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

Um Mecanismo de Busca Especulativa de Múltiplos Fluxos de Instruções

por

RAFAEL RAMOS DOS SANTOS

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

Prof. Dr. Philippe Olivier Alexandre Navaux
Orientador

Porto Alegre, dezembro de 1997

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Santos, Rafael Ramos dos

Um Mecanismo de Busca Especulativa de Múltiplos Fluxos de Instruções / por Rafael Ramos dos Santos. — Porto Alegre: CPGCC da UFRGS, 1997.

?? f.: il.

Dissertação (mestrado) — Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 1997. Orientador: Navaux, Philippe Olivier Alexandre.

1. Pipelining. 2. Paralelismo a Nível de Instruccão. 3. Arquiteturas Superescalares. 4. Multifluxo. I. Navaux, Philippe Olivier Alexandre. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Profa. Wrana Panizzi
Pró-Reitor de Pós-Graduação: Prof. José Carlos Ferraz Hennemann
Diretor do Instituto de Informática: Prof. Roberto Tom Price
Coordenadora do PPGC: Profa. Carla Maria Dal Sasso Freitas
Bibliotecária-chefe do Instituto de Informática: Zita Prates de Oliveira

Dedicado à minha família e em especial à minha esposa Tatiana.

Agradecimentos

Agradeço a todos os que tornaram possível a realização deste trabalho, sobretudo àqueles que estavam mais próximos e que me deram apoio e incentivo.

Agradeço ao CNPq pela concessão da bolsa de mestrado e à UFRGS pelo suporte técnico.

Agradeço ao professor Philippe Navaux, meu orientador, pelo incentivo e pela oportunidade de realizar este trabalho e muitos outros. Agradeço também aos professores Eliseu Monteiro Chaves Filho, Sergio Bampi e João Netto por todas as conversar e dicas a respeito da condução deste trabalho.

Agradeço aos amigos e colegas pelos momentos de reflexão e descontração e ao pessoal da biblioteca pela paciência durante a inúmeras consultas quanto às regras de normatização.

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

Este trabalho apresenta um novo modelo de busca especulativa de múltiplos fluxos de instruções em arquiteturas superescalares. A avaliação de desempenho de uma arquitetura superescalar com esta característica é também apresentada como forma de validar o modelo proposto e comparar seu desempenho frente a uma arquitetura superescalar real.

O modelo em questão pretende eliminar a latência de busca de instruções introduzida pela ocorrência de comandos de desvio em pipelines superescalares. O desempenho de uma arquitetura superescalar dotada de escalonamento dinâmico de instruções, previsão de desvios e execução especulativa é bastante inferior ao desempenho máximo teórico esperado. Como demonstrado em outros trabalhos, isto ocorre devido às constantes quebras de fluxo, derivadas de instruções de desvio, e do conseqüente esvazimento da fila de instruções.

O emprego desta técnica permite encadear instruções pertencentes a diferentes fluxos lógicos, logo após a identificação de uma instrução de desvio, disponibilizando um maior número de instruções ao mecanismo de escalonamento dinâmico e diminuindo o número de ciclos com despacho nulo devido as quebras de fluxo.

Algumas considerações sobre a implementação do modelo descrito são apresentadas ao final do trabalho assim como sugestões para trabalhos futuros.

Palavras-chave: Pipelining, Paralelismo a Nível de Instrução, Arquiteturas Superescalares, Multifluxo

TITLE: "A MULTISTREAMED SPECULATIVE INSTRUCTION FETCH MECHANISM"

Abstract

This work presents a new model to fetch instructions along multiple streams in superscalar pipelines. Also, the performance evaluation of a superscalar architecture including this feature is presented in order to validate the model and to compare its performance with a real superscalar architecture.

The proposed technique intents to eliminate the instruction fetch latency introduced by branch instructions in superscalar pipelines. The performance delivered by a superscalar architecture which incorporate dynamic instruction scheduling, branch prediction and speculative execution is not the expected one which should be at least proportional to the number of functional units. Related works have shown that constant stream breaks caused by disruptions in the sequential flow of control reduce the amount of instructions into the instruction queue. This technique allows instruction fetch through different logic streams, as soon as the branch instruction has been detected during the fetch.

The scheduler needs a large instruction window to be able to schedule efficiently consequently the instructions window should hold as many instructions as possible to allow an efficient schedule. The improvement realized by the proposed scheme is to increase the size of the instruction window by putting there more instructions avoiding interruptions on the event of branch occurrence.

Some considerations about the implementation of this model are presented at final as well as suggestions to future works.

Keywords: Pipelining, Instruction-Level Parallelism, Superscalar Architectures

1 Introdução

Arquiteturas superescalares potencialmente são capazes de executar mais de uma instrução em um único ciclo de máquina. Dependências de controle possuem um papel determinante no desempenho das arquiteturas superescalares e são provocadas pelas instruções de desvio. A instrução que deve ser executada após uma instrução de desvio só é conhecida quando a instrução de desvio é completada criando-se assim uma relação de dependência. A conseqüência é uma redução do número médio de instruções despachadas por ciclo, e do desempenho da arquitetura [LEE 84, LIL 88, SAN 96].

Arquiteturas superescalares empregam técnicas de previsão de desvios juntamente com a execução especulativa de instruções para reduzirem a penalidade causada pelas dependências de controle na performance do processador. Estas técnicas até então foram utilizadas como a forma mais eficiente e econômica para contornar o problema dos desvios. O aumento na capacidade de integração e a redução no custo do hardware está possibilitando que novas técnicas, anteriormente inviabilizadas em função de seu custo, sejam consideradas como técnicas mais agressivas de exploração do paralelismo de instrução.

É essencial observar que a previsão de desvios possibilita apenas a continuidade da busca de instruções na presença de dependências de controle. Estes mecanismos não atacam a outra parte do problema, qual seja, o fato das dependências de controle tornarem a execução dependente do resultado de um desvio condicional [CHA 94]. Uma instrução de desvio condicional deve ser antes executada para que seja determinado se as instruções acessadas antecipadamente pelo mecanismo de previsão de desvios podem ser executadas.

Arquiteturas multifluxo vêm tomando importância cada vez maior na atual tendência das arquiteturas de processadores que exploram o paralelismo a nível de instrução. Uma arquitetura com múltiplos fluxos de instruções, é aquela em que múltiplas instruções, pertencentes a fluxos logicamente distintos, podem ser executadas simultaneamente [SOH 95]. Arquiteturas VLIW e superescalares buscam mais de uma instrução em cada acesso à memória, entretanto, as instruções buscadas pertencem a um único fluxo lógico de instruções, o que pode limitar o número de instruções em condição de serem despachadas no caso de existirem dependências entre estas instruções.

Muitas pesquisas vêm sendo desenvolvidas com o objetivo de encontrar novos paradigmas que sustentem a ordem de oito a dezesseis instruções executadas por ciclo [WAL 93]. Embora as arquiteturas superescalares possam executar mais de uma instrução por ciclo, deparam-se com sérios problemas na busca de tais índices.

As atuais arquiteturas paralelas de processadores apresentam duas deficiências. Primeiro, o modelo de execução de instruções adotado limita a disponibilidade de instruções independentes que podem ser executadas em paralelo. Segundo, os mecanismos de tratamento de dependências entre instruções ainda não são capazes de anular totalmente o custo destas dependências. Isto é particularmente crítico nos mecanismos dedicados às dependências de controle [CHA 94].

Enquanto os processadores incluem mais unidades funcionais para fornecerem um maior nível de paralelismo, um significativo limite para o grau de paralelismo é encontrado devido ao tamanho do bloco básico. Uma alternativa para a previsão de desvios é executar especulativamente ambas as ramificações do desvio, ao invés de prever se o desvio é tomado ou não-tomado, e anular a ramificação incorreta tão logo o resultado do desvio seja conhecido. Desta forma, a penalidade do desvio é eliminada, embora sejam necessários mais recursos computacionais [SAN 96]. Com o aumento da densidade

de integração e com as altas freqüências alcançadas esta alternativa compõe uma forte tendência.

Uma arquitetura multifluxo pode executar múltiplos fluxos de instruções simultaneamente. Tal facilidade oferece oportunidades para que as limitações acima mencionadas sejam minimizadas. Em arquiteturas deste tipo, instruções que executam em paralelo podem ser extraídas de diferentes fluxos. Assim, os fluxos de instruções agem como novas fontes de paralelismo, conferindo à arquitetura um desempenho potencialmente maior. Além disso, penalidades ocasionadas por dependências entre instruções podem ser mascaradas com o maior paralelismo proveniente da execução simultânea de múltiplos fluxos [CHA 96].

O *speed-up* de uma arquitetura superescalar é diretamente proporcional ao seu *IPC*, número médio de instruções executadas por ciclo. É esperado obter-se um *speed-up* proporcional ao número de unidades funcionais paralelas existentes no processador. No entanto, as constantes quebras de fluxo de controle, ocasionadas pela previsão de desvios como tomados, fazem com que a fila de instruções seja freqüentemente esvaziada e que não existam instruções para despacho. Logo, o *IPC* é reduzido drasticamente pelo esvaziamento da fila de instruções e o *speed-up* esperado não é alcançado em razão deste fator principal, como será provado posteriormente.

Neste trabalho é apresentado um modelo de busca especulativa de instruções que permite o encadeamento de instruções pertencentes a fluxos lógicos distintos evitando que a fila de instruções seja esvaziada e aumentando o número de instruções disponíveis ao escalonamento. Com esta filosofia pretende-se aumentar o *IPC* de arquiteturas superescalares possibilitando *speed-ups* mais elevados e próximos ao ideal.

Este mecanismo permite mascarar a latência de acesso às instruções que se encontram no destino alvejado por desvios condicionais sem aumentar drasticamente a complexidade do processador utilizando técnicas bastante conhecidas e empregadas em processadores comercializados atualmente. Pequenas modificações são propostas e os resultados obtidos são analisados e comparados com os resultados fornecidos por uma arquitetura superescalar real.

Questões e decisões sobre a viabilidade de implementação de tal mecanismo são consideradas e discutidas ao longo do trabalho a fim de permitir uma visão mais clara e concisa do novo modelo de arquitetura superescalar proposto e do seu desempenho.

No Capítulo 2 são descritas as características básicas de arquiteturas superescalares. Alguns conceitos são introduzidos e é apresentado um modelo de execução superescalar tipicamente encontrado nas arquiteturas atuais, a partir do qual foram realizados alguns dos experimentos empregados na análise de desempenho do modelo multifluxo.

No Capítulo 3 são abordados os fatores que limitam o desempenho de arquiteturas superescalares. São expostos alguns comentários a respeito de cada uma destas limitações concentrando-se na parte que aborda dependências de controle.

Algumas das técnicas mais empregadas para a redução do custo de desvios são apresentadas no Capítulo 4. Considerações sobre a implementação e desempenho são tecidas para cada uma das técnicas apresentadas. No Capítulo 5 são discutidos os trabalhos correlatos e no Capítulo 6 é apresentado o modelo de busca especulativa de instruções em múltiplos fluxos.

No Capítulo 7 são apresentados aspectos do ambiente experimental onde foram realizados os experimentos analisados no capítulo seguinte. O desempenho do modelo Multifluxo é analisado no Capítulo 8 onde vários gráficos são apresentados. Por fim, as conclusões são apresentadas no Capítulo 9.

2 Arquiteturas Superescalares

O termo escalar é empregado para distinguir a execução de um única instrução (manipulando operandos discretos) daquelas instruções (vetoriais) que desencadeiam a ativação em paralelo de múltiplos elementos de processamento, todos executando a mesma operação.

Existe um outro modelo de processamento no qual as instruções que serão executadas concorrentemente não precisam possuir o mesmo código de operação, isto é, elas não são vetoriais, embora múltiplas instruções (escalares) possam estar sendo executadas simultaneamente. Por esse motivo máquinas desse novo modelo de processamento são denominadas superescalares.

Como acontece no processamento vetorial, máquinas superescalares possuem múltiplas unidades funcionais que podem ser ativadas em paralelo, resultando num tempo de execução menor do que aquele requerido pelos modelos tradicionais de processamento. Por exemplo, se considerarmos que cada instrução de um processador é executada num ciclo de máquina e que n instruções foram executadas seqüencialmente, então é de se esperar um tempo de processamento b vezes menor se b instruções forem executadas por ciclo de máquina [FER 92].

Nos processadores superescalares, para que o paralelismo de baixo nível (i.e., paralelismo a nível de instrução) possa ser explorado, é necessário que haja um mecanismo eficiente de detecção e seleção de instruções que possam ser executadas em paralelo. Vários são os fatores que podem influenciar o funcionamento eficiente destes mecanismos, como: número e tipo das unidades funcionais, tempo de latência das instruções, relações de dependência entre instruções, tamanho da janela de instruções, mecanismos de previsão de desvios e execução especulativa, esquema de interconexão dos componentes, número de instruções despachadas por ciclo de máquina etc.

Os algoritmos destinados a detectar e selecionar instruções para execução em máquinas superescalares são denominados algoritmos de despacho ou algoritmos de escalonamento. Com o objetivo de reduzir a complexidade da unidade de controle, em alguns processadores, a tarefa de escalonamento é realizada durante a fase de geração de código objeto (compilação). Neste caso, o algoritmo de escalonamento é considerado estático.

O algoritmo de despacho pode modificar a ordem em que as instruções serão executadas, sem afetar a equivalência semântica do programa de aplicação, determinando as unidades funcionais que devem ser ativadas durante cada ciclo de máquina. Se ao contrário do caso anterior, este algoritmo for implementado em hardware, diretamente na unidade de controle, e a detecção do paralelismo for realizada em tempo de execução, este algoritmo é considerado dinâmico.

Quanto ao escalonamento das instruções, o algoritmo leva em conta as dependências de dados e as restrições no uso dos recursos da arquitetura.

O conflito no uso dos recursos do hardware e as dependências de dados entre as instruções são exemplos de fatores que limitam a taxa de ocupação das unidades funcionais, degradando o desempenho do processador superescalar. A inclusão de múltiplas cópias dos diversos tipos de unidades funcionais no processador, não garante a redução do tempo de processamento. As dependências de dados por exemplo, impossibilitam a execução antecipada de uma instrução cujo início está condicionado ao término de uma outra que a precede, impedindo deste modo, a ativação de uma unidade funcional ociosa.

2.1 Máquinas com Múltiplas Unidades Funcionais

Existem duas estratégias básicas para implementar unidades funcionais de um processador [FER 92]:

- Concentrar num único dispositivo funcional as funções que serão realizadas pelo processador, ou;
- Separar as funções em grupos, destinando para cada grupo um ou mais dispositivos autônomos especializados na execução das funções do grupo.

A primeira estratégia, apesar de muito empregada pelos projetistas, não oferece ganho de performance ao processador. No entanto, o custo final do produto é o fator determinante desta escolha. Os critérios que são considerados pelos projetistas estão relacionados com a redução do número de pinos, de barramentos de interconexão e com a utilização de unidades funcionais de propósito geral, facilmente encontradas no mercado.

Por outro lado, a segunda estratégia oferece mais oportunidades para aumentar o desempenho dos processadores que a utilizam. Através da separação/especialização, as instruções são executadas mais rapidamente do que num processador utilizando a primeira estratégia. Dois são os motivos que levam a esta conclusão:

1. a seqüência de sinais de controle para ativar uma unidade funcional especializada, é mais curta e menos complexa, requerendo intervalos de tempos menores (geração/transmissão) e por conseqüência reduzem o tempo total de processamento das máquinas com unidades autônomas;
2. a parte operacional (*data path*) de uma unidade especializada possui um número menor de componentes e por este motivo, o tempo de retardo do circuito (*circuit delay time*) é mais reduzido.

A presença de unidades funcionais separadas, permite a execução simultânea de múltiplas instruções. Enquanto que nas máquinas convencionais a unidade de controle processa serialmente as instruções de máquina, nestas máquinas o despacho de instruções para unidades funcionais pode ser abreviado desde que exista um mecanismo de verificação de dependências que examine as instruções antecipadamente (*look-ahead*) assim como a disponibilidade de unidades funcionais ociosas.

```
1. A := B * C;
2. D := E + F;
3. G := H and I;
4. J := K lshift L;
5. IF M > N
6.    THEN P := Q;
```

FIGURA 2.1 - Trecho de Programa Pascal

A Figura 2.1 mostra um trecho de programa PASCAL. Neste exemplo, a seqüência de comandos não apresenta nenhuma dependência de dados. O comando 6 depende somente da verificação da condição imposta pelo comando 5. Todas os outros comandos

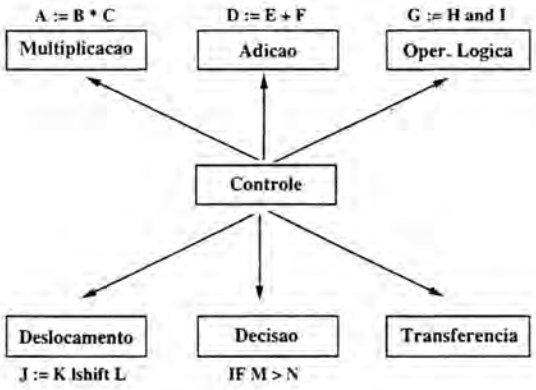

FIGURA 2.2 - Recursos de Máquina

poderiam ser executados em paralelo, desde que a máquina alvo dispusesse de unidades funcionais especializadas para executar a seqüência de comandos proposta.

Por exemplo, num primeiro instante, os comandos $1, 2, 3, 4$ e 5 poderiam ser executados pelas unidades funcionais apropriadas, como mostrado na Figura 2.2. Nesta arquitetura exemplo, as unidades funcionais (Multiplicação, Adição, Oper. Lógicas, Deslocamento e Decisão) poderiam ser ativadas pela unidade de controle tão logo esta decodificasse as instruções do trecho de programa proposto, Figura 2.1. Somente após a execução do comando 5, unidade de Decisão, a unidade de controle poderia despachar o comando 6 para a unidade funcional de Transferência. Neste caso supomos a verificação da condição $M > N$, isto é, se a condição do comando *IF* for verdadeira, o fluxo de controle será então desviado para a cláusula *THEN*.

A diferença no tempo de execução e a freqüência de ocorrência de certas instruções em programas típicos, estimularam o desenvolvimento de processadores com múltiplas unidades funcionais. Interessados em desenvolver processadores capazes de executar concorrentemente duas ou mais instruções, provenientes de um mesmo programa objeto, os fabricantes iniciaram alguns projetos nesta direção. Nestas arquiteturas, ao invés de esperar pela conclusão da instrução corrente, o algoritmo de despacho das instruções, antecipa a busca dos comandos subseqüentes, despachando-os para as unidades funcionais livres [FER 92]. A Figura 2.3 mostra como o trecho de programa da Figura 2.1 poderia ser executado numa máquina com múltiplas unidades funcionais independentes, Figura 2.2.

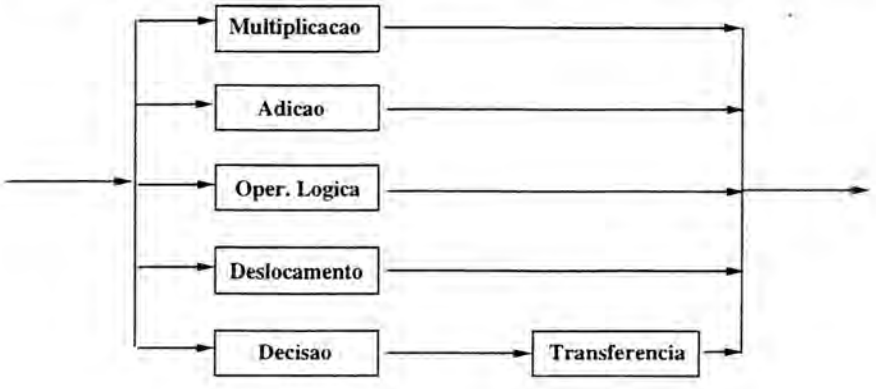

FIGURA 2.3 - Execução numa Arquitetura com Múltiplas Unidades Funcionais

Nas arquiteturas superescalares aos invés de comandos como mostrado no exemplo anterior são executadas na realidade as instruções derivadas da compilação do programa. Cada comando gera uma seqüência de instruções que são então processadas pelo processador.

2.2 Janela de Instruções

Somente a presença de múltiplas unidades funcionais autônomas não garante que o paralelismo de baixo nível possa ser explorado. Apesar de permitir a execução simultânea de múltiplas instruções, é necessário que exista um mecanismo que, antecipadamente, verifique se existem dependências entre as diferentes instruções do programa objeto. Por exemplo, após a transferência de uma instrução para uma unidade funcional, o processador não precisa aguardar sua conclusão para iniciar o tratamento da instrução subseqüente. E, se não houver dependência de dados entre a próxima instrução o algoritmo de despacho pode imediatamente despachá-la para uma outra unidade funcional, desde que esta esteja disponível.

É notório que o tempo de busca de instruções (acesso à memória) afeta substancialmente o tempo total de processamento de um programa. O tempo de atendimento poderá ser abreviado se a instrução sucessora já se encontrar no interior do processador. Para viabilizar o exame antecipado de instruções (*look-ahead*), algumas arquiteturas possuem um conjunto de registradores para armazenar as instruções. Este conjunto de registradores atua como um *buffer*, e na maioria dos casos, torna transparente o tempo de busca das instruções. Este *buffer* é denominado *Janela de Instruções*, pois contém um trecho do programa em execução.

O emprego de janelas de instruções em conjunto com sofisticadas técnicas de busca antecipada de instruções (*prefetching*) produz um sensível aumento no desempenho de processadores. Geralmente existe uma unidade funcional separada responsável por preencher a janela de instruções de acordo com a técnica utilizada. A Figura 2.4 mostra uma UI (Unidade de Instruções) e a janela de Instruções correspondente.

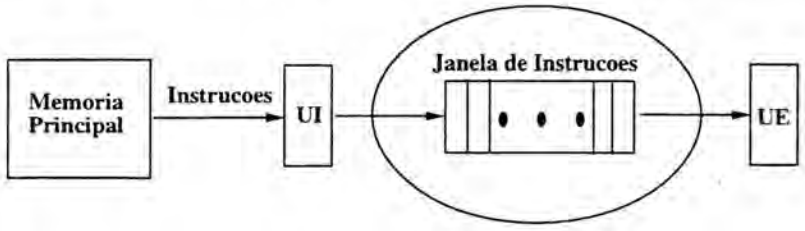

FIGURA 2.4 - Unidade de Instruções e Janela de Instruções

Para aumentar o desempenho do processador superescalar o algoritmo de despacho tenta minimizar o efeito das dependências de dados, retardando o início de instruções dependentes, e antecipando o início de outras. Para que isto seja viável é preciso que o processador possa despachar para as unidades funcionais, instruções fora da ordem especificada originalmente.

Para detectar as instruções que podem ser executadas em paralelo, os algoritmos de escalonamento dinâmico precisam ter acesso a um bloco de instruções de máquina. A janela de instruções é usada para armazenar instruções pré-buscadas (*prefetching*) da

memória. Na técnica de despacho seqüencial, as instruções contidas na janela são atendidas segundo a política *FIFO* (*First in First out*). Ou seja, em cada ciclo e não ocorrendo conflitos, a instrução mais antiga da janela é examinada, sendo despachada para uma das unidades funcionais. A medida em que as instruções da janela são despachadas, a unidade de instruções traz outros comandos para a janela. Quando não for possível despachar a instrução mais antiga da janela, em razão de dependências de dados ou por falta de uma unidade funcional, o despacho é interrompido e as instruções subseqüentes deixam de ser examinadas, passando a aguardar o despacho da instrução mais antiga da janela (i.e., da instrução que provocou a interrupção do mecanismo de despacho).

O desejo de aumentar a ocupação de unidades funcionais existentes na arquitetura, exige que cada instrução seja despachada e executada, tão logo esteja livre de dependências de dados. Portanto, para que este objetivo possa ser atingido, é necessário permitir não somente a execução fora de ordem, mas também é preciso dotar o processador da capacidade de realizar o despacho de todas as instruções que não apresentam conflito no uso de dados e recursos. Ou seja, é necessário realizar o despacho de instruções fora de ordem e de mais de uma instrução por ciclo de processador [SAN 94].

A inclusão de uma janela de instruções permite:

- despachar múltiplas instruções no mesmo ciclo de máquina, sendo possível ainda despachar instruções fora de ordem; e,
- alocar dinamicamente os recursos de maneira a suprir a demanda de instruções prontas para execução.

A cada ciclo de máquina, a unidade de instruções de um processador superescalar transfere um trecho de instruções para a janela. O trecho de instruções é transferido a partir da memória principal ou de uma memória *cache*. A unidade de instruções emprega o conteúdo do contador de programa (*Program counter*) para realizar esta busca.

Instruções contidas na janela são examinadas e despachadas para as unidades funcionais pelo algoritmo de despacho. Em seguida, um outro ciclo de processador é iniciado e o procedimento repete-se. Usualmente, as instruções são introduzidas a partir de um dos extremos da janela, sendo deslocadas em direção ao outro extremo à medida em que são despachadas para as unidades.

Além dos fatores mencionados anteriormente (dependências de dados e conflitos no uso dos recursos), o desempenho de processadores que fazem busca antecipada (*look-ahead*) de instruções é bastante afetado por um terceiro fator: os comandos de desvio condicional ou dependências de controle.

No caso de um comando de ramificação modificar o fluxo de controle seqüencial, as instruções já armazenadas na janela precisam ser descartadas (*flushed*) para darem lugar ao trecho de programa alvejado.

Quando um comando de desvio é encontrado, é necessário notificar a unidade de instruções (UI) de modo que ela possa redirecionar as buscas. Se a transferência de controle for do tipo incondicional (*unconditional branch*), a UI precisa apenas interromper a busca do trecho corrente, passando imediatamente para a busca do trecho de instrução referenciado pelo comando de desvio. Neste caso, o próprio mecanismo de despacho pode fazer a notificação (desde que o endereço da instrução alvo seja conhecido).

Todavia, se a instrução de desvio for do tipo condicional (*conditional branch*), o tratamento é mais sofisticado, pois não se sabe antecipadamente se a transferência de controle ocorrerá. Uma solução simples é interromper a UI até que a unidade responsável

pela avaliação da condição determine o novo conteúdo do contador de programa, i.e., se o PC será incrementado ou se receberá o endereço da instrução alvejada pelo comando de ramificação.

Interromper a operação da UI durante a avaliação de uma condição é inaceitável em virtude da perda que seria observada no desempenho da arquitetura superescalar, já que a freqüência de execução de instruções de desvios atinge faixas elevadas.

Devido a este terceiro fator degradante de desempenho, estudos vêm sendo feitos na área de previsão de desvios e execução especulativa. A seção 4.6 refere-se a este tema de estudo e descreve algumas das técnicas exploradas pelos projetistas.

2.3 Estrutura Genérica dos Pipelines Superescalares

Nesta seção apresenta-se a descrição do funcionamento de alguns estágios do pipeline tipicamente encontrado em arquiteturas superescalares convencionais. A figura 2.5 mostra esta estrutura.

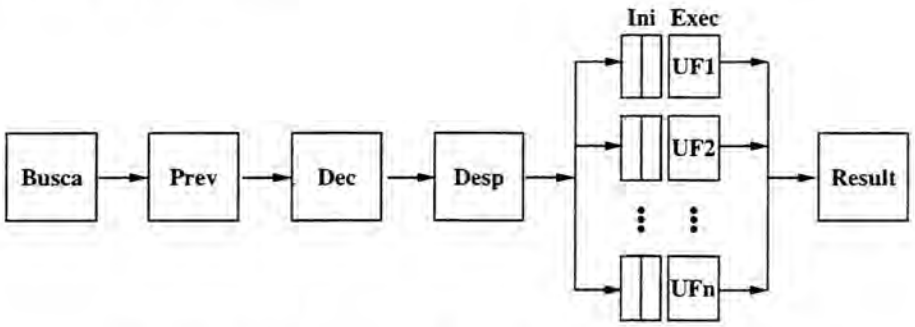

FIGURA 2.5 - Estágios de um Pipeline Superescalar

No ciclo n, o estágio de *Busca* acessa *largura de busca* instruções (*Fetchwidth*) na cache de instruções colocando-as num buffer denominado buffer de busca (*Fetch Buffer*).

No ciclo $n + 1$, o estágio de *Previsão* transfere estas instruções, do buffer de busca para a fila de instruções (*Instruction Queue*), examinando o tipo de cada uma delas. Quando uma instrução de desvio é encontrada, uma previsão do destino alvejado por esta instrução é feita. A previsão feita define o caminho que será seguido pelo estágio de *Busca* no próximo ciclo estabelecendo uma dependência de controle.

Uma dependência de controle existe de um comando Ci para Cj se o comando Cj deve ser executado somente se o comando Ci produzir certo valor [LIL 94]. Este tipo de dependência ocorre, por exemplo, quando o comando Ci é um comando condicional e Cj é executado somente se a condição avaliada por Ci for verdadeira.

Um dos maiores problemas no projeto de *pipelines* é garantir o fluxo contínuo de instruções através do *pipeline* permitindo desta forma aproximação ao desempenho máximo teórico. O fluxo de instruções pode ser interrompido por dois motivos. Primeiro, o tempo de acesso à memória para buscar uma instrução é tão longo que uma requisição, feita pelo estágio de busca, por outra instrução, não é satisfeita no tempo de estágio do *pipeline*. Segundo, a troca no fluxo esperado de instruções, devido a um desvio por exemplo, faz com que parte do conteúdo do *pipeline* seja descartado, e o *pipeline* seja recarregado.

Um desvio é tomado (*taken*) sempre que o fluxo de controle é desviado para o destino especificado como endereço alvo do desvio (*target address*). Desvios não-tomados (*not-taken*) são desvios que não transferem o fluxo de controle para o endereço alvo e portanto prosseguem a execução através da instrução adjacente à instrução de desvio. Tipicamente, desvios condicionais ou de controle de *loops* podem ser tomados ou não. Desvios incondicionais são sempre tomados.

Quando um desvio é previsto como tomado, todas as instruções acessadas antecipadamente pelo estágio de busca, que se encontram após a instrução de desvio, são descartadas. O endereço alvo do desvio é informado ao estágio de *Busca* que redireciona o acesso à cache de instruções recomeçando a busca através do caminho previsto. Em caso contrário, ou seja, quando o desvio é previsto como não-tomado, o estágio de previsão simplesmente continua a transferir as instruções para a fila de instruções até que o buffer de busca esteja vazio ou a fila de instruções esteja cheia, por exemplo.

O terceiro estágio faz a decodificação das instruções presentes na fila de instruções no ciclo $n+2$. Estas instruções são despachadas pelo estágio de *Despacho* no ciclo $n+3$, alocando entradas nas estações de reserva das unidades funcionais existentes na arquitetura e no buffer de reordenação. No ciclo seguinte, $n+4$, instruções que estão prontas são enviadas às unidades funcionais livres que as executarão no ciclo $n+5$, conforme as dependências de dados forem resolvidas. Após serem executadas, as instruções são reordenadas no ciclo $n+6$ e os registradores da máquina são atualizados com os valores produzidos por instruções executadas corretamente.

Descreveu-se de maneira sucinta o funcionamento de sete estágios normalmente encontrados em arquiteturas superescalares. Exemplos de processadores com características semelhantes às descritas neste capítulo são: PowerPC [CHA 94a, BEC 93, SON94], Pentium [ALP 93, BEK 91], Pentium Pro [INT 96a, INT 96b], IBM Risc System/6000 [GRO 90, OEH 90] e Alpha 21164 [EDM95].

Todas as simulações que serão apresentadas neste trabalho utilizaram um modelo de arquitetura superescalar baseado no modelo aqui apresentado. Variações alternativas foram empregadas em alguns dos experimentos realizados e serão descritas oportunamente ao longo do texto.

3 Limitações Fundamentais no Paralelismo de Instruções

Processadores superescalares são conceitualmente simples, mas o fato de acrescentar unidades funcionais ao *pipeline* assim como a capacidade de acessar mais instruções num mesmo ciclo por si só não garantem um aumento de desempenho. O aumento do paralelismo necessário, isto é, o aumento do número de instruções que devem estar prontas para serem executadas deve aumentar à medida em que aumenta o número de recursos para a sua execução.

As instruções de um programa na maior parte das vezes não são independentes uma das outras, logo estão interrelacionadas. Estas inter-relações impossibilitam algumas instruções de ocuparem os mesmos estágios do *pipeline*.

São basicamente três os fatores que podem limitar o grau de paralelismo explorado em um *pipeline* superescalar:

- Conflito de Recursos;
- Dependências de Dados, e;
- Dependências de Controle ou Procedurais.

Neste capítulo são descritas cada uma das limitações e como afetam o desempenho de uma arquitetura superescalares.

3.1 Conflito de Recursos

Uma instrução usa recursos do processador a cada estágio do *pipeline*. E estes recursos incluem memória, registradores, unidades funcionais, barramento, etc. Um conflito de recursos acontece quando duas instruções tentam utilizar o mesmo recurso ao mesmo tempo.

Conflito de recursos também acontecem em processadores escalares. Por exemplo, a busca de uma instrução pode coincidir com uma operação de carga de dados no barramento de memória. Entretanto, em um processador superescalar esta probabilidade é muito maior porque duas instruções podem estar tentando ocupar o mesmo estágio do *pipeline* para utilizar os recursos dedicados àquele estágio.

Remover um conflito é uma tarefa relativamente simples, mas isto não tem necessariamente um custo efetivo. Um conflito é eliminado pela replicação do recurso que está causando conflito. A solução pode ser facilmente encontrada através da replicação ou aumento do número de recursos que constituem o gargalo. No entanto um balanceamento preciso da arquitetura deve ser realizado para permitir uma análise de relação custo benefício. Nem sempre aumentar o número de recursos, mesmo que estes constituam um gargalo, resulta em ganho. Outros gargalos podem surgir ou até mesmo o custo de implementação pode ser tão alto que inviabilize a implementação do produto.

O balanceamento da arquitetura não é uma tarefa trivial. O número e tipo de recursos disponibilizados na arquitetura depende da funcionalidade que se deseja dar ao processador. Uma das vantagens na utilização de técnicas de *pipeline* e processadores superescalares é a exploração do paralelismo natural existente nos programas de forma imperceptível ao usuário, adicionalmente, sua operação independe do aplicativo.

O conflito de recursos pode ser um indicador de benefício [JOH 91]. Processadores escalares possuem várias unidades funcionais independentes que são utilizadas uma por vez. Já processadores superescalares podem fazer uso destas unidades funcionais ao mesmo tempo, portanto usando-as de maneira mais eficiente.

3.2 Dependências de Dados

Dependência de dados existe entre duas instruções quando estes fazem referência à uma mesma posição de memória ou a um mesmo registrador físico. O fragmento de código da Figura 3.1 mostra uma dependência direta (*read-after-write hazard*) do comando $S1$ para o $S2$, uma vez que o comando $S2$ precisa do valor A produzido pelo comando $S1$.

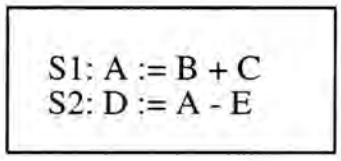

FIGURA 3.1 - Fragmento de Código com Dependência Direta de Dados

Dois comandos que escrevem em uma mesma posição de memória causam uma dependência de saída (*write-after-write hazard*). No fragmento de programa da Figura 3.2 o comando $I1$ deve ser executado antes do comando $I3$, uma vez que o comando $I2$ usa o resultado produzido por $I1$ (isto é, existe uma dependência direta de $I1$ para $I2$). Neste exemplo, uma anti-dependência (*write-after-read*) ocorre entre $I2$ e $I3$, já que $I2$ usa o valor X como operando, que é sobrescrito por $I3$, logo $I2$ deve ser executado antes de $I3$.

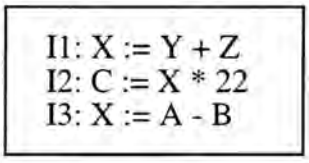

FIGURA 3.2 - Fragmento de Código com Dependências Direta, de Saída e Anti-dependência

Em resumo, existem três tipos de dependências de dados: verdadeira, de saída e anti-dependência. As dependências de dados podem ser resolvidas através do escalonamento de instruções, que pode ser dinâmico ou estático.

O escalonamento dinâmico é feito em tempo de execução e exige hardware complexo para execução desta tarefa. Escalonamento estático é feito em tempo de compilação e exige compiladores capazes de detectar as dependências entre dados de diferentes instruções e reordenar o código de forma a reduzir o efeito das dependências. Ambas as formas de escalonamento são correntemente empregadas nos processadores superescalares da atualidade. Na literatura da área podemos encontrar diversas técnicas de escalonamento de instruções que atenuam o problema das dependências de dados. Entre elas podemos citar:

- Algoritmo de Tomasulo [TOM 67]
  Escalonamento dinâmico de instruções empregado originalmente no IBM 360/91.
- Algoritmo de Thornton ou *Scoreboard* [THO 64]
  Escalonamento dinâmico de instruções empregado originalmente no CDC-6600.
- *List Scheduling* [LAN 80]
  Escalonamento estático de instruções através de compactação local.
- *Trace Scheduling* [FIS 81]
  Escalonamento estático de instruções através de compactação global.

3.3 Dependências de Controle

Uma dependência de controle existe de um comando Ci para Cj se o comando Cj deve ser executado somente se o comando Ci produzir certo valor [LIL 94]. Este tipo de dependência ocorre, por exemplo, quando o comando Ci é um comando condicional e Cj é executado somente se a condição avaliada por Ci for verdadeira.

Um dos maiores problemas no projeto de *pipelines* é garantir o fluxo contínuo de instruções através do *pipeline* permitindo desta forma aproximação ao desempenho máximo teórico. O fluxo de instruções pode ser interrompido por dois motivos. Primeiro, o tempo de acesso à memória para buscar uma instrução é tão longo que uma requisição, feita pelo estágio de busca, por outra instrução, não será satisfeita no tempo de estágio do *pipeline*. Segundo, a troca no fluxo esperado de instruções, devido à um desvio por exemplo, faz com que parte do conteúdo do *pipeline* seja descartado, e o *pipeline* seja recarregado.

Nas seções anteriores apresentamos características de dois fatores que limitam a performance de um *pipeline*. Conflitos de recurso são limitações físicas da arquitetura e podem ser solucionados através do re-balanceamento da arquitetura. Dependências de dados ocorrem quando há conflito entre operandos ou resultados de dois ou mais comandos em um programa e, podem ser resolvidos através do escalonamento de instruções.

Nesta seção apresentamos as características de desvios que causam dependências de controle em programas de aplicação e salientamos os problemas envolvidos que causam penalidades no desempenho de processadores *pipeline*.

3.3.1 Tipos de Desvios

Segundo [LIL 88] os três tipos básicos de desvios em processadores tradicionais são: incondicional, condicional e controle de *loops* (controle de *loops* são talvez um caso especial de desvios condicionais otimizados pela regularidade das estruturas de *loop*).

Um desvio incondicional sempre altera o fluxo de controle do programa. O endereço alvo faz parte da própria instrução e é conhecido durante a compilação ou durante a carga do programa. Conseqüentemente, o processador pode tratar um desvio incondicional como um fluxo seqüencial de programa, exceto que o contador de programa (PC) é carregado com um novo endereço ao invés de ser simplesmente incrementado. Desvios incondicionais "saltam"através de blocos de dados, para o início de programas, para subrotinas, etc.

Em um desvio condicional o processador deve fazer algumas avaliações antes de poder determinar o caminho de execução correto. A decisão normalmente deriva de um

código de condição (como um teste binário sobre o resultado da última operação) e resulta na seleção ou da instrução que está no endereço destino ou na próxima instrução seqüencial. O endereço alvo tipicamente é conhecido durante o tempo de compilação. Um exemplo de desvio condicional é o construtor *if-then*.

A Figura 3.3, mostra um exemplo típico de desvio condicional, onde:

- *i -1* é o comando de desvio condicional;
- *i* é a instrução adjacente;
- *j* é a instrução alvejada;
- *not-Taken* é o caminho seguido caso a condição seja falsa; e,
- *Taken* é o caminho seguido caso a condição seja verdadeira.

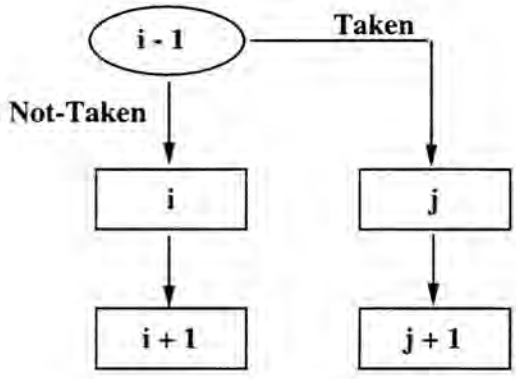

FIGURA 3.3 - Exemplo de Fluxo de Instruções com Desvio Condicional

O desvio em um comando de controle de *loops* é usualmente tomado e o endereço alvo é conhecido quando o programa é compilado ou carregado. O desvio na maioria das vezes transfere o controle de volta ao início do *loop* e é executado um número fixo de vezes ou depende de alguma condição variável.

3.3.2 Comportamento de Desvios

Desvios podem afetar dramaticamente o desempenho de um *pipeline*. Para conseguir algum resultado na atenuação do efeito causado pelos desvios existentes em programas devemos observar como os desvios se comportam.

Um desvio é tomado (*taken*) sempre que o fluxo de controle é desviado para o destino especificado como endereço alvo do desvio (*target address*). Desvios não-tomados (*not-taken* ou *untake*) são desvios que não transferem o fluxo de controle para o endereço alvo e portanto prosseguem a execução através da instrução adjacente à instrução de desvio. Tipicamente, desvios condicionais ou de controle de *loops* podem ser tomados ou não. Desvios incondicionais são sempre tomados.

[HEN 96] apresenta um estudo do comportamento de desvios e classifica as trocas no fluxo de controle em quatro tipos:

- Conditional Branches (desvios condicionais);

- Jumps (desvios incondicionais);
- Procedure calls (chamada a procedimentos), e;
- Procedure returns (retorno de procedimentos).

É conveniente conhecer a freqüência relativa destes eventos que podem usar diferentes instruções e podem ter diferentes comportamentos. Para três *benchmarks* (*TeX, Spice, GCC*), [HEN 96] obteve as seguintes freqüências para diferentes tipos de instruções de fluxo de controle.

A Figura 3.4 mostra a freqüência para os três tipos de desvios. Cada coluna mostra um dos tipos de desvios para cada um dos programas *benchmark*.

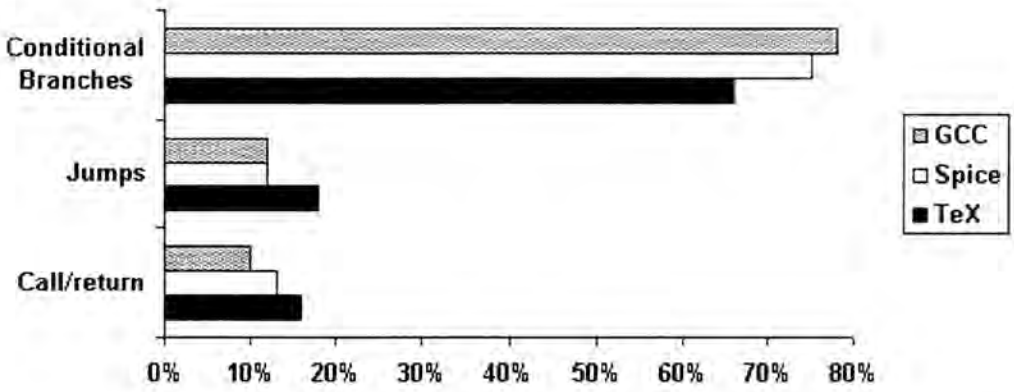

FIGURA 3.4 - Freqüência de Ocorrência de Desvios

Através dos dados mostrados no gráfico da Figura 3.4, considerando que as dependências de controle afetam drasticamente o desempenho de processadores *pipeline*, e que os desvios condicionais são os que melhor representam este efeito negativo, este capítulo concentrou-se no estudo de dependências de controle.

3.3.3 Penalidade de Desvios

Para analisar o efeito das instruções de desvio na performance de processadores *pipeline* considere o *pipeline* da Figura 3.5.

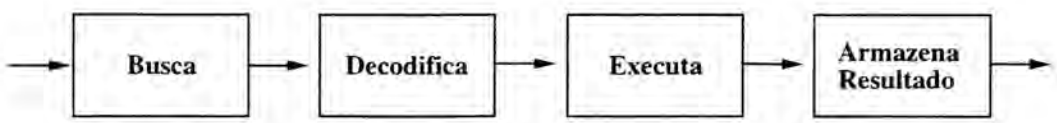

FIGURA 3.5 - Exemplo de Processador *Pipeline*

O primeiro estágio do *pipeline* busca a próxima instrução da memória. O próximo estágio decodifica a instrução e o terceiro estágio a executa. O quarto estágio armazena o resultado da operação executada. Se a instrução é um desvio condicional, somente no estágio de execução o processador saberá se o desvio será tomado.

A Figura 3.6 mostra a penalidade causada por uma instrução de desvio condicional *B* em um *pipeline* escalar. Somente após saber o resultado do desvio *B* o primeiro estágio pode buscar a próxima instrução no caminho correto.

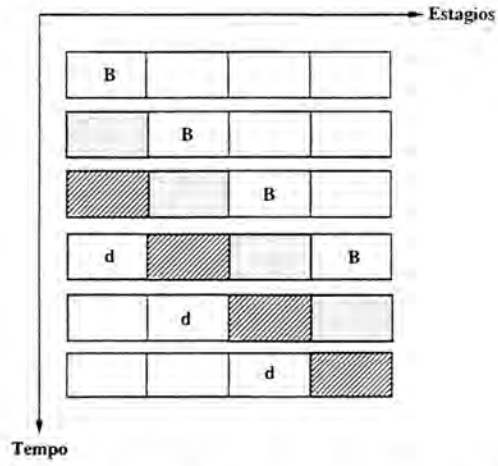

FIGURA 3.6 - Efeito das Dependências de Controle em *Pipelines* Escalares

A Figura 3.7 corresponde a um *pipeline* superescalar onde duas instruções podem ser processadas em cada estágio. Em ambas as Figuras (3.6 e 3.7) *B* é uma instrução de desvio que transfere o controle para a instrução *d*. As partes hachuradas representam estágios vazios.

No ciclo em que a instrução de desvio é decodificada, o endereço da instrução destino ainda não é conhecido. Se o *pipeline* continuar operando normalmente, as instruções seqüenciais serão acessadas e processadas. No entanto, se o desvio foi tomado (*Taken*), estas instruções não deveriam ter sido executadas. Para eliminar este risco, a solução mais simples consiste em suspender a busca de novas instruções até que a instrução de desvio tenha sido executada e o endereço alvo seja conhecido, esta técnica é conhecida como *Pipeline Stall After Branch* [TAL 95]. Somente após executar a instrução de desvio o processador pode definir em que endereço está a próxima instrução a ser buscada.

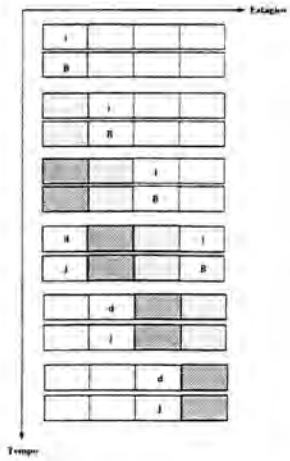

FIGURA 3.7 - Efeito das Depencências de Controle em *Pipelines* Superescalares

O latência de desvio é definida como sendo o intervalo de tempo entre a

decodificação da instrução de desvio e a decodificação da instrução destino, enquanto a penalidade de desvio é o número de instruções que deixam de ser executadas durante a latência do desvio [LIL 88]. No exemplo da Figura 3.6 o retardo de desvio é igual a dois ciclos tanto no *pipeline* escalar quanto no superescalar. No entanto, no *pipeline* escalar a penalidade de desvio é duas instruções, enquanto no *pipeline* superescalar é de quatro. Este exemplo mostra que a capacidade da arquitetura superescalar de processar várias instruções simultaneamente acaba por multiplicar os efeitos negativos das dependências de controle. Isto acontece porque uma eventual paralisação do *pipeline* impede que um maior número de instruções sejam executadas. Este efeito é tanto mais severo quanto maior for o grau de paralelismo da arquitetura.

Uma instrução de desvio exige que o processador determine dois tipos de informação antes de executar a instrução:

1. o resultado do desvio, e;
2. o endereço alvo do desvio.

3.3.4 Computando o Resultado do Desvio

O resultado de um desvio é baseado na comparação de dois valores distintos ou no resultado de uma operação aritmética. Esta computação pode ser uma computação explícita através de uma instrução de comparação, ou pode estar combinada com a própria instrução de desvio [TAL 95]. Se o resultado do desvio é computado por uma instrução separada, é necessário destinar recursos do sistema para armazenar o resultado intermediário até que a instrução de desvio seja executada. Neste caso, o resultado pode ser armazenado em registradores de propósito geral, ocupando recursos do sistema que poderiam ser utilizados por outras instruções. Alguns processadores incorporam registradores de condição, *flags*, para registrar o resultado destas comparações. Outros processadores também incluem conjuntos de códigos de condição, permitindo que o resultado de mais de um desvio pendente seja armazenado simultaneamente.

A computação separada do resultado de um desvio permite que desvios, cujos resultados são computados suficientemente antes do desvio, sejam resolvidos antes que o desvio seja encontrado.

Mesmo existindo alguns benefícios no emprego de instruções separadas de comparação e desvio, elas podem trazer algum custo. O fato da separação da computação, do resultado de um desvio e da instrução de desvio propriamente dita, requerer um registrador separado para armazenar o resultado da comparação, resulta em um aumento na lógica de controle de dependências do processador, uma vez que o processador deve verificar a dependência de dados com relação à este registrador também. O tamanho do código é aumentado, já que cada desvio requer duas instruções. Devido à estes fatores, muitos processadores utilizam instruções combinadas de comparação e desvio. Esta técnica reduz a quantidade de hardware em relação ao requerido para a computação separada do resultado de um desvio mas, faz com que todos os desvios causem penalidades no desempenho do processador, a menos que sejam manipulados por alguma das técnicas discutidas no Capítulo 4.

3.3.5 Computando o Endereço Alvo do Desvio

Se um desvio condicional é tomado, ou seja, sua condição é verdadeira, o processador deve conhecer o endereço alvo do desvio para poder fazer a transferência de

controle. O método mais simples é codificar o deslocamento (*offset*) entre o desvio condicional e o endereço alvejado na própria instrução de desvio. Estas instruções de desvio são chamadas de Desvio Relativo ao PC. Esta técnica não exige nenhum recurso adicional para armazenar o endereço de desvio, no entanto, arquiteturas que empregam instruções de tamanho fixo, limitam a distância entre o desvio e o endereço alvo ao tamanho do campo de *offset*.

Outra alternativa é utilizar instruções separadas, para armazenar o endereço alvo em um registrador específico para que posteriormente seja utilizado pela instrução de desvio. Neste caso, não há limitação de distância entre o desvio e o endereço destino a não ser pelo tamanho do registrador. Além disto, permite que o endereço alvo seja calculado antes mesmo de o desvio ser encontrado, o que diminui a probabilidade de causar penalidade na execução do desvio.

Quando o endereço alvo do desvio é computado por uma instrução separada, ele deve ser armazenado em um registrador até que a instrução de desvio esteja pronta para utilizá-lo.

3.4 Análise do Problema de Desvios em Pipelines Superescalares

Neste capítulo são comparados o desempenho (*speed-up*) de uma arquitetura que apresenta mecanismo perfeito de previsão de desvios com o desempenho de uma arquitetura real que possui mecanismo de previsão de desvios e execução especulativa existentes em microprocessadores atualmente comercializados.

Os resultados apresentados nesta seção foram obtidos com configurações idênticas para as arquiteturas ideal e real de acordo com os seguintes parâmetros:

- Largura de Busca de 8 instruções;
- Buffer e Fila de Instruções com 16 e 32 posições, respectivamente;
- *Branch History Table*: 1024 entradas/*4-way set associative*;
- *Pattern History Table*: 4096 entradas;
- Largura de Despacho de 8 instruções;
- Unidades Funcionais (homogêneas) variando de 2 a 8, cada uma com 8 estações de reserva;
- 1 Unidade de Desvios com 8 estações de reserva;
- 8 Barramentos de Resultados, e;
- Buffer de Reordenação com 64 posições.

Os gráficos das figuras 3.8 e 3.9 mostram que o desempenho da arquitetura superescalar real é bastante inferior ao desempenho da arquitetura ideal. Segundo [CHA 96], os fatores que limitam o desempenho da arquitetura ideal são as dependências de dados e a disponibilidade de recursos. Já na arquitetura real existe um outro fator predominante que causa o esvaziamento da fila de instruções e a redução do número médio de instruções despachadas para execução. O número de ciclos com despacho nulo

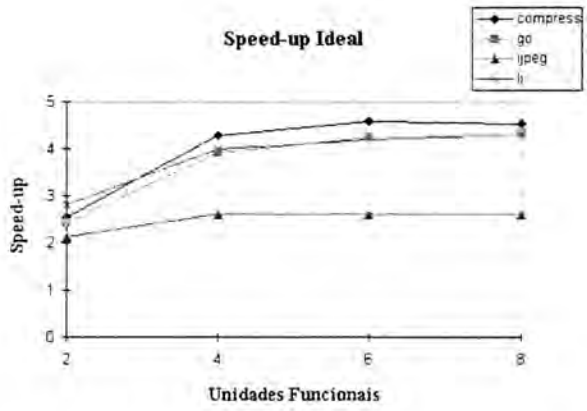

FIGURA 3.8 - Speed-up da Arquitetura Ideal

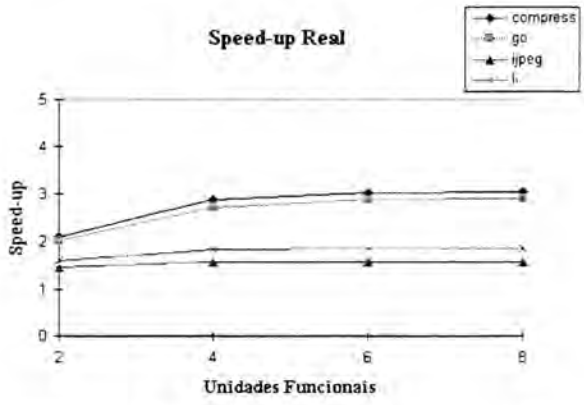

FIGURA 3.9 - Speed-up da Arquitetura Real

é bastante superior na arquitetura real em função da descontinuidade na transferência de instruções para a fila de instruções decorrente da previsão de desvios tomados.

O gráfico da figura 3.10 mostra que a existência de ciclos com despacho nulo na arquitetura real deve-se a três fatores: disponibilidade de recursos, fila de instruções vazia e despacho bloqueado devido a previsões incorretas.

Como se pode observar, a existência de ciclos com despacho nulo deve-se principalmente a ocorrência de fila de instruções vazia. É importante salientar que o aumento da taxa de indisponibilidade de recursos, mesmo com o aumento do número de unidades funcionais, ocorre devido a profundidade de especulação mantida em todas as configurações.

Analisando-se os fatores que causam o esvaziamento da fila de instruções pode-se notar que as constantes quebras de fluxo são o principal fator. No gráfico 3.11 verifica-se que a fila de instruções é esvaziada devido as previsões incorretas e as constantes quebras de fluxo. No entanto, as constantes quebras de fluxo são a principal causa do esvaziamento da fila de instruções. É importante salientar aqui que os mecanismos atualmente empregados não resolvem a quebras de fluxo.

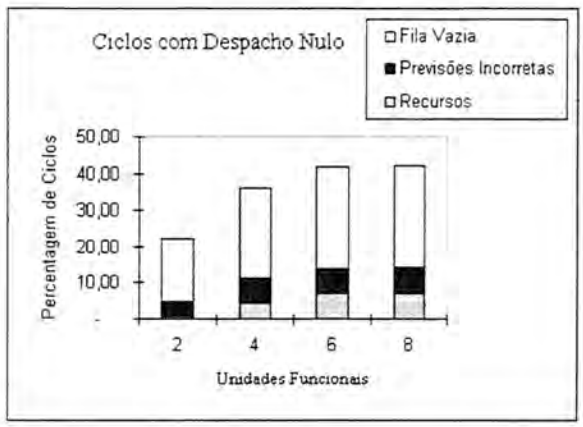

FIGURA 3.10 - Gráfico de Ciclos com Despacho Nulo

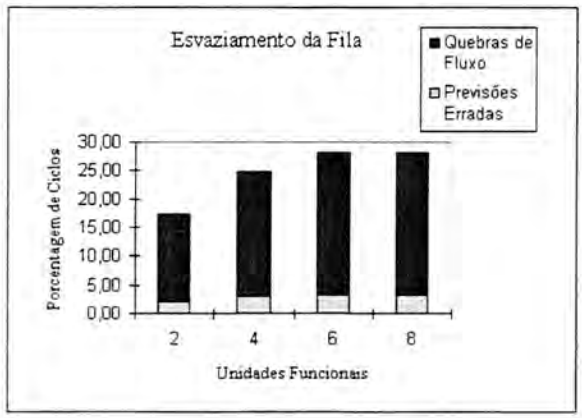

FIGURA 3.11 - Ocorrência de Fila Vazia

4 Técnicas para Redução do Custo de Desvios

Este capítulo discute algumas das técnicas utilizadas para reduzir a penalidade causada por instruções de desvio no desempenho de processadores *pipelined*. Primeiro, soluções mais básicas são apresentadas, embora fáceis de implementar, não apresentam tão boa eficiência. Estas, são seguidas por esquemas mais avançados, mais complexos, mas que permitem porém uma redução significativa na penalidade causada por este tipo de instruções.

4.1 *Pipeline Stall After Branch*

O esquema mais básico para manipular desvios condicionais é simplesmente bloquear a busca de instruções no *pipeline* quando um desvio é encontrado. Instruções anteriores ao desvio podem seguir através do *pipeline* até completarem a execução. Uma vez que o resultado e o destino do desvio tenham sido determinados, as instruções que logicamente seguem o desvio podem então entrar no *pipeline*.

Este esquema permite que um número inaceitável de ciclos sejam desperdiçados enquanto o processador está esperando para que o endereço destino do desvio seja resolvido. A utilização do *pipeline* é ainda menor se considerarmos processadores mais avançados como os superescalares ou *superpipelines*. Como resultado, este esquema é raramente utilizado.

4.2 *Delayed Branch*

A Figura 4.1 mostra como as instruções de um programa podem ser reorganizadas de modo a reduzir o custo do processamento de comandos de desvio num processador que emprega a técnica *Delayed Branch*.

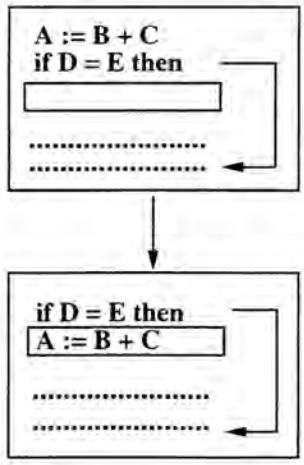

FIGURA 4.1 - Reorganização de Instruções Segundo a Técnica *Delayed Branch*

O compilador tenta movimentar as instruções do programa de aplicação, alocando-as após o comando de desvio condicional. Para fazer estas movimentações, o compilador

deve levar em consideração as relações de dependência entre as instruções, de maneira que a equivalência semântica do prograna seja preservada [FER 92]. Os retângulos da Figura 4.1 representam um trecho de programa antes e depois da reorganização. No retângulo superior da figura, uma instrução do tipo *NOP* é precedida pelo comando de desvio condicional. Após a reorganização, retângulo inferior da Figura 4.1, o comando de atribuição foi movimentado, passando a ocupar a posição alocada anteriormente ao comando *NOP*. Esta reorganização no código objeto reduz o custo de execução dos comandos de desvio, resultando num tempo de processamento menor.

Este esquema delega ao compilador a tarefa de preencher os *delay slots* (retângulos menores na Figura 4.1) depois de cada instrução de desvio. A tendência a medida em que o *pipeline* se torna mais profundo é que o número de *delay slots* cresça tornando mais difícil a tarefa do compilador em preencher àqueles *slots* com instruções válidas. Infelizmente, este mecanismo não apresenta uma boa escalabilidade, limitando sua eficácia em arquiteturas superescalares. Em uma arquitetura escalar, onde o *pipeline* acessa apenas uma instrução por ciclo, a instrução de desvio e as instruções nas posições de atraso (*delay slots*) são acessadas em ciclos diferentes. Ao contrário, em um *pipeline* superescalar, múltiplas instruções são acessadas em um mesmo ciclo. As instruções nas posições de atraso, que antes eram acessadas em ciclos distintos, são agora acessadas juntamente com a instrução de desvio, e assim deixam de mascarar o retardo do desvio. Torna-se necessário aumentar o número de posições de atraso, chegando até d x l posições adicionais, onde d é o retardo do desvio e l é o número de instruções acessadas. Este aumento no número de posições de atraso dificulta o preenchimento com instruções úteis, aumenta o uso de instruções *NOP* que consomem ciclos e não contribuem para a execução do programa [CHA 95].

4.3 Processadores *Multistreamed*

Processadores *Multistreamed*, ou Processadores Multifluxo, são capazes de despachar simultaneamente instruções de diferentes *threads* [TAL 95]. Enquanto instruções pertencentes ao mesmo fluxo de instruções preenchem as posições de atraso, na técnica *Delayed Branch*, os processadores *multistreamed* preenchem estas posições com instruções pertencentes à outros fluxos de instruções. Processadores *multistreamed* podem preencher as posições de atraso dinamicamente, enquanto que na técnica *Delayed Branch*, as posições de atraso são preenchidas estaticamente (isto é, em tempo de compilação).

Esta técnica mascara o custo de um desvio preenchendo as posições de atraso com instruções provenientes de diferentes fluxos de instruções. Logo, o processador esta constantemente executando instruções válidas de outros fluxos, o que aumenta a chance de manter as unidades funcionais ocupadas com instruções válidas.

4.4 *Fetch Taken and Not-Taken Paths*

Assumindo que o endereço destino de um desvio está disponível, o processador pode buscar instruções de ambos os possíveis caminhos de um desvio enquanto está esperando pela resolução da instrução de desvio propriamente dita. Desta forma, as instruções pertencentes ao fluxo correto estarão prontas no momento em que o desvio

for executado.

A desvantagem de buscar instruções de ambos os fluxos está no aumento do número de acessos à cache de instruções. O único benefício de ter buscado instruções de ambos os caminhos é simplesmente que o sucessor lógico do desvio já passou através do estágio de busca do *pipeline*, mas ainda deve passar pelos estágios restantes. Um possível benefício é que a instrução que será executada após o desvio já está na cache de instruções, mas muitos processadores que implementam este esquema não atendem uma falha (*miss*) causada por buscar um caminho alternativo do desvio. Isto porque o caminho que causou a falha pode ser o caminho incorreto e, neste caso, não é necessário preencher a cache com linhas possivelmente desnecessárias, que poderiam sobrepor instruções correntemente presentes na cache e que serão executadas num futuro próximo.

Uma variação mais agressiva desta técnica é buscar e executar ambos os caminhos de um desvio, alterando o estado do processador somente com os resultados do caminho correto no momento em que resultado do desvio for conhecido. Este esquema é raramente empregado uma vez que requer duplicação de maior parte do *pipeline* de instrução e por isto é completamente dispendioso. Entretanto, como o custo do hardware continua a decair, é possível que este esquema venha a ser considerado no futuro [TAL 95].

4.5 *Branch Folding*

Algumas arquiteturas, no seu conjunto de instruções, incluem instruções específicas para configurar as condições de desvio (*flags* de condição) e trocar o fluxo de controle. Condições de desvio são tipicamente realizadas com o resultado de uma comparação específica entre dois registradores, ou como o resultado de uma operação aritmética. Neste caso, é possível que a instrução que resolva a condição em que o desvio está baseado tenha completado sua execução e o resultado do desvio seja conhecido quando o desvio é encontrado. Este "desvio resolvido"pode ser removido do fluxo de instruções e substituído pelo seu sucessor lógico. Com a técnica *branch folding* é alcançado o efeito de um desvio de ciclo zero (*zero-cycle branch*). Esta técnica é usada pelo IBM RISC System 6000 e pelo PowerPC 601.

4.6 Previsão de desvios

Os comandos de desvio incondicional e condicional são as instruções de ramificação mais freqüentemente executadas. Instruções de desvio incondicional sempre alteram o fluxo de controle seqüencial do programa. Estas instruções, geralmente possuem um campo que armazena o endereço lógico alvo da ramificação. Neste caso, o endereço efetivo pode ser obtido durante a compilação ou no momento em que o programa é carregado na memória para execução. Tendo em vista que o endereço da instrução sucessora de uma instrução de desvio incondicional é único, então ela pode ser manipulada da mesma forma como se fosse uma instrução que altera o fluxo de controle.

O mesmo não ocorre com instruções de desvio condicional. Quando uma instrução deste tipo é executada, a unidade de controle precisa decidir qual das duas ramificações no fluxo de controle será usada. Por esta razão, o aproveitamento dos recursos pode ser afetado, pois a janela de instruções pode não conter o trecho de instruções que será executado após a avaliação de um comando de desvio condicional.

"Avaliar antecipadamente o endereço alvo é uma estratégia vantajosa [FER 92]". Tão logo o endereço da instrução alvo esteja disponível, a unidade de controle poderá buscá-la antecipadamente, armazenando-a num dos registradores do processador. A avaliação do endereço alvejado e a busca antecipada da instrução correspondente, são atividades que podem ser levadas a cabo em paralelo com a execução de outras instruções que precedem o comando de ramificação condicional.

Se a condição associada ao desvio for verdadeira, então a instrução sucessora (que já está no interior do processador, por exemplo na janela de instruções) pode ser iniciada imediatamente, o que reduz a penalidade usualmente imposta por comandos de desvio condicional. Se a condição for falsa, a instrução alvo poderá ser descartada.

Em uma situação extrema, as dependências de controle interrompem a busca de novas instruções até que a instrução de desvio condicional seja executada. Isto reduz a taxa de busca de instruções afetando o balanceamento da arquitetura. O potencial das arquiteturas superescalares é efetivamente explorado somente quando o fornecimento de instruções é suficiente para manter as unidades funcionais ocupadas. Se a taxa de busca for menor que a taxa de execução potencial, o desempenho fica limitado pelo acesso às instruções [CHA 94].

Para reduzir os efeitos das dependências de controle, é necessário permitir que a busca de instruções continue na presença de desvios. Para um desvio condicional, o fluxo de controle prossegue através da instrução seqüencial quando o desvio é não-tomado, ou a partir de alguma outra instrução quando o desvio é tomado. A interrupção da busca acontece porque não é possível saber, *a priori*, através de qual destes possíveis ramos o fluxo de controle prosseguirá. No entanto, ao invés de simplesmente suspender a busca, pode ser mais vantajoso prever o resultado do desvio e continuar acessando as instruções através do caminho previsto. Se eventualmente a previsão estiver errada, as instruções acessadas indevidamente devem ser descartadas e a busca deve ser redirecionada para o destino correto. Neste caso, o desvio continua a introduzir um custo. No entanto, se a previsão for correta, a busca pode prosseguir normalmente e o custo do desvio terá sido efetivamente anulado. A eficácia deste método depende, basicamente, da freqüência de acerto das previsões.

4.6.1 Previsão Estática

Na previsão estática o resultado previsto para um desvio é sempre o mesmo. Este tipo de previsão também é chamada de previsão fixada e pode ser executada a nível de software, durante a compilação, ou a nível de hardware, em tempo de execução.

As técnicas de previsão de desvio, em alguns processadores, assumem que o fluxo de controle sempre será transferido, ou não, para a instrução alvejada pelo comando de desvio condicional. As previsões estáticas podem ser implementadas segundo três variações:

1. O desvio sempre ocorrerá;
2. O desvio nunca ocorrerá; e,
3. Código de operação determina a previsão.

Tendo em vista que a maioria dos desvios condicionais provocam uma transferência no fluxo de controle, então a primeira técnica explora este fato, prevendo que a sucessora de um comando de desvio é a instrução contida no endereço alvo. Logo, a unidade de

instruções busca o trecho de código alvejado ao invés do trecho de código adjacente ao comando de desvio (p.e., o IBM 360/91).

A segunda técnica de previsão é implementada na maioria dos processadores que fazem busca antecipada de instruções (*look-ahead processors*): assumindo que a transferência de controle nunca ocorrerá, a unidade de instruções continua buscando as instruções adjacentes ao comando de desvio (p.e., i960CA, MC68020, VAX 11/780).

Ao invés de fixar uma única direção, a terceira técnica de previsão de desvios leva em consideração o código de operação do comando de ramificação para decidir se o fluxo de controle será transferido para o endereço alvejado ou não: para alguns tipos de desvio, a técnica prediz que a ramificação ocorrerá, para outros tipos, a previsão indica execução seqüencial, i.e., a transferência de controle não ocorrerá.

Esta técnica de previsão usa os resultados de estudos sobre o comportamento dos diversos tipos de comandos de desvio, levando em consideração a probabilidade do controle ser transferido. Por exemplo, comandos de desvio que controlam laços de programas, provocam uma transferência de controle na maioria das vezes. O mesmo ocorre com instruções de chamada à procedimentos (*call*). Por outro lado, o comportamento de desvios usados na implementação de estruturas do tipo " *if - then - else* ", não é previsível.

Os comandos de desvio são monitorados através de programas representativos determinando o número de vezes que cada tipo de comando de desvio provocou a transferência de controle.

Após esta etapa, usando a freqüência dinâmica de cada tipo de desvio, a previsão é fixada, sendo armazenada, por exemplo numa memória do tipo ROM no interior do processador. Em tempo de execução, a terceira técnica consulta esta memória para decidir se o desvio ocorrerá ou não.

4.6.2 Previsão Dinâmica

A previsão dinâmica de desvios ocorre em tempo de execução, sendo realizada apenas a nível de hardware. Ao contrário do caso estático, onde a previsão de um desvio é fixa em todas as suas execuções, agora a previsão é feita com base no histórico de execuções anteriores do desvio, podendo ser modificada nas execuções futuras.

As técnicas dinâmicas verificam o que ocorreu anteriormente, durante as últimas execuções do comando de desvio, e usando estas informações tentam reproduzir o comportamento dominante do comando.

As informações indicando o que ocorreu recentemente quando da execução de alguns comandos de desvio, ficam armazenadas numa pequena tabela localizada no interior do processador. Esta tabela é denominada Tabela de História (*Branch History Table*). Por exemplo, o processador pode incluir uma pequena tabela para armazenar informações relacionadas com as mais recentes execuções dos comandos de desvio. Os campos de cada entrada poderiam conter:

1. O endereço do desvio e o endereço da instrução sucessora; ou,
2. O endereço do desvio e a instrução sucessora;

No primeiro caso, a tabela armazena em um dos campos o endereço de alguns dos comandos de desvio recentemente executados. Em outro campo armazena o endereço da instrução alvejada por cada comando quando da última execução. Com isto, tão logo uma instrução tenha sido buscada, o mecanismo de previsão de desvios, pode iniciar a busca

da instrução sucessora. O endereço da instrução é usado como chave para acesso à tabela. Se a instrução estiver armazenada no campo de *endereço do desvio* isto significa que o endereço no campo *endereço da sucessora* será usado para buscar a próxima instrução. Observe que esta técnica prediz que o desvio ocorrerá desde que o endereço do desvio esteja contido na tabela. Se ao contrário, o endereço de um desvio não estiver contido na tabela, o mecanismo prediz que o fluxo seqüencial de busca não será interrompido.

Após a busca do comando de desvio, o campo *endereço da sucessora* da entrada correspondente é usado para acessar a memória principal ou a *cache* de instruções, e a busca da instrução sucessora pode ser iniciada imediatamente. Neste caso, torna-se desnecessário aguardar pela decodificação do comando de desvio; pela avaliação do endereço efetivo por ele alvejado; e pelo resultado do comando de desvio. Em outras palavras, as fases de execução de um comando de desvio podem ser realizadas em paralelo com a busca da instrução alvo.

No segundo caso, ao invés de armazenar, no segundo campo, o endereço da instrução sucessora ao comando de desvio, a tabela guarda a própria instrução sucessora. Nesta tabela, o segundo campo chama-se *instrução sucessora*. Com este artifício, além das vantagens do esquema anterior, elimina-se a busca antecipada, já que a instrução já se encontra na tabela.

Esta tabela é manipulada da mesma forma que no primeiro caso, dispensando porém a busca antecipada. Por esta razão, além de ser usada pelo mecanismo de previsão de desvios, a tabela também desempenha o papel de uma janela alternativa de instruções, i.e., uma segunda janela contendo comandos provenientes do fluxo de controle que será executado se o desvio ocorrer.

Num processador superescalar, as instruções contidas nesta janela alternativa poderiam ser despachadas e executadas imediatamente, mesmo antes de sabermos se a previsão foi correta. Nesta caso, para que a equivalência semântica do programa original seja mantida, o hardware deve prover facilidades de cancelar a instrução e recuperar as modificações feitas. Alternativamente, o compilador poderia gerar código para neutralizar o efeito provocado pela execução indevida da instrução sucessora.

Com o objetivo de reduzir o tempo de acesso às informações da tabela, pode-se empregar o uso de uma memória associativa na implementação física da tabela. Assim, o conteúdo do PC é comparado simultaneamente com o campo *endereço do desvio* de todas as entradas da tabela. Se um dos endereços na tabela for igual ao conteúdo do contador de programa, a correspondente instrução sucessora pode ser obtida imediatamente.

Devido ao custo, estas tabelas possuem um número limitado de entradas, o que faz com que nem todos os comandos de desvio de um programa qualquer possam ser armazenados ao mesmo tempo. Algoritmos de substituição, semelhantes aos usados pelos Sistemas Operacionais na implementação de memória virtual, podem ser empregados no gerenciamento das entradas da tabela.

Previsão Dinâmica Baseada na História

Nas técnicas de previsão dinâmica, o processador usa informações (história) sobre desvios que já executou anteriormente para prever o destino destes quando da próxima execução. Por exemplo, o processador pode manter uma pequena tabela para instruções de desvio recentemente executadas com vários bits em cada entrada. O processador usa os bits de história, armazenados na tabela, para fazer uma previsão de acordo com alguma heurística [CHA 94].

Um mecanismo bem simples de previsão dinâmica é aquele que usa uma tabela de história de desvios, ou BHT (*Branch History Table*). Esta tabela é implementada sob a

forma de uma pequena memória (normalmente integrada com o processador em um único dispositivo), onde cada entrada armazena um ou mais bits usados para registrar a história dos desvios.

Quando uma instrução é acessada, ela é pré-decodificada para que seja determinado se aquela é uma instrução de desvio. Se for o caso, os bits menos significativos do endereço da instrução são usados para indexar a BHT. O bit na entrada selecionada fornece a previsão do desvio: por exemplo, 0 (zero) indica que o desvio será previsto como tomado, enquanto um bit 1 indicaria previsão de desvio não-tomado. A instrução a ser acessada no próximo ciclo é determinada de acordo com esta indicação.

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

No caso mais simples, cada entrada na BHT possui um único bit, o que permite o registro de apenas dois estados: *N*, indica que o desvio foi não-tomado em sua última execução e que a previsão será não-tomado na execução corrente; *T*, indica que o desvio foi tomado na execução anterior e que será previsto como tomado na execução corrente.

A Figura 4.2 mostra o autômato utilizado para previsão com 1 bit de história. O nome de cada estado indica a previsão (*T* para tomado e *N* para não-tomado). O rótulo em cada arco indica o resultado de um desvio. Logo, a partir de um estado e um resultado o autômato fornece um novo estado, ou seja, uma previsão baseada na última execução do desvio.

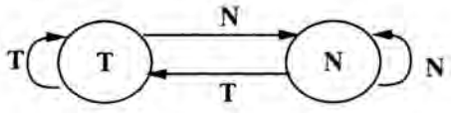

FIGURA 4.2 - Autômato para Previão com 1 bit de História

O principal problema na previsão dinâmica com BHT é a ocorrência de colisões [CHA 94]. Considere duas instruções de desvio em endereços diferentes, mas cujos bits menos significativos coincidem. Estes dois desvios selecionam a mesma entrada na BHT, e assim a informação de previsão de um desvio é usada na previsão do outro desvio, reduzindo a taxa de acerto na previsão de ambos os desvios.

Uma alternativa é fazer com que cada entrada na BHT possua o endereço de uma instrução de desvio, juntamente com os bits de previsão. Na busca da instrução, o endereço completo da instrução acessada é comparado associativamente com os endereços armazenados na BHT. Caso ocorra um *hit*, são usados os bits de previsão na entrada onde está o endereço coincidente. Se no acesso à BHT ocorre um *miss*, é feita uma previsão estática do desvio para determinar qual instrução será acessada no próximo ciclo. Se após a decodificação for verificado que a instrução que ocasionou o *miss* é um desvio, o seu endereço e o resultado da execução serão inseridos na BHT.

Um aperfeiçoamento deste mecanismo consiste em armazenar em cada entrada o endereço da instrução destino obtido na última execução de um desvio. Com isto, o hardware necessário para determinar o endereço destino de um desvio é incluído apenas no estágio de execução do *pipeline*. Este dispositivo modificado é chamado de tabela de destinos de desvios, ou BTB (*Branch Target Buffer*).

Branch Target Buffer

A *Branch Target Buffer*, Figura 4.3, é uma memória cache especial associada normalmente com o estágio de busca do *pipeline*. Cada entrada na *BTB* consiste de três campos: o endereço da instrução de desvio, os bits de história e o endereço destino mais recente para aquele desvio.

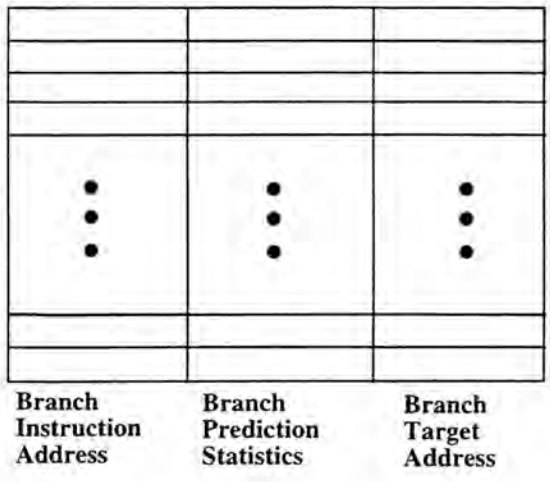

FIGURA 4.3 - Organização da *Branch Target Buffer*

A idéia básica de todos os esquemas de previsão de desvios é que, sob circunstâncias similares, existe uma grande probabilidade de que o desvio tenha o mesmo comportamento que teve quando fora executado pela última vez [TAL 95]. Portanto, mais importante do que a *hit ratio* da tabela, o algoritmo de previsão deve ser capaz de mapear as últimas ocorrências de um desvio e fornecer uma previsão condizente com o que, na maioria das vezes, aconteceu com aquele desvio.

O número de bits de história (previsão) é um fator de extrema relevância na escolha do algoritmo de previsão. Na figura 4.2 mostrou-se um autômato para previsão com 1 bit de história. O maior problema em se usar esta técnica é quando se faz necessário prever o destino de desvios de controle de laços, e o laço é executado mais de uma vez (*loops* aninhados). Para n iterações de um *loop*, as primeiras $n-1$ iterações são tomadas e o desvio ao final do *loop* é previsto corretamente. Entretanto, ao final da última iteração o desvio é não-tomado e a previsão é incorreta, uma vez que, durante a última execução o desvio foi tomado.

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

Em um mecanismo de previsão com 2 bits de história, é possível registrar o resultado das duas últimas execuções, e a próxima previsão é modificada apenas se as duas últimas previões foram incorretas.

A figura 4.4 mostra o autômato utilizado para a previsão com 2 bits. Nos estados onde os dois bits coincidem, a previsão segue o resultado indicado por ambos. Nos estados onde os dois bits diferem, a previsão segue a indicação do bit que registra o estado mais antigo. Estudos realizados por [LEE 84] mostram que, com dois bits de previsão, é possível alcançar uma taxa média de acerto individual de 90%. Com uma taxa nédia de *hit* de 95% na tabela de previsão, isto significa uma taxa média de acerto de 85%.

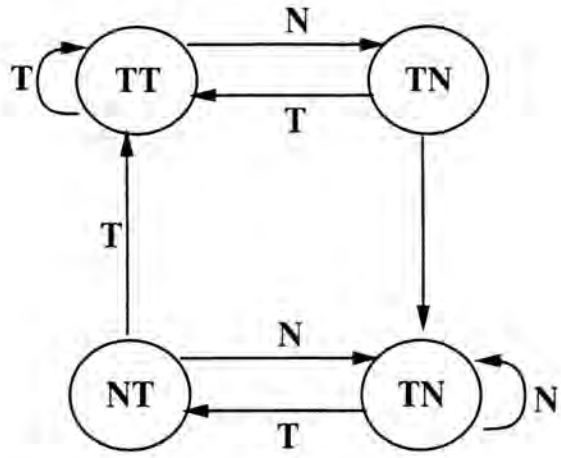

FIGURA 4.4 - Autômato de Previsão com 2 Bits de História

A BTB funciona da seguinte maneira: o estágio de busca compara o endereço da instrução que está buscando com os endereços que estão na BTB. Se o endereço está presente na BTB então uma previsão é feita em função dos bits de história correspondentes. Se a previsão diz que o desvio será tomado, então o endereço no campo de destino será usado para acessar a próxima instrução. Quando o desvio é resolvido, no estágio de execução, a BTB pode ser corrigida com a informação correta sobre o que aconteceu com o desvio, caso a previsão feita anteriormente tenha sido incorreta.

O funcionamento é semelhante ao do mecanismo com BHT associativa, com apenas algumas diferenças. No caso de *miss*, o endereço destino também é inserido na entrada juntamente com o endereço do desvio. Quando acontece uma previsão incorreta, o endereço destino é atualizado para refletir o destino correto do desvio. A Figura 4.5 mostra o diagrama para a previsão com BTB.

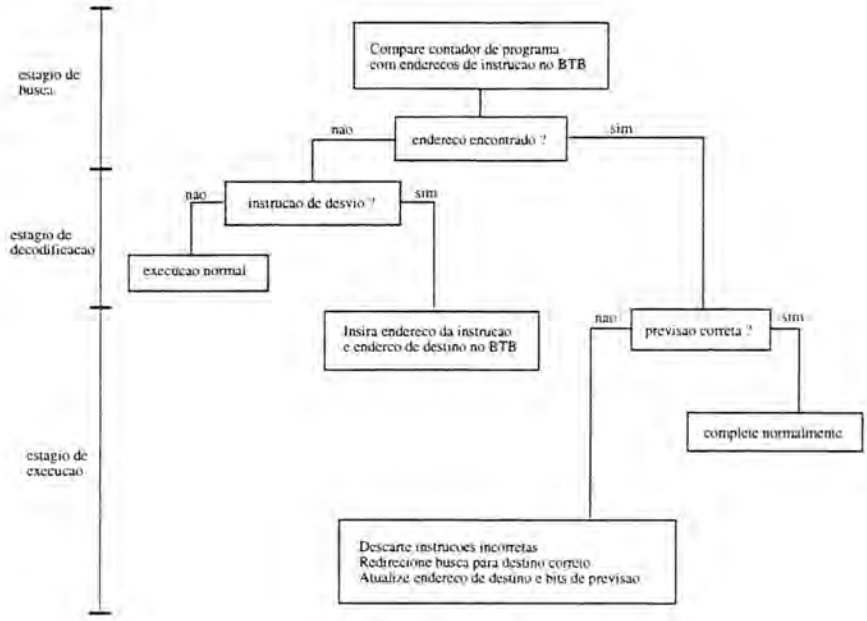

FIGURA 4.5 - Operação de uma *Branch Target Buffer*

Para que a técnica de previsão de desvios seja realmente eficaz, é necessário que a

maioria das previsões sejam corretas. Em uma BHT associativa ou em uma BTB, a taxa de acertos depende de dois fatores:

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

Logo, a taxa de acerto é dada por:

$$TA = HitRatio * FA \tag{4.1}$$

A fórmula 4.1 produz a taxa de acerto da previsão de desvios. Onde, a variável $HitRatio$ é freqüência com que as informações são encontradas na tabela de previsão e, a variável FA é a freqüência de acerto na previsão de desvios.

O primeiro fator depende da configuração da tabela de previsão, i.e., do número de entradas. A taxa de acerto individual depende em parte do número de bits de previsão. Com um número maior de bits é possível registrar a história dos desvios a partir de um passado mais distante.

Previsão Dinâmica em Dois Níveis

A execução superescalar é uma forma efetiva de explorar o paralelismo a nível de instrução. A eficiência entretanto, depende criticamente da precisão do mecanismo de previsão de desvios empregado [PAN 92].

O esquema de previsão em dois níveis, apresentou em média uma precisão de 97% de acerto nas previsões em nove benchmarks do conjunto SPEC. A precisão das previsões mostrou-se 4% melhor que outros mecanismos estáticos e/ou dinâmicos, segundo os estudos realizados por [YEH 91].

Uma vez que previsões incorretas causam o esvaziamamento do pipeline e consequentemente a invalidação da execução especulativa de instruções em andamento, o melhoramento no desempenho de processadores com o emprego desta técnica deve ser considerado.

Yeh e *Patt* [YEH 91] propuseram a idéia de coletar dinamicamente dois níveis de história de desvios. O primeiro nível, armazena a história dos últimos K desvios encontrados. O segundo nível, armazena o que aconteceu com as últimas j ocorrências de um padrão específico para os K desvios.

O primeiro nível é denominado *History Register Table* e o segundo nível *Pattern Table*. O endereço de um desvio é mapeado para acessar o primeiro nivel assim como em uma BTB convencional. Após mapear a entrada correta, o registrador de história (*Branch History Register*) fornece o padrão de bits que irá determinar qual entrada será acessada no segundo nível [YEH 93].

Ao acessar o segundo nível, o mecanismo dispõe então do bit de previsão que indicará o caminho a ser seguido pelo estágio de busca para acessar as instruções seguintes.

Após a instrução de desvio ser executada, o resultado (*Taken* ou *Not-Taken*) é deslocado para dentro do registrador de história, da entrada correspondente no primeiro nível, modificando o padrão para os desvios que mapearão aquela entrada futuramente. A lógica de transição de estado avalia o resultado do desvio juntamente com a previsão feita anteriormente para este, e fornece o novo bit de previsão que será usado posteriormente. A figura 4.6 ilustra as principais estruturas do mecanismo de previsão em dois níveis.

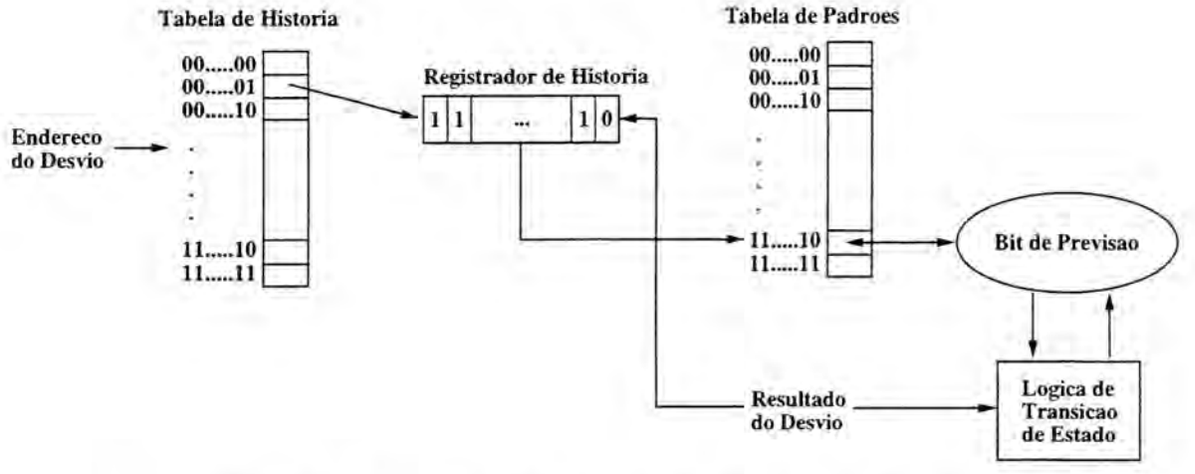

FIGURA 4.6 - Estrutura do Mecanismo de Previsão em Dois Níveis

Outros mecanismos têm sido propostos recentemente baseados na previsão em dois níveis. A literatura é vasta nessa área de pesquisa demonstrando que este é um ponto importante no projeto de arquiteturas superescalares.

4.7 Execução Especulativa

É essencial observar que a previsão de desvios possibilita apenas a continuidade da busca de instruções na presença de dependência de controle. Estes mecanismos não atacam a outra parte do problema, qual seja, o fato das dependências de controle tornarem a execução dependente do resultado de um desvio condicional. Uma instrução de desvio condicional deve ser antes executada para que seja determinado se as instruções acessadas antecipadamente pelo mecanismo de previsão de desvios podem, de fato, serem executadas [CHA 94].

Um processador superescalar alcança altos níveis de desempenho através da busca e execução de instruções presumindo que os desvios foram corretamente previstos e que não ocorreram exceções. Isto é chamado de execução especulativa [JOH 91]. O despacho e execução de instruções prosseguem, através do caminho previsto, enquanto o resultado do desvio não é conhecido.

É importante observar que na ocorrência de previsões incorretas, as instruções que vêm logo após uma instrução de desvio, e que podem ter sido executadas, não devem afetar o estado da arquitetura. A corretude da execução de um programa, em arquiteturas com execução especulativa, exige que mecanismos eficientes de recuperação garantam a equivalência semântica de tal programa. Os resultados produzidos pela execução de instruções que se encontram em caminhos previstos incorretamente, devem ser descartados.

Na abordagem mais simples, dependências de controle são satisfeitas bloqueando-se o despacho de novas instruções até que o desvio seja executado. Em uma solução menos restritiva, as instruções buscadas antecipadamente pelo esquema de previsão são decodificadas e executadas especulativamente, não alterando o estado da arquitetura. Se, após a execução do desvio for verificado que tais instruções pertencem ao ramo correto, é permitido que os resultados produzidos modifiquem efetivamente o estado interno

da arquitetura. Caso contrário, os resultados são descartados e é reiniciada a busca das instruções pertencentes ao ramo correto. Esta solução é denominada execução especulativa de instruções. A execução especulativa de instruções encontra-se associada a um mecanismo de previsão de desvios, pois é necessário que a busca de instruções continue durante a execução do desvio. Além disto, o mecanismo de previsão deve ser eficiente, de modo que na maioria dos casos a execução especulativa de instruções seja válida.

A arquitetura deve prover meios eficientes para garantir a equivalência semântica do programa caso as instruções acessadas pelo mecanismo de previsão forem executadas indevidamente. Em um caso seqüencial, as instruções dependentes de um comando de desvio condicional só seriam executadas após a verificação da condição imposta pelo comando, i.e., é necessário que o comando de desvio seja executado para depois buscar as instruções seguintes, decodificá-las e executá-las.

Os mecanismos de previsão de desvios garantem, com taxas de acertos elevadas em alguns casos, mas nem sempre com taxas de acerto de 100%, que a busca de instruções não seja interrompida quando da ocorrência de um comando de desvio. Na seção 4.6 foram descritas algumas técnicas de previsão, estáticas e dinâmicas, que abordam este aspecto. No entanto, somente a busca destas instruções não garante a utilização potencial da arquitetura superescalar. As instruções buscadas podem pertencer ao ramo correto ou não. A execução especulativa destas instruções deve ocorrer de forma dependente ao comando de desvio que as antecede. Embora, não interrompa a execução de instruções com dependência de controle, a execução especulativa, no pior caso, pode sofrer da ineficiência do mecanismo de previsão, i.e., se as instruções buscadas antecipadamente não pertencerem ao ramo correto, verificado somente após a execução do comando de desvio, todas as modificações causadas devem ser desfeitas e o novo trecho de código deve ser executado.

Para reparar possíveis erros nas previsões e permitir que seja mantido um estado consistente da arquitetura, diversos mecanismo foram propostos. *Smith* e *Pleskun* [SMI 85] propuseram uma estrutura para implementar recuperação: *Reorder Buffer*, atualmente utilizada em microprocessadores superescalares que fazem uso da técnica de execução especulativa.

O *Reorder Buffer* - *RB* é gerenciado como uma fila (*FIFO* - *First-in, First-out*). Quando uma instrução é decodificada, é alocada uma entrada no topo do *RB*. O resultado da instrução é escrito na entrada alocada depois que a instrução é completada. Quando o valor chega na saída, é escrito nos registradores arquiteturais, se nenhuma exceção está relacionada com a instrução em questão. Se a instrução ainda não completou, quando encontra a saída, o *RB* não avança até que esta tenha sido terminada. No entanto, o mecanismo de decodificação continua o processamento enquanto houverem entradas disponíveis no *RB*. Quando ocorre uma excecão, o conteúdo do *RB* é descartado, e o processador reverte para um estado consistente anterior à instrução que causou a exceção.

Algumas variações deste modelo foram também propostas e são amplamente utilizadas nos dias de hoje [SMI 85, JOH 91]. Existem várias formas de implementar recuperação de estado na ocorrência de falhas [JOH 91]. Mas, em processadores superescalares, a recuperação e reinício de operação acontecem freqüentemente, em função da ocorrência de previsões erradas, e por isso precisam ser executadas rapidamente. Alguns esquemas supõem que exceções não ocorrem com freqüência, e neste caso, não poderiam ser empregados em arquiteturas superescalares.

4.8 Execução Predicada

Todos os esquemas apresentados anteriormente esforçam-se para reduzir o impacto causado por instruções de desvio no desempenho através da diminuição do número de desvios que incorrem em penalidade. Outro esquema, chamado Execução Predicada, reduz o impacto de instruções de desvio através da substituição de certos tipos de desvios por instruções que modificam o estado do processador somente se certas condições forem verdadeiras [TYS 94, TAL 95].

Para os desvios condicionais cujos endereços alvo estão localizados a poucas instruções depois do desvio são candidatas à execução predicada. Desvios que se enquadram nesta categoria são denominados *short forward-going conditional branches*. Estes desvios compreendem a maior parte de todas as instruções de desvios, e muitos deles são não-tomados (*not-taken*) [TYS 94]. Na execução predicada, os desvios que possuem as características descritas acima são removidos, e as instruções que seriam saltadas são substituídas por instruções equivalentes que somente modificam o estado do processador se a condição em que o desvio estava baseado for verdadeira.

Execução predicada tem melhor performance em relação a técnicas como previsão de desvios somente quando a maior parte dos desvios removidos são não-tomados, se os desvios são tomados muitas das vezes, o código que estava seguindo-os raramente seriam executados [TAL 95]. Entretanto, com execução predicada, as instruções que normalmente seriam saltadas pelo desvio sempre seriam executadas, no entanto elas nunca modificariam o estado do processador. Portanto, execução predicada é usualmente aplicada para àqueles desvios que são na maior parte das vezes não-tomados.

Como resultado, nem todos os desvios condicionais são candidatos à execução predicada. Entretanto, previsão de desvios e execução predicada fazem uma poderosa combinação. Execução predicada reduz o número de desvios em um programa, e o esquema de previsão de desvios trata os desvios restantes que não foram substituídos por instruções predicadas. A execução predicada mostrou uma redução de 30% no número de desvios em um programa [TYS 94].

5 Trabalhos Correlatos

Arquiteturas superescalares estão sendo largamente empregadas em estações de trabalho e computadores pessoais de alta performance. Vários outros processadores com lançamento previsto a curto e médio prazo estarão baseadas nesta tecnologia.

Estas arquiteturas são capazes de despachar e executar multiplas instruções em um único ciclo. Embora, apresentem um paralelismo potencial maior que as máquinas com *pipeline* escalar, a limitação imposta pelas dependências entre instruções torna-se gradativamente maior a medida em que mais unidades funcionais são acrescidas à arquitetura.

Muitos dos estudos que estão sendo realizados são direcionados ao tratamento das dependências vizando atenuar o efeito causado por elas sobre a performance da máquina. Dependências de dados são tratadas por algoritmos de escalonamento que tratam da reorganização da seqüência original de instruções, objetivando executar àquelas que já se encontram prontas, ou que possuem todos os seus operandos prontos no momento da execução. Já as dependências de controle impõem uma ordem de precedência na execução de instruções impedindo que um grau elevado de paralelismo seja obtido. Conseqüentemente, as unidades funcionais tendem a permanecer ociosas até que estas dependências seja resolvidas.

Atualmente, a execução especulativa é a forma mais sofisticada de tratamento das dependências de controle, estando presentes nas arquiteturas superescalares que se situam numa faixa mais alta de desempenho.

O conceito de multifluxo foi proposto ainda na década de 70 por [FLY 72, KEL 75], com o objetivo de aumentar a utilização dos recursos em arquiteturas paralelas. No entanto, este conceito começou a ser de fato explorado apenas recentemente, pois os avanços na tecnologia de integração tornaram viável a implementação de tal arquitetura.

Muitas pesquisas vêm sendo desenvolvidas com o objetivo de encontrar novos paradigmas que sustentem a ordem de oito a dezesseis instruções executadas por ciclo. Embora as arquiteturas superescalares possam executar mais de uma instrução por ciclo, deparam-se com sérios problemas na busca de tais índices.

As atuais arquiteturas paralelas de processadores apresentam duas deficiências. Primeiro, o modelo de execução de instruções adotado limita a disponibilidade de instruções independentes que possam ser executadas em paralelo. Segundo, os mecanismos de tratamento de dependências entre instruções ainda não são capazes de anular totalmente o custo destas dependências. Isto é particularmente crítico nos mecanismos dedicados às dependências de controle.

Enquanto os processadores incluem mais unidades funcionais para fornecer um maior nível de paralelismo, um significante limite para o grau de paralelismo é encontrado devido ao tamanho do bloco básico. Uma alternativa para a previsão de desvios é executar especulativamente ambas as ramificações do desvio, ao invés de prever se o desvio é tomado ou não-tomado, e anular a ramificação incorreta tão logo o resultado do desvio seja conhecido. Desta forma, a penalidade do desvio é eliminada, embora sejam necessários mais recursos computacionais [SAN 96]. Com o aumento da densidade de integração e o barateamento do hardware esta alternativa compõe uma forte tendência.

Uma arquitetura multifluxo pode executar múltiplos fluxos de instruções simultaneamente. Tal facilidade oferece oportunidades para que as limitações acima mencionadas sejam minimizadas. Em arquiteturas deste tipo, instruções que executam em paralelo podem ser extraídas de diferentes fluxos. Assim, os fluxos de instruções agem como

novas fontes de paralelismo, conferindo à arquitetura um desempenho potencialmente maior. Além disso, penalidades ocasionadas por dependências entre instruções podem ser mascaradas com o maior paralelismo proveniente da execução simultânea de múltiplos fluxos [CHA 96].

Em 1991, *Wolfe* [WOL 91] propôs uma arquitetura multifluxo como uma extensão de uma arquitetura VLIW. Neste modelo, o número de fluxos pode variar dinamicamente durante a execução do programa, mas a determinação destes fluxos é feita durante a compilação.

Hirata [HIR 91] propôs uma arquitetura multifluxo com um mecanismo de escalonamento dinâmico de instruções, no qual uma instrução é enviada para a unidade funcional quando não existem dependências de dados.

Sohi [SOH 95], em 1995, propôs um modelo que emprega execução especulativa, no qual cada fluxo corresponde à execução de um dos ramos originários de uma instrução de desvio.

Os processadores superescalares em particular possuem uma única, janela central de instruções de onde quatro, oito, ou mais instruções independentes devem ser extraídas e despachadas a cada ciclo. A arquitetura Multiescalar [SOH 95] que está sendo pesquisada no *Wisconsin Multiscalar Project* está direcionada para combater este problema. A idéia principal é dividir a janela central de instruções em pequenas tarefas (*tasks*) e associá-las a pequenas janelas para múltiplos elementos de processamento.

A arquitetura multiescalar combina elementos de ambos processamento seqüencial e multiprocessamento. Instruções que pertencem a uma tarefa são comumentemente seqüenciais, e a execução da tarefa em um único elemento de processamento tira vantagem deste fato (logo, pequenos níveis de paralelismo podem ser explorados internamente à tarefa com a utilização de processamento superescalar).

Tendo múltiplos elementos de processamento executando concorrentemente fluxos distintos de controle, o paralelismo entre tarefas pode ser explorado. O processamento multiescalar possui uma forte semelhança com o multiprocessamento, no entanto, existe uma diferença básica. O multiprocessamento requer sincronização explícita e comunicação entre as tarefas, provida pelo programador ou pelo compilador. Por outro lado, o processamento multiescalar mantém um modelo de programação seqüencial. Uma vez que não existe sincronização explícita, a execução concorrente de tarefas é especulativa por natureza, e quaisquer dependências de controle ou dados são resolvidas em hardware. A natureza especulativa do processamento multiescalar tráz grande potencial de paralelismo.

Algumas pesquisas têm sido publicadas propondo novos mecanismos de busca de instruções [WAL 98]. Outros tratam sobre novos mecanismos de previsão de desvios onde elevadas taxas de acerto são obtidas [KNI92, SEZ 96]. Ainda há uma área bastante repleta de espaços onde diversos trabalhos e pesquisas podem estar inseridas. Arquiteturas *Multithreaded* também vêm tomando importância no campo da pesquisa, onde o paralelismo de instrução é extraído de diferentes *threads* [TYS92, LEN92].

6 O Modelo Multifluxo Proposto

Neste capítulo, trata-se sobre as modificações introduzidas na arquitetura superescalar real com o intuito de eliminar o efeito das quebras de fluxo e disponiblizar um maior número de instruções ao despacho.

6.1 O Mecanismo de Busca Especulativa Empregado

Em [TAL 95], é apresentada uma técnica para a redução do problema dos desvios condicionais, denominada *Fetch Taken and not-Taken Paths*. Em [CHA 96], é apresentado um novo modelo de execução paralela de instruções baseado na técnica mencionada. Neste novo modelo, ambas as ramificações de um desvio são buscadas antecipadamente e armazenadas no *buffer* de busca permitindo o encadeamento das instruções que vêm após uma instrução de desvio, mesmo quanto a previsão for de desvio tomado. Esta técnica pretende eliminar as quebras de fluxo introduzidas quando são feitas previsões de desvios tomados.

Quando a previsão é feita, os dois possíveis caminhos do desvio, já estão presentes no *pipeline*. Desta forma, não se faz necessário redirecionar o estágio de busca ao caminho previsto, haja visto que as instruções pertencentes a este já se encontram no *pipeline*.

É desejável que a cada ciclo, toda a largura de busca seja preenchida com instruções, e que estas sejam transferidas para a fila de instruções, possibilitando a decodificação e posteriormente o despacho para execução, mesmo quando um desvio é previsto como tomado. Além disto, novas instruções podem ser colocadas no *buffer* de busca no ciclo imediatamente após uma previsão. Aliado a um mecanismo de previsão que ofereça altas taxas de acerto, este funcionamento pode resultar em um fluxo elevado de entrada na fila de instruções mascarando as quebras de fluxo.

Para realizar esta operação, a arquitetura deve detectar uma instrução de desvio no próprio estágio de busca e, no ciclo seguinte, iniciar a busca de instruções nos dois possíveis caminhos do desvio. Quando um desvio é previsto como tomado, a transferência para a fila de instruções não é interrompida após a instrução de desvio.

Nas máquinas superescalares convencionais (instruções de um único fluxo de controle) quando um desvio é previsto como tomado, a busca é redirecionada para o caminho não adjacente à instrução de desvio (quebra de fluxo). Neste caso, são introduzidos alguns ciclos até que o endereço de busca tenha sido corrigido e a instrução sucessora, no fluxo lógico, tenha sido acessada (latência de acesso). O número de ciclos desperdiçados depende da arquitetura em questão.

Na técnica acima, três fatores devem ser considerados:

- Taxa de acertos da previsão de desvios: o mecanismo de previsão de desvios deve garantir uma taxa alta de acertos, para que o encadeamento de instruções mantenha a continuidade da entrada de instruções;
- Latência de acesso às instruções: instruções no destino do desvio devem ser acessadas a tempo de serem encadeadas. O acesso de instruções no destino do desvio pode provocar falhas na memória cache;
- Possibilidade da criação de novos fluxos quando um novo desvio é encontrado:

profundidade de especulação na busca antecipada.

6.2 O Novo Estágio de Busca

Na arquitetura superescalar multifluxo analisada, o estágio de *Busca* foi modificado para permitir que ambos os caminhos de um desvio sejam acessados e armazenados tão logo o desvio tenha sido detectado.

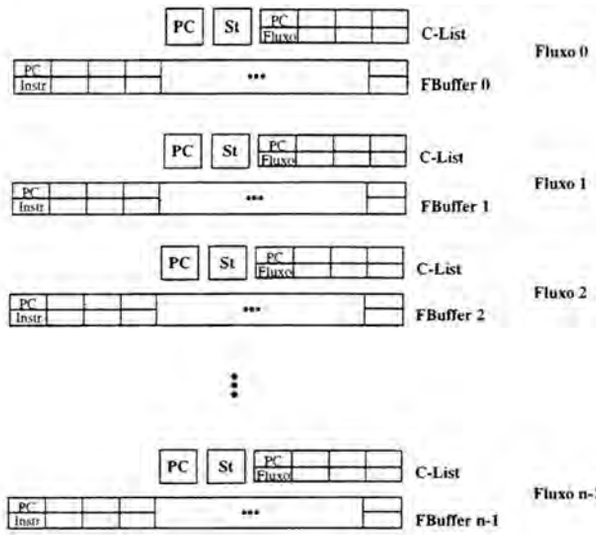

FIGURA 6.1 - Estrutura de Buffers do Novo Estágio de *Busca*

A figura 6.1 sugere a nova estrutura de *buffers* do estágio de *Busca* do pipeline multifluxo. O número de buffers determina a profundidade de especulação na busca de instruções.

Cada *buffer* possui quatro elementos independentes:

- Program Counter
- Status Bit
- Children List
- Fetch Buffer

No *Fetch Buffer* são armazenadas as instruções acessadas. O *PC* é empregado para apontar as instruções que estão ao longo de um fluxo. O *status bit* indica a ocupação da estrutura por algum fluxo e a *C-List* indica os fluxos filhos que se originam de um fluxo principal.

A *C-List* armazena o endereço de uma instrução de desvio e o identificador do seu fluxo filho, permitindo que o estágio de previsão redirecione a transferência automaticamente ao encontrar uma instrução de desvio.

6.3 Funcionamento da Arquitetura Multifluxo

Nesta subseção descreve-se o funcionamento dos estágios de *Busca* e *Previsão* modificados na arquitetura multifluxo.

O estágio de *Busca* acessa instruções colocando-as no *Fetch Buffer*. Ao detectar uma instrução de desvio, durante a busca, um novo fluxo é criado e inicializado, se houverem recursos disponíveis.

A criação de um fluxo compreende a atualização da *C-List* do *buffer* corrente com o endereço (*PC*) do desvio detectado e com a identificação do *buffer* alocado para armazenar as instruções que serão buscadas no caminho alvejado por este desvio. Para cada desvio encontrado são possíveis dois caminhos. O caminho não tomado segue sendo buscado e armazenado no mesmo buffer do desvio em questão, poupando recursos. Já o caminho tomado passa a ser buscado no ciclo seguinte e armazenado no buffer alocado para este fluxo.

A inicialização de um fluxo corresponde a inicialização do *PC* do *buffer* alocado com o endereço alvo do desvio e do *status bit* indicando a ocupação da estrutura.

O estágio de previsão transfere as instruções do fluxo principal para a fila de instruções como nas arquiteturas superescalares convencionais observando o tipo de cada instrução. Porém, ao encontrar uma instrução de desvio, a previsão é realizada e ao invés de redirecionar o estágio de *Busca*, em caso de previsão de desvio tomado, este estágio somente encadeia as instruções que estão no fluxo filho daquele desvio, descartando as instruções adjacentes. O fluxo principal passa a ser então o fluxo filho do desvio em questão.

Caso a previsão seja de desvio não tomado, o fluxo filho é descartado e as instruções que sucedem o desvio são transferidas para a fila de instruções. Quando um fluxo é descartado, todos os fluxos filhos originados a partir deste são também descartados de maneira recursiva liberando recursos para que novos fluxos sejam especulados.

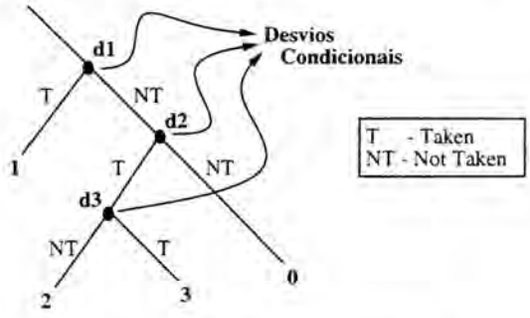

FIGURA 6.2 - Ocorrência de Desvios e Identificação de Fluxos

A figura 6.2 exemplifica a ocorrência de três desvios em um trecho de programa hipotético mostrando como os fluxos são numerados e identificados. Para cada desvio que acontece, um novo fluxo é criado e inicializado. As instruções que se encontram no caminho que seria previsto como não tomado fazem parte do mesmo fluxo em que está o desvio, ou melhor, do fluxo principal.

6.4 Estudo de Caso

Considere o trecho de código abaixo 6.3. Considere também que todos os desvios são previstos como tomados, pior situação, e que o fluxo de controle correto para o trecho de programa exemplo é o que aparece circundado. Neste exemplo, as instruções *d1, d2, d3 e d4* são instruções de desvio condicional que possuem dois possíveis caminhos: *T:Taken* e *NT:Not-Taken*.

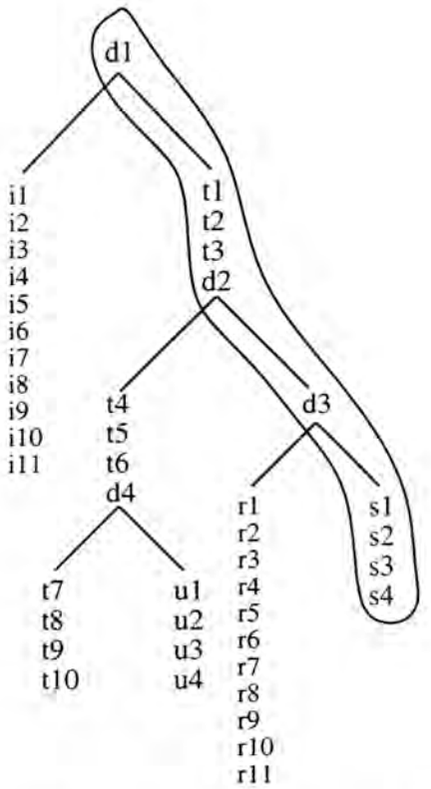

FIGURA 6.3 - Exemplo de Trecho de Código

A figura 6.4 mostra a ocorrência das quebras de fluxo quando desvios são previstos como tomados numa arquitetura que possui largura de busca e despacho ambas de 4 instruções. Neste caso, sempre que um desvio é previsto como tomado, é inserido um ciclo adicional para que o estágio de busca possa acessar as instruções no caminho alvejado. Observe que todos os estágios que se encontram achurados representam uma perda potencial, pois quatro instruções poderiam estarem sendo processadas nestes estágios. A partir do momento em que a primeira instrução atinge o estágio de despacho (*c+3*), são necessários 7 ciclos para que todas as 10 instruções *d1, t1, t2, t3, d2, d3, s1, s2, s3, s4* sejam despachadas, conforme o exemplo de trecho de código utilizado.

Após estes 7 ciclos foram despachadas 10 instruções, o que caracteriza uma média de despacho de *1,42* instruções/ciclo. Isto é o que normalmente acontece numa arquitetura capaz de acessar instruções de somente um dos fluxos de um desvio.

Considere o exemplo de quebra de fluxo da figura 6.4. Ao introduzir o mecanismo de múltiplos fluxos apresentado neste capítulo, obter-se-ia uma média de *2.0* instruções prontas para despacho por ciclo, considerando a possibilidade de acessar somente instruções de 2 fluxos por ciclo. A figura 6.5 ilustra esta nova situação.

Observa-se que uma média de *2.0* instruções são despachadas por ciclo contra ape-

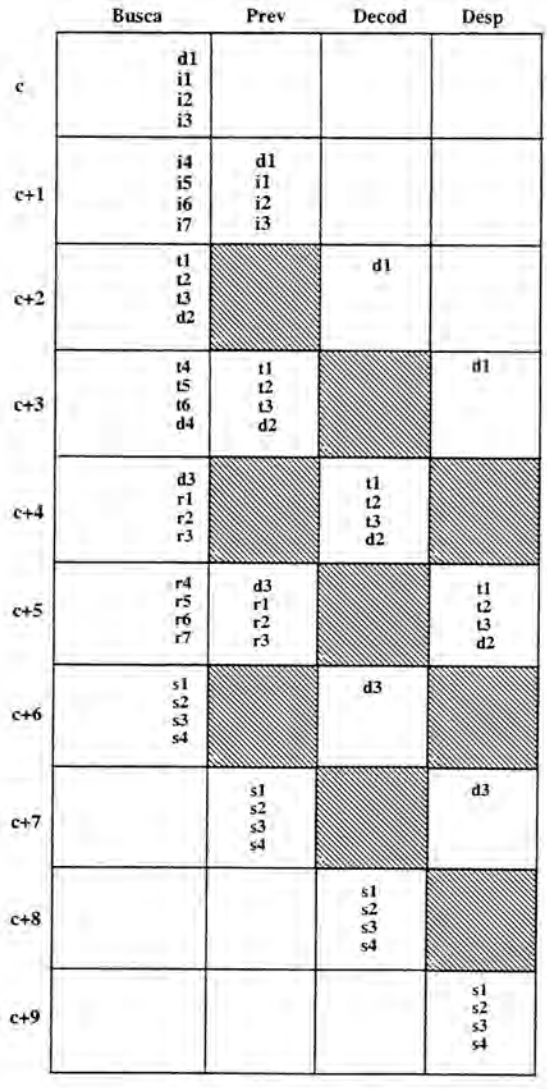

FIGURA 6.4 - Exemplo de Quebra de Fluxo

nas *1.42* instruções despachadas no primeiro caso. Isto representa um aumento de *40%* ao empregar a técnica de busca especulativa utilizando apenas dois *buffers* de busca.

Considere agora o exemplo da figura 6.6. Neste caso, foram utilizados 3 *buffers* de busca possibilitando acessar instruções de 3 diferentes fluxos por ciclo. A média de instruções prontas para despacho aumenta para *2.5* instruções por ciclo, o que representa um ganho de *76.05%* em relação so primeiro exemplo onde apenas um fluxo é acessado por ciclo.

Observe também, que a posição do desvio dentro do bloco de instruções acessado influi diretamente na média de instruções despachadas. Isto acontece, por exemplo, com as instruções *d1* e *d3* que estão na primeira posição do bloco acessado.

Otimizações em tempo de compilação poderiam ser introduzidas para rearranjar o código tentando manter as instruções de desvio sempre nas posições mais próximas ao fim do bloco de instruções acessadas. Este tipo de otimização esta intimamente relacionada com as características da arquitetura, neste caso, com a largura de busca.

O modelo de busca especulativa aqui apresentado possibilita um aumento do fluxo de instruções prontas para despacho e posterior execução. Os casos aqui apresentados possibilitaram uma análise qualitativa do efeito proporcionado pelo emprego desta nova

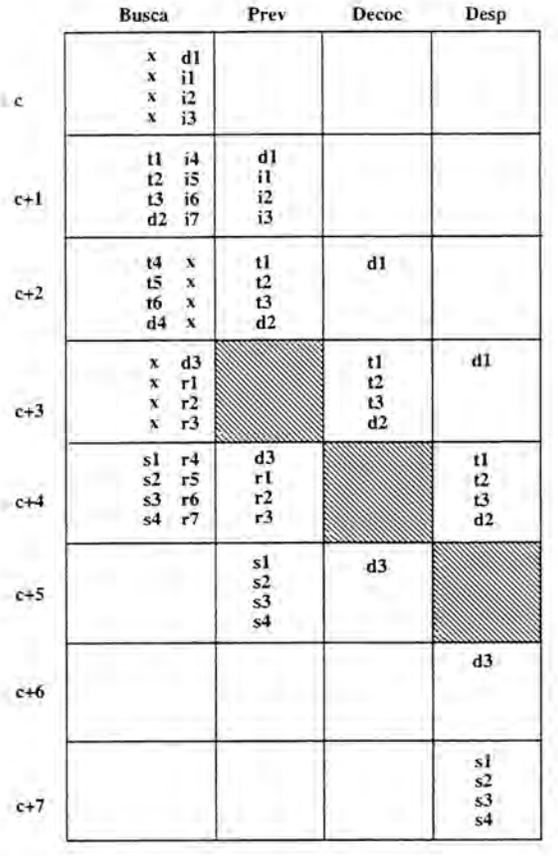

FIGURA 6.5 - Exemplo de Redução do Efeito das Quebras de fluxo, com 2 Fluxos

técnica. Situações próximas as reais serão apresentadas, estudadas e analisadas posteriormente no decorrer desta pesquisa.

Pequenas modificações introduzidas na arquitetura, com razoável simplicidade e baixo nível de complexidade podem oferecer um aumento significativo do paralelismo disponível. Não se tem conhecimento, num primeiro momento, quais são as limitações deste modelo de busca e quais são os possíveis problemas que estão ligados a busca especultaiva de instruções a não ser o maior exigência no que diz respeito ao acesso a memória cache de instruções.

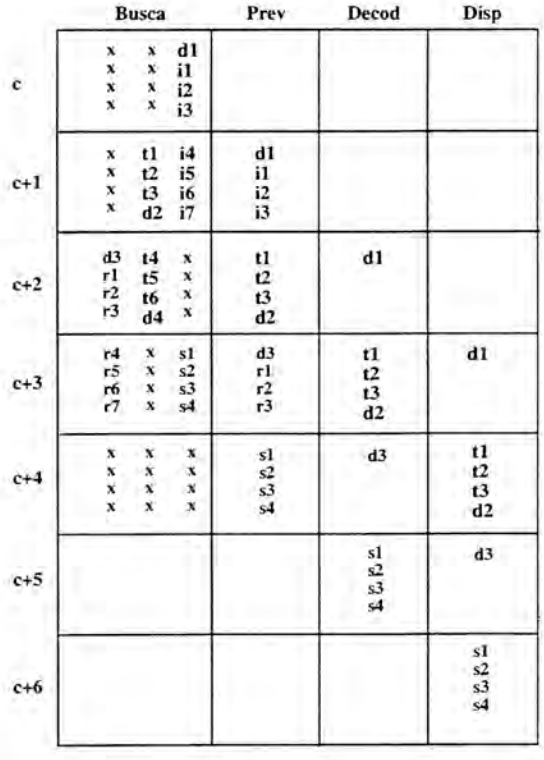

FIGURA 6.6 - Exemplo de Redução do Efeito das Quebras de Fluxo, com 3 Fluxos

7 Ambiente Experimental

A presente pesquisa tem como objetivo principal avaliar o desempenho de uma arquitetura superescalar multifluxo. Com o intuito de fornecer resultados consistentes foram realizadas centenas de simulações utilizando-se 4 (quatro) arquiteturas diferentes e 4 (quatro) benchmarks do conjunto SPECint95 [SPE 95].

Nesta seção descreve-se o funcionamento de cada um dos simuladores e apresenta-se uma descrição sobre os benchmarks utilizados nas simulações.

Foram empregados nos experimentos os programas *compress, go, li* e *ijpeg* todos pertencentes ao conjunto SPECint95. Cada simulação realizada cobriu a execução de *10.000.000* de instruções para cada programa de teste.

7.1 Metodologia

A figura 7.1 representa a metodologia utilizada para avaliação da arquitetura multifluxo. Cada um dos programas de benchmark foi compilado através do compilador *gcc* em ambiente SunOS 4.1.1 gerando executáveis segundo o padrão SPARC V7.

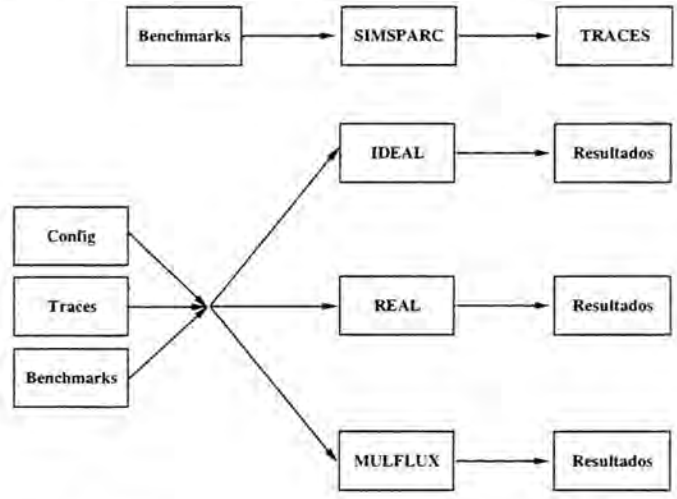

FIGURA 7.1 - Metodologia de Simulação

Os benchmarks foram então submetidos ao simulador SIMSPARC [CHA 94]. Este simulador é do tipo execution-driven, e através da simulação de execução dos programas, foram gerados arquivos de *traces* que permitiram mapear todas as ocorrências de desvios condicionais e seus resultados, traçando o fluxo de controle dos programas durante a execução de *10.000.000* de instruções.

Após a geração dos *traces*, foram es tabelecidas diferentes configurações de arquiteturas superescalares, veja seção 7.3. Foram empregados então, 3 simuladores de arquiteturas superescalares com características específicas que permitiram obter análises de desempenho sob diferentes circunstâncias. Estes simuladores, veja seção 7.2, são do tipo *trace-driven* e seguem o fluxo de controle determinado pela execução dos programas através do SIMSPARC.

O objetivo principal é analisar o desempenho do modelo multifluxo comparando-o

com o desempenho ideal e o desempenho obtido por uma arquitetura real visando analisar o ganho potencial oferecido pelo modelo em questão.

Para cada um dos simuladores superescalares foram fornecidas como entradas os programas de benchmark, arquivos de configuração que determinaram o perfil da arquitetura e os arquivos de *traces* gerados anteriormente. Como saída obteve-se dados de desempenho que permitiram analisar o comportamento destas arquiteturas para cada um dos benchmarks.

7.2 Simuladores

Os simuladores de 3 arquiteturas superescalares empregados são do tipo *trace-driven* que utilizam arquivos de *trace* gerados por um simulador da arquitetura SPARC V7 [SUN 87]. Abaixo descreve-se o funcionamento de cada arquitetura superescalar empregada nos experimentos.

1. Simulador Ideal
   Simula a execução de programas fazendo previsão de desvios perfeita. Já no estágio de *Busca (fetch)* o simulador lê os arquivos de *trace* e passa a buscar instruções no fluxo correto de execução do programa eliminando totalmente a latência de acesso normalmente causada pela ocorrência de instruções de desvio.

2. Simulador Real
   Simula a execução de programas fazendo previsão de desvios em dois níveis [YEH 91]. Nestas simulações verificam-se as causas principais do esvaziamento da fila de instruções em função da ocorrência de desvios e a conseqüente queda de desempenho.

3. Simulador Multifluxo
   Simula a execução de programas utilizando a técnica de busca especulativa em múltiplos fluxos. Nestas simulações verifica-se a redução drástica do esvaziamento da fila de instruções e um aumento potencial no paralelismo disponível.

A principal diferença entre os simuladores encontra-se nos estágios de *Busca* e *Previsão* conforme a descrição acima. O simulador *Ideal* representa uma arquitetura que oferece desempenho próximo ao ideal, limitado somente pelas dependências de dados e pelo conflito de recursos. O simulador *Real*, representa uma arquitetura com características encontradas nas arquiteturas superescalares convencionais encontradas no mercado. E por fim, o simulador *Mulflux* representa o modelo de busca especulativa estudado.

7.3 Configuração dos Simuladores e Resultados Fornecidos

Os simuladores empregados nos experimentos apresentados neste trabalho são parametrizados conforme arquivos de configuração definidos pelo usuário. Conforme a tabela 7.1, estes arquivos permitem definir:

Após executar um programa, conforme as configurações fornecidas, os simuladores geram arquivos de resultado com as seguintes informações:

TABELA 7.1 - Parâmetros de Configuração dos Simuladores

1.	Número de instruções a serem simuladas
2.	Número e Tipo de Unidades Funcionais
3.	Número de Estações de Reserva
4.	Número de Fluxos (no caso do simulador *Mulflux*)
5.	Largura de Busca
6.	Tamanho da fila de Instruções
7.	Largura de Despacho
8.	Número de Barramentos de Resultados
9.	Tamanho do Buffer de Reordenação (*Reorder Buffer - ROB*)
10.	Tamanho e Associatividade da Cache L2
11.	Tamanho e Associatividade da Icache
12.	Tamanho e Associatividade da Dcache
13.	Tamanho e Associatividade da *History Register Table - HRT*
14.	Tamanho da *Pattern History Table - PHT*
15.	Latência das Instruções
16.	Ponto de Entrada para Execução do Programa

7.4 Benchmarks

Foram empregados nos experimentos os programas *compress, go, ijpeg* e *li* todos pertencentes ao conjunto SPECint95. Todas as simulações realizadas cobriram a execução de *10.000.000* de instruções para cada programa de teste.

Os dados apresentados na tabela 7.3 foram extraídos a partir da execução seqüencial de cada um dos *benchmarks*. Esta execução foi realizada a partir do simulador *SIMSPARC - execution-driven* da arquitetura SPARC, gerando os arquivos de *traces* que foram utilizados nas demais simulações.

TABELA 7.2 - Resultados de Saída dos Simuladores

1	Número de instruções efetivamente simuladas
2	Número de Ciclos gastos na simulação
3	*Speed-up*, em função da execução escalar (*SIMSPARC*)
4	Média de Instruções Executadas por Ciclo (*IPC*)
5	Número Total de Instruções Acessadas
6	Média de Instruções Acessadas por Ciclo
7	Taxas de *Hit* das caches L2, Icache e DCache
8	Taxa de *Hit* da *BHT*
9	Precisão da *PHT*
10	Precisão do Mecanismo de Previsão
11	Número Total de Desvios Previstos
12	Número de Desvios Previstos como Tomados e Não-Tomados (*T* e *NT*)
13	Número de Desvios Executados
14	Número de Desvios Tomados e Não-Tomados (*T* e *NT*)
15	Percentual de Ciclos em que não houve Previsão devido a fila de instruções cheia ou buffer de busca vazio
16	Perfil de Despacho
17	Média de Instruções Despachadas por Ciclo
18	Percentual de Despacho Nulo devido a Indisponibilidade de Recursos
19	Percentual de Despacho Nulo devido a *Reorder Buffer* Cheio
20	Percentual de Despacho Nulo devido a Estações de Reservas Lotadas
21	Percentual de Despacho Nulo devido a Profundidade de Especulação
22	Percentual de Despacho Nulo devido a Previsões Incorretas
23	Percentual de Despacho Nulo devido a Quebras de Fluxo
24	Penalidade causada pelos esvaziamentos da Fila de Instruções
25	Penalidade causada pelas previsões incorretas
26	Penalidade causada pela Especulação Incorreta
27	Perfil do Término de Execução
28	Utilização de cada Unidade Funcional

TABELA 7.3 - Características dos Benchmarks

Benchmarks	Instruções Simuladas	Ciclos	CPI	Desvios Condicionais	(%)	Tomados (%)	Não-Tomados (%)
compress	10×10^6	12717265	1.27	1055630	11%	82.92	17.08
go	10×10^6	12572511	1.26	1070617	11%	70.99	29.01
ijpeg	10×10^6	12656689	1.27	2301463	23%	80.82	19.18
li	10×10^6	12982557	1.30	2023238	20%	74.00	26.00

8 Analisando o Desempenho do Modelo Multifluxo

Neste capítulo analisa-se o desempenho do modelo multifluxo baseado em dados extraídos através de diversas simulações. Os experimentos realizados simularam diferentes configurações para a máquina real e para a máquina multifluxo. Os testes iniciaram variando-se a largura de busca entre 6, 8, 12 e 16 instruções por ciclo para 2, 4 e 8 fluxos no modelo multifluxo. Para a máquina real simularam-se apenas a variação da largura de busca, já que somente um fluxo pode ser acessado por vez.

O gráfico da figura 8.1 apresenta a porcentagem de ciclos em que ocorreu despacho nulo para as quatro máquinas diferentes. *Mulflux-2*, *Mulflux-4* e *Mulflux-8* correspondendo às simulações da máquina multifluxo para 2, 4 e 8 fluxos respectivamente, e *Real* às simulações da máquina real.

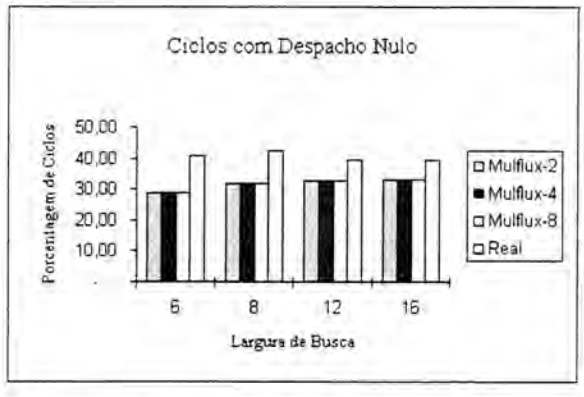

FIGURA 8.1 - Comparação de Ciclos c/ Despacho Nulo

Observa-se que em todas as situações o modelo multifluxo apresentou taxas menores de ocorrência de ciclos com despacho nulo. Na máquina *Real*, a taxa de despacho nulo diminui à medida em que a largura de busca aumenta, indo de *40,50%*, para largura de busca igual a 6, para *39,30%* em configurações com largura de busca igual a 16 instruções por ciclo.

As taxas de despacho nulo no modelo multifluxo, apesar de serem menores do que na máquina *Real*, crescem à medida em que a largura de busca é aumentada. Isto ocorre devido ao aumento das previsões erradas e ao aumento da indisponibilidade de recursos. Outros estudos estão sendo realizados objetivando a utilização de mecanismos mais eficientes de previsão de desvios e o balanceamento ideal da arquitetura. Com isto, pode-se diminuir então os ciclos com despacho nulo à medida em que a taxa de especulação, na busca, é aumentada. Para os demais casos, utilizou-se apenas as configurações com dois fluxos devido à melhor relação custo e desempenho apresentada.

Os gráficos das figuras 8.2 e 8.3 apresentam a variação das componentes que causam despacho nulo nas máquinas *Mulflux* e *Real*, respectivamente. A soma das três componentes fornece a porcentagem de ciclos em que ocorreu despacho nulo.

A discrepância entre as componentes na máquina *Real* é notável. A ocorrência de fila vazia é a componente que mais contribui para a existência de despacho nulo, fator que não se verifica na máquina *Mulflux*. É de se esperar que a partir do momento em que existem mais instruções disponíveis para despacho, e posterior execução, o número de recursos disponíveis torne-se o fator determinante para a ocorrência de despacho nu-

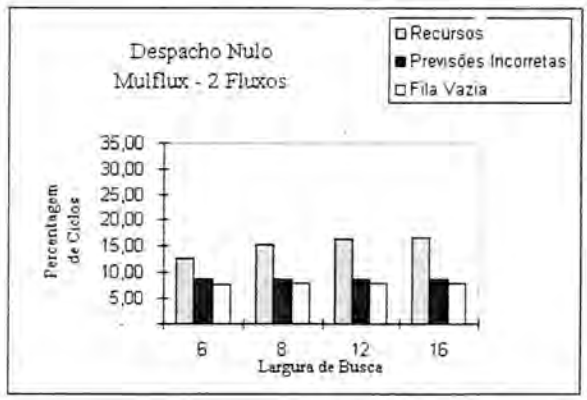

FIGURA 8.2 - Componentes que Causam Ciclos com Despacho Nulo na Máquina *Mulflux*

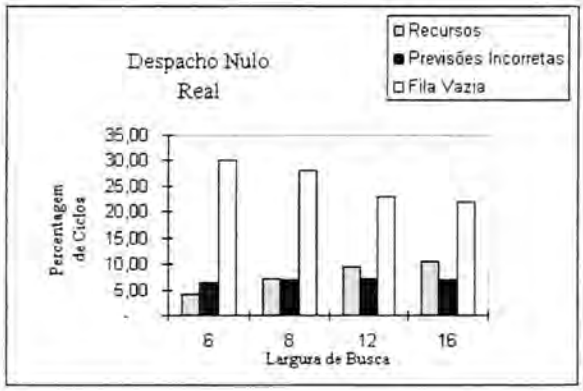

FIGURA 8.3 - Componentes que Causam Ciclos com Despacho Nulo na Máquina Real

lo. Este fator é observado nitidamente na máquina *Mulflux* onde esta torna-se a maior componente.

A ocorrência de fila vazia na máquina *Real* decresce de *30,05%*, com largura de busca igual a 6, para *21,79%* com largura de busca igual a 16. Na máquina *Mulflux* esta taxa fica entre *7,73%* e *7,80%* para as mesmas configurações.

Os gráficos das figuras 8.4 e 8.5 provam a eficiência do modelo multifluxo na redução da ocorrência de fila vazia devido as quebras de fluxo, para as máquinas *Mulflux* e *Real*, respectivamente. Na máquina *Mulflux* as quebras de fluxo não ultrapassam os *5,24%* enquanto que na máquina *Real* esta porcentagem chega aos *27,07%*.

O modelo multifluxo possibilitou uma redução em torno de *74,24%* da ocorrência de fila vazia. O efeito das quebras de fluxo foram reduzidos em *80,64%* na máquina *Mulflux*. No entanto, percebe-se a importância de concentrar os próximos trabalhos no balanceamento e profundidade de especulação da arquitetura.

8.1 Analisando o Impacto da Implementação do Modelo Multifluxo

Antes de considerar a implementação do modelo multifluxo, deve-se considerar primeiramente o impacto sobre os blocos adjacentes. Os problemas relacionados com a largura de busca e o acesso à memória cache de instruções serão brevemente discutidos nesta seção.

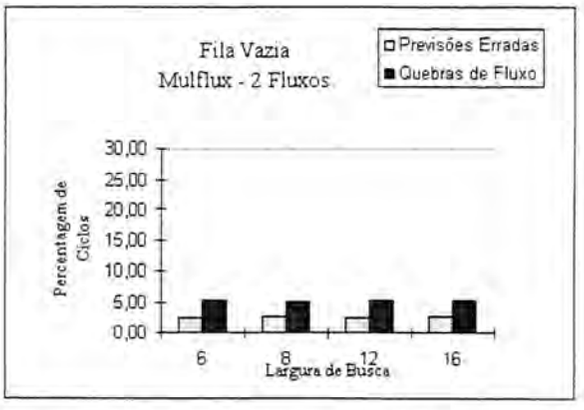

FIGURA 8.4 - Fatores que Causam Esvaziamento da Fila de Instruções na Máquina *Mulflux*

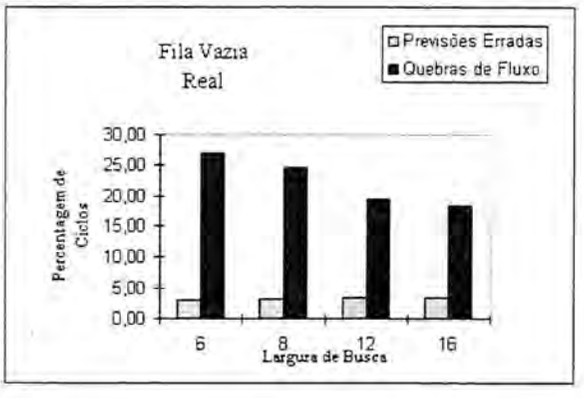

FIGURA 8.5 - Fatores que Causam Esvaziamento da Fila de Instruções na Máquina Real

Nos resultados apresentados na seção anterior, não foram considerados os problemas que envolvem a cache de instruções, ou seja, supôs-se a utilização de um mecanismo de cache perfeito, sem falhas. O uso do modelo multifluxo pode gerar um número maior de falhas na cache de instruções devido ao aumento do número de acessos. O número total de instruções acessadas aumenta quando se faz necessário acessar instruções em diferentes fluxos e o desempenho do modelo pode ser afetado por este fator.

Alguns experimentos foram realizados empregando-se uma cache de instruções semelhante a encontrada no processador Pentium [BEK 91] para observar a performance do modelo sob condições próximas à real. Na cache utilizada, a latência introduzida por uma falha (*miss*) é de 3 ciclos na icache L1, e 13 ciclos quando a falha propaga-se para a cache L2.

Também nos experimentos anteriores, considerou-se que a largura de busca é multiplicada pelo número de *buffers* válidos. Por exemplo, se a largura de busca é igual a 8 instruções por ciclo, e existem 4 *buffers* de busca inicializados, efetivamente buscar-se-iam 32 instruções num único ciclo. Isto foi assumido em todos os experimentos realizados com a máquina *Mulflux*.

Para reduzir os problemas relacionados com a largura de busca e aumento de acessos à cache, foi proposto um mecanismo denominado *Dynamic Split Fetch - DSF* que consiste em dividir a largura de busca entre os fluxos inicializados e válidos. Assim, por exemplo, se existem 4 *buffers* de busca inicializados, e a largura de busca é de 8 instruções por ciclo, serão buscadas 2 instruções para cada fluxo mantendo-se a largura de busca efe-

tiva em 8 instruções.

Quando a largura de busca aumenta na máquina *Mulflux*, figura 8.1, os ciclos com despacho nulo, onde nenhuma instrução é despachada para as estações de reserva, aumentam também. Isto ocorre porque o conflito de recursos tornou-se o maior problema na máquina *Mulflux* assim como acontecera nas simulações com a máquina *Ideal*.

8.2 Analisando o Desempenho Final

Nesta seção será analisado o desempenho final obtido ao empregar-se o modelo multifluxo comparando-o com o desempenho da arquitetura *Real*. Alguns problemas foram detectados e serão também discutidos.

O gráfico da figura 8.6 mostra que o aumento do número de unidades funcionais proporciona aumento de desempenho na máquina *Mulflux*.

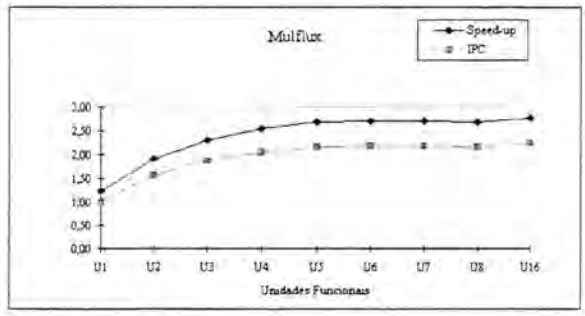

FIGURA 8.6 - Aumento do Desempenho ao Aumentar o Número de Unidades Funcionais na Máquina *Mulflux*

Entretanto, ao observar a taxa de despacho nulo, quando aumenta-se o número de unidades funcionais, percebe-se que esta aumenta juntamente com o número de unidades acrescidas. Isto deve-se ao fato de que fora mantida a mesma profundidade de especulação para todas as configurações. A máquina analisada consistiu de n unidades funcionais e uma única unidade de desvios. Quando esta se encontra saturada, o despacho de instruções é bloqueado caracterizando indisponibilidade de recursos. Isto é mostrado nos gráficos das figuras 8.7 e 8.8.

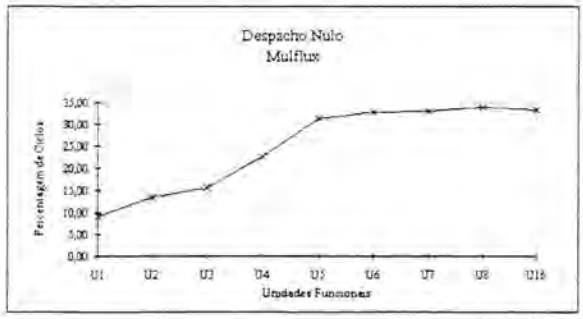

FIGURA 8.7 - Aumento do Despacho Nulo ao Aumentar o Número de Unidades Funcionais na Máquina *Mulflux*

Com estes resultados prova-se que o modelo multifluxo apresenta características semelhantes à arquitetura ideal onde as limitações estão estritamente relacionadas à indisponibilidade de recursos. Esta é a maior componente da ocorrência de despachos nulo

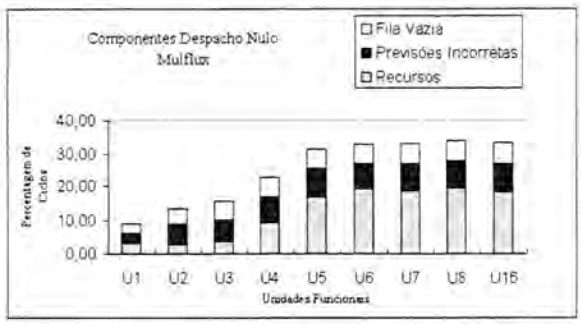

FIGURA 8.8 - Componentes do Despacho Nulo ao Aumentar o Número de Unidades Funcionais na Máquina *Mulflux*

no modelo analisado, logo, a profundidade de especulação assim como o balanceamento da arquitetua devem ser analisados cuidadosamente.

Para analisar o problema que envolve a cache de instruções, foram realizadas simulações com cinco máquinas diferentes:

- *Mulflux* com cache de instruções perfeita (sem falhas);
- *Mulflux* com cache de instruções normal;
- *Mulflux* com cache de instruções normal e *Dynamic Split Fetch*;
- *Real* com cache de instruções perfeita, e;
- *Real* com cache de instruções normal;

A cache de instruções normal utilizada assemelha-se à encontrada nos processadores Pentium [BEK 91]. Os gráficos que serão analisados agora, foram obtidos para configurações idênticas para cada máquina, sendo:

- Largura de Busca de 8 instruções;
- Buffer de Busca e Fila de Instruções com 16 e 32 posições, respectivamente;
- Largura de Despacho de 8 instruções;
- 8 Unidades Funcionais omogêneas com 8 estações de reserva cada;
- 1 Unidade de Desvios com 8 estações de reserva;
- 8 Barramentos de Resultado, e;
- Buffer de Reordenação com 64 posições.

O gráfico 8.9 apresenta o percentual de ciclos com despacho nulo para cada uma das máquinas. A máquina multifluxo testada foi configurada com 4 *buffers de busca* e apresentou o melhor desempenho utilizando cache perfeita e sem dividir a largura de busca pelos fluxos válidos.

Observe que a taxa de despacho nulo aumenta ao inserir-se o mecanismo de cache normal na máquina *Mulflux*, porém permanece similar ao introduzir-se o mecanismo de *DSF* com cache normal. Esta alternativa permite uma redução do número de acessos à cache de instruções pois a largura de busca total é dividida entre todos os fluxos possíveis.

A taxa de despacho nulo aumenta sensivelmente nas configurações da máquina *Real*. O aumento na taxa de despacho nulo ao introduzir-se cache normal na máquina *Real* é proporcional ao aumento introduzido ao inserir-se o mesmo mecanismo na máquina *Mulflux*.

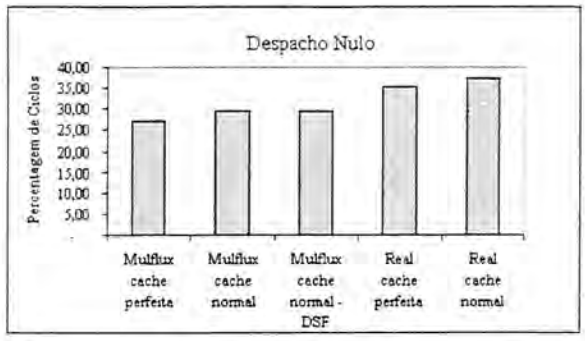

FIGURA 8.9 - Comparação do Percentual de Ciclos com Despacho Nulo

A figura 8.10 apresenta os componentes que causaram despacho nulo em cada uma das configurações. Neste gráfico observa-se nitidamente que na máquina multifluxo a ocorrência de fila vazia é a menor componente. A situação assemelha-se ao comportamente de uma máquina ideal onde as dependências de controle não afetam drasticamente o desempenho da arquitetura como acontece na máquina *Real*. Outro aspecto importante e que caracteriza uma forte aproximação ao modelo ideal é que a contenção de recursos é o principal causador de despachos nulos.

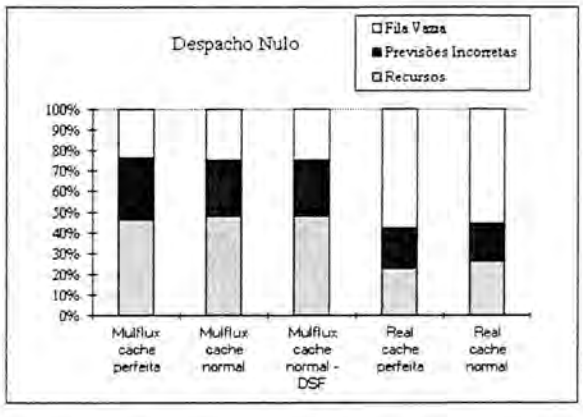

FIGURA 8.10 - Componentes do Despacho Nulo

No gráfico da figura 8.11, percebe-se a diminuição drástica das quebras de fluxo na máquina *Mulflux*. Estas aumentam ao introduzir-se a cache normal, mas permanecem razoavelmente mais baixas do que na máquina *Real* com cache perfeita ou normal.

Por fim, a figura 8.12 mostra o speed-up e a média de instruções executadas por ciclo em cada máquina. Neste gráfico nota-se que o speed-up não aumenta significativamente pois a indisponibilidade de recursos tornou-se maior na máquina multifluxo.

Estudos relativos ao balanceamento da arquitetura multifluxo e à profundidade de especulação utilizada serão realizados futuramente na busca de uma configuração que ofereça desempenho final próximo ao da máquina *Ideal*.

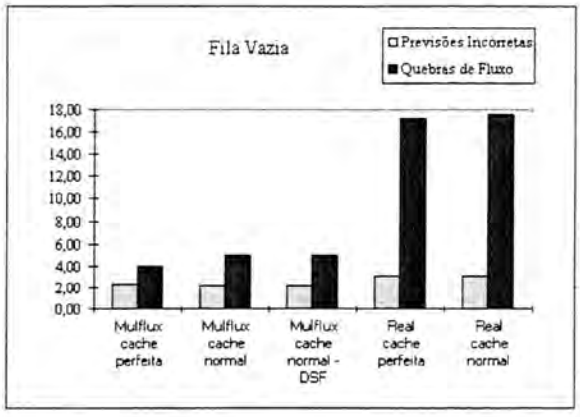

FIGURA 8.11 - Fatores que Contribuem para a Ocorrência de Fila Vazia

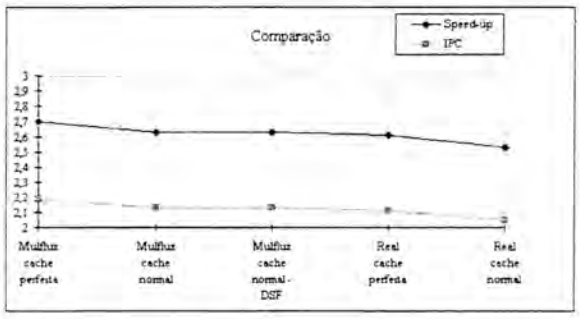

FIGURA 8.12 - Speed-up e IPC das Máquinas *Mulflux* e *Real*

9 Conclusões

O *speed-up* de uma arquitetura superescalar é diretamente proporcional ao seu *IPC*, número médio de instruções executadas por ciclo. E, é esperado obter-se um *speed-up* proporcional ao número de unidades funcionais paralelas existentes no processador. No entanto, as constantes quebras de fluxo ocasionadas pela previsão de desvios como tomados fazem com que a fila de instruções seja freqüentemente esvaziada e que não existam instruções para despacho. Logo, o *IPC* é reduzido drasticamente pelo esvaziamento da fila de instruções e o *speed-up* esperado não é alcançado em razão deste fator principal.

Muitos programas oferecem paralelismo suficiente para executar muitas instruções por ciclo, e os processadores superescalares têm sido projetados para decodificar e executar um número cada vez maior de instruções. No entanto, devido às dependências de controle, mecanismos convencionais de busca e previsão de desvios não dão suporte a esta alta demanda.

Não são recentes os estudos nesta área, mas somente nos últimos anos tem sido viável a utilização de mecanismos mais agressivos de exploração do paralelismo, principalmente a nível de processador, em função do baixo custo do hardware e do aumento da densidade de integração dos circuitos, fator que tende a estagnar, impondo um limite físico.

É necessária a existência de mecanismos que possibilitem a extração de mais paralelismo do código, assim como a obtenção de mais paralelismo de máquina.

Em geral, arquiteturas *VLIW* e superescalares buscam mais de uma instrução em cada acesso à memória. Apesar disto, nestas arquiteturas, as instruções buscadas em cada acesso fazem parte de um único fluxo lógico de instruções. Nota-se que a existência de dependências, entre as instruções pertencentes ao fluxo em questão, pode afetar drasticamente o escalonamento destas instruções, uma vez que o escalonador não tem alternativas para contornar este problema.

Uma regra básica na arquitetura de computadores é que o processador não pode executar uma aplicação mais rapidamente do que pode acessar as instrucões que compõem o programa de aplicação. Como demonstrado e analisado neste trabalho, as dependências de controle geram constantes quebras de fluxo fazendo com que o mecanismo de busca de instruções não consiga manter a fila de instruções cheia. Logo, o desempenho máximo teórico do processador não é atingido.

Neste trabalho, foi analisado um novo modelo de busca de instruções que permite o encadeamento de instruções provenientes de diferentes fluxos lógicos. O modelo multifluxo mostrou-se eficiente na redução da ocorrência de fila vazia.

Encadeando instruções provenientes de fluxos lógicos distintos, este modelo atenua o efeito causado pelas constantes quebras de fluxo oriundas da previsão de desvios tomados. Este modelo possibilitou uma redução de aproximadamente *74,27%* da ocorrência de fila vazia. O efeito causado pelas quebras de fluxo sofreu redução em torno de *80,64%* na máquina *Mulflux*.

Outro aspecto interessante é que na máquina *Mulflux* os despachos nulos aumentaram a medida em que aumentou-se a largura de busca enquanto na máquina *Real* estes diminuiram. Isto explica-se devido ao aumento da indisponibilidade de recursos, devido ao maior fluxo de instruções e ao aumento do número de previsões erradas.

Apesar de reduzir drasticamente a ocorrência de fila vazia, verificou-se que a diminuição dos despachos nulos não decresceu como esperado. Na máquina *Mulflux*

as taxas de despacho nulo ficaram entre *28,99%* e *33,11%* enquanto na máquina *Real* estas taxas ficaram entre *39,30%* e *40,50%*. O aumento do fluxo de instruções na fila de instruções causou um maior conflito de recursos o que fez com que os ciclos com despacho nulo, no modelo multifluxo, não fossem reduzidos drasticamente. Configurações com um maior número de unidades funcionais podem oferecer melhores resultados se balanceadas e configuradas para dar vazão ao maior fluxo de instruções oferecido pelo modelo em questão.

Nos experimentos realizados, a cache de instruções normal apresentou desempenho semelhante nas máquinas *Mulflux* e *Real*. A utilização da técnica *Dynamic Split Fetch - DSF* constituiu uma estratégia vantajosa que permite manter constante o número de linhas acessadas na cache de instruções.

A priori podemos detectar um outro gargalo no sistema ao empregar o modelo multifluxo. A profundidade de especulação determina o número de desvios que podem ser especulativamente executados. Se o número médio de instruções aumenta devido a maior quantidade de instruções que tendem a preencher a fila de instruções, é muito provável que se tenha também um número maior de desvios a serem executados. A saturação da unidade de desvios pode retardar a execução das demais instruções e mascarar o potencial desempenho oferecido pelo modelo multifluxo.

Outros estudos estão em andamento e farão parte da seqüência deste trabalho. A avaliação do impacto da implantação do modelo multifluxo em arquiteturas reais, sob a ótica da cache de instruções, é um deles. O aumento do número de acessos à cache ou o acesso a pontos distantes em instantes próximos pode tornar o sistema de cache um gargalo.

Outro estudo importante reflete o balanceamento da arquitetura como um todo, a determinação do número ideal de unidades funcionais, largura de busca, largura de despacho, profundidade de especulação e tamanho do buffers estão sendo estudados e considerados como fatores fundamentais para o desenvolvimento das próximas etapas.

Bibliografia

[ALP 93] ALPERT, Donald; AVNON, Dror. Achitecture of the Pentium Microprocessor. **IEEE Micro**, New York, p. 11-21, June 1993.

[BEC 93] BECKER, Michael C. et al. The PowerPC 601 Microprocessor. **IEEE Micro**, New York, p. 54-67, October 1993.

[BEK 91] BEKERMAN, Michael; MENDELTON, Avi. Analyzing Pentium Performance. **IEEE Micro**, New York, p. 72-83, October 1991.

[CHA 94] CHAVES FILHO, Eliseu Monteiro. **Arquiteturas Super Escalares: Efeito de Alguns Parâmetros sobre o Desempenho**. Rio de Janeiro: COPPE/UFRJ, 1994. Tese de Doutorado.

[CHA 94a] CHAKRAVARTY, Dipto; CANNON, Casey. **PowerPC:** Concepts, Architecture, and Design. [S.I.]: McGraw Hill, 1994.

[CHA 95] CHAVES FILHO, Eliseu M.; FERNANDES, Edil S. T. On the Performance of Superscalar Processors. **Journal of the Brazilian Computer Society**, Rio de Janeiro, v.2, n.1, p. 38-48, July 1995.

[CHA 96] CHAVES FILHO, Eliseu M. et al. Uma Arquitetura Super Escalar com Múltiplos Fluxos de Instruções. In: SBAC-PAD, 8., 1996, Recife. **Anais...** Recife: SBC/UFPE, 1996. p.67-77.

[EDM95] EDMONDSON, John H. et al. Superscalar Instruction Execution in the 21164 Alpha Microprocessor. **IEEE Micro**, New York, p. 33-43, April 1995.

[FER 92] FERNANDES, Edil S. T.; SANTOS, Ana D. **Arquiteturas Super Escalares:** Detecção e Exploração do Paralelismo de Baixo Nível. Porto Alegre: II da UFRGS, 1992. 135p. Trabalho apresentado na Escola de Computação, 8., 1992, Gramado.

[FIS 81] FISHER, J. A. A Trace Scheduling: A Technique for Global Microcode Compactation. **IEEE Transactions on Computers**, New York, v.30, n.7, p. 478-490, July 1981.

[FLY 72] FLYNN, M. J.; Some Computer Organizations and Their Effectiveness. **IEEE Transactions on Computers**, New York, v.21, n.9, p. 948-960, September 1972.

[GRO 90] GROHOSKI, G. F. Machine Organization of the IBM RISC System/6000 Processor. **IBM Journal of Research and Development**, [S.I.], v.34, n.1, p.37-58, January 1990.

[HEN 96] HENNESSY, John L.; PATTERSON, David A **Computer Architecture:** A Quantitative Approach. 2.ed. Palo Alto: Morgan Kaufmann, 1996.

[HIR 91] HIRATA H. et al. An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 10., 1991. **Proceedings...** New York: ACM, 1991. p. 136-145.

[INT 96a] INTEL CORPORATION. *Pentium Pro Family Developer's Manual*: Specifications. [S.I.] Intel Corporation, 1996.

[INT 96b] INTEL CORPORATION. *Pentium Pro Family Developer's Manual*: Programmer's Reference Manual. [S.I.] Intel Corporation, 1996.

[JOH 91] JOHNSON, Mike. **Superscalar Microprocessor Design**. Englewood Cliffs: Prentince Hall, 1991.

[KEL 75] KELLER, R. M. Look-Ahead Processors. **ACM Computing Surveys**,New York, v.4, n.4, p.177-195, December 1975.

[YEH 91] YEH, Tse-Yu; PATT, Yale N. Two-Level Adaptive Training Branch Prediction. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 24., 1991. **Proceedings...** New York: ACM, 1991. p. 51-61.

[YEH 93] YEH, Tse-Yu; PATT, Yale N. A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 20., 1993. **Proceedings...** New York: ACM, 1993. p. 257-266.

[LAN 80] LANDSKOV, David et al. Local Microcode Compactation Techniques. **Computing Surveys**, New York, v.12, n.3, p.261-294, September 1980.

[LEE 84] LEE, J. K. F.; SMITH, A. J. Branch Prediction Strategies and Branch Target Buffer Design. **Computer**, Los Alamitos, v.17, n.1, p.6-22, Jan. 1984.

[LEN92] LENIR, Philip et al. Exploiting Instruction-Level Parallelism: The Multithreaded Approach. **IEEE Micro**, New York, 1992.

[LIL 88] LILJA, David J. Reducing the Branch Penalty in Pipelined Processors. **Computer**, Los Alamitos, v.21, n.7, p.47-55, July 1988.

[LIL 94] LILJA, David J. Exploiting the Parallelism Available in Loops. **Computer**, Los Alamitos, v.27, n.2, p. 13-26, Feb. 1994.

[OEH 90] OEHLER, R. R.; GROVES, R. D. IBM RISC System/6000 Processor Architecture. **IBM Journal of Research and Development**, New York, v.34, n.1, p.23-36, January 1990.

[KNI92] KNIESER, Michael J.; PAPACHRISTOU, Christos A. Y-Pipe: A Conditional Branching Scheme Without Pipeline Delays. In: INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 25., 1992. **Proceedings...** New York: ACM, 1992. p. 125-128.

[PAN 92] PAN, Shien-Tai et al. Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation. **ACM SIGPLAN Notices**, New York, v.27, n.9, p. 76-84, September 1992.

[SAN 94] SANTOS, Ana Dolejsi. **Efeito da Execução Condicional em Arquiteturas Paralelas**, Rio de Janeiro: COPPE/UFRJ, 1994. Tese de Doutorado.

[SAN 96] SANTOS, Rafael R. **Arquiteturas Superescalares: Um Estudo sobre Dependências de Controle.** Porto Alegre: CPGCC da UFRGS, 1996. Trabalho Individual.

[SAN 97] SANTOS, Rafael R.; NAVAUX, Philippe O. A. Mecanismo de Busca Especulativa de Múltiplos Fluxos de Instruções. In: SBAC-PAD, 9., 1997, Campos do Jordão. **Anais...** Campos do Jordão: SBC/USP, 1997. p.65-78.

[SEZ 96] SEZNEC, André et al. Multiple-Block Ahead Branch Predictors. In: CONFERENCE ON ARCHITECTURAL SUPPORT FOR PORGRAMMING LANGUAGES AND OPERATING SYSTEMS, 7., 1996. **Proceedings...** Boston: ACM, 1996.

[SOH 95] SOHI, G. S.; BREACH, S. E.; VIJAYKUMAR, T. N. Multiscalar Processors. **Computer Architecture News**, New York, v.23, n.2, p. 414-425. Trabalho apresentado no ISCA'95, Santa Margherita Ligure, Italy.

[SON94] SONG, S. Peter; DENMAN, Marvin; CHANG, Joe. The PowerPC 604 RISC Microprocessor. **IEEE Micro**, New York, p. 8-17, October 1994.

[SMI 85] SMITH, J. E.; PLESZKUN, A. R. Implementation of Precise Interrupts in Pipelined Processors. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 12., 1985. **Proceedings...** New York: IEEE, 1985. p.36-44.

[SPE 95] Stadard Performance Evaluation Corporation. *SPEC CPU95 Version 1.10:* Unix and Windows NT. Standard Performance Evaluation Corporation, 1997.

[SUN 87] SUN MICROSYSTEMS. **The SPARC Architecture Manual Version 7**. Mountain View, CA: Sun Microsystems, 1987.

[TAL 95] TALCOTT, Adam R. **Reducing the Impact of the Branch Problem in Superpipelined and Superscalar Processors**. Santa Barbara: University of California, 1995. Ph. D. Thesis.

[TOM 67] TOMASULO, R. M. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. **IBM Journal of Research and Development**, Armonk, v.11, n.1, p. 25-33, Jan. 1967.

[THO 64] THORNTON, James E. Parallel Operation in the Control Data 6600. In: AFIPS FALL JOINT COMPUTER CONFERENCE, 1964. **Proceedings...** [S.l.:s.n.], 1964. v.26, p.30-40.

[TYS92] TYSON, Gary et al. MISC: A Multiple Instruction Stream Computer. **IEEE Micro**, New York, 1992.

[TYS 94] TYSON, G. S. The Effects of Predicated Execution on Branch Prediction. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 27., 1994. **Proceedings...** New York: ACM, 1995. p. 196-206.

[WAL 93] WALLACE, Steven D. **Performance Analisys of a Superscalar Architecture**. Irvine: University of California, 1993. Ph. D. Thesis

[WAL 98] WALLACE, Steven D.; BAGHERZADEH, Nader. Instruction Fetching Mechanisms for Superscalar Microprocessors. **IEEE Transactions on Parallel and Distributed Systems**, New York, v.9, n.6, 1998.

[WOL 91] WOLFE, A.; SHEN, J. P. A Variable Instruction Stream Extension to the VLIW Architecture. **ACM SIGPLAN Notices**, New York, v.26, n.4, p.2-14, April 1991.

Informática  UFRGS

CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

"Um Mecanismo de Busca Especulativa de Múltiplos Fluxos de Instruções "

por

Rafael Ramos dos Santos

Dissertação apresentada aos Senhores:

Eliseu Monteiro Chaves Filho

Prof. Dr. Eliseu Monteiro Chaves Filho (COPPE/UFRJ)

Prof. Dr. Sérgio Bampi

Prof. Dr. João César Netto

Vista e permitida a impressão.
Porto Alegre, 21/06/99.

Prof. Dr. Philippe Olivier Alexandre Navaux,
Orientador.