Influência das Características de Processadores e Aplicações no Nível de Blocos Básicos

Francis B. Moreira, Marco A. Z. Alves, Matthias Diener, Philippe O. A. Navaux
Instituto de Informática
Universidade Federal do Rio Grande do Sul, Porto Alegre, Brasil
{fbmoreira, mazalves, mdiener, navaux}@inf.ufrgs.br

***Resumo*—Este trabalho avalia o desempenho do processador e aplicações no nível de blocos básicos, usando um simulador de microarquitetura para construir o perfil dinâmico do comportamento de cada bloco básico de uma aplicação. Foram analisadas diversas características relacionadas ao desempenho de cada bloco, tais como faltas de dados nas memórias *cache*, falhas nas predições de saltos e contenções nas unidades funcionais. Baseando-se nessa análise, podemos determinar a influência de tais características no desempenho de cada carga de trabalho, e assim, investigar possíveis otimizações para aumentar o paralelismo em nível de instruções no nível arquitetural.**

Os experimentos mostram que as características mais importantes para o desempenho do processador são as faltas de dados no último nível de memória *cache* e falhas nas predições de saltos. A contenção nas unidades funcionais e as faltas de dados nas memórias *cache* L1 e L2 são menos relevantes, uma vez que grande parte destas latências são escondidas pela execução fora de ordem do processador superescalar e pelo mecanismo de *prefetching*.

Quando eliminadas estas duas fontes de degradação do desempenho, através da simulação de um processador livre de faltas de dados no último nível de *cache* ou com um preditor de saltos perfeito, observamos respectivamente melhorias de desempenho máximas de 78% e 127% (em média 10% e 23%).

***Keywords*-Arquitetura de Computadores; Análise de Desempenho; Blocos Básicos;**

I. Introdução

A indústria de processadores de propósito geral continua a aumentar o desempenho de *cores* via paralelismo de instruções[1], vetorização e tecnologia multi-core [2]. Entretanto, a falta de conhecimento do compilador em relação às características arquiteturais, como quantidade de unidades funcionais e registradores, reduz o potencial de otimizações no código. A evidência disto dá-se na grande quantidade de hardware em processadores superescalares modernos, mas baixos ganhos de desempenho em instruções por ciclo (IPC) [3].

Este desperdício de hardware deve-se a dependências de código entre instruções, intrínsecas a programas e necessárias para expressão de algoritmos. A falta de otimização na expressão de tal algoritmo, como acessos à memória desalinhados e comportamento de desvios imprevisível, acarreta na degradação de desempenho do código. Entretanto tais limitações não ocorrem em todos blocos de um código. Alguns exemplos de problemas específicos a blocos com baixo desempenho são acessos a memória delinquentes [4] e desvios imprevisíveis [5].

A ideia de identificar os gargalos de desempenho no nível de blocos de código já foi utilizada em outros trabalhos, onde a maior parte destes usam análise estática[4], [6], [7] para gerar perfis dos programas. Estas ideias são efetivas e interessantes para computação de alto desempenho, porém, em sua aplicação, elas geralmente implicam em adição de novas instruções, o que acarreta em falta de portabilidade, e passos adicionais de compilação ou traços de execução, tornando-as caras ou até mesmo impraticáveis em sistemas de propósito geral.

Entre os mecanismos dinâmicos, a maior parte efetua a análise em nível de instruções. Uma granularidade tão fina implica em custos adicionais para coleta e armazenamento de informações por vezes irrelevantes.

Nesse artigo, é usada uma análise dinâmica de desempenho em nível de blocos. Isto leva a um melhor entendimento das partes de código que possuem representatividade no programa, possibilitando ainda a caracterização de eventuais problemas em tais blocos.

O objetivo deste trabalho é identificar os gargalos de desempenho do hardware superescalar atual, assim como obter estatísticas detalhadas por bloco de código, as quais possam ser utilizadas futuramente por outros mecanismos, como controle de agressividade de um *prefetcher* [8]. Para isto é feita uma análise do comportamento de *branched blocks* com as aplicações da carga de trabalho SPEC-CPU2006.

Para obtenção de características precisas, faz-se necessário um mecanismo com conhecimento do hardware e seu estado atual durante a execução. Neste trabalho é usado um simulador com precisão de ciclos para obtenção dinâmica das estatísticas referentes a cada bloco, habilitando acesso a informações internas com conhecimento de hardware, tal como a contenção em unidades funcionais.

As principais contribuições deste artigo são:

1. Criação e avaliação de uma metodologia para caracterizar informações de desempenho em nível de blocos de código. Esta metodologia é inspirada em trabalhos que utilizam a ferramenta Pin [9], [10], [11]. O objetivo desta metodologia é avaliar o desempenho dos blocos dinamicamente.

2. Avaliação de características arquiteturais atuais. Esta avaliação é feita considerando os principais gargalos encontrados em nossos experimentos. Assim, utilizando uma modelagem de arquitetura perfeita, cada um dos problemas foi analisado separadamente, obtendo assim os ganhos máximos teóricos caso tal gargalo fosse eliminado.

II. Motivação

A definição de bloco básico [6], [7], [12], [13] é de um trecho de código com um único ponto de entrada e um único ponto de saída. Nesta definição desvios incondicionais, tais como chamadas de função, são tratados como pontos de saída.

Considerando que a caracterização em nível de blocos básicos representa uma granularidade fina, optou-se por relaxar a definição de blocos básicos de forma a não considerar os desvios incondicionais. Tal escolha foi feita baseada no fato de que os saltos incondicionais não sinalizam uma repetição direta de um bloco, logo podemos caracterizar trechos de códigos puramente sequenciais dos trechos que correspondem a laços de repetição.

Nesse artigo foi criada a definição de *branched block*, o qual começa com o primeiro endereço após um desvio condicional e termina no primeiro desvio condicional encontrado. A diferença em relação a um bloco básico normal é que o bloco básico termina em qualquer desvio. Usando apenas desvios condicionais, conseguimos uma granularidade mais grossa e agregação de comportamento entre blocos que não precisam ser divididos.

Na Figura 1, pode-se ver uma ilustração do comportamento do programa com saltos condicionais e incondicionais. Cada bloco básico representa um vértice no grafo. Desvios condicionais são representados por linhas pontilhadas e desvios incondicionais por linhas contínuas. Como um desvio condicional pode levar a dois blocos diferentes, com instruções e comportamentos diferentes, misturá-los dificultaria a detecção de seu comportamento predominante. Um desvio incondicional não apresenta tal problema, e pode-se olhar o código que segue como uma continuação da aresta. Adicionalmente, desvios condicionais e incondicionais são tratados diferentemente pelo processador, uma vez que os desvios incondicionais utilizam apenas o *Branch Target Buffer* sem necessidade de predição do desvio, nem o cálculo para decidir sobre o desvio.

Pode-se ver na Figura 1 seis blocos básicos (BBL), representados por retângulos, sendo que estes BBLs formam quatro *branched blocks*. O primeiro *branched block* é composto pelo BBL 1 apenas. O segundo *branched block* é composto pelos BBLs 2, 4 e 6. O terceiro *branched block* é composto pelo BBL 3. E o quarto *branched block* é composto pelos BBLs 5 e 6. O comportamento repetitivo é característico do BBL 3, enquanto os BBLs 2, 4, 5 e 6 são códigos sequenciais de transição que podem ser agregados.

Ao considerar apenas os desvios condicionais como delimitadores de blocos (*branched block*), os blocos obtidos serão basicamente de dois tipos, os blocos de comportamento repetitivo e previsível e os blocos de execução única. Porém, estes blocos de execução única podem se agregar para formar uma repetição de um laço externo maior.

Uma vez definida a granularidade a ser avaliada, a próxima questão relevante diz respeito sobre quais informações coletar. Vários trabalhos que utilizam análise de blocos básicos requerem processamento complexo [4],

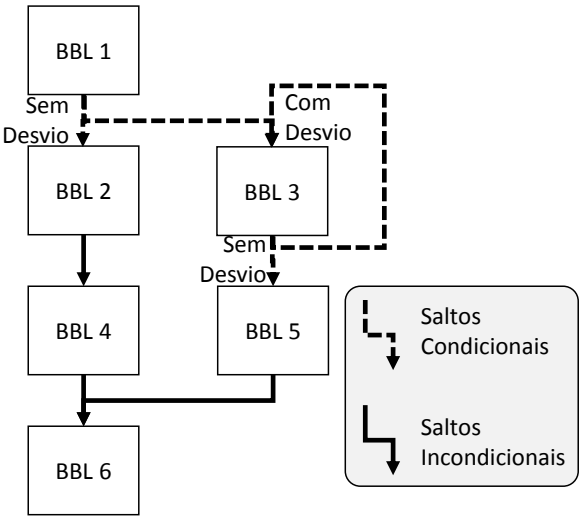

Figura 1. Grafo demonstrando vértices e arestas, e a diferença de um *branched block*. Cada vértice representa um bloco básico, linhas pontilhadas representam um desvio condicional e linhas contínuas representam a simples sequência de código ou um salto incondicional

[6], porém, como neste artigo deseja-se apenas procurar características de cada bloco com relação direta ao seu desempenho, optou-se por eleger apenas a principal característica de cada bloco (mais detalhes serão apresentados na seção III).

Logo, pode-se entender as correlações dos diversos mecanismos da unidade de processamento, e melhorar ou desabilitar tais mecanismos para cada bloco, tal como um *prefetcher* de ponteiros para blocos onde ele não é necessário, ou unidades de ponto flutuante para um programa que usa apenas inteiros. Dessa forma tal análise pode trazer *insights* sobre formas de melhorar o desempenho de futuras arquiteturas, bem como formas de reduzir o consumo de energia desses sistemas sem afetar o desempenho.

A. Caracterização de Blocos de Código

Blocos de código com baixo número de instruções por ciclo afetam a média de desempenho do programa inteiro quando executados frequentemente. Porém, esta mesma frequência de execução gera um padrão de comportamento, e portanto a chance de detectá-lo e melhorar o desempenho do bloco.

Para motivar tal ideia, na Figura 2 é possível ver a relação entre desempenho e as principais características de todos *branched blocks* do programa *libquantum*. Na figura, para cada bloco temos cinco barras mostrando as estatísticas do bloco. As características do código visivelmente afetam o desempenho na maior parte dos blocos. Como exemplo de blocos sem características, podemos observar os blocos 10 e 11. Estes tem desempenho diferente devido à sua composição de instruções diferentes. Não pode-se esperar que um bloco com dependências de código seja executado com o mesmo desempenho que um código completamente livre de dependências.

Nos blocos 2, 8, 13, 14 e 15 pode-se notar apenas a característica de contenção na unidade lógico-aritmética

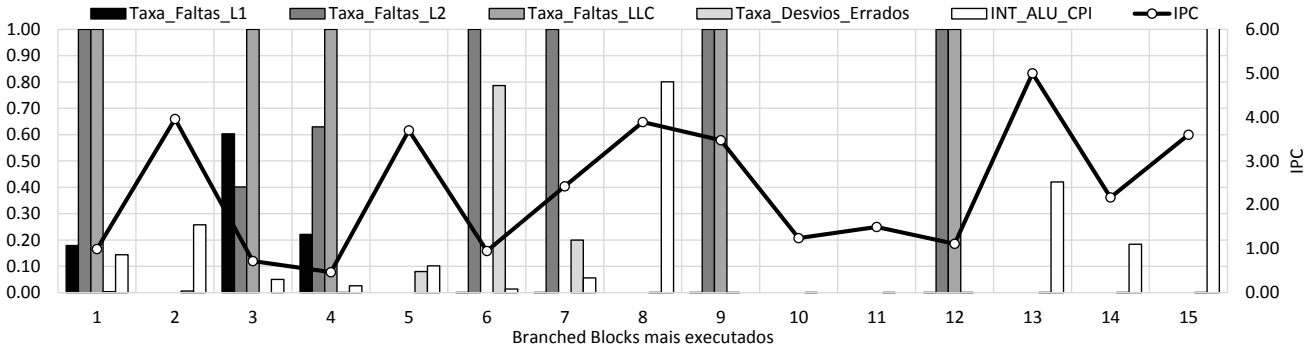

Figura 2. Estatísticas de cada um dos blocos mais relevantes do *libquantum* e sua relação com o desempenho (IPC)

de inteiros. Embora esta característica deveria denotar um problema para o desempenho, pode-se notar que quanto maior a contenção, maior o desempenho. Isto deve-se ao fato de que, nestes blocos, nenhuma outra característica estar presente, e portanto, chegam instruções suficientes às unidades para mantê-las ocupadas durante a execução daquele trecho de código, criando uma relação direta entre contenção e desempenho, que pode ser observada no IPC (*Instructions Per Cycle*) superior a 5 no bloco 13.

Para os blocos 3 e 4 pode-se observar o pior desempenho, devido à alta taxa de falta de dados em todos os níveis de memória *cache*. No bloco 6 o mesmo ocorre devido à alta taxa de erros na predição de desvios.

Caso a arquitetura possuísse um *prefetcher* melhor de tal arquitetura reduzindo as taxas de falta de dados nas caches, poderíamos obter melhora de desempenho significativa para os blocos 1, 3, 4 e 12. Mais características poderiam ser obtidas para obter o comportamento dos blocos, mas estas foram escolhidas empiricamente após experiências com o programa.

B. Desafios

O principal desafio após a escolha de *branched blocks* é o problema que chamamos de *deslocamento de informação*. Na arquitetura superescalar com execução fora de ordem, a graduação de um desvio, estágio final em que o fluxo de instruções já está no caminho correto e a instrução pode ser considerada completada, contém toda informação do que foi processado desde a graduação do desvio anterior (a graduação em ordem garante isto). Porém, como novas instruções não esperam pela graduação do desvio para serem executadas, elas podem entrar no estágio de execução, influenciando alguma estatística (e.g. uma instrução pode gerar uma falta de dados no acesso à memória cache), e tal estatística seria armazenada pertencendo ao bloco que acaba no desvio previamente mencionado. Portanto, o bloco conteria informações do bloco que vem após ele, distorcendo a informação entre blocos.

Isto não seria um problema se todas estatísticas fossem obtidas no mesmo ciclo, ou distorcidas em uma distância fixa. Mas estatísticas como acesso a cache de dados do nível mais próximo ao processador e do nível mais longe levam tempos diferentes para serem registradas, gerando uma distorção variável para cada bloco. Quanto menor a granularidade do programa analisado, maior o efeito desta distorção, e este foi mais um dos motivos que levou à criação de *branched blocks*, cuja granularidade é mais grossa.

Para eliminar este problema, no simulador foi criada uma tabela para armazenar estatísticas referentes a cada desvio condicional (*branched block*). Sempre que uma nova estatística deve ser registrada, é feita uma busca pela primeira instrução de desvio condicional que viria após esta instrução (informação obtida na fila de reordenamento), assim, apenas a entrada daquele desvio condicional é atualizada. Logo, quando tal desvio entrar no estágio de graduação (em-ordem), garantimos que todas as instruções referentes ao *branched block* já foram executadas e graduadas, além de que suas estatísticas foram salvas no lugar correto da tabela de estatísticas.

III. Metodologia

Nesta seção são inicialmente descritos os detalhes do mecanismo de obtenção de estatísticas. Após, são detalhadas características do processador modelado, assim como detalhes adicionais dos experimentos realizados.

A. Detalhes do Mecanismo

As seguintes estatísticas foram escolhidas dada a sua significância e frequência. Falta de dados em todos níveis de cache de dados, erros na predição de desvios e contenção nas unidades funcionais referentes a lógica-aritmética de inteiros, lógica-aritmética de ponto flutuante, multiplicação de ponto flutuante e divisão de ponto flutuante. Demais características como cache de instruções e contenção em outras unidades foram omitidas devido à baixa representatividade ($< 1\%$).

Para garantir estatísticas precisas, cria-se um *buffer* circular que observa todos desvios que residem na fila de reordenamento. Cada entrada no *buffer* contém espaço para armazenar estatísticas variadas. As seguintes mudanças foram feitas no simulador para obter as estatísticas:

1) Sempre que um desvio condicional é inserido na fila de entrada (*fetch buffer*), cria-se uma entrada para esse desvio, usando como índice o *número de operação*. Esse *número de operação* é um identificador único da instrução usado para ordenamento dentro do pipeline.

2) Sempre que uma estatística é registrada, procura-se neste *buffer* circular pelo primeiro desvio cujo *número de operação* seja maior que o número de operação da instrução que está gerando a estatística. Incrementa-se a entrada daquele desvio para estatística referente. Lembrando que todo desvio condicional sinaliza o final de um *branched block*.

3) Quando um desvio entra no estágio de graduação, comparam-se todas estatísticas referentes a ele, multiplicadas pelos seus pesos, e elege-se a mais relevante denotando-a como característica representante do bloco. Ou seja, apenas uma característica representa cada bloco.

B. Detalhes do Processador e Experimentos

Para validar o processo de obtenção de estatísticas, usa-se um simulador de arquitetura x86 com precisão de ciclo desenvolvido pelo grupo de pesquisa, modelando a arquitetura Intel Sandy-Bridge. Toda contenção de unidades funcionais, dependência de registradores e restrições de sistema no processador são simuladas. Na Tabela I são detalhadas as especificações da arquitetura simulada.

Tabela I
PARÂMETROS ARQUITETURAIS SIMULADOS

Processador	2 GHz; 8 cores, Busca e graduação em ordem; 14 estágios 16 B por busca; Decodificação e graduação de 5 instruções; Renomeação/despacho/execução de 5 micro instruções; Buffer busca 18 pos.; Buffer decodificação 28 pos.; 3-alu, 1-mul. e 1-div. unidades de int. (1-3-20 ciclos); 1-alu, 1-mul. e 1-div. unidades de fp. (5-5-20 ciclos); 1-load e 1-store unidades de memória (1-1 ciclos); ROB 168 pos. ; MOB: 64 pos.leitura e 36 pos. escrita;
Preditor de Saltos	1 salto por busca; Executa até 8 saltos sem previsão; BTB de 4 K, 4-vias conj. assoc.; LRU; Preditor de 2 níveis; BHT de 16 K, previsões de 2-bits;
Cache L1 Dados	32 KB, 8-way, 2-ciclos; Linhas de 64 B; LRU; MSHR entradas: 4-request, 6-write-back, 2-prefetch; Stride Prefetch: 2-degree, tabela de 64-strides ;
Cache L1 Instruções	32 KB, 8-way, 2-ciclos; Linhas de 64 B; LRU; MSHR entradas: 4-request, 2-prefetch; Stride Prefetch: grau 2 e tabela de 64-strides;
Cache L2	Privada, 256 KB, 8-way, 4-ciclos; Linha de 64 B; LRU; MSHR entradas: 8-request, 12-write-back, 4-prefetch; Stream Prefetch: grau 2, distância de 16 e 128-streams; Content-Directed Prefetch: grau 1, largura 2;
Cache L3	Compartilhado, 8 MB (8-bancos), 1 MB por banco; 16-way, 10-ciclos; Linha de 64 B; LRU; Inclusiva; protocolo de coerência MOESI; MSHR entradas: 32-request, 32-write-back; Interconexão Anel Bi-direcional;
Controlador DRAM e Barramento	Controlador DRAM On-chip, 4-canais; 8 bancos DRAM por canal, 8 KB row buffer por banco; DDR3, 8 burst length, Frequência em 2:1 (Bus/Core); CAS, RP, RCD e RAS com 9-9-9-28 ciclos de latência;

Avalia-se neste artigo a carga de trabalho SPEC-CPU2006 com entrada tamanho ref. Cada programa foi compilado com gcc 4.7.6, com opções $-O3$ e $-static$. Simula-se cada programa executando as 200 milhões de instruções mais representativas, selecionadas pelo Pin-points [11]. Considerando que as extensões SIMD (SSE e MMX) não são modeladas, espera-se maior contenção para operações de ponto flutuante.

Os pesos das características usados são de 8 para falta de dados na memória cache L1, 32 na memória cache L2, 200 no último nível de memória cache, 16 para erros de predição de desvio, e um para todos os ciclos de contenção de unidades funcionais. Estes valores representam as latências médias que ocorrem em cada evento, portanto, a relevância do evento dentro do bloco.

Para medir o desempenho de um único *core*, é levada em consideração a média de instruções por ciclo do traço de execução inteiro, após *warm-up* de 10 milhões de instruções para evitar os efeitos de uma memória cache fria. Para avaliar a precisão do mecanismo em detectar as características, melhora-se cada característica individualmente a fim de observar se os ganhos de desempenho para cada programa se correlacionam com as características observadas para todos os blocos do mesmo.

IV. RESULTADOS

Para avaliar a metodologia e analisar a influência de cada característica, são realizados os seguintes testes. Primeiramente observa-se a distribuição das características dos blocos de cada programa. Após, simula-se o que ocorreria com cada programa se as latências ligadas a cada característica fossem eliminadas.

A. Análise da Distribuição de Características

A Figura 3 apresenta barras empilhadas representando a distribuição de características para todos *branched blocks* executados no programa. Para obtenção dos dados, foram diferenciadas 8 características e *outros*, ou seja, blocos onde nenhuma das características foi observada. Isto não significa que o bloco não possa ter problemas de desempenho, mas sim que tais problemas não estão sendo registrados entre as características escolhidas (tal qual a contenção nas unidades funcionais de geração de endereço de memória).

A primeira observação é a contenção presente em toda carga de trabalho para as unidades lógico-aritméticas de inteiros. O processador simula 3 dessas unidades, e a maior parte das instruções comuns é executada nelas, portanto, a contenção ocorre em todo programa.

Também podemos ver que a cache L2 não é um problema grave em nenhum programa devido ao *prefetcher* híbrido agressivo. Os erros de predição de desvios estão presentes em todos programas, mas como estamos limitados a um desvio por bloco, em geral podemos observar que a taxa de erro na predição de desvios é menor do que 10%. A única exceção é o programa *gobmk*. Por final, a maior parte dos blocos é caracterizada como *outros*, mostrando que o desempenho do código nestes blocos é provavelmente limitado por dependências de código.

B. Validação por Grupos de Características

Nas figuras seguintes os resultados são mostrados comparando o IPC para as execuções com arquitetura perfeita em relação ao modelo base. Apenas uma característica é tratada como perfeita em cada teste. Porém, como um bloco pode sofrer de diversos problemas, ao eliminar uma fonte de problema a segunda fonte pode ainda influenciar no desempenho do bloco, evitando o aumento de IPC.

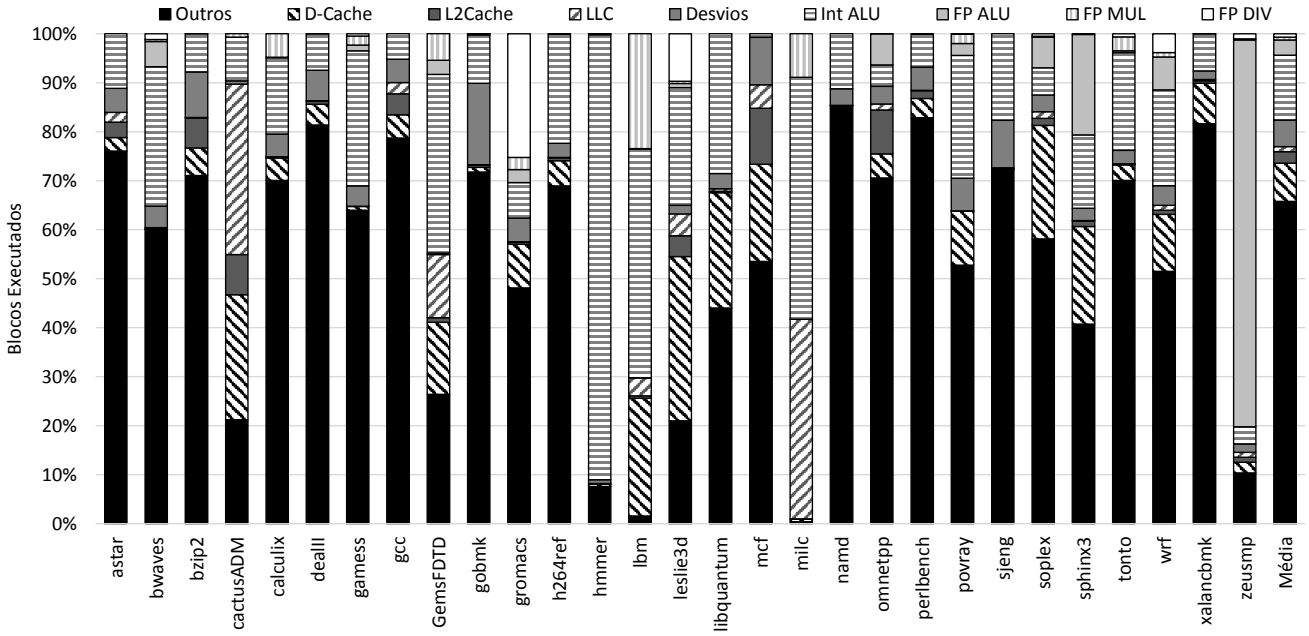

Figura 3. Estatísticas obtidas para o conjunto de programas SPEC-CPU2006. Observe que embora os blocos se repitam, o gráfico apresenta todas as repetições sem agregações por bloco.

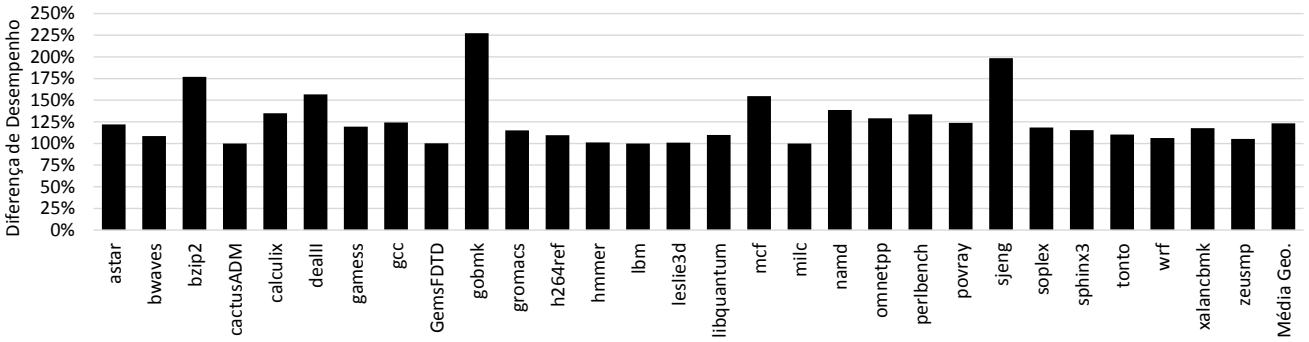

Figura 4. Ganhos de IPC comparando um preditor de desvios perfeito em relação ao processador base

Para eliminar erros na predição de desvios, um preditor perfeito é modelado. Para eliminar faltas de dados nas caches, são simuladas caches de 1 GB, e também, caches que nunca erram. Para remover contenção nas unidades funcionais, a configuração do processador a ser modelado é modificada para simular 168 unidades funcionais daquele tipo avaliado, assim como largura de todos estágios necessários para permitir a execução das instruções em paralelo.

Na Figura 4, pode-se observar que os ganhos de desempenho de um preditor de desvios perfeito correlacionam-se bem com os programas que tinham número grande de erros na predição de desvios. Os programas *gobmk* e *sjeng* tiveram seu IPC dobrado, com ganhos de 127% e 98%, já que eram os que apresentavam maiores problemas. Programas que não mostraram problemas com desvios, tal qual *GemsFDTD*, *hmmer* e *lbm* não obtiveram ganhos. Em média, o ganho de desempenho foi de 21,55%.

O comportamento não é consistente em todos programas. Isto pode ser notado comparando os programas *omnetpp* e *wrf*. Onde o primeiro tem uma característica dominante de erro na predição de desvios em apenas 3% dos blocos, com melhora de IPC de 29%, o segundo melhora apenas 6%, sendo que a característica é dominante em mais de 4% dos blocos. O motivo desta inconsistência é que enquanto os blocos do *wrf* são caracterizados como erro na predição de desvio, isto não significa que outros problemas não ocorram no bloco. Simplesmente significa que o erro na predição era provavelmente o maior problema, mas as dependências de código ainda podem parar o pipeline e manter o desempenho baixo.

Na Figura 5 as barras demonstram o resultado do uso de uma cache de 1 GB para cada nível, porém, mantendo a mesma latência de acesso. Como nenhum programa usa mais que 1 GB de memória, esse experimento demonstra os benefícios próximos ao máximo a ser obtido com aumento nas caches. Porém, os ganhos de desempenho não são convincentes, e faltas de dado continuaram apresentando-se como característica nos blocos devido às faltas de dado compulsórias. Os ganhos foram de até 17% e 11% com os programas *lbm* e *gcc*, com média de ganhos de 3,30% para uma memória cache L1 de 1 GB.

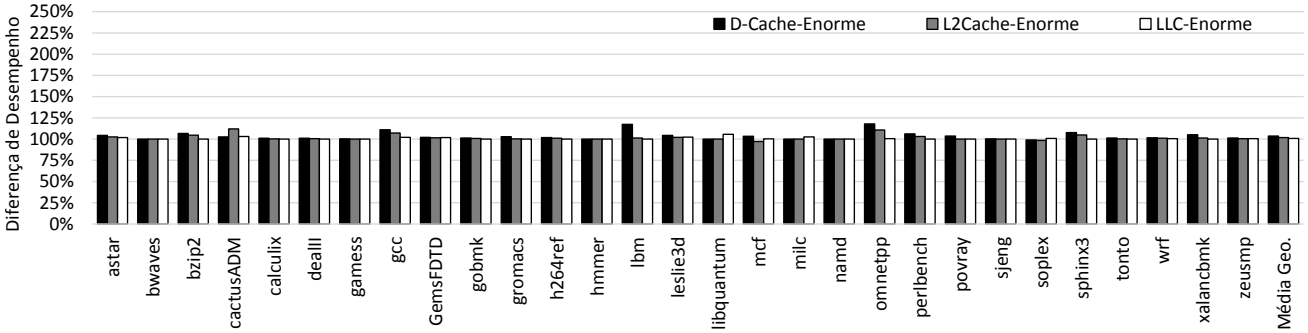

Figura 5. Desempenho dado 1 Gigabyte de tamanho para os diferentes níveis de cache

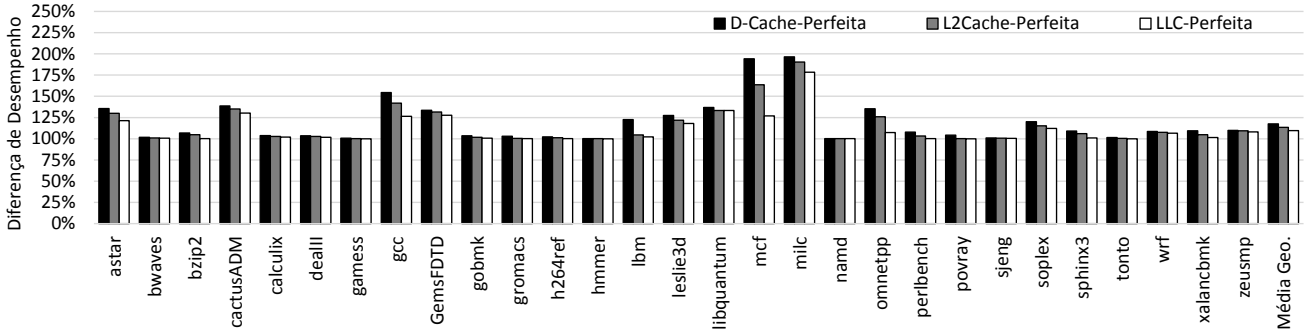

Figura 6. Desempenho considerando 100% de taxa de acerto para todos níveis de cache

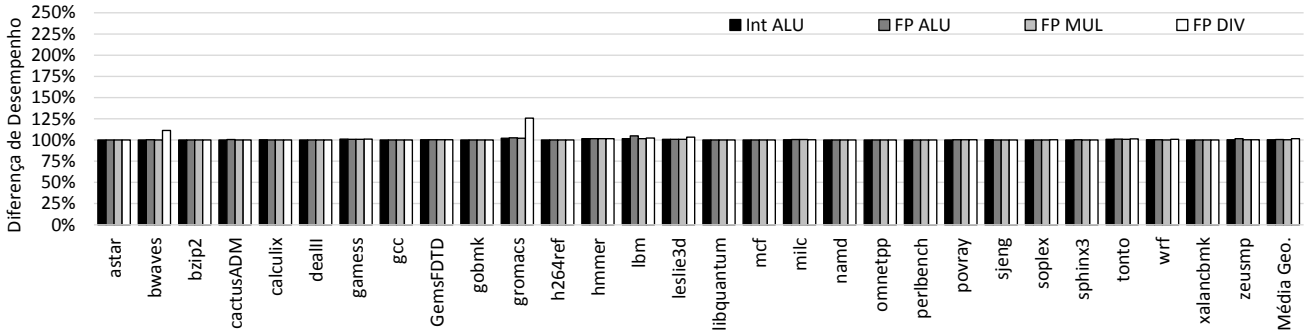

Figura 7. Desempenho dadas unidades funcionais infinitas para cada tipo de unidade funcional

Na Figura 6, pode-se observar os ganhos eliminando-se faltas de dados nos diferentes níveis de cache, como se estas memórias sempre pudessem fornecer o dado requisitado. Este experimento mostra os ganhos máximos teóricos caso a arquitetura usasse um *prefetcher* perfeito que fosse capaz de buscar os dados antes de serem utilizados, porém, sem remover dados úteis da memória *cache*. Cada barra mostra o desempenho para cada nível. A melhora de desempenho em todos níveis correlaciona-se fortemente com a falta de dados no último nível de cache. Isso significa que o acesso a memória principal impõe uma latência que o processador não consegue mascarar com a execução de outras instruções em paralelo.

Os ganhos foram de até 96%, 90% e 78% de desempenho para os três níveis de memória cache do programa *milc*, e de 94%, 64% e 27% para o programa *mcf*. Podemos observar que o programa *milc* consegue esconder as latências de memórias cache muito melhor que o programa *mcf*. Em média, os ganhos foram de 18%, 14% e 10% para os diferentes níveis de cache perfeitas.

Existem aqui também alguns contra-exemplos. O primeiro mostra-se no programa *libquantum* não demonstrar muitos blocos sofrendo da característica de falta de dados no último nível de cache, porém gerando grandes melhorias de IPC com caches perfeitas. Isto deve-se provavelmente à contenção nas unidades funcionais de memória, que ao conseguirem tratamento rápido para requisições, não sofrem tal contenção ou efeitos de espera devido à dependências. E segundo, o programa *gcc* também gera ganhos desproporcionais de desempenho, de 54%, quando comparado aos outros dado sua pequena quantidade de blocos caracterizados como falta de dados nas caches.

Por último, examina-se o resultado ao eliminar a contenção nas unidades funcionais. Foram escolhidos três tipos de unidades funcionais, já que as outras apareciam em menos do que 1% dos blocos. As simulações foram rodadas agora com 168 unidades de cada tipo, de cada vez, para garantir nenhuma contenção. Esse número de

unidades funcionais foi escolhido considerando o tamanho do *Reorder Buffer* simulado.

Na Figura 7, embora a contenção da unidade lógico-aritmética seja a característica mais frequente de toda a carga de trabalho, resolvê-la gera pouco ganho de desempenho. Uma vez que o processador consegue executar rapidamente mais instruções as dependências verdadeiras entre instruções interrompe os ganhos de desempenho. Em média, não houve mudança significativa no desempenho.

Na mesma figura apresentam-se os resultados para a unidade lógico-aritmética de ponto flutuante onde o programa *lbm* melhora 5%. Para a multiplicação de ponto flutuante, mesmo para o programa *lbm*, que possui a maior taxa de blocos com tal característica, nenhuma diferença é visível. Em média não ocorre mudança no desempenho de multiplicações. Para a divisão de ponto flutuante, os únicos programas a melhorar são o *gromacs* e o *bwaves*, em 26% e 11%, devido à alta latência da unidade e sua forte presença nos programas. Curiosamente, o programa *bwaves* tem uma porção relativamente baixa de problemas com unidades de ponto flutuante, o que significa que a unidade pode se tornar um gargalo importante no sistema. Porém, a média de toda carga de trabalho indica apenas 2% de melhoria no desempenho, contra-indicando o uso de mais unidades de divisão de ponto flutuante.

Em geral, a contenção nas unidades é facilmente escondida pelo paralelismo do processador superescalar. Os experimentos de contenção foram repetidos usando um preditor de desvios perfeito e uma cache de nível 1 perfeita para verificar se a melhora na contenção não estava sendo prejudicada por estes outros fatores. O ganho no desempenho ainda assim foi insignificante.

V. Trabalhos Correlatos

Este trabalho é inspirado no conhecimento e experiência acumulado na escrita do simulador usado. Na identificação de blocos básicos gerados pelo Pin como código característico para simular o programa inteiro, deduziu-se que, se existisse conhecimento a priori de como cada bloco básico se comportava, poderia-se melhorar seu desempenho.

Em Panait *et al.* [4], os autores classificam instruções de acesso a memória estaticamente, de acordo com várias heurísticas, como as operações usadas para cálculo de endereço do dado, registradores usados no cálculo e frequência de execução. Define-se a *delinquência* de um acesso à memória, caracterizando instruções que são responsáveis pela maioria das faltas de dados nas caches durante a execução. Com análise estática os autores apontam apenas 10% do número total de instruções de acesso a memória como responsáveis por mais de 90% das faltas de dado na cache nível 1. Criando perfis de blocos básicos junto ao compilador, eles reduzem estes números para 1.3% de instruções responsáveis por 82% de todas faltas de dados na cache de nível 1. Este trabalho portanto mostra quão eficiente e necessária é a idéia de analisar características do código a nível de bloco.

Sherwood *et al.* [14] é o trabalho precursor ao SimPoints [10] e Pinpoints [11]. Nele, os autores caracterizam o comportamento de programas inteiros baseados na análise de distribuição de blocos básicos. Eles criam o conceito de vetores de blocos básicos (BBV) para caracterizar um programa, e escolhem fatias do programa para representarem todo o programa dada a similaridade entre os BBVs do programa e das fatias. Portanto, eles conseguem identificar partes do programa com estas diferenças, como a fase de inicialização, e o verdadeiro código que ocupa maior parte do tempo do processador.

Usa-se Simpoints em nossa metodologia, mas a diferença neste artigo é que ao invés de usarmos apenas os blocos para gerarmos porções representativas, analisamos exatamente qual o problema de cada porção representativa de cada programa, e como caracterizá-las de modo significante. Este trabalho correlaciona-se pois a agregação de característica feita em nosso artigo torna explícito o que a similaridade entre vetores de blocos básicos torna implícita na comparação direta: o que acontece na organização da arquitetura dada a execução de um bloco.

O recente trabalho de Kambadur *et al.* [7] usa um método de análise simples que eles criam, chamado de *Parallel Basic Block Vectors*. Cada entrada no vetor contém um histograma de quantas *threads* estavam executando para cada execução do bloco. Isto permite aos autores verificar quais blocos executam em que níveis de paralelismo de *threads*, claramente identificando blocos sequenciais e paralelos. Também identificam assim quais são as regiões mais críticas em termos de desempenho, ao analisar que blocos necessitam ser executados sequencialmente. O trabalho então ilustra vários cenários onde tal análise pode ser útil, tal como particionamento de aplicações paralelas e sequenciais, características do programa por grau de paralelismo e análise de *hotspots* de paralelismo.

Tal trabalho pode ser visto como caracterização de paralelismo em nível de *thread* (TLP), enquanto neste artigo almeja-se caracterizar um bloco dado o seu paralelismo de instruções (ILP), agregar sua informação e então caracterizá-los dados seus principais problemas.

A vantagem do método descrito neste artigo é de que, como as estatísticas são modeladas em uma simulação, têm-se conhecimento detalhado de hardware para obtenção de qualquer estatística detalhada no momento exato que for necessário. Assim consegue-se evitar distorções em relação às características dos blocos devido à habilidade de controlar o seu registro, o que leva análises estáticas sobre processadores superescalares à imprecisão.

VI. Conclusões

Com os resultados da análise na carga de trabalho SPEC-CPU2006, extraem-se as seguintes conclusões. Desvios e acessos a cache ainda devem ser os principais alvos de pesquisa, pois demonstraram-se muito mais relevantes e limitantes ao desempenho do processador. A contenção nas unidades funcionais tem pouca relevância, pois o processador superescalar fora de ordem consegue mascarar tais latências ou o ganho de desempenho é estagnado por dependências verdadeiras existentes no código. Assim, mesmo um número pequeno de unidades funcionais está

apto a atingir níveis próximo de desempenho com uma relação de custo-benefício bem favorável. A única unidade que foi uma exceção é a unidade de divisão de ponto flutuante, devido à sua enorme latência, onde foi possível no programa *gromacs* obter um ganho de desempenho de 26% ao se eliminar a contenção.

Adicionalmente, observou-se que entre os níveis de cache, para maioria dos programas, o processador superescalar com execução fora de ordem consegue efetivamente esconder a latência de acesso à cache de nível 1, e tolerar uma falta de dados e consequente acesso à cache privada de nível 2. Os resultados mostram que o maior detrimento ao desempenho são os acessos a memória principal. Além disso, considerando os atuais *prefetchers* agressivos, a maior parte das faltas de dados na arquitetura avaliada são compulsórias. Podemos deduzir isto das médias geométricas de ganhos de desempenho para as simulações de validação de caches perfeitas para cada nível, onde obtivemos ganhos médios de 18%, 14 % e 10%. Assim, nota-se que o ganho mais significativo é no último nível ao evitar a latência da memória principal, obtendo ganhos de até 78%.

A característica com maior presença nos testes foi a contenção nas unidades lógico-aritméticas de inteiros, embora a influência de tal contenção tenha sido insignificante. A característica com maior influência em toda carga de trabalho foi a de erros na predição de desvios, com média de ganhos de desempenho de 23%, chegando até um máximo de 127% de ganhos no programa *gobmk*. Embora a predição de desvios seja provavelmente uma das características mais otimizadas e trabalhadas nos processadores atuais, ela continua sendo a mais relevante em impacto no desempenho, pois não é usada nenhuma técnica para mascarar a latência de um erro na predição.

Para evitar distorções presentes em trabalhos anteriores, são coletadas estatísticas durante a execução em um simulador. Agregam-se então as informações das estatísticas de cada bloco na característica mais relevante representando o bloco. Tal técnica é útil, por exemplo, ao concentrar os acessos de memória cache nos blocos corretos ao invés de distribuí-los pelos blocos do programa.

Como trabalho futuro, almeja-se coletar estatísticas relacionadas diretamente às dependências verdadeiras, tal qual quantos ciclos uma instrução teve que esperar pelo término de outra, e o que causou a demora desta outra instrução.

AGRADECIMENTOS

Os autores gostariam de agradecer ao Laércio Lima Pilla pela revisão, e aos professores Luigi Carro e Sergio Bampi da Universidade Federal do Rio Grande do Sul por suas valiosas sugestões. Este trabalho foi parcialmente financiado pela FAPERGS e CNPq.

REFERÊNCIAS

[1] S. Jarp, A. Lazzaro, J. Leduc, and A. Nowak, "Evaluation of the Intel Sandy Bridge-EP server processor," *CERN openlab: Switzerland, March*, 2012.

[2] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*. IEEE, 2011, pp. 365–376.

[3] D. W. Wall, "Wrl research report 93/6," DEC Western Research Laborator, Tech. Rep., 1993.

[4] V.-M. Panait, A. Sasturkar, and W.-F. Wong, "Static identification of delinquent loads," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, 2004, pp. 303–314.

[5] H. Gao, Y. Ma, M. Dimitrov, and H. Zhou, "Address-branch correlation: A novel locality for long-latency hard-to-predict branches," in *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, 2008, pp. 74–85.

[6] D. Ansaloni, S. Kell, Y. Zheng, L. Bulej, W. Binder, and P. Tuma, "Enabling modularity and re-use in dynamic program analysis tools for the java virtual machine," in *ECOOP 2013–Object-Oriented Programming*. Springer, 2013, pp. 352–377.

[7] M. Kambadur, K. Tang, and M. A. Kim, "Harmony: collection and analysis of parallel block vectors," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 452–463. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337159.2337211

[8] S. Srinath, O. Mutlu, H. Kim, and Y. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, 2007, pp. 63–74.

[9] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065034

[10] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program phase analysis," *Journal of Instruction Level Parallelism*, vol. 7, no. 4, pp. 1–28, 2005.

[11] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large intel itanium programs with dynamic instrumentation," in *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, 2004, pp. 81–92.

[12] J. Cocke, "Global common subexpression elimination," *SIGPLAN Not.*, vol. 5, no. 7, pp. 20–24, Jul. 1970. [Online]. Available: http://doi.acm.org/10.1145/390013.808480

[13] J. Huang and D. Lilja, "Extending value reuse to basic blocks with compiler support," *Computers, IEEE Transactions on*, vol. 49, no. 4, pp. 331–347, 2000.

[14] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," in *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*, 2001, pp. 3–14.