Dependências de Controle em Arquiteturas Super Escalares e Mecanismos para o seu Tratamento

Rafael R. dos Santos[1]
Philippe O. A. Navaux[2]

Universidade Federal do Rio Grande do Sul - UFRGS Curso de Pós-Graduação em Ciência da Computação - CPGCC
Av. Bento Goncalves, 9500 Cx.P.15064 - CEP:91501-970
Porto Alegre - RS - Brasil

Resumo

Este trabalho analisa as dependências de controle em arquiteturas Super Escalares. Neste contexto, este trabalho avalia as limitações fundamentais do desempenho de processadores *pipeline* e superescalares dando ênfase às dependências de controle. Por este motivo, as principais técnicas empregadas na redução do custo de desvios são discutidas e os resultados esperados, assim como os problemas envolvidos em cada uma das técnicas, são analisados. Por fim, apresentamos sugestoes e tendencias na arquitetura de processadores paralelos a nível de instrução.

Abstract

This work analizes the superscalar processors control dependency. In this context, this work analizes the fundamental limitations in the performance of pipelined and superscalar processors foccusing the control dependency. Thus, the basic schemes used to reduce the branch penalties are discussed and the expected results and problems are analized. Finally, suggestions and trends in the instruction-level parallel processors architecture are discussed.

1 Introdução

O paralelismo a nível de instrução (*Instruction-Level Parallelism - ILP*) é o mecanismo adotado por vários processadores da atualidade para alcançar níveis mais altos de desempenho. O uso de técnicas de *pipeline* é um exemplo típico desta tendência.

Com os avanços na tecnologia de integração e com a necessidade de maior desempenho, requerido por aplicações cada vez mais complexas, processadores com várias unidades funcionais têm sido construídos. Recentemente, algumas arquiteturas *pipeline* capazes de despachar múltiplas instruções foram projetadas e lançadas no mercado [CHA 94].

Em arquiteturas *pipeline* escalares, a execução de uma instrução por ciclo é um limite ou patamar, portanto, técnicas alternativas ou adicionais precisam ser exploradas para melhorar o desempenho [WAL 93].

[1]B.Sc.; Mestrando – E-mail: rrsantos@inf.ufrgs.br
[2]Prof. Dr.; Inst. Informática – E-mail: navaux@inf.ufrgs.br

Nos últimos anos, houve um crescente interesse no projeto de processadores baseados na noção de paralelismo a nível de instrução (*ILP*), ou paralelismo de baixo nível. Existem diferentes formas de exploração do *ILP*. Uma alternativa usa escalonamento em tempo de execução para avaliar as dependências entre as instruções e para executá-las concorrentemente. Um processador baseado nesta técnica é chamado superescalar. Outra forma, comumentemente conhecida como *Very Large Instruction Word - VLIW* é estritamente baseada na análise em tempo de compilação para a extração do paralelismo.

Arquiteturas superescalares e arquiteturas *VLIW* aumentam o desempenho do processador através da redução do número de ciclos por instrução (*CPI*). Estas arquiteturas exploram o paralelismo a nível de instrução através do despacho de múltiplas instruções por ciclo. Processadores *VLIW* requerem sofisticados compiladores para extrair o paralelismo a nível de instrução, antes da execução, o que resulta em expansão de código. Arquiteturas superescalares, utilizam escalonamento dinâmico para extrair o paralelismo em tempo de execução, em contraste ao escalonamento estático. Esta vantagem, entretanto, é acompanhada de um aumento significante da complexidade do hardware subjacente.

Um *pipeline* de instrução de n estágios executa n instruções simultaneamente. Desta forma, um processador *pipeline* teoricamente possibilita um aumento de performance diretamente proporcional ao número de estágios se comparado à um processador convencional. No entanto, este desempenho é raramente alcançado devido a presença de problemas resultantes das dependências entre instruções no *pipeline*, e que são expostas por sua execução paralela. Três tipos de problemas podem limitar o desempenho em um *pipeline* de instrução: conflito de recursos, dependência de dados e dependência de controle

Conflito de recursos resultam de processadores que possuem um número insuficiente de recursos disponíveis, enquanto dependências de dados resultam de dependências entre resultados e operandos de instruções separadas. Dependências de controle resultam de transferências no fluxo de controle causadas por instruções de desvio no fluxo de instruções.

Conflito de recursos podem ser removidos através do acréscimo do número de unidades funcionais ou por particionar unidades funcionais existentes em diversos estágios (*pipelining*). Dependências de dados são prontamente eliminadas através de esquemas como renomeação de registradores ou através do escalonamento destas instruções. O escalonamento consiste em reordenar as instruções, de forma a antecipar a execução de instruções independentes enquanto instruções com dependências não podem ser executadas.

As técnicas desenvolvidas para tratar as dependências de controle procuram reduzir o custo de execução das instruções de desvio. O custo de uma instrução de desvio é a soma de duas componentes: a primeira é o tempo necessário para que o resultado do desvio seja determinado; a segunda, é o tempo para que a instrução alvo do desvio seja acessada.

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 das dependências entre instruções ainda não são capazes de anular totalmente o custo destas dependências. Isto é particularmente crítico nos mecanismos dedicados ao tratamento das dependências de controle.

Neste trabalho, estudamos os tipos de instruções de desvio e o efeito que causam no desempenho de processadores superescalares. Técnicas básicas empregadas na redução do custo de desvios são discutidas e os resultados esperados são analisados.

2 Limitações Fundamentais em Arquiteturas Super Escalares

Processadores superescalares são conceitualmente simples, mas existe muito mais do que alargar o *pipeline* de um processador pata obter mais performance. O simples fato de se alargar o *pipeline* permite que mais de uma instrução seja executada por ciclo, mas não existe garantia de que qualquer seqüência de instruções possa tirar proveito desta capacidade [JOH 91]. As instruções não são independentes umas das outras, porém estão interrelacionadas. Estas interrelações impossibilitam o algumas instruções de ocuparem os mesmos estágio no *pipeline*.

Uma seqüência de instruções pode possuir três características que limitam o desempenho de um processador superescalar:

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

2.1 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. 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*.

2.1.1 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 1, 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.

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.

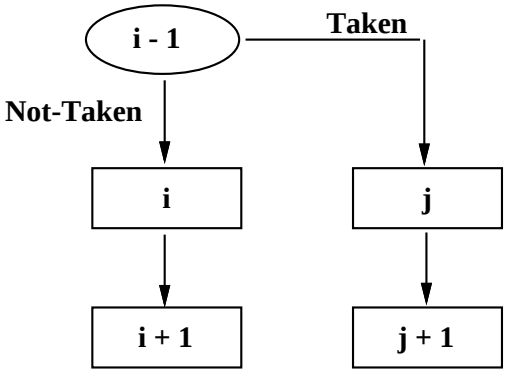

Figura 1 Exemplo de Fluxo de Instruções com Desvio Condicional

2.1.2 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 90] 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, como cada evento é diferente, então podem usar diferentes instruções, e podem ter diferentes comportamentos. Para três *benchmarks* (*TeX, Spice, GCC*), [HEN 90] obteve as seguintes freqüências para diferentes tipos de instruções de fluxo de controle.

A Figura 2 mostra a freqüência para os três tipos de desvios. Cada coluna mostra um dos tipos de desvios para cada um dos programas *benchmark*.

Através dos dados mostrados no gráfico da Figura 2, considerando que as dependências de controle afetam drasticamente o desempenho de processadores *pipeline*, e que o desvios condicionais são os que melhor representam este efeito negativo, este trabalho concetra-se no estudo de dependências de controle e técnicas para redução do custo de desvios.

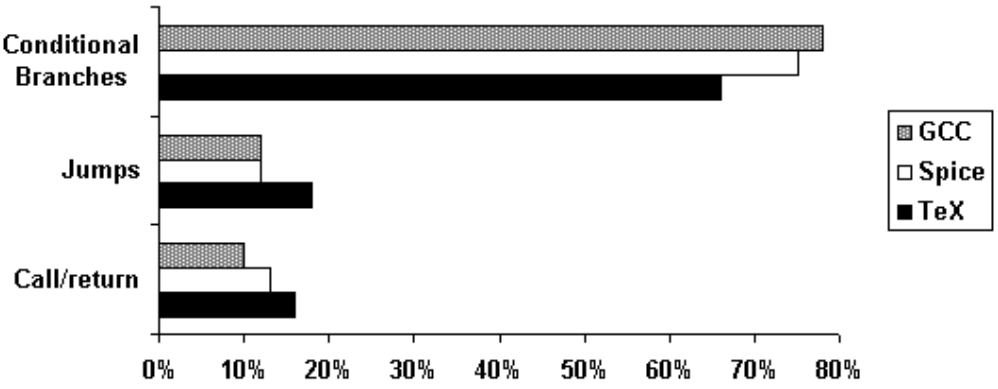

Figura 2 Gráfico de Freqüência da Ocorrência de Desvios

2.1.3 Penalidade de Desvios

Para analizar o efeito das instruções de desvio na performance de processadores *pipeline* considere o *pipeline* da Figura 3.

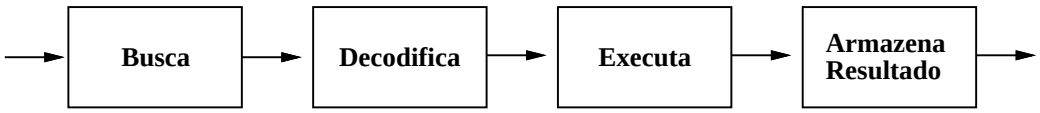

Figura 3 Exemplo de Processador *Pipeline*

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 4 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.

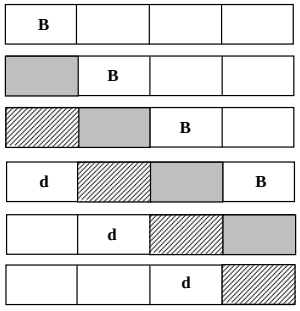

Figura 4 Efeito das Dependências de Controle em *Pipelines* Escalares

A Figura 5 corresponde a um *pipeline* superescalar onde duas instruções podem ser processadas em cada estágio. Em ambas as Figuras (4 e 5) 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 não foi tomado (*not-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 seja 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.

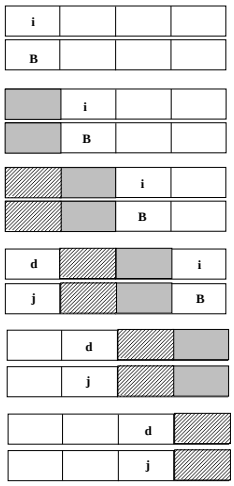

Figura 5 Efeito das Dependências de Controle em *Pipelines* Superescalares

O retardo de desvio é definido como 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 o retardo de desvio [LIL 88]. No exemplo da Figuras 4 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.

2.1.4 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. Alguns processadores também inclueme 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 do desvio ser encontrado.

Enquanto existem alguns benefícios no emprego de instruções separadas de comparação e desvio, elas podem trazer algum custo. O fato de que a separação da computação, do resultado de um desvio da instrução de desvio propriamente dita, requer 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. 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 a menos que sejam manipulados por alguma das técnicas discutidas no Capítulo 3.

2.1.5 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 registradore 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 propabilidade 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. Algumas arquiteturas definem registradores especiais para armazenar o endereço alvo de desvios como é o caso das arquiteturas POWER e PowerPC da IBM [TAL 95].

3 Técnicas para Redução do Custo de Desvios

Dependências de controle são o maior obstáculo para ativar o aumento no *throughput* teórico de instruções possível com um *pipeline* de n estágios [TAL 95]. Este capítulo discute algumas das técnicas utilizadas para reduzir o número de desvios que causam alguma penalidade no desempenho de processadores *pipeline*. Primeiro, soluções mais básicas são apresentadas, enquanto são fáceis de implementar, não apresentam tão boa eficiência. Estas são seguidas por mais avançados esquemas, mais complicados, mas que permitem porém uma redução drástica no número de desvios que causam esta penalidade.

3.1 *Pipeline Stall After Branch*

O mais básico esquema 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, a 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.

3.2 *Delayed Branch*

A Figura 6 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*.

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

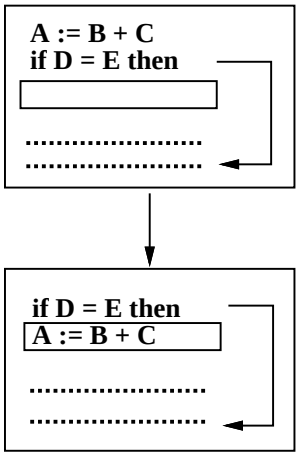

Figura 6 Reorganização de Instruções Segundo a Técnica *Delayed Branch*

o compilador deve levar em consideração as relações de dependência entre as instruções, de maneira que a equivalência semântica do prograna seja preservada. Os retângulos da Figura 6 representam um trecho de programa antes e depois da reorganização. Na parte superior da figura, uma instrução do tipo *NOP* é precedida pelo comando de desvio condicional. Após a reorganização, retângulo inferior da Figura 6, 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 6) 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 94].

3.3 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 com a 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.

3.4 *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 de carga na 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 manipulam 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 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].

3.5 *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 setadas 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 computa 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.

3.6 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". 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.

3.6.1 Previsão Dinâmica De Desvios

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:

(a) O endereço do desvio e o endereço da instrução sucessora; ou,

(b) 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, podemos usar 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 (paginação, segmentação etc.), podem ser empregados no gerenciamento das entradas da tabela.

3.6.2 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.

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 aquele 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 deve ser 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 7 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, previsão baseada na última execução do desvio.

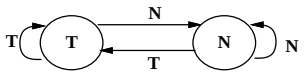

Figura 7 Autômato para Previsão com 1 bit de História

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*).

3.6.3 Branch Target Buffer

A *Branch Target Buffer*, Figura 8, é uma memória cache especial associada 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.

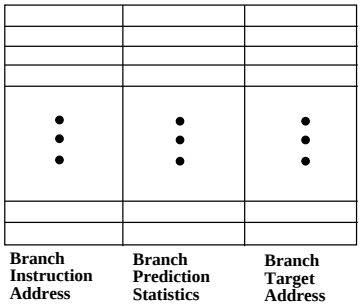

Figura 8 Organização da *Branch Target Buffer*

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 àquele desvio.

O número de bits de história (previsão) é um fator de extrema relevância na escolha do algoritmo de previsão. Na figura 7 mostramos 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 midificada apenas se as duas últimas previões foram incorretas.

A figura 9 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%.

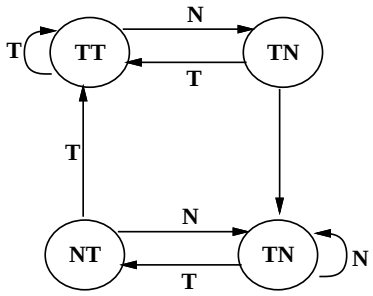

Figura 9 Autômato de Previsão com 2 Bits de História

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 10 mostra o diagrama para a previsão com BTB.

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:

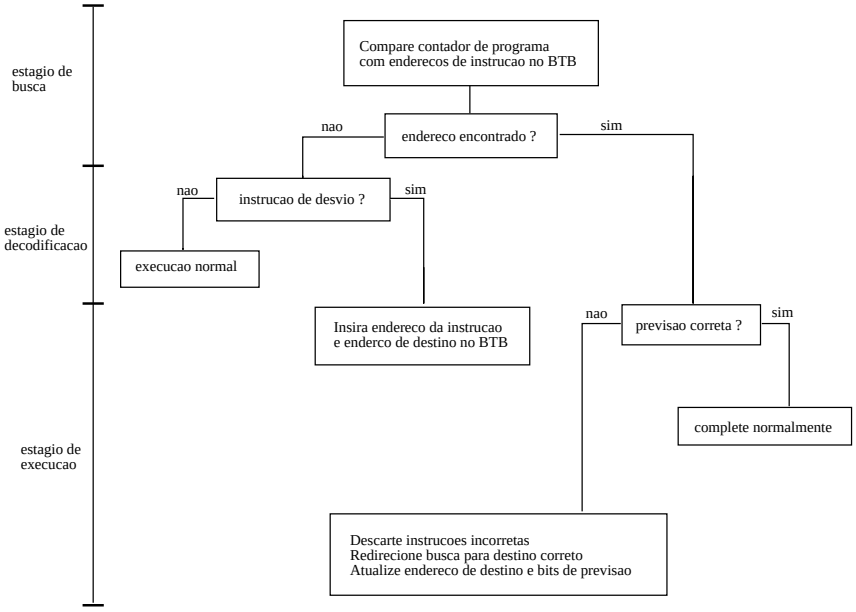

Figura 10 Operação de uma *Branch Target Buffer*

- 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:

$$TA = HitRatio * FA \tag{1}$$

A fórmula 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.

4 Conclusão

Processadores como o IBM System RS/6000, Motorola 88110, PowerPC 601, 604 e 620, Pentium e Pentium Pro, DEC Alpha 21064 e 21164, SuperSPARC e SuperSPARC 2, são exemplos de processadores superescalares comercialmente utilizados que têm a capacidade de despachar mais de uma instrução por ciclo de máquina para as

diversas unidades funcionais existentes internamente. Característica esta que permite a exploração do paralelismo a nível de instrução ou, a execução de múltiplas instruções por ciclo.

Ao longo deste trabalho procuramos dar ênfase às dependências de controle que limitam o paralelismo em processadores *pipeline* e superescalares. 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.

O impacto causado pelas instruções de desvio no desempenho de processadores superescalares pode ser amenizado através do emprego de técnicas de previsão de desvios associadas à execução especulativa de instruções. No entanto, não é possível afirmar que tais técnicas possam oferecer garantias de que os efeitos negativos, causados pela existência de instruções de desvio, serão eliminados. A precisão do mecanismo de previsão de desvios é essencial para que a execução especulativa de instruções possa beneficiar o grau de paralelismo alcançado ao longo da execução de um programa.

A previsão de desvios nada mais é do que uma tentativa, baseada nas ocorrências anteriores, de acertar o fluxo de controle por onde o programa passará. O fluxo previsto pode ser executado especulativamente de maneira que a penalidade causada pela instrução de desvio seja anulada. No entanto, se a previsão estiver incorreta, a execução especulativa não proporciona o ganho esperado já que o fluxo que deveria ser executado, preservando a seqüência original do programa, deve ser buscado e executado. Logo, o desempenho fica limitado àquela instrução de desvio.

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.

[SOH 95] propõe um modelo de execução denominado "multi-escalar". Neste modelo é gerado o gráfico de fluxo de controle (CFG) do código do programa que posteriormente é dividido em tarefas. Cada tarefa pode englobar um ou mais blocos básicos do programa original. Assim, uma tarefa contém um conjunto de instruções que podem escalonadas e executadas pelas unidades de processamento. Em resumo, cada unidade de processamento recebe uma das tarefas do CFG e escalona instruções pertencentes àquela tarefa. O objetivo deste modelo é estabelecer uma janela de instruções ampla e precisa de onde instruções independentes podem extraidas e despachadas para execução paralela. Deve-se observar que o fato de existirem várias tarefas em execução simultaneamente aumenta a possibilidade de manter-se ocupados todos os recursos do processador. Porém, são necessários mecanismos de

comunicação e sincronismo entre a tarefas já que pode haver dependência entre estas.

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. Com o aumento da densidade de integração e o barateamento do hardware esta alternativa compõe uma forte tendência. O conflito de recursos e as dependências de dados, ora desconsiderados, podem se tornar relevantes já que são necessários mais recursos para a execução dos possíveis fluxos de um desvio assim como podem existir dependências entre os dados pertencentes aos diferentes fluxos.

Referências

[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)

[HEN 90] HENNESSY, John L.; PATTERSON, David A *Computer Architecture: A Quantitative Approach.* Palo Alto: Morgan Kaufmann, 1990.

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

[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.

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

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

[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.

[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).