Trace Cache: Fundamentos, Alternativas e Desempenho

Gabriela Batista Leão – RA
087348
Institute of Computing – State
University of Campinas
Rua Roxo Moreira, 1450, Cid.
Universitária, Barão Geraldo
(+55) 64 3288 0207
gabileao@gmail.com

RESUMO

Este trabalho descreve os fundamentos da trace cache, um mecanismo que visa aumentar a largura de banda da unidade de busca para que multiplos blocos básicos não-contínuos possam ser recuperados em um mesmo ciclo. Duas alternativas à trace cache são também descritas e analisadas. Finalmente, o desempenho dos três métodos são avaliados e comparados. Como resultado, a trace cache apresenta desempenho muito superior às outras abordagens, tanto para latências de 1 ciclo quanto para latências maiores.

Categorias e Descritores de Assunto

B.3.2 [**Design Styles**]: Cache Memories – *trace cache, mecanismo de busca de banda larga, previsão de desvios, otimização, desempenho.*

Termos Gerais

Measurement, Performance, Design.

Palavras-Chave

Trace cache, mecanismo de busca de banda larga, múltiplas previsões de desvios, otimização, desempenho, cache de instruções intercalada.

1. INTRODUÇÃO

As unidades de busca de instruções convencionais, mesmo aquelas capazes de atenderem buscas por múltiplas instruções em um mesmo ciclo, são limitadas a buscar um único bloco básico, fazendo com que tal mecanismo seja um gargalo que restringe o desempenho geral de toda a arquitetura, mesmo que seus demais componentes sejam otimizados ao máximo. Este problema ocorre porque não é possível buscar a instrução de um desvio e seu endereço alvo em um mesmo ciclo. Em outras palavras, é inútil escalar a quantidade de instruções que podem ser expedidas paralelamente pelo processador se a taxa de busca de instruções por ciclo é baixa.

Consequentemente, meios de aumentar a largura de banda do mecanismo de busca foram propostos, como a extensão ao Branch Target Buffer (BTB) [1], denominado Branch Address Cache (BAC) [2], que permite que múltiplos previsores de desvios condicionais calculem endereços alvo no mesmo ciclo, e o Collapsing Buffer [3], que explora o uso de uma cache de instruções intercalada, ou com múltiplas portas de leitura, para buscar instruções de desvio e seus endereços alvo no mesmo ciclo, a partir de bancos diferentes da cache de instruções. No entanto, essas alternativas colocam muita complexidade no caminho crítico da unidade de busca, criando um limitador de desempenho.

Este trabalho descreve uma técnica chamada Trace Cache (TC) [4] que aumenta consideravelmente a lagura de banda do mecanismo de busca, colocando sua complexidade de implementação fora do caminho crítico da unidade de busca, por armazenar em cada linha de uma cache especializada segmentos (*traces*) não contínuos, cada qual sendo limitado a um bloco básico.

Em resumo, a trace cache armazena, em tempo de execução, segmentos dinâmicos não-contínuos em um espaço contínuo, o que proporciona maior exploração do princípio da localidade desde que uma única busca pode recuperar vários blocos básicos de uma só vez. Deve ser notado que o sucesso da trace cache depende essencialmente do princípio da temporalidade, pois os segmentos são armazenados somente depois de serem executados pela primeira vez, supondo-se que serão executados novamente em breve.

Porque a trace cache armazena segmentos dinâmicos e não estáticos, o segmento já estará pronto para ser usado pelo decodificador quando ele for buscado uma outra vez.

Este artigo está descrito como se segue. Seção 1 apresentou a motivação e o contexto por trás do projeto da trace cache; Seção 2 provê uma visão geral sobre o funcionamento da unidade de busca convencional, cuja implementação está presente em praticamente todas as arquiteturas correntes, e sua limitação, focando sempre a busca de instruções de desvios condicionais; Seção 3 oferece uma visão de alto nível dos fundamentos, funcionamento e espaço de projeto da trace cache; Seção 4 descreve em detalhes os dois mecanismos de busca, alternativos à trace cache, já mencionados na Introdução, juntamente com suas limitações; na Seção 5 são feitas análises de desempenho comparando as três abordagens referidas neste trabalho e, finalmente, algumas conclusões são feitas na Seção 6.

2. UNIDADE DE BUSCA CONVENCIONAL

A Figura l mostra os componentes de uma unidade de busca de instruções convencional e as interações entre eles. É importante ressaltar que o papel da trace cache não é substituir a cache de instruções, muito menos a unidade de busca de intruções, mas apenas melhorar o desempenho geral desse mecanismo. Portanto, nesta seção uma implementação simples da unidade de busca é

apresentada, enquanto na seção seguinte a trace cache é incorporada a ela.

O mecanismo da Figura 1 é capaz de buscar múltiplas instruções no mesmo ciclo, até que o primeiro desvio condicional previsto como tomado seja encontrado. Em outras palavras, o mecanismo permite apenas a busca de instruções contínuas, i.e. um desvio e seu alvo não podem ser buscados no mesmo ciclo.

A vazão de instruções que pode ser buscada paralelamente também depende da quantidade de entradas do previsor de desvio disponíveis (Multiple Branch Predictor) e do limite máximo de instruções que pode ser abrigado por cada linha da cache de instruções, os quais são 3 e 16, respectivamente, como mostrado na Figura 1. Esta é a configuração arquitetural considerada neste trabalho.

Isso significa que em um mesmo ciclo podem ser buscadas no máximo 16 instruções e feitas 3 previsões de desvio para um único bloco básico, mas se um desvio é previsto como tomado antes que estes limites sejam alcançados, a busca recuperará uma quantidade de instruções inferior àquela suportada pelo hardware, limitando o desempenho geral da arquitetura.

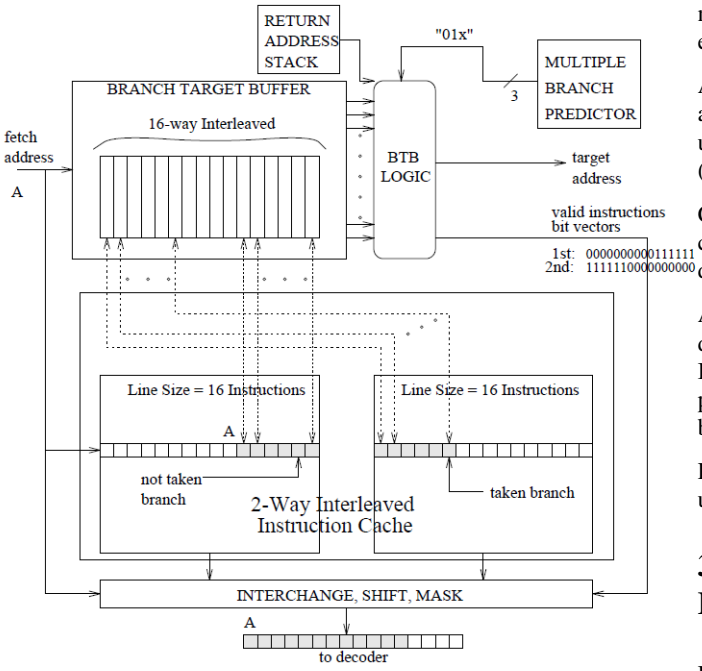

Figura 1. Unidade de busca convencional.

Para permitir acessos simultâneos, tanto a cache de instruções quanto o Branch Target Buffer (BTB) devem ser intercalados ou possuir múltiplas portas de leitura. No caso da cache, isso permite buscar instruções em duas linhas consecutivas, i.e. em bancos diferentes (vide Figura 1), ao mesmo tempo, até que um desvio previsto como tomado apareça.

Na verdade, a cache intercalada é necessária para garantir que uma linha completa seja buscada, i.e. 16 instruções de uma só vez, ou que o máximo de instruções seja recuperado até o surgimento do primeiro desvio previsto como tomado. Este esquema por si só garante maior largura de banda do mecanismo de busca, se comparado a alternativas com cache de instruções não-intercalada.

Como cada busca na cache pode recuperar um subconjuto das instruções em um banco da cache e um segundo subconjunto em outro, essas intruções precisam ser alinhadas. Isso requer complexidade mínima sendo implementada pelo componente INTERCHANGE, SHIFT, MASK mostrado na Figura 1.

O alinhamento de instruções requer a troca da ordem das instruções das duas linhas da cache, um *shifter* que desloca para a esquerda as instruções fazendo com que elas caibam dentro de uma finalização de 16 instruções de largura, e uma lógica de mascaramento de instruções não usadas. Após esse processamento, o conjunto de instruções resultatnte é entregue ao decodificador.

Quando um endereço é buscado, ele é indexado, ao mesmo tempo, tanto pela cache de instruções quanto por todos os bancos do BTB. Isso permite detectar quais das instruções constituem um desvio e prover, simulaneamente, seus endereços alvo, os quais serão utilizados no próximo ciclo da busca. Desta forma, o BTB precisa ser intercalado de forma a permitir que todas as instruções sendo buscadas na cache possam ser verificadas se são de desvio ou não, em paralelo. Portanto, o nível de intercalação (16-way, neste caso) deve ser igual a quantidade de instruções armazenadas em uma linha da cache.

A lógica do BTB pode detectar também outros tipos de desvios, além dos condicionais, como jumps e returns. Quando detectado um return, o BTB retira do topo da pilha de endereços de retorno (Return Address Stack - RAS) o endereço do alvo do desvio.

Como pode ser visto na Figura 1, o previsor de desvio é um componente independente do BTB. Isso permite a impleemntação de vários tipos de previsores, sem causar prejuízos ao BTB.

A lógica do BTB (BTB Logic) recebe como entradas a previsão do desvio gerada pelo previsor de desvio múltiplo (Multiple Branch Predictor) e a informação de hit do BTB, e as combina para dar origem ao endereço alvo, i.e. o próximo endereço da busca.

Próxima seção descreve a trace cache e como ela interage com a unidade de busca.

3. FUNDAMENTOS, FUNCIONAMENTO E ESPAÇO DE PROJETO DA TRACE CACHE

Deve ser notado que a unidade de busca convencional somente é capaz de buscar instruções contínuas. Encontrado um desvio previsto como tomado, não é mais possível recuperar as próximas instruções no mesmo ciclo, i.e. a cada cilco somente um único bloco básico pode ser buscado.

O problema mencionado é resolvido por uma trace cache, uma cache de instruções especializada que armazena sequências dinâmicas não-contínuas de instruções de forma contínua, integrada na unidade de busca convencional. Veja Figura 2.

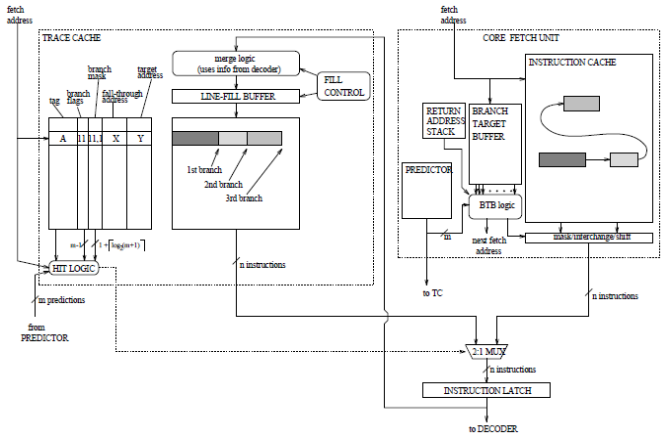

Figura 2. Unidade de busca com trace cache integrada.

A trace cache é composta por segmentos de instruções, informação de controle e lógica do buffer de abastecimento de linha.

Um segmento da trace cache consiste basicamente de *n* instruções, das quais *m* podem ser de desvios condicionais. O número máximo de instruções *n* é limitado pelo tamanho físico da linha da trace cache, assim como a quantidade de desvios condicionais depende da capacidade do previsor de desvios.

Se o previsor de desvio pode prever apenas 3 desvios por ciclo de clock, então cada segmento pode ter apenas 3 desvios condicionais. De forma análoga, a quantidade de instruções que pode ser executada por ciclo depende da quantidade de instruções que podem ser despachadas pela arquitetura em questão. Um processador superescalar capaz de despachar 16 instruções em um mesmo ciclo de clock poderá buscar, no máximo, 16 instruções na trace cache (considerando que o tamanho da linha da trace cache armazena a mesma quantidade de instruções que pode ser despachada pela arquitetura).

A informação de controle é bastante parecida com aquela encontrada no vetor de tag das caches de instruções convencionais, mas dotado de campos adicionais. Logo abaixo é mostrada a lista desses campos, juntamente com suas funcionalidades correspondentes:

- Bit de validade: este bit indica se o segmento é válido.
- Tag: identifica o endereço de início do segmento.
- Flags de desvio: existe um único bit para cada desvio dentro do segmento para indicar o caminho seguido depois do desvio (tomado/não tomado). Há apenas m-1 bits, pois nenhuma instrução segue o último desvio.
- Máscara do desvio: este estado é necessário para indicar (1) o número de desvios no segmento e (2) se o segmento termina em um desvio ou não. Isto é necessário para comparar o número correto de previsões de desvios contra o número de flags de desvio, quando checando por um hit do segmento. Essas informações são necessárias para que o previsor de desvio também possa saber quantas previsões foram utilizadas. Se a última instrução do segmento é um desvio, então seu flag de desvio correspondente não precisa ser checado, desde que não há instruções depois dele.
- Fall-through address: se o último desvio no segmento é previsto como não-tomado, este endereço é usado como a próximo endereço de busca.
- Target address: se o último desvio no segmento é previsto como tomado, este endereço é usado como o próximo endereço de busca.

Observe na Figura 2 que a trace cache é acessada em paralelo com a cache de instruções e BTB, usando o endereço de busca corrente. O previsor gera múltiplas previsões de desvios enquanto as caches são acessadas.

O endereço da busca é usado juntamente com as previsões de desvio geradas para determinar se a sequência do segmento sendo lido pela trace cache combina com a sequência de blocos básicos prevista.

Mais detalhadamente, um hit na trace cache requer que o endereço da busca combine com a tag e que as previsões de desvio combinem com os flags de desvio. Dado um hit, um segmento inteiro de instruções alimenta o decodificador. Sob um miss, a busca continua normalmente na cache de instruções, neste caso procurando por instruções contínuas.

A lógica do buffer de abastecimento opera sob o miss da trace cache. Ela intercepta as instruções providas pela cache de instruções, as combina e armazena para posteriormente entregá-las como um segmento para a trace cache. O buffer de abastecimento armazena apenas um segmento por vez, sendo este finalizado e entregue à uma linha da trace cache quando a quantidade máxima de instruções, *n*, ou a de desvios, *m,* é atingida. É responsabilidade do buffer de abastecimento é também criar e atualizar os flags de desvio, o target address e o fall-through address.

3.1 Espaço de Projeto da Trace Cache

Nesta subseção é descrita uma lista de opções de projeto da trace cache e seus impactos sobre o desempenho e a complexidade de implementação do mecanismo de busca.

- Associatividade: o projeto mais simples da trace cache possui mapeamento direto, mas como acontece com a cache de instruções convencional, esse mapeamento pode ser implementado de formas mais sofisticadas para reduzir misses de conflito. No entanto, um mapeamento mais sofisticado implica em maior tempo de acesso e substituição mais complexa de segmento.
- Caminhos múltiplos: um ponto fraco importante da trace cache é que a partir de um enedereço de início de segmento, apenas um segmento pode ser armazenado. Isto pode provocar duplicações de segmentos (ou de parte deles) em várias linhas da trace cache. Considere, por exemplo, que um segmento ABC é buscado e está armazenado na trace cache. Isso dará um hit e tudo segue seu curso normal. Mas, imagine se um segmento ABD é buscado e agora ele não está na cache. Isso dará um miss na trace cache e a unidade de abastecimento enviará esse segmento para ser gravado em uma nova linha da trace cache. Note que o segmento AB é replicado em 2 linhas da trace cache, neste caso.
- Partial Matches: uma alternativa para prover associatividade de caminho é permitir hits parciais.

Neste caso, ao invés da abordagem "tudo ou nada", a trace cache pode prover apenas a parte do segmento que combina com os blocos básicos sendo buscados. O custo disso é que campos similares aos target address e fall-through devem ser implementados nos blocos básicos intermediários do segmento.

- Questões de abastecimento de linha de cache: uma questão de projeto da unidade de abastecimento da trace cache é se a unidade de abastecimento despacha para a trace cache segmentos com alvos de desvio especultivos ou apenas alvos que já possuem o resultado exato da previsão. Um outro problema é que enquanto a unidade de abastecimento está sendo preenchida, servindo ao miss corrente, um outro miss pode ocorrer, desde que a trace cache ainda continua recebendo pedidos de busca de blocos básicos. O que fazer em relação ao novo miss? Há basicamente três alternativas para isso, sendo: (1) ignorar o novo miss; (2) demorar a servir o novo miss, enquanto o miss anterior ainda está sendo servido pela unidade de abasteciemnto; ou (3) implementar várias unidades de abastecimento para permitir que múltiplos misses sejam servidos ao mesmo tempo.

- Seleção de segmento: no projeto mais simples, há segmentos que são colocados na trace cache mas não são reutilizados. Tais segmentos podem tirar o lugar de segmentos úteis, causando misses desnecessários. Uma otimização que pode ser feita é implementar um pequeno buffer que armazena os segmentos mais recentes, com a finalidade de que apenas os segmentos que recebem um hit ou mais são enviados para a trace cache. Essa medida aumenta a taxa de hit da trace cache.

- Trace cache vítima: a trace cache pode usar uma cache vítima para armazenar permanentemente segmentos úteis que seriam substituídos por segmentos inúteis na trace cache.

Uma opção muito importante de projeto encontrada em processadores de propósito geral como Intel Pentium 4 e Intel Xeon é que a unidade de abastecimento além de montar o segmento, ela já o decodifica. Assim, o segmento provido pela trace cache é despachado diretamente para a unidade de execução, sem precisar passar pelo decodificador [5].

4. ALTERNATIVAS À TRACE CACHE E SUAS LIMITAÇÕES

Há muitas abordagens alternativas à trace cache para solucionar o problema de buscar mais do que um bloco básico por ciclo. No entanto, a grande maioria delas é baseada em dois mecanismos, Cache de Endereços de Desvio (Branch Addresses Cache - BAC) e Collapsing Buffer, que serão descritos em maiores detalhes nos parágrafos que se seguem. Deve ser ressaltado que ambas alternativas desempenham suas funcionalidades em tempo de compilação, i.e. operam sobre instruções estáticas, ao contrário da trace cache que processa segmentos dinâmicos – em tempo de execução.

A BAC é composta por quatro componentes principais: (1) uma cache de endereço de desvio; (2) um previsor de desvio múltiplo; (3) uma cache de instrução intercalada; e (4) uma rede de alinhamento e intercâmbio.

A BAC estende o buffer de alvo de desvios (BTB) para múltiplos previsores por armazenar uma árvore de endereços alvo e fall-through, como descrito na Figura 3 abaixo, onde o número de níveis da árvore depende da quantidade de desvios que podem ser previstos durante o mesmo ciclo.

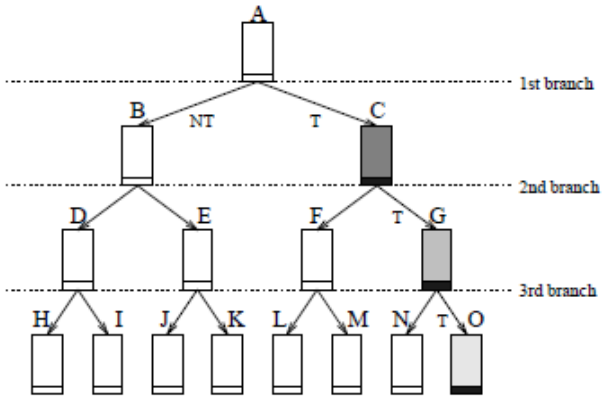

Figura 3. Árvore armazenada na BAC. Para 3 desvios há, no máximo 14 blocos básicos.

O projeto da BAC é feito em duas etapas. Na primeira, o endereço de busca indexa a linha da BAC correspondente e recupera até 14 blocos básicos, dos quais são selecionados 3 quando comparados contra as previsões de desvio realizadas.

No segundo estágio a cache de instruções, que tem múltiplas portas ou é intercalada, lê os três blocos básicos selecionados, ao mesmo tempo. Como esses blocos (ou as partes de cada um) podem estar espalhados aleatoriamente pelos bancos da cache, eles precisam ser alinhados e combinados seguindo a ordem do programa dinâmico. Isto é feito por uma rede de alinhamento e mascaramento. É importante notar que esses dois estágios são *pipelined*, i.e. enquanto os 3 blocos básicos estão sendo buscados na cache de instruções a BAC começa um novo ciclo usando o endereço do último bloco básico.

Se ocorre um miss na BAC referente a um endereço sendo buscado, uma entrada na BAC é alocada para o grafo do fluxo de controle começando nesse endereço. É neste momento que o endereço alvo do desvio e o endereço fall-through são calculados.

Esses endereços são atualizados como os desvios vão sendo descobertos durante a execução. Consequentemente, podem existir buracos em cada linha da BAC, quando nem todos os desvios foram ainda encontrados. O mascaramento do segundo estágio é responsável por remover instruções intermediárias não usadas dos blocos básicos, antes que eles sejam despachados para o decodificador. Figura 4 mostra a arquitetura lógica da BAC.

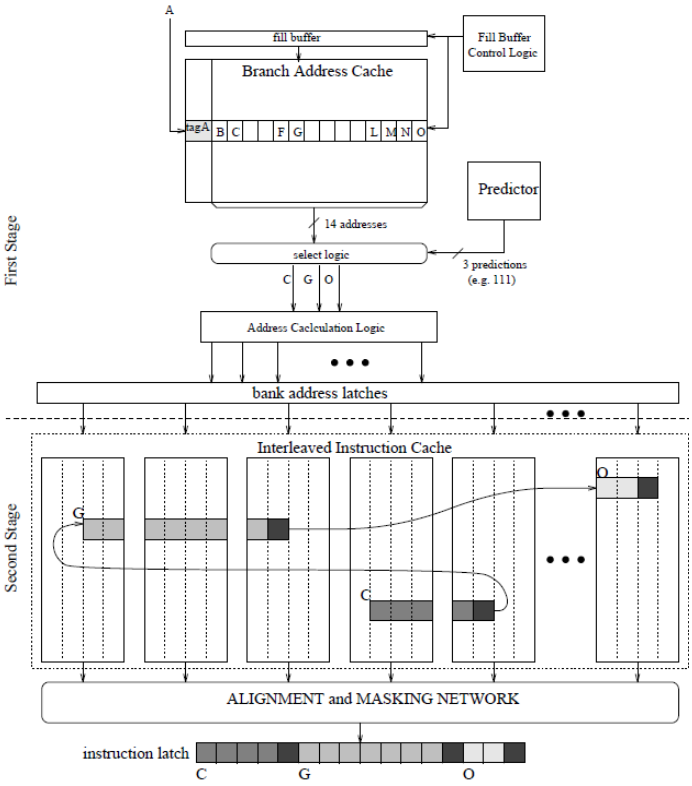

Figura 4. Visão geral da BAC.

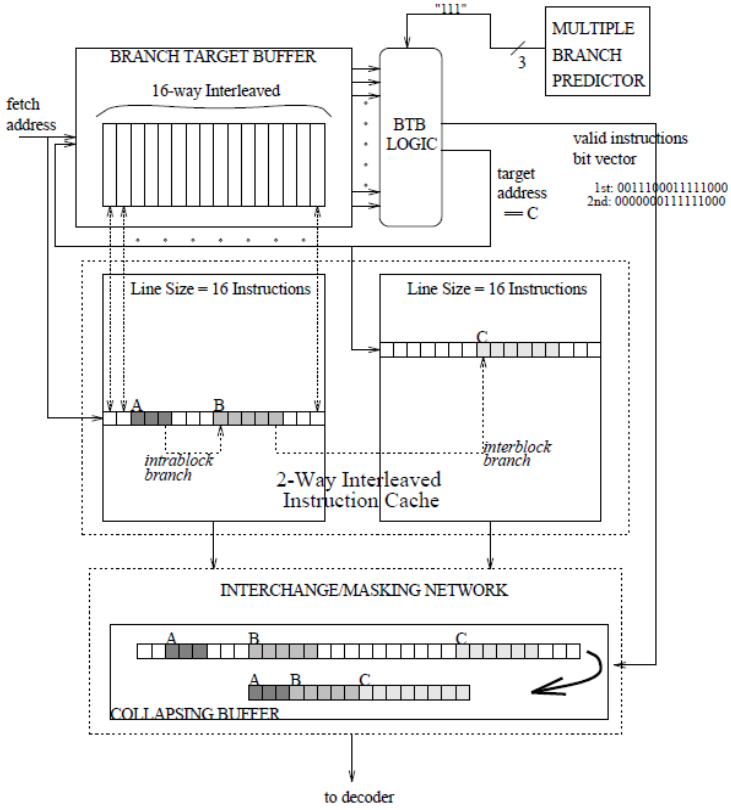

Figura 5. Visão geral do Collapsing Buffer.

O Collapsing Buffer (CB), mostrado na Figura 5, é composto por: (1) uma cache de instruções intercalada; (2) um buffer de alvo de desvio intercalado (BTB); (3) um previsor de desvios múltiplos; (4) lógica do BTB especial; e (5) uma rede de alinhamento e intercâmbio caracterizando um collapsing buffer.

O hadware desse mecanismo é muito similar ao mecanismo de busca convencional, já descrito anteriormente, mostrado na Figura 1, com duas diferenças importantes: a lógica do BTB é capaz de detectar desvios dentro da mesma linha da cache de instruções e uma única busca origina dois acessos ao BTB, o que permite buscar por desvios em duas linhas diferentes da cache.

Além disso, o componente de alinhamento e mascaramento de instruções conta com um collapsing buffer, cuja funcionalidade é combinar blocos básicos não-contínuos, usando informações enviadas pela lógica do BTB.

Na Figura 5, 3 blocos básicos são buscados, sendo que os blocos A e B estão no mesmo banco da cache de instruções, mas C está em outro.

O endereço de busca de A indexa o BTB intercalado e ele informa que a linha da cache correspondente possui 2 desvios condicionais, um na quinta instrução com endereço alvo B, e outro na instrução 13 com endereço alvo C. Além dessas informações o BTB também fornece para a cache as previsões para esses desvios e um vetor indicando que instruções são válidas para cada bloco e o endereço alvo C para o bloco básico B.

Em consequência, utilizando o endereço A e o endereço C, a cache intercalada pode recuperar blocos básicos não-contínuos, em um mesmo ciclo. Enquanto a busca é desempenhada na cache de instruções, uma segunda pesquisa no BTB, utilizando C como endereço, é realizada em paralelo. Ela determina quais as instruções válidas na segunda linha da cache e determina o endereço do próximo ciclo de busca, i.e. a instrução que seguirá C.

Finalmente, o collapse buffer recebe todas essas instruções válidas, as ordena, mascara e combina. O resultado final desse processamento é despachado para o decodificador.

Há limitações importantes com relação a essas duas alternativas, as quais formam a base para os outros métodos que aumentam a largura de banda do mecanismo de busca. A primeira e mais importante é que a complexidade das duas abordagens colocam muita complexidade no caminho crítico da unidade de busca e para suportá-las há necessidade de criar estágios extras de pipeline, o que aumenta a latência. Por exemplo, o collapsing buffer coloca 3 estágios extras no pipeline, como mostrado na Figura 6, relembrando que primeiro o BTB é pesquisado e informações de controle são reportadas (primeiro estágio extra); estas informações são então usadas pela cache de instruções para buscar os blocos básicos e, paralelamente, a segunda pesquisa ao BTB é realizada para recuperar informações de controle a respeito do bloco básico localizado na segunda linha da cache (segundo estágio extra); e as intruções são organizadas pelo collapsing buffer

Figura 6. Estágios extras impostos pela abordagem collapsing buffer.

Como esses esquemas não utilizam uma trace cache, mas uma cache de instruções comum, as instruções armazenadas nesta são estáticas. Assim, para representar o programa dinâmico, a cada ciclo instruções não contínuas têm que ser buscadas, ordenadas, mascaradas e combinadas. Em outras palavras, instruções estáticas não-contínuas devem ser buscadas da cache de instrução e montadas em uma sequência dinâmica. Isso aumenta o overhead do mecanismo de busca, de uma forma generalizada.

Para que o CB acesse a duas linhas em cada ciclo de busca, elas devem estar em bancos diferentes. Caso contrário, o CB só é capaz de buscar blocos básicos armazenados em uma só linha.

A não utilização de trace cache implica que para explorar paralelismo a cache de intruções tem que possuir múltiplas portas de leitura.

A trace cache não possui nenhuma dessas limitações, desde que a complexidade é tirada do caminho crítico da unidade de busca e colocada do lado da unidade de abastecimento de linha da trace cache. Além disso, a trace cache já armazena segmentos dinâmicos, prontos para serem enviados para o decodificador. Nenhuma transformação de bloco básico estático em dinâmico é necessária.

5. AVALIAÇÃO DE DESEMPENHO

Nos parágrafos seguintes desta seção são feitas análises de desempenho comparando as três abordagens descritas anteriormente. A métrica escolhida para tal comparação é a quantidade de instruções completadas por ciclo (IPC).

O mix das aplicações testadas sobre essas abordagens é constituído por 6 benchmarks de inteiros e 6 de ponto flutuante da suíte SPEC.

Como casos base da comparação serão usadas duas unidades de busca convencionais, como aquela descrita na Figura 1. A primeira unidade, denominada SEQ.1 nesta seção, em sua forma tradicional é capaz de buscar apenas 1 bloco básico por ciclo, enquanto a segunda unidade de busca é capaz de buscar 3 blocos básicos contínuos, i.e. múltiplos desvios não-tomados, por ciclo, referida como SEQ.3.

As configurações da unidade de busca, referente a cada um dos 3 mecanismos é descrita pela Tabela 1, enquanto Tabela 2 retrata seus desempenhos, assumindo que apenas 1 ciclo é gasto pela unidade de busca.

Tabela 1. Parâmetros de configuração da unidade de busca.

SIMULATION PARAMETER		INSTRUCTION SUPPLY MECHANISM		
		TC	CB	BAC
instruction fetch limit		16 instructions per cycle		
Multiple Branch Predictor	BHR	14 bits		
	PHT	2^{14} 2 bit counters (4 KB storage)		
	# pred/cycle	up to 3 predictions each cycle		
Instruction Cache	size	128 KB		
	associativity	direct mapped		
	line size	16 instructions	16 instructions	4 instructions
	interleave factor	2 way	2 way	8 way
	miss penalty	10 cycles		
Return Address Stack	depth	unlimited		
Branch Target Buffer	size	1024 entries	1024 entries	n/a
	associativity	direct mapped	direct mapped	
	interleave factor	16 way	16 way	
Trace Cache	size	64 entries	n/a	
	associativity	direct mapped		
	line size	16 instructions		
	# concurrent fills	1		
Branch Address Cache	size	n/a		1024 entries
	associativity			direct mapped
	# concurrent fills			1

Tabela 2. Comparação de desempenho entre as três alternativas para inteiros e ponto flutuantes, considerando latência de busca de 1 ciclo.

Benchmark	SEQ.1	SEQ.3	BAC	CB	TC
eqntott	3.05	3.30	3.96	4.16	4.24
espresso	3.17	4.10	4.42	4.61	5.20
xlisp	2.63	3.10	3.29	3.43	3.57
gcc	2.16	2.32	2.24	2.47	2.50
sc	3.17	3.64	4.10	4.33	4.43
compress	3.28	3.72	3.82	4.02	4.13
doduc	4.22	4.37	4.15	4.48	4.48
tomcatv	10.5	11.9	12.4	13.9	14.2
nasa7	8.41	8.56	8.52	8.63	10.5
mdljdp2	6.03	7.47	7.09	8.24	8.36
swm256	8.88	9.26	9.19	9.55	9.60
su2cor	4.66	4.77	4.72	4.90	5.02

Como pode ser notado, há melhoria de desempenho considerável de SEQ.1 para SEQ.3, refletido apenas pela possibilidade de buscar múltiplos blocos básicos contínuos, ao mesmo tempo, ao invés de apenas um.

Essa melhoria é superior a 7% para todos os benchmarks de inteiros e flutuantes, com metade dos inteiros mostrando ganho de desempenho igual ou superior a 15%. Dois benchmarks de ponto flutuante, nomeadamente nasa7 e su2cor, não tiveram melhorias significantes.

Quando a trace cache é adicionada a SEQ.3, o desempenho dos benchmarks de ponto flutuante sofrem uma grande melhoria, superior a 12%. Já os inteiros experimentam o mesmo ganho de desempenho que ocorreu quando da extensão de SEQ.1 para a SEQ.3. Desta forma, a adição da trace cache provê melhoria igual ou superior a 8% para todos os inteiros, se comparado ao caso base SEQ.3, como descrito no Gráfico 1.

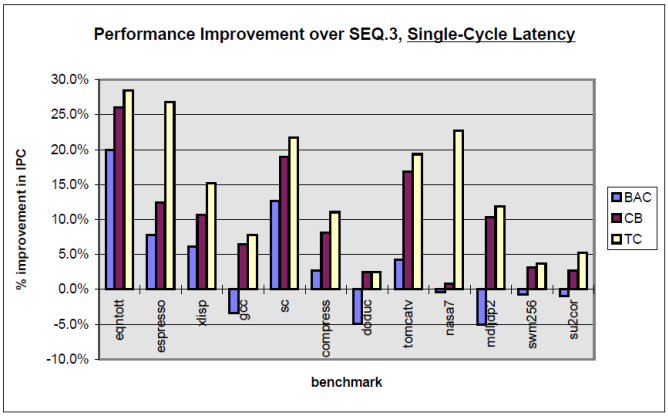

Gráfico 1. Ganhos de desempenho das três abordagens sobre SEQ.3, considerando latência de 1 ciclo para o mecanismo de busca.

O desempenho da BAC é o pior dentre as três abordagens. Em alguns casos ele é mesmo pior do que SEQ.3. Isso é a consequência dos conflitos dos bancos da cache de instruções. Tabela 3 mostra que se os conflitos são ignorados a BAC apresenta desempenho similar ao da TC.

Tabela 3. Com[arativo do desempenho de BAC com e sem conflitos em relação a TC.

Benchmark	BAC (conflicts)	BAC (no conflicts)	TC
eqntott	3.96	4.24	4.24
espresso	4.42	5.34	5.20
xlisp	3.29	3.46	3.57
gcc	2.24	2.36	2.50
sc	4.10	4.48	4.43
compress	3.82	3.93	4.13

No entanto, como BAC e CB adicionam estágios extras de pipeline antes e/ou depois da cache de instruções, por uma questão de realismo deve-se considerar latências de busca superiores a 1 ciclo. Gráfico 2 mostra a comparação dos 3 métodos de busca de instruções, com latências variáveis. Neste caso, L2 e L3 significam latência de 2 e 3 ciclos, respectivamente.

Com latências variáveis, para a maioria dos benchmarks, BAC com 2 ciclos de latência e CB com 3 ciclos desempenham pior do que SEQ.3, enquanto TC apresenta sempre desempenho superior às outras abordagens.

Outro resultado interessante é que os benchmarks de inteiros sofrem mais com o aumento da latência do que os de ponto flutuante.

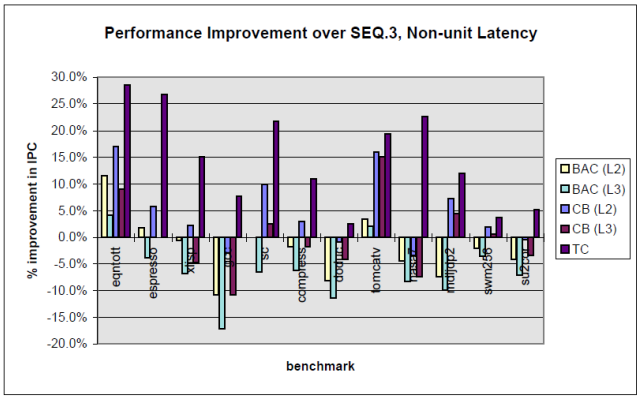

Gráfico 2. Ganhos de desempenho das três abordagens sobre SEQ.3, considerando latências de ciclos variáveis.

6. CONCLUSÃO

Trace cache mostra ser uma abordagem muito promissora, com um amplo espaço de projeto a ser explorado, possuindo desempenho superior às principais alternativas já propostas.

Ela é muito atraente por colocar a complexidade de implementação fora do caminho crítico da unidade de busca e já armazenar segmentos dinâmicos, prontos para serem servidos ao decodificador.

Apesar de a TC apresentar ganhos de desempenho satisfatórios, novas alternativas de projeto podem ser exploradas para que ganhos maiores sejam atingidos.

7. REFERÊNCIAS

[1] Lee, J., Smith, A. J. 1984. Branch Prediction Strategies and Branch Target Buffer Design. *IEEE Computer Critical Systemst.* 17, 1 (Dez. 1993), 795-825. DOI= http://doi.acm.org/10.1109/MC.1984.1658927 .

[2] Yeh, T-Y., Marr, D., and Patt, Y. 1993. Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache. In *Proceedings of the International Conference on Supercomputing* (Tóquio, Japão, Julho 19 - 23, 1993).http://portal.acm.org/citation.cfm?id=165939.165956&coll=GUIDE&dl=GUIDE&CFID=92099187&CFTOKEN=51653586.

[3] Conte, T., et. al. 1995. Optimization of Instruction Fetch Mechanisms for High Issue Rates. In *Proceedings of the SIGARCH Computer Architecture News* (Santa Margherita Ligure, Itália, May, 1995). DOI= http://portal.acm.org/citation.cfm?id=225830.224444.

[4] Rotenberg, E., Bennet, S. and Smith, J. 1996. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture* (Paris, França, Julho 19 - 23, 1996)http://portal.acm.org/citation.cfm?id=243854.

[5] Patel, S. J. 1999. *Trace Cache Design for Wide-Issue Superscalar Processors*. Doctoral Thesis. University of Michigan. Acesso 30 de maio de 2010. http://sunzi.lib.hku.hk/ER/detail/hkul/2687580