Trace Caches: Uma Abordagem Conceitual

Roberto Pereira, RA:089089
Instituto de Computação – IC
Universidade Estadual de Campinas – Unicamp
Av. Albert Einstein, 1251, Campinas - SP
robertop.ihc@gmail.com

ABSTRACT

Superscalar processors are constantly searching for improvements in order to become more efficient. The fetch engine of instructions is one of the bottlenecks of superscalar processors, because it limits the improvements that can be achieved trough the parallel execution of instructions. The Trace Cache is a solution that aims to achieve a high bandwidth in instruction fetch through the storage and reuse of traces of instructions that are dynamically identified. This paper presents a conceptual approach about trace cache and describes how it can help for improving the provision of instructions for execution on superscalar processors. We also present some related work and some proposals for improving the original proposal, as well as our final considerations on the study.

RESUMO

Os processadores superescalares estão numa busca constante por aperfeiçoamentos de modo a tornarem-se mais eficientes. O mecanismo de busca de instruções é considerado um dos principais gargalos dos processadores superescalares, limitando as melhorias que podem ser obtidas pela execução paralela das instruções. A *Trace Cache* é uma solução proposta para alcançar uma alta largura de banda na busca de instruções por meio do armazenamento e reutilização de *trace*s de instrução que são identificados dinamicamente. Neste artigo, apresentamos uma abordagem conceitual sobre *trace cache* e descrevemos como a mesma pode colaborar para melhorar o fornecimento de instruções para a execução em processadores superescalares. Também apresentamos alguns trabalhos relacionados e algumas propostas de aperfeiçoamento da proposta original, assim como nossas considerações finais sobre o estudo realizado.

Categorias e Termos Descritores

B.3.2 [**Hardware\Memory Structures**]: *Cache* Memories

Termos Gerais

Documentation, Performance, Design, Theory.

Palavras-Chave

Trace Cache, Desempenho, Superescalar, ILP.

1. INTRODUÇÃO

Os processadores superescalares possuem *pipelines* que permitem a execução de mais de uma instrução no mesmo ciclo de *clock*, ou seja, simultaneamente [1][2][3]. Para alcançar um alto desempenho (*performance*), a largura de banda de execução dos processadores modernos tem aumentado progressivamente. Entretanto, para que esse aumento na banda se converta em melhora de desempenho, a unidade de execução precisa ser provida com um número cada vez maior de instruções úteis por ciclo (IPC).

De acordo com Rotemberg *et al.* [1], a organização dos processadores superescalares possui dois mecanismos distintos e claramente divididos chamados de "busca" (*fetch*) e "execução" (*execution*) de instruções, separados por *buffers* (que podem ser filas, estações de reserva, etc.) que atuam como um banco de instruções. Neste contexto, o papel do mecanismo de busca é buscar as instruções, decodificá-las e colocá-las no *buffer*, enquanto o papel do mecanismo de execução é retirar e executar essas instruções levando em conta as suas dependências de dados e as restrições de recursos.

Segundo Rotemberg *et al.* [2], os *buffers* de instrução são coletivamente chamados de "janela de instrução" (*instruction window*) e, é nessa janela, que é possível se aplicar os conceitos do Paralelismo em Nível de Instrução (ILP) nos programas seqüenciais. Naturalmente, quanto maior for essa janela, maiores serão as oportunidades de se encontrar instruções independentes que possam ser executadas paralelamente. Assim, existe uma tendência no projeto de processadores superescalares em se construir janelas de instrução maiores e, ao mesmo tempo, de se prover uma maior emissão e execução de instruções. Além disso, normalmente, o mecanismo de execução de instruções nos processadores superescalares é composto por unidades funcionais paralelas [1], e o mecanismo de busca pode especular múltiplos desvios objetivando fornecer um fluxo de instruções contínuo à janela de instruções de modo a tirar o máximo de proveito do ILP disponível.

Entretanto, como argumentado em Rotemberg *et al.* [1] [2] e em Berndl & Hendren [4], para conseguir manter essa tendência de aumentar o desempenho aumentando a escala na qual as técnicas de ILP são aplicadas, é preciso encontrar um equilíbrio entre todas as partes do processador, uma vez que um gargalo em qualquer uma das partes diminuiria os benefícios obtidos por essas técnicas. Neste contexto, os trabalhos de [1] [2] [3] [4] e [5] citam claramente que ocorre um grande aumento na demanda pelo fornecimento de instruções para serem executadas e que isso, normalmente, se torna um gargalo que limita o ILP. Afinal, de que adianta duplicar recursos e largura de banda se apenas uma quantidade limitada e inferior de instruções pode ser buscada e decodificada simultaneamente? Ou seja, o pico da taxa de busca de instruções deve ser compatível com o pico da taxa de despacho de instruções.

Assim, o fornecimento de instruções se torna um elemento-chave no desempenho dos processadores superescalares [4], pois como mencionam Rotemberg *et al*. [2], as unidades de busca normalmente eram limitadas a uma única previsão de desvio por

ciclo. Conseqüentemente, não era possível buscar mais de um bloco básico[1] a cada ciclo de *clock*. Devido à grande quantidade de desvios existentes em programas típicos e ao tamanho médio pequeno dos blocos básicos[2], Postiff *et al.* [4] afirmam que buscar instruções pertencentes à múltiplos blocos básicos em um único ciclo é algo crítico para evitar gargalos e proporcionar um bom desempenho.

No que diz respeito a busca de instruções, buscar um único bloco básico a cada ciclo é suficiente para implementações que entregam no máximo até quatro instruções por ciclo. Entretanto, de acordo com Rotemberg *et al*. [1] [2], essa quantidade não é suficiente para processadores com taxas mais elevadas. Ao conseguir prever múltiplos desvios, torna-se possível buscar múltiplos blocos básicos contínuos em um mesmo ciclo. Porém, o limite da quantidade de instruções que podem ser buscadas ainda está relacionado à freqüência de desvios que são tomados, pois nesse caso, é preciso buscar as instruções que estão abaixo do desvio tomado no mesmo ciclo no qual o desvio foi buscado. É neste contexto que Rotemberg *et al*. [1] apresenta a proposta de *Trace Cache*.

O mecanismo de *Trace Cache* é uma solução para o problema de se realizar o *fetch* de múltiplos desvios em um único ciclo [4]. Ela armazena o rastreamento do fluxo de execução dinâmico das instruções, de modo que instruções que antes eram não contínuas (separadas por desvios condicionais) passam a aparecer de forma contínua. Segundo Rotemberg *et al.*[1], idealizadores da *trace cache*, a quantidade de desvios condicionais que podem ser previstos em um único ciclo, o alinhamento de instruções não contínuas e a latência da unidade de busca, são questões que devem ser tratadas, e que são consideradas na proposta de *trace cache* para melhorar o desempenho dos processadores superescalares.

Este artigo está organizado da seguinte forma: na seção 2 explica-se o conceito de *trace cache* e demonstra-se sua estrutura e aplicação. Na seção 3 apresenta-se alguns trabalhos relacionados e, também, algumas discussões sobre propostas de aperfeiçoamentos da *trace cache* original proposta por Rotemberg *et al* [1] [2]. Finalmente, na seção 4 são expostas as considerações finais sobre o trabalho.

2. *TRACE CACHE*

A principal função da unidade de busca (*fetch*) é fornecer um fluxo dinâmico de instruções ao decodificador (*decoder*) [5]. Entretanto, Rotenberg *et al.* [1] mencionam que as instruções são colocadas na *cache* de acordo com sua ordem de compilação, o que é favorável para códigos nos quais não há desvios ou que possuem grandes blocos básicos: mas essa não é a realidade comum dos programas normalmente encontrados. Como os autores demonstram em [1] e [2] por meio de estatísticas obtidas da análise de códigos de inteiros (*integer codes*), o tamanho dos blocos básicos é de 4 a 5 instruções, e o percentual de desvios tomados varia de 61 a 86%.

Desta forma, Rotenberg *et al.* [1] propõem uma *cache* de instruções especial para capturar a seqüência dinâmica das instruções: a *trace cache*. De acordo com a Figura 1, cada linha da *trace cache* armazena determinado instante (*trace*) do fluxo de instrução dinâmico. Um *trace* é **uma seqüência de no máximo "n" instruções** (determinado pelo tamanho da linha da *cache*) **e de no máximo "m" blocos básicos** iniciando de qualquer ponto do fluxo dinâmico de instruções (determinado pelo *throughput* do previsor de desvios). Assim, um *trace* é especificado por um endereço inicial e por uma seqüência de até m-1 resultados de desvios que descrevem o caminho seguido.

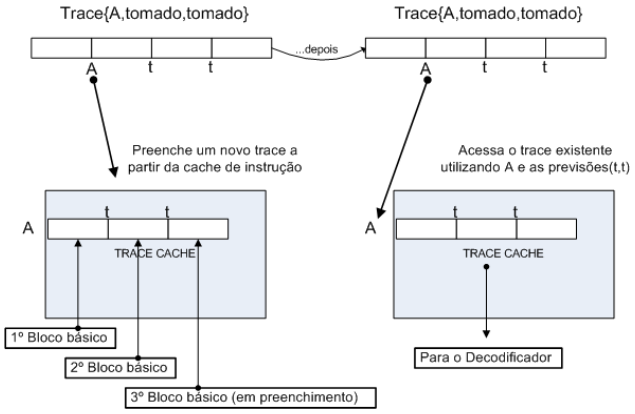

Figura 1 – Visão em alto nível da *trace cache* (adaptado de [1]).

Na primeira vez que um *trace* é encontrado, aloca-se uma linha na *trace cache* e, de acordo com a busca das instruções da *cache* de instruções, preenche-se a linha da *trace cache*. Se no tempo de execução uma mesma linha for encontrada novamente (verifica-se isso comparando o endereço inicial e as previsões para os desvios), ela já estará disponível na *trace cache* e poderá ser fornecida diretamente ao decodificador. Caso contrário, a busca ocorrerá normalmente da *cache* de instruções. Deste modo, é possível observar que a abordagem utilizada pela *trace cache* **fundamenta-se em seqüências dinâmicas de código sendo reutilizadas**. Isso está relacionado ao princípio da **localidade temporal** [1] (instruções utilizadas recentemente tendem a ser utilizadas novamente em um futuro próximo) **e ao comportamento dos desvios** (boa parte dos desvios tende a possuir um viés para alguma direção, devido a isso é que a exatidão da previsão de desvio normalmente é alta).

Para ilustrar, considere o exemplo de uma seqüência dinâmica de blocos básicos ilustrado pela Figura 2(a) — as setas indicam os desvios tomados. Mesmo com múltiplas previsões de desvios por ciclo, para buscar as instruções dos blocos básicos "ABCDE" seriam necessários 4 ciclos. Isso se deve ao fato de que as instruções se encontram armazenadas em lugares não contínuos. Agora considerando o exemplo da Figura 2(b), a mesma seqüência de blocos que aparecia de forma não contínua na *cache* de instrução, aparece agora de forma contínua na *trace cache*.

[1] Define-se por bloco básico um trecho de código que não possui desvios, a não ser, talvez, da sua última instrução.

[2] Rotenberg *et al.* [2] sugerem um tamanho médio de 4 a 6 instruções por bloco básico.

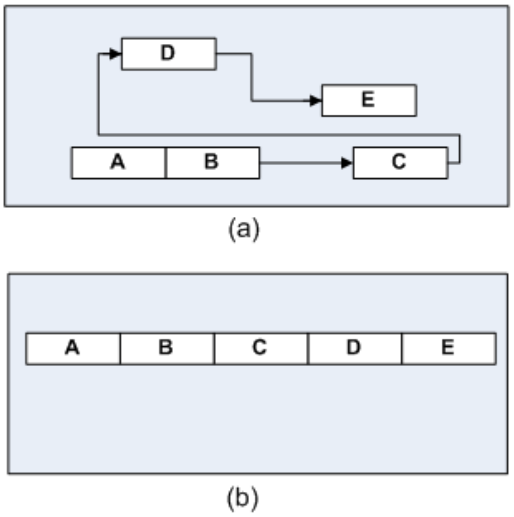

Figura 2 – Armazenamento de seqüências não contínuas de instruções. (a) *Cache* de Instrução. (b) *Trace Cache*. (adaptado de [2]).

2.1 Um exemplo Prático

Durante a execução de um programa, a *trace cache* verificará o resultado das previsões de desvios e, caso eles sejam tomados, buscará as instruções necessárias antecipadamente, obtendo as instruções corretas que realmente serão executadas. Se houver erro na previsão do desvio, então será necessário voltar e buscar as instruções corretas na memória. Para ilustrar o funcionamento da *trace cache*, considere o código do exemplo abaixo, extraído de [13]. Este código corresponde a um laço de repetição no qual existe um desvio que sempre é tomado.

Endereço	Instrução	
000F:0001	AND	EBX, ECX
000F:0002	DEC	EAX
000F:0003	CMP	EAX, EBX
000F:0004	JL	0007 *//salta para a instrução 7*
000F:0005	PUSH	EAX
000F:0006	JMP	0001
000F:0007	PUSH	EBX *//continua a execução*
000F:0008	XOR	EAX, EBX
000F:0009	ADD	EAX, 1
000F:000A	PUSH	EAX
000F:000B	MOV	EAX, ESI
000F:000C	JMP	0001 *//retorna para o topo*

Quadro 1 – Código-Fonte para Exemplificação. (adaptado de [13]).

O fluxo seqüencial de código, seguindo-se pelo endereço, seria: 1, 2, 3, 4, 7, 8, 9, A, B, C, devido ao desvio tomado na instrução 4 que ocasiona o salto para a instrução 7. Uma cache convencional capturaria as instruções em ordem seqüencial (1, 2, 3, 4, 5,...). Ao identificar o desvio tomado em 4, a *trace cache* é capaz de carregar as instruções dos dois blocos básicos (1, 2, 3, 4) e (7, 8, 9, A, B, C) numa forma contínua, de modo que o resultado seja: (1, 2, 3, 4, 7, 8, 9, A, B, C). Deste modo, a *trace cache* evitou que instruções desnecessárias ocupassem espaço e fossem carregadas. Dependendo da quantidade de instruções que podem ser carregadas simultaneamente, algumas instruções úteis poderiam ter ficado de fora, as instruções estariam em uma ordem errada e, portanto, isso geraria um *miss* na *chace*, deixando o processo mais lento.

2.2 A Arquitetura da *Trace Cache*

Em Rotemberg *et al.* [1] e [2] é possível encontrar a arquitetura proposta originalmente para a *trace cache* de forma detalhada. Nesta subseção, apresenta-se os seus principais componentes e conceitos de forma simplificada.

A arquitetura da *trace cache*, ilustrada pela Figura 3, foi concebida com o objetivo de prover uma alta largura de banda na busca de instruções com uma baixa latência [2].

O previsor do próximo *trace* trata os *trace*s como unidades básicas predizendo explicitamente seqüência de *trace*s. Jacobson *et al.* [6] argumentam que essa predição explícita, não somente remove restrições relacionadas ao número de desvios em um *trace*, como também colabora para o alcance de uma taxa média de exatidão de previsão mais alta do que seria obtida por meio de previsores simples. A saída produzia pelo previsor é o "identificador de *trace*" (*trace identifier*), o qual permite identificar um determinado *trace* de forma única pelo seu PC (*Program Count*) e pelas saídas de todos os desvios condicionais que compõem o *trace*[3]. Deste modo, um *trace cache* hit ocorre quando algum *trace* existente corresponde exatamente com o identificador de *trace* previsto.

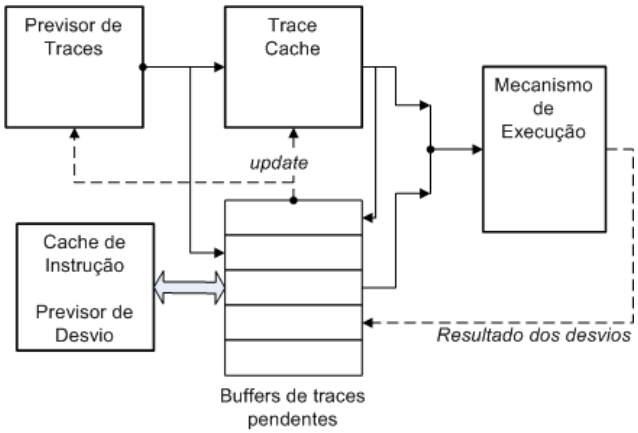

Figura 3 – Micro-arquitetura (adaptado de [2]).

De acordo com Rotemberg *et al.* [2], o previsor de *trace* e a *trace cache* juntos, proporcionam um rápido seqüenciamento em nível de *trace*. Porém, isso nem sempre proporciona o *trace* necessário, principalmente quando se está iniciando um programa ou quando se alcança uma região nova no código que ainda é desconhecida tanto para o previsor quanto para a própria *trace cache*. Para compensar essa limitação, faz-se necessário utilizar, também, o seqüenciamento em nível de instrução.

O *buffer* de *trace*s pendentes (*outstanding trace buffers*) é utilizado tanto para construir novos *trace*s que não estão na *trace cache*, quanto para rastrear as saídas dos desvios tão logo elas estejam disponíveis no mecanismo de execução, permitindo assim, a detecção de previsões erradas e a correção dos *trace*s que

[3] Essa decisão de projeto também permite a existência de diferentes *trace*s que começam com um mesmo endereço (redundância).

a compõem [2]. Ainda considerando a Figura 3, cada *trace* buscado é despachado simultaneamente para o mecanismo de execução e para o *buffer*. Assim, no caso de um *miss* (de não encontrar o *trace* desejado na *trace cache*), apenas a previsão do *trace* é recebida pelo *buffer* alocado. A própria previsão fornece informações suficientes para se construir o *trace* a partir da *cache* de instruções[4].

Com relação ao *trace cache hit* e *trace cache miss*, podemos simplificar a explicação da seguinte forma: o mecanismo de busca apresenta simultaneamente um endereço para a *trace cache*, para a *cache* convencional e para a unidade de previsão de desvios. **Se** a *trace cache* possuir um *trace* que se inicia com o mesmo endereço e que também é compatível com a informação de previsão de desvios, **então** ocorre um *hit* e o respectivo *trace* é retornado. **Se** a *trace cache* possui um endereço compatível, mas as informações de previsão não conferem completamente, **então** ocorre um *hit* parcial. Nesse caso, a *cache* de instrução é acessada simultaneamente com a *trace cache*[5]. Nos casos em que a *trace cache* não contém um *trace* começando com o endereço especificado, então ocorre um *trace cache miss*. A *cache* de instrução fornece então a linha contendo o endereço requerido para a unidade de execução e um novo *trace* vai sendo construído.

A arquitetura proposta para a *trace cache* foi validada pelos seus idealizadores em [1] [2] e [6], e também foi discutida por uma série de outros trabalhos, alguns dos quais serão brevemente citados na próxima seção. Os principais resultados obtidos por meio de simulações comparativas permitiram afirmar que a *trace cache* melhora o desempenho numa faixa de 15 a 35% quando comparada a outros mecanismos de busca de múltiplos blocos igualmente sofisticados (mas contínuos).

Também foi possível observar que *trace*s maiores melhoram a exatidão do previsor de *trace*s, enquanto *trace*s menores proporcionam uma taxa de exatidão inferior. Em [1] [2] e [6] os autores concluíram, também, que o desempenho global não é tão afetado pelo tamanho e pela associatividade da *trace cache* do modo como seria o esperado. Isso se deve parcialmente ao robusto seqüenciamento em nível de instrução. De acordo com os resultados obtidos, o IPC não variou mais que 10% em uma ampla gama de configurações.

Outro ponto que deve ser destacado é que a vantagem em se utilizar uma *trace cache* é obtida ao preço da redundância no armazenamento de instruções (vários *trace*s com diferentes previsões de desvios — Ramirez *et al.* [12] apresentam críticas e alternativas para essa questão). Isso pode causar efeitos colaterais na eficiência da *trace cache* devido ao seu limite de tamanho. Finalmente, Rotemberg *et al*. [1][2] e Jacobson *et al*. também concluíram que uma *cache* de instrução combinada com um previsor "agressivo" pode buscar qualquer número de blocos básicos por ciclo, rendendo um aumento de 5 a 25% sobre buscas que retornam um único bloco.

[4] Esse procedimento normalmente leva vários ciclos devido aos desvios previstos como tomados.

[5] A menos que seja necessário economizar energia e, portanto, o acesso não deva ser realizado em paralelo.

2.3 Aplicação Comercial

De acordo com as informações de [13], a *trace cache* é utilizada em processadores modernos como o *Pentium IV*, o *Sparc* e o *Athlon*. O *Pentium IV* é um processador superescalar que se utiliza de paralelismo em nível de instrução. De acordo com Hinton *et al.* [14], a *trace cache* é o nível primário de memória *cache* no *Pentium IV* (L1), conseguindo armazenar até doze mil microoperações, e podendo entregar até três instruções em um único ciclo de *clock*. Ainda, a *trace cache* do *Pentium IV* possui uma taxa de *hit* comparável com a de uma *cache* de instruções de 8 a 16 KB, e a sua aplicação visa, principalmente, minimizar o gargalo existente entre a decodificação de instruções e a sua respectiva execução.

O gargalo entre as unidades de decodificação e de execução de instruções ocorre porque o *Pentium IV* possui um conjunto de instruções complexo. Isso implica na necessidade de haver um decodificador para transformar as instruções complexas em micro-instruções simplificadas para então enviá-las ao *pipeline*. A *trace cache* colabora para que instruções que foram usadas recentemente não precisem ser decodificadas novamente e, conseqüentemente, colabora para uma redução no gargalo [13], além de prover os benefícios já mencionados nas seções anteriores. Em Hennessy & Patterson [5] é possível encontrar várias informações e discussões sobre a utilização da *trace cache* no *Pentium IV*. Provavelmente, esta foi uma das primeiras utilizações da *trace cache* em processadores comerciais.

3. TRABALHOS RELACIONADOS

Nesta seção, apresenta-se brevemente alguns detalhes sobre trabalhos que forneceram bases para o desenvolvimento da proposta de *trace cache*, bem como trabalhos posteriores que propõem modificações e aperfeiçoamentos na proposta original.

3.1 Trabalhos Anteriores

Dentre os trabalhos que geraram idéias para a proposta de trace cache, Rotemberg *et. al* [1] e [2] enfatizam as iniciativas de Yeh *et al.* [7] e Dutta *et al.* [8], descritas abaixo. Segundo Rotenberg *et al.* [2], outras iniciativas de utilização da trace cache foram propostas de forma independente pela comunidade de pesquisa, e maiores comentários sobre as mesmas podem ser encontrados diretamente em [1] e [2].

A pesquisa de Yeh *et al*. [7] considerou um mecanismo de busca que proporcionava uma alta largura de banda ao prever múltiplos endereços de destino dos desvios a cada ciclo — os autores propuseram uma *cache* de endereços de desvios como sendo uma evolução natural do *buffer* de destino de desvios. Assim, um *hit* nessa *cache* combinado com múltiplas previsões de desvios produzia o endereço de início de vários blocos básicos.

A proposta de Franklin *et al.* [8] se baseia em uma abordagem similar à citada anteriormente em Yeh *et al.* [7], com uma diferença na forma de prever múltiplos desvios em um único ciclo: eles incorporaram múltiplas previsões em uma única previsão. Por exemplo: em vez de fazer duas previsões de desvio selecionando um entre dois caminhos, faz-se uma única previsão selecionando um entre quatro caminhos possíveis.

3.2 Propostas e Estudos sobre *Trace Cache*

O trabalho de Postiff *et al.* [4] compara o desempenho da *trace cache* com o limite teórico de um mecanismo de busca de três blocos. Os autores definem várias métricas novas para formalizar a análise da *trace cache* (por exemplo: métricas de fragmentação, duplicação, indexabilidade e eficiência). Neste estudo, os autores demonstram que o desempenho utilizando a *trace cache* é mais limitado por erros na previsão dos desvios do que pela capacidade de se buscar vários blocos por ciclo. Segundo eles, na medida em que se melhora a previsão de desvios, a alta duplicação e a conseqüente baixa eficiência são percebidas como uma das razões pelas quais a *trace cache* não atinge o seu limite máximo de desempenho.

Em [9], os autores exploram a utilização do compilador para otimizar a disposição das instruções na memória. Com isso, possibilita-se permitir que o código faça uma melhor utilização dos recursos de *hardware*, independentemente de detalhes específicos do processador ou da arquitetura, a fim de aumentar o desempenho na busca de instruções. O *Software Trace Cache* (STC) é um algoritmo de *layout* de código que visa não apenas melhorara taxa de *hit* na *cache* de instruções, mas também, um aumento na largura da busca efetiva do mecanismo de *fetch* (quantidade de instruções que podem ser buscadas simultaneamente). O STC organiza os blocos em cadeias tentando fazer com que blocos básicos executados seqüencialmente residam em posições contínuas na memória. O algoritmo mapeia a cadeia de blocos básicos na memória para minimizar os conflitos de *miss* em seções críticas do programa. O trabalho de Ramirez *et al.* [9] apresenta uma análise e avaliação detalhada do impacto do STC, e das otimizações de código de uma forma geral, em três aspectos principais do desempenho de busca de instruções: a largura efetiva da busca, a taxa de *hit* na *cache* de instrução, e a exatidão da previsão de desvios. Os resultados demonstram que os códigos otimizados possuem características especiais que os tornam mais propícios para um alto desempenho na busca de instruções: possuem uma taxa muito elevada de desvios não-tomados e executam longas cadeias seqüenciais de instruções, além de fazerem um uso eficaz das linhas da *cache* de instruções, mapeando apenas instruções úteis que serão executadas em um curto intervalo de tempo, aumentando, assim, tanto a localidade espacial quanto a localidade temporal.

Em [10], os autores apresentam uma nova *trace cache* baseada em blocos e que, segundo simulações, poderia alcançar um maior desempenho de IPC com um armazenamento mais eficiente dos *trace*s. Em vez de armazenar explicitamente as instruções de um *trace*, os ponteiros para blocos que constituem o *trace* são armazenados em uma tabela de *trace*s muito menor do que a utilizada pela *trace cache* original. A *trace cache* baseada em blocos proposta pelos autores renomeia os endereços de busca em nível de bloco básico e armazena os blocos alinhados em uma *cache* de blocos. Os blocos são construídos pelo acesso à *cache* de blocos utilizando os ponteiros de bloco da tabela de *trace*. Os autores compararam o desempenho potencial da proposta com o desempenho da *trace cache* convencional. De acordo com os resultados demonstrados em [10], a nova proposta pode alcançar um IPC maior com um impacto menor no tempo de ciclo.

Hossain *et al.* [11] apresentam parâmetros e expressões analíticas que descrevem o desempenho da busca de instrução. Os autores implementaram expressões analíticas em um programa utilizado para explorar os parâmetros e suas respectivas influências no desempenho do mecanismo de *fetch* de instruções (o programa é denominado de *Tulip*). As taxas de busca de instrução previstas pelas expressões propostas pelos autores apresentaram uma diferença de 7% comparadas com as taxas apresentadas pelos programas do *benchmark SPEC2000*. Além disso, o programa também foi utilizado para tentar identificar e compreender tendências de desempenho da *trace cache*.

Em [12] os autores argumentam que os recursos de *hardware* dos mecanismos da *trace cache* podem ter seu custo de implementação reduzido sem ocasionar perda de desempenho se a replicação de *trace*s entre a *cache* de instrução e a *trace cache* for eliminada. Os autores demonstram que a *trace cache* gera um alto grau de redundância entre os *trace*s armazenados na *trace cache* e os *trace*s gerados pelo compilador e que já estão presentes na *cache* de instrução. Além disso, os autores também abordam que algumas técnicas de reorganização de código, como a STC apresentada em [9], adotam estratégias que colaboram para aumentar ainda mais a redundância. Deste modo, a proposta do trabalho de [12] é efetuar um armazenamento seletivo dos *trace*s de modo a evitar a redundância entre a *cache* de instruções e a *trace cache*. Segundo os autores, isso pode ser obtido modificando-se a unidade que preenche os novos *trace*s para que ela armazene apenas os *trace*s de desvios tomados (uma vez que eles não podem ser obtidos em um único ciclo). Os resultados exibidos demonstram que, com as modificações propostas, uma *trace cache* de 2KB com 32 entradas apresenta um desempenho tão bom quanto uma *trace cache* de 128 KB com 2048 entradas (sem as modificações). Isso enfatiza, segundo [12], que a cooperação entre *software* e *hardware* é fundamental para aumentar o desempenho e reduzir os requisitos de *hardware* necessários para o mecanismo de *fetch* de instruções.

4. CONSIDERAÇÕES FINAIS

A abordagem de *trace cache* apresentada neste artigo se constitui como uma forma efetiva de aumentar a capacidade de fornecimento de instruções para os processadores superescalares. Como mencionado nas seções anteriores, para que seja possível aproveitar os benefícios do paralelismo em nível de instrução, os processadores superescalares precisam que um grande número de instruções seja decodificado e esteja pronto para ser executado a cada ciclo. Deste modo, ao utilizar uma abordagem dinâmica para identificar o fluxo de execução e, assim, conseguir prever desvios e organizar os blocos básicos de forma contínua, a *trace cache* colabora para que mais instruções possam ser buscadas em um único ciclo de *clock*.

Deste modo, o mecanismo de *trace cache* pode ser compreendido como uma solução para o problema de se realizar o *fetch* de múltiplos desvios em um único ciclo. Isso porque a *trace cache* colabora para aumentar a quantidade de desvios condicionais que podem ser previstos em um único ciclo, além de melhorar o alinhamento de instruções não contínuas e de reduzir a latência da unidade de busca.

O fato da *trace cache* ser utilizada em processadores modernos, tais como o *Pentium IV*, o *Athlon* e o *Sparc*, demonstra que as contribuições da mesma são realmente efetivas. No caso do *Pentium IV*, pode-se verificar também o benefício da redução na quantidade de decodificação de instruções, devido ao fato da *trace cache* evitar que instruções decodificadas recentemente,

precisem ser re-decodificadas novamente. Essa vantagem fica clara em programas que utilizam laços de repetição (*for*, *while*, *repeat*), nos quais sem a *trace cache*, as instruções executadas precisariam ser decodificadas a cada nova iteração.

Entretanto, apesar dos benefícios e da viabilidade da abordagem de *trace cache*, é preciso levar em conta que a adição de funcionalidades extras sempre torna o processador maior e mais complexo. Logo, a *trace cache* pode adicionar um grande número de transistores a alguma parte do processador que já esteja muito grande. Considerando que o tamanho de um processador impacta diretamente no seu custo de fabricação, a *trace cache* pode torná-lo mais rápido, mas também o tornará mais caro.

Outro ponto que merece ser mencionado é que a colaboração da *trace cache* para a melhoria do desempenho deve-se, em grande parte, à eficiência do previsor de desvios. Um previsor com uma baixa taxa de acertos certamente comprometerá a colaboração da *trace cache* para a melhora do desempenho geral. Apesar dessa dependência, isto não vem sendo apresentado como um fator limitante, talvez, devido a pesquisas e resultados eficientes em abordagens para a previsão de desvios.

5. REFERÊNCIAS

[1] Rotenberg, E., Benett, S. and Smith, J. E. 1996. *Trace Cache*: a Low Latency Approach to High Bandwidth Instruction Fetching. *In the Proceedings of 29th Annual International Symposium on Microarchitecture*. France.

[2] Rotenberg, E., Benett, S. and Smith, J. E. 1999. A *Trace Cache* Microarchitecture and Evaluation. *IEEE Transactions on Computers*. Vol. 48. Nº 2. Pages 111-120.

[3] Berndl, M. and Hendren, L. Dynamic Profiling and *Trace Cache* Generation. 2003. *In Proceedings of the International Symposium on Code Generation and Optimization (CGO'03)*.

[4] Postiff, M., Tyson, G. and Mudge, T. Performance Limits of *Trace Caches*. 1999. *Journal of Instruction-Level Parallelism*. Pages 1-17.

[5] Hennessy, J. L. and Patterson, D. A. *Computer Architecture: A Quantitative Approach*. 2006. 4ª Ed. Elsevier.

[6] Jacobson, Q., Rotenberg, E. and Smith, J. Path-Based Next *Trace* Prediction. 1997. *In Proceedings of 30th Int. Symp. Microarchitecture*. PAFES 14-23.

[7] Yeh, T.-Y., Marr, D. T. and Patt, Y. N. Increasing the instruction fetch rate via multiple branch prediction and a branch address *cache*. 1993. *In Proceedings of 7th Intl. Conf. on Supercomputing*, pp. 67–76.

[8] Dutta, S. and Franklin, M. Control flow prediction with treelike subgraphs for superscalar processors. 1995. *In Proceedings of 28th Intl. Symp. on Microarchitecture*, pp. 258–263.

[9] Ramirez, A., Larriba-Pey, J. and Valero, M. Software *Trace Cache*. 2005. *IEEE Transactions on Computers*, Vol. 54, Nº. 1. Pages 22-35.

[10] Black, B., Rychlik, B. and Shen, J. P. The Block-based *Trace Cache*. 1999. *in Proceedings of the 26th Annual International Symposium on Computer Architecture*.

[11] Hossain, A., Pease, J. D., Burns, J. S. and Parveen, N. *Trace Cache* Performance Parameters. 2002. *In Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'02).*

[12] Ramirez, A., Larriba-Pey, J. L., and Valero, M. 2000. *Trace cache* redundancy: Red and blue *traces*. *In Proceedings of the 6th International Symposium on High-Performance Computer Architecture*. pp. 325-333.

[13] Everything2. Trace Cache. 2002. Disponível em: *http://everything2.com/title/Trace%2520cache*. Acesso em 10 de Junho de 2009.

[14] Hinton, G., Sager, D., Upton, M., Boggs, D., Carmean, D., Kyker, A., and Roussel, P. The microarchitecture of the Pentium 4 processor. 2001. *Intel Technology Journal Q1*.