruções e inseridas em uma fila de busca. Nesta fase também costuma ser realizada a previsão de desvios. Assim, as principais estruturas manipuladas são a *cache* de instruções, a fila de busca e as tabelas de previsão de desvios. Em muitas arquiteturas, antes da fase de busca, existe uma fase de pré-busca (*pre-fetching*), que antecipa a busca de instruções da *cache* secundária ou memória. Em algumas arquiteturas [DWY 92], o estágio de busca também pré-decodifica as instruções para facilitar posteriormente a decodificação. A figura 2.6 mostra as estruturas envolvidas nesta fase.

A *cache* de instrução, é uma pequena memória chamada de *cache* primária, que contém as instruções com maior probabilidade de serem utilizadas em determinado momento, e serve para reduzir a latência de acesso à memória e aumentar a largura de busca de instruções. A *cache* é organizada em blocos, ou linhas, contendo instruções consecutivas. O contador de programa (PC) é usado para procurar os conteúdos da *cache* associativamente, e determinar se a instrução endereçada está presente ou não em uma de suas linhas. Se sim, ocorre então um acerto na *cache* (*cache hit*), senão, ocorre uma falta na *cache*. Na ocorrência de falta na *cache* a linha em questão deve ser buscada da *cache* secundária, que geralmente contém dados e instruções juntas, ou da memória principal, e colocada na *cache* primária, mas isto pode levar vários ciclos. Um dos principais causadores de faltas na *cache* é a ocorrência de instruções de desvios, que podem romper a fronteira da localidade das instruções.

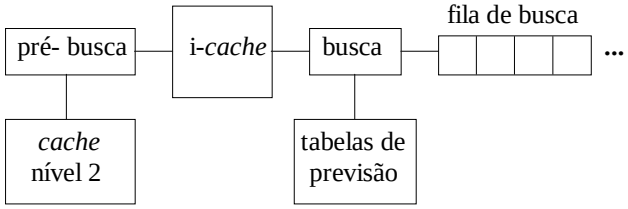


FIGURA 2.6 - Representação da Fase de Busca

Para processadores superescalares, a fase de busca deve ser capaz de buscar múltiplas instruções por ciclo. Para suportar esta largura de busca torna-se quase que obrigatório separar a *cache* de dados da *cache* de instruções. O número de instruções buscadas por ciclo deve ser, no mínimo, igual à taxa de decodificação e execução de instruções, e é algo geralmente maior **[**SMI 95]. Com isso, a ocorrência de falta na *cache* pode ser um fator muito degradante no desempenho, pois um número maior de instruções pode não ser executado. Várias alternativas são utilizadas para reduzir esta penalidade, das quais podem ser destacadas:

A mais simples talvez seja a própria fila de busca, que permite ao processador estocar instruções para suportar a ocorrência de faltas na *cache*. Em algumas arquiteturas, tais como MIPS [MIP 95] e o Pentium [AND 95], existe uma segunda fila de busca adicional (*resume buffer*) para conter as instruções que deveriam ser descartadas pelo mecanismo de previsão, por não pertencerem ao caminho previsto, mas que eventualmente podem ser recuperadas rapidamente caso o mecanismo de previsão falhe.

Outra alternativa simples, já citada, para reduzir esta penalidade é a utilização de pré-busca da próxima linha, onde a linha seguinte àquela que está sendo acessada correntemente na *cache* é buscada antecipadamente sob certas condições [LEE 95]. Esta política antecipa a busca da linha i+1 quando a linha i estiver sendo referenciada. Existem as seguintes opções: pré-busca a linha i+1 incondicionalmente, pré-busca somente na falta da linha i ou pré-busca somente se a linha i é referenciada pela primeira vez na *cache*.

Com relação às faltas na *cache* causadas pelos desvios, uma solução é antecipar a execução dos mesmos [LEE 84] antes que o processador tenha efetuado a busca do caminho correto. Isto pode ser obtido com a transferência de instruções legais chamadas "antecipadores de desvios" (*delayed branches*), que são instruções que o processador deve necessariamente executar, para depois da instrução de desvio.

Com isso, enquanto o processador resolve a instrução de desvio, ele pode continuar buscando e executando um certo número de instruções legais, em tempo hábil de saber qual o caminho correto para o desvio e efetuar a busca sem perder tempo de processamento. Estas instruções legais podem ser retiradas do bloco de código anterior ao desvio, ou de um dos ramos do desvio, de modo que não alterem o processamento. Pesquisas mostram que o número de instruções legais obtidas nestas condições é pouco (aproximadamente uma instrução).

Ainda para os desvios, a alternativa mais utilizada é a previsão de desvios [JOH 91]**,** que consiste em antecipar qual caminho será definido na conclusão do desvio condicional,

mesmo antes da instrução de desvio ser executada. Normalmente, a instrução de desvio condicional já é detectada na busca de instruções, e uma previsão do caminho a seguir é feita dinamicamente. Quando a instrução de desvio passa a ser decodificada, o caminho previsto (tomado ou não-tomado) já pode ser buscado. Existe outra forma de previsão chamada estática mas que não é usual em processadores superescalares, a não ser aqueles do tipo VLIW [GON 97a]

Outra alternativa que elimina as previsões incorretas dos desvios condicionais, está em buscar e executar ambos caminhos do desvio (*multi-path execution)*. Após a solução do desvio, descarta-se somente o resultado do caminho incorreto. Esta abordagem tem sido usada em processadores escalares tais como o IBM 370/168 e IBM 3033.

Para seguir eficientemente ambos os caminhos nos desvios condicionais, o processador deve ter muitos recursos. Deve-se duplicar os recursos a cada desvio condicional encontrado, enquanto o processador determina o resultado do primeiro, fazendo que a necessidade de *hardware* cresça 8 ou 16 vezes para não sofrer as penalidades dos desvios [JOH 91]. Esta arquitetura possui *hardware* equivalente a uma arquitetura multi-tarefas simultâneas.

Mesmo com estas condições onerosas em termos de *hardware*, muitos trabalhos recentes têm utilizado a técnica de execução de múltiplos caminhos, com bons resultados. Em [AND 93] é proposta uma implementação desta técnica chamada *Two-Way Boosting*, onde a melhor configuração do modelo atinge um ganho de desempenho de 1.67x sobre um processador escalar. Em [SAN 97] também é proposta uma busca especulativa de múltiplos fluxos, cuja simulação mostrou uma redução de aproximadamente 74,27% da ocorrência de fila vazia. Pierce e Mudge [PIE 96] também fizeram algo parecido, chamado *wrong-path pre-fetching*, que pode atingir 14% de redução de ciclos perdidos em *cache* padrão e até 20% em casos mais restritos.

Outra alternativa bastante recente propõe o uso de uma técnica chamada de *Trace Cache*, como um complemento da *cache* de instrução convencional. Além de permitir que a *cache* seja mais eficiente e tenha menos faltas, também alivia a necessidade de decodificar trechos repetidos de códigos. Esta técnica mantém uma *cache* de instruções com trechos de código (*traces*) já decodificados, e para programas inteiros SPEC92, Rotenberg, Bennett e Smith [ROT 96] mostraram que uma *trace cache* de 4Kb melhora o desempenho em média 28% sobre uma busca seqüencial convencional.

2.2.1.1 Previsão Dinâmica de Desvios

A previsão dinâmica de desvios é a técnica mais utilizada nos processadores atuais para reduzir a penalidade causada pelos desvios condicionais. Existem vários mecanismos de previsão de desvios, que se diferenciam pelo algoritmo utilizado, mas, de uma forma geral, todos necessitam executar os quatro passos seguintes [SMI 95]:

Etapa 1: Reconhecimento de um desvio condicional: geralmente usa-se uma lógica de pré-decodificação antes do carregamento das instruções na *cache* (*cache* inteligente), que gera *bits* de pré-decodificação e os armazena na *cache* juntamente com

as instruções, para ser posteriormente analisados pela fase de busca. Os processadores MIPS R10000 e AMD K5 utilizam este tipo de pré-decodificação.

Etapa 2: Determinação do resultado do desvio (tomado ou não-tomado): muitas vezes os dados de um desvio não estão disponíveis quando o mesmo é executado, necessitando assim de um mecanismo de previsão. A tabela de endereços alvos ou BTB (*branch-target buffer*) tem sido usada como mecanismo para previsão de desvios e de busca de instruções, armazenando o histórico dos desvios. Lee e Smith [LEE 84] discutem possíveis implementações de BTB, envolvendo custo, taxa de acerto, endereçamento e organização. Bray e Flynn [BRA 91] analisam a utilização BTB fora e dentro da *cache* de instruções.

Etapa 3: Cálculo do endereço alvo do desvio: geralmente requer uma adição de inteiros. Na maioria das arquiteturas, um endereço alvo de um desvio é relativo ao contador de programa (PC) e usa um deslocamento (*offset*) contido na própria instrução, o que elimina a necessidade de leitura de registrador. Este endereço pode ser encontrado na BTB, caso contrário, se o desvio estiver sendo executado pela primeira vez, o tempo de obtenção do endereço pode ser maior.

Etapa 4: Transferência do controle redirecionando a busca: Quando o desvio é tomado, existe geralmente uma perda de ciclos de processamento na transferência de controle (reconhecimento do desvio, atualização do contador de programa, busca de instruções no endereço alvo). Decodificar a instrução de desvio condicional, extrair o endereço alvo da instrução e aplicar este endereço a *cache*, gasta muito mais tempo do que se o *hardware* detectar e prever o desvio durante a busca de instruções. *Hardware* que usa previsão de desvio condicional, supera a penalidade do desvio condicional mais eficientemente do que o uso de “antecipadores de desvio” (ver seção anterior) por *software*.

2.2.1.1.1 Estruturas e Algoritmos para Previsão de Desvios

Conforme dito anteriormente, a previsão dinâmica é feita através da BTB que registra o histórico dos desvios. Esta tabela é usada para prever qual o caminho a ser seguido por uma instrução de desvio condicional, baseado nos resultados anteriores, contando com o fato de que um desvio condicional é estável por um certo período de tempo.

Tradicionalmente, uma BTB é organizada como uma *cache* onde cada entrada consiste do endereço da instrução de desvio, de um campo para previsão e do endereço alvo do desvio. Uma possível alternativa para a previsão é armazenar um contador junto a cada entrada na BTB [SMI 95], e quando um desvio é tomado, o contador é incrementado (até um valor máximo), caso contrário ele sofre um decremento (até um valor mínimo). Desta forma, a BTB consegue manter o resultado dominante para cada desvio. Usualmente este contador possui um ou dois *bits*.

Este tipo de organização é chamado de BTB acoplada. O Intel Pentium utiliza uma BTB acoplada onde somente os desvios tomados são armazenados, e portanto somente estes podem ser previstos dinamicamente. Cada entrada possui um campo de 2-bits para

previsão e na ocorrência de uma falta na BTB é utilizada uma previsão fixa onde o caminho escolhido é o não-tomado [LEE 95].

Em alguns processadores, a previsão é feita em 2 fases, como no caso do PowerPC 620 [DIE 95]: a primeira no estágio de busca e a segunda no estágio de decodificação. O PowerPC 620 tenta resolver a previsão de um desvio já no estágio de busca baseado somente no endereço do desvio, usando para isto uma tabela chamada BTAC (*cache* associativa de 2 vias que contém somente 256 entradas) para obter o endereço alvo do desvio. Posteriormente, o estágio de decodificação realiza outra previsão usando a BHT (tabela de mapeamento direto de 2048 entradas), que faz a previsão baseada em 2 bits com maior segurança que na BTAC.

Se a previsão feita pela BHT difere da previsão feita pela BTAC, a previsão feita pela BTAC é descartada e a busca prossegue no caminho previsto pela BHT. Ambas BHT e BTAC são atualizadas com o endereço do caminho escolhido. A tabela 2.1 mostra as estatísticas dos resultados obtidos com esta técnica.

TABELA 2.1 - Utilização de BHT e BTAC no PowerPC 620

Processamento do desvio		**Média de acerto**
Resultado Obtido	não-tomado	25.05%
	tomado	74.95%
Previsão na BTAC	correto	85.65%
	incorreto	14.35%
Previsão na BHT	resolvido	24.71%
	correto	68.27%
	incorreto	7.02%
média final de acerto		92.98%

Em [SMI 81], Smith avalia várias técnicas de previsão de desvios, onde a melhor técnica atinge uma média de 92,55% de acerto sobre um conjunto de programas. A tabela 2.2 mostra a média de acerto das técnicas por ele apresentadas, que são brevemente descritas a seguir:

- **n.1** - Prevê que todos os desvios serão tomados

- **n.1a** - Prevê que somente certos códigos de operação de desvios serão tomados. Os demais serão não-tomados.

- **n.2** - Sempre prevê que um desvio será decidido de acordo com a sua última execução (um *bit* de história). Se ele ainda não foi executado, a previsão será para tomado.

- **n.3** - Prevê que somente desvios para posições anteriores da memória serão tomados (*backward taken*). Os demais serão não-tomados (*forward not-taken*).

- **n.4** - Mantém uma tabela com os mais recentes desvios não-tomados. Prevê que somente desvios encontrados na tabela serão não-tomados. Caso contrário, serão tomados. As entradas podem ser retiradas da tabela caso sejam tomadas, ou por falta de espaço durante a inserção de novas entradas (usa política LRU para a substituição).

- **n.5** - Mantém um *bit* de historia para cada instrução na *cache*. Funciona como a política 2, só que na *i-cache*.

- **n.6** - Mantém uma memória (*history table*) acessada por uma função (*hash*) do endereço do desvio. Cada entrada contém um *bit* de história. Contém os resultados dos desvios mais recentes. A função sempre retorna um endereço na tabela.

- **n.7** – Mesmo que a estratégia 6 mas usa um contador ao invés de único *bit* de história. O contador sofre incremento e decremento na medida que os desvios são tomados e não-tomados, respectivamente. Usa o *bit* de sinal do contador para prever.

Dez anos depois de Smith, Yeh e Patt [YEH 91] propuseram um previsor de desvios adaptativo de dois níveis (*two-level adaptative training*), cujo algoritmo de previsão se altera de acordo com informações em tempo de execução. Várias configurações desta técnica foram simuladas e comparadas com outros esquemas estáticos e dinâmicos. Em nove programas SPEC, esta técnica obteve 97% de corretismo nas previsões, contra 93% (melhor caso) das demais técnicas. Esta alternativa é também chamada de BTB disjunta, onde a informação de previsão é contida em uma estrutura separada chamada Tabela de Comportamento ou PHT (*pattern history tabela*). Neste caso, todos os desvios podem ser previstos dinamicamente.

TABELA 2.2 - Taxas de Acertos Segundo Smith

Estratégia	**Acerto (%)**
n.1	76.68%
n.1a	86.70%
n.2	90.40%
n.3	80.28%
n.4 c/ 8 entradas	90.4%
n.6 c/ 16 wds	90.41%
n.7 c/ 16 wds, 2 bits	92.55%

Existem diferentes implementações de PHT, onde o exemplo mais simples é o método degenerado. Neste método, a PHT contém 2^k entradas e utiliza um registrador de deslocamento de k-*bits* (registrador de história global). Este registrador contém os k-últimos resultados dos desvios, com 1 para tomado e 0 para não-tomado, e serve como indexador da PHT para os desvios que não estão na PHT. Os desvios na PHT são indexados pelos seus respectivos endereços [LEE 95]. Outros trabalhos investigando previsão de desvios foram desenvolvidos por McFarling [MCF 93], Young [YOU 95], Uht [UHT 97] e recentemente Kesller [KES 99].

TABELA 2.3 - Previsão de Desvios nos Processadores Atuais

Processador Comercial Atual	**Previsão de Desvios**
Intel i486	estática: sempre não-tomado
Sun SuperSparc	estática: sempre tomado
HP PA-7x00	estática: BTFN - tomado para trás e não-tomado para frente
DEC Alpha 21064, AMD-K5	dinâmica: 1 bit de previsão
NexGen 586, PowerPC 604, Cyrix 6x86, Cyrix M2, MIPS R10000	dinâmica: 2 bits de previsão
Intel Pentium Pro, AMD-K6	dinâmica: adaptativo em dois níveis
DEC Alpha 21264	dinâmica (seletor) e híbrida

McFarling desenvolveu um mecanismo de previsão de desvios que combinando outros mecanismos permite uma taxa de acerto de até 98.1%. Young analisou esquemas para previsão de desvios correlacionados. Uht apresentou um resumo das técnicas usadas em alguns dos processadores comerciais atuais, conforme mostra a tabela 2.3. Kesller

apresentou o mecanismo de previsão do Alpha 21264, o qual é um exemplo de algoritmo complexo aplicado a um processador real.

2.2.1.2 Políticas de Execução Especulativa

Um desvio mal-previsto (*branch mispredict*) ocorre quando um endereço alvo incorreto é previsto para o desvio, fazendo com que o processador busque instruções que não devem ser executadas (*wrong path*). Como isso, o desempenho do sistema de memória pode ser afetado por dois fatores, caso ocorra uma falta na *cache* de instruções durante esta busca [LEE 95]:

1) instruções úteis da *cache* podem ser substituídas por instruções incorretas, poluindo a *cache* e aumentando a probabilidade de novas faltas;

2) o barramento entre a *cache* e o próximo nível na hierarquia de memória pode estar ocupado com a busca do caminho incorreto, enquanto outra falta na *cache* sobre um caminho correto precisa ser processada.

Estes dois fatores proporcionam um maior prejuízo de tempo, e de quantidade de instruções executadas, pois os caminhos corretos podem não compensar os ciclos perdidos com os caminhos incorretos, então o desempenho diminui [DWY 92]. No PowerPC 620 [DIE 95] as taxas de má previsão e de faltas na *cache* podem ser vistas na tabela 2.4.

TABELA 2.4 - Taxas de Mal-Previsão e Faltas na *I-cache* do PowerPC 620

Programas	**Má-Previsão**	**Falta na *Cache***
Ocorrência Média	6.36%	0.12%

As instruções que são executadas antes que o processador tenha certeza que elas realmente devem ser executadas, são chamadas especulativas, por que o processador não sabe se de fato elas devem ou não ser executadas. Algumas políticas podem ser usadas durante a execução de um caminho especulativo para tratar a falta na *cache* e tentar aliviar a problemática [LEE 95]: Milagrosa, Otimista, Resume e Pessimista.

- **Política Milagrosa**: Em uma primeira análise, parece ideal atender uma falta na *cache* somente se o programa estiver sendo executado sobre o caminho correto. Isto evita a poluição da *cache* com o caminho incorreto e reduz a quantidade de tráfego de memória. A política que possui estas características é conhecida por Milagrosa e é impossível de ser implementada. Ela é considerada como o limite máximo no desempenho entre as políticas analisadas.

- **Política Otimista**: Uma falta na *cache* é sempre atendida pois esta política assume que a previsão de desvios é boa e que a execução de um caminho incorreto é rara. Além disso, as instruções buscadas ao longo de um caminho incorreto podem ser utilizadas posteriormente, servindo como uma pré-busca. Mas ainda assim podem ocorrer desvios mal-previstos, e nesse caso, a política Otimista pode perder desempenho quando ocorrer uma falta na *cache* para um caminho que não precisa ser buscado. Para tentar reduzir esta perda, a política Resume pode ser utilizada.

- **Política Resume**: permite que o processador permaneça em execução mesmo quando existe uma falta na *cache* que está pendente para um caminho incorreto. Para isso, quando a linha faltante da *cache* for buscada da memória, ela é guardada em uma estrutura auxiliar (*resume buffer*), ao invés de ser colocada diretamente na *cache*. Isto evita poluir o caminho correto que já se encontra na *cache*. O armazenamento da linha na *cache* ocorrerá na próxima falta, sem interferência com a operação normal da *cache*. Nas faltas subseqüentes, o índice da linha faltante e o índice na estrutura auxiliar serão comparados para evitar acesso desnecessário à memória, caso sejam iguais.

- **Política Pessimista**: evita acessos desnecessários à memória, a menos que as instruções sejam utilizadas. Isto evita a poluição da *cache* e mantém o barramento livre para um acesso estritamente necessário. Ela espera que todos os desvios pendentes sejam resolvidos e que todas as instruções anteriores sejam decodificadas, e busca somente no caminho correto.

2.2.2 Decodificação e Despacho de Instruções

Este estágio do *pipeline* se responsabiliza pela decodificação das instruções que estão na fila de instruções, e pelo despacho destas para as filas de remessa juntas às unidades funcionais, para posterior remessa e execução. Segundo Smith [SMI 95], a tarefa da decodificação inclui a geração de uma tupla para cada instrução decodificada, contendo basicamente a operação a ser executada, a identificação dos elementos de armazenamento onde os operandos residem e a identificação da localização onde o resultado será colocado.

Durante o despacho, o processador pode alocar entradas na fila de reordenação, nas filas de remessa e se necessário, na fila de renomeação do registrador destino [DIE 95]. Como as instruções podem ser despachadas para diferentes unidades funcionais e pelo fato destas instruções poderem ser executadas na medida que estiverem prontas, essas instruções podem então ser executadas fora-de-ordem (*out-of-order*), ou seja, em outra ordem qualquer que não seja a original do programa.

A largura de despacho, isto é, o número máximo de instruções que podem ser despachadas simultaneamente é uma característica específica de cada processador. A tabela 2.5 mostra estas larguras em alguns dos principais processadores atuais, onde se nota a predominância de 4 instruções por ciclo. Com isso conclui-se que o IPC máximo destes processadores jamais será maior que 4, e no caso do referido Pentium, jamais será maior que 2.

TABELA 2.5 - Largura de Despacho nos Processadores Atuais

Processador	**Larg. Desp.**
Pentium	2
R10000	4
AMD K5	4
PowerPC 620	4

Esta fase também se responsabiliza pela detecção e controle das dependências verdadeiras (RAW), bem como da eliminação das falsas dependências (WAW e WAR), e pode usar alguma variação do método de renomeação de registradores [MIP 95] ou do

algoritmo de Tomasulo [TOM 67, FER 92, GON 98c], que são detalhados nas seções seguintes. As estruturas envolvidas neste estágio são visualizadas na figura 2.7.

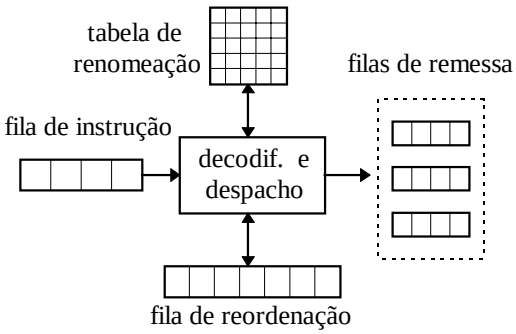


FIGURA 2.7 - Representação da Fase de Decodificação e Despacho

Durante a decodificação, uma série de condições deve ser satisfeita para que diferentes instruções possam ser executadas em paralelo. O Pentium, por exemplo, pode decodificar e executar 2 instruções consecutivas simples I1 e I2. Se ambas não podem ser executadas em paralelo, elas seguem em seqüência. Para a execução paralela no Pentium, 4 condições devem ser satisfeitas [SAI 93]: 1) I1 e I2 devem ser instruções simples; 2) I1 não pode ser uma instrução de desvio; 3) O destino de I1 não pode ser fonte de I2 e 4) O destino de I1 não pode ser destino de I2.

A fila de reordenação serve para garantir a reordenação de instruções executadas fora-de-ordem, e é implementada segundo o modelo FIFO (*first in first out*). Sempre que uma instrução é despachada, uma entrada para esta instrução é inserida no final da fila de reordenação, e quando esta mesma instrução conclui sua execução, seu resultado é também colocado nesta mesma entrada. Note que este resultado pode ser um código de erro também. Quando há uma seqüência ininterrupta de instruções concluídas com término normal no início da fila de reordenação, estas entradas podem ser retiradas da fila (*commit*) e os resultados são gravados nos registradores (*write-back*) [JOH 91]. Pode-se concluir que o paralelismo obtido na execução do programa se resume ao número médio de instruções retiradas da fila de reordenação por unidade de tempo. A figura 2.8 [SMI 95] ilustra a fila de reordenação.

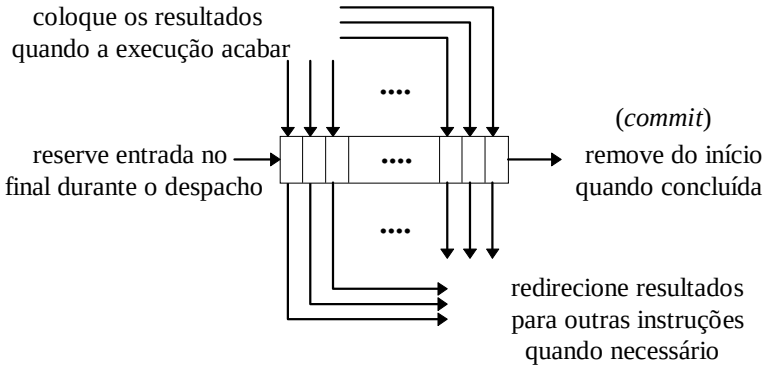


FIGURA 2.8 - Estrutura da Fila de Reordenação

Uma instrução entra na janela de instruções quando ela é despachada, e sai quando ela é concluída. A janela de instruções [JOU 95b] pode ser considerada como uma abstração

da fila de reordenação, pois essa limita a quantidade de instruções despachadas. Quando a janela de instruções aumenta, um maior número de instruções paralelas pode ser processado. Mas cada entrada adicional na fila de reordenação implica em maior custo de *hardware*.

Algumas implementações superescalares estabelecem uma grande janela de instruções para extrair a maior quantidade de paralelismo possível através da execução fora-de-ordem, tornando a arquitetura maior e complexa, como no caso da Metaflow SPARC. Outras arquiteturas exploram a simplicidade da arquitetura através de altas freqüências de processamento (*clock*), como no caso da DEC Alpha 21064, que decodifica 2 instruções por ciclo e despacha instruções em ordem [FRA 94].

A tabela 2.6 [JOU 95b] mostra o desempenho médio obtido em simulação com programas inteiros e de ponto flutuante SPEC92, dada em *IPC*, de acordo com o tamanho da fila de reordenação e a largura de *pipeline* (número máximo de instruções que podem ser buscadas, decodificadas, despachadas, executadas e concluídas por tempo). Nesta simulação, foi considerado um número infinito de unidades funcionais.

Comumente não existe a utilização de técnicas para o despacho de instruções da fila de instruções em arquiteturas superescalares, pois as instruções são sempre despachadas em ordem, a menos que as instruções sejam preparadas pelo compilador, tal como em arquiteturas VLIW.

TABELA 2.6 - Desempenho com Diferentes Janelas de Instruções

Programas com Inteiros do SPEC 92				
largura do *pipeline*	2	4	6	8
tamanho da janela	16	32	64	96
IPC	2.11	4.09	5.91	7.36
Programas com Ponto Flutuante do SPEC 92				
largura do *pipeline*	2	4	6	8
tamanho da janela	16	32	64	96
IPC	1.96	2.86	3.62	4.14

Usando uma abordagem similar Melvin e Shebanow [MEL 98] propõem um suporte de *hardware* para gerar instruções do tipo VLIW em arquiteturas superescalares. Nesta proposta, existe uma unidade de *hardware* dedicada, chamado de *Fill Unit*, que prepara as instruções longas antes de serem colocadas na *cache*. Isto permite que após a busca das instruções, as mesmas possam ser despachadas fora de ordem.

Na continuidade do trabalho de Melvin e Shebanow, Franklin e Smotherman [FRA 94] utilizam a *Fill Unit* para implementar uma arquitetura que combina as vantagens da compatibilidade de código, existente nas arquiteturas superescalares, com a ausência de decodificadores complexos para checagem de dependências, como nas arquiteturas VLIW. Algumas condições podem causar paradas nesta fase, e são descritas a seguir. Na seqüência, a tabela 2.7 mostra a freqüência de ocorrência destas condições para o PowerPC 620 [DIE 95]:

> **1. Restrições de execução seqüencial:** certas instruções (instruções de execução seqüencial) devem aguardar a conclusão de instruções anteriores a elas, para que as instruções posteriores possam ser despachadas.

2. Desvio espera por mtspr: alguns desvios acessam o contador durante o despacho. A instrução *mtspr* (*move to special purpose register*), que escreve neste contador causa uma dependência implícita de dados e força que os desvios esperem a finalização desta. Esta situação é também bastante rara.

3. Saturação de porta de leitura no banco de registradores: quando durante o despacho, uma porta necessária não está disponível.

4. Saturação das filas de remessa: não existe fila de remessa disponível na unidade funcional associada à instrução que está sendo despachada. Um trabalho que analisa este problema foi publicado em [GON 98b].

5. Despacho múltiplo para a mesma fila de remessa: uma fila de remessa possui somente uma porta de escrita e impossibilita múltipla escrita.

6. Saturação da fila de renomeação: não existe entrada disponível na fila de renomeação. Para cada instrução despachada, seu registrador destino deve ser renomeado. Esta estrutura é vista na seção 2.2.5.1.

7. Saturação do fila de reordenação: não existe entrada disponível na fila de reordenação.

TABELA 2.7 - Freqüência das Condições que Impedem o Despacho

condição da parada	média de ocorrência
1	0.00%
2	1.41%
3	1.37%
4	32.34%
5	16.47%
6	7.63%
7	14.66%
sem parada	26.12%

Apesar destas arquiteturas possuírem *hardware* para a execução simultânea de várias instruções, existe a necessidade de mecanismos que escalonem as instruções de forma paralela e eficiente, que é mais bem detalhado na seção 2.2.5.

2.2.3 Remessa e Execução de Instruções

Conforme mostra a figura 2.9, após as instruções serem despachadas para as filas de remessa, elas são remetidas para as unidades funcionais, se já estiverem prontas sem operandos não-resolvidos. Já nas unidades funcionais, as instruções são executadas em um ou mais ciclos do processador, dependendo do tipo da instrução (inteiros, ponto-flutuantes, desvios, memória), e podem acessar a memória de dados. No final da execução, os resultados geralmente são enviados para todas as instruções que estão esperando nas filas de remessa e para as respectivas entradas do fila de reordenação.

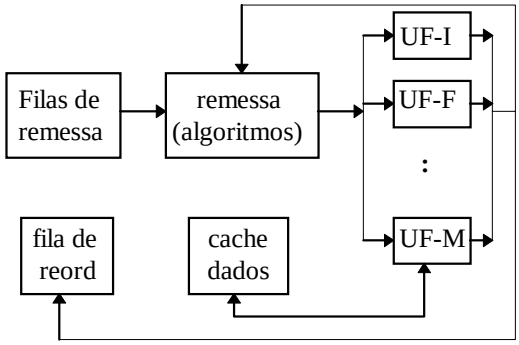


FIGURA 2.9 - Representação da Fase de Remessa e Execução

Assim sendo, o desempenho de uma arquitetura superescalar durante a remessa e execução das instruções depende diretamente dos seguintes fatores: topologia das filas de remessa, algoritmo de remessa, configuração das unidades funcionais e de outras questões de acesso à memória. Estas estruturas e algoritmos são discutidos nas próximas seções.

2.2.3.1 Topologias para as Filas de Remessa

Smith e Sohi [SMI 95] descrevem três possíveis configurações para as filas de remessa: fila única, onde a remessa é em-ordem para qualquer unidade funcional; múltiplas filas, organizadas de acordo com o tipo das instruções que elas armazenam, onde a remessa em cada fila é em ordem, mas entre as filas a remessa pode ser fora-de-ordem; e múltiplas filas de estações de reserva, onde todas as instruções podem ser remetidas fora de ordem entre e dentro das filas. A princípio, as topologias podem ser configuradas de diferentes maneiras, utilizando várias filas em ou fora de ordem. As diferentes topologias das filas de remessa podem implicar em:

- Distribuir as filas de remessa tende a diminuir a complexidade da "arbitragem" entre as instruções prontas para serem remetidas.

- Unificar as filas de remessa permite melhor aproveitar suas entradas e assim reduzir seu tamanho.

Cinco dos mais recentes microprocessadores utilizam diferentes topologias [JOU 95a]: O P6 da Intel utiliza uma fila de remessa unificada, para todas as operações (T). O PA-8000 da Hewlett-Packard utiliza 2 sub-filas: a primeira para operações de memória (M) e a segunda para inteiros, desvios e ponto-flutuantes (IDF). O R10000 da MIPS utiliza 3 sub-filas: para operações de inteiros e desvios (ID), ponto-flutuante (F) e de memória (M). O HaL Sparc64 utiliza 5 sub-filas: inteiros (I), flutuantes (F), memória (M), desvios (D) e endereços (A). A IBM e Motorola distribuem a fila de remessa entre cada uma das unidades funcionais do PowerPC.

Jourdan [JOU 95a] simulou o desempenho desta 5 configurações topológicas. No esquema unificado, as instruções são despachadas para uma fila de remessa central que contém todas as instruções que estão esperando para serem remetidas. No esquema

semi-unificado, a fila de remessa pode ser dividida em várias partes, onde cada uma pode ser associada a um conjunto de unidades funcionais, iguais ou distintas. Na topologia distribuída, a fila de remessa é quebrada em estações de reserva, onde cada uma delas é associada a uma única unidade funcional.

Em suas simulações, Jourdan variou o tamanho da janela de instruções (J), que indica o tamanho da fila de reordenação, e a largura de despacho (L). A tabela 2.8 resume os tamanhos ótimos para diferentes configurações topológicas de filas de remessa. Jourdan mostrou que tamanhos superiores não implicam em aumento substancial de desempenho.

Qualquer que seja a topologia, cada barramento de despacho deve ser ligado a todas as entradas da fila de remessa. Como o número de entradas nesta fila diminui quando a topologia se aproxima de uma configuração unificada, o procedimento de despacho também se torna mais simples e mais rápido [JOU 95a].

TABELA 2.8 - Tamanhos Ótimos para as Topologias de Remessa

L	J	1 fila	2 sub-filas		3 sub-filas			5 sub-filas					distribuída
		T	**IDF**	**M**	**ID**	**F**	**M**	**I**	**D**	**F**	**A**	**M**	
4	32	28	24	12	20	24	12	16	8	24	8	12	76
6	64	48	36	24	36	32	24	32	16	32	16	24	134
8	96	72	52	36	52	44	36	48	24	44	24	36	198

Assim, para reduzir a complexidade e o tempo de despacho em uma topologia não-unificada, um limite pode ser aplicado ao número de instruções que podem ser despachadas para cada parte da fila de remessa. Mas as perdas de desempenho são altas. Em compensação, em uma topologia distribuída, o procedimento de remessa, ou seja, a arbitragem para os acessos das unidades funcionais, é totalmente simples. Mas na medida que 2 ou mais unidades compartilham várias entradas, a arbitragem torna-se mais complexa e consome mais tempo. Outro trabalho apresentado em [GON 98b] analisa a utilização de fila única e mostra que o aumento de desempenho pode ser substancial quando os fluxos de instruções são balanceados.

2.2.3.2 Políticas de Remessa

A política de remessa (*issue policy*) é um algoritmo que gerencia o envio das instruções da fila de remessa para as unidades funcionais, que pode ser feito fora-de-ordem. Ela pode ser definida como uma checagem, em tempo de execução, pela disponibilidade de dados e recursos [SMI 95]. Idealmente, uma instrução está pronta para executar tão logo seus operandos estejam disponíveis. Contudo, em um processador superescalar existem várias condições que podem impedir a remessa de instruções para as unidades funcionais [DIE 95]. Algumas dessas condições são mostradas a seguir e as freqüências de ocorrências das mesmas, para o PowerPC 620, são mostradas na tabela 2.9.

1. **Desabilitação da execução fora-de-ordem:** algumas instruções devem ser remetidas em ordem, como no caso das instruções de desvio e de ponto-flutuante.

2. **Restrições de execução seqüencial:** certas instruções (instruções de execução seqüencial) devem aguardar a conclusão de instruções anteriores antes de iniciarem a sua execução, e todas as instruções posteriores a ela só podem ser

despachadas após a sua conclusão. Geralmente, são instruções que escrevem sobre registradores não renomeados (dependência de saída não eliminada) ou que realizam I/O.

3. **Espera por operandos fontes:** os operandos ainda não foram determinados por outras instruções precedentes.

4. **Espera por unidades de execução:** quando a unidade de execução solicitada não está disponível ou quando várias instruções utilizam a mesma unidade funcional. No segundo caso, somente uma instrução é remetida.

TABELA 2.9 - Freqüência das Condições que Impedem a Remessa

condição da parada	média de ocorrência
1	1.90%
2	3.15%
3	23.58%
4	5.80
sem parada	65.57%

Outras condições envolvendo barramentos e portas para o arquivo de registradores são extremamente importantes e devem ser consideradas [SMI 95].

Butler e Patt [BUT 92] avaliaram várias técnicas de remessa de instruções, considerando fila de remessa com topologias unificada e distribuída. Para cada unidade funcional, o *hardware* de remessa deve escalonar uma instrução pronta, dentro de um conjunto de instruções prontas disponíveis, utilizando para isto uma heurística de seleção. A seguir são apresentadas 8 técnicas.

1. **Caminho do Desvio**: As instruções que estão no caminho da solução de um desvio, recebem maior prioridade. Na impossibilidade, usa-se a política “A Mais Antiga Primeiro”.

2. **Comprimento da Cadeia**: Cada instrução recebe uma prioridade, de acordo com o tamanho máximo da cadeia de dependentes (correntemente na máquina) da respectiva instrução. Na impossibilidade, usa-se “A Mais Antiga Primeiro”.

3. **Prioridade Fixa**: É previamente definida uma prioridade para cada entrada da fila de remessa.

4. **Somente a Cabeça**: Somente a instrução mais velha (na “cabeça” da fila de remessa) é checada para um possível escalonamento.

5. **Dependência Máxima**: As instruções recebem prioridades baseadas no número de instruções dependentes que residem “correntemente” na fila de remessa (isto é, o número de instruções esperando pelo resultado que ainda será produzido).

6. **A Mais Velha Primeira**: Uma dada instrução tem maior prioridade do que qualquer outra que a segue na ordem de execução seqüencial.

7. **Rotação Aleatória**: Seleciona-se um “ponto de partida” aleatoriamente na fila de remessa, e o escalonamento prossegue em rotação até que uma instrução pronta seja encontrada. É similar ao “Prioridade Fixa”, exceto que o ponto de partida é aleatório.

8. **Aleatória**: As instruções são escolhidas de forma aleatória dentro do conjunto de instruções prontas disponíveis.

Nota-se que várias destas técnicas (caminho do desvio, comprimento da cadeia e dependência máxima) precisam de suporte de compilação para fornecer informação de prioridade. Nas simulações apresentadas em [BUT 92], foi suposto que o *hardware* sempre dispunha destas informações.

Os resultados das simulações mostram que a maioria dos algoritmos de escalonamento dinâmico obtém desempenho similar. Isto é devido ao número tipicamente baixo de instruções prontas esperando para execução. Verificou-se também que uma fila de remessa unificada causa pouca melhoria no desempenho de inteiros, mas melhora o desempenho de ponto-flutuante cerca de 25%.

A figura 2.10 mostra em histograma o número médio de instruções prontas em cada parte da fila de remessa distribuída e na unificada, para o programa *gcc*. Como pode ser visto, a distribuição das instruções prontas é brutalmente inclinada para 0 e 1, de modo que existe pouca oportunidade para que as diferentes técnicas gerem diferentes escalonamentos (*schedules*).

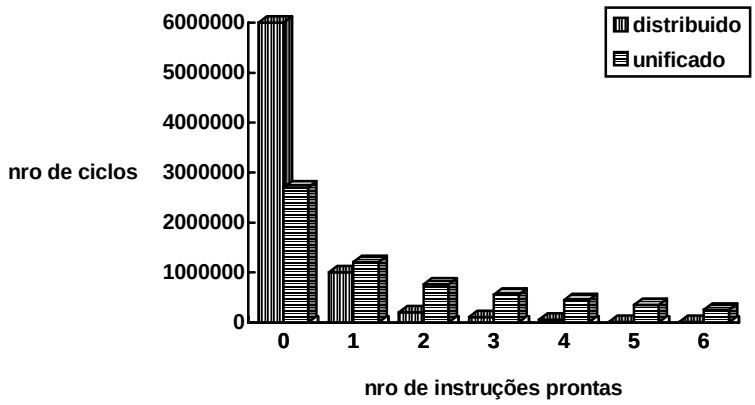


FIGURA 2.10 - Freqüência de Instruções Prontas na Remessa para o *gcc*

2.2.3.3 Configuração das Unidades Funcionais

Os processadores atuais apresentam diferentes quantidades de unidades funcionais O balanceamento das unidades funcionais implica na determinação dos números e tipos das unidades funcionais, de forma a proporcionar um melhor desempenho na execução das aplicações.

O ideal seria a utilização de unidades funcionais homogêneas, para permitir a execução de qualquer tipo de instrução. Mas esta decisão implica em aumento de custo e de complexidade, de forma desnecessária, da mesma forma que o tempo de execução pode

também ser elevado em unidades que poderiam executar simples cálculos. De fato, esta atitude é inaceitável.

Preocupados com esta questão, Jourdan, Sainrat e Litaize [JOU 95b] tentaram estabelecer critérios para a configuração das unidades funcionais, de forma determinar o melhor custo/benefício para processadores superescalares com “múltiplas remessas fora de ordem” (*multiple-issue out-of-order*). Foram executadas simulações em programas do SPEC92 (6 inteiros e 14 de ponto-flutuante), cujos resultados principais são descritos nesta seção.

A tabela 2.10 mostra o *IPC* médio das simulações sobre os programas SPEC92 de inteiros (Cint 92) e de ponto-flutuantes (Cfp 92), de acordo com o número ótimo de unidades de inteiros (UI) e de portas de memória (PM), para processadores com diferentes larguras de despacho. Para esta simulação, foi considerado um número infinito de unidades de ponto flutuante.

Depois de estabelecido o número ótimo de unidades de inteiros para cada largura do processador, foi variado o número de unidades de ponto flutuante. Foi definida a unidade de ponto flutuante (FPU) como sendo uma unidade capaz de executar todos os sub-grupos de instruções em ponto-flutuante. A tabela 2.11 resume as configurações ótimas.

Nas tabelas 2.10 e 2.11 nota-se que para um processador de largura 2, isto é, um processador que despacha até 2 instruções por ciclo, não é necessário mais do que 2 unidades de inteiros e 1 unidade de ponto flutuante para que o desempenho se estabilize. Da mesma forma, para processadores de largura 8, não são necessárias mais do que 5 unidades de inteiros e 2 de ponto flutuante. Nota-se que a proporção do número de unidades de inteiros pelo número de unidades de ponto flutuante é algo do tipo 2**:**1.

TABELA 2.10 - IPC Médio do SPEC 92 Variando Unidades de Inteiros

largura do processador	2	4	6	8
número de UI / PM	2 / 1	3 / 2	4 / 3	5 / 4
IPC (Cint 92)	2.0	3.76	5.44	6.69
IPC (Cfp 92)	1.83	2.74	3.49	4.02

Estes resultados foram comparados com a configuração executiva do PowerPC 604, que usa fila de reordenação de 16 entradas e largura 4. A melhor configuração relatada por Jourdan, Sainrat e Litaize é a do processador de grau 8 com *cache* de 4 portas e 3 unidades de inteiros, que oferece um ganho de 2.86 sobre a configuração citada do PowerPC 604, para instruções do Cint 92. Para instruções do Cfp 92, o melhor ganho foi de 2.09.

TABELA 2.11 - IPC Médio do SPEC 92 Variando Unidades de P. Flutuante

largura do processador	2	4	6	8
número de FPU	1	2	2	2
IPC (Cfp 92)	1.70	2.63	3.29	3.68

2.2.3.4 Questões de Acesso à Memória

Entre os fatores que mais degradam o desempenho de um processador superescalar está o acesso a memória, que inibe a execução de muitas instruções com dependências de dados durante muitos ciclos do processador. Para reduzir as latências das operações de memória, são utilizadas memórias *cache* hierárquicas, com uma memória pequena e rápida no primeiro nível (*cache* primária) e uma outra maior e mais lenta no segundo nível (*cache* secundária). Nestes modelos, espera-se que o dado seja encontrado na memória *cache* primária [SMI 95].

Diferente das instruções aritméticas, que permitem a identificação dos operandos na decodificação, a maioria das instruções de leitura e escrita em memória não permite a identificação das localizações de memória na fase de decodificação. A determinação da localização de memória que será acessada requer um cálculo de endereço, geralmente uma adição de inteiros. Assim, estas instruções são remetidas para uma fase de cálculo de endereços. Após este cálculo, o endereço é convertido para um endereço físico, usualmente através de uma tabela de conversão de endereços chamada TLB (*translation lookaside buffer*). Uma vez que o endereço físico está calculado, a operação de leitura/escrita pode ser submetida à memória. Em muitas implementações estas duas operações podem ser sobrepostas: a *cache* é acessada juntamente com o cálculo do endereço para posterior comparação para se saber se o endereço está presente.

Alguns processadores superescalares permitem somente uma única operação de memória por ciclo, mas isto está se tornando um gargalo no desempenho. Para permitir múltiplos acessos à memória, a hierarquia de memória deve ter multi-portas, sendo geralmente suficiente que isto seja feito somente na *cache* primária, pois a *cache* secundária é acessada menos constantemente [SMI 95].

Devido a grande quantidade de operações de memória, não é prático permitir que estas operações sejam executadas fora-de-ordem usando as mesmas técnicas para operações de registradores. Para permitir sobreposição destas instruções, usa-se uma fila de endereços de escrita (*store address buffers*) para armazenar os endereços de todas as escritas pendentes. Antes de uma operação de escrita ser remetida para a memória, esta fila é checada para saber se existem operações pendentes para o mesmo endereço da instrução a ser executada.

Os endereços do escrita permanecem nesta fila até: 1) o dado estar disponível e 2) a instrução de escrita estar pronta para ser retirada da fila de reordenação (para garantir interrupções precisas). Os endereços das operações de leitura são checados com os endereços da fila de endereços de escrita. Se existe correspondência, a operação de leitura deve também esperar pela finalização da escrita, em uma fila de endereços de leitura. A figura 2.11 mostra uma possível implementação simplificada para este método.

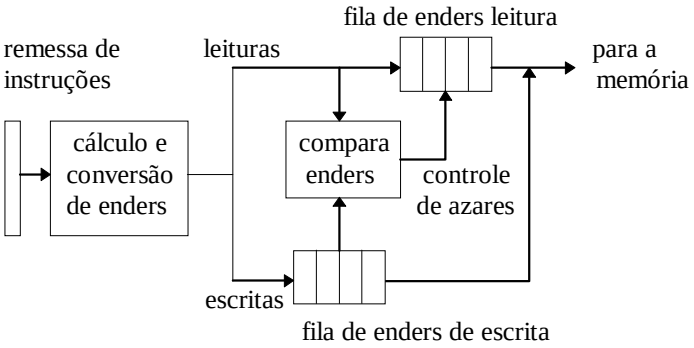


FIGURA 2.11 - Esquema das Filas de de Leitura/Escrita em memória

2.2.4 Conclusão da Instrução

A fase final da vida de uma instrução é a conclusão (*commit*), onde o resultado da instrução modifica o estado lógico do processador. A proposta desta fase é implementar a aparência do modelo de execução seqüencial, ainda que a execução real seja especulativa e fora-de-ordem. Existem 2 tipos de estados de uma arquitetura superescalar [SMI 95]:

- estado físico: é o estado que está sendo atualizado continuamente na medida em que as operações são executadas. Como as instruções podem ser executadas especulativamente, este estado pode não representar o estado correto da arquitetura, pois muitos resultados indevidos podem ter sido gerados.
- estado lógico ou arquitetural: é estado que deve ser recuperado até o momento de uma interrupção. Este estado representa o estado correto da arquitetura, ou seja, o estado em que a arquitetura deveria estar se o programa tivesse sido executado seqüencialmente até o momento da interrupção, onde nenhuma instrução inválida pode interferir nos resultados.

Para forçar um gerenciamento preciso de interrupção, somente é permitido a uma instrução completar quando todas as instruções anteriores não puderem mais produzir nenhuma interrupção. As principais instruções que podem produzir interrupções são desvio condicional, divisão e acesso a memória [JOU 95a]. Existem pelo menos duas técnicas para recuperar o estado da arquitetura, de forma precisa.

Na primeira técnica, os resultados das instruções atualizam imediatamente o estado físico da máquina, no próprio arquivo de registradores da arquitetura. Esta técnica utiliza uma estrutura auxiliar chamada tabela de história (*history buffer*) para armazenar o estado lógico, que é registrado nesta estrutura, de tempos em tempos, quando se tem certeza que os resultados não são mais especulativos.

A tabela de história pode ser tratada sob diferentes nomes: em [CHE 96] é chamada de “arquivo de registradores futuros” e em [JOU 95a] ela é conhecida como tabela de atualização (*update buffer*). Uma fila de reordenação pode ser utilizada para se

determinar quais instruções já não são mais especulativas e podem ser gravadas na tabela de história. A figura 2.12 mostra a fase de finalização segundo esta técnica.

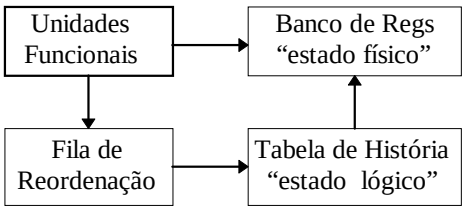


FIGURA 2.12 - Conclusão com Tabela de História

A segunda técnica utiliza a própria fila de reordenação para recuperar o estado lógico do processador. Nesta técnica, o estado físico ou especulativo é armazenado diretamente nos campos da fila de reordenação, imediatamente quando as instruções são executadas. Já o estado lógico, que é mantido no próprio arquivo de registradores da arquitetura, é atualizado na ordem seqüencial do programa, quando as instruções são removidas da fila de reordenação por não serem mais especulativas.

Os resultados são gravados no banco de registradores da arquitetura (e na memória no caso de escrita) e o espaço é liberado na fila de reordenação. Também nesta fase, um sinal é enviado para fila de endereços de escrita (vide seção 2.2.3.4) e o dado é armazenado na memória. A figura 2.13 mostra a representação da fase de conclusão segundo esta técnica.

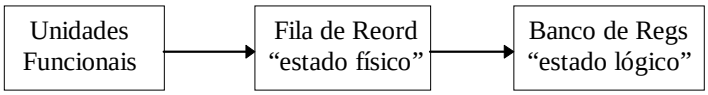


FIGURA 2.13 - Conclusão com Fila de Reordenação

Nota-se que a segunda técnica envolve apenas duas entidades. De fato isto ocorre por que os resultados imediatos são gravados diretamente dentro da fila de reordenação, que deve conter campos suficientes para abrigar os registradores das instruções.

O tratamento preciso de interrupções torna-se difícil quando o tempo de resposta deve ser curto. Para isto, o estado do processador deve ser visível para o sistema operacional e a aplicação deve ser reconstruída para o estado que o processador teria até o momento anterior a interrupção. Isto requer um alto custo de implementação, principalmente para processadores com execução fora-de-ordem.

Além disso, a existência de desvios em cerca de 15% a 30% das instruções executadas para muitas aplicações diminui a efetividade de tais processadores, se as instruções seguintes ao desvio não resolvido não puderem ser executadas. Neste caso, como já visto, o desempenho pode ser aumentado através da execução especulativa sobre um caminho previsto de instruções.

De uma forma geral, o despacho de instruções e a conclusão dos resultados podem ser feitos de 3 formas básicas, proporcionando desempenhos diferentes nas arquiteturas [JOH 91]:

1) despacho em ordem e conclusão em ordem: o despacho respeita a ordem em que as instruções aparecem e os resultados são concluídos na mesma ordem. Neste caso, uma instrução não pode ser despachada antes do despacho de suas precedentes (podem até começar juntas) e o resultado de uma instrução posterior não pode ser concluído antes dos resultados de suas instruções precedentes terem sido concluídos. Este sistema pode causar paradas no despacho e na conclusão.

2) despacho em ordem e conclusão fora de ordem: o despacho respeita a ordem em que as instruções aparecem mas os resultados podem ser concluídos em qualquer ordem. Este sistema tenta eliminar as paradas na conclusão.

3) despacho fora de ordem e conclusão fora de ordem: ambos despacho e conclusão podem ser feitos em qualquer ordem. Este sistema tenta eliminar as paradas em no despacho e na conclusão, obviamente necessitando de uma reordenação nos resultados finais.

2.2.5 Escalonamento Dinâmico de Instruções

Duas tendências têm sido utilizadas para o escalonamento de instruções [GON 97]: algoritmos dinâmicos e algoritmos estáticos. Na primeira, o *hardware* possui um algoritmo capaz de escolher a instrução que deve ser despachada e decidir para onde deve ser despachada. Este algoritmo utiliza recursos de *hardware* e garante a execução correta das instruções, bem como controla a recuperação do estado do processador durante a ocorrência de exceções. Na segunda, cabe ao compilador gerar o código eficiente de forma que as unidades funcionais possam executar um maior número de instruções independentes por ciclo. Esta segunda abordagem é direcionada para arquiteturas VLIW.

Apesar de alguns processadores, tais como aqueles da linha DEC Alpha 21164 e UltraSparc III, não utilizarem escalonamento dinâmico e tentarem obter alto desempenho através do uso de altas freqüências de processamento do *hardware*, a maioria dos processadores atuais realizam algum tipo de escalonamento dinâmico.

Escalonamento dinâmico refere-se a uma classe de técnicas de escalonamento escutada pelo *hardware* em tempo de execução. O objetivo do escalonamento dinâmico é melhorar os resultados sobre o escalonamento estático, através da adição de informações, em tempo de execução, na decisão de escalonamento. A quantidade de *hardware* utilizado pelo escalonamento dinâmico depende do algoritmo utilizado [BUT 92]. São conhecidos pelo menos 3 casos onde o escalonamento dinâmico pode melhorar o desempenho sobre um escalonamento de código estático [CHA 91]:

- **Desvio no *Pipeline* para Operações de Leitura em Memória*:*** permite que uma operação de leitura em memória seja executada antes da finalização da operação de escrita correspondente, desde que seus endereços não sejam coincidentes. O *pipeline* é adaptado para permitir que os resultados da escrita, quando já calculado mas ainda não concluído, seja redirecionado para a instrução de leitura. Pelo fato que o compilador pode não conhecer o endereço das operações de memória, o escalonador estático pode

não ser capaz de tirar vantagens desta situação, enquanto que em tempo de execução, estes endereços podem ser resolvidos e permitir que o escalonador dinâmico detecte as oportunidades perdidas pelo escalonador estático.

- **Sobreposição das Iterações de Laços:** Geralmente, para grande laços (*loops*), o escalonador estático pode não conseguir desenrolá-los (vide *loop unrolling* ou *loop peeling* em [WOL 96]) para evitar expansão desnecessária de código. Com execução fora-de-ordem, as execuções de instruções de diferentes iterações de um laço podem ser sobrepostas.

- **Tolerância aos Atrasos Causados pelas Faltas na *Cache* de Dados:** Para o escalonamento estático, o *pipeline* de instrução é bloqueado na ocorrência de uma falta na *cache* de dados. Já com execução fora-de-ordem, o *hardware* permite que operações independentes antecipem a espera pelas operações de memória.

Existem 2 aspectos na execução de um programa que podem influenciar no escalonamento dinâmico [BUT 92]: A execução especulativa e as latências não-determináveis. A primeira, que surge com a previsão de desvios, permite que instruções anteriores a um desvio possam ser escalonadas juntas com aquelas que estão no caminho previsto. Na segunda, a faltas na *cache* de dados podem causar atraso na busca da memória, prejudicando o escalonamento.

Duas principais técnicas de escalonamento dinâmico de instruções são comumente utilizadas no projeto de arquiteturas superescalares [FER 92]: renomeação de registradores e algoritmo de Tomasulo, que passam a ser explicadas na próxima seção.

2.2.5.1 Renomeação de Registradores

Em função da quantidade limitada de recursos, o objetivo de um compilador é armazenar um maior número de valores em um menor número de registradores, tanto quanto seja possível, a fim de melhor aproveitá-los. Esta característica causa um grande número de conflitos [JOH 91]: anti-dependências e dependências de saída.

Dentro de um programa estático [SMI 95], os elementos de armazenamento (registradores e localizações de memória) possuem referências lógicas, e devem ser mapeados (renomeados) para os registradores físicos existentes. Durante este processo, as falsas dependências são eliminadas.

O processador remove estes conflitos através da alocação dinâmica de registradores adicionais. Geralmente, o processador aloca um novo registrador para cada novo valor produzido, ou seja, para cada instrução que escreve em um registrador destino. O exemplo da figura 2.14 mostra os DAGs (grafos de dependências) para os respectivos trechos de código antes e após a renomeação.

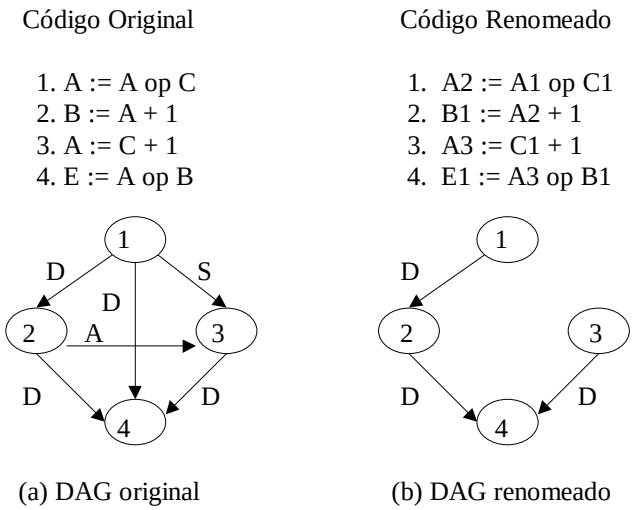


FIGURA 2.14 - DAGs Antes e Após Renomeação

Nos DAGs da figura 2.14, pode-se notar que sem renomeação, todas as instruções somente podem ser executadas seqüencialmente, em função das diversas dependências. Com renomeação, as instruções 1 e 3 podem ser executadas em paralelo. Assim, com o uso desta técnica, um maior paralelismo pode ser extraído dos programas. Existem pelo menos 2 métodos de renomeação de registradores, comumente usados [SMI 95].

No primeiro, existe um arquivo de registradores físicos maior do que o arquivo de registradores lógicos. Uma tabela de mapeamento é usada para associar um registrador físico ao registrador lógico corrente. As instruções são decodificadas e a renomeação é executada na ordem seqüencial do programa. A arquitetura MIPS utiliza este tipo de renomeação, sendo 64 o número de registradores físicos e 32 o número de registradores lógicos [MIP 95].

A figura 2.15 exemplifica este tipo de renomeação de registradores físicos. A instrução considerada é uma "*add r3, r3, 4*" (soma o conteúdo de R3 com o valor 4 e coloca o resultado novamente em R3). Os registradores lógicos (compilados) são denotados por letras minúsculas, e os registradores físicos por letras maiúsculas. No lado esquerdo desta figura (antes), a tabela de mapeamento mostra o registrador r3 mapeado para o registrador físico R1, e o primeiro registrador disponível na lista de registradores livres é o R2.

Durante a renomeação, o registrador fonte r3 da respectiva instrução é trocado por R1, que é o registrador físico já mapeado por uma instrução precedente. Já o registrador destino, que seria novamente o r3, é renomeado para o próximo disponível da lista, o R2, que é colocado na tabela de mapeamento. Após este mapeamento, uma instrução subseqüente que precisar ler o resultado desta instrução *add* tem seu registrador fonte mapeado para R2.

Após os registradores físicos terem sido lidos pela última vez, eles podem ser colocados de volta na lista de registradores livres. Dependendo da implementação, isto pode requisitar *hardware* complicado. Uma possibilidade é associar um contador com cada registrador físico. O contador é incrementado cada vez que o registrador fonte é

renomeado para o registrador físico, durante o despacho, e sofre um decremento cada vez que seu conteúdo é lido por uma instrução é executada. Quando o contador chega em zero (0) o registrador pode ser liberado.

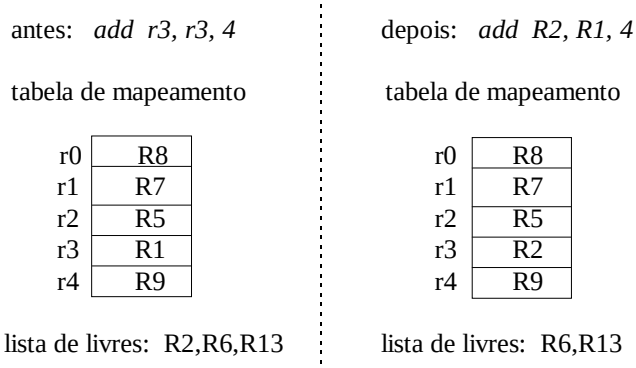


FIGURA 2.15 - Exemplo de Renomeação Usando Registradores Físicos

Um método mais simples do que o método do contador consiste em esperar até que o registrador lógico já renomeado anteriormente seja novamente renomeado e que as instruções que o utilizam como fonte já tenham sido concluídas.

No segundo método de renomeação, é usado um arquivo de registradores físicos de mesmo tamanho que o número de registradores lógicos, e o relacionamento um-a-um é mantido. Adicionalmente, aproveita-se a fila de reordenação, com uma entrada para cada instrução ativa e campos para acomodar os operandos destinos. Os registradores lógicos são mapeados provisoriamente na fila de reordenação enquanto a instrução ainda é especulativa. Quando a instrução é retirada da fila de reordenação, o registrador lógico é mapeado diretamente no arquivo físico de registradores. Uma tabela de mapeamento é utilizada para informar o local correto em que se encontra o registrador lógico.

Quando uma instrução é decodificada, seu registrador destino (que armazena o resultado) é primeiramente associado a uma entrada na fila de reordenação e a tabela de mapeamento é marcada corretamente. Quando a instrução é executada, o resultado é colocado nesta mesma entrada. Os identificadores dos registradores fontes lógicos das instruções seguintes são usados para acessar a tabela de mapeamento.

A figura 2.16 exemplifica o processo de renomeação aplicado a instrução "*add r3, r3, 4*", que é similar ao método anterior. No referido exemplo, no momento em que a instrução está pronta para ser despachada (parte esquerda da figura) os registradores *r0, r1 e r2* residem no próprio arquivo de registradores (mapeamento direto), e o registrador *r3* já está mapeado na entrada *reord6* da fila de reordenação (mapeamento provisório), por alguma instrução anterior e ficará ali até que esta mesma entrada atinja a cabeça da lista e seja concluída. Neste caso, o resultado é gravado no arquivo de registradores e a tabela é atualizada para mapear diretamente.

A tabela também indica se o registrador físico definitivo já contém o valor requisitado ou se ele pode ser encontrado na fila de reordenação. Quando a fila de reordenação está cheia, o despacho é interrompido. Como parte do processo de renomeação, o registrador fonte r3 é trocado com a entrada *reord6* e uma nova entrada *reord8* é alocada na cauda

da fila de reordenação. Este número é então registrado na tabela de mapeamento para as instruções que usam o r3 como registrador fonte.

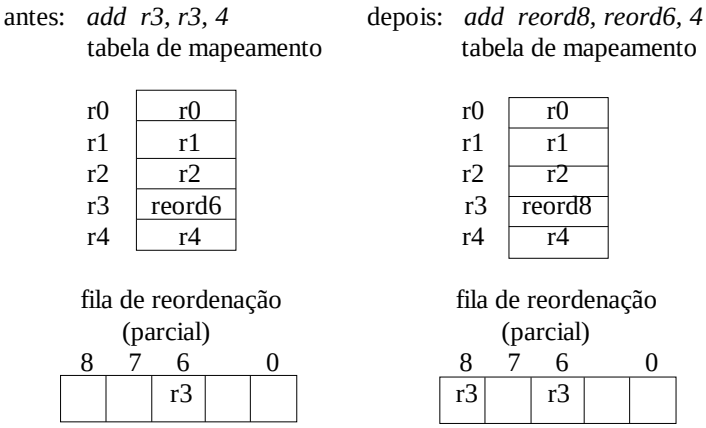


FIGURA 2.16 - Exemplo de Renomeação Usando Fila de Reordenação

Pode-se observar em ambos os métodos a necessidade da fila de reordenação para conter as informações da seqüencialidade, controle e interrupção. A diferença é que no primeiro método ela não precisa conter os dados dos operandos das instruções especulativas, pois estes dados podem ser escritos diretamente no arquivo registradores físicos, que em contrapartida deve ser maior. No segundo método, o gerenciamento da tabela de renomeação pode ser mais complicado, pois quando as entradas na fila de reordenação sofrem deslocamento, a tabela deve ser atualizada.

2.2.5.2 Algoritmo de Tomasulo

Este algoritmo [GON 98c] explora a existência de múltiplas unidades funcionais, permitindo a execução simultânea de instruções independentes e preservando a precedência inerente do fluxo de instruções. O algoritmo é pequeno e logicamente simples para ser implementado em *hardware*, permitindo concorrência automática com um mínimo esforço do compilador. O algoritmo foi inicialmente proposto para o IBM System/360 Model 91 por Tomasulo [TOM 67].

Em uma arquitetura superescalar com algoritmo de Tomasulo, cada instrução, que está na fila de instruções, é decodificada e despachada para uma estação de reserva específica, em ordem. As estações de reserva são organizadas em filas de remessa (vide seção 2.2.3.1) dedicadas a unidades funcionais específicas, de acordo com os tipos das instruções. O modelo básico do *hardware* pode ser visto na figura 2.17.

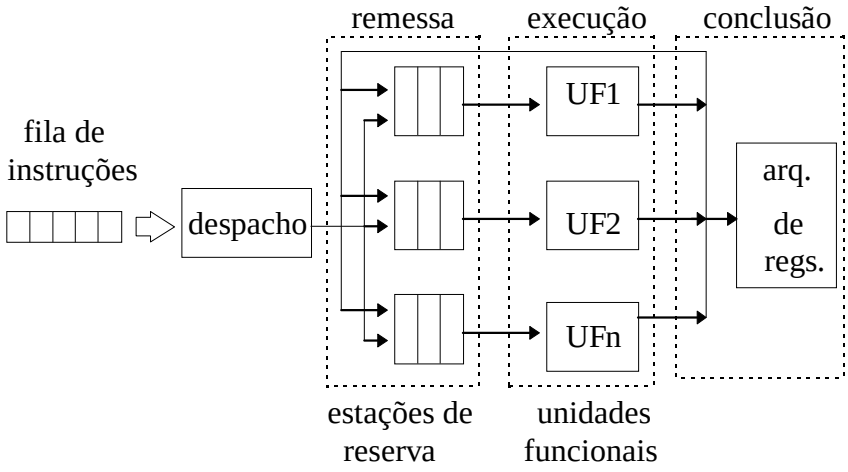


FIGURA 2.17 - Modelo Básico do *Hardware* de Tomasulo

Cada estação de reserva pode ser identificada pelo *hardware* através de um endereço. As estações de reserva servem para liberar o decodificador, mesmo quando as unidades funcionais estão ocupadas, ou mesmo quando as instruções não estão totalmente prontas para serem executadas (faltando operandos).

Cada registrador tem uma tarja (*tag*) associada, para identificar qual a instrução que deve atualizar o seu conteúdo, bem como um *bit* de ocupação (*busy-bit*). Cada estação de reserva contém diferentes campos para armazenar a instrução decodificada, incluindo: a) campos para os registradores fonte e destino (operandos), b) campo para os controles decodificados e c) campos para as tarjas dos registradores (um para cada operando). Dentro do campo de controles também existe um *bit* de ocupação da estação de reserva. O *hardware* possui um barramento comum de retorno de dados que permite que os resultados das unidades funcionais estejam disponíveis simultaneamente a todas as estações de reserva e ao conjunto de registradores.

Existem 3 tipos de dependências entre as instruções [JOA 95]: dependência direta, anti-dependência e dependência de saída. Para controlar estas dependências, o algoritmo de Tomasulo não permite que um registrador seja utilizado até que seu conteúdo reflita o resultado da operação mais recente. Assim, durante o despacho, cada instrução tem os *bits* de ocupação dos seus registradores fonte e destino verificados. As dependências são tratadas da seguinte forma:

- **Sem Dependência:** Se os *bits* de ocupação de ambos registradores fonte e destino estão livres, a instrução não é dependente de nenhuma outra instrução anterior. Neste caso, os conteúdos dos registradores são enviados para a estação de reserva apropriada. Depois, o *bit* de ocupação do registrador destino é marcado como ocupado e sua tarja é marcada com o endereço da estação de reserva utilizada. O *bit* de ocupação da estação de reserva utilizada também é marcado como ocupado.

- **Dependência Direta:** Para a instrução que tem o seu registrador fonte ocupado, tem-se uma dependência direta e o conteúdo deste registrador ainda não está atualizado. Neste caso, o despacho é feito mesmo sem este dado, mas sua tarja é despachada (contendo o endereço da estação de reserva da instrução que deve fornecer o operando faltante) para a tarja junto ao campo do operando fonte na estação de reserva

apropriada. Ambos o *bit* de ocupação e a tarja do registrador fonte são mantidos inalterados.

- **Dependência de Saída:** Para a instrução que tem o seu registrador destino ocupado, tem-se uma dependência de saída, mas o conteúdo atualizado do registrador destino não interessa. Assim, o *bit* de ocupação do registrador destino é mantido ocupado mas sua tarja é atualizada para o endereço da nova estação de reserva, pois a tarja de um registrador ocupado deve conter sempre o endereço da estação de reserva associada com a última instrução que deve atualizar o conteúdo deste registrador.

Como os operandos faltantes das instruções despachadas anteriormente são fornecidos pelo barramento, esta instrução não precisa aguardar o término das instruções precedentes, que utiliza o mesmo registrador destino. Esta é uma das grandes vantagens do algoritmo de Tomasulo, que permite a execução paralela das instruções com dependência de saída, pois somente a última atualiza o registrador.

- **Anti-Dependência:** Como as instruções são despachadas conforme a ordem original do programa, as anti-dependências são respeitadas naturalmente pelo algoritmo de Tomasulo, pois uma instrução que apenas lê o conteúdo de um registrador, é despachada antes que o conteúdo do registrador seja alterado por uma outra instrução posterior. E mesmo que o dado ainda não esteja pronto, o barramento se responsabiliza pelo envio posterior do dado faltante.

Junto às estações de reserva existe uma lógica de remessa de instruções, que detecta quais instruções estão prontas, verificando as tarjas, e seleciona uma ou mais, de acordo com algum algoritmo, e as remete, fora de ordem, para as unidades funcionais apropriadas e disponíveis. Quando uma instrução é concluída, a unidade funcional envia uma mensagem para todas as estações de reserva e para o conjunto de registradores. Sempre que uma instrução é executada, uma estação de reserva é liberada e um registrador destino tem seu *bit* de ocupação marcado como livre.

O envio múltiplo (*broadcast*) contém o resultado da operação e o endereço da estação de reserva que continha a instrução executada. Dentro do conjunto de registradores, aquele que tiver uma tarja igual ao que está no barramento, tem o seu conteúdo atualizado. Neste caso, ambos os campos *bit* de ocupação e a tarja são apagados e o registrador fica livre. O mesmo acontece com os campos dos operandos das estações de reserva que contém uma tarja igual ao do barramento. Neste caso, muitas das instruções que estavam aguardando operandos nas estações de reserva se tornam prontas e podem ser remetidas.

Quando a fila de remessa da unidade funcional para o qual a instrução deve ser despachada está cheia, o despacho das instruções é interrompido, já que o despacho é feito em ordem, e isto conseqüentemente interrompe a decodificação. Este problema reduz o desempenho da arquitetura superescalar e é conhecido como saturação da fila de remessa [GON 98c]. O problema da saturação da fila de remessa é mais degradante quando existem estações de reserva disponíveis nas outras filas sem poderem ser utilizadas.

Uma possível solução de *software* para tratar este problema foi apresentada em [GON 97b], onde um algoritmo executado pelo compilador intercala os diferentes tipos de

instruções, reduzindo o tamanho das seqüências de instruções adjacentes de mesmo tipo e reduzindo a saturação de uma mesma fila de remessa. Uma outra alternativa de *hardware*, já discutida, é a utilização de fila de remessa unificada (vide seção 2.2.3.1), que foi avaliada em [GON 98b], mostrando que só existe aumento substancial no desempenho se os fluxos de instruções são balanceados entre as diferentes unidades funcionais.

3 Arquiteturas Multi-tarefas Simultâneas

Conforme dito anteriormente, o desempenho dos processadores atuais tem sido limitado pelas dependências de dados, que é inerente às aplicações convencionais (mono-tarefa). Para sobrepor a este obstáculo, muitas arquiteturas estão sendo projetadas para explorar o paralelismo de aplicações de múltiplas tarefas. Estas arquiteturas são chamadas de Multi-Tarefas e têm sido implementadas sob diferentes modelos.

A idéia de execução muti-tarefa surgiu dos sistemas operacionais multi-tarefas, onde o paralelismo é virtual e é obtido pela intercalação de tarefas no tempo. Posteriormente, quando os processadores começaram a permitir auto-acoplamento e alta-compactação em circuitos integrados, as tarefas começaram a ser executadas sobre arquiteturas de multi-processadores, tais como o sistema MTA [SHA 98], que é baseado no processador Tera [ALV 90]. Uma análise deste tipo de arquitetura pode ser encontrada em [THE 94]. Paralelamente, uma outra tendência se iniciou com o advento das redes locais e as tarefas puderam ser executadas sobre processadores remotos também. Neste caso, o controle do paralelismo é feito por algum tipo de plataforma, normalmente baseada em *sockets* [TUR 93].

Na tentativa de reduzir o volume do *hardware* e aumentar a sua utilização, o compartilhamento de recursos tem sido um dos principais interesses no projeto de computadores. Neste sentido, o mais ambicioso desejo talvez seja o de desenvolver um processador simples que seja capaz de executar múltiplas tarefas eficientemente. Enquanto isto não é possível, um tipo de arquitetura intermediária entre multi-processador e processador simples, chamada de Multi-Tarefa, tornou-se uma das mais interessantes abordagens. Seu *hardware* tem muitos recursos duplicados, necessários para manter o contexto das diferentes tarefas, mas também tem muitos recursos compartilhados, tais como *caches* e unidades funcionais.

Muitas arquiteturas Multi-Tarefas podem despachar para o *pipeline* somente uma instrução por ciclo, de uma diferente tarefa cada vez. Embora elas tenham bastante complexidade e *hardware*, elas são processadores escalares ainda. Todos os contextos são mantidos carregados na arquitetura e somente um entre aqueles não-bloqueados pode ser utilizado para despachar uma instrução em determinado momento. Exemplos deste tipo de arquitetura podem ser encontrados em [ALV 90, NEM 91]. Devido ao baixo desempenho obtido individualmente por cada tarefa, uma outra abordagem tem sido usada para permitir troca de contexto somente após a execução de um bloco de instruções a cada ciclo, de uma diferente tarefa a cada vez, limitado quase sempre por uma instrução de longa latência. Neste caso, as trocas de contexto não ocorrem após intervalos maiores de tempos do que aquelas do método anterior. Seguindo esta abordagem, novas técnicas de entrelaçamento de instruções foram propostas [LAU 94], mas o *pipeline* permanecia ainda escalar.

Com o advento da tecnologia de processadores superescalares, as arquiteturas Multi-Tarefas começaram a explorar execução fora-de-ordem para cada tarefa, contudo, as tarefas somente poderiam ser escalonadas uma por vez, em um estilo do tipo *round-robin*. Muita pesquisa tem investigado este tipo de arquitetura [GUL 96, LOI 96, SIG 96], e uma comparação entre esta abordagem e de multi-processadores é apresentada em [UNG 96].

Recentemente, uma técnica mais agressiva para projetar arquiteturas Multi-Tarefas visa a execução de instruções de diferentes tarefas simultaneamente, durante um mesmo ciclo do processador, usando um conjunto de unidades funcionais compartilhadas. Em 1992, Hirata [HIR 92] apresentou uma arquitetura chamada de *Simultaneous Instruction Issuing*. Dois anos depois, Yamamoto [YAM 94] propôs uma arquitetura chamada *Multistreamed Superscalar* que possuía a capacidade de executar instruções provenientes de diferentes fluxos simultaneamente. Em 1995, Tullsen [TUL 95] propôs uma arquitetura similar que foi chamada de SMT (*Simultaneous Multithreaded*). Atualmente, o acrônimo SMT é usado para indicar este tipo de arquitetura.

Uma arquitetura SMT é capaz de atingir alto *ipc* (instruções por ciclo) devido a três principais razões. Primeiro, porque o paralelismo existente em várias tarefas é intenso e pode ser muito maior caso não haja comunicação entre elas. Segundo, porque o escalonamento de instruções não-bloqueadas para o despacho pode esconder as altas latências de outras instruções bloqueadas, tais como aquelas de acesso à memória. Finalmente, porque através da mistura das instruções de diferentes tarefas, a execução SMT tira melhor proveito dos recursos que poderiam estar ociosos, tais como unidades funcionais, devido à maior variedade de tipos de instruções.

De uma forma geral, uma arquitetura multi-tarefas simultâneas básica pode ser vista na figura 3.1, onde se nota que a principal diferença entre esta arquitetura e uma arquitetura de multi-processador está no compartilhamento das unidades funcionais (UF1...UFm) e memórias (*Caches* de dados e instruções). Já as estruturas de armazenamento da arquitetura (filas de instruções e arquivo de registradores) são separadas fisicamente, ou pelo menos logicamente, e devem ser grandes de forma suficiente para abrigar contextos distintos dentro da arquitetura.

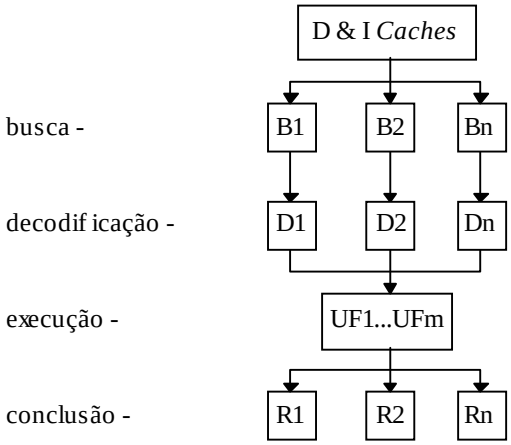


FIGURA 3.1 - Arquitetura Multi-Tarefas Simultâneas Básica

Deve-se observar que as tarefas que são executadas pela arquitetura SMT permanecem ativas enquanto não for efetuada uma troca no nível do sistema operacional. Assim, no nível de *hardware*, não é *muito* correto dizer que ocorre “troca de contexto” mas sim “chaveamento de contexto”, mas, devido a esta terminologia ser usual, ela é mantida aqui neste trabalho. Apesar de serem promissoras, o desenvolvimento de arquiteturas multi-tarefas simultâneas é desmotivado pela falta de aplicações com múltiplas tarefas, pois a maioria esmagadora das aplicações existentes é seqüencial. Sabe-se também que

essas arquiteturas somente alcançam paralelismo elevado na presença de um grande numero de tarefas, e que a comunicação entre elas é um fator degradante. Com isso, surge também o aumento das faltas na *cache* decorrente do aumento do número de tarefas. Na tentativa de melhor compreender as arquiteturas SMT e resolver os problemas citados, alguns trabalhos de avaliação e desempenho foram realizados e são descritos na seção seguinte.

3.1 Avaliação e Desempenho de Arquiteturas SMT

Embora exista muita pesquisa sobre o conceito multi-tarefas, o conceito multi-tarefas simultâneas é moderno e recente. Poucas arquiteturas têm sido propostas e poucos trabalhos de avaliação e desempenho foram feitos sobre elas. A seguir, são apresentados alguns destes trabalhos.

Tullsen [TUL 96] analisou diferentes políticas de busca e de remessa de instruções em arquiteturas SMT. Em seu trabalho, foi concluído que a busca de instruções deve favorecer aquelas tarefas que possuem menos instruções dentro do *pipeline*, embora esta questão não seja tão importante a ponto de causar grandes alterações no desempenho.

Em outro trabalho feito por Hily [HIL 98] foram analisadas questões sobre a hierarquia de memória, deixando claro que a contenção na *cache* nível 2 não pode ser ignorada, caso contrário, os resultados das simulações poderiam ser super estimados. Também, Hily mostrou que para obter melhores desempenhos a *cache* precisa de alta associatividade, o tamanho dos blocos deve ser pequeno e o tamanho das *caches* deve ser tão grande quanto o número de tarefas. Em função desta última restrição, o tamanho das arquiteturas SMT deve ser restrito.

Também, Lo [LO 98] trabalhou com um problema muito sério relacionado com os conflitos na *cache* causados pela coincidência dos endereços de memória das diferentes tarefas, gerados pelo compilador. Em seu trabalho foi concluído que muitos conflitos podem ser eliminados por *software* usando uma política adequada de mapeamento de endereçamento virtual-físico e através do uso de deslocamentos de endereçamento diferente por tarefa. Além disso, Lo trabalhou sobre compiladores [LO 97a] e sobre o desenvolvimento de técnicas de software [LO 97b] para reduzir a quantidade de registradores necessários para executar múltiplas tarefas.

Em outro trabalho [SIG 99] vários modelos de *caches* e estratégias de substituição de blocos foram avaliados por Sigmund em arquiteturas SMT. Seu maior interesse é usar este tipo de arquitetura em aplicações de vídeo multimídia, tais como descompressão de imagens. Em seus estudos, ele concluiu que a escolha da política de substituição da *cache* é fundamental quando existem restrições no barramento de memória. Outros trabalhos também podem ser vistos em [GUL 96, ESP 97, KRI 97, GON 99, MAD 99, TOR 99].

Quase todas estas pesquisas têm mostrado que com execução multi-tarefas as *caches* sofrem maior degradação, devido aos conflitos de endereçamento. Nas simulações realizadas por Tullsen [TUL 95], o compartilhamento da *cache* foi o fator dominante na ocorrência de ciclos perdidos, pois variando de 1 a 8 tarefas, a freqüência de faltas na *cache* de instruções nível 1 cresceu de 1% a 14%. Em [TUL 96], Tullsen também concluiu que um desempenho considerável pode ser obtido com uma boa política de

pré-busca e que dentre os principais parâmetros arquiteturais, a largura de busca pode ser o principal gargalo de uma arquitetura multi-tarefas.

Segundo Lo [LO 98], existem dois tipos de interferências na cache: interferência destrutiva, que ocorre quando as instruções/dados de uma tarefa são trocadas pelas instruções/dados de outra tarefa, e interferência construtiva, quando as instruções/dados acessadas por uma tarefa são também acessadas simultaneamente por uma outra tarefa, resultando em menos faltas na *cache*. Utilizando um processador multi-tarefas, Gulati [GUL 96] também determinou em seus experimentos que o aumento do número de tarefas de 1 até 6 causa uma variação da taxa média de acerto na *cache* de 97.33% para 76.21%, o que representa um aumento global de 21.12% na taxa de faltas na *cache*. Nemirovsky [NEM 98] também mostrou que em uma arquitetura multi-fluxo, na medida que o numero de fluxo aumenta, ocorre maior interferência no acesso a *cache* podendo chegar a 200% no pior caso para 4 fluxos.

A questão toda é que a alta demanda de instruções das arquiteturas SMT aumenta a eficiência dos recursos do processador [BUJ 99], mas a unidade de busca torna-se um gargalo, e mesmo aumentando o numero de níveis no sistema de memória, Rinker [RIN 98] afirma que as faltas na *cache* somente são reduzidas em no máximo 1%, de forma que ainda são muito degradantes, produzindo tempos de ociosidade do processador que podem exceder os 50%. Visando uma melhor exemplificação da abordagem SMT, a seguir são resumidas as arquiteturas precursoras citadas no início deste capítulo.

3.2 Arquitetura de Hirata

Hirata [HIR 92] propôs uma arquitetura de processador multi-tarefas onde as instruções de diferentes tarefas são despachadas simultaneamente para múltiplas unidades funcionais. Pode-se dizer que na década de 90, esta arquitetura foi a primeira a propor um *pipeline* SMT, apesar de que a origem desta denominação não se deve a Hirata e sim a Tullsen [TUL 95]. A arquitetura de Hirata possui vários *thread slots* que contém basicamente uma fila de instrução e uma unidade de decodificação. Cada *thread slot* é associado a um contador de programa. A unidade de busca de instruções, bem como as unidades funcionais, são fisicamente compartilhadas entre os processadores lógicos. A figura 3.2 mostra uma visão geral do *pipeline* SMT.

As instruções de desvios são executadas dentro da unidade de decodificação. As dependências de dados de uma instrução são checadas usando a técnica *scoreboarding*, permitindo que as instruções sejam despachadas, para as estações *standby*, livres de dependências (operandos prontos). As instruções são dinamicamente escalonadas, pelas unidades de escalonamento de instruções, e remetidas para as unidades funcionais. A diferença com Tomasulo é que as instruções nas estações *standby* estão completas, não necessitando assim do barramento comum de dados (para retorno dos resultados).

Existe um arquivo grande, de registradores, dividido em bancos privados por tarefa, que servem para armazenar os contextos de cada uma. Existem também registradores especiais (*queues*) para habilitar comunicação entre *slots*. O estágio de escalonamento de instruções lê os operandos dos registradores fontes e sinaliza os *bits* de *scoreboarding* para os registradores destinos, os quais somente são desligados no último

estágio. Nesta arquitetura, enquanto algumas tarefas são bloqueadas por causa dos desvios, outras instruções de outras tarefas são executadas.

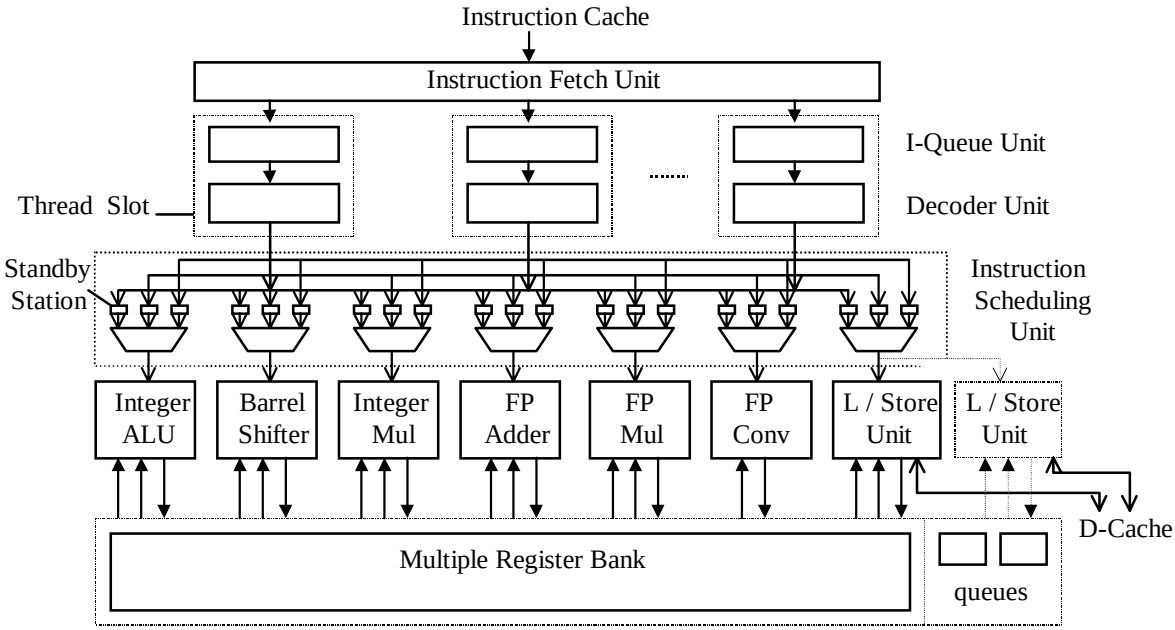


FIGURA 3.2 - Arquitetura Básica de Hirata

Esta arquitetura utiliza política de prioridade de múltiplos-níveis, para escalonar instruções dinamicamente das estações *standby*. Cada *thread slot* recebe uma prioridade diferente. Assim, as instruções oriundas dos *slots* de maior prioridade são escalonadas primeiramente. Para evitar *starvation*, as prioridades são roladas entre os deferentes *thread slots*. Com isso, a prioridade mais baixa é rolada para o *thread slot* que contém a prioridade mais alta e vice-versa. Existem 2 modos de rotação básicos: modo de rotação implícita e modo de rotação explícita. No modo de rotação implícita, a rotação ocorre sempre após um dado número de ciclos (rotação intervalar). No modo de rotação explícito, a rotação ocorre através da execução de uma instrução especial inserida pelo compilador.

A arquitetura foi simulada e os desempenhos foram calculados com relação a um processamento seqüencial sobre uma arquitetura RISC básica com *pipeline* de 7 estágios. Foi utilizado somente o programa *ray-tracing* com otimização máxima, tornando os experimentos de Hirata pouco significativos. O programa foi executado em uma estação de trabalho e os *traces* (imagem do fluxo de execução) foram transladados para o simulador. A tabela 3.1 mostra os resultados das simulações.

TABELA 3.1 - Desempenhos da Arquitetura de Hirata

Nro de *thread slots*	com 1 unidade de *load/store*	com 2 unidades de *load/store*
2	1.83	2.02
4	2.89	3.72
8	3.22	5.79

3.3 Arquitetura de Yamamoto

Em 1994 Yamamoto [YAM 94] apresentou uma proposta de arquitetura SMT chamada *Multistreamed Superscalar Processor*, que combina a técnica de executar múltiplas instruções por ciclo, característica dos processadores superescalares, com a habilidade de entrelaçar dinamicamente a execução de múltiplos fluxos de instruções, característica dos processadores multi-tarefas (*multistreamed*). Esta arquitetura despacha instruções de múltiplos fluxos simultaneamente, dentro do mesmo ciclo. Isto é feito armazenando cada contexto internamente e permitindo que um escalonador selecione instruções de todos os fluxos ativos, visando sempre o despacho do maior número possível. Se os fluxos são independentes, a quantidade de instruções que são despachadas aumenta quando o número de fluxos aumenta.

Devido ao custo das simulações, tanto em complexidade quanto em tempo, um modelo analítico simples foi inicialmente utilizado para estimar o desempenho global da arquitetura. Posteriormente, uma simulação dirigida por execução de programas SPEC89 mostrou os benefícios no desempenho da arquitetura através da emulação de uma versão do IBM RS/6000. Os resultados da modelagem e da simulação foram comparados e a diferença no desempenho não superou 4% O diagrama de blocos da arquitetura é mostrado na figura 3.3.

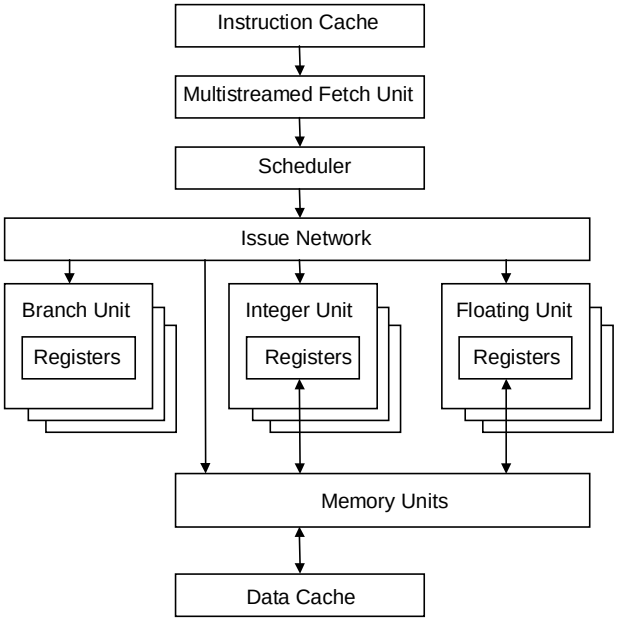


FIGURA 3.3 - Diagrama de Blocos da Arquitetura de Yamamoto

O modelo arquitetural é dividido em 3 maiores componentes: os fluxos de instruções, o escalonador e as unidades funcionais. Um fluxo de instruções consiste de um contexto e de uma fila contendo as próximas instruções a serem executadas. A seção desta fila considerada pelo escalonador para ser despachada é chamada de “janela de remessa do fluxo”. As instruções dentro desta janela são despachadas fora de ordem quando não existem dependências (de dados, de controle e de recursos) entre elas. O escalonador percorre, em estilo *round-robin*, as instruções de cada janela e rotula as instruções

independentes. Uma janela global contém todas as instruções prontas de todos os fluxos que são remetidas para as unidades funcionais sem restrições.

Todas as unidades funcionais são totalmente "pipelinizadas", isto é, permitem receber uma nova instrução a cada ciclo. Entretanto, as instruções de diferentes tipos têm diferentes tempos de execução, como por exemplo, uma divisão inteira leva 17 ciclos para ser concluída. As unidades funcionais são compartilhadas por todos os fluxos. O processador é capaz de executar *n* instruções simultaneamente. Cada tarefa de uma aplicação é associada a um fluxo dentro do processador, com contador de programa e conjunto de registradores privados. Os resultados médios sobre 3 diferentes configurações arquiteturais, relativos ao desempenho da modelagem (MOD) e da simulação (SIM), são mostrados na figura 3.4, onde os programas de avaliação foram executados em torno de 100 milhões de instruções.

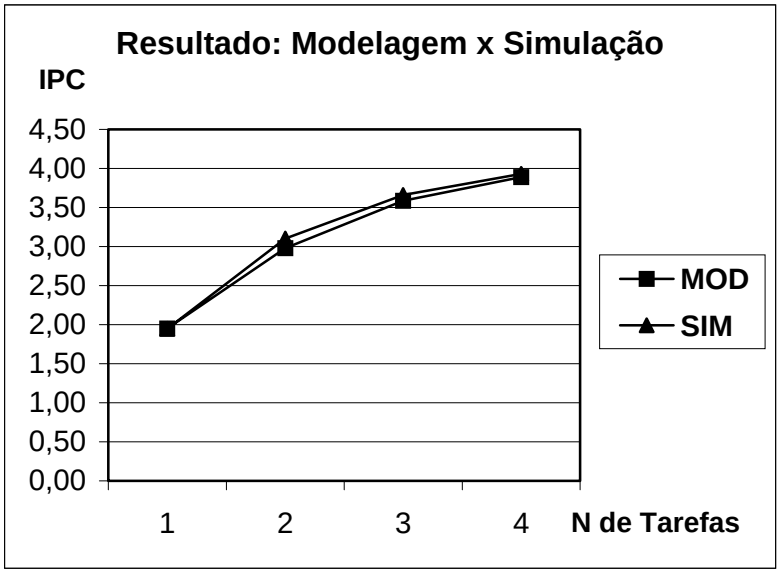


FIGURA 3.4 - Resultados dos Experimentos de Yamamoto

Duas grandes conclusões foram feitas por Yamamoto. A primeira delas é que as arquiteturas superescalares não são capazes de explorar as unidades funcionais em grande quantidade, devido ao limite inerente do paralelismo no nível de instrução contido nas aplicações mono-tarefa. Entretanto, as aplicações multi-tarefas ultrapassam este limite. A abordagem superescalar multi-tarefas é um método efetivo para explorar este paralelismo, tirando melhor proveito das unidades funcionais. A segunda delas é que a modelagem analítica provê uma rápida maneira para examinar as muitas variações de uma arquitetura em um alto nível de abstração. Esta técnica requer somente algumas simples descrições da carga de trabalho (*workload*) e da arquitetura, como parâmetros de entrada. Estes parâmetros são facilmente medidos usando ferramentas comumente disponíveis.

3.4 Arquitetura de Tullsen

Tullsen [TUL 95] propôs uma arquitetura que ficou conhecida pelo nome de SMT (*Simultaneous Multithreaded*), mas que basicamente apresenta características similares

àquelas outras propostas por Hirata e Yamamoto. A arquitetura SMT, mostrada na figura 3.5, é derivada de um processador superescalar com execução fora-de-ordem, que representava na época, segundo Tullsen, uma projeção do projeto de um processador para os próximos 5 anos no futuro.

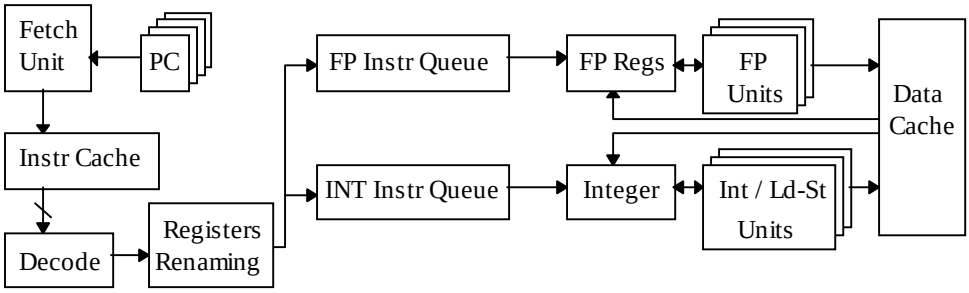


FIGURA 3.5 - Arquitetura Básica de Tullsen

O processador contém: busca de 8 instruções por ciclo, previsão de desvios com BTB, decodificação e renomeação. Similarmente ao MIPS R10000 e HP PA-8000, existem duas filas de instruções para conter as instruções já decodificadas e renomeadas. Destas filas, as instruções são remetidas fora-de-ordem quando seus operandos estão prontos. Os principais componentes e características desta arquitetura são:

- vários contadores de programa e um mecanismo pelo qual a unidade de busca seleciona um deles em cada ciclo.
- uma pilha de retorno separado para cada tarefa para as previsões dos retornos de sub-rotinas
- mecanismos, por tarefa, para conclusão das instruções, esvaziamento do *pipeline* e para o gerenciamento de exceções.
- um identificador de tarefa em cada entrada na BTB para evitar previsão de desvios “fantasmas”, e
- um grande arquivo de registradores, contendo bancos de registradores locais para as tarefas e registradores adicionais para a renomeação.

A busca é feita em um estilo *round-robin* entre todos os contadores de programas das tarefas existentes, com exceção daqueles que sofreram faltas na *cache* de instruções. Para suportar *n* tarefas, foi necessário um mínimo de *n**32 registradores inteiros físicos, mais aqueles necessários para a renomeação. O número de registradores disponíveis para renomeação determina o número de instruções que podem estar no processador entre os estágios de renomeação e a conclusão.

O processador contém 3 unidades de ponto flutuante e 6 unidades de inteiros, onde 4 destas executam também operações de memória. Todas as unidades funcionais são completamente “pipelinizadas”. As filas de remessa possuem cada uma, 32 entradas. A *cache* é estruturada em bancos com multi-portas. A cada ciclo, uma tarefa é buscada embora o estágio de busca pode acessar mais de uma tarefa, desde que não conflitam sobre o mesmo banco da *cache*. Usando programas de avaliação do SPEC92, Tullsen

comparou [TUL 95] os desempenhos de diferentes modelos de arquiteturas multi-tarefas conforme definidos a seguir:

- *Multitarefa de Grão Fino*: somente uma tarefa despacha instruções a cada ciclo, podendo utilizar toda a largura de despacho do processador.

- SMT Total: representa um processador superescalar SMT completamente flexível. Nele, todas as 8 tarefas disputam cada um dos 8 *slots* a cada ciclo.

- SMT 1x; SMT 2x, e SMT 4x**:** Estes 3 modelos limitam o número de instruções que cada tarefa pode despachar (número de instruções ativas na janela de instruções) a cada ciclo. Por exemplo, no processador SMT 2x, cada tarefa pode despachar um máximo de 2 instruções por ciclo. Esta característica obriga que no mínimo 4 tarefas sejam necessárias para preencher os 8 *slots* a cada ciclo.

- SMT Limitada: Cada contexto de *hardware* está conectado diretamente à exatamente uma de cada tipo de unidade funcional. Por exemplo, se o *hardware* suporta 8 tarefas e existem 4 unidades funcionais, cada unidade funcional poderá receber instruções de exatamente 2 tarefas. A divisão das unidades funcionais entre as tarefas é menos dinâmico do que nos outros modelos, mas cada unidade funcional é ainda compartilhada. Uma conexão de 1 para 1 faz com que o modelo se aproxime de um multi-processador *single-chip*.

Como resultados destes experimentos, observou-se que a arquitetura Multitarefa de Grão Fino provê um ganho máximo de 2.1 (com 3,2 ipc) sobre uma execução mono-tarefa (com 1,5 ipc). Para este modelo, existe muito pouca vantagem quando o número de tarefas é maior que 4. Este resultado é similar àqueles de estudos anteriores, que mostraram que multi-tarefas de Grão Fino e de Grão Grosso trazem benefícios somente para 2 até 5 tarefas. Também se observou que uma SMT obtém um ganho máximo sobre um mono-tarefa de 3.2 até 4.2, com um IPC máximo de 6.3. O gráfico da figura 3.6 mostra todos os resultados deste experimento.

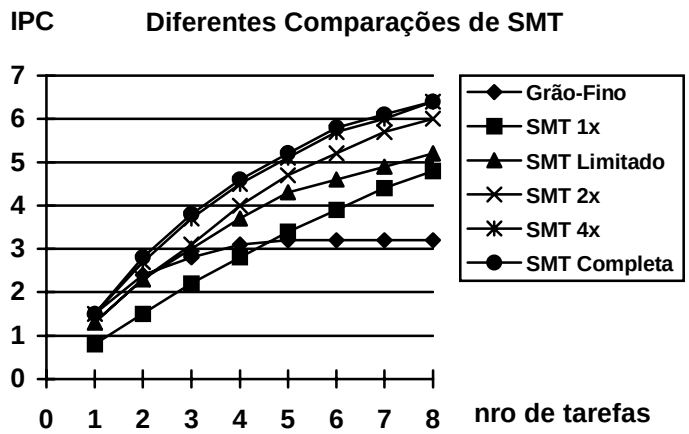


FIGURA 3.6 - Desempenhos da Arquitetura de Tullsen

Em outro trabalho [TUL 96], Tullsen mostrou que as arquiteturas SMT podem obter ganhos no desempenho sem alterações excessivas sobre um processador superescalar convencional com "*pipeline* largo". As simulações mostraram um desempenho de 5.4 instruções por ciclo: um ganho de 2.5 sobre um processador superescalar com recursos de *hardware* similares. Neste mesmo trabalho, também foram avaliadas várias heurísticas simples para a seleção de tarefas que, quando aplicadas ao mecanismo de busca, podem aumentar o desempenho mais do que 37%.

O simulador desenvolvido por Tullsen executa código objeto Alpha e modela o *pipeline* de execução, hierarquia de memória, TLBs e lógica de previsão de desvios. A carga de trabalho simulada é composta por programas SPEC 92 com os programas representando as tarefas, que exigem mais da arquitetura do que um programa paralelo. Cada programa executou 300 milhões de instruções. Os programas foram compilados com o *Multiflow Trace Scheduling Compiler*, modificado para produzir código Alpha. Ele foi escolhido pela alta qualidade da otimização de código (*loop unrolling*, *instruction scheduling* e *alignment*).

O maior desempenho obtido (com 8 tarefas) é 84% maior que o de um processador superescalar (± 3,8ipc x 2,1ipc). Com relação a uma aplicação mono-tarefa, a SMT básica obteve um desempenho apenas 2% menor (±2,05 IPC vs 2,1 IPC). A tabela 3.2 resume algumas métricas da arquitetura SMT básica.

TABELA 3.2 - Comparações de Técnicas de Busca Segundo Tullsen

Métrica	Número de Tarefas		
	1	4	8
Faltas na *i-cache*	2,5%	7,8%	14,1%
Faltas na *d-cache*	3,1%	6,5%	11,3%
Taxa de erro de previsão br/jmp	7,2%	13,8%	23,0%
int / fp IQ-cheia (% de ciclos)	21,0%	19,0%	12,0%
instruções incorretas buscadas	24,0%	7,0%	7,0%
instruções incorretas remetidas	9,0%	4,0%	3,0%

O trabalho também avaliou o desempenho da busca, mantendo constante a largura de busca e atacando 3 fatores: 1) eficiência da busca (dividindo a busca entre diferentes tarefas), 2) efetividade da busca (melhorando a qualidade das instruções buscadas) e 3) disponibilidade da busca (eliminando condições que bloqueiam a busca). Tentou-se reduzir a fragmentação (quando o bloco retorna da *cache* sem estar totalmente preenchido) do bloco de busca através da busca de múltiplas tarefas a cada ciclo, enquanto mantinha-se a largura máxima da busca constante. Foram avaliadas as seguintes técnicas:

- **RR.1.8** (RR=*Round-Robin)* – busca 1 tarefa por ciclo com até 8 instruções por tarefa

- **RR.2.4**, **RR.4.2** - busca menos instruções por tarefas, de um total de 2 ou 4 tarefas, respectivamente.

- **RR.2.8** - busca as 8 instruções mais flexíveis de 2 tarefas. A implementação pode ser feita buscando-se de duas tarefas, 8 instruções de cada. Então,

utilizam-se quantas instruções for possível da primeira tarefa e completa com instruções da segunda tarefa, até um total de 8 instruções.

Também, foram analisadas políticas que ajudam na identificação da "melhor" tarefa disponível para buscar a cada ciclo. Todas as políticas tentam melhorar a técnica de *round-robin* (RR) da busca utilizando parâmetros de outras partes do processador. A primeira ataca a busca do caminho errado (*wrong-path*) e as demais atacam a saturação da fila de instruções com muitas das instruções não sendo remetidas (IQ *clog*). Todas as técnicas de busca provêem ganhos sobre a técnica de *round-robin* (RR). As tabelas 3.3 e 3.4 resumem os desempenhos aproximados em ipc.

- **BRCOUNT** - Conta-se o número de instruções de desvios que estão nos estágios seguintes à busca (decodificação, renomeação e fila de instruções), favorecendo as tarefas com menos desvios para serem resolvidos.

- **MISSCOUNT** - Uma instrução de longa latência de memória pode bloquear a fila de instruções, causando a saturação da mesma com instruções bloqueadas à espera do resultado da instrução de memória. Esta política previne esta situação dando maior prioridade para aquelas tarefas que possuem as menores quantidades de faltas pendentes na *d-cache*.

- **ICOUNT** - Esta é a solução mais geral de *IQ clog*. Aqui, a prioridade é dada para as tarefas com o menor numero de instruções nos estágios seguintes à busca.

- **IQPOSN** - Da mesma forma que ICOUNT, IQPOSN tenta minimizar o IQ *clog* dando menor prioridade às tarefas com instruções próximas à cabeça da fila de instruções (as instruções mais antigas estão na cabeça da fila de instruções). Tarefas com instruções mais antigas são mais propensas a causar IQ *clog*.

TABELA 3.3 - Desempenhos com Busca de 1 Tarefa por Ciclo

Desempenho Aproximado (ipc)				
Técnica Usada	Número de Tarefas			
	2	4	6	8
RR.1.8	2,8	3,6	3,9	3,8
BRCOUNT.1.8	2,8	3,6	4,2	4,3
MISSCOUNT.1.8	2,8	3,6	4,1	4,2
ICOUNT.1.8	3,3	4,2	4,4	4,5
IQPOSN.1.8	3,2	4,0	4,3	4,4

Tullsen concluiu que a busca de instruções é o principal gargalo nesta arquitetura, o que não implica que os demais fatores possam ser desprezados, tais como largura de busca, tamanho do arquivo de registradores, previsão de desvios e execução especulativa. A arquitetura proposta requer pouco *hardware* a mais do que uma arquitetura superescalar convencional e consegue obter desempenhos semelhantes para aplicações mono-tarefa mas com ganho superior a 2,5 na execução de 8 tarefas (5,4 ipc).

TABELA 3.4 - Desempenhos com Busca de 2 Tarefas por Ciclo

Desempenho Aproximado (ipc)				
Técnica Usada	Número de Tarefas			
	2	4	6	8
RR.2.8	2,8	3,7	4,3	4,3
BRCOUNT.2.8	2,8	3,7	4,3	4,5
MISSCOUNT.2.8	2,8	3,8	4,4	4,6
ICOUNT.2.8	3,3	4,6	5,1	5,3
IQPOSN.2.8	3,3	4,5	5,0	5,2

4 Projeto da Arquitetura SEMPRE

Sabe-se que grande parte dos processadores atuais é utilizada em estações de trabalho ou em servidores de rede, com seus recursos compartilhados por diferentes aplicações. Estas aplicações em conjunto com o próprio sistema operacional formam um conjunto de processos em execução, que pode ser chamado de sistema multi-processos. Essa disponibilidade de processos independentes pode justificar o desenvolvimento de arquiteturas multi-tarefas simultâneas, desde que estes processos possam ser executados simultaneamente pelo *hardware*.

Além disso, se o *hardware* pode gerenciar as execuções desses processos, muitos pequenos intervalos de tempo que normalmente são perdidos com o escalonamento e as trocas de contexto podem ser poupadas. Adicionalmente, um mecanismo de pré-busca de instruções pode ser usado para buscar processos que ainda serão executados e o problema da degradação da *cache* em função do aumento do número de processos pode ser reduzido.

Visando desenvolver processadores voltados para a execução de sistemas multi-processos, o presente trabalho apresenta o projeto da arquitetura SEMPRE. Esta arquitetura não deixa de ser multi-tarefas simultâneas e nem tão pouco superescalar, mas caracteriza-se pela capacidade adicional de extrair o paralelismo existente entre processos, com eficiente mecanismo de pré-busca, além de executar diretamente pelo *hardware* muitas das operações (onerosas em consumo de tempo de cpu) normalmente executadas no nível do sistema operacional. Estas características específicas permitem um maior desempenho na execução das aplicações. As seções seguintes descrevem a arquitetura SEMPRE.

4.1 Descrição Geral da Arquitetura

Utilizando os conceitos e técnicas apresentadas nos capítulos anteriores, a arquitetura SEMPRE foi projetada e é apresentada neste capítulo de acordo com sua proposta original. A arquitetura SEMPRE (SuperEscalar com Múltiplos PRocessos em Execução) foi proposta inicialmente em [GON 98a] e [GON 98b] como uma arquitetura multi-tarefas simultâneas com capacidade de executar processos. A figura 4.1 mostra os 5 estágios funcionais básicos de seu *pipeline* superescalar: busca, decodificação, execução, término (ou retorno) e conclusão, providos de técnicas usuais de previsão de desvios (busca e execução especulativa), renomeação de registradores e execução fora-de-ordem.

A presente arquitetura é capaz de executar diferentes fluxos de instruções provenientes de diferentes aplicações (ou processos). As instruções destes processos são buscadas em diferentes posições da memória, e são executadas simultaneamente (“misturadas”) dentro do *pipeline* SMT. Além disso, a arquitetura SEMPRE escolhe diretamente por *hardware* quais processos são executados em dado momento. Para permitir esta funcionalidade, diversas estruturas são utilizadas:

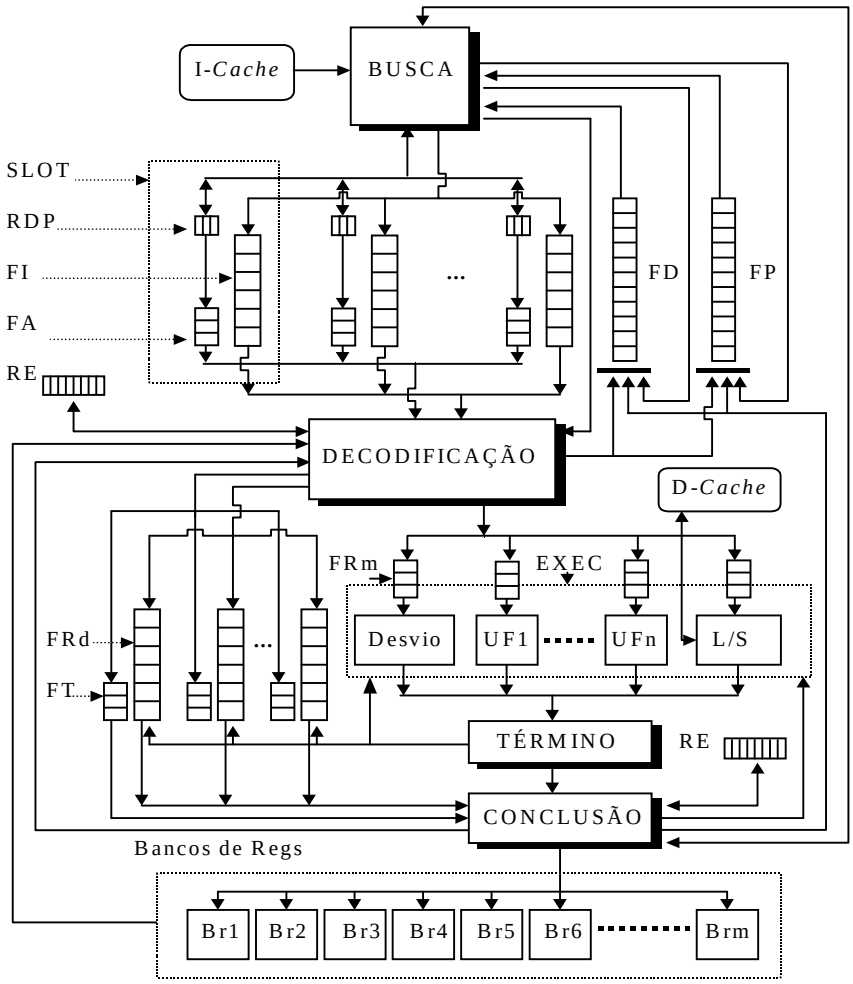


FIGURA 4.1 - Arquitetura SEMPRE

- Um conjunto de *slots* para conter informações sobre os processos que são simultaneamente executados. Cada *slot* contém um registrador de descritor de processo (RDP), que é usado para fornecer o endereço de busca do próximo bloco de instrução para o referido *slot*; uma fila de instruções (FI) onde são colocadas as instruções buscadas e uma fila de processos ativos (FA), para controlar quais processos que estão presentes dentro de um *slot*. Cada fila de ativos é necessária para que a arquitetura possa trocar o contexto de um processo e inserir instruções de um outro processo dentro do mesmo *slot* (ver seção 3.1), antes que o primeiro tenha sido totalmente concluído. Cada entrada na fila de ativos pode armazenar um descritor de processo. Uma entrada nessa fila é inserida quando um novo processo é escalonado para o *slot*. Neste momento, uma cópia da entrada anterior é reinserida no final da fila de processos, para que o processo que foi trocado possa ser novamente escalonado.

- Uma fila de bancos disponíveis (FD), para conter os identificadores dos bancos de registradores que não estão sendo utilizados pelos processos. Inicialmente todas as entradas estão disponíveis, e durante a execução, uma entrada nesta fila é retirada quando um novo processo é criado e é devolvida quando as últimas instruções de um processo são concluídas.

- Uma fila de processos (FP), para conter descritores dos processos que são criados e que podem ser escalonados para a execução simultânea dentro do *pipeline*. Cada descritor compõe-se basicamente de: contador de programa (CP), *fatia de tempo* (TS) fornecido pelo sistema operacional, identificador do banco de registradores (Bid) utilizado para o contexto do processo, e o *status* do processo (Stt), que indica o seu estado atual, que pode ser "pronto", "suspenso" ou "morto". Neste trabalho, "descritor de processo" é tratado apenas como "processo".

- Um conjunto de filas de remessa (FRm), sendo uma para cada unidade funcional, que contém instruções para serem remetidas para as unidades funcionais. Estas filas são preenchidas por instruções prontas despachadas pelo estágio de decodificação.

- Um conjunto de unidades funcionais heterogêneas compartilhadas por todos os processos.

- Dois registradores de escalonamento (RE). O primeiro auxilia no escalonamento de instruções prontas das filas de instruções e o segundo auxilia no escalonamento de instruções terminadas durante o estágio de conclusão.

- Um conjunto de registradores, dividido em bancos (Br's) privados, para armazenar os contextos dos processos. Cada banco é identificado por um Bid diferente, que faz parte do descritor de processo e permite a leitura e a escrita nos registradores corretos.

- Um conjunto de filas de reordenação (FRd), cada uma associada a um *slot* diferente com duas finalidades principais: 1) manter a ordem de execução do programa original e 2) prover renomeação de registradores. Da mesma forma como as filas de instruções, as filas de reordenação podem conter instruções provenientes de mais de um processo.

- Um conjunto de filas de processos em trânsito (FT), que contém os identificadores dos processos que possuem instruções nas filas de reordenação. Cada FT é associada a uma FRd, e conseqüentemente, a um *slot* diferente. Uma entrada nessa fila é inserida quando a primeira instrução de um processo é despachada de um *slot* e é retirada quando a última instrução do processo deixa a fila de reordenação.

As filas de processos ativos e de processos em trânsito, além de servirem para o controle dos processos que estão sendo executados no *pipeline*, evitam que as informações nelas contidas sejam colocadas em cada uma das instruções, o que implicaria em aumento considerável dos caminhos das instruções. Todas as estruturas apresentadas são manipuladas durante os diferentes estágios da arquitetura, que são discutidos nas próximas seções.

4.2 Estágio de Busca

Em arquiteturas multi-tarefas, o estágio de busca pode se tornar um gargalo, pois a *cache* é compartilhada por diferentes processos e pode sofrer séria degradação. A arquitetura SEMPRE alivia esta degradação através de um mecanismo de pré-busca conforme discutido na seção 4.2.2. De uma forma geral, as principais funções

do estágio de busca são: busca de instruções da *cache*, previsão de desvios, troca de contexto e execução de instruções privilegiadas, explicadas a seguir.

4.2.1 Busca de Instruções da *Cache*

Durante a busca de instruções na *i-cache*, é buscado um bloco de instruções por vez, para cada *slot* da arquitetura, no estilo *round-robin*. Este algoritmo é utilizado devido a sua simplicidade e também pelo fato de ter sido amplamente pesquisado por Tullsen [TUL 96] sob diversas adaptações. Este tipo de escalonamento é chamado de "escalonamento da busca" e é utilizado simplesmente pela dificuldade em se buscar diferentes fluxos simultaneamente. Pode-se dizer que a busca não é simultânea, e também não pode ser confundida com a troca de contexto, que é explicada mais a seguir.

Uma das maiores preocupações com relação à busca de instruções é garantir que o *slot* não fique vazio enquanto outros processos estiverem sendo buscados. Nesse sentido, a quantidade de instruções a serem buscadas se torna um fator importante. Para satisfazer esta necessidade, Hirata [HIR 92] sugere que cada bloco contenha pelo menos N*T instruções, onde N indica o número de *slots* existentes e T o tempo médio necessário para buscar o bloco na *cache*. Isto tende a favorecer que cada *slot* permaneça com instruções até a próxima busca. A modelagem analítica apresentada no capítulo 5 mostra que existem outros fatores que devem ser considerados para se chegar ao tamanho do bloco ideal, tais como a freqüência de decodificação e a taxa de faltas na *cache*.

Uma vez selecionado o *slot*, a busca é feita a partir do endereço do contador de programa localizado no registrador RDP, e as instruções buscadas são inseridas na respectiva fila de instruções. Durante esta inserção, a busca checa por instruções de desvios e instruções especiais, executando ações específicas nestes casos. A figura 4.2 mostra o estágio de busca e alguns dos seus relacionamentos.

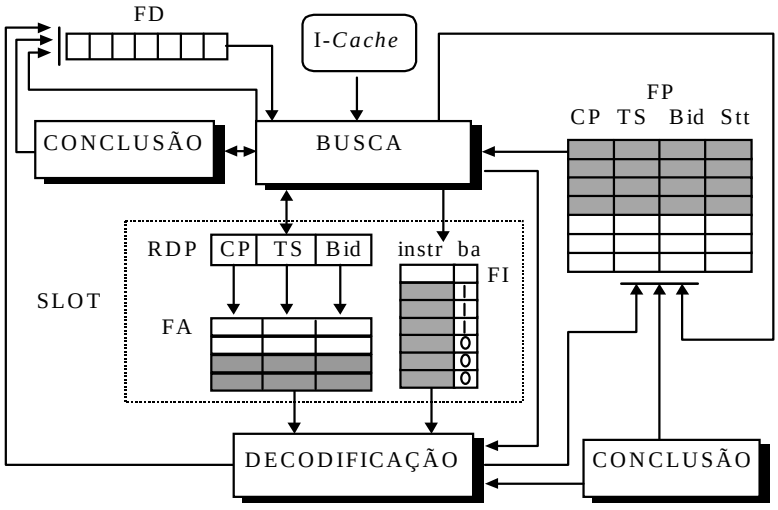


FIGURA 4.2 - Esquema do Estágio de Busca

Deve-se observar que existe uma tabela de páginas para cada processo, da forma tradicional, também chamada de TLB (*translation lookaside buffer*), e que, durante a

busca, o endereço lógico é convertido para o endereço físico. De uma forma geral, uma falta na tabela de páginas também é considerada como uma falta na *cache* de instruções. Conforme dito, o endereço desta tabela é provido inicialmente junto com o contador de programa durante a criação do processo.

4.2.2 Estágio de Pré-Busca de Instruções

Em arquiteturas superescalares mono-tarefa (convencionais), uma alternativa simples para reduzir a penalidade causada pelas faltas na *i-cache* é a utilização de pré-busca de instruções [LEE 95]. Este tipo de falta é do tipo intra-tarefa e um estudo mais detalhado sobre diversas políticas de pré-busca de instruções para estas arquiteturas pode ser encontrado em [SAN 00]. Em arquiteturas SMT, a grande quantidade de tarefas disputando a mesma *cache* impõe um novo tipo de degradação: as faltas entre-tarefas na *i-cache*, que ocorrem quando uma tarefa escalonada para a execução causa a retirada das instruções de outra tarefa que está na *cache*.

Apesar disto, as faltas na *i-cache* na arquitetura SEMPRE não são tão preocupantes quanto em arquiteturas mono-tarefa, pois elas podem ser escondidas pela troca de contexto, substituindo a tarefa interrompida por outra tarefa escalonada. Mas um fator pode impedir esta facilidade: a não existência de tarefas prontas na *i-cache* nível 1 durante uma solicitação de troca de contexto. Assim, torna-se necessário o desenvolvimento de um mecanismo de pré-busca capaz de antecipar a busca de instruções de uma tarefa, da *cache* L2 para a *cache* L1, antes que ela seja escalonada.

Quando o escalonamento é feito pelo sistema operacional ou por algum programa, isto não e possível, pois a falta somente é detectada após o escalonamento da tarefa. No caso do escalonamento por *hardware*, como é o caso da arquitetura SEMPRE, uma política de pré-busca obtém informações sobre a política de escalonamento e antecipa a busca das instruções dos processos antes mesmo deles serem escalonados. Este tipo de pré-busca reduz as faltas causadas pelas interferências entre-tarefas e garante que as faltas intra-tarefa sejam mascaradas.

O estágio de busca da arquitetura foi modelado para suportar este mecanismo de pré-busca, conforme mostra a figura 4.3. A unidade de busca efetua a busca para aqueles processos que ocupam os *slots* da arquitetura, utilizando para isto os registradores RDP dentro da tabela de busca (TB). Enquanto isso, a unidade de pré-busca se responsabiliza em buscar instruções para processos que ainda serão escalonados futuramente e que ainda não estão presentes na *cache* L1. Para gerenciar este mecanismo, cada entrada na fila de processos FP contém um *flag* chamado “miss-*status*” que informa se o referido processo está ou não na *cache* L1, de acordo com valores 1 ou 0, respectivamente.

Para cada processo buscado, o seu *miss-status* é marcado com 1. Sempre que um processo tem seu contexto trocado pela ocorrência de uma falta na *i-cache*, este processo volta para a fila de processos com o *miss-status* marcado com 0. A eficiência do método é obtida quando na ocorrência de troca de contexto, há pelo menos um processo na fila de processos com o *miss-status* igual a 1. A arquitetura SEMPRE utiliza um algoritmo de escalonamento *round-robin*, qualquer tipo de algoritmo de

escalonamento pode ser usado, pois a unidade de busca deve informar para a unidade de pré-busca qual o PC do processo a ser pré-buscado.

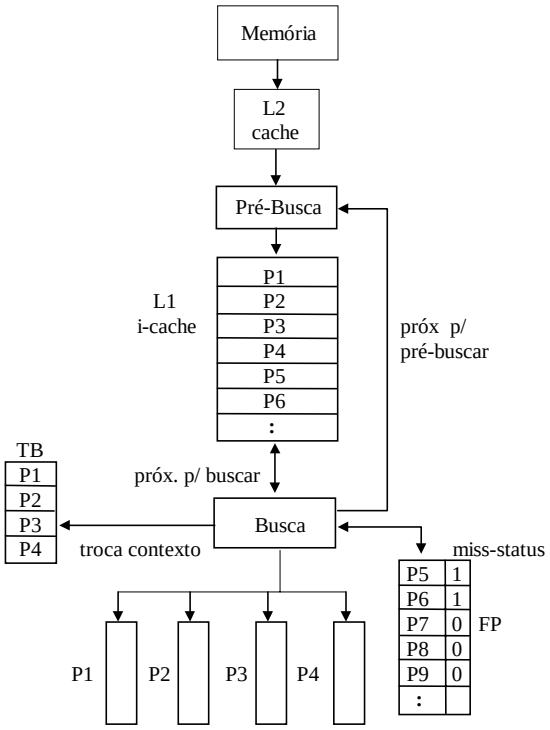


FIGURA 4.3 - Mecanismo de Pré-Busca da Arquitetura SEMPRE

Na figura 4.3 é apresentada uma possível situação da tabela de busca e da fila de processos, considerando a atuação da busca e da pré-busca de instruções. Nesta figura, os processos P1, P2, P3 e P4 estão presentes nos *slots* e, conseqüentemente, estão sendo buscados. Enquanto isso, verificando a fila de processos, a pré-busca vai antecipando a busca de processos que serão os próximos a serem escalonados. Os processos P5 e P6 já foram pré-buscados e seus respectivos *miss-status* estão marcados com 1. No referido exemplo, a *i-cache* mantém 6 processos.

Com o objetivo de analisar o comportamento e desempenho, bem como direcionar as tomadas de decisões durante o desenvolvimento do simulador dirigido por execução, o mecanismo de pré-busca foi modelado analiticamente e mostrou ser bastante eficiente sobre uma outra arquitetura SMT convencional, conforme mostra o capítulo 5.

4.2.3 Previsão de Desvios

Na ocorrência de uma instrução de desvio condicional, o estágio de busca faz uma previsão utilizando um mecanismo baseado em um *bit* de história, que pode obter até 90,23% de acerto nas previsões segundo Smith [SMI 81]. Como os diferentes endereços físicos indicam diferentes processos, eles são utilizados para acessar uma única tabela de previsão de desvios. Isto facilita o gerenciamento, embora a tabela tenha tamanho equivalente ao de várias tabelas convencionais. A utilização de um único *bit* de história

evita uma maior expansão ainda desta tabela. A arquitetura SEMPRE foi projetada para prever um único desvio por busca efetuada.

Para o primeiro desvio encontrado na ordem das instruções buscadas, o estágio de busca consulta o *bit* de história dentro da tabela e faz a previsão. Se a previsão for de desvio tomado, somente as instruções até a instrução de desvio são inseridas na fila de instruções, caso contrário, as demais instruções no caminho não-tomado também são inseridas até o ponto em que ocorrer uma nova instrução de desvio ou até o total de instruções buscadas. A previsão feita (tomado ou não tomado) é armazenada juntamente com a instrução na fila de instruções e na *cache* também. O campo CP do RDP também é atualizado para o endereço previsto, para que as próximas buscas sejam feitas a partir do endereço alvo. O estágio de conclusão se responsabiliza pela verificação da corretismo da previsão.

4.2.4 Troca de Contextos

As arquiteturas multi-tarefas tradicionais (“não-simultâneas”) trocam o contexto das suas tarefas, parando a busca de uma e iniciando a busca de outra, sobre o mesmo *pipeline*, e por isso somente uma única tarefa ocupa os recursos do *hardware* em determinado momento. Já as arquiteturas SMT, tradicionalmente executam um número de tarefas simultaneamente e a troca de contexto fica a cargo do sistema operacional. Na arquitetura SEMPRE, devido à existência de uma fila de processos e ao escalonamento ser feito pela própria arquitetura, o contexto de qualquer tarefa pode ser trocado por outro, dinamicamente, aproveitando assim o tempo que normalmente é perdido com a espera do tratamento das faltas na *cache* de instruções, que pode ser utilizado para executar uma outra tarefa ativa. Assim, dizemos que a arquitetura SEMPRE é sensível à faltas na *cache*.

Na arquitetura SEMPRE, em dado momento, existem diversas instruções sendo buscadas para diferentes *slots* e, conseqüentemente, existem instruções de diferentes contextos sendo executadas. Uma troca de contexto implica em parar a busca de instruções para determinado processo e iniciar a busca para outro processo a ser escalonado para o mesmo *slot*. A escolha de um novo *slot* a ser buscado faz parte apenas do escalonamento da busca, conforme já explicado, e não deve ser confundida com a troca de contexto em si.

A troca de contexto não requer o esvaziamento do *slot* para o posterior preenchimento com outro contexto. Na arquitetura SEMPRE, as instruções do novo contexto são concatenadas, na seqüência, com as instruções do contexto antigo, ainda remanescentes no *slot*. Isto antecipa a busca do próximo contexto, e por conseqüência, maximiza a utilização das filas de instruções. O controle de quais contextos que estão inseridos no *slot* é feito através da fila de ativos (FA), que contém uma entrada para cada contexto ativo do *slot*. De uma forma geral, a troca de contexto ocorre nas seguintes situações:

1) durante a ocorrência de falta na *i-cache*: quando a unidade de busca solicita a busca na *cache* de instruções e ocorre uma falta, o contexto é trocado. Uma falta na *cache* geralmente ocorre durante uma instrução de desvio tomado (ou previsto como tal) cujo endereço alvo não está presente na *i-cache*, ou durante a ocorrência das

demais instruções de transferência de controle tais como *jump, call* ou *return*. Durante a ocorrência dessas instruções, o contador de programa é atualizado e a busca é interrompida para o referido processo. Este processo volta novamente para a fila de processos para ser escalonado, embora as suas instruções já buscadas sigam a execução normal pelo *pipeline*. Em tempo correto, um mecanismo de pré-busca (subseção 4.2.2) antecipa o carregamento das instruções ao longo do novo caminho antes que o processo seja novamente escalonado.

2) durante a ocorrência de instruções de entrada e saída: se durante a busca é detectada alguma instrução de entrada e saída, tal como *in* e *out*, a busca é também interrompida. Isto é feito devido ao longo tempo de espera para a conclusão destas operações, que pode ser aproveitado para a execução de outros processos. Neste caso, é necessário trocar o contexto a partir desta instrução, ou seja, nenhuma outra instrução seguinte é colocada na fila de instruções para não bloquear o despacho das outras instruções do novo processo, caso contrário, de nada adianta a troca.

3) durante o término dos processos: quando a unidade de busca detectar uma instrução de término normal de processo, o contexto é trocado a partir desta instrução. As instruções que foram buscadas após esta instrução são descartadas, pois a arquitetura não sabe a qual processo elas pertencem.

4) mediante o término da fatia de tempo do processo: todo processo possui uma "fatia de tempo", que determina o tempo máximo de execução ininterrupta, e que na arquitetura SEMPRE é controlada diretamente pelo *hardware*. Este tempo é medido em ciclos. O valor inicial é fornecido pelo sistema operacional e a unidade de busca controla o decremento de 1 a cada ciclo decorrido. A cada troca de contexto a fatia de tempo é reinicializada.

5) mediante solicitação da unidade de conclusão: na confirmação de previsão incorreta ou na ocorrência de exceções, a unidade de conclusão solicita uma troca de contexto, ou para que o endereço de busca seja corrigido, ou para que o processo seja morto. Se o processo estiver na fila de processos, quando ele for novamente escalonado ele é descartado ou tem o seu contador de programa atualizado. Se ele estiver em busca, seu contexto é trocado e a unidade de decodificação irá tratar dele. Muitas instruções podem ser removidas do *pipeline* (seção 4.7).

6) mediante a execução de instruções privilegiadas: durante a execução de algumas instruções pelo sistema operacional, conforme explicadas na próxima seção.

Sempre que é necessária a troca de contexto, o estágio de busca escalona um processo da fila de processos e verifica o seu campo *status* (*Stt*). Se o *status* indicar *processo-suspenso*, ele é re-inserido no final da fila de processos. Se o *status* indicar *processo-morto*, ele é descartado e seu o identificador do banco de registradores utilizado é inserido na fila de bancos disponíveis. Mas se o *status* indicar *processo-pronto*, ele é inserido na fila de ativos e uma cópia do mesmo descritor é mantido no RDP, para as futuras referências durante a busca de instruções. Se o *status* indicar *previsão-incorreta* o endereço alvo é corrigido e o processo é tratado como um processo pronto.

O contador de programa do processo anterior que se encontra na fila de ativos, é atualizado com o CP contido no RDP, antes do seu contexto ser trocado. Este processo

ainda permanece na fila de ativos até que sua última instrução seja despachada, embora uma cópia do seu descritor é colocada de volta na fila de processos para agilizar o seu futuro escalonamento. Se por ventura este mesmo processo é escalonado novamente para a execução, sendo que ainda restem instruções suas dentro de alguma fila de instrução devido a uma busca anterior, ele somente pode ser escalonado para este mesmo *slot*. Esta técnica serve para evitar que um mesmo processo seja buscado para duas filas de instruções diferentes, o que poderia acarretar a decodificação/despacho fora de ordem. Mesmo após o processo voltar para a fila de processos, ainda poderá haver instruções deste mesmo processo em trânsito pelo *pipeline*. As novas instruções são inseridas na fila de instruções com o *bit* alternador inverso ao *bit* alternador do contexto anterior (veja figura 4.2). A utilização desse *bit* é para facilitar a remoção destas instruções quando for o caso (vide seção 4.7).

4.2.5 Execução de Instruções Privilegiadas

Na arquitetura SEMPRE o estágio de busca executa algumas funções, antes executadas pelo sistema operacional. O sistema operacional gerencia os processos e o *hardware* apenas provê facilidades para tal. A figura 4.4 mostra os estados e transições de um processo no nível de arquitetura. As instruções privilegiadas definidas são 4.

- ***create*** **op1 op2 rd :** cria um novo processo - obtém um identificador da fila de banco*s* disponíveis e insere um novo processo na fila de processos, contendo: o *Bid*, o contador de programa (que deve ser fornecido em *op1*) e a *fatia de tempo* (que deve ser fornecida em *op2*). O *Bid* é retornado para o sistema operacional em *rd*, como identificador do processo. O contador de programa provido inicialmente pelo sistema operacional contém dois campos: o endereço de base, que serve para acessar a tabela de páginas e o deslocamento propriamente dito, que é utilizado para os incrementos durante a execução dos processos. A arquitetura SEMPRE limita o número máximo de processos em execução simultânea de acordo com o número de bancos de registradores disponíveis. Caso isso ocorra, um código de erro é retornado em *rd*.

Esta decisão limita apenas a capacidade do sistema operacional de criar novos processos dentro da arquitetura, mas não o impede de criar processos em memória. Na verdade, o sistema operacional deve ser hábil para controlar e gerenciar um número muito maior de processos, mas deixando parte deste serviço para arquitetura, que o fará com muito menos custo. Além disso, o sistema operacional pode também suspender um processo de menor importância e desocupar um banco de registradores para um outro processo mais prioritário.

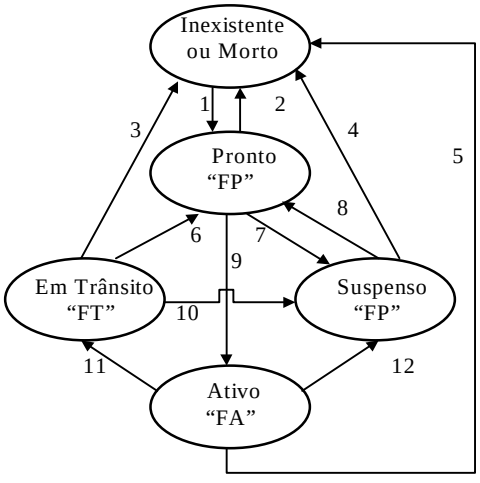
- ***kill*** **op1 rd :** mata um processo - O processo cujo *Bid* é fornecido em *op1* é eliminado da arquitetura. O *rd* retorna o contador de programa do processo. Quando a unidade de busca detecta esta instrução, a localização do referido processo é desconhecida e algumas ações são tomadas. Caso o processo esteja sendo buscado, a busca de instruções é interrompida e o contexto do processo é trocado. Nesse caso, o estágio de decodificação é sinalizado para tratar da morte deste processo. Se o processo estiver esperando na fila de processos, ele é removido quando for escalonado novamente, pois a unidade de busca mantém uma tabela de processos a serem removidos. O estágio de busca também sinaliza o estágio de conclusão para que as

instruções do referido processo, que estão em trânsito no *pipeline*, sejam removidas (ver mais na seção 4.7).

- ***suspend*** **op1 op2 rd:** suspende um processo - O processo, cujo identificador está em *op1*, é "suspenso". O *rd* retorna o sucesso da operação. O processo permanece nesta condição até que uma instrução *resume* seja encontrada para ele. A busca também é interrompida quando for o caso. Inicialmente, a unidade de busca guarda o identificador deste processo até que o mesmo seja escalonado novamente. Neste momento, seu *status* é marcado como "suspenso", o processo é re-inserido na fila de processos e seu identificador pode ser descartado. As instruções do referido processo que estão em trânsito não precisam ser removidas, pois foram buscadas antes da instrução *suspend*.

Existem dois modos de suspensão: em banco de registradores e em memória, que deve ser informado em *op2*. Na suspensão em memória, o contexto do processo é gravado em memória e o banco de registradores é disponibilizado, permitindo uma maior dinâmica para o sistema operacional quando o número de processos for maior que o número de bancos de registradores disponíveis. Na configuração padrão, a suspensão ocorre em banco de registradores.

- ***resume*** **op1 rd :** reassume um processo suspenso - O processo, cujo identificador está em *op1* e que está marcado como "suspenso", é simplesmente remarcado como "pronto" durante o próximo escalonamento. O *rd* retorna o sucesso da operação. No caso do processo ter sido suspenso em memória, existe a necessidade de um banco de registradores disponível para que a operação seja bem sucedida. Neste caso, o contexto do processo é primeiramente descarregado no referido banco de registradores. A figura 4.4 mostra o diagrama de transição de estados de processos.



Transições

1 - *create*
2 - *kill*
3 - *kill* ou execução irregular
4 - *kill*
5 - *kill*
6 - final de execução parcial do contexto
7 - *suspend*
8 - *resume*
9 - escalona (troca de contexto)
10 - *suspend*
11 - decodifica (inverte *bit* alternador)
12 - *suspend*

FIGURA 4.4 - Diagrama de Estados dos Processos na Arquitetura

O funcionamento resumido do estágio de busca é mostrado no algoritmo a seguir.

Algoritmo Busca;
1 **Para** cada_ciclo **Faça**
2 Seleciona_*slot* (*ns*) ; /* *round robin* */
3 **Se** *slot*_ocioso (*ns*) /* nenhum processo ocupando-o */
4 **Então** troca_contexto (*ns*) ; /* escalona processo */

```
5           volta_passo_1 ;
6   Fim-Se ;
7   Busca_instruções_para_o_processo (RDP(ns).CP, flag) ;
8   Se ocorrer_falta_na_cache (flag)
9      Então  troca_contexto (ns) ; /* escalona novo processo */
10             volta_passo_1
11  Fim-Se ;
12  Repita para cada_instrução_buscada
13  Caso instrução Seja:
14            branch: prevê_desvio (interrompe);
15            create: cria_processo ;
16            kill: mata_processo (interrompe) ;
17            suspend: suspende_processo (interrompe);
18            resume: reassume_processo ;
19     Fim-Caso ;
20     Coloca_instrução_na_FI (ns);
21  Até (slot_cheio(ns) ou (última_instrução) ou (interrompe)) ;
22  Se (RDP(ns).TS ≤ 0) ou (interrompe)
23     Então troca_contexto (ns) ;
24  Fim-Se ;
25  Fim-Para ;          /* repita para sempre */
```

4.3 Escalonamento das Instruções dos *Slots* e Decodificação

De uma forma geral, o estágio de decodificação executa três principais atividades em conjunto: escalonamento das instruções dos *slots*, decodificação e despacho (incluindo renomeação de registradores) para as filas de remessa (FRm) das unidades funcionais. A figura 4.5 mostra os relacionamentos existentes no mecanismo de decodificação, cujas atividades são esclarecidas a seguir:

O algoritmo de escalonamento, chamado aqui de "*round-robin* ponderado", utiliza um registrador RE contendo um identificador para cada *slot* da arquitetura, juntamente com um *bit* de despacho, que informa se alguma instrução daquele *slot* já foi despachada alguma vez. Inicialmente, todos os *bits* de despacho contêm 0 (zero), indicando que não houve nenhum despacho de nenhum *slot*. O algoritmo percorre o RE circularmente, e para cada *slot* ele inicia a decodificação das instruções. Durante esta fase inicial de decodificação é verificado se existe instrução pronta nas extremidades das filas de instruções, verificando os *bits* de ocupação dos registradores fontes. Também é verificado se existe entrada livre na fila de remessa apropriada. As instruções que satisfazem estas exigências são rotuladas como aptas para despacho.

As instruções aptas para despacho são decodificadas por completo e são então despachadas, juntamente com o número do *slot* e os controles decodificados. Para cada *slot* de onde uma ou mais instruções são despachadas, o *bit* de despacho é marcado com 1. Após o despacho, o RE é reorganizado de forma que os *slots* que já tiveram instruções despachadas são colocados no final do RE, para permitir que os outros *slots* sejam os próximos a serem escalonados. Este algoritmo favorece os processos com

maior número de instruções aptas, ao mesmo tempo em que cede a todos os *slots* a oportunidade de despachar. Quando todos os *bits* de despacho do registrador RE estão com 1, eles são todos reinicializados novamente com 0.

4.3.1 Despacho das Instruções

As instruções contidas nas primeiras entradas das filas de instruções dos *slots* são decodificadas e despachadas. Em sua definição original, a arquitetura SEMPRE previa somente o despacho de instruções prontas, pois, Segundo Hily e Seznec [HIL 98], a execução de instruções não-prontas em arquiteturas SMT não permite substancial melhoria no desempenho. Além disso, são conhecidas pelo menos 3 justificativas para se despachar instruções prontas:

1. a quantidade de instruções prontas é maior do que em arquiteturas superescalares tradicionais, pois são analisados vários fluxos de instruções independentes permitindo maior flexibilidade no escalonamento;

2. evita a saturação das filas de remessa com instruções que não podem ser remetidas imediatamente, e que podem estar impedindo que as instruções prontas de outros processos sejam executadas.

3. evita a necessidade de barramento de resultados, como no algoritmo de Tomasulo (seção 2.2.5.2), simplificando o projeto da arquitetura.

Entretanto, como não se pode garantir que o número de processos seja elevado, o desempenho pode ser pequeno se a arquitetura permitir somente o despacho de instruções prontas. Quando o número de processos é pequeno, é necessária uma maior quantidade de especulação para se obter bons resultados. Nas simulações realizadas no presente trabalho, foi observado que o despacho de instruções não-prontas aumenta substancialmente o desempenho em arquiteturas SMT com pequeno número de processos e grande largura de *pipeline*. Por este motivo, a decisão inicial foi modificada e a arquitetura SEMPRE foi projetada para despachar instruções não-prontas.

Quando uma instrução é despachada, a fila de instruções associada é deslocada e a próxima instrução é visualizada no início da fila de instruções. Quando o estágio de decodificação detecta uma inversão do *bit* alternador (*ba*), a fila de ativos também é deslocada, e a última entrada é então retirada da fila de instruções. Note que uma cópia do descritor de processo já foi inserida na fila de processos logo quando o processo trocou de contexto. A devolução do descritor de processo para a fila de processos, logo durante a troca de contexto, agiliza o próximo escalonamento do mesmo.

Aqueles processos que são ou solicitados pela unidade de busca ou pela unidade de conclusão para serem mortos pelo decodificador têm as suas instruções, que ainda não foram despachas, também removidas da fila de instruções, e as demais já despachadas são removidas durante o estágio de conclusão (vide também seção 4.7).

Em caso de execução especulativa, a unidade de decodificação também é solicitada pela unidade de conclusão para remover as instruções de um processo que está sendo executado no caminho incorreto. Neste caso, o descritor é devolvido para a fila de

processos para que a unidade de busca atualize o endereço alvo. Seu *status* é marcado com *previsão-incorreta*.

A primeira instrução decodificada de um novo contexto é despachada juntamente com uma cópia do identificador do processo associado que está na última entrada da fila de ativos. Este identificador é inserido na fila de processos em trânsito, associada ao respectivo *slot*, para prover informações aos estágios de término e conclusão sobre qual banco de registradores os dados devem ser referenciados. Este identificador é retirado da fila de processos em trânsito quando a última instrução do processo é concluída.

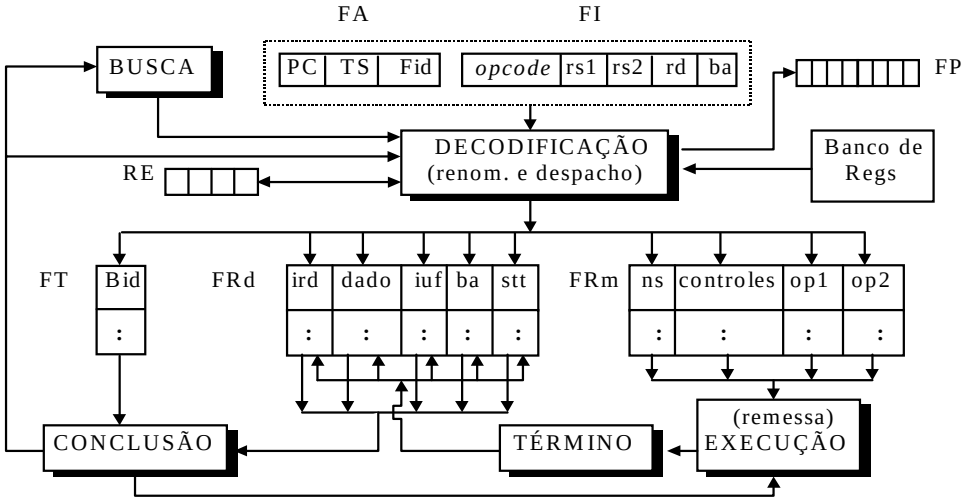


FIGURA 4.5 - Esquema do Estágio de Decodificação

Para cada instrução despachada, o registrador destino (*rd*) tem o seu *bit*-de-ocupação marcado como “ocupado”. Além disso, é inserida uma entrada na fila de reordenação (FRd) associada ao *slot* de onde a instrução foi despachada. Esta entrada contém o identificador do registrador destino, um campo para o dado resultante, o identificador da unidade funcional que executará a respectiva instrução, o *bit* alternador e o *status* da instrução, conforme mostra a figura 4.5.

4.3.2 Renomeação de Registradores Baseada na Fila de Reordenação

A renomeação de registradores da arquitetura SEMPRE mapeia os registradores destinos nas entradas das filas de reordenação (campo “dado”), semelhante a renomeação via fila de reordenação descrito na seção 2.2.5.1. Existem dois motivos básicos que pelos quais se faz necessário o uso de renomeação de registradores na arquitetura SEMPRE: 1) para garantir que instruções especulativas não alterem o estado da arquitetura e 2) eliminar as falsas dependências que existem entre as instruções.

Muitas das instruções são especulativas por terem sido despachadas após um desvio condicional não resolvido. Nesse caso, as filas de reordenação servem para armazenar os resultados temporários das instruções (renomeação) até que as mesmas sejam definitivamente escritas no banco de registradores. Além disso, as filas de reordenação servem para conter os dados gerados pelas instruções e garantir o controle das falsas dependências, pois isto elimina os conflitos entre os registradores destinos.

Posteriormente, no estágio de conclusão, o dado da fila de reordenação é retirado e gravado no registrador destino. O funcionamento resumido do estágio de decodificação é mostrado no algoritmo a seguir

```
Algoritmo Decodificação
1. Inicializa_RE ;      /* coloca 0 no vetor RE */
2. Para cada_ciclo Faça
3. Repita
4.     Seleciona_slot (ns) ;          /* round robin ponderado */
5.     Identifica_última_instrução (ns, Bid, instr, tipo) ;  /* analisa FI e FA */
6.     Se  não_cheia_FRm (tipo, entrada1) e
7.         não_cheia_FRd (ns, entrada2)
8.     Então  despacha_instrução_na_FRm (entrada1, ns, instr, info) ;
9.            insere_entrada_na_FRd (entrada2, iuf, info) ;
10.           desloca_FI (ns) ;   /* retira a ultima entrada */
11.           Se      é_primeira_instrução_de_novo_contexto
12.                   Então  copia_identificador_da_FA_para_FT (ns) ;
13.           Fim-Se ;
14.    Fim-Se ;
15.    Até preencher_barramento_de_decodificação_despacho ;
16.    Fim-Para ;          /* repita para sempre */
```

4.4 Estágio de Execução

Neste estágio, cada unidade funcional executa as instruções de sua respectiva fila de remessa, retirando aquelas que estão com seus operandos resolvidos (instruções prontas), providenciando a leitura dos mesmos, que podem estar localizados nos registradores reais ou ainda mapeados nas entradas das filas de reordenação. Neste estágio, existe um algoritmo de escalonamento, também baseado no estilo *round-robin*, que visita todas as filas de remessa e escolhe um conjunto misto de instruções provenientes de diferentes tarefas, para que seja executado. Uma maior prioridade é dada para as instruções de memória e de desvio. Apesar de que não constam nos gráficos aqui mostrados, algumas filas de leitura/escrita existem para garantir a integridade dos dados na memória, através do controle de dependências, conforme explicado na seção 2.2.3.4.

As unidades funcionais operam de forma independente, com latências diferentes e dependentes do tipo da instrução, assim como nas arquiteturas superescalares convencionais. Os operandos são atualizados pelo estágio de término, através de barramento de retorno de resultados da execução. Se uma instrução causar uma exceção, tal como divisão por zero ou acesso não permitido, o resultado da instrução é enviado para o estágio de término como um código de erro.

Um fator importante, o qual se apresenta nas arquiteturas multi-tarefas simultâneas, é a degradação da *cache* de dados, assim como na *cache* de instrução. A unidade funcional de leitura/escrita em memória é capaz de atender a cada vez uma solicitação de acesso proveniente de um programa diferente localizado em uma posição de memória diferente. Nesse sentido, o desenvolvimento de uma técnica de pré-busca de dados também pode

tirar proveito da fila de instruções. Estudos estão sendo feitos no sentido de aproveitar a técnica de pré-busca definida na subseção 4.2.2 para pré-buscar ambos dados e instruções, já que a *cache* L2 é unificada. O funcionamento básico de cada unidade funcional é mostrado no algoritmo a seguir.

Algoritmo Execução ;
1. **Para** cada_ciclo **Faça**
2. **Repita**
3. Escalona_*slot*;
4. Verifica_disponibilidade_de_UF;
2. Retira_instrução_pronta_da_fila_de_remessa (*instr*, *ns*) ;
3. **Até** preencher_barramento_de_remessa;
4. Executa_instruções (*instrs*, *resultados*) ;
5. Envia_resultados_para_término (*ns*, *resultados*) ;
6. **Fim-Para ;** /* repita para sempre */

4.5 Estágio de Término

Este estágio atualiza as entradas das filas de reordenação, para cada instrução que termina a sua execução, segundo o esquema de renomeação, da seguinte maneira: para cada unidade funcional que conclui a execução de uma instrução, a unidade de término procura na fila de reordenação (na ordem inversa a da inserção) associada ao *slot* da instrução concluída, a primeira entrada cujo campo *iuf* (identificador da unidade funcional) corresponde ao da unidade funcional em questão. Nesta entrada, ela grava o resultado da operação no campo *dado* e altera o campo de *status* para *instrução-terminada*.

Deve-se observar que as instruções despachadas para a mesma fila de remessa são inseridas e retiradas em ordem. Também, as instruções dentro de uma fila de reordenação estão na mesma ordem em que estavam dentro do *slot*. Assim, para uma instrução pertencente a um contexto posterior terminar a sua execução antes de outra instrução pertencente a um contexto anterior elas devem ser executadas por unidades funcionais diferentes. Como as instruções dentro de uma fila de reordenação são pesquisadas pelo identificador da unidade funcional executora, jamais haverá coincidência entre dois registradores, com os mesmos identificadores, utilizados por duas instruções de contextos distintos. Isto garante a eliminação das falsas dependências.

Em caso de exceção, o *status* é preenchido com *execução-excedida*, a ser tratado no estágio de conclusão. Também, em caso de instrução de desvio, o resultado obtido (tomado ou não tomado) é comparado com a previsão inicial. Se os valores diferem, o *status* da fila de reordenação é marcado com *previsão-incorreta* para ser tratado pelo estagio de conclusão e o campo *dado* é preenchido com o endereço correto do alvo. Caso contrário, o campo de *status* é marcado de forma normal, com *instrução-terminada*.

Os resultados das operações também são enviados para as filas de remessa, para preencher os campos dos operandos faltantes das operações com dependência de dados.

Isto permite que muitas operações fiquem prontas para execução. O funcionamento básico do estágio de término é mostrado no algoritmo a seguir.

```
Algoritmo Término ;
Para cada_ciclo Faça
   Repita
   1. Seleciona_unidade_funcional_com_resultado_pronto (iuf);
   2. Recebe_resultado_da_unidade_funcional (iuf, ns, resultado) ;
   3. Atualiza_FRd (ns, iuf, resultado) ;
   4. Atualiza_FRm (iuf, resultado) ;
   Até completar_largura_de_término ;
Fim-Para ;          /* repeat forever */
```

4.6 Estágio de Conclusão

Da mesma forma como feito pelo estágio de busca, o estágio de conclusão também utiliza um algoritmo de escalonamento *round-robin* ponderado que verifica quais instruções estão *terminadas* nas extremidades das filas de reordenação analisando o campo de *status*. O algoritmo de escalonamento é utilizado aqui porque a quantidade de instruções que podem ser concluídas muitas vezes é maior que a largura de conclusão. O algoritmo básico deste estágio pode ser visto a seguir.

```
Algoritmo Conclusão ;
1. Para cada_ciclo Faça
2. Repita
3.      Seleciona_FRd ;          /* round-robin ponderado */
4.      Caso resultado Seja:
5.           instrução-excedida:
6.           previsão-incorreta:     descarta_instruções ;
7.                                   avisa_busca_e_decodificação ;
8.           instrução-terminada:    grava_no_banco_correto (ns, resultado) ;
9.      Fim-Caso ;
10.Até completar_largura_de_conclusão ;
11.Fim-Para ;          /* repeat forever */
```

As instruções escolhidas são retiradas em-ordem dentro de cada fila de reordenação, através de deslocamento, e o resultado (campo *dado*) de cada instrução é gravado no registrador destino (apontado por *ird*) do banco correspondente (campo *Bid* da primeira entrada da fila de processos em trânsito). Este mesmo registrador tem seu *bit* de ocupação marcado como *livre* e pode ser lido por outra instrução.

Quando termina a execução de um contexto (através da verificação da inversão do *bit* alternador na fila de reordenação correspondente), o identificador do processo é retirado da fila de processos em trânsito. Analisando o *status* das instruções, o estágio de conclusão controla a execução especulativa, ocorrência de exceções e mortes de processos. Nesse caso, os estágios de busca e decodificação são alertados para tal. Em muitas situações, o estágio de busca também solicita a morte de um processo (ver mais na seção 4.7).

4.7 Remoção de Instruções Inconvenientes

Instruções inconvenientes são instruções cujos resultados devem ser desconsiderados. Como estas instruções podem estar ocupando muitos dos recursos do *hardware* (que poderiam ser utilizados por instruções úteis), elas são removidas o mais rapidamente possível. Estas instruções surgem em três situações básicas:

1. Instruções especuladas incorretamente: são detectadas durante o estágio de conclusão, quando o *status* indicar *previsão-incorreta*. Nesse caso, ambas as unidades de busca e decodificação são avisadas para tratar desta situação. Se for preciso, o contexto do processo é trocado, as instruções executadas no caminho incorreto são descartadas e o contador de programa é atualizado.

2. Instruções de um processo morto pelo sistema operacional: são detectadas no estágio de busca, quando uma instrução *kill* é encontrada. Nesse caso, ambas as unidades de decodificação e de conclusão são avisadas para tratar desta situação. Também, se for preciso, o contexto do processo é trocado, as instruções ainda em execução são descartadas, seu banco de registradores é disponibilizado e o processo é retirado do *pipeline*.

3. Instruções seguintes a uma exceção: são detectadas no estágio de conclusão, quando o *status* de uma instrução indica *execução-excedida*. Assim, ambas as unidades de busca e decodificação são avisadas para tratar desta situação e este processo é tratado como um processo a ser morto.

Deve-se considerar que as instruções inconvenientes estão misturadas pelo *pipeline* juntamente com outras instruções úteis, o que dificulta o processo de remoção. Mas a arquitetura SEMPRE possibilita a localização destas instruções através das diferentes filas de processos. Estas instruções podem estar em 3 regiões prováveis:

1. Quando o processo está na fila de processos: neste caso, as instruções estão dentro do banco, ou na memória (para alguns processos suspensos).

2. Quando o processo está na fila de ativos: neste caso, as instruções estão todas em ordem dentro da fila de instruções, e podem ainda estar sendo buscadas pelo estágio de busca caso o RDP contenha também o descritor do processo em questão.

3. Quando o processo está na fila de processos em trânsito: neste caso, muitas instruções podem estar nas filas de remessa e muitas já podem ter sido terminadas e seus resultados gravados na fila de reordenação associada. Além disso, muitas instruções podem ainda estar sendo buscadas e decodificadas.

A remoção das instruções destas filas é feita rapidamente deslocando-se todas as entradas até o aparecimento de um *bit* alternador invertido na instrução situada no início da fila, ou então, até o esvaziamento da fila. A comparação de um único *bit* é o principal motivo da rapidez desta operação.

4.8 Execução dos Múltiplos Processos

A execução dos múltiplos processos na arquitetura SEMPRE começa quando ela entra em funcionamento. Neste momento, o *hardware* insere automaticamente, como primeira entrada na fila de processos, o descritor do carregador do *boot* do sistema operacional, que já está em memória ROM, e executa uma instrução *create* para iniciar sua execução. A partir daí, com o sistema operacional em execução, os demais processos podem ser criados.

No nível da arquitetura, todos os processos criados pelo sistema operacional estão aptos a serem executados. Para algum processo ser suspenso, o sistema operacional deve executar explicitamente uma instrução *suspend*, e posteriormente uma instrução *resume*, se desejar que o processo seja novamente escalonado. Para a arquitetura, não interessa o motivo de uma suspensão, mas para o sistema operacional ela pode indicar várias situações tais como: sincronização entre processos, troca de mensagens, espera de recursos diversos etc. A figura 4.6 mostra a visão básica do sistema operacional com relação às transições de estado dos processos.

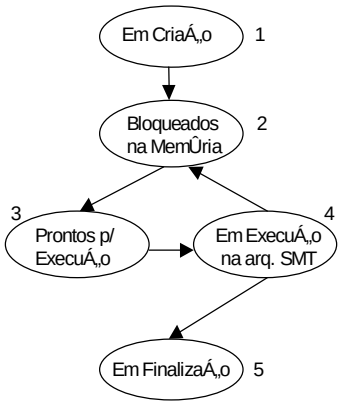


FIGURA 4.6 - Visão Geral do Sistema Operacional

No nível do sistema operacional, a arquitetura comporta-se como um multi-processador, permitindo que vários processos sejam colocados para a execução simultânea. O sistema operacional pode gerenciar os processos em diferentes filas de acordo com seus interesses. O sistema operacional não se preocupa com as trocas de contexto daqueles processos que estão em execução, que são feitas automaticamente pelo *hardware*, mas deve se preocupar com a troca de contexto dos processos que estão em memória. No nível da arquitetura, existem também as trocas de contexto por término da fatia de tempo e por faltas na *cache*, mas que não são sentidas pelo sistema operacional.

5 Modelagem Analítica da Arquitetura

Para evitar o desenvolvimento de arquiteturas que não atendam as necessidades de projeto, duas principais técnicas podem ser utilizadas para estimar o desempenho antes da sua implementação [SAG 99]: a simulação computacional e a modelagem analítica. A primeira abordagem é mais utilizada pois consegue representar um maior detalhamento da arquitetura a ser implementada e com isso provê resultados mais precisos, porém requer muito tempo de programação e simulação. Já a segunda abordagem esconde muitos detalhes e com isso provê resultados menos precisos, porém pode ser realizada de forma mais rápida e simplificada.

Em função destas características, a modelagem analítica pode ser utilizada com sucesso antes da simulação computacional pois, além de facilitar a análise comportamental da arquitetura proposta, logo no início do projeto, provê informações importantes que direcionam o desenvolvimento do próprio simulador, diminuindo o tempo de implementação deste. Muita pesquisa tem sido feita na área de modelagem analítica e muitas arquiteturas têm sido modeladas [MAR 84], [SAA 90], [YAM 94], [JAC 96] e [KAN 97].

Conforme já mencionado, um simulador da arquitetura SEMPRE foi desenvolvido e utilizado para validar o projeto da arquitetura. Para facilitar o desenvolvimento deste simulador e antecipar estimativas de comportamento da arquitetura na presença de processos na *cache*, o estágio de busca foi antes modelado analiticamente utilizando a ferramenta DSPN Express [LIN 98], que é baseada em Redes de Petri.

Esta modelagem permite analisar os possíveis gargalos que existem quando vários processos disputam a *cache* de instruções, mediante diferentes configurações. Na presente tese, o estágio de busca foi modelado com e sem mecanismo de pré-busca, para se poder detectar os ganhos de desempenho proporcionados por tal mecanismo. Este trabalho foi feito de forma cooperativa com o aluno de mestrado Rafael Linden Sagula, que desenvolveu um trabalho na área de modelagem analítica [SAG 99b] e [GON 99].

Deve-se ressaltar que o objetivo principal deste trabalho não é a modelagem analítica da arquitetura, e sim a arquitetura propriamente dita. Nesse sentido, não foi dada ênfase para o esclarecimento dos modelos, mas foi para os resultados obtidos. As seções seguintes apresentam os modelos e os resultados obtidos.

5.1 Modelagem da Unidade de Busca Ideal

Um dos principais desafios no desenvolvimento de uma arquitetura multi-tarefas é o projeto de uma unidade de busca eficiente que seja capaz de sustentar uma alta taxa de ocupação das filas de instruções. Nesse sentido, o desempenho da arquitetura em relação à busca de instruções pode ser medido pelo número médio de instruções disponíveis em todas as filas de instruções. Este índice de desempenho é aqui chamado de TOFI (Taxa de Ocupação das Filas de Instruções).

Nesta modelagem é considerada uma unidade de busca ideal, que não é afetada pelas interferências entre-tarefas e sempre que um contexto é trocado existe um outro processo pronto na fila de processos cujas instruções já estão na *i-cache*. Assim, os processos que estão sendo buscados estão sujeitos somente as interferências intra-tarefa, que são as responsáveis pela ocorrência de faltas na *cache* de instruções.

Primeiramente, foi desenvolvido um modelo estático para 4 *slots* conforme mostra a figura 5.1, que posteriormente foi transformado no modelo otimizado da figura 5.2. Este segundo modelo permite, em tempo de resolução, definir a configuração dos seguintes parâmetros arquiteturais: taxa de acertos na *i-cache* nível 1, latência da *cache* nível 1, largura de busca, latência de execução das instruções e número de *slots* (número de processos que são buscados).

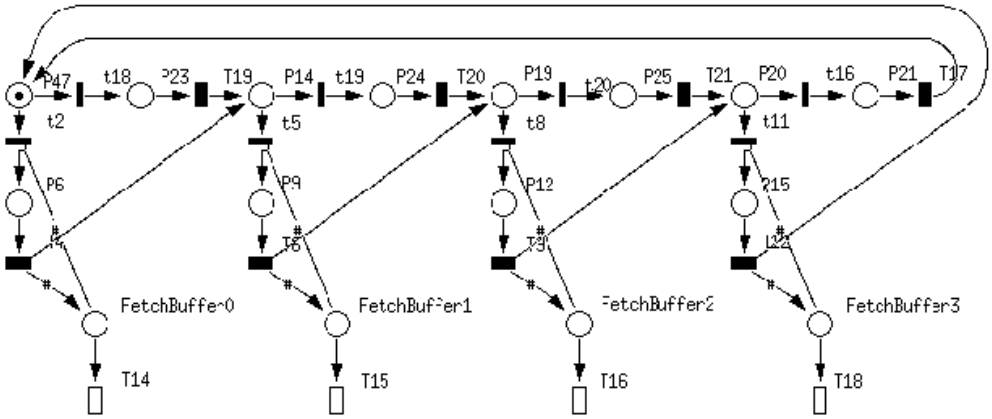


FIGURA 5.1 - Modelo para 4 *Slots* do Mecanismo de Busca

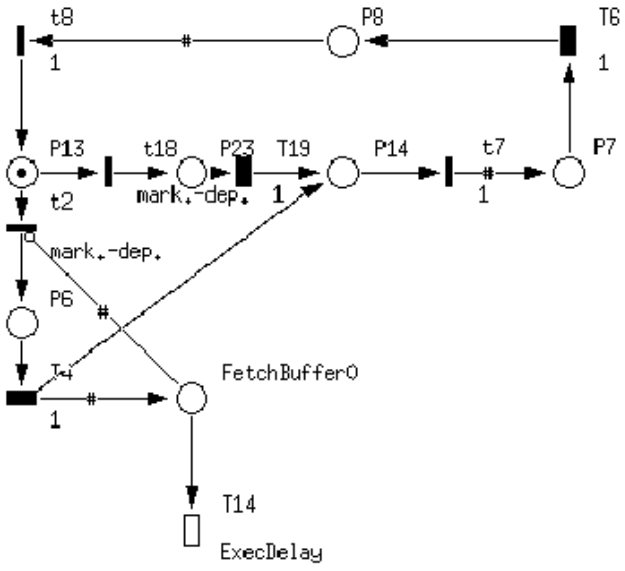


FIGURA 5.2 - Modelo Otimizado do Mecanismo de Busca

Para poder analisar os limites da unidade de busca, este modelo considera existir um número infinito de unidades funcionais capazes de consumir todas as instruções que estão nas filas de instruções. O objetivo é saber até que ponto a unidade de busca consegue manter elevado o TOFI. A latência da *cache* de instruções foi considerada de 1 ciclo para acerto na *cache*. Como uma falta na *i-cache* tem sua latência mascarada pela

troca de contexto, tanto na ocorrência de acerto quanto falta na *i-cache* o próximo *slot* é buscado no tempo máximo de 1 ciclo. Nesse sentido, as latências de acesso a *cache* nível 2 não são consideradas.

Na primeira configuração desta modelagem, é variado o número de *slots* (de 2 até 16) para diferentes larguras de busca (4, 6, 8 e 10), onde o TOFI obtido pode ser visto na tabela 5.1. No gráfico da figura 5.3, pode-se observar que os valores convergem para um limite máximo quando o número de *slots* é aumentado. Considerou-se uma taxa de acertos na *i-cache* de 80% com latência de execução de 1 ciclo.

TABELA 5.1 - TOFI em Função dos Número de *Slots* e Largura de Busca

Número	**Largura de Busca**			
de *slots*	**4**	**6**	**8**	**10**
2	4,871567	8,60887	12,52798	16,50313
3	6,273544	11,50576	17,18173	23,04421
4	7,090579	13,48623	20,68061	28,26982
5	7,536037	14,78429	23,24309	32,36248
6	7,769324	15,60911	25,08772	35,53814
7	7,887886	16,1169	26,38691	37,97561
8	7,946594	16,41913	27,27599	39,81361
9	7,97501	16,59328	27,86485	41,1658
10	7,988463	16,69041	28,24186	42,13206
11	7,994814	16,74332	28,47547	42,80083
12	7,997645	16,77113	28,6158	43,24952
13	7,998981	16,78571	28,69768	43,54153
14	7,999565	16,79299	28,74442	43,72587
15	7,999783	16,79661	28,77021	43,83975
16	7,999927	16,79829	28,78446	43,9081

Observa-se, nesses resultados, que o aumento no número de *slots* não causa modificações no TOFI a partir de um certo valor, pois a largura de busca passa a ser o limitante nesse caso. Além disso, algumas combinações diferentes geram resultados aproximados, como é o caso onde a largura de busca é 6 e o número de *slots* é maior que 8. Os valores obtidos nesse caso são próximos daqueles com largura de busca 8 e o número de *slots* 4, ou então com largura de busca 10 e número de *slots* 2.

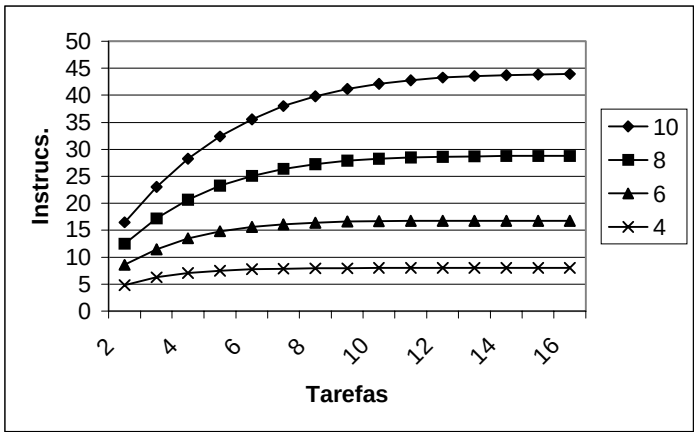


FIGURA 5.3 - TOFI x Número de *Slots* x Largura de Busca

Assim, outros fatores limitantes de desempenho devem ser levados em consideração para que a melhor alternativa seja escolhida. Mas também é possível que a decisão seja

baseada em fatores não relacionados à arquitetura, mas sim em restrições de projeto, tais como, por exemplo, custo final da arquitetura. Sabe-se que, quanto maior a largura de busca utilizada os componentes relacionados (*cache* e barramento) são mais caros, entretanto, o aumento no número de *slots* exige que uma área maior no *chip* seja utilizada somente para áreas de armazenamento e mecanismos de controle.

A figura 5.4 mostra a segunda configuração da modelagem, onde são variados a taxa de acerto na *i-cache* e o número de *slots*. Para tanto, manteve-se a largura de busca em 8 instruções e a latência das instruções nas FIs em 1 ciclo. Observa-se que o TOFI é diretamente proporcional à taxa de acerto na *i-cache*. Isso ocorre principalmente porque outros fatores que influenciam no desempenho da *cache* não são modelados, como por exemplo o número de *slots*, que diminui a taxa de acerto na *i-cache*. Entretanto, a ocorrência de poucos acertos na *cache* sugere que um número maior de *slots* seja utilizado para manter o *pipeline* com um maior número de instruções.

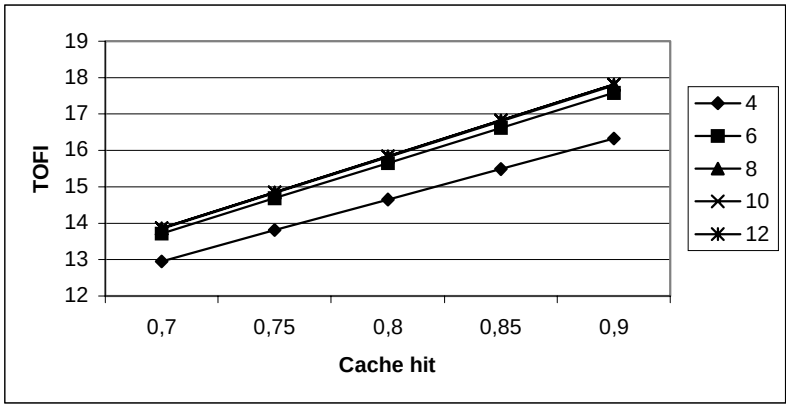


FIGURA 5.4 - TOFI x Número de *Slots* x Taxa de Acertos na *i-cache*

Na figura 5.5, é variada a latência das instruções nas FIs de 0,4 ciclo até 4 ciclos em média por instrução, o que significa que o grau de paralelismo varia de 0,26 instrução por ciclo (no caso de 3.8) até 2,5 instruções por ciclo (no caso de 0,4). Manteve-se a largura de busca em 8 instruções e o número de *slots* é também variado. Observa-se que, para aplicações com alto grau de paralelismo, a variação no número de *slots* não influencia muito no TOFI. Da mesma forma, essa variação cresce a medida que o paralelismo diminui.

Em uma primeira análise, pode-se concluir que, com uma latência de 0,5 ciclo para as instruções nas FIs, o aumento na quantidade de *slots* não proporciona nenhum ganho com relação ao TOFI, justificando assim a utilização de um número menor de *slots*. Com uma análise mais profunda na modelagem, percebe-se que este fato não necessariamente implica em maior desempenho em nível de instruções por ciclo, pois apesar do TOFI ser constante em ambas configurações, provavelmente a configuração com maior número de *slots* executa um maior número de instruções no nível global da arquitetura.

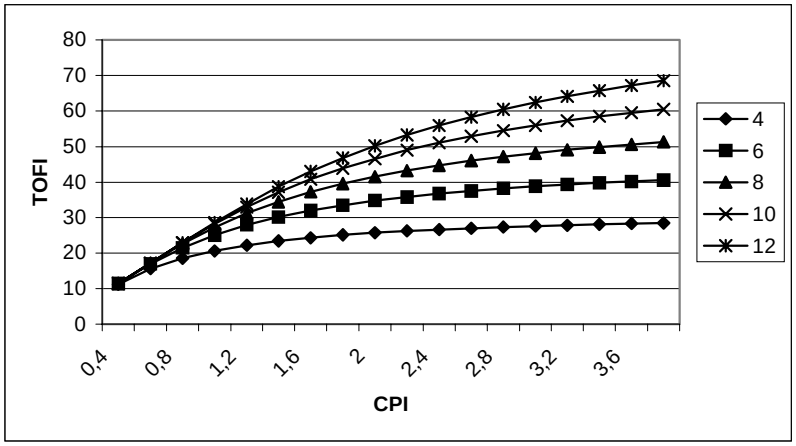


FIGURA 5.5 - TOFI x Latência de Instruções x Número deS*lots*

5.2 Modelagem da Unidade de Busca com Pré-Busca

Para analisar o comportamento de uma unidade de busca mais realista, que pode ser afetada pela ocorrência de faltas na *cache* durante as trocas de contexto e pelos atrasos decorrentes dos acessos na *cache* nível 2 (L2), foi desenvolvido um novo modelo que é apresentado na figura 5.6. Para reduzir as penalidades causadas por estes fatores, este modelo considera o mecanismo de pré-busca proposto na seção 4.2.2.

Neste modelo é suposto que todos os processos contidos na fila de processos estão inicialmente fora da *i-cache* L1. Assim, existe uma degradação momentânea enquanto não existem processos prontos para serem escalonados, mas na medida em que os processos são carregados pela pré-busca, eles passam a ser escalonados e permanecem no *slot* durante um tempo até que ocorra uma falta na *i-cache*.

Quando ocorre uma falta na *i-cache*, estes processos voltam novamente para a fila de processos e podem ser novamente pré-buscados. Nota-se que neste mecanismo, a *cache* L1 não se preocupa em obter os dados da *cache* L2 imediatamente na ocorrência da falta, pois esta tarefa e feita pela pré-busca em função da fila de processos. A pré-busca modelada tenta garantir que todos os processos escalonados da fila de processos já tenham sido carregados na *cache* de instruções, mascarando a latência com a troca de contexto, mas não impede a ocorrência das faltas intra-tarefa.

Considera-se *Capacidade de Despacho* como o número de instruções disponíveis para despacho, a cada ciclo, ou seja, a quantidade de instruções presentes nas últimas entradas das filas de instruções dos *slots* da arquitetura, prontas para a decodificação, e que a cada ciclo são todas consumidas. Uma vez que a modelagem foi feita em cima do mecanismo de busca e de pré-busca, a *Capacidade de Despacho* é usada nesta modelagem como medida de desempenho.

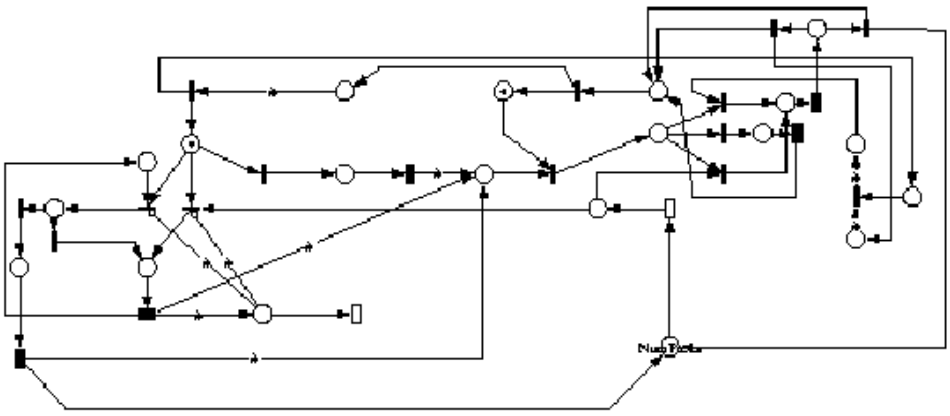


FIGURA 5.6 - Modelo Analítico da Arquitetura SEMPRE com Pré-Busca

Dois outros modelos que representam uma SMT Normal e uma SMT Ideal são obtidos a partir desse mesmo modelo, através de pequenas modificações. O modelo da SMT Normal considera que a arquitetura não dispõe de mecanismo de pré-busca, fazendo que a busca das instruções seja feita somente após a ocorrência de falta na *cache*. Este modelo representa o pior caso. O modelo da SMT Ideal supõe a existência de um mecanismo de pré-busca 100% perfeito, e é semelhante ao modelo considerado na seção 5.1. Neste modelo, que é considerado o melhor caso, sempre que há necessidade de uma troca de contexto, há processos já pré-buscados na *cache* L1 para serem escalonados.

Todos os gráficos apresentam uma legenda que descreve os 3 modelos utilizados nos experimentos: SMT Ideal, SEMPRE e SMT Normal. A configuração básica representa uma arquitetura com largura de busca de 8 instruções por ciclo; com 8 *slots* de busca; taxa de acerto na *i-cache* L1 de 80%; latência de pré-busca de 4 ciclos, que corresponde aos 3 ciclos obtidos no processador Pentium acrescidos de 1 ciclo para justificar a complexidade do mecanismo de pré-busca; e 16 processos presentes na arquitetura.

Na figura 5.7 pode-se observar o comportamento dos modelos em função da variação da taxa de acerto na *cache*. Com taxa acima de 80%, o modelo proposto de pré-busca iguala-se ao modelo ideal. Isso ocorre porque essa taxa não é suficiente para vencer a velocidade do mecanismo de pré-busca, ou seja, o fluxo de processos que sofrem faltas nos *slots* (1 processo a cada 5 ciclos em média) é menos intenso do que o numero de processos pré-buscados pelo mecanismo proposto (1 processo a cada 4 ciclos em média), mantendo sempre disponibilidade de processos prontos na *cache*.

Quando a taxa de acerto na *cache* é muito baixa a quantidade de trocas de contexto é muito intensa e a arquitetura com o mecanismo proposto não consegue atender a demanda de processos para serem pré-buscados, limitando o desempenho da arquitetura aos mesmos valores daqueles obtidos na arquitetura sem pré-busca. Entretanto, tais valores de acertos são incomuns (55%) e não ocorrem nas *caches* atuais. Obviamente que, nesta situação, o melhor caso supera em mais de 100% os demais modelos, pois nunca é afetado pela latência de pré-busca, o que lhe permite fazer a troca de contexto imediatamente na ocorrência de falta na *cache*.

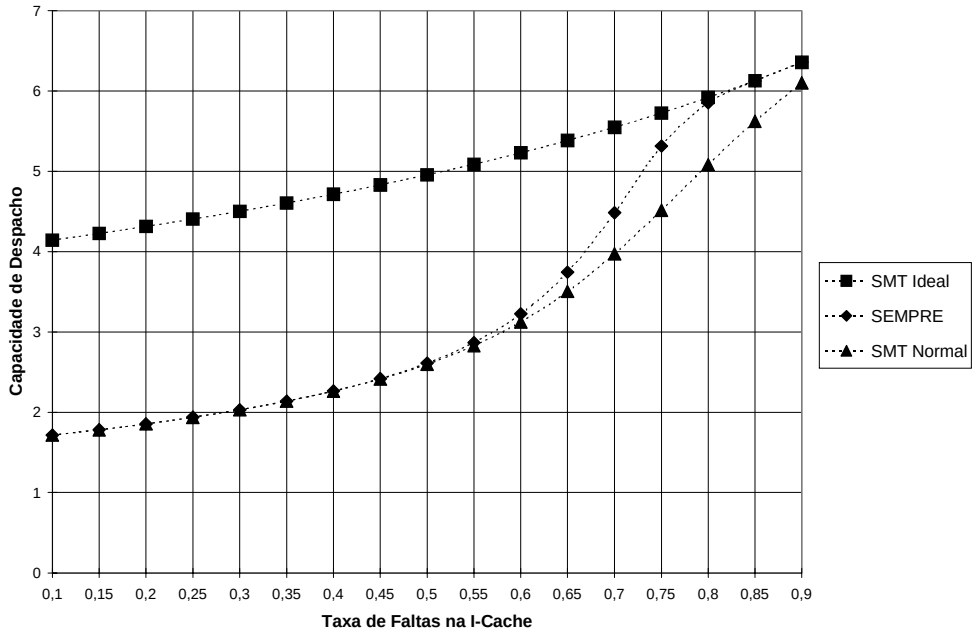


FIGURA 5.7 - Capacidade de Despacho x Faltas na *I-cache*

O melhor desempenho do mecanismo proposto é obtido na faixa de 70% a 85% de acerto na *cache*, que corresponde a um ganho de 9% a 18% em relação ao mecanismo sem pré-busca. Esses resultados são ótimos, pois esses valores de acertos na *cache* são esperados quando muitas tarefas são buscadas simultaneamente [TUL 95] e [TUL 96].

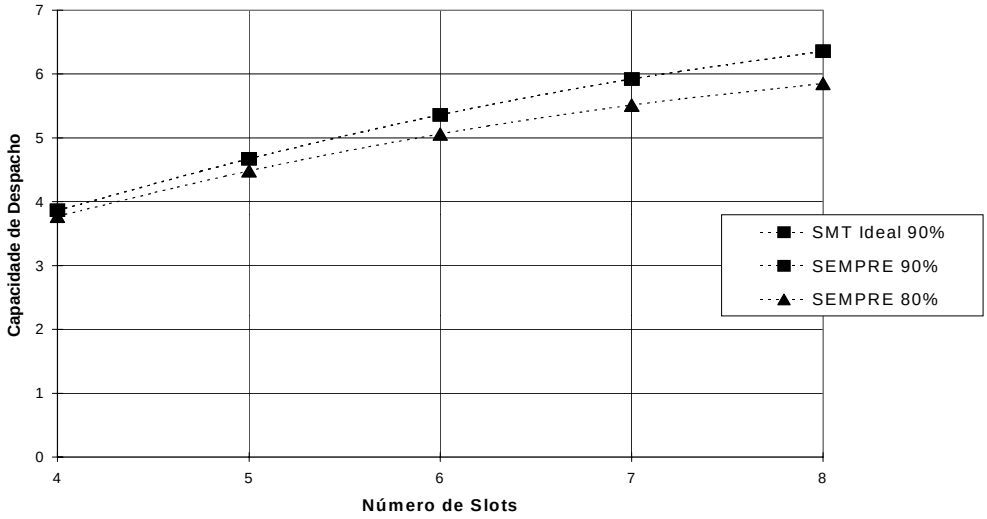


FIGURA 5.8 - Capacidade de Despacho x Número de *Slots*

O gráfico da figura 5.8 mostra o comportamento do mecanismo proposto e do melhor caso em função da variação do numero de *slots*. Nesta figura, quando a taxa de acerto na *cache* é de 90%, os desempenhos de ambas arquiteturas permanecem os mesmos. Contudo, quando esta taxa decresce 10%, a penalidade do mecanismo proposto decresce também porque quanto maior é a taxa de falta na *cache*, maior é o número de troca de contexto, e este fator é mais degradante se a quantidade de processos também é maior.

O gráfico da figura 5.9 mostra o comportamento de várias configurações para o modelo proposto e o pior caso, em função da variação da latência de acesso a *cache* L2. Em todas as configurações o modelo proposto supera o pior caso, mas as diferenças percentuais são maiores na medida em a taxa de acerto na *cache* se torna maior. Nota-se também que este ganho relativo inicia-se pequeno, aumenta substancialmente e depois volta a reduzir. A região onde se concentram os maiores ganhos se desloca para latências menores ou maiores em função de taxas de acerto na *cache* serem menores ou maiores, respectivamente.

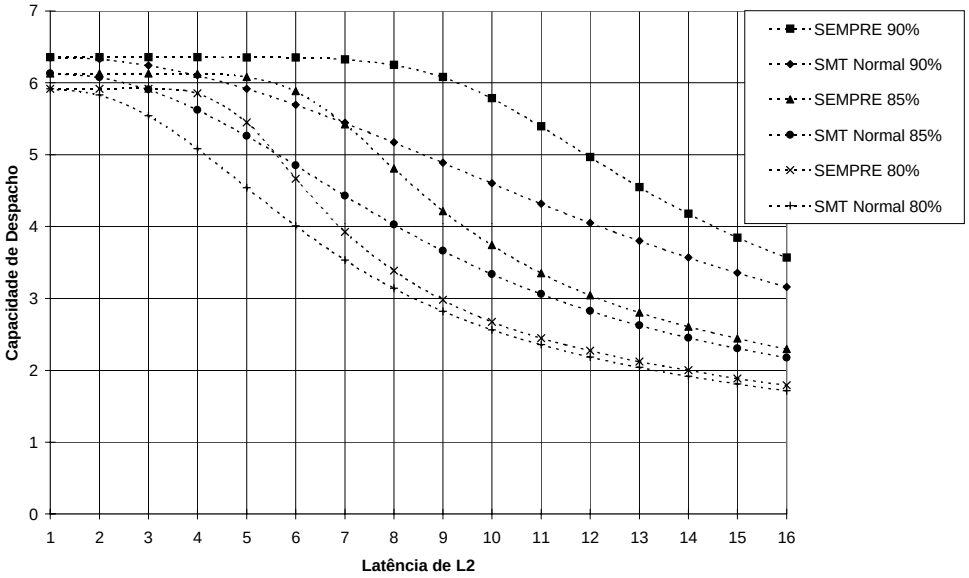


FIGURA 5.9 - Capacidade de Despacho x Latência da *Cache* L2

Assim, quando a taxa de acerto na *cache* aumenta, os melhores ganhos relativos são obtidos quando as latências são maiores, ainda que neste caso a capacidade de despacho efetiva tenha reduzido. Da mesma forma, quando a taxa de acerto na *cache* diminui os piores ganhos são obtidos quando as latências são menores, ainda que a capacidade de despacho efetiva tenha aumentado.

6 Simulação e Validação da Arquitetura

Para validar o projeto da arquitetura SEMPRE, foi desenvolvido um simulador dirigido por execução, a partir da ferramenta *SimpleScalar*. Primeiramente, foi feito um simulador SEMPRE básico, com o qual foi realizada uma série de simulações para avaliar aspectos ainda não conhecidos neste tipo de arquitetura. Posteriormente, foram adicionados neste simulador o escalonamento de processos e o estágio de pré-busca. Novas simulações foram realizadas então, comprovando os benefícios de tal projeto. As próximas seções descrevem as etapas da simulação e validação da arquitetura SEMPRE.

6.1 A Ferramenta *SimpleScalar*

A ferramenta *SimpleScalar Tool Set* [BUR 97] foi desenvolvida na Universidade de Wisconsin (Madison) como parte do projeto *MultiScalar*. Atualmente, ela tem grande popularidade e é usada como ferramenta de apoio para o desenvolvimento de simuladores de arquiteturas dirigidos por execução (*execution driven*) em diversos grupos de pesquisa. Ela é composta de um conjunto de rotinas em linguagem C que podem ser usadas para decodificar binários que estão em um formato específico "*ss*" (uma variação do conjunto de instruções MIPS), simular *caches* e previsores de desvios, além de funções de gerenciamento de recursos e entrada/saída. O pacote contém também um conjunto de programas de avaliação já pré-compilados em formato *ss*, facilitando o teste dos simuladores em desenvolvimento.

Como exemplos de utilização, já estão disponíveis alguns simuladores prontos. Um destes simuladores, chamado de *sim-outorder*, simula uma arquitetura superescalar de última geração, que incorpora previsão de desvios, renomeação de registradores e execução fora-de-ordem. Este simulador implementa uma Unidade de Atualização de Registradores (RUU) [SOH 90] para armazenar as instruções e tratar as dependências e executar a renomeação de registradores. A RUU mantém as instruções até que elas sejam concluídas e não sejam mais especulativas, funcionando com uma fila de reordenação circular (vide seção 2.2.2). Usando a RUU, o esquema de renomeação trabalha como explicado na seção 2.2.5.1, no qual a fila de reordenação incorpora internamente as estações de reserva. A arquitetura simulada pelo *sim-outorder* possui um *pipeline* de 6 estágios: busca, decodificação, remessa, execução, retorno e conclusão, conforme mostra a figura 6.1.

O primeiro estágio busca as instruções de cache de instruções (*i-cache*) armazenando-as em uma fila de instruções (i-fila). Durante esta busca, as instruções de desvios podem ser previstas usando uma das técnicas disponíveis (previsão de dois níveis, previsão combinada, previsão de caminho tomado e previsão de caminho não-tomado). Os endereços virtuais das instruções e dados são mapeados para endereços reais através de tabelas de tradução específica, i-TLB e d-TLB, respectivamente. As faltas na *cache* ou na tabela de instruções ocasionam o bloqueio da unidade de busca, por um número de ciclos a ser determinado em tempo de execução. O número de instruções úteis buscadas por ciclo depende da largura de busca (barramento de busca), da disponibilidade de entradas na fila de instruções e da exatidão da previsão de desvios. As instruções

disponíveis na fila de instruções são decodificadas e renomeadas no segundo estágio do *pipeline*, sendo armazenadas na RUU (ruu-q). As instruções de memória (*load* e *store*) são quebradas em duas: uma instrução de soma (*add*) para calcular o endereço efetivo da memória, que é armazenada na RUU, e uma instrução de acesso à porta de memória, que é armazenada na fila de memória (ls-q).

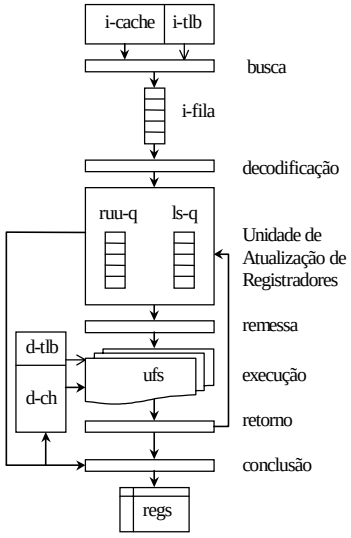


FIGURA 6.1 - Visão Geral da Arquitetura Simulada pelo Sim-OutOrder

A fila ls-q também é composta por um conjunto de estações de reserva ordenadas [SMI 95], assim com a ruu-q. Ambas contêm as informações da decodificação, operandos, *bits* de ocupação e rótulos para o controle das dependências. O número de instruções decodificadas por ciclo depende da largura de decodificação (barramento de decodificação) e da disponibilidade de instruções nas filas de instruções e de estações de reserva na RUU. O terceiro estágio verifica quais instruções das filas ruu-q e ls-q estão prontas para executar, isto é, quais delas possuem todos os operandos disponíveis e todas as dependências de memória satisfeitas, e as remessa para as unidades funcionais apropriadas. O número de instruções remetidas por ciclo depende do número de instruções prontas, da largura de remessa (barramento de remessa) e da disponibilidade de unidades funcionais e portas de memória.

O quarto estágio do *pipeline* executa as instruções e cada unidade funcional é mantida ocupada durante a latência da operação que estiver sendo executada. Entre as instruções de memória, somente as instruções de leitura são executadas neste estágio. As instruções de escrita são executadas no último estágio (conclusão), quando a computação já não for mais especulativa. As instruções de memória e de desvios são executadas com maior prioridade sobre as demais. Os resultados das instruções executadas são retornados para a RUU para liberar outras instruções que estavam com operandos faltantes. O estágio de conclusão verifica a fila ruu-q e retira as instruções concluídas em ordem. Neste momento, quando uma instrução de cálculo de endereço efetivo de memória (*add*) é retirada, a última entrada da ls-q, que deve ser a outra componente da operação de memória, também é retirada. Quando uma instrução de desvio é encontrada, a validação da previsão é feita. Se a previsão foi incorreta, todas as entradas da ruu-q e ls-q são eliminadas e a busca de instruções é redirecionada para o endereço correto. Os

resultados não especulativos são armazenados nos registradores e na memória definitivamente.

Durante cada ciclo simulado, todos os estágios do *pipeline* são executados e as estatísticas da execução são coletadas. O simulador *sim-outorder* é totalmente configurável e permite definir *caches*, TLBs, previsores de desvios, tão bem quanto outros parâmetros internos da arquitetura mencionados nos parágrafos anteriores. Este simulador foi usado para desenvolver um simulador SMT básico da arquitetura SEMPRE, conforme explicado na próxima seção.

6.2 Simulador SEMPRE Básico

Um simulador SMT [GON 00], base para a arquitetura SEMPRE, foi desenvolvido a partir de modificações no simulador *sim-outorder* mencionado na seção anterior. Esta implementação foi feita nos laboratórios do Departamento de Arquitetura de Computadores da Universidade Politécnica da Catalunha, Espanha, em cooperação com os professores Mateo Valero e Eduard Ayguadé. O primeiro passo para esta implementação foi o desenvolvimento intermediário de um simulador para uma arquitetura de multi-processador, o qual foi feito através da replicação de todas as estruturas, onde cada réplica implementa um processador igual ao *sim-outorder* e pode ser indexada por um número diferente (identificador do processador).

Com isso, todas as variáveis simples foram transformadas em variáveis vetoriais e todas as vetoriais em matriciais, e assim por diante. Também, todas as funções foram redefinidas para aceitar como parâmetro o identificador do processador que deve estar sendo manipulado em determinada chamada de função. Cada processador é dedicado a executar apenas uma aplicação. Em cada ciclo, todos os processadores executam as suas aplicações em paralelo. Os resultados individuais de cada aplicação são os mesmos que aqueles reportados pelo *sim-outorder* original. A execução completa do multi-processador se comporta como se vários processadores *sim-outorder* fossem executados paralelamente. O código básico do multi-processador é visto na figura 6.2.

```
Para ( ; ; )   /* cada iteração representa 1 ciclo */
{   n_ciclos++;
    Para (proc = 1 ; proc < = n_procs ; proc++)
    {   Conclusão (proc, nro_inst, acabou);
        Se (acabou) Quebra_Para;
        Retorno (proc);
        Remessa (proc);
        Decodificação (proc);
        Busca (proc);
        total_inst[proc] = total_inst[proc] + nro_inst;
    }
    Se (acabou) Quebra_Para;
}
Para (proc = 1 ; proc < = n_procs ; proc++)
    ipc[proc] = (total_inst[proc] / n_ciclos);
```

FIGURA 6.2 - Código Simplificado do Simulador de Multiprocessador

Após a implementação do simulador de multi-processador, todos os estágios do *pipeline* foram unificados e muitos recursos passaram a ser compartilhados, criando assim um simulador SMT, conforme mostra a figura 6.3. Observe que, neste trabalho, cada tarefa corresponde a uma aplicação diferente. Cada conjunto de recursos incluindo banco de registradores, tabelas e filas e usado para manter o contexto de uma tarefa é chamado *slot*. O primeiro estágio deste novo *pipeline* agora busca um bloco de instruções por ciclo, de apenas uma das tarefas, alternando para uma outra tarefa a cada novo ciclo, em *round-robin*. O tamanho do bloco buscado é limitado à largura do barramento de busca. Os outros estágios também manipulam um bloco de instruções por ciclo, mas contendo uma mistura de instruções provenientes de diferentes tarefas simultaneamente. Em cada estágio, o tamanho deste bloco misto limita-se à largura do barramento correspondente. Os blocos mistos são construídos escalonando uma instrução de cada tarefa, em *round-robin*, até que seja suficiente para preencher o barramento correspondente ou não haja mais instruções para serem escalonadas.

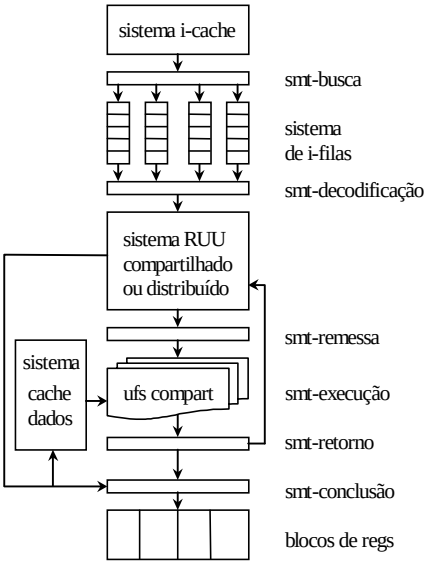


FIGURA 6.3 - Visão Geral da Arquitetura SMT Básica

Existe uma fila de instrução (i-fila) para cada *slot* para garantir que cada tarefa tenha suas instruções buscadas e também para facilitar a mistura de instruções dentro do *pipeline*. O primeiro estágio busca instruções da *cache* nível 1, priorizando as tarefas com menos instruções no *pipeline*. Esta técnica, chamada de ICOUNT, provê melhor desempenho segundo Tullsen [TUL 96]. Das filas de instruções, um conjunto de instruções é decodificado e despachado em ordem, para as estações de reserva (ruu-q e ls-q). Das estações de reserva, as instruções são remetidas para um conjunto único de unidades funcionais compartilhadas. Com relação aos registradores, cada *slot* possui um banco privado para armazenar o contexto de uma tarefa.

Muitas características foram herdadas do simulador *sim-outorder* original, tais como execução fora-de-ordem e especulativa, previsão de desvios e renomeação de registradores. Contudo, novas características foram adicionadas. A primeira delas é a capacidade de utilização de diferentes organizações para as estações de reserva. Duas diferentes topologias podem ser configuradas, conforme mostra a figura 6.4: (a) topologia distribuída por tarefa e (b) topologia compartilhada. Nesta figura, ambas

topologias recebem um bloco misto de instruções do estágio de decodificação em uma arquitetura SMT-4 (4 *slots*/4 tarefas).

Em 1995, Jourdan [JOU 95a] avaliou o desempenho de diferentes topologias de remessa em arquiteturas superescalares, conforme consta na seção 2.2.3.1. Neste estudo verificou-se que a topologia compartilhada poderia tirar melhor proveito das estações de reserva do que a topologia distribuída, apesar de que, Palacharla [PAL 97] mostrou que a complexidade da lógica para a manipulação de estações de reserva compartilhadas pode ser bem maior do que aquela necessária para uma topologia distribuída. Em seu trabalho, Palacharla concluiu que a lógica de remessa em processadores superescalares poderia se tornar um grande gargalo no futuro.

Contudo, devido à natureza mais dinâmica das arquiteturas SMT, em função da existência de várias tarefas executando simultaneamente, e dependendo da exatidão na previsão dos desvios e da taxa de faltas na *cache*, além de outros fatores tais como a topologia da *cache* e os tipos das unidades funcionais, um novo estudo sobre a topologia de remessa em arquiteturas SMT se faz necessário. A existência desta facilidade no simulador SMT aqui apresentado permitiu fazer esta análise conforme mostra a seção 6.2.1, visando obter informações que possibilitem desenvolver uma lógica de remessa mais simplificada possível.

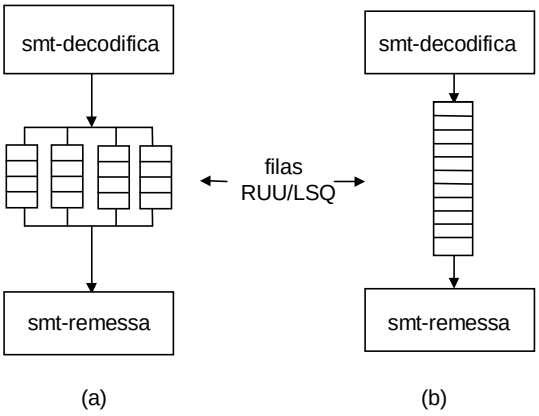


FIGURA 6.4 - Topologias de Remessa em Arquiteturas SMT

A segunda nova característica deste simulador é a configuração da profundidade de decodificação, que define o número de instruções de cada fila de instruções que pode ser inspecionado simultaneamente a cada ciclo. Se cada fila de instrução possui *n* entradas, então *n* é a máxima profundidade de decodificação. Note que as instruções inspecionadas não são necessariamente decodificadas e/ou despachadas.

Os estudos anteriores têm considerado que qualquer entrada das filas de instruções pode ser inspecionada e, se possível, decodificada para preencher o respectivo barramento. Desta forma, o decodificador pode fazer uma melhor escolha de quais instruções despachar. Em contra-partida, esta capacidade exige uma lógica de decodificação mais complexa e o tempo de ciclo é maior. Se for possível reduzir o número de instruções inspecionadas de cada *slot*, sem significante prejuízo no desempenho final, o decodificador pode ser simplificado e o tempo de ciclo diminuído. A figura 6.5 mostra 2 exemplos de *pipeline*s SMT de 4 *slots* com profundidades de decodificação 1 e 2 (sub-figuras (a) e (b), respectivamente). Nesta figura, o *pipeline* da sub-figura (a) pode

decodificar no máximo uma instrução de cada fila de instrução a cada ciclo, enquanto que o *pipeline* da sub-figura (b) pode decodificar até 2 instruções de cada fila de instrução a cada ciclo. Assim, o primeiro *pipeline* pode inspecionar até 4 instruções, para decodificar e despachar até 4 instruções. O segundo *pipeline* pode inspecionar até 8 instruções, para decodificar e despachar até 4 instruções.

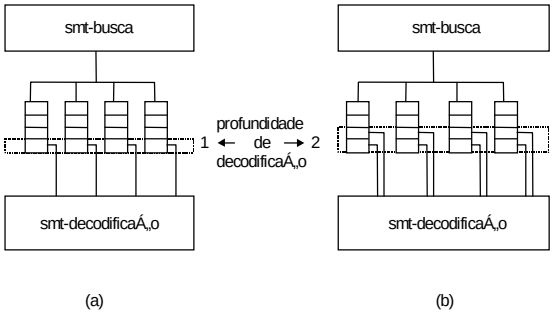


FIGURA 6.5 - Exemplos da Profundidade de Decodificação

Além disso, se a profundidade de decodificação é pequena, o barramento de despacho pode não ser preenchido completamente devido à quantidade insuficiente de instruções inspecionadas por ciclo. Esta situação pode ocorrer ainda que existam outras instruções nas filas de instruções. Devido à sua importância, a profundidade de decodificação foi avaliada, visando obter informações que possibilitem o desenvolvimento de uma lógica de decodificação mais simplificada possível, conforme mostra a seção 6.2.2.

A terceira nova característica deste simulador é permitir a definição de diferentes hierarquias de memória, podendo utilizar vários módulos sobre barramentos independentes, e vários bancos multiplexados sobre o mesmo barramento dentro de um mesmo módulo, como exemplificado na figura 6.6 para uma arquitetura SMT de 4 *slots*. Nesta figura existem 2 módulos de *cache* de instruções que podem ser acessados paralelamente. Cada um serve 2 filas de instruções distintas. Dentro de cada módulo existem 2 bancos que podem ser acessados exclusivamente. Cada banco pode ser usado por mais de uma tarefa, entretanto, cada tarefa deve ser carregada em um único banco somente. O compartilhamento de um mesmo banco de *cache* é permitido porque um campo que identifica a tarefa proprietária foi incluído no bloco da *cache*. A escolha de uma melhor topologia de *cache* pode ser fundamental quando se deseja aumentar o desempenho, e por isso foi avaliada, conforme mostra a seção 6.2.3.

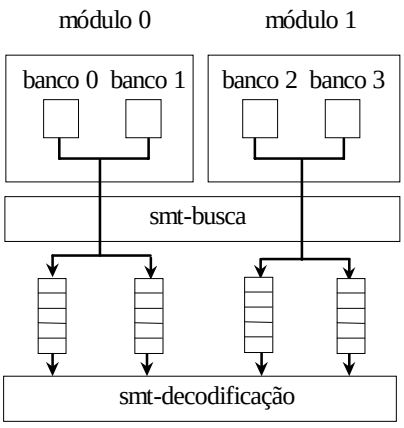


FIGURA 6.6 - Exemplos de Topologias de Cache de Instruções

A última nova característica do simulador SMT é a de permitir o controle da taxa de acerto (exatidão) da previsão de desvios. Este mecanismo é chamado de previsão forçada e sua existência se deve a seguinte justificativa. Em alguns casos, uma arquitetura SMT com poucas tarefas necessita de um bom previsor para explorar eficientemente o paralelismo no nível de instrução e se aprofundar na especulação de cada tarefa. Por outro lado, havendo um maior número de tarefas no sistema, a necessidade de especular profundamente cada tarefa diminui, aliviando assim a complexidade do previsor. Estas questões foram avaliadas usando esta facilidade do simulador SMT em controlar a eficiência do previsor. Diferentes taxas de acerto foram forçadas. Os resultados são mostrados na seção 6.24. Este previsor funciona como segue.

Através de parâmetros de configuração, o usuário define a taxa de acerto que a previsão deve seguir durante a execução. Assim, quando uma instrução de desvio é alcançada no estágio de decodificação, o mecanismo de previsão forçada sorteia aleatoriamente um número randômico de 1 a 100. Se este número é menor que a taxa de acerto pretendida, o mecanismo de previsão força uma previsão correta, caso contrário, é forçada uma previsão errada. Este mecanismo foi introduzido no estágio de decodificação, pois na fase de busca ainda não é possível conhecer o endereço alvo correto que possa garantir a previsão correta. Deve-se notar também que este mecanismo de previsão está relacionado somente com o acerto ou erro na direção do desvio, não sendo considerado o acerto ou erro no endereço alvo. Esse por sua vez é sempre conhecido, como se existisse uma tabela infinita de endereços previstos. As próximas seções apresentam os resultados das diferentes características da arquitetura SMT básica que foram avaliadas.

6.2.1 Avaliação e Desempenho da Topologia de Remessa

Antes de serem descritas as simulações nesta seção, algumas definições e alguns valores que são utilizados no presente trabalho de doutorado precisam ser esclarecidas. Primeiro: uma arquitetura SMT-*x* é definida como sendo uma arquitetura de *x slots* com capacidade de executar *x* programas simultaneamente. Segundo: os desempenhos são medidos em *ipc* (instruções por ciclo) e o *ganho* (*speedup*) é definido como a diferença percentual entre dois valores de *ipc*. Terceiro: todas as simulações foram executadas até o primeiro programa completar 250 milhões de instruções, cujas 50 milhões de instruções iniciais foram descartadas para remover a instabilidade inicial da execução. Por último, as principais latências arquiteturais, bem como os 5 tipos de unidades funcionais básicas, são definidas de acordo com a tabela 6.1, sendo as mesmas comumente utilizadas pelos simuladores do *SimpleScalar*.

As simulações aqui apresentadas consideram arquiteturas SMT-4 e SMT-8 devido a duas razões. A primeira é que os trabalhos anteriores já mostraram que o ganho de desempenho é satisfatório para este número de tarefas, se comparado com a quantidade de *hardware* utilizado. E a segunda é que muitas pessoas acreditam que a tecnologia atual para o projeto de arquiteturas de computadores é suficiente para permitir o desenvolvimento de um processador real com estas configurações.

Foram usados 8 programas de avaliação do SPEC95 para medir o desempenho: 4 de inteiros (*perl*, *ijpeg*, *compress* e *li)* e 4 de ponto-flutuantes (*swim*, *mgrid*, *wave5* e

fpppp). Para calcular os desempenhos para a arquitetura SMT-4 e visando diluir as diferenças no desempenho existente entre os diferentes programas, foram executadas 4 combinações de 4 programas cada uma (2 de inteiros e 2 de ponto-flutuante). Para a arquitetura SMT-8, foi executada apenas uma combinação. A tabela 6.2 mostra todas as combinações consideradas.

TABELA 6.1 - Latências do *Pipeline* SMT

Tipos de latências	**Número de ciclos**	
l1 hit ; l2 hit ; tlb miss	1 ; 6 ; 30	
l2 miss (para n+1 blocos)	(18 + n*2)	
unidade funcional int-alu	1	
unidade funcional fp-alu	2	
unidade funcional ld/st	1	
unidade funcional int-mul	Div	20
	Mult	3
unidade funcional fp-mult	Sqrt	24
	Div	12
	Mult	4

TABELA 6.2 - Grupos de Programas de Avaliação

SMT-4	1	*swim, perl, mgrid, ijpeg*
	2	*wave5, compress, fpppp, li*
	3	*li, fpppp, ijpeg, mgrid*
	4	*compress, wave5, perl, swim*
SMT-8	*swim, perl, mgrid, ijpeg, wave5, compress, fpppp, li*	

Duas medidas para a quantidade total de *hardware* foram consideradas, conforme mostrado na tabela 6.3. *W* indica a largura do *pipeline*, *itlb*/*dtlb* indicam as dimensões das tabelas TLBs de instruções e dados, *il1*/*dl1* indicam as dimensões das *caches* níveis 1 de instruções e dados, *ul2* indica a dimensão da *cache* de nível 2 unificada, fus indica os tipos/quantidades das unidades funcionais e *ruu/lsq sizes* indicam os números de entradas nas filas ruu e ls-q. O *hardware* básico define uma quantidade mínima necessária para explorar pelo menos uma instrução de cada tarefa a cada ciclo. O *hardware* avançado define o dobro da quantidade mínima, visando explorar um pouco mais de execução especulativa dentro de cada tarefa. Ambas medidas são modestas em termos de arquiteturas SMT, visando sempre uma implementação real. O barramento do *pipeline* possui sempre a mesma largura em todos os estágios.

TABELA 6.3 - Quantidade de *Hardware*

hardware	**SMT-4**	**SMT-8**
básico	w=4, itlb=64k, dtlb=128k ul2=(128k, a=4), il1=dl1=(16k, a=4) 7 fus =(2 int-alu, 2 fp-alu, 1 int-mult, 1 fp-mult, 1 ld/st) ruu size = 16 / lsq size = 8	w=8, itlb=128k, dtlb=256k ul2=(256k,a=8), il1=dl =(32k, a=8) 14 fus = (4 int-alu, 4 fp-alu, 2 int-mult, 2 fp-mult, 2 ld/st) ruu size = 32 / lsq size = 16
avançado	w=8, itlb=128k, dtlb=256k ul2=(256k, a=4), il1=dl1=(32k, a=4) 14 fus = (4 int-alu, 4 fp-alu, 2 int-mult, 2 fp-mult, 2 ld/st) ruu size = 32 / lsq size = 16	w=16, itlb=256k, dtlb=512k ul2=(512k, a=8), il1=(2x32k, a=8) dl1=(64k, a=8) 28 fus = (8 int-alu, 8 fp-alu, 4 int-mult, 4 fp-mult, 4 ld/st) ruu size = 64 / lsq size = 32

Os dois tipos de topologias descritas na seção 6.2, conforme mostra a figura 6.7 de forma mais completa, foram avaliados. Observe que, em relação às unidades funcionais,

ambas topologias compartilhada e distribuída permanecem compartilhadas, ou seja, qualquer estação de reserva independente da topologia aqui analisada é compartilhada por qualquer unidade funcional. Ressaltamos que o compartilhamento e a distribuição referidos aqui são relativos às tarefas e não às unidades funcionais (vide seção 2.2.3.1).

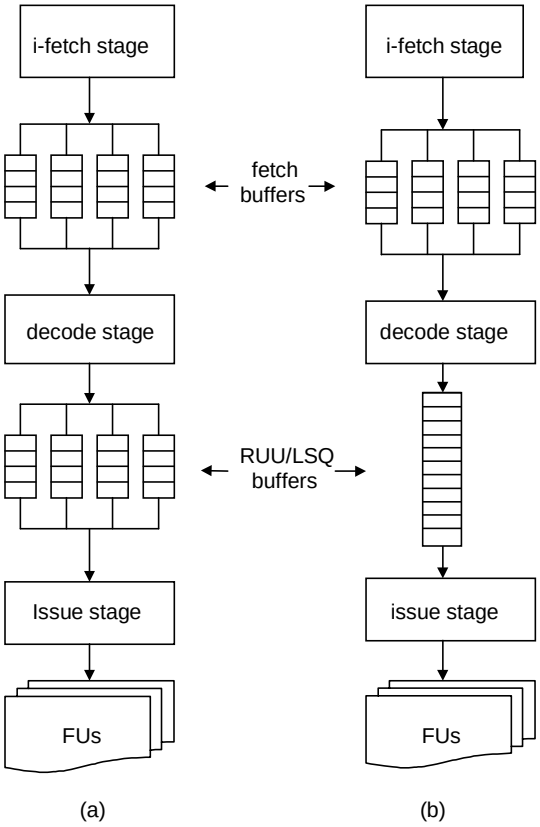


FIGURA 6.7 - Tipos de Topologias Analisadas

Nestas simulações, as quantidades de estações de reserva e de registradores de renomeação são mantidas iguais em ambas as topologias. Usando o *hardware* básico, o desempenho individual de cada programa de avaliação é similar em ambas arquiteturas SMT-4 e SMT-8. Estes resultados, bem como o desempenho global, podem ser vistos nas figuras 6.8 e 6.9. O desempenho da topologia distribuída é melhor e a tabela 6.4 mostra que o ganho não é desprezível (8.04% para SMT-4 e 7.26% para SMT-8).

Estes resultados parecem peculiares porque se imagina que a topologia compartilhada deve favorecer as tarefas com mais paralelismo e assim aumentar o desempenho global, tal como acontece em arquiteturas superescalares [JOU 95a]. Entretanto, é necessário considerar a justificativa que se segue.

Quando uma arquitetura SMT usa uma técnica balanceada para escalonar instruções de várias tarefas, tal como a técnica de *round-robin* aqui utilizada, esta tenta garantir uma quantidade similar de instruções decodificadas para cada tarefa, independentemente do grau de paralelismo no nível de instrução de cada uma. Além disso, quando o *pipeline* despacha instruções não-prontas, todas as tarefas têm possibilidades equivalentes de despachar instruções, a menos que não existam instruções nas filas de instruções.

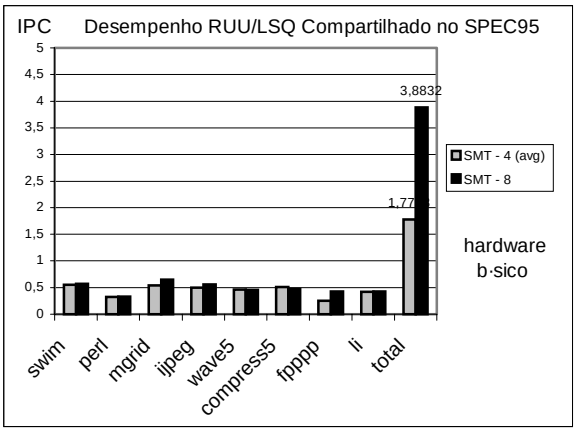


FIGURA 6.8 - Desempenho da Topologia Básica Compartilhada

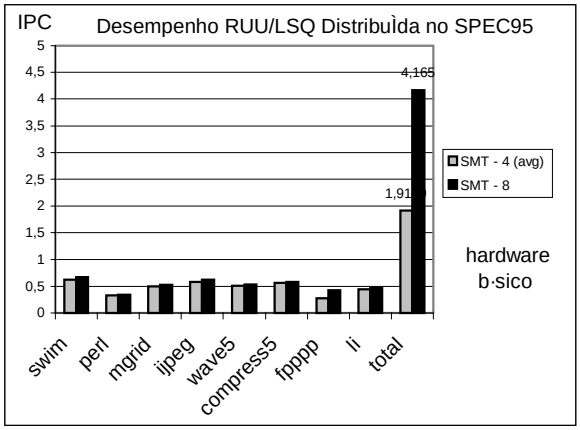


FIGURA 6.9 - Desempenho do Topologia Básica Distribuída

TABELA 6.4 - Resultados da Topologia Básica

Desempenho e Ganho (distribuída / compartilhada)				
tipo	**SMT - 4**		**SMT - 8**	
topologia	**dist**	**comp**	**distr**	**comp**
ipc	1,92	1,77	4,17	3,88
Ganho	8,04%		7,26%	

Existem duas situações que podem permitir um melhor desempenho para o compartilhamento das estações de reserva em arquiteturas SMT. A primeira delas ocorre se o *pipeline* despachar somente instruções prontas. A segunda ocorre se as tarefas tiverem, entre si, consideráveis diferenças entre as taxas de faltas na *cache* e/ou de previsões incorretas de desvios, o que não é perceptível quando se executam várias combinações de programas e calcula-se a média. Em ambas situações, algumas tarefas podem despachar mais instruções do que as outras e a topologia compartilhada deve ser mais bem utilizada.

Outras simulações, usando o *hardware* avançado, foram feitas. Devido ao fato de que o tamanho médio dos blocos básicos dos programas utilizados é pequeno, a largura do barramento de busca foi quebrada em duas partes na arquitetura SMT-8 para permitir a utilização de 2 módulos independentes de *cache* e prover melhor desempenho. Não é justificável buscar 16 instruções de uma única tarefa sendo que o fluxo pode ser

quebrado logo após as primeiras instruções, além de que se deve considerar também que a unidade de busca está capacitada para prever um único desvio. Assim, a arquitetura busca 2 blocos de 8 instruções de duas diferentes tarefas simultaneamente. As figuras 6.10 e 6.11 mostram os desempenhos individuais de cada programa de avaliação bem como o desempenho global. A tabela 6.5 apresenta os respectivos ganhos de cada arquitetura SMT.

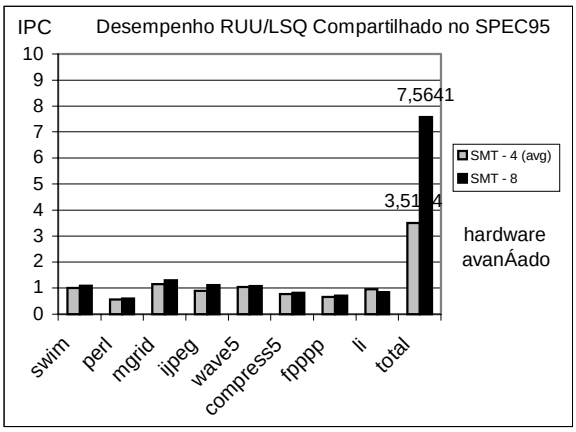


FIGURA 6.10 - Desempenho da Topologia Avançada Compartilhada

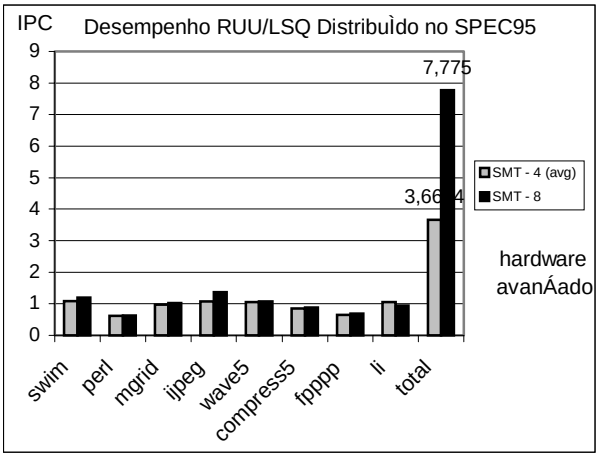


FIGURA 6.11 - Desempenho da Topologia Avançada Distribuída

TABELA 6.5 - Resultados da Topologia Avançada

Desempenho e ganho (distribuída / compartilhada)				
tipos	**SMT - 4**		**SMT - 8**	
topologia	**dist**	**comp**	**distr**	**comp**
ipc	3,67	3,51	7,78	7,56
Ganho	4,41%		2,79%	

O desempenho da topologia distribuída mantém-se melhor e o comportamento individual de cada programa de avaliação é similar àquele obtido usando o *hardware* básico. Entretanto, o ganho (distribuído sobre o compartilhado) para ambas arquiteturas SMT-4 e SMT-8 sofreu decréscimo de 8,04% para 4,41% e de 7,26% para 2,79%, respectivamente. Contudo, estes resultados são compreensíveis.

As diferenças nos desempenhos entre as duas topologias tendem a ser menores quando o *hardware* aumenta em uma proporção maior do que o paralelismo entre as tarefas. Esta situação garante oportunidades de execução para todas as tarefas, independente da topologia de remessa utilizada. De fato, se o *hardware* fosse ilimitado e se o tempo de ciclo não fosse um problema, não existiriam diferenças entre estes desempenhos.

6.2.2 Avaliação e Desempenho da Profundidade de Decodificação

A análise da profundidade de decodificação, conforme explicado na seção 6.2, pode contribuir no projeto de um estágio de decodificação mais simples e que minimize o tempo de ciclo, e por isto ela é apresentada aqui. A figura 6.5 exemplifica este conceito.

Com mencionado anteriormente, o decodificador escalona instruções usando uma política *round-robin* para investigar as diferentes filas de instruções, escalonando uma instrução por tarefa no primeiro nível de profundidade. Contudo, se o barramento de despacho ainda não está completo, o decodificador pode continuar escalonando instruções em outros ciclos do *round-robin* em níveis mais profundos até um limite pré-estabelecido na configuração.

A implementação deste conceito em *hardware* requer a utilização de registradores de deslocamento nas entradas das filas de instruções, a fim de permitir que todas as instruções dentro delas possam ser decodificadas. Neste trabalho, foram analisados diferentes limites para a profundidade de decodificação, para as topologias distribuída e compartilhada, em arquiteturas SMT-4 e SMT-8. Estas simulações também se baseiam nas informações contidas nas tabelas 6.1, 6.2 e 6.3 da seção anterior. Adicionalmente, tem-se que a arquitetura SMT-4 usa 4 filas de instruções de 4 entradas cada uma e a SMT-8 usa 8 filas de instruções de 8 entradas cada uma. A tabela 6.6 mostra os resultados das simulações, usando o *hardware* básico.

TABELA 6.6 - Desempenho *Hardware* Básico x Prof. de Decodificação

Efeito da profundidade de decodificação no desempenho (ipc) do *hardware* básico.

Profund	**RUU/LSQ Distr**		**RUU/LSQ comp**	
decodif	**SMT - 4**	**SMT - 8**	**SMT - 4**	**SMT - 8**
1	1,85	4,06	1,79	3,90
2	1,91	4,16	1,78	3,89
3	1,91	4,16	1,78	3,89
Max	1,92	4,17	1,77	3,88

Considerando a topologia distribuída, é possível ver que, com profundidade de decodificação de 2, a arquitetura SMT-4 obtém 99,5% do máximo desempenho, que é alcançado com profundidade de decodificação máxima (4, neste caso). Isto significa um prejuízo de somente 0,5% no desempenho final, mas uma redução de 50% sobre a lógica de decodificação, pois somente 2 entradas de cada fila de instruções, de um total de 4, precisam ser inspecionadas. Para a arquitetura SMT-8, o prejuízo no desempenho é de somente 0,2% e a redução da lógica de decodificação é de 75%, pois somente 2 entradas, de um total de 8, precisam ser inspecionadas.

Considerando a topologia compartilhada, os resultados parecem peculiares, pois com profundidade de decodificação menor, o desempenho é maior do que com profundidade de decodificação maior. Contudo, esta situação ocorre devido ao comportamento

imprevisível de uma computação SMT. De fato, foi verificado nas simulações realizadas neste trabalho que uma pequena redução no desempenho de uma tarefa pode permitir um aumento maior no desempenho de outra tarefa.

Conseqüentemente, a execução de uma pequena quantidade de instruções de alta latência, provenientes de algumas das tarefas, pode proporcionar a execução de uma quantidade maior de instruções de pequena latência, provenientes de outras tarefas. Assim, quando existe um limite para a profundidade de decodificação, a interferência entre instruções é reduzida pois cada tarefa tem mais oportunidades de ser executada. Esta característica favorece a topologia compartilhada que obtém a melhor performance, naturalmente, usando uma lógica de decodificação mínima.

Este fato não ocorre quando a topologia distribuída é usada, pois cada tarefa tem sua janela de instruções privada e não existem interferências entre elas nas estações de reserva. Além disso, em todos os casos, o desempenho da topologia distribuída se mostrou melhor do que o desempenho da topologia compartilhada.

Também, foram realizadas simulações usando o *hardware* avançado e os resultados são mostrados na tabela 6.7. Nesta tabela pode-se ver que a topologia distribuída apresenta um comportamento similar àquele obtido com o *hardware* básico. Quando a profundidade de decodificação é 2, o prejuízo do desempenho, com relação ao melhor desempenho, é de 3,5% para a arquitetura SMT-4 e de 1,8% para a arquitetura SMT-8.

TABELA 6.7 - Desempenho *Hardware* Avançado x Prof. de Decodificação

Efeito da profundidade de decodificação no desempenho (ipc) do *hardware* avançado.

profund	RUU/LSQ distr		RUU/LSQ comp	
decodif	SMT - 4	SMT - 8	SMT - 4	SMT - 8
1	3,04	6,18	3,16	6,59
2	3,54	7,64	3,53	7,58
3	3,66	7,74	3,52	7,57
Max	3,67	7,78	3,51	7,56

Contudo, ambas arquiteturas com topologias compartilhadas obtêm um prejuízo de desempenho quando a profundidade de decodificação é 1. Isto permite concluir que, neste caso, os recursos de *hardware* são suficientes para não serem usados satisfatoriamente se for permitida a inspeção de somente uma instrução de cada fila de instruções.

6.2.3 Avaliação e Desempenho da Topologia da *Cache* de Instruções

Um dos principais problemas relacionados às arquiteturas SMT é a perda de desempenho da *cache* de instruções devido aos conflitos de endereços de memória entre as várias tarefas [GON 99]. A topologia com que a *cache* é estruturada pode ser oportuna para minimizar este prejuízo. Visando entender melhor esta situação, neste trabalho foram avaliadas diversas topologias para duas configurações básicas de processadores SMT, capazes de executar 4 e 8 tarefas simultaneamente [GON 00]. Todas as variações topológicas para a mesma arquitetura usam a mesma quantidade de *hardware*, conforme mostra a tabela 6.8. As latências e os tipos de unidades funcionais dos processadores são as mesmas da tabela 6.1

TABELA 6.8 - Quantidade de *Hardware*

SMT-4	**SMT-8**
Pipeline width = 8, Unif-l2-cache = 128k, Instr-l1-cache = 16k, Data-l1-cache = 16k, 7 Funct-units = (2 int-alu, 2 fp-alu, 1 int-mult, 1 fp-mult, 1 ld/st), (ruu, lsq) sizes = (16, 8) entries.	*Pipeline* width = 16, Unif-l2-cache = 256k, Instr-l1-cache = 32k, Data-l1-cache = 32k, 14 Funct-units = (4 int-alu, 4 fp-alu, 2 int-mult, 2 fp-mult, 2 ld/st), (ruu, lsq) sizes = (32, 16) entries.

As topologias consideradas são chamadas de *Cachexyz*, onde *x* é a modularidade da *cache*, que define o número de módulos conectados em barramentos diferentes e que podem ser acessados paralelamente; *y* é a separatividade da *cache*, que define o número total de bancos multiplexados distribuídos igualmente entre os módulos e *z* é associatividade da *cache*, que define o número de entradas de cada banco da *cache* associada ao mesmo endereço. O produto *yz* (separatividade vezes associatividade) é chamado *evt* (espaço vetorial de tarefas). Visando prover suficiente espaço de memória que permita a co-existência de várias tarefas na *cache*, o *etv* deve ser maior do que o número de tarefas que a compartilham. Também, o número máximo de tarefas alocadas dentro do mesmo módulo de *cache* deve ser *evt/x*.

Dentro de um módulo, a multiplexação dos bancos simplifica a complexidade do *hardware* externo, tornando possível o uso de somente um barramento de busca, embora força o acesso exclusivo de somente um banco por vez. As figuras 6.12 e 6.13 mostram as topologias da *cache* simuladas neste trabalho, para as arquiteturas SMT-4 e SMT-8, respectivamente. Também, quando a *cache* é estruturada em 2 módulos, o barramento de busca é dividido em 2 partes. Os programas utilizados são os mesmos das simulações anteriores, com exceção do programa *compress* que foi substituído pelo *gcc*, conforme mostra as combinações da tabela 6.9.

TABELA 6.9 - Grupos de Programas de Avaliação

SMT-4	1	*swim, perl, mgrid, ijpeg*
	2	*wave5, gcc, fpppp, li*
	3	*li, fpppp, ijpeg, mgrid*
	4	gcc, *wave5, perl, swim*
SMT-8	*swim, perl, mgrid, ijpeg, wave5, gcc, fpppp, li*	

A figura 6.14 mostra o desempenho obtido pela arquitetura SMT-4 para as topologias de *caches* previamente definidas. Nesta figura, é possível ver que o programa *ijpeg* obtém o melhor desempenho individual (mais que 1 ipc) enquanto o *perl* o pior (próximo de 0.5 ipc). Note também que o desempenho global alcança mais de 3 *ipc* para todas as topologias. Além disso, pode-se observar que a mudança de topologia não causa significante diferença nos desempenhos individuais de cada programa de avaliação. Agora, em relação ao desempenho global, esta diferença é maior, como mostra a tabela 6.10. Esta tabela mostra que o *ganho* da topologia *Cache114* (melhor desempenho global) sobre a topologia *Cache241* (pior desempenho global) alcança 9.48%. Além disso, nesta mesma figura, pode-se fazer duas considerações sobre o *evt*. Primeiro: a associatividade é mais importante do que a separatividade quando se deseja melhor desempenho. Segundo: a modularidade 2 não provoca melhoria no desempenho.

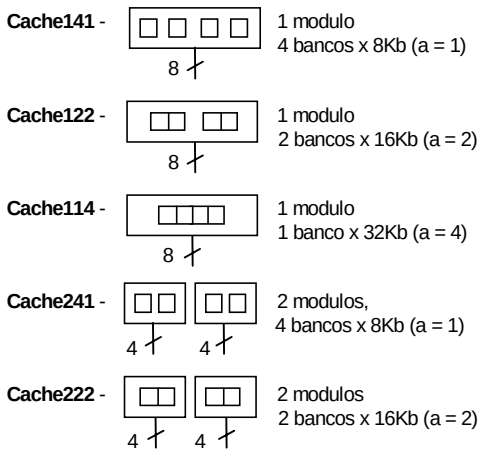


FIGURA 6.12 - Topologias de *Cache* de Instruções para a SMT-4

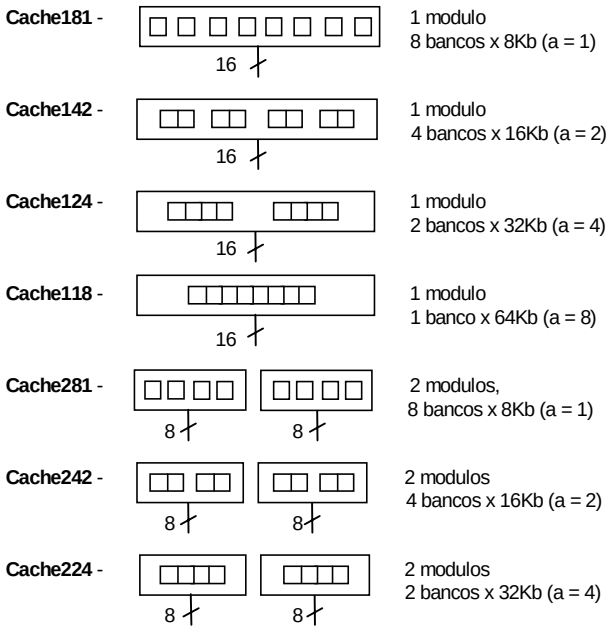


FIGURA 6.13 - Topologias de *cache* de Instruções para a SMT-8

A figura 6.15 mostra os resultados obtidos pela arquitetura SMT-8. O desempenho global é muito maior do que aquele obtido pela arquitetura SMT-4, alcançando mais que 7 *ipc* nas topologias *Cache242* e *Cache224*. O melhor *ganho* foi obtido pela topologia *Cache224* sobre a topologia *Cache181,* alcançando 34.74%, conforme mostra a tabela 6.10. Contudo, o desempenho médio entre todas as topologias para cada programa é similar em ambas arquiteturas SMT-4 e SMT-8, conforme mostra a figura 6.16. Esta situação ocorre porque os estágios do *pipeline* SMT aqui apresentado são baseados no algoritmo *round-robin*, que dá prioridade maior para o paralelismo entre tarefas do que para o paralelismo dentro das tarefas (no nível de instrução). A figura 6.16 também mostra, junto aos topos das barras, os *ganhos* no desempenho médio de uma arquitetura sobre a outra (do maior sobre o menor), para cada programa de avaliação. Por exemplo, os *ganhos* são de 1.62%, 1.31% e 0.34%, para os programas *perl*, *mgrid* e *ijpeg*, respectivamente, e o máximo ganho foi alcançado pelo programa *swim*, em torno de 10.70% para a arquitetura SMT-4. Os melhores ganhos foram obtidos pela SMT-4 sobre

a SMT-8 com os programas de ponto-flutuante *wave*, *fpppp* e *swim*, alcançando de 8% a 11%.

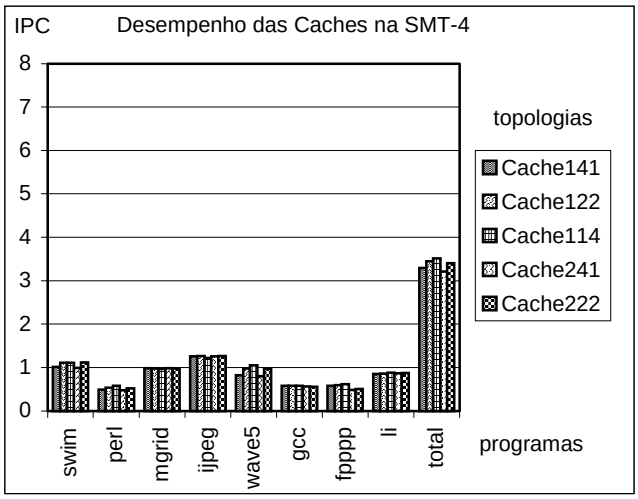


FIGURA 6.14 - Desempenho das Topologias de *Cache* na SMT-4

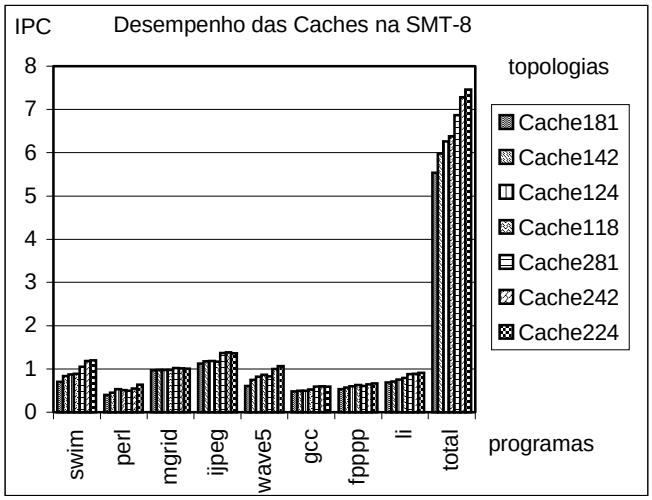


FIGURA 6.15 - Desempenho das Topologias de *Cache* na SMT-8

Estas simulações permitem verificar que a associatividade é mais importante do que a separatividade, quando se deseja melhor desempenho, assim como ocorre nas arquiteturas SMT-4. Este fato ocorre porque a separatividade não permite o uso dos bancos por qualquer tarefa, em oposto à associatividade. Contudo, contrariamente à arquitetura SMT-4, a utilização de modularidade 2 em arquiteturas SMT-8 provê melhor desempenho. Esta situação ocorre devido à largura do barramento de busca. Na arquitetura SMT-4, a largura de busca de 4 instruções não é suficiente para explorar o *ilp* (paralelismo no nível de instrução) disponível dentro de cada tarefa, devido à quebra dos blocos básicos. Este fato não ocorre na arquitetura SMT-8 pois a largura de 8 instruções garante uma maior porcentagem de aproveitamento dos blocos básicos durante a busca, que tendem a ser acessados de forma mais completa.

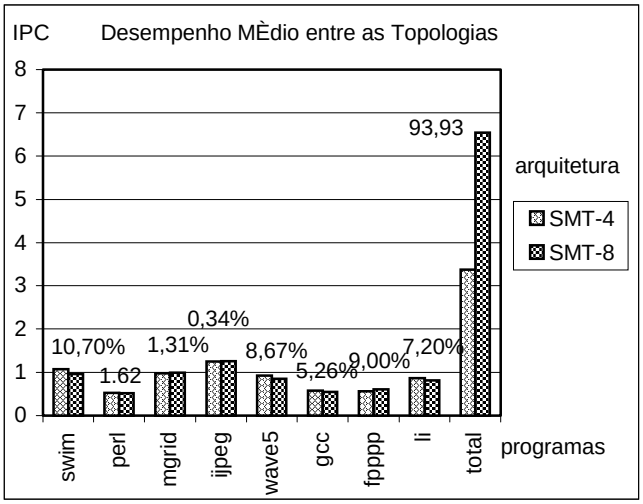


FIGURA 6.16 - Desempenho Médio entre as Topologias

TABELA 6.10 - Máximo Ganho de Desempenho

SMT-4	**SMT-8**
Cache114/Cache241	Cache224/Cache181
9,48%	34,74%

Também, a relação entre desempenho e complexidade, quando se avalia a associatividade, separatividade e modularidade, pode ser discutida. Alta associatividade é muito cara para implementar em termos de ciclo de processador, pois o sistema de *cache* deve ser capaz de investigar um grande número de posições a cada acesso na tentativa de encontrar o endereço desejado. Contudo, esta técnica pode reduzir conflitos e prover um melhor aproveitamento da *cache*. Alternativamente, a separatividade provê rápido acesso ao banco alvo através da multiplexação e reduz conflitos também. Entretanto, os bancos sub-utilizados por algumas tarefas não podem ser aproveitados por outras tarefas.

Em relação a modularidade, a utilização de mais do que um módulo de *cache* requer a divisão do estágio de busca em dois sub-estágios menores. Além disso, as filas de instruções sub-utilizadas por um módulo não podem ser usadas por outros módulos. Contudo, devido à simplicidade de cada sub-estágio e à ausência de conflitos entre módulos, esta abordagem poderia ser usada para melhorar o desempenho em alguns casos.

6.2.4 Avaliação e Desempenho da Variação da Certeza da Previsão

Arquiteturas superescalares são usadas em muitos microprocessadores em estado da arte, tais como [AND 95], PowerPC [DIE 95], MIPS R10000 [MIP 95], UltraSparc [ULT 96] e Alpha 21264 [KES 99]. Estas arquiteturas possuem várias unidades funcionais e podem explorar, diretamente pelo *hardware*, o paralelismo no nível de instrução (*ilp*). Já as arquiteturas SMT, que possuem uma maior quantidade de *hardware* para armazenar diferentes contextos, podem explorar, adicionalmente, o paralelismo no nível das tarefas (*tlp*). Infelizmente, estes paralelismos são limitados por várias razões, conforme discutido na seção 2.2, tais como as dependências de controle

[LEE 95] causadas pelas instruções de desvios. Os desvios quebram o fluxo de instruções, e reduzem assim as chances de execução em paralelo. Entretanto, muitas técnicas de previsão de desvios bem sucedidas foram desenvolvidas e usadas em arquiteturas superescalares, provendo altas taxas de acerto. Alguns previsores podem atingir 98% de acerto [MCF 93]

Apesar dessas taxas de acerto serem bastante altas, a ocorrência de desvios mal previstos, mesmo que em pequena quantidade, faz com que o processador execute instruções ao longo do caminho incorreto. Desta forma, o desempenho é reduzido devido à pelo menos 3 motivos. Primeiro, devido à poluição na *cache* de instruções, pois instruções úteis podem ser substituídas por instruções inúteis, provocando futuras faltas na *cache*. Segundo, devido ao barramento da *cache*, pois enquanto uma falta na *cache* no caminho incorreto está sendo processada e ocupando o barramento, uma outra falta no caminho correto pode estar bloqueada a espera de atendimento. Finalmente, devido ao *pipeline* precisar ser esvaziado durante a confirmação da má previsão, fazendo que todo o processamento seguinte ao desvio e que já estava sendo processado se torne inútil.

Atualmente, estas mesmas técnicas são usadas em arquiteturas SMT, mas não existe nenhum estudo que avalie se o seu uso mantém a mesma efetividade ou se as más previsões causam o mesmo impacto como em arquiteturas superescalares. Como em arquiteturas SMT o ganho de desempenho por algumas tarefas pode causar a perda de desempenho pelas outras, o efeito do uso de técnicas de previsão de desvios em arquiteturas SMT deve ser avaliado. Assim, visando desvendar em parte estas questões, neste trabalho é avaliado o efeito da variação da taxa de acerto de previsão em arquiteturas SMT. Além disso, os resultados são comparados com aqueles obtidos em arquiteturas superescalares, com o objetivo de descobrir se ambos comportamentos são similares. Este trabalho foi desenvolvido em conjunto com outros pesquisadores [GON 01].

Para fazer estas simulações, o simulador superescalar *sim-outorder* descrito na seção 6.1 foi adaptado para incorporar o mecanismo de previsão forçada que existe no simulador SMT, descrito na seção 6.2, o qual permite ao usuário definir, em tempo de configuração, a taxa de acerto desejada na previsão de desvios.

Nestas simulações foram utilizados os mesmos 8 programas SPEC95 de acordo com os grupos da tabela 6.9. As latências e unidades funcionais também são as mesmas da tabela 6.1. Novamente, todas as simulações foram executadas até o primeiro programa completar 250 milhões de instruções, sendo as primeiras 50 milhões descartadas devido à instabilidade inicial dos códigos. Os experimentos foram divididos em 6 grupos, usando como critério o tipo da arquitetura simulada (superescalar, SMT-4 ou SMT-8) e o volume dos recursos alocados (pequeno ou grande), como mostra a tabela 6.11. O *hardware* pequeno representa uma possível implementação baseada em padrões atuais do estado da arte, enquanto que o *hardware* grande representa uma previsão no futuro, visando processamento altamente paralelo.

As primeiras simulações foram feitas em arquiteturas superescalares. A certeza do previsor de desvios foi variada e os resultados em *ipc* foram coletados. Note que o previsor foi especialmente projetado para permitir alcançar o grau de certeza desejado independente de nenhum esquema específico de previsão de desvios. De fato, o previsor

é baseado na previsão perfeita, de forma que o resultado correto é sempre conhecido e em determinadas situações, quando se deseja manter a taxa de acerto definida pelo usuário, algumas más-previsões são intencionalmente forçadas.

TABELA 6.11 - Quantidade de *Hardware*

	Superescalar	**SMT-4**	**SMT-8**
pequena	width = 8, l2-cache = 512k, l1-cache = 32k, 10 fus = 3 IA, 3 FA, 2 LS, 1 IM, 1 FM, ruu = 32 entries, lsq = 16 entries.	width = 16, l2-cache = 1M, l1-cache = 64k, 17 fus = 5 IA, 5 FA, 3 LS, 2 IM, 2 FM, ruu = 64 entries, lsq = 32 entries.	width = 16, l2-cache = 1M, l1-cache = 64k, 17 fus = 5 IA, 5 FA, 3 LS, 2 IM, 2 FM, ruu = 64 entries, lsq = 32 entries.
grande	width = 16, l2-cache = 1M, l1-cache = 64k, 17 fus = 5 IA, 5 FA, 3 LS, 2 IM, 2 FM, ruu = 64 entries, lsq = 32 entries.	width = 64, l2-cache = 4M, l1-cache = 256k, 68 fus = 20 IA, 20 FA, 12 LS, 8 IM, 8 FM, ruu = 256 entries, lsq = 128 entries.	width = 128, l2-cache = 8M, l1-cache = 512k, 136 fus = 40 IA, 40 FA, 24 LS, 16 IM, 16 FM, ruu = 512 entries, lsq = 256 entries.

As figuras 6.17 e 6.18 mostram o número de instruções por ciclo de cada programa de avaliação como uma função da taxa de acerto da previsão de desvios. Em ambas arquiteturas pequena e grande os programas apresentam um comportamento similar, apesar do melhor desempenho obtido pela arquitetura grande. Os programas *mgrid* e *fpppp* não são muito afetados pela melhoria da previsão, apresentando desempenhos quase que constantes. Para estes programas, o aumento de desempenho somente foi obtido com o aumento dos recursos de *hardware*, ou seja, da arquitetura pequena para a grande. Já os programas *ijpeg* e *li* apresentam significante aumento de desempenho com o aumento da taxa de acerto da previsão. Isto significa que estes programas são mais sensíveis à qualidade da previsão de desvios.

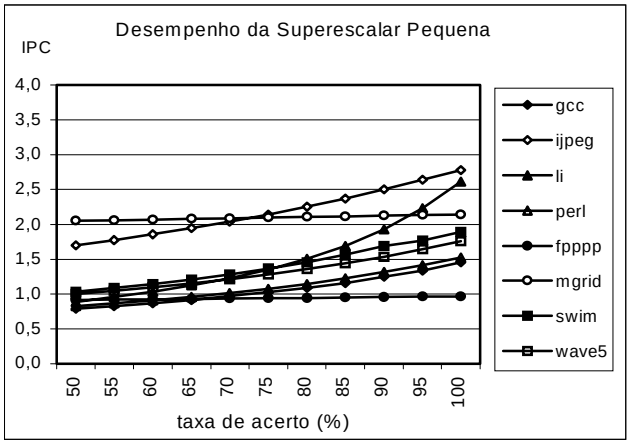


FIGURA 6.17 - Desempenho da Arquitetura Superescalar Pequena

Na arquitetura superescalar pequena, o programa *ijpeg* supera o desempenho do *mgrid* quando a taxa de acerto atinja próximo de 75%, e o programa *li* supera o desempenho do *fpppp* quando a taxa de acerto chega a 55%. Na arquitetura superescalar grande, estes

valores são 85% e 78%, respectivamente. De certa forma, isto significa que com o aumento do *hardware*, os programas mais sensíveis à previsão devem dispor de mecanismos mais eficientes para poderem tirar melhor proveito dos recursos disponíveis. Melhor ainda, pode-se dizer que eles devem errar menos para ocupar mais o *pipeline*, já que este foi aumentado.

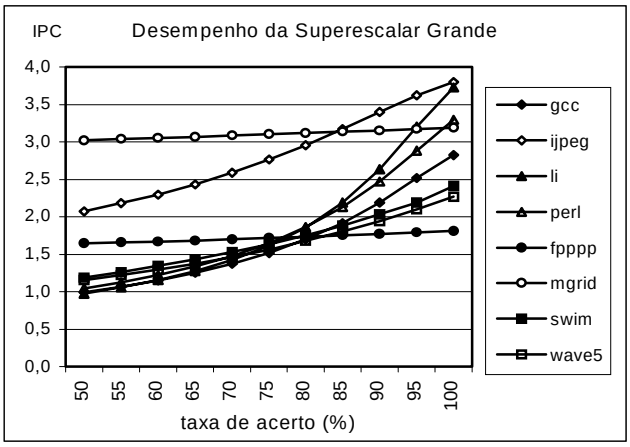


FIGURA 6.18 - Desempenho da Arquitetura Superescalar Grande

A figura 6.19 mostra o desempenho médio em ambas as arquiteturas superescalares. De fato, pode-se notar que, quando a taxa de acerto é melhorada, a arquitetura grande tem seu desempenho aumentado mais intensamente do que na arquitetura pequena. Este fato ocorre porque quando o previsor é mais eficiente, as possibilidades de exploração de mais paralelismo no nível de instrução aumentam devido à redução das quebras dos blocos básicos. Contudo, esta vantagem somente é possível quando os recursos estão disponíveis. Por outro lado, quando o *hardware* aumenta, também aumentam as possibilidades de exploração de mais paralelismo no nível de instrução devido ao melhor aproveitamento das instruções que estão sendo executadas em paralelo durante a decodificação. Assim, embora exista um aumento de desempenho pelo simples aumento da arquitetura, este aumento é mais significativo quando a taxa de acerto é maior.

Uma outra questão importante na figura 6.19 é que na arquitetura superescalar pequena não é possível obter um *ipc* maior que 2, ainda que com previsão perfeita. Contudo, na arquitetura grande, este desempenho pode ser obtido com previsão de 80%. Assim, o esforço que é feito para melhorar o previsor pode ser direcionado para expandir o *hardware*, obtendo assim os mesmos resultados. Na verdade, o custo do aumento do *hardware* ou da complexidade do previsor, levando sempre em consideração o tempo de ciclo, deve ser considerado como fator de peso na determinação do melhor benefício. Entretanto, não se pode esquecer que o desempenho varia em função de vários outros fatores, tais como *caches*, barramentos e portas.

Além disso, a melhoria na taxa de acerto de 95% para 100% causa um aumento de 7% a 9% no desempenho global nas arquiteturas pequena e grande, respectivamente, como mostra a figura 6.20. Nesta figura, o ganho em cada valor de taxa de acerto é calculado através da diferença percentual do desempenho obtido naquele ponto e aquele obtido com a taxa de acerto imediatamente inferior. Por exemplo, o ganho da arquitetura grande com 85% de taxa de acerto, sobre a mesma arquitetura com taxa de acerto de 80%, é em torno de 8%.

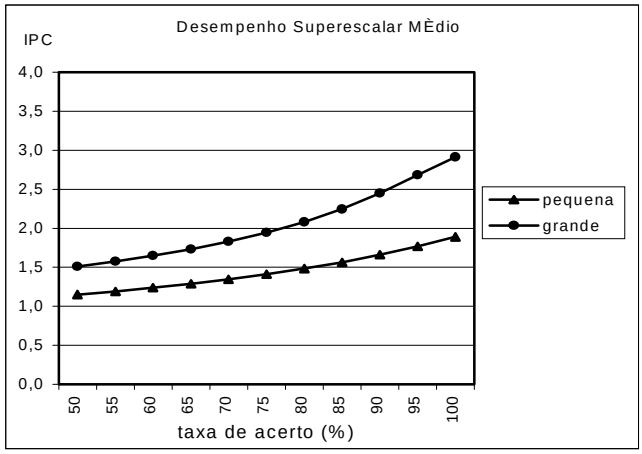


FIGURA 6.19 - Desempenho Médio da Arquitetura Superescalar

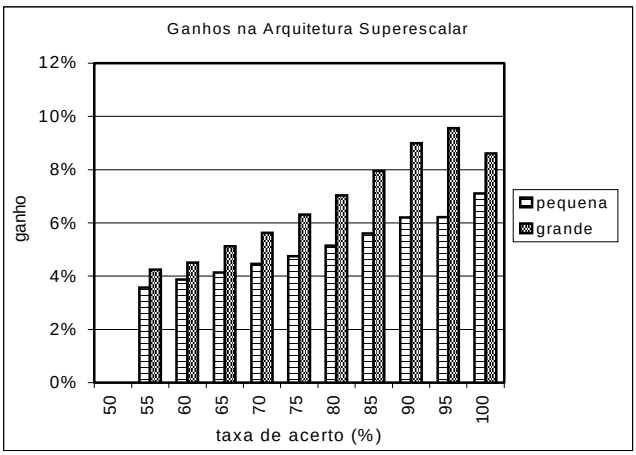


FIGURA 6.20 - Ganhos com a Melhoria da Previsão na Superescalar

A próxima arquitetura a ser simulada foi a SMT-4. As figuras 6.21 e 6.23 mostram o desempenho de cada programa em ambas arquiteturas pequena e grande, respectivamente. Em ambas arquiteturas, os programas *ijpeg* e *li* têm alcançado os melhores desempenhos novamente. Também, na arquitetura SMT-4, os programas *mgrid* e *fpppp* mantiveram o mesmo comportamento observado na arquitetura superescalar, apresentando um desempenho semelhante através de todas as configurações.

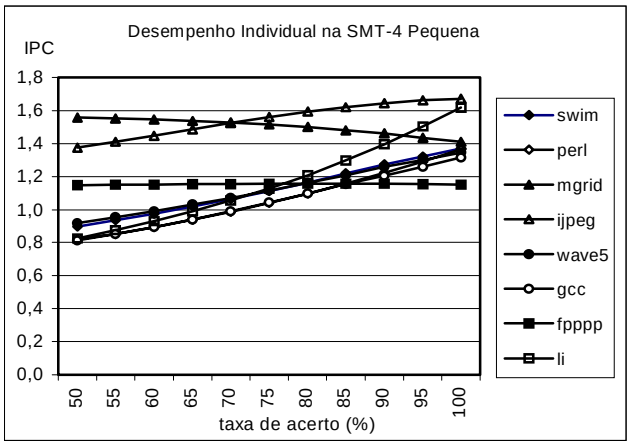


FIGURA 6.21 - Desempenho Individual na Arquitetura SMT-4 Pequena

Nota-se que na arquitetura SMT-4 pequena, os programas *mgrid* e *fpppp* foram penalizados pela melhoria da previsão. Parece estranho mas não se pode esquecer que vários programas compartilham os mesmos recursos, principalmente as unidades funcionais, e interferem entre si. Assim, quando o previsor é melhorado, as chances de obtenção de mais paralelismo aumentam para todas as tarefas, reduzindo assim a quantidade de recursos disponíveis. Como alguns programas são mais sensíveis à previsão e obtêm maior paralelismo com o aumento da taxa de acerto, eles se apoderam de mais recursos e os demais programas, não sensíveis à previsão, são penalizados. Já na arquitetura SMT-8, a quantidade de recursos disponíveis é grande o suficiente para atender a esta demanda de recursos e esta situação quase não ocorre.

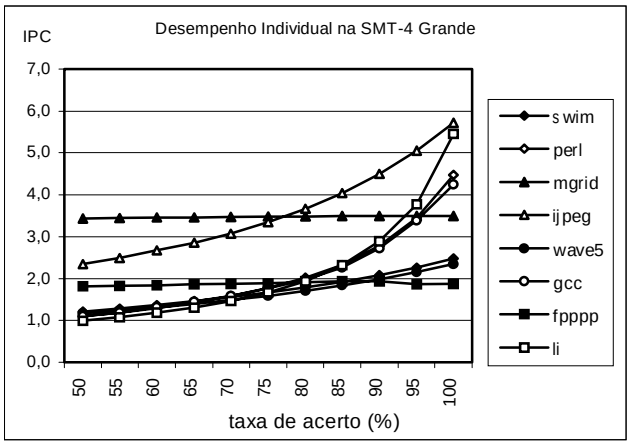


FIGURA 6.22 - Desempenho Individual na Arquitetura SMT-4 Grande

A figura 6.23 mostra o desempenho global para ambas as arquiteturas SMT-4 pequena e grande. O desempenho é medido como a média aritmética dos desempenhos de cada grupo de programa simulado (vide tabela 6.9), que por sua vez é soma dos *ipcs* individuais de cada programa dentro do respectivo grupo. O maior impacto causado pelo aumento da taxa de acerto na previsão de desvios foi encontrado na arquitetura SMT-4 grande, assim como na arquitetura superescalar. Contudo, não existe nenhuma intersecção entre as curvas da SMT-4 pequena e grande. Embora a SMT-4 pequena tenha recursos compatíveis com a arquitetura superescalar grande, seu desempenho é superior, pois extrai paralelismo de 4 tarefas. Em arquiteturas SMT, tanto o paralelismo no nível de instrução quanto aquele no nível de tarefas são explorados.

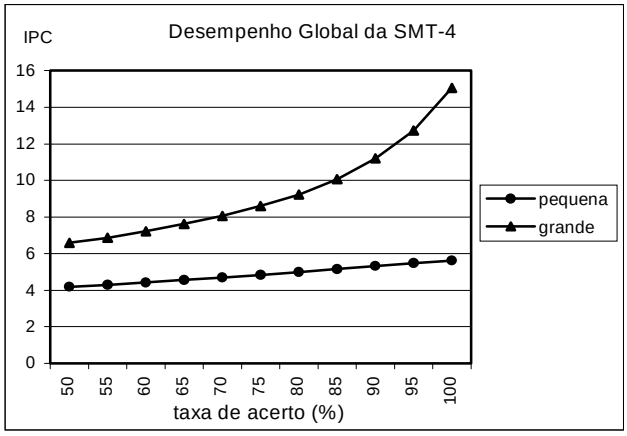


FIGURA 6.23 - Desempenho Global da Arquitetura SMT-4

Apesar disto, as curvas de desempenho destas arquiteturas são semelhantes. Isto significa que nesta arquitetura SMT, o previsor não necessita ser tão preciso quanto naquela arquitetura superescalar. Isto ocorre porque o paralelismo em nível de instrução é em parte trocado pelo paralelismo entre as tarefas e assim a especulação é menos intensa. A figura 6.24 mostra os ganhos obtidos pela melhoria na previsão, com interpretação análoga àquela da figura 6.20. Neste gráfico, nós podemos ver que a previsão perfeita na SMT-4 grande implica em um ganho de 18% sobre a mesma arquitetura com taxa de previsão de 95%. Foi observado que o previsor afeta o desempenho devido a quantidade de recursos disponíveis ser suficiente para suprir o paralelismo entre a tarefas e ainda prover recursos para o paralelismo no nível de instrução dentro de cada tarefa. Assim, conclui-se que, quando o paralelismo entre as tarefas é a principal fonte de paralelismo, o aumento da taxa de acerto na previsão quase não interfere no desempenho final.

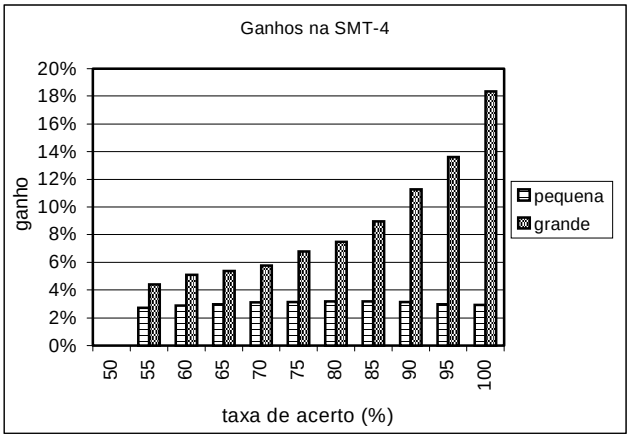


FIGURA 6.24 - Ganhos com a Melhoria da Previsão na SMT-4

Para medir o impacto da previsão em arquiteturas SMT mais agressivas, foi simulada a execução simultânea de 8 tarefas. A figura 6.25 mostra os desempenhos individuais para cada programa de avaliação na arquitetura SMT-8 pequena, anteriormente especificada Neste gráfico é possível ver claramente que o aumento no número de tarefas gera uma sobrecarga de paralelismo disponível entre tarefas, causando a diminuição do desempenho individual quando melhoramos a taxa de acerto da previsão de desvios. Quando comparada com a arquitetura SMT-4 pequena (figura 6.21) é possível notar que a falta de recursos neste caso prejudica alguns programas.

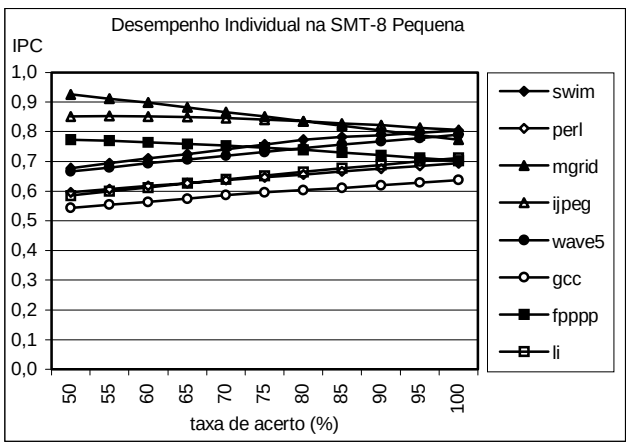


FIGURA 6.25 - Desempenho Individual na Arquitetura SMT-8 Pequena

A figura 6.26 mostra os resultados obtidos pela arquitetura SMT-8 grande. Mais uma vez, os experimentos confirmam que os programas *mgrid* e *fpppp* não são afetados pela melhoria da previsão. Em uma análise mais apurada dos seus códigos, foi verificado que o número de instruções por desvios está entre 70 e 80 (tamanho médio dos blocos básicos). Assim, o desempenho deste programas é limitado principalmente pelas dependências de dados e de recursos, pois o número de desvios é pequeno.

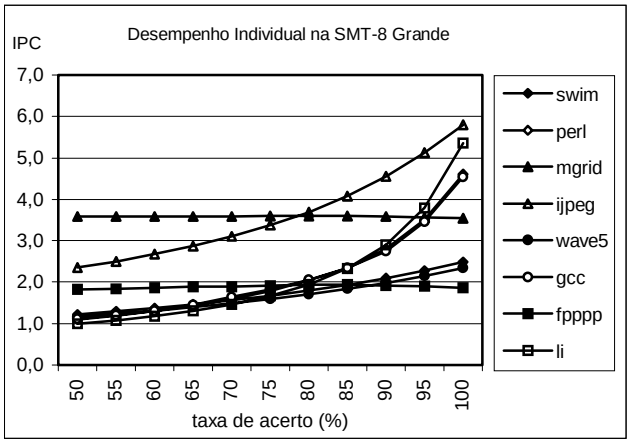


FIGURA 6.26 - Desempenho Individual na SMT-8 Grande

A comparação entre o desempenho global de ambas arquiteturas SMT-8 é mostrada na figura 6.27. Nota-se que a arquitetura SMT-8 pequena quase não foi afetada pelo aumento da taxa de acerto. Da mesma forma como na SMT-4 pequena (figura 6.23), a disponibilidade de recursos, se comparada com o número de tarefas, não permite que o paralelismo no nível de instrução seja explorado muito bem. Entretanto, a SMT-8 grande proporciona uma grande melhoria no desempenho com a melhoria da previsão. Isto ocorre por que existem muitos recursos que permitam que o paralelismo no nível de instrução seja aproveitado em conjunto com o paralelismo entre tarefas.

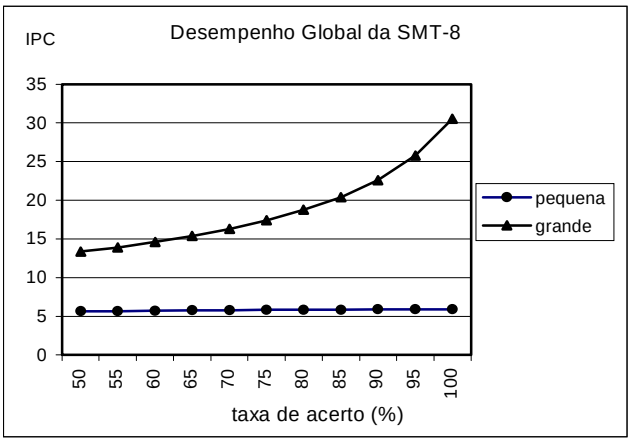


FIGURA 6.27 - Desempenho Global da Arquitetura SMT-8

A figura 6.28 mostra o ganho causado pela melhoria da previsão para as arquiteturas SMT-8 pequena e grande, cuja interpretação é análoga àquela da figura 6.20. De fato, a arquitetura SMT-8 pequena tem seu ganho levemente reduzido a cada aumento da taxa de acerto. Por outro lado, a SMT-8 grande apresenta ganhos de 18% quando a taxa de acerto passa de 95% para 100%.

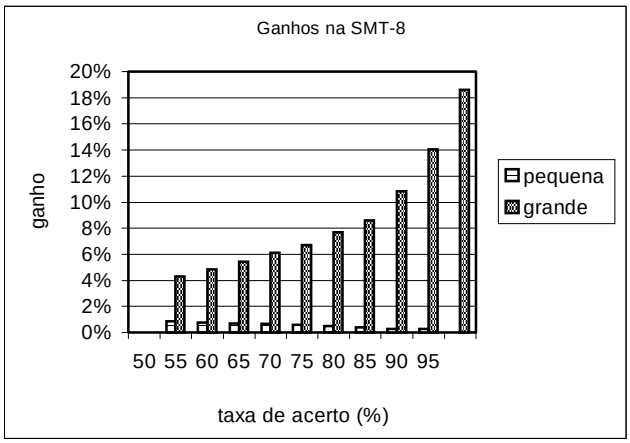


FIGURA 6.28 - Ganhos com a Melhoria da Previsão na SMT-8

6.3 Adicionando Escalonamento de Processos

Como próxima etapa no desenvolvimento da arquitetura SEMPRE, o simulador SMT básico foi alterado para incluir o escalonamento de processos. Para isto, uma nova estrutura foi inserida para representar a fila de processos, mantendo todas as suas características anteriores inalteradas. Também foi inserido o estágio de pré-busca de processos. A figura 6.29 apresenta a arquitetura modificada.

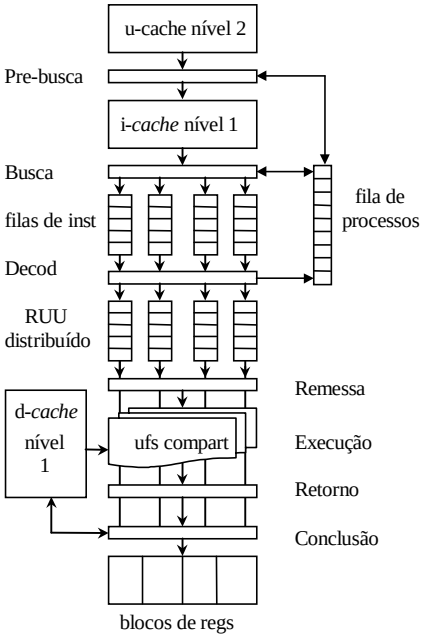


FIGURA 6.29 - Arquitetura SEMPRE Simulada

O estágio de pré-busca de processos realiza também a função de troca de contexto, além daquela inicialmente prevista para ele (vide seção 4.2.2). Suas atividades principais atualmente são 5: 1) percorrer os *slots* de busca e verificar quais estão vazios ou com processos bloqueados; 2) percorrer a fila de processos e verificar quais estão prontos na *cache* de instruções; 3) realizar a troca de contexto (retirar processos da fila de prontos e colocar nos *slots* de busca; 4) solicitar a busca de instruções na *cache* de instruções para

o próximo processo a ser escalonado e 5) controlar o decremento das fatias de tempos dos processos que estão nos *slots*.

Deve-se notar agora que a arquitetura pode manter um número maior de processos do que o numero de *slots*. Como exemplo, pode-se dizer que uma arquitetura contendo 4 *slots* pode executar 8 processos. Os 4 *slots* mantém 4 processos ativos e que podem ser simultaneamente executados. Para isso, são utilizados 4 filas de instruções (filas de busca), 4 filas RUU e 8 bancos de registradores para manter os contextos. A largura do *pipeline* continua sendo aquela necessária para uma arquitetura SMT de 4 *slots*. Em suma, os recursos que são utilizados a mais, do que aqueles comumente utilizados em uma arquitetura SMT convencional, são basicamente os bancos de registradores excedentes e os barramentos para conectar todos estes bancos aos *slots*.

As filas de instruções e as filas RUU podem agora compartilhar segmentos de código de diferentes processos. Isto implica que durante a ocorrência de uma má previsão, a necessidade de esvaziar o *pipeline* das instruções de determinado processo obriga que os demais processos, contidos nas filas afetadas, também sejam afetados. Ou seja, a penalidade sofrida por um processo pela má previsão, no que se refere ao tempo de restabelecer o *pipeline*, implica em penalizar também outros processos associados às partes do *pipeline* afetadas. Na implementação atual, esta penalidade é de 3 ciclos.

Para facilitar a implementação do conceito de compartilhamento das filas de instruções e das filas RUU, foram criadas filas virtuais. Isto significa que, na realidade, o simulador continua mantendo cada processo em seu ambiente separado, cada um com suas filas privadas (filas reais) e, para simular a utilização de filas compartilhadas, foram criadas filas virtuais contendo ponteiros para as filas reais, conforme mostra a figura 6.30. Esta figura mostra um exemplo de uma fila virtual, que poderia ser tanto uma fila de instruções quanto uma fila RUU, contendo instruções de diferentes processos, e ao mesmo tempo mostra também a situação real, onde cada processo tem sua fila privada com suas respectivas instruções.

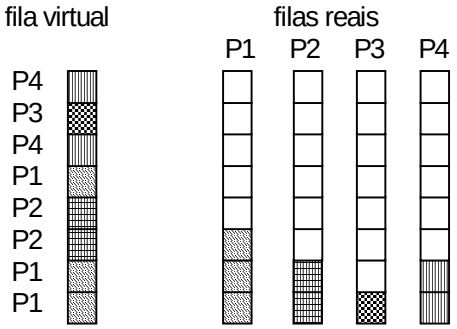


FIGURA 6.30 - Esquema de Filas Reais e Virtuais

Não se pode esquecer que o simulador SMT básico já funciona tal como vários simuladores *sim-outorder* concorrentes. Com a utilização das filas virtuais, esta característica continua sendo mantida e a engrenagem básica continua sendo a original da ferramenta *SimpleScalar*. Assim, a recuperação correta do estado do processador é feita de forma bastante simples, recuperando somente as filas reais associadas ao processo penalizado e atualizando as filas virtuais relacionadas. Esta técnica também

garante estabilidade ao simulador, pois com a preservação do funcionamento original do *SimpleScalar*, o corretismo da execução também é mantido.

Com a inserção da fila de processos, a *cache* de instruções passa a trabalhar de acordo com o algoritmo de escalonamento de processos. O efeito desta mudança é o mesmo como se o algoritmo de reposição da *cache* fosse controlado pelo escalonamento de processos. Na ocorrência de uma falta na *cache*, a busca na *cache* nível 2 não é ativada pela própria *cache* nível 1, como normalmente é feito. Agora, o processo faltante vai para a fila de processos e esta busca será coordenada futuramente quando houver o interesse do estágio de pré-busca.

Uma outra nova característica inserida no simulador SMT básico foi a utilização da fatia de tempo por processo (*time-slice*). Agora, mesmo que não ocorra nenhuma falta na *cache*, para um determinado processo ativo, sua execução é bloqueada após a decorrência de um certo número de ciclos. Esta atividade é normalmente executada pelo sistema operacional e, no presente trabalho, as duas abordagens foram simuladas e comparadas, confirmando que o escalonamento por *hardware* é mais eficiente. As próximas seções mostram as simulações realizadas, bem como os desempenhos e ganhos obtidos com esta nova arquitetura.

6.3.1 Análise dos Desempenhos e Ganhos

Não é justo comparar o desempenho da arquitetura SEMPRE com uma arquitetura superescalar convencional, pois entende-se que uma arquitetura SMT possui mais recursos e dispõe de mais instruções independentes, mesmo porque a diferença do paralelismo obtido por estas duas abordagens já é comumente conhecida. Se existe algum interesse em se medir estes valores, pode-se interpretá-los pelos resultados apresentados na seção 6.2.4. A finalidade das simulações realizadas no presente trabalho é mostrar que os benefícios da arquitetura SEMPRE são aparentes mesmo se comparados com aqueles obtidos em outra arquitetura SMT equivalente.

Para testar a arquitetura sob um conjunto mais expressivo de programas, em todas as próximas simulações apresentadas nesta tese foram utilizados 14 diferentes programas do SPEC95, 7 de inteiros (*perl*, *ijpeg*, *gcc*, *li*, *compress* e *vortex*) e 7 de ponto flutuante (*swim*, *mgrid*, *wave5*, *fpppp*, *applu* e *turb3d*). Várias combinações foram utilizadas de acordo com o número de programas sendo simulado e de acordo com o número de *slots* da arquitetura. Estas combinações podem ser vistas nos Anexos 1 e 2. Novamente, foram considerados dois tipos de arquiteturas, de 4 e 8 *slots*, por motivos já esclarecidos anteriormente, com a diferença que o numero de processos pode ser maior. A tabela 6.3 mostra o volume do *hardware* básico utilizado.

TABELA 6.3 - Quantidade de *Hardware*

Arquitetura com 4 *Slots*	**Arquitetura com 8 *Slots***
width=8, ul2=(256k, a=16)	width=16, ul2=(512k,a=16)
il1=dl1=(32k, a=16)	il1= dl1 = (64k, a=16)
15 fu =(4 int-alu, 2 fp-alu, 3	30 fu = (8 int-alu, 4 fp-alu, 6
int-mult, 4 fp-mult, 2 ld/st)	int-mult, 8 fp-mult, 4 ld/st)
ruu size = 32 / lsq size = 16	ruu size = 64 / lsq size = 32

Cada simulação foi executada até o primeiro programa completar 200 milhões de instruções. Também foram desprezadas as primeiras 100 milhões de instruções de todos os programas. As latências consideradas são as mesmas consideradas em todas as simulações anteriores (vide tabela 6.1). O primeiro módulo de simulações foi realizado na arquitetura SMT básica para se obter uma referência do desempenho de uma arquitetura SMT tradicional, sem considerar a interferência do sistema operacional (S.O.), ou seja, tanto o escalonamento de processos quanto a utilização da fatia de tempo não foram considerados. Os resultados em *ipc* são resumidos na tabela 6.4.

TABELA 6.4 - Desempenho da SMT Tradicional Sem S.O.

SMT-4	**SMT-8**
3,90 ipc	6,84 ipc

Foi então simulada a arquitetura SEMPRE, denotada de SEMPRE-4.x ou SEMPRE-8.x, conforme tenham 4 ou 8 *slots*, respectivamente, sendo que "x" indica o número de processos em execução. Para a arquitetura SEMPRE-4 foram considerados 4, 6, 8, 10 e 12 processos, e para a arquitetura SEMPRE-8 foram considerados 8, 10, 12, 14 e 16 processos. Foi definida uma fatia de tempo de 10.000 ciclos, em função do número total de ciclos simulados, que é bem pequeno se comparado com uma execução real. Uma fatia de tempo real seria algo próxima de 10.000.000 de ciclos, segundo informado em [ROB 96]. Os gráficos 6.31 e 6.32 mostram os desempenhos obtidos.

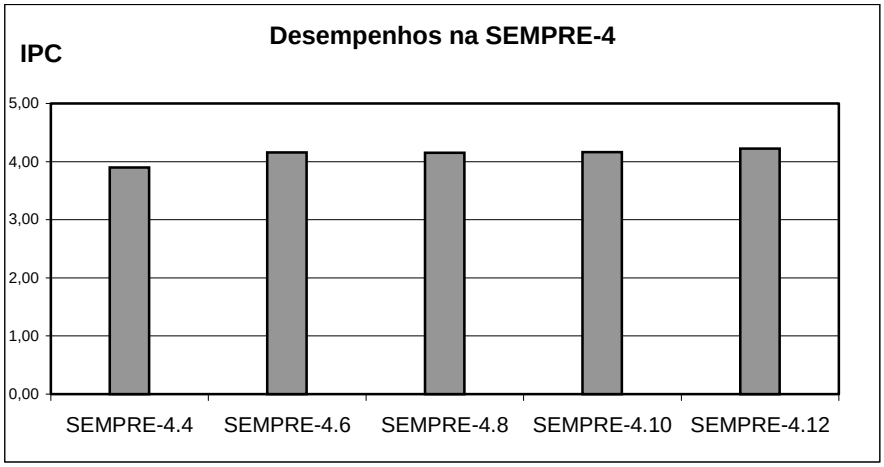


FIGURA 6.31 - Desempenhos na SEMPRE-4

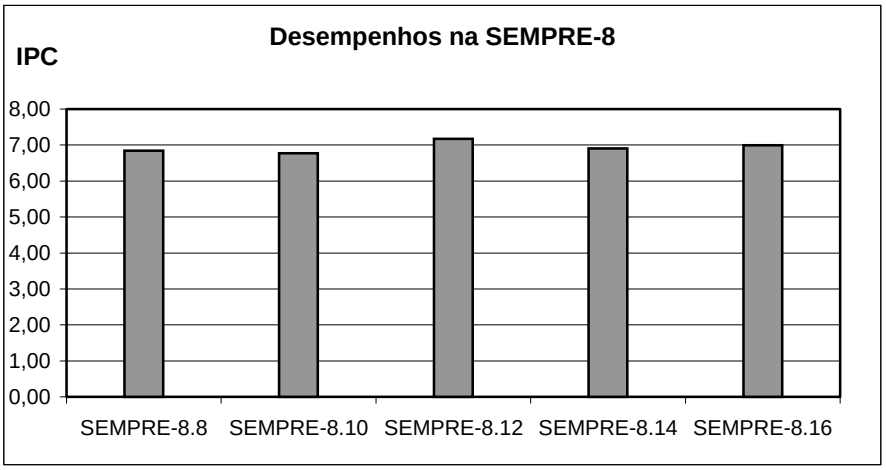


FIGURA 6.32 - Desempenhos na SEMPRE-8

Pode-se observar nestes gráficos que os desempenhos para diferentes números de processos não apresentam diferenças muito discrepantes entre si. Para a arquitetura SEMPRE-4, parece haver uma tendência no aumento do desempenho quando o número de processos aumenta. Esta tendência não é observada na SEMPRE-8. Deve ser salientado que o desempenho depende muito da combinação de programas que foi utilizada, e por isso, quanto maior o número de combinações utilizadas, a média dos resultados se comporta de forma mais tendenciosa. O ganhos sobre as arquiteturas SMT tradicionais são apresentados nos gráficos das figuras 6.33 e 6.34.

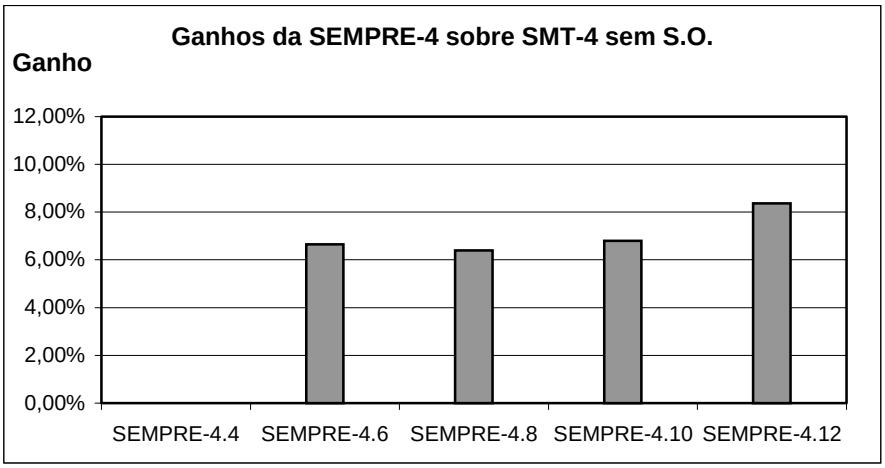


FIGURA 6.33 - Ganhos SEMPRE-4 x SMT-4 (sem S.O.)

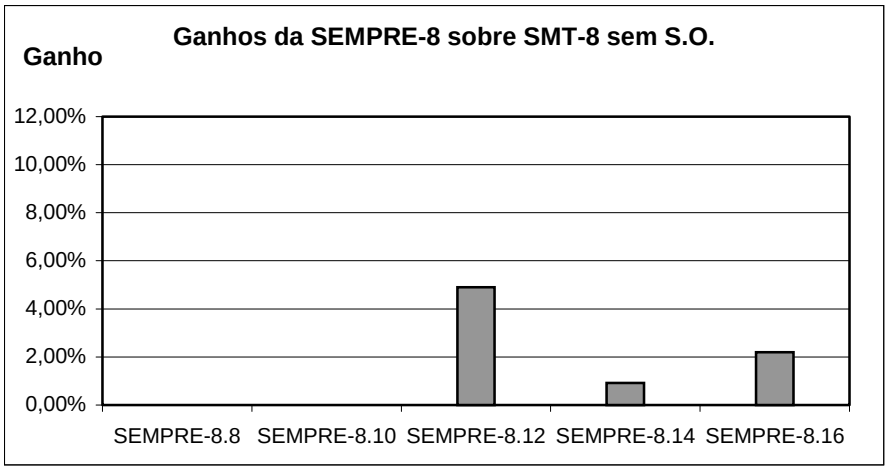


FIGURA 6.34 - Ganhos SEMPRE-8 x SMT-8 (sem S.O.)

Independentemente de qualquer tipo de tendência em acréscimo ou decréscimo do ganho obtido, pode-se observar que o desempenho da arquitetura SEMPRE é sempre superior, podendo atingir valores acima de 8% para 4 *slots* e acima de 4% para 8 *slots*, no desempenho médio final. Percebe-se, de uma forma geral, que os ganhos sobre 4 *slots* são mais expressivos e distribuídos que os ganhos sobre 8 *slots*. Deve ser observado no entanto, que a arquitetura SMT-4 foi simulada com até 12 processos, que representam 3 vezes o numero de *slots*, enquanto que a arquitetura SMT-8 foi simulada com até 16 processos, que representam apenas o dobro do número de *slots*.

A arquitetura SMT-8 deveria ser simulada com até 24 processos para visualizar uma correspondência mais justa entre ambas, no entanto, devido aos limites do simulador,

esta possibilidade ficou inviabilizada. Uma observação mais apurada permite notar que existe uma tendência em que o desempenho se reduza na medida em que o número de processos atinja o dobro do número de *slots*, depois volta a subir novamente devido à pré-busca de processos por *hardware*, que tira proveito do maior numero de processos. Obviamente, o ganho da arquitetura SEMPRE executando o mesmo número de processos que o de *slots* disponíveis é nulo, pois o desempenho neste caso é o mesmo que aquele obtido pela arquitetura SMT tradicional.

O ganho da arquitetura SEMPRE é provido basicamente mascarando as faltas na *cache* de instruções, sustentado pelo escalonamento de processos que substitui os tempos de paradas na busca (que ocorre na SMT tradicional) pela execução de outros processos (que ocorre na SEMPRE). Para avaliar a influência da escolha adequada da fatia de tempo, outras simulações foram realizadas sobre as arquiteturas SMT-4.12 e SMT-8.16. Os gráficos das figuras 6.35 e 6.36 mostram o desempenho obtido em função da variação do valor da fatia de tempo. Foram utilizadas fatias de tempo pequenas tal como de 100 ciclos e até fatias reais de tempo tal como de 10 milhões de ciclos. De certa forma, pode-se observar com fatias de tempo intermediárias os desempenhos são maiores, principalmente para a arquitetura SEMPRE-8.

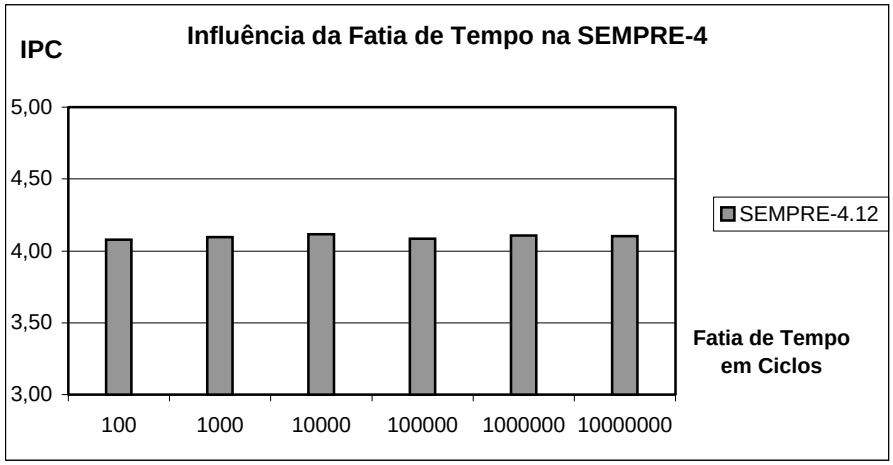


FIGURA 6.35 - Variando a fatia de Tempo da Arquitetura SEMPRE-4

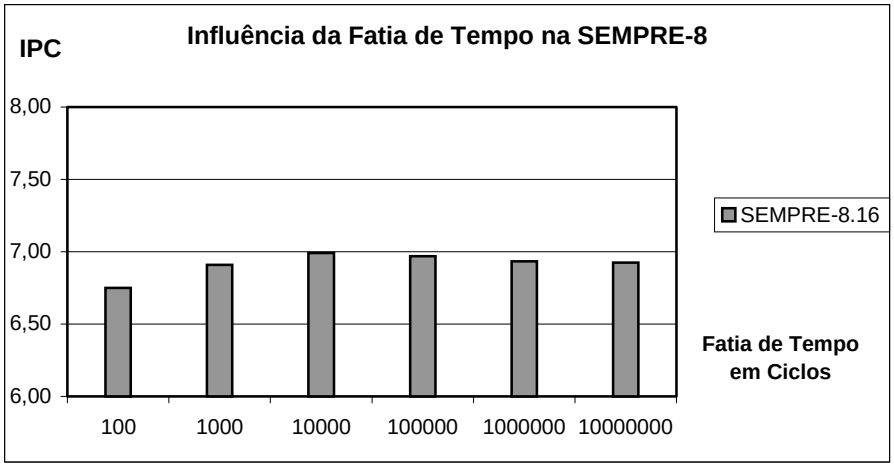


FIGURA 6.36 - Variando a fatia de Tempo da Arquitetura SEMPRE-8

Sabe-se que o sistema operacional utiliza uma fatia de tempo bastante grande com relação ao tempo de troca de contexto, para que o desempenho final não fique prejudicado, ou melhor, para que a quantidade de trabalho útil seja maior que a quantidade de trabalho inútil perdido com as trocas de contexto. Obviamente, se fosse possível, o sistema operacional trocaria os contextos instantaneamente, já que assim a ilusão de paralelismo seria maior e os processos não teriam que esperar muito tempo para retornar a execução. Essa possibilidade é, em parte, viabilizada com a arquitetura SEMPRE, que permite a utilização de fatias de tempo menores com desempenho sempre superior aquele obtido pela arquitetura SMT tradicional. Na próxima seção é considerada a interferência do sistema operacional sobre uma arquitetura SMT, permitindo observar que o ganho da arquitetura SEMPRE é muito mais expressivo do que obtido até agora.

6.3.2 Análise da Influência do Sistema Operacional

Nos trabalhos que têm sido realizados com este tipo de arquitetura, não tem sido analisada a troca de contexto, que normalmente é realizada pelo sistema operacional. O simulador SMT básico foi alterado para inserir esta característica, originando assim o simulador SOSMT, que inclui um escalonamento de processos no nível do sistema operacional. A diferença básica deste simulador com o simulador SEMPRE é que as trocas de contextos são realizadas pelo sistema operacional e não pelo *hardware*. Com isso, uma vez que os processos estão ativos nos *slots* da arquitetura, eles somente serão trocados mediante o término da fatia de tempo, controlado pelo sistema operacional, já que ainda não estão sendo considerados os bloqueios por entrada e saída e outros. A arquitetura SOSMT não é sensível as faltas na *cache* de instruções, como é o caso da arquitetura SEMPRE. Os gráficos das figuras 6.37 e 6.38 mostram os desempenhos obtidos.

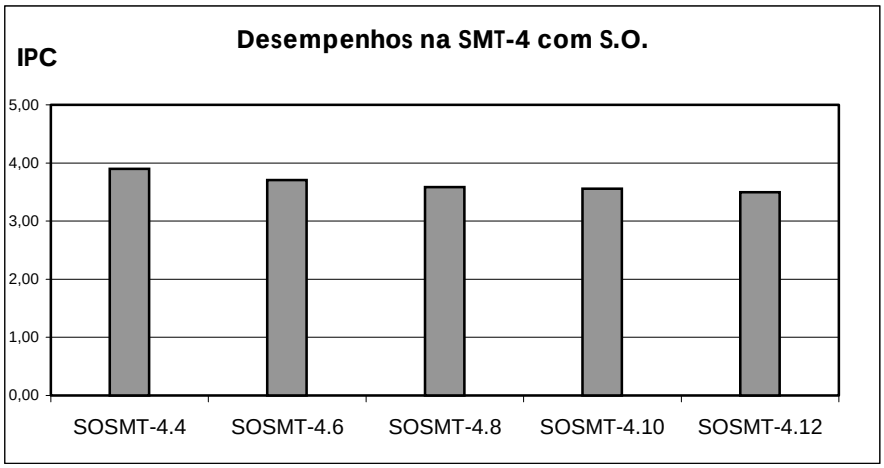


FIGURA 6.37 - Desempenhos na SMT-4 na Presença do S.O.

Pode-se observar nestas figuras que, de uma forma geral, o desempenho da arquitetura decresce na medida em que aumenta o número de processos no sistema. Este fato é mais intenso na arquitetura SMT-4, que de 4 processos para 12 processos sofreu uma perda de mais de 10% do desempenho. Este fato é causado não pelo prejuízo com o tempo da troca de contexto, que não está sendo considerado aqui devido pois é pequeno, mas sim devido ao não aproveitamento dos tempos de faltas na *cache* de instruções. Isto se

agrava na medida em que um maior número de processos passa a ser escalonado pelo sistema operacional, pois a somatória total do tempo perdido também aumenta. Os gráficos das figuras 6.39 e 6.40 mostram as perdas obtidas nas diferentes configurações consideradas.

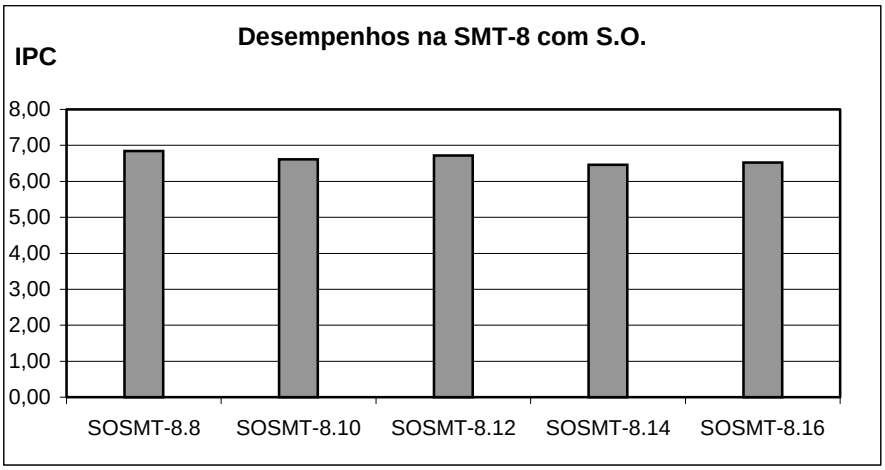


FIGURA 6.38 - Desempenhos na SMT-8 na Presença do S.O.

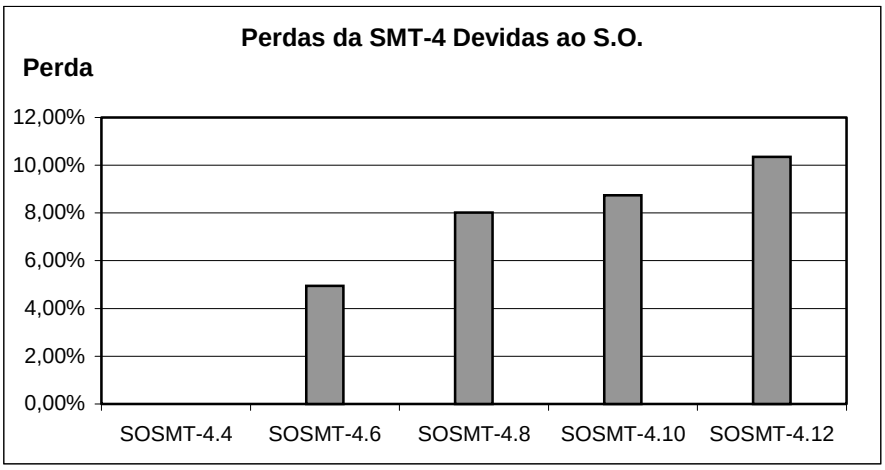


FIGURA 6.39 - Perdas de Desempenho na SMT-4 Devidas ao S.O.

Uma observação importante a ser feita é que, quando um processo permanece ativo no *slot* de forma ininterrupta por um determinado tempo, sem ser substituído quando ocorrem as faltas na *cache* de instruções, este pode causar uma super-utilização desta *cache* em benefício próprio, retirando muitas instruções dos demais processos e provocando futuramente faltas na *cache*. Porém, o grau desta interferência depende do tamanho da fatia de tempo.

Por isso, foram também realizadas simulações analisando a variação da fatia de tempo nas arquiteturas SMT escalonadas pelo sistema operacional e os resultados são mostrados nos gráficos das figuras 6.41 e 6.42, para arquiteturas SMT-4 e SMT-8, respectivamente. Observa-se então que diferentemente da arquitetura SEMPRE, que tira melhor proveito de fatias de tempos intermediárias, as arquiteturas SMT, quando escalonadas pelo sistema operacional, tiram melhores proveitos de fatias de tempos maiores. Esta conclusão se deve ao fato de que, com uma fatia de tempo maior, cada processo recentemente escalonado dispõe de tempo suficiente para estabilizar as suas

instruções na *cache*. Pode-se dizer que usando grandes fatias de tempo, o prejuízo das faltas na *cache* se dilui no tempo. Inclusive, este é uma das justificativas para uma fatia de tempo de 10 milhões de ciclos [ROB 96]. Uma nova observação nestes gráficos, permite notar que para este valor de tempo os desempenhos se aproximam do máximo e se estabilizam.

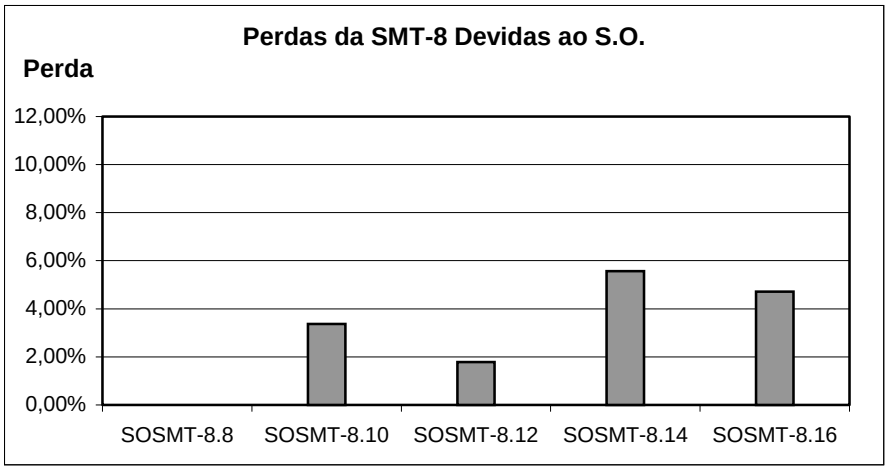


FIGURA 6.40 - Perdas de Desempenho na SMT-8 Devidas ao S.O.

De qualquer forma, é perceptível que a variação da fatia de tempo causa um maior impacto nas arquiteturas escalonadas pelo sistema operacional do que em arquiteturas escalonadas pelo *hardware*. Este comportamento é compreensível pois, no primeiro caso, as trocas de contexto exclusivamente dependem deste valor e no segundo caso, nem tanto, pois é provável que na maior parte do tempo as trocas de contextos sejam dirigidas pelas faltas na *cache* de instruções e que as fatias de tempo nem sejam consumidas até o final.

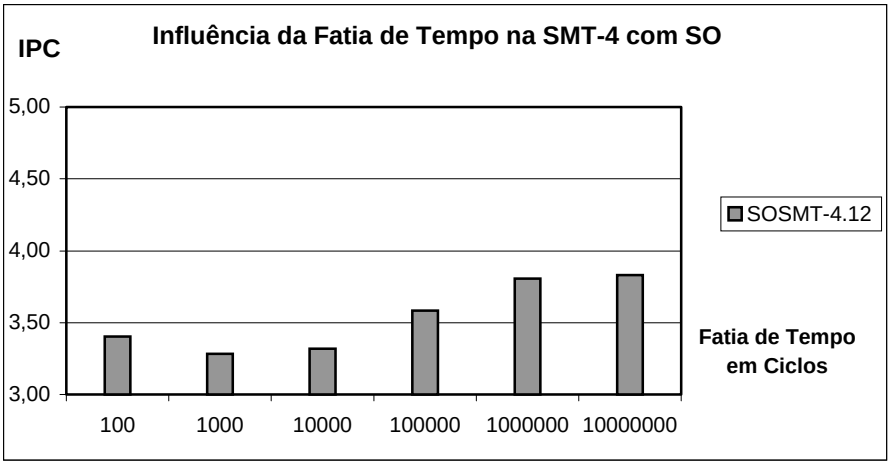


FIGURA 6.41 - Variando a fatia de Tempo na Arquitetura SMT-4 Pelo S.O.

Voltando a analisar os gráficos das figuras 6.35 e 6.36, pode-se agora entender porque usando fatias de tempos não muito grandes e nem muito pequenas os desempenhos na arquitetura SEMPRE são maiores. Conforme já deduzido, quando são utilizadas fatias de tempo muito grandes, aqueles processos que se estabilizam primeiro passam a dominar a *cache*. Mas como a troca de contexto também pode ocorrer devido às faltas na *cache*, os demais processos que ainda não estão estabilizados não têm grandes

chances de serem executados, pois quando são escalonados, em pouco tempo eles cedem seus lugares novamente para os outros, devido justamente às faltas na *cache*. Por outro lado, quando são utilizadas fatias de tempo muito pequenas, existem muito menos chances para que os processos se estabilizem na *cache*, ou pior, não existe tempo para que uma quantidade razoável das instruções mais utilizadas de cada processo permaneça na *cache*, pois muito mais rapidamente os contextos são trocados. Neste sentido, os valores intermediários para as fatias de tempo são mais indicados. De qualquer forma, seja qual for a fatia de tempo utilizada, uma arquitetura escalonada por *hardware* provê maior desempenho do que uma arquitetura escalonada pelo sistema operacional, conforme mostram os gráficos das figuras 6.43 e 6.44.

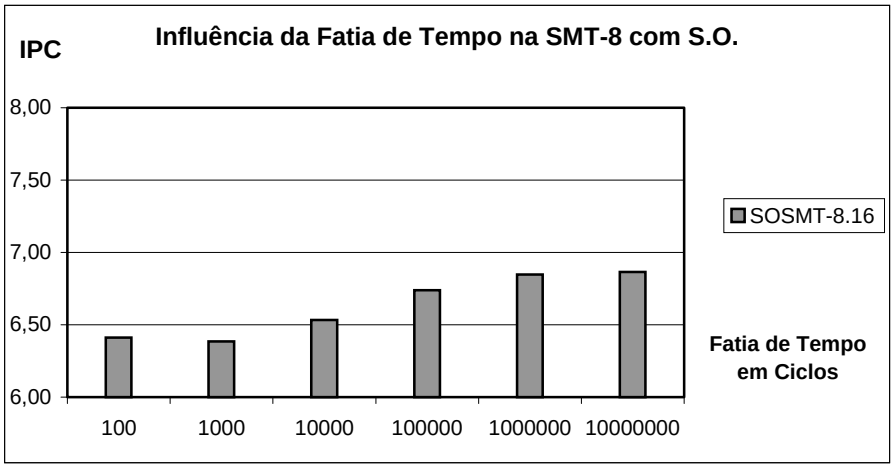


FIGURA 6.42 - Variando a fatia de Tempo na Arquitetura SMT-8 Pelo S.O.

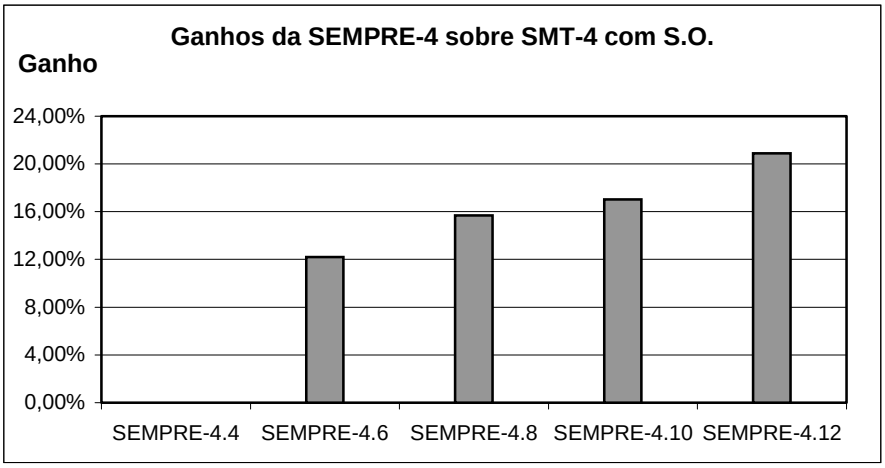


FIGURA 6.43 - Ganhos da SEMPRE-4 sobre SMT-4 com S.O.

Notoriamente, o ganho da arquitetura SEMPRE sobre uma arquitetura SMT-4 escalonada pelo sistema operacional atinge valores em torno de 12% a 21%. Os mesmos índices de ganhos não são tão altos para as arquiteturas de 8 *slots*, que ainda assim superam valores de 7%. Estes resultados confirmam a vantagem do escalonamento por *hardware*, e não podemos esquecer que este ganho é considerado sobre arquiteturas com *pipeline* de larguras e número de estruturas equivalentes, com exceção do número de bancos de registradores, que é maior na arquitetura SEMPRE. Conclusivamente, não foram encontrados valores contrários em nenhuma das simulações realizadas neste trabalho onde a arquitetura SEMPRE tivesse algum desempenho inferior.

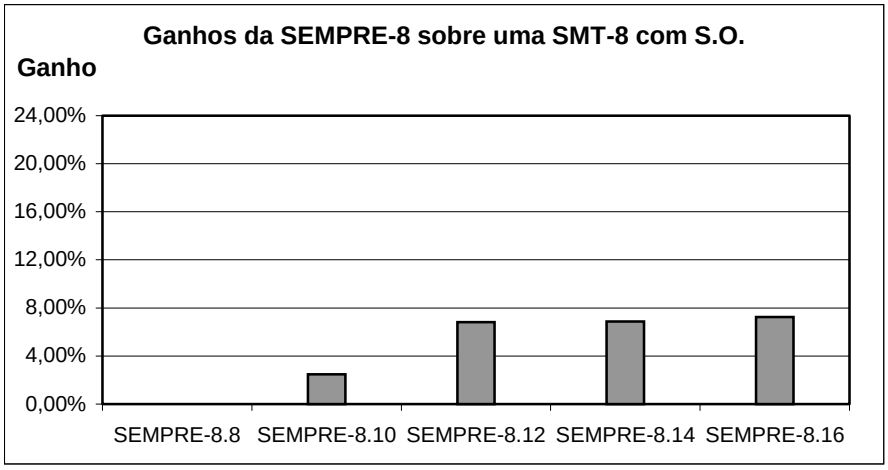


FIGURA 6.44 - Ganhos da SEMPRE-8 sobre SMT-8 com S.O.

Apesar de não ser muito correto comparar a arquitetura SEMPRE com aquelas apresentadas no capítulo 3, pois seria necessária uma certificação de que ambas possuem as mesmas características básicas e configurações, tais como tempos de latências, conjunto de programas simulados, conjunto de instruções e dimensões do *pipeline*, a tabela 6.5 mostra os desempenhos (em *ipc*) máximos aproximados obtidos por cada uma, apenas de forma ilustrativa. Porém, vale a pena ressaltar que a arquitetura SEMPRE se diferencia das demais pela sua capacidade de execução e escalonamento de processos.

TABELA 6.5 - Desempenhos Obtidos em Diversas Arquiteturas SMT

Arquitetura	**Hirata**	**Yamamoto**	**Tullsen**	**SEMPRE**
4 Slots	3.7	3.9	4.6	4.2
8 Slots	5.8	------	5.3	7.2

7 Conclusões e Trabalhos Futuros

O presente trabalho apresenta uma arquitetura multi-tarefas simultâneas voltada para a execução de múltiplos processos, existentes em grande quantidade nas estações de trabalho compartilhadas e servidores de rede. Esta arquitetura é capaz de explorar o paralelismo dentro de cada processo, executando instruções fora-de-ordem, tão bem quanto explorar o paralelismo entre os processos, escondendo latências e aumentando o desempenho final do sistema. Com relação ao projeto, as principais facilidades desta arquitetura são:

- A arquitetura permite a execução simultânea de várias aplicações, além de prover toda parte de escalonamento básico de processos, facilitando assim o trabalho do sistema operacional e tornando-o mais poderoso. Além disso, os aplicativos que são executados simultaneamente são processos independentes que quase sempre não comunicam entre si, facilitando a obtenção de paralelismo.

- A arquitetura suporta troca antecipada de contexto, bem como múltiplos processos compartilhando o mesmo *slot*. A gerência destes processos é realizada através das filas de processos ativos, maximizando a utilização do *hardware* que tradicionalmente abriga uma única tarefa por *slot*.

- A arquitetura é baseada em filas do tipo FIFO e algoritmos de escalonamento baseados em *round-robin*, provendo rapidez na execução e simplicidade da lógica de manipulação.

- A arquitetura contém um conjunto de instruções específicas para facilitar o desenvolvimento do sistema operacional. Estas instruções são simples e de fácil decodificação, sendo resolvidas diretamente pelo estágio de busca de instruções. Isto permite ganho de tempo que tradicionalmente é perdido com gerenciamento de processos e troca de contexto no nível de sistema operacional.

- A arquitetura suporta remoção de instruções inconvenientes, quando solicitadas pelo sistema operacional, ou quando provenientes de caminhos incorretos dos desvios ou de exceções, permitindo assim a recuperação do estado preciso da arquitetura. Estas características são suportadas graças ao uso de um *bit* alternador nas entradas das filas de instruções e de reordenação.

- A arquitetura suporta um mecanismo de pré-busca de processos que proporciona maior confiabilidade nas trocas de contextos, através da redução da freqüência de faltas na *cache* de instruções.

Antes que o simulador da arquitetura SEMPRE estivesse pronto foi realizada uma análise comportamental da arquitetura na presença de processos na *cache* através de uma modelagem analítica da unidade de busca. Embora o processo de simulação dirigida pela execução de código binário seja mais eficiente, os resultados dessa modelagem são extremamente importantes pois permitiram estimar que :

- As latências da *cache* L2 constituem um fator que degrada o desempenho da arquitetura e o mecanismo de pré-busca pode ser utilizado para diminuir este prejuízo. Em alguns casos, a capacidade de despacho é aumentada em 25%. Também, o mecanismo de pré-busca tolera latências de acesso na *cache* L2 de 9 ciclos com o mesmo desempenho que uma arquitetura similar sem pré-busca tolera latências de apenas 4 ciclos.

- O aumento do número de processos que são escalonados causa pouca variação no ganho relativo entre o mecanismo proposto e o melhor caso, pois com muito poucos processos a mais já é possível prover eficiente troca de contexto e o mecanismo proposto se comporta como o melhor caso.

O simulador SEMPRE de base desenvolvido e descrito neste trabalho é uma ferramenta chave que permite analisar e avaliar o desempenho de diferentes configurações deste tipo de arquitetura, facilitando a fase de projeto de microprocessadores agressivos. As funcionalidades e capacidades deste simulador foram testadas de diferentes formas, envolvendo a análise e avaliação da topologia de remessa, profundidade de decodificação, topologias de *cache* e previsão de desvios.

Assim, considerando uma arquitetura SMT que escalona instruções em um estilo *round-robin* e, considerando a execução simulada de programas do SPEC95 e, considerando as diferentes características de *hardware* utilizadas em cada caso aqui avaliado, diversas novas conclusões, que são contribuições diretas deste trabalho foram obtidas, das quais pode-se destacar:

- As estações de reserva devem ser organizadas em uma topologia distribuída por tarefa. A justificativa principal para esta conclusão não é somente o ganho mínimo de desempenho de 2,8% sobre outra arquitetura SMT equivalente com topologia compartilhada, mas sim o benefício bem conhecido de que a topologia distribuída é mais simples de ser implementada. Assim, não existe justificativa para se usar a topologia compartilhada em uma arquitetura SMT do tipo aqui apresentado.

- Para se obter mais do que 96% do máximo desempenho de uma arquitetura SMT, em todos os casos simulados, não foi necessário decodificar mais do que 2 instruções de cada tarefa por ciclo. Esta conclusão permite simplificar 50% da lógica de decodificação em uma arquitetura SMT-4 e 75% sobre uma SMT-8.

- A utilização de um espaço vetorial de tarefas (*evt*) menor do que o número de tarefas que estão sendo executadas na arquitetura SMT pode reduzir o desempenho ou até tornar impossível a execução das aplicações. Esta situação ocorre devido aos conflitos de endereçamento, os quais aumentam a quantidade de faltas na cache. Conforme a configuração da *cache* de instruções, o processador não é capaz de executar instruções úteis, permanecendo quase que todo o tempo realizando substituição de blocos da *cache* em função de tão elevado número de faltas. Quando as topologias de *cache* são organizadas apropriadamente, elas podem prover razoável ganho de desempenho umas sobre as outras, alcançando mais de 9% em uma arquitetura SMT-4 e mais de 30% em uma arquitetura SMT-8

- A associatividade da *cache* pode contribuir mais significativamente para o aumento d desempenho do que a separatividade, apesar de implicar em mais alto custo de *hardware*. Isto ocorre porque a associatividade possibilita que um maior número de tarefas possa acessar a mesma *cache* e a ociosidade seja assim reduzida. Ao contrário, a

separatividade não permite que uma cache dedicada a uma tarefa seja utilizada por outra. Também, a modularidade pode aumentar o desempenho, dependendo da largura do barramento de busca. Isto será verdade se o barramento que ficar disponível para cada módulo for suficiente para explorar o bloco básico das tarefas

- Se a razão entre o número de tarefas e o número de recursos (tarefas/recursos) é alta, o paralelismo entre tarefas é suficiente para ocupar os recursos do *pipeline*. Neste caso, pouca especulação é realizada em cada tarefa e a taxa de acerto do mecanismo de previsão não terá muita importância, permitindo assim o uso de previsores simples. Caso contrário, se a razão tarefas/recursos é baixa, existe muito *hardware* para poucas tarefas. Neste caso, o *pipeline* tem chance de executar uma maior quantidade de paralelismo no nível de instrução, aumentando a especulação e necessitando assim de um previsor mais eficiente para obter maior desempenho final. De uma forma geral, quanto maior é o paralelismo no nível de instrução, mais significativo será o efeito da melhoria do previsor. Em nossos experimentos, o maior benefício alcançado com o aumento de 5% na taxa de acerto da previsão foi de 18% para uma arquitetura SMT-8.

- Com o aumento do *hardware*, os programas mais sensíveis à previsão devem dispor de mecanismos mais eficientes que permitam tirar melhor proveito dos recursos disponíveis. Melhor ainda, podemos dizer que eles devem errar menos para ocupar mais o *pipeline*, já que este foi aumentado. Por exemplo, na arquitetura superescalar pequena, o programa *ijpeg* superou o desempenho do *mgrid* quando a taxa de acerto atingiu próximo de 75%, e o programa *li* superou o desempenho do *fpppp* quando a taxa de acerto chegou a 55%. Na arquitetura superescalar grande estes valores foram 85% e 78%, respectivamente.

- Em muitas situações, o mesmo ganho de desempenho alcançado por uma arquitetura quando se melhora o previsor de desvios pode ser alcançado aumentando o *pipeline* mesmo que com previsor mais simples. Em nossos experimentos, temos observado esta característica nas arquiteturas superescalares. Neste caso, uma arquitetura superescalar pequena com 100% de acerto de previsão obtém o mesmo desempenho que uma arquitetura superescalar grande com 80% de acerto. Assim, em vez de melhorarmos o previsor em uma arquitetura pequena, poderíamos substituí-la por uma arquitetura maior e usar um previsor mais simples. Obviamente, nem sempre é conveniente trocar a complexidade do previsor pela complexidade do aumento do *pipeline*, que por sua vez causa aumento de complexidade em todos os estágios do *pipeline*.

O simulador SEMPRE básico foi estendido para incorporar as características específicas da arquitetura. Apesar de ainda não ter sido implementadas todas as suas características propostas no projeto original, a arquitetura apresentou resultados satisfatórios no que diz respeito às vantagens do sistema de escalonamento e pré-busca de processos. Estes resultados validam a idéia principal do presente projeto. Uma observação importante a ser feita é que a arquitetura SMT, representada pelo simulador de base, utiliza uma única estrutura, a RUU, para representar as duas estruturas inicialmente propostas para a arquitetura SEMPRE: a fila de remessa (FRm) e a fila de reordenação (FRd). Esta mudança no entanto, não causou substancial alteração no funcionamento lógico da arquitetura proposta, nem tão pouco a desviou de seu objetivo principal. Simplesmente foi incorporada por uma questão de facilidade e operacionalização.

Através do mecanismo de escalonamento e pré-busca de processos por *hardware*, a arquitetura SEMPRE obtém diversas vantagens sobre uma outra arquitetura SMT equivalente, onde se pode destacar:

- Desconsiderando a influência do sistema operacional, conclui-se que o desempenho da arquitetura SEMPRE é sempre superior ao de uma arquitetura SMT tradicional, podendo atingir ganhos acima de 9% para 4 *slots* e acima de 8% para 8 *slots*. Este benefício é causado pela pré-busca de processos, que permite trocar um processo bloqueado por outro que esteja pronto na *cache* de instruções.

- Considerando a existência de um sistema operacional que realiza o escalonamento de processos em uma arquitetura SMT tradicional, conclui-se que ganho da arquitetura SEMPRE sobre esta arquitetura atinge valores em torno de 11% a 22%. Os índices de ganhos para as arquiteturas de 8 *slots*, apesar de menores, superam valores de 8%. Este benefício é causado por um melhor aproveitamento das fatias de tempo dos processos, que quando gerenciadas pelo sistema operacional não são devidamente aproveitadas devidos as constantes faltas na *cache* de instruções.

Não se pode esquecer que o objetivo maior da arquitetura SEMPRE é o de possibilitar o aproveitamento do paralelismo existente nos processos que normalmente são executados concorrentemente nas estações de trabalho e servidores de rede. As simulações não somente mostram que isto é possível, mas também mostram que o escalonamento de *hardware* provê maior desempenho para a arquitetura SEMPRE do que em outras arquiteturas tradicionalmente escalonadas pelo sistema operacional.

Uma observação final é que, apesar da implementação de uma arquitetura multi-tarefas simultâneas requerer a utilização de uma quantidade muito grande de *hardware*, muitas vezes inviabilizada nos dias de hoje devido às limitações de espaço, temperatura e freqüência, acredita-se que num futuro próximo existirá a disponibilidade de bilhões de transistores para a confecção de circuitos úteis. Além disso, se o custo desta máquina parece ser inviável economicamente, deve-se considerar que um único processador poderá substituir a utilização de muitos outros.

7.1 Trabalhos Futuros

A arquitetura SEMPRE, segundo a proposta original e completa, é bastante complexa e requer muito esforço de pesquisa e desenvolvimento. O trabalho aqui apresentado é a ponta de um "*iceberg*" que levará ainda alguns anos para ser desvendado. Para a conclusão da primeira versão completa do simulador da arquitetura, há a necessidade de pelo menos mais uma outra tese de doutorado e uma dissertação de mestrado. Para o aprimoramento da arquitetura e extensão das suas capacidades funcionais, após a conclusão da primeira versão completa, acreditamos que sejam necessárias mais duas dissertações de mestrado. Os temas sugeridos para estes trabalhos são descritos a seguir:

1. **Implementação de Primitivas de Gerenciamento de Processos (dissertação de mestrado):**

As primitivas *create*(), *kill*(), *suspend*() e *resume*() precisam ser implementadas para dar suporte para o desenvolvimento do sistema operacional. Como metas básicas, as atividades básicas deste trabalho serão duas: a) definição de instruções em código *ss*

(*SimpleScalar*) que representem as referidas primitivas, através da devida alteração do arquivo “machine.def” e b) adaptação do estágio de busca para a execução das respectivas primitivas.

2. **Implementação de um Sistema Operacional (tese de doutorado):**

Este trabalho é mais complexo e requer um maior tempo de dedicação e programação. O objetivo principal deste trabalho é desenvolver uma aplicação que execute pelo menos duas funções básicas de um sistema operacional: a) atender solicitações básicas do usuário, como por exemplo, atender aos comandos de *exec* (solicitação para executar um programa), *ps* (solicitação para ver quais processos estão sendo executados) e *kill* (matar determinado processo) e b) gerenciar a execução dos processos presentes no sistema, utilizando para isto as primitivas providas pela arquitetura.

3. **Implementação de Primitivas de Troca de Mensagens (dissertação de mestrado):**

Estas primitivas não foram especificadas no projeto teórico, que visa a execução de processos independentes e não comunicantes, embora seja de vital importância para expandir as capacidades funcionais da arquitetura e permitir uma maior abrangência na execução das aplicações. Este trabalho terá como meta básica o desenvolvimento de uma biblioteca em linguagem C que possa ser incorporada nas aplicações comunicantes, permitindo a utilização das primitivas básicas de envio e recepção de mensagens. Deverá ser garantido que esta biblioteca possa ser compilada para o formato de código *ss*.

4. **Desenvolvimento de Aplicações Paralelas de Avaliação e Desempenho (dissertação de mestrado):**

Este trabalho visa adaptar um conjunto de aplicativos paralelos comumente utilizados para resolver problemas complexos, tais como programas de tratamento de imagens, e que possam ser utilizados como programas de avaliação e desempenho. Estes programas deverão fazer uso das primitivas de troca de mensagens propostas no item 3 e posteriormente serão compilados para o formato de código *ss*.

Anexo 1 Grupos de Programas de Avaliação para 4 *slots*

4 Processos	1	*swim, perl, mgrid, ijpeg*
	2	*wave5, gcc, fpppp,* li
	3	applu, *compress, turb3d, vortex*
	4	*mgrid, ijpeg, wave5, gcc*
	5	*fpppp,* li, applu, *compress*
	6	*turb3d, vortex, hydro2d,* m88ksim
6 Processos	1	*swim, perl, mgrid, ijpeg, wave5, gcc*
	2	*wave5, gcc, fpppp, li, applu, compress*
	3	applu, *compress, turb3d, vortex, hydro2d,* m88ksim
	4	*mgrid, ijpeg, wave5, gcc, fpppp,* li
	5	*fpppp, li, applu, compress, turb3d, vortex,*
	6	*turb3d, vortex, hydro2d, m88ksim, swim, li*
8 Processos	1	*swim, perl, mgrid, ijpeg, wave5, gcc, fpppp, li*
	2	*wave5, gcc, fpppp, li, applu, compress, turb3d, vortex*
	3	applu, *compress, turb3d, vortex, hydro2d,* m88ksim, *swim,* li
	4	*mgrid, ijpeg, wave5, gcc, fpppp, li, applu, compress*
	5	*fpppp, li, applu, compress, turb3d, vortex, hydro2d, m88ksim*
	6	*turb3d, vortex, hydro2d, m88ksim, swim, li, swim, perl*
10 Processos	1	*swim, perl, mgrid, ijpeg, wave5, gcc, fpppp, li, applu, compress*
	2	*wave5, gcc, fpppp, li, applu, compress, turb3d, vortex, hydro2d,* m88ksim
	3	applu, *compress, turb3d, vortex, hydro2d,* m88ksim, *swim, li, swim, perl*
	4	*mgrid, ijpeg, wave5, gcc, fpppp, li, applu, compress, turb3d, vortex*
	5	*fpppp, li, applu, compress, turb3d, vortex, hydro2d, m88ksim, swim, li*
	6	*turb3d, vortex, hydro2d, m88ksim, swim, li, swim, perl, mgrid, ijpeg*
12 Processos	1	*swim, perl, mgrid, ijpeg, wave5, gcc, fpppp, li, applu, compress, turb3d, vortex*
	2	*wave5, gcc, fpppp, li, applu, compress, turb3d, vortex, hydro2d,* m88ksim, *swim,* li
	3	applu, *compress, turb3d, vortex, hydro2d,* m88ksim, *swim, li, swim, perl, mgrid, ijpeg*
	4	*mgrid, ijpeg, wave5, gcc, fpppp, li, applu, compress, turb3d, vortex, hydro2d, m88ksim*
	5	*fpppp, li, applu, compress, turb3d, vortex, hydro2d, m88ksim, swim, li, , swim, perl*
	6	*turb3d, vortex, hydro2d, m88ksim, swim, li, swim, perl, mgrid, ijpeg, wave5, gcc*

Anexo 2 Grupos de Programas de Avaliação para 8 *slots*

8 Processos	1	*swim, perl, mgrid, ijpeg, wave5, gcc, fpppp, li*
	2	*applu, compress, turb3d, vortex, hydro2d, m88ksim, swim, li*
	3	*wave5, gcc, fpppp, li, applu, compress, turb3d, vortex*
	4	
10 Processos	1	*swim, perl, mgrid, ijpeg, wave5, gcc, fpppp, li, applu, compress*
	2	*applu, compress, turb3d, vortex, hydro2d, m88ksim, swim, li, swim, perl*
	3	*wave5, gcc, fpppp, li, applu, compress, turb3d, vortex, hydro2d, m88ksim*
12 Processos	1	*swim, perl, mgrid, ijpeg, wave5, gcc, fpppp, li, applu, compress, turb3d, vortex*
	2	*applu, compress, turb3d, vortex, hydro2d, m88ksim, swim, li, swim, perl, mgrid, ijpeg*
	3	*wave5, gcc, fpppp, li, applu, compress, turb3d, vortex, hydro2d, m88ksim, swim, li*
14 Processos	1	*swim, perl, mgrid, ijpeg, wave5, gcc, fpppp, li, applu, compress, turb3d, vortex, hydro2d, m88ksim*
	2	*applu, compress, turb3d, vortex, hydro2d, m88ksim, swim, li, swim, perl, mgrid, ijpeg, wave5, gcc*
	3	*wave5, gcc, fpppp, li, applu, compress, turb3d, vortex, hydro2d, m88ksim, swim, li, swim, perl*
16 Processos	1	*swim, perl, mgrid, ijpeg, wave5, gcc, fpppp, li, applu, compress, turb3d, vortex, hydro2d, m88ksim, swim, li*
	2	*applu, compress, turb3d, vortex, hydro2d, m88ksim, swim, li, swim, perl, mgrid, ijpeg, wave5, gcc, fpppp, li*
	3	*wave5, gcc, fpppp, li, applu, compress, turb3d, vortex, hydro2d, m88ksim, swim, li, swim, perl, mgrid, ijpeg*

Anexo 3 Publicações

[GON 97b] GONÇALVES, R. A. L.; NAVAUX, P. O. A. Arquitetura Superescalar com Escalonamento Estático com Permutação de Tipos de Instruções. In: CONFERÊNCIA LATINO AMERICANA DE INFORMÁTICA, CLEI Panel 97, 23., 1997, Valparaiso, Chile. **Anais…** Valparaiso: CLEI/UTFSM, 1997.

[GON 98a] GONÇALVES, R. A. L.; NAVAUX, P. O. A. SEMPRE: Uma Arquitetura SuperEscalar com Múltiplos PRocessos em Execução. In: SIMPÓSIO BRASILEIRO DE ARQUITETURAS DE COMPUTADORES - PROCESSAMENTO DE ALTO DESEMPENHO, SBAC-PAD, 10., 1998, Búzios, Brazil. **Anais…** Rio de Janeiro: SBC/COPPE/UFRJ, 1998.

[GON 98b] GONÇALVES, R. A. L.; NAVAUX, P. O. A. Proposta de uma Arquitetura *Multi-Threading* Voltada para Sistemas Multi-Processos. In: CONGRESSO ARGENTINO DE CIÊNCIA DA COMPUTAÇÃO, 1998, CACIC, 4., Neuquén, Argentina. **Anais…** [S.l.:s.n.], 1998.

[GON 98c] GONÇALVES, R. A. L.; NAVAUX, P. O. A. A Utilização de Buffer de Remessa Unificado em Arquiteturas Superescalares com Fluxos Balanceados de Instruções. In: CONFERÊNCIA LATINO AMERICANA DE INFORMÁTICA, CLEI Panel, 24., 1998, Quito, Equador. **Anais…** [S.l.:s.n.], 1998.

[GON 99] GONÇALVES, R. A. L. et al. *Process Prefetching for a Simultaneous Multithreaded Architecture*. In: SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, SBAC-PAD, 11., 1999, Natal, Brasil. **Proceedings…** Porto Alegre-RS: SBC/II/UFRGS, 1999.

[SAG 99b] SAGULA, R. L.; GONÇALVES, R. A. L.; DIVÉRIO, T.; NAVAUX, Philippe O. A. A Utilização de Modelagem Analítica no Projeto de Arquiteturas de Processadores. In: CONFERÊNCIA LATINO AMERICANA DE INFORMÁTICA, CLEI Panel, 25., 1999, Assunção, Paraguai. **Anais…** [S.l.:s.n.], 1999.

[GON 00] GONÇALVES, R. A. L.; AYGUADÉ, E. P.; VALERO, M.; NAVAUX, P. O. A. A Simulator for SMT Architecture: Evaluating Instruction Cache Topologies, In: SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, SBAC-PAD'00, 12., 2000, São Pedro, Brasil. **Proceedings…** São Carlos-SP: SBC/UFSCar, 2000.

[GON 01] GONÇALVES, R. A. L. et al. Evaluating the Effects of Branch Prediction Accuracy on the Performance of SMT Architectures, In: EUROMICRO WORKSHOP ON PARALLEL AND DISTRIBUTED PROCESSING, EUROMICRO PDP, 9., 2001, Mantova, Italy. **Proceedings…** [S.l.:s.n.], 2001.

Bibliografia

[ALV 90] ALVERSON, R. et al. The Tera Computer System. In: INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, 4., 1990, Amsterdam. **Proceedings…** [S.l.:s.n.], 1990.

[AND 93] ANDO, H. et al. Speculative Execution and Reducing Branch Penalty in a Parallel Issue Machine. In: IEEE INTERNATIONAL CONFERENCE ON COMPUTER DESIGN: VLSI IN COMPUTERS & PROCESSORS, 1993, Cambridge, Massachusetts. **Proceedings…** [S.l.:s.n.],1993.

[AND 95] ANDERSON, D.; SHANLEY, T. **Pentium Processor System Architecture**. 2nd ed. Massachusetts: Addison-Wesley, 1995. 433p.

[BRA 91] BRAY, B. K.; FLYNN, M. J. Strategies for Branch Target Buffers. In: INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 24., 1991, New Mexico. **Proceedings…** [S.l.:s.n.], 1991.

[BUR 97] BURGER, D.; AUSTIN, T. M. **The SimpleScalar Tool Set**: Version 2.0. Madison: University of Wisconsin, 1997. (Technical Report, n.1342).

[BUJ 99] BURNS, J.; GAUDIOT J.-L. Exploring the SMT Fetch Bottleneck. In: WORKSHOP ON MULTITHREADED EXECUTION, ARCHITECTURE AND COMPILATION, MTEAC, 1999, Orlando, Florida. **Proceedings…** [S.l.:s.n.], 1999.

[BUT 91] BUTLER, M. et al. Single Instruction Stream Parallelism Is Greater Than Two. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, ISCA, 18., 1991, Toronto, Canada. **Proceedings…** [S.l.:s.n.], 1991.

[BUT 92] BUTLER, M.; PATT, Y. An Investigation of the Performance of Various Dynamic Scheduling Techniques. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 25., 1992, Portland, Oregon. **Proceedings…** [S.l.:s.n.], 1992.

[CHA 91] CHANG, P.P. et al. Comparing Static and Dynamic Code Scheduling for Multiple-Instruction-Issue Processors. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 24., 1991, New Mexico. **Proceedings…** [S.l.:s.n.], 1991.

[CHD 94] CHAKRAVARTY, D.; CANNON, C. **PowerPC:** Concepts, Architecture, and Design. USA: McGraw-Hill,1994. 363p. Ranade Workstations Series.

[CHE 96] CHAVES FILHO, E. M. et al. Uma Arquitetura Superescalar com Múltiplos Fluxos de Instruções. In: SIMPÓSIO BRASILEIRO DE ARQUITETURAS DE COMPUTADORES E COMPUTAÇÃO DE ALTO DESEMPENHO, SBAC-PAD, 8., Recife. **Anais...** Recife: SBC/UFPE, 1996.

[DIE 95] DIEP, T.A.; NELSON, C.; SHEN, J.P. Performance Evaluation of the PowerPC 620 Microarchitecture. In: ANNUAL INTERNATIONAL

SYMPOSIUM ON COMPUTER ARCHITECTURE, ISCA, 22., 1995, Santa Margherita Ligure, Italy. **Proceedings…** [S.l.:s.n.], 1995.

[DWY 92] DWYER, H.; TORNG, H.C. An Out-of-Order SuperScalar Processor with Speculative Execution and Fast, Precise Interrupts. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 25., 1992, Portland, Oregon. **Proceedings…** [S.l.:s.n.], 1992.

[FER 92] FERNANDES, E. S. T.; SANTOS, A. D. Arquiteturas Super Escalares: Detecção e Exploração do Paralelismo de Baixo Nível. In: ESCOLA DE COMPUTAÇÃO, 7., 1992, Gramado-RS. **Anais…** [S.l.:s.n.], 1992.

[ESP 97] ESPASA, R.; VALERO, M. Simultaneous Multithreaded Vector Architecture: Merging ILP and DLP for High Performance. In: HIGH PERFORMANCE COMPUTING, HiPC, 1997, India. **Proceedings…** [S.l.:s.n.], 1997.

[FRA 94] FRANKLIN, M.; SMOTHERMAN, M. A Fill-Unit Approach to Multiple Instruction Issue. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 27., 1994, San Jose, CA/USA. **Proceedings…** [S.l.:s.n.], 1994.

[GOL 94] GOLDEN, M.; MUDGE, T. A Comparison of Two *Pipeline* Organizations. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 27., 1994, San Jose, CA/USA. **Proceedings…** [S.l.:s.n.], 1994.

[GON 97a] GONÇALVES, R. A. L.; NAVAUX, P. O. A. **Escalonamento de Código para Arquiteturas SuperEscalares**: Trabalho Individual. Porto Alegre: CPGCC da UFRGS, 1997.

[GUL 96] GULATI, M.; BAGHERZAD, N. Performance Study of a Multithreaded Superscalar Microprocessor. In: HIGH PERFORMANCE AND PARALLEL COMPUTER ARCHITECTURE, HPCA, 1996, California. **Proceedings…** [S.l.:s.n.], 1996.

[HEN 94] HENNESSY, J.; PATTERSON, D. A. **Computer Architecture**: A Quantitative Approach. San Mateo, CA: Morgan Kaufmann, 1994.

[HIL 98] HILY, S.; SEZNEC, A. **Out-of-Order Execution May Not Be Cost-Effctive on Processors Featuring Simultaneous Multithreading**. Rennes: IRISA (Institut de Recherche en Informatique et Systèmes Aléatoires), 1998. (Technical Report, n.1179).

[HIR 92] HIRATA, H. et al. An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, ISCA, 19., 1992. **Proceedings…** [S.l.:s.n.], 1992.

[JAC 96] JACOB, BRUCE L. et al. An Analytical Model for Designing Memory Hierarchies. **IEEE Transactions on Computer**, New York, v.45, n.10, Oct. 1996.

[JOH 91] JOHNSON, M. **Superscalar Microprocessor Design**. Englewood Cliffs, New Jersey: Prentice Hall, 1991. 288p. Series in Innovative Technology.

[JOU 89] JOUPPI, N. P.; WALL, D. W. **Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines**. Palo Alto: Digital Western Research Laboratory,1989. Research Report.

[JOU 95a] JOURDAN, S.; SAINRAT, P.; LITAIZE, D. An Investigation of the Performance of Various Instruction-Issue Buffer Topologies. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 28., 1995, Ann Arbor, Michigan. **Proceedings…** [S.l.:s.n.], 1995.

[JOU 95b] JOURDAN, S.; SAINRAT, P.; LITAIZE, D. Exploring Configurations of Functional Units in an Out-of-Order SuperScalar Processor. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, ISCA, 22., 1995. **Proceedings…** [S.l.:s.n.], 1995.

[JOA 95] JOÃO JUNIOR, M.; AUDE, J.S. Compactação Local de Código para um SPARC Superescalar. In: SIMPÓSIO BRASILEIRO DE ARQUITETURAS DE COMPUTADORES E COMPUTAÇÃO DE ALTO DESEMPENHO, SBAC-PAD, 7., Canela. **Anais...**[S.l.:s.n.], 1995.

[KAN 97] KANT, L.; SANDERS, W.H. Analysis of the Distribution of Consecutive Cell Losses in an ATM Switch Using Stochastic Activity Networks. **International Journal of Computer Systems Science & Engineering on ATM Switching**, [S.l.], v.12, n.2, Mar. 1997.

[KES 99] KESSLER, RICHARD E. The Alpha 21264 microprocessor. **IEEE Micro**, [S.l.], v.19, n.2, Mar./Apr. 1999.

[KRI 97] KRISHNA, B. H.; GOVINDARAJAN, R. Classification and Performance Evaluation of Simultaneous Multithreaded Architectures. In: HIGH PERFORMANCE COMPUTING, HiPC, 1997, India. **Proceedings…** [S.l.:s.n.], 1997.

[LAU 94] LAUDON, J. et al. Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations. In: INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, ASPLOS, 1994. **Proceedings…** [S.l.:s.n.], 1994.

[LEE 84] LEE, J. K. F.; SMITH, A. J. Branch Prediction Strategies and Branch Target Buffer Design. **IEEE Computer Magazine**, [S.l.], 1984.

[LEE 95] LEE, D. et al. Instruction Cache Fetch Policies for Speculative Execution. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, ISCA, 22., 1995. **Proceedings…** [S.l.:s.n.], 1995.

[LIN 98] LINDEMANN, C. **Performance Modelling with Deterministic and Stochastic Petri Nets**. [S.l.]: John Wiley and Sons, 1998.

[LIP 96] LIPASTI, M.H.; SHEN, J.P. Exceeding the Dataflow Limit via Value Prediction. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 29., 1996, Paris. **Proceedings…** [S.l.:s.n.], 1996.

[LOI 96] LOIKKANEN, M.; BAGHERZADEH, N. A Fine-Grain Multithreading Superscalar Architecture. In: PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, PACT, 1996. **Proceedings…** [S.l.:s.n.], 1996.

[LO 97a] LO, J. et al. Tuning Compiler Optimizations for Simultaneous Multithreading. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 30., 1997. **Proceedings…** [S.l.:s.n.], 1997.

[LO 97b] LO, J. et al. **Software-Directed Register Deallocation for Simultaneous Multithreaded Processors**. Washington: University of Washington, 1997. (Technical Report, #UW-CSE-97-12-01).

[LO 98] LO, J.L. et al. An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, ISCA, 25., 1998. **Proceedings…** [S.l.:s.n.], 1998.

[MAR 84] MARSAN, M.; BALDO, G.; CONTE, G. A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems. **ACM Transactions on Computer Systems**, [S.l.], 1984.

[MAD 99] MADÓN, D.; SÁNCHES; E.; MONNIER, S. A Study of a Simultaneous Multithreaded Processor Implementation. In: EUROPAR CONFERENCE, 1999. **Proceedings…** [S.l.:s.n.], 1999..

[MCF 93] MCFARLING, S. **Combining Branch Predictors**. Palo Alto: Digital Western Research Laboratory, 1993. (Technical Note, NT-36).

[MEL 88] MELVIN, S.W.; SHEBANOW, M.C.; PATT, Y.N. Hardware Support for Large Atomic Units in Dynamically Scheduled Machines. In: ANNUAL WORKSHOP ON MICROPROGRAMMING AND MICROARCHITECTURE, MICRO, 21., 1988, San Diego, California. **Proceedings…** [S.l.:s.n.], 1988.

[NEM 91] NEMIROVSKY, M. D.; BREWER, F.; WOOD, R. C. DISC: Dynamic Instruction Stream Computer. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 24., 1991, Albuquerque, USA. **Proceedings…** [S.l.:s.n.], 1991.

[MIP 95] MIPS. **R10000 Microprocessor User's Manual**: Version 1.0. Mountain View, California: MIPS Technologies, 1995.

[NEM 98] NEMIROVSKY, M.; YAMAMOTO, W. Quantitative Study od Data Caches on a Multistreamed Architecture. In: WORKSHOP ON MULTITHREADED EXECUTION, ARCHITECTURE AND COMPILATION, MTEAC, 1998, Las Vegas, Nevada. **Proceedings…** [S.l.:s.n.], 1998.

[PAL 97] PALACHARLA, S.; JOUPPI, N. P.; SMITH, J. E. Complexity-Effective Superscalar Processors. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, ISCA, 24., 1997, Denver, USA. **Proceedings…** [S.l.:s.n.], 1997.

[PAR 91] PARK, W. W. et al. Performance Advantages of Multithreaded Processors. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, 1991. **Proceedings…** [S.l.:s.n.], 1991.

[PIE 96] PIERCE, J.; MUDGE, T. Wrong-Path Instruction Prefetching. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 29., 1996, Paris. **Proceedings…** [S.l.:s.n.], 1996.

[RIN 98] RINKER, R.E.; TAMMA, R.; NAJJAR, W. Evaluation of Cache Assisted Multithreaded Architecture. In: WORKSHOP ON MULTITHREADED EXECUTION, ARCHITECTURE AND COMPILATION, MTEAC, 1998, Las Vegas, Nevada. **Proceedings…** [S.l.:s.n.], 1998.

[ROB 96] ROBBINS, K. A. AND ROBBINS, S. **Practical Unix Programming**: a guide to concurrency, communication, and multithreading. [S.l.]: Prentice-Hall, 1996.

[ROT 96] ROTENBERG, E.; BENNETT, S.; SMITH, J.E. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 29., 1996, Paris. **Proceedings…** [S.l.:s.n.], 1996.

[SAA 90] SAAVEDRA-BARRERA, R. H.; CULLER, D. E.; VON EICKEN, T. Analysis of Multithreaded Architectures for Parallel Computing. In: ANNUAL ACM SYMPOSIUM ON PARALLEL ALGORITHMS AND ARCHITECTURE, 2., 1990, Crete, Greece. **Proceedings…** [S.l.:s.n.], 1993.

[SAG 99] SAGULA, R. L.; DIVÉRIO, T.; NAVAUX, PHILIPPE O. A. **Modelagem Analítica**: Trabalho Individual. Porto Alegre: PPGC da UFRGS, 1999.

[SAI 93] SAINI, A. Design of the Intel Pentium Processor. In: IEEE INTERNATIONAL CONFERENCE ON COMPUTER DESIGN: VLSI IN COMPUTERS & PROCESSORS, 1993, Cambridge, Massachusetts. **Proceedings…** [S.l.:s.n.], 1993.

[SAN 97] SANTOS, RAFAEL R.; NAVAUX, P. O. A. Mecanismo de Busca Especulativa de Múltiplos Fluxos de Instruções. In: SIMPÓSIO BRASILEIRO DE ARQUITETURAS DE COMPUTADORES E COMPUTAÇÃO DE ALTO DESEMPENHO, SBAC-PAD, 9., 1997, Campos do Jordão. **Anais...** [S.l.:s.n.], 1997.

[SAN 00] SANTOS, T. G. S. **Analisando Mecanismos de Pré-Busca na Hierarquia de Memória dos Microprocessadores Superescalares**. Porto Alegre: PPGC da UFRGS, 2000. Dissertação de Mestrado.

[SHA 98] SNAVELY, A. et al. Multi-processor Performance on the Tera MTA. In: INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, 1998. **Proceedings…** [S.l.:s.n.], 1998.

[SIG 96] SIGMUND, U.; UNGERER, T. Identifying Bottlenecks in a Multithreaded Superscalar Microprocessor. In: EUROPAR CONFERENCE, 1996, Lyon. **Proceedings…** [S.l.:s.n.], 1996.

[SIG 99] SIGMUND, U.; UNGERER, T. Memory Hierarchy Studies of Multimedia-enhanced Simultaneous Multithreaded Processors for MPEC-2 Video Decompression. In: WORKSHOP ON MULTITHREADED EXECUTION, ARCHITECTURE AND COMPILATION, MTEAC, 2000, Toulouse. **Proceedings…** [S.l.:s.n.], 2000.

[SMI 81] SMITH, J. E. A Study of Branch Prediction Strategies. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, ISCA, 8., 1981, Minneapolis, Minnesota. **Proceedings…** [S.l.:s.n.], 1981.

[SMI 95] SMITH, J.E.; SOHI, G.S. The Microarchitecture of SuperScalar Processors. **Proceedings of the IEEE,** [S.l.], v.83, n.12, Dec. 1995.

[SOH 90] SOHI, G. S. Instruction Issue Logic for High Performance, Interruptible, Multiple Functional Unit, Pipelined Computers. **IEEE Transactions on Computers**, New York, v.39, n.3, Mar. 1990.

[THE 94] THEKKATH, R.; EGGERS, S.J. The Effectiveness of Multiple *Hardware* Contexts. In: INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, ASPLOS, 1994. **Proceedings…** [S.l.:s.n.], 1994.

[TOM 67] TOMASULO, R.M. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. **IBM Journal of Research and Development**. [S.l.], v.11, Jan. 1967.

[TOR 99] TORRANT, M. et al. A Simultaneous Multithreading Simulator. In: HIGH PERFORMANCE AND PARALLEL COMPUTER ARCHITECTURE, HPCA, 6., 1999. **Proceedings…** [S.l.:s.n.], 1999.

[TRA 92] TRAN, T.; WU, C. Limitation of Superscalar Microprocessor Performance. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 25., 1992, Portland, Oregon. **Proceedings…** [S.l.:s.n.], 1992.

[TSA 96] TSAI, J.-Y.; YEW, P.-C. The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation. In: PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, PACT, 1996. **Proceedings…** [S.l.:s.n.], 1996.

[TUL 95] TULLSEN, D. M. et al. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, ISCA, 22., 1995, Santa Margherita Ligure, Italy. **Proceedings…** [S.l.:s.n.], 1995.

[TUL 96] TULLSEN, D.M. et al. Exploiting Choice: Instruction Fetch and Issue on an Implementabela Simultaneous Multithreading Processor. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER

ARCHITECTURE, ISCA, 23., 1996, Philadelphia, PA. **Proceedings…** [S.l.:s.n.], 1996.

[TUR 93] TURCOTTE, L.H. **A Survey of Software Enviroments for Exploiting Network Computing Resources**. Mississippi: Center for Computational Field Simulation, 1993. Technical Report.

[UHT 97] UHT, A. K.; SINDAGI, V.; SOMANATHAN, S. Branch Effect Reduction Techniques. **IEEE Computer Magazine**, [S.l.], v.30, n.5, May 1997.

[ULT 96] ULTRASPARC. **User's Manual, UltraSPARC-I/UltraSPARC-II:** Revision 2.0. Mountain View, CA: Sun Microsystems, 1996.

[UNG 96] UNGERER, T.; SIGMUND, U. Evaluating A Multithreaded Superscalar Microprocessor Versus a Multiprocessor Chip. In: WORKSHOP ON PARALLEL SYSTEMS AND ALGORITHMS, Forschungszentrum Julich, PASA, 4., 1996. **Proceedings…** [S.l.:s.n.], 1996.

[WAL 93] WALL, D. W. **Limits of Instruction-Level Parallelism**. Palo Alto: Digital, Western Research Laboratory, 1993. Research Report.

[WAL 98] WALLACE, S.; CALDER, B.; TULLSEN, D. M. Threaded Multiple Path Execution. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, ISCA, 25., 1998. **Proceedings…** [S.l.:s.n.], 1998.

[WOL 96] WOLF, M. **High Performance Compilers for Parallel Computing**. Redwood City: Addison-Wesley, 1996. 570p.

[YAM 94] YAMAMOTO, W. et al. Performance Estimation of Multistreamed, Superscalar Processors. In: INTERNATIONAL CONFERENCE ON SYSTEMS SCIENCES, ICSS, 1994, Hawaii. **Proceedings…** [S.l.:s.n.], 1994.

[YEH 91] YEH, T.-Y.; PATT, Y.N. Two-Level Adaptative Training Branch Prediction. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 24., 1991, New Mexico. **Proceedings…** [S.l.:s.n.], 1991.

[YOU 95] YOUNG, C.; GLOY, N.; SMITH, M. D. A Comparative Analysis of Schemes for Correlated Branch Prediction. . In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, ISCA, 22., 1995, Santa Margherita Ligure, Italy. **Proceedings…** [S.l.:s.n.], 1995.

[YOU 96] YOUNG, J. L. **Por Dentro do Power PC.** São Paulo, Brasil: Berkeley, 1996. 313p.