Trace Cache

Divino César Soares
Instituto de Computação
Universidade de Campinas
divcesar@gmail.com

Raphael Zinsly
Instituto de Computação
Universidade de Campinas
raphael.zinsly@gmail.com

Resumo
Este trabalho apresenta a técnica de *trace cache*, um mecanismo que tem o objetivo de suprir os altos requisitos de vazão exigidos pelas unidades de execução *superscalares* modernas. *Trace cache* não vem substituir as *caches* de instruções e/ou dados, ao invés disso, é uma nova forma de *cache* de instruções onde blocos básicos do fluxo de instrução dinâmico são armazenados nas linhas da cache. Um estudo comparativo desta técnica é realizado.

Categorias e Descritores de Assunto
B.3.2 [**Design Styles**]: Cache Memories

Termos Gerais
Cache, Latencia, Vazão

Palavras Chaves
Trace Cache, Memória, Instruction Cache, Trace

1. INTRODUÇÃO
Processadores superescalares de alto desempenho consistem basicamente de duas partes: um mecanismo de *front-end* e um mecanismo de execução. Dentro do mecanismo de *front-end* fazem parte a busca de instruções, renomeação, decodificação e despacho. O mecanismo de execução realiza a entrega de instruções, a leitura dos registradores, a execução e a escrita do resultado. Entre os dois mecanismos, há os *buffers* de instrução, que tornam possíveis a execução fora de ordem. Devido ao fato de que técnicas para aumentar o paralelismo em nível de instrução são amplamente utilizadas no projeto de processadores superescalares, a janela de despachos vai se tornando cada vez maior e a predição de desvios mais profunda. Portanto, a largura de banda da busca de instruções se torna um gargalo para o desempenho. Outros fatores importantes também aparecem quando a taxa de instruções despachadas excedem quatro por ciclo: o número de desvios que podem ser despachados, o alinhamento de blocos de instrução não contínuos e a latência da unidade de busca.

A tarefa da unidade de busca é alimentar o decodificador com o fluxo dinâmico de instruções. Um problema é que as instruções são colocadas na *cache* de intruções em ordem de compilação. Esta forma favorece programas em que o código não tem desvios ou com blocos básicos grandes. O que não é o comportamento padrão. Uma *trace cache* [3] é uma *cache* de instruções especial que captura a sequência dinâmica das instruções. Uma *trace* é uma sequência de instruções de no máximo n instruções e m blocos básicos, partindo que qualquer ponto do fluxo dinâmico de instruções. O limite n é o tamanho da linha da *trace cache* e m é a taxa de transferência do mecanismo de predição de desvios.

A primeira vez que uma sequência de instruções é encontrada, ela é alocada a uma linha da *trace cache*. A linha é preenchida conforme as instruções são buscadas da *cache* de instruções. Se uma sequência de instruções for encontrada novamente durante a execução do programa, ela já está disponível na *trace cache* e é diretamente alimentada no decodificador. A *trace cache* se baseia na localidade temporal - uma sequência de instruções é armazenada na *cache* esperando que seja reusada - e no comportamento dos desvios - a maioria dos desvios tendem a seguir uma direção, é por isso que as predições costumam ter uma alta precisão.

A Seção 2 descreve o projeto e o funcionamento de uma *trace cache*, a Seção 2.3 apresenta questões relacionadas ao projeto de uma *trace cache* e seus pontos fracos. A Seção 3 descreve alternativas ao usa de uma *trace cache*. A Seção 4 traz uma comparação entre a *trace cache* e outras técnicas. Por fim, na Seção 5 são feitas algumas conclusões sobre o trabalho.

2. TRACE CACHE
Nesta seção será descrito o funcionamento da *trace cache* e como esta pode ser incluída no projeto de uma unidade de busca e decoficação simples. Note que a *trace cache* não substituirá as unidades de *cache* existentes mas sim adicionará novas funcionalidades ao sistema, por este motivo será dada uma visão geral do funcionamento de uma unidade de instruções.

2.1 A Unidade de Busca
A Figura 1 mostra os componentes de uma unidade de busca de instruções convencional e as interações entre eles. Este

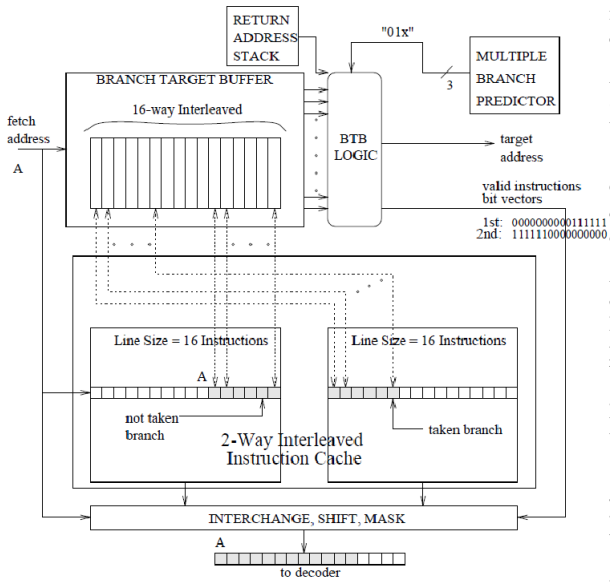

Figure 1: Unidade de simples busca de intruções.

mecanismo simples consegue fazer a busca de um número fixo de instruções por ciclo e/ou um número fixo de instruções de desvio, o limite que ocorrer primeiro encerra a busca. Esse limite no número de instruções é imposto pela largura do caminho de dados da unidade o qual neste trabalho é considerado 16, e o limite no número de instruções condicionais é definido pela capacidade do preditor de desvios, neste trabalho considerado como 3.

Portanto a arquitetura ilustrada é capaz de buscar no máximo 16 instruções, porém se um desvio condicional ocorrer neste intervalo a busca de intruções é encerrada com um número inferior de instruções.

Para conseguir buscar múltiplas instruções ao mesmo tempo, as unidades de cache devem permitir o acesso a diversas linhas simultaneamente, o que é alcançado quando a cache possui diversas portas de leitura e possui organização intercalada das linhas - o que permite que duas linhas armazenadas consecutivamente sejam lidas simultaneamente.

Porém essa capacidade da cache traz certa complexidade na recuperação dos dados. Para permitir que os dados sejam lidos da cache em paralelo, isto é, ler várias linhas simultaneamente, os dados são divididos em "bancos"(*cache banks*). Como o fluxo de instruções sendo executado apresenta uma estrutura rígida e suas instruções não podem ser executadas fora de ordem as instruções obtidas destes vários bancos precisam ser alinhadas antes de serem enviadas para a unidade de execução. As unidades de INTERCHANGE, SHIFT e MASK são responsáveis por realizar este trabalho.

Além da cache intercalada o sistema ilustrado na Figura 1 utiliza um buffer de endereços alvos de desvios [2] (*BTB - Branch Target Buffer*). A BTB funciona da seguinte maneira: Sempre que uma instrução é consultada na *cache* de instruções o endereço desta instrução é utilizado para indexar dentro de uma tabela da BTB, se o endereço estiver presente na BTB significa que a instrução é um desvio condicional (que foi armazenado na BTB em uma execução anterior desta instrução), na tabela da BTB está relacionado ao endereço da instrução o endereço alvo do desvio. Isso permite detectar quais das instruções constituem um desvio e prover, simultaneamente, 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 (máximo de 16) sendo buscadas na cache possam ser verificadas em paralelo se são de desvio ou não. Portanto, o nível de intercalação (16-way, neste caso) deve ser igual a quantidade de instruções armazenadas em uma linha da *cache*.

Além da previsão de desvios o BTB também é capaz de tratar instruções não condicionais como JUMP, RET, etc. Quando uma rotina inicia sua execução, como parte do preâmbulo de execução ela tem o endereço de retorno armazenado em uma pilha (para permitir alinhamento e recursão), essa pilha é representada pelo RAS (*Return Address Stack*) na Figura 1. Quando uma instrução RET é executada o endereço de retorno é retirado do topo da pilha de endereços de retorno.

Na próxima subseção será descrita a adição de um módulo de *trace cache* à esta arquitetura tradicional.

2.2 Adicionando a Trace Cache

A Figura 2 mostra a unidade de decoficação tradicional juntamente com o módulo de *trace cache*.

As instruções de um programa são armazenadas tanto na *cache* como na memória principal em uma forma "estática", por estática queremos dizer que a forma como as instruções estão organizadas não refletem necessariamente o comportamento do fluxo de instruções do programa. A forma como o programa está armazenado é, em geral, a que foi determinada como mais conveniente para o compilador.

As unidades tradicionais de busca e *cache* de instruções são capazes de explorar a localidade implícita na forma de representação e execução das instruções do programa. No entanto, a unidade de busca de instruções não é capaz de buscar instruções que pertençam a blocos básicos que não são adjacentes, isto é, a unidade de busca e cache é otimizada para a localidade espacial, porém, o fluxo dinâmico de instruções muitas vezes executa blocos básicos que não estão adjacentes (devido a desvios incondicionais).

O principal objetivo por trás do módulo de *trace cache* é capturar este comportamento dinâmico do fluxo de instruções e armazená-lo para o subsequente reuso.

As linhas da *trace cache* são reconhecidas como segmentos de instruções ou um *trace*, isto é um conjunto de blocos básicos que foram executados sequencialmente em algum momento durante a execução do programa. Além dos dados as linhas da *trace cache* contém informações de controle e lógica do buffer de abastecimento de linha.

A parte de dados de uma linha da *trace cache*, ou seja, o *trace* efetivamente. É armazenado como um segmento de

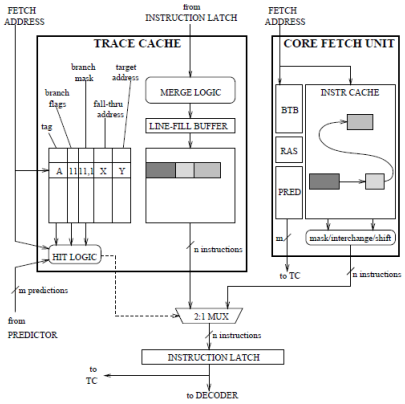

Figure 2: Busca de intruções com Trace Cache.

no máximo n instruções ou m blocos básicos, veja Figura 3. O número de instruções por linha n é determinado pela capacidade de armazenamento da cache e o número de blocos básicos m é determinado pela quantidade de previsões que o previsor de desvios é capaz de fazer.

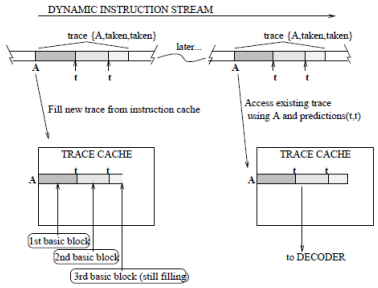

Figure 3: Criação e reuso de traces na trace cache.

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).

As informações de controle utilizadas nas linhas da *trace cache* são as seguintes:

1. **Bit de válidade:** este bit indica se a linha é válido.
2. **Tag:** rótulo do endereço de início do segmento.
3. **Flags de desvio:** bit indicando se o desvio foi ou não tomado.
4. **Máscara de 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.
5. **Fall-through address:** se o último desvio no segmento é previsto como não-tomado, este endereço é usado como a próximo endereço de busca.
6. **Target address:** se o último desvio no segmento é previsto como tomado, este endereço é usado como o próximo endereço de busca.

Um hit na trace cache requer que o endereço da busca combine com a *tag* e que as previsões de desvio combinem com as *flags* de desvio. O processo é como segue: O endereço da próxima instrução a ser buscada é utilizado como entrada simultaneamente à cache de instruções, à BTB e à *trace cache*. O previsor de desvio faz múltiplas previsões de desvios enquanto as *caches* são acessadas. O endereço da instrução é usado juntamente com as previsões de desvio para determinar se a sequência de instruções armazenados em uma linha da *trace cache* combina com a sequência de blocos básicos prevista. Quando ocorrer um *hit*, a linha inteira da *trace cache* é passada para o decodificador de instruções. Quando ocorre um *miss*, a busca de instruções continua normalmente na cache de instruções.

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 também criar e atualizar as *flags* de desvio, o *target address* e o *fall-through address*.

2.3 Decisões no Projeto de Trace Cache

Nesta subseção são considerados diversas questões relacionadas com o projeto das *caches de traces* além de elucidar alguns de seus pontos fracos.

1. **Associatividade:** As possíveis formas de mapeamento são basicamente as mesmas de uma *cache* de instruções tradicionais. Por exemplo, é possível utilizar um mapeamento direto ou uma forma de mapeamento mais complexa. Uma forma de mapeamento direto reduz o

hit time porém pode aumentar a quantidade de conflitos na *cache*. Uma forma de mapeamento mais complexa pode trazer uma taxa de acertos maior, porém ao preço de um maior espaço no chip, custo maior e lógica mais complexa.

2. **Criação de traces:** A principal questão que surge é: o que fazer quando um *trace* está sendo construído e há a necessidade de iniciar a criação de um outro *trace* (um *trace* iniciando em outro ponto do fluxo de intruções)? Um novo *trace* é criado sempre que a instrução de desvio atual não for encontrada na *trace cache*. Existem três respostas aceitáveis para esta pergunta:
(1) ignorar o novo *miss*.
(2) demorar a servir o novo *miss*, enquanto o *miss* anterior ainda está sendo servido pela unidade de abastecimento.
(3) implementar várias unidades de abastecimento para permitir que múltiplos *misses* sejam servidos ao mesmo tempo.

3. **Trace cache vítima:** a *trace cache* pode usar uma *cache* de vítima para armazenar os *traces* que são descartados quando um novo *trace* precisar ser armazenado e não houver espaço livre.

4. **Seleção de segmento:** na implementação trivial da *trace cache* todos os *traces* são armazenados, isto pode ser uma desvantagem pois nem todos os *traces* são frequentemente reutilizados. Uma melhoria seria o armazenamento de *traces* que foram executados mais do que um limiar.

5. **Decodificação:** um adicional para o projeto de *trace cache* é a decodificação das instruções a medida que os *traces* são formados/armazenados na *cache*.

6. **Caminhos múltiplos:** uma desvantagem da *trace cache* é a duplicação de partes dos *traces* armazenados. Considere dois *traces* que possuem o mesmo prefixo mas não são iguais, estes *traces* serão armazenados em duas linhas diferentes da *cache* e utilizarão quase o dobro do espaço para armazenar a mesma informação.

7. **Partial Matches:** outra desvantagem da *trace cache* é a não aceitação de casamentos parciais de *traces*, isto é, todo o padrão de desvio do *trace* deve ser igual ao padrão de desvio predito para o fluxo de instruções. Se fosse possível a execução de um prefixo do *trace* armazenado certamente teríamos ganhos de performance.

Na próxima seção serão mostrados duas alternativas à *trace cache* existentes.

3. OUTROS MECANISMOS DE BUSCA EM LARGA ESCALA

Há outras abordagens que também solucionam o problema de buscar várias instruções por ciclo. Duas dessas abordagens serão descritas nesta seção e serão usadas para comparação com a *trace cache*. A primeira delas é o *collapsing buffer* [1], ele é composto por uma cache de instruções intercalada; um buffer de alvo de desvio intercalado (BTB); um previsor de desvios múltiplos; e uma rede de alinhamento e intercâmbio. Ele é bem similar ao mecanismo de busca convencional descrito na Seção 2, com duas diferenças, uma deles é que ele possui uma lógica especial do BTB que permite buscar por desvios em duas linhas diferentes da cache. A outra diferença é o fato de que 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.

A segunda abordagem é a *cache* de endereço de desvios [5] (*Branch Addresses Cache*), ela é composta por quatro componentes: uma *cache* de endereço de desvio; um previsor de desvio múltiplo; uma *cache* de instrução intercalada; e uma rede de alinhamento e intercâmbio. A *cache* de endereço de desvios estende o BTB para múltiplos previsores por armazenar uma árvore de endereços alvo e *fallthrough*, onde o número de níveis da árvore depende da quantidade de desvios que podem ser previstos durante o mesmo ciclo.

As duas técnicas desempenham suas tarefas em tempo de compilação, ao contrário da *trace cache* que o faz em tempo de execução. A complexidade de ambas as 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.

4. RESULTADOS

Os resultados são medidos através do número de instruções completadas por ciclo (IPC), que é uma métrica de desempenho. Foram utilizados seis *benchmarks* de inteiros SPEC92 e seis *benchmarks* de ponto flutuante do *Instruction Benchmark Suite* (IBS) [4]. A média harmônica é usada para fazer a média de desempenho dos benchmarks. A comparação será feita entre a *trace cache* (TC) e as técnicas *collapsing buffer* (CB) e *cache* de endereço de desvios (BAC). A Figura 4 indica os parâmetros da unidade de busca utilizados em todos os experimentos.

SIMULATION PARAMETER		INSTR 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/cyc	up to 3 predictions each cycle		
Instr Cache	size	128 KB		
	assoc	direct mapped		
	interleave	2-way	2-way	8-way
	line size	16 instr	16 instr	4 instr
	prefetch	none	none	3 lines
	miss penalty	10 cycles		
Ret Stack	depth	unlimited		
Branch Target Buffer	size	1K entries	1K entries	n/a
	assoc	dir map	dir map	
	interleave	16-way	16-way	
Trace Cache	size	64 entries	n/a	
	assoc	dir map		
	line size	16 instr		
	# conc fills	1		
Branch Address Cache	size	n/a		1K entries
	assoc			dir map
	# conc fills			1

Figure 4: Parâmetros da unidade de busca

A base para as comparações foi a unidade de busca da *trace cache* descrita na Seção 2, esta base será chamada de "sequencial"(SEQ), já que somente instruções sequenciais podem ser buscadas em um dado ciclo. Duas variações de SEQ foram simuladas: SEQ.1 admite apenas um bloco básico por ciclo e SEQ.3 pode buscar até quatro blocos básicos contínuos. Os parâmetros de SEQ são os mesmos de TC, mas sem a *trace cache*.

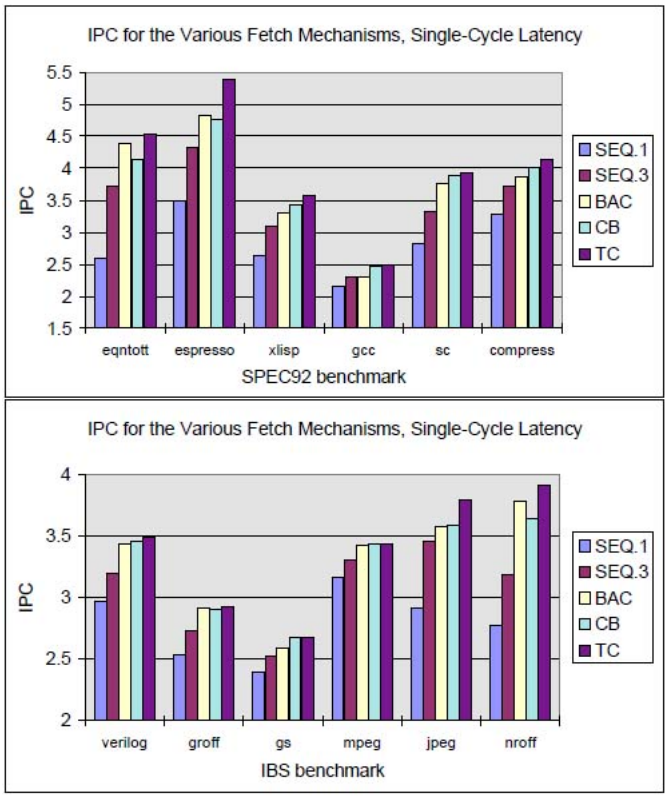

Figure 5: Resultados em IPC (1 ciclo de latência)

Os resultados são divididos em dois conjuntos. O primeiro conjunto assume que a unidade de busca tem somente 1 ciclo de latência, isso foi feito para isolar a habilidade dos mecanismos de fornecer largura de banda para instruções. A Figura 5 mostra um gráfico comparando as técnicas nos *benchmarks* SPEC e um gráfico nos *benchmarks* IBS, é possível perceber que o SEQ.3 tem um desempenho maior que o SEQ.1. A Figura 6 mostra a melhora do SEQ.3 sobre o SEQ.1. O gráfico da Figura 7 mostra a melhora do desempenho de TC, CB e BAC sobre SEQ.3.

O segundo conjunto de testes mostra o que acontece quando os estágios extras no pipeline inseridos pelas técnicas BAC e CB são simulados. A Figura 8 mostra os resultados com as latências de 2 (L2) e 3 (L3) ciclos das técnicas BAC e CB. É possível ver que estas técnicas tiveram um desempenho bem menos que TC e em alguns casos pior que SEQ.3, usado como base. Outro ponto é que os *benchmarks* de inteiros sofrem mais com o aumento da latência que os de ponto flutuante.

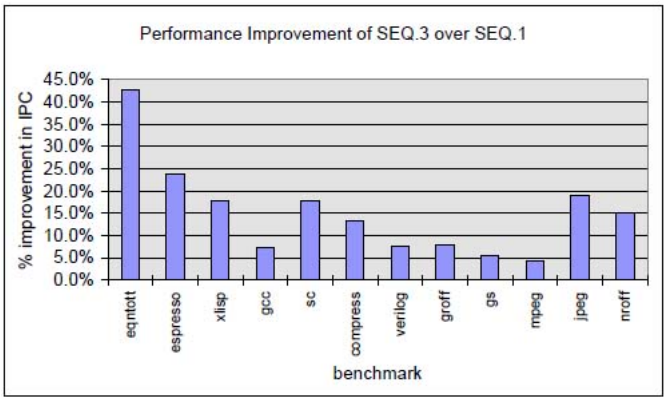

Figure 6: Melhora do SEQ.3 sobre o SEQ.1

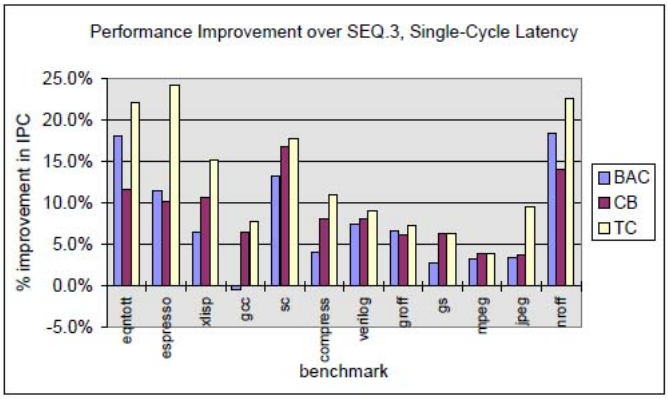

Figure 7: Melhora das téquinas comparadas ao SEQ.3

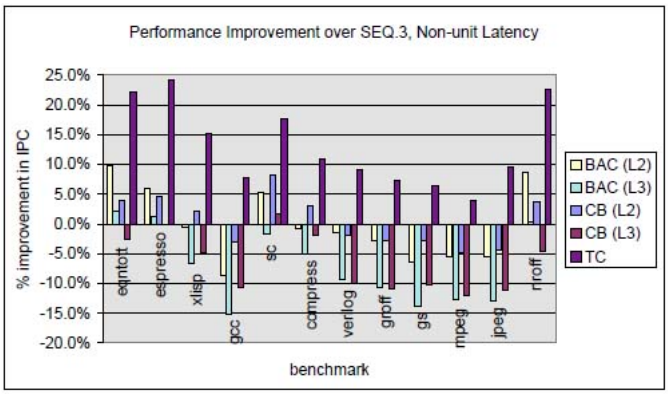

Figure 8: Desmpenho das técnicas comparadas ao SEQ.3

5. CONCLUSÃO

A *trace cache* se mostra como uma abordagem eficiente e superior as alternativas propostas. É importante projetar mecanismos de busca capazes de buscar vários desvios por ciclo; no entanto, esse mecanismo não deve adicionar muita

latência para alcançar esse objetivo. A *trace cache* consegue alcançar os dois objetivos.

Porém é preciso continuar pesquisando projetos que alcancem melhor eficiência e explorar novas alternativas, pois ainda pode ser possível melhorar ainda mais o processo de busca de instruções.

6. REFERÊNCIAS

[1] T. Conte, K.Menezes, P.Mills, and B. Patel. Optimization of instruction fetch mechanisms for high issue rates. In *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995.

[2] J. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer Critical Systems*, 1993.

[3] E. Rotenberg, S. Bennet, and J. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*, 1996.

[4] R. Uhlig, D. Nagle, T. Mudge, S. Sechrest, and J. Emer. Instruction fetching: Coping with code bloat. In *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995.

[5] T. Y. Yeh, D. T. Marr, and Y. N. Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. In *Proceedings of the 7th International Conference on Supercomputing*, 1993.