

Comparação do número de acessos à memória e TLB *misses* com páginas de 4KB ou 4MB

Gustavo Ciotto Pinton¹

¹Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Av. Albert Einstein, 1251, Cidade Universitária, Campinas/SP
Brasil, CEP 13083-852, Fone: [19] 3521-5838

ral17136@unicamp.br

Abstract. *This report describes the number of TLB misses for data and instruction, in addition to memory and page table accesses executed by the SPEC CPU2006 benchmarks with the largest memory footprints. Such measures were achieved thanks to the implementation of a new pintool, capable of modelling data and instruction TLB caches with 4KB and 4MB memory pages. In this report, we have considered L1, L2 and L3 caches with different size in order to compare the total memory table accesses with total memory accesses. Besides, in order to get test results faster, we used Intel's Program Record/Replay Toolkit (pinplay) with SPEC CPU2006 pinballs downloaded from the Internet corresponding to the `ref` input set. Finally, we wrote a toy benchmark to test memory access and allocation exclusively.*

Resumo. *Este relatório apresenta o número de acessos à memória, à tabela de páginas e TLB misses para os benchmarks do SPEC CPU2006 com maiores memory footprints. Tais medidas foram realizadas através da implementação de uma nova pintool, capaz de reproduzir o modelo de TLB de instruções e dados com páginas de 4KB e 4MB. Neste relatório, foram considerados os níveis L1, L2 e L3 de cache a fim de compararmos o efeito dos acessos à tabela de páginas em relação aos acessos totais à memória. Além disso, a fim de agilizar os testes, fez-se uso da ferramenta pinplay, também desenvolvida pela Intel, para a execução da respectiva pintool. Os pinballs foram retirados da Internet e correspondem ao testes cujas entradas correspondem ao conjunto `ref`. Por fim, implementou-se um benchmark adicional com intuito de testar exclusivamente a alocação e acesso da memória.*

1. Introdução

O espaço de endereço *físico* é definido como o intervalo de endereços que podem ser utilizados para indexar efetivamente a memória física de um dispositivo e que podem ser gerados pelo processador no seu barramento. O espaço de endereço *virtual*, por sua vez, é o intervalo de endereços que uma aplicação pode utilizar. Cabe ao sistema operacional gerenciar o mapeamento entre estes dois *tipos* de memória. A virtualização de um programa oferece duas vantagens principais. A primeira corresponde ao fato de que programas não precisam ser alterados quando executados em máquinas com diferentes tamanhos de memória física. Em segundo, a virtualização garante a proteção do espaço de memória de cada aplicação, à medida que isola seus espaços de memória virtual através de alguns *bits* adicionais.

O espaço de endereços, tanto o virtual como o físico, é dividido em *páginas*, que neste relatório, terão tamanho de 4KB ou 4MB. Uma página pode ser armazenada tanto na memória principal quanto em disco, caso a primeira esteja já totalmente ocupada. O sistema operacional realiza toda a gerência da memória física, cabendo a ele alocar ou desalocar (inclusive, em que

endereço) páginas e atribuir endereços físicos aos virtuais utilizados pelas diversas aplicações. Esse mapeamento é realizado através de uma estrutura chamada de *tabela de páginas*, que é mantida em um endereço especial da memória principal.

É importante lembrar que duas ou mais páginas virtuais podem ser mapeadas para uma mesma página física da memória. Essa técnica, chamada de *virtual addressing*, é utilizada em aplicações que compartilham memória entre si, como, por exemplo, em bibliotecas tais como a *glibc* e requer mecanismos especiais de atualização de *caches* ou *buffers*, uma vez que mudanças realizadas em uma das aplicações devem ser refletidas nos demais.

Uma possível implementação da tabela de páginas, visando a economia de memória, é através de um sistema multi-níveis ou hierárquico. Neste caso, o endereço que deve ser traduzido é dividido em $N + 1$ seções, em que N representa o número de níveis escolhido. Cada uma das seções possui o índice da tabela ocupado pelo endereço naquele respectivo nível, sendo que uma tabela é composta por ponteiros para tabelas do próximo nível. Por exemplo, se em uma máquina de 32 *bits*, uma página tivesse 4KB e cada entrada ocupasse 4 *bytes*, e se houvesse apenas um nível, a tabela completa ocuparia $4 * 2^{10} * 4 = 2^{14}$ *bytes* na memória. Desta maneira, para acessar apenas um dado, seriam necessários 2¹⁴ *bytes*. Por outro lado, se 2 níveis fossem utilizados e se 10 *bits* fossem usados para indexar cada um dos dois níveis, teríamos $4 * 2^{10} = 4\text{KB}$ para o primeiro nível e outros 4KB para o segundo. Assim, neste caso, precisaríamos de apenas 8KB, representando uma economia de $\frac{2^{14}}{8 * 2^{10}} = 2$ vezes. Para a implementação da *pintool*, assumiu-se que essa hierarquia possuía 3 níveis.

Durante a execução de um programa, muitas operações de leitura (de instruções e dados) e escrita na memória são requisitadas. Para cada uma destas operações, o mapeamento teria que ser obtido e diversos outros acessos adicionais à memória seriam realizados (para um tabela de páginas com 3 níveis, por exemplo, seriam necessários 3 outros acessos), resultando, assim, na degradação da sua performance. Para evitar este cenário, um cache da tabela de página é adicionado aos processadores e é chamado de TLB, do inglês *Translation Lookaside Buffer*. Tal cache recebe como entrada o número da página virtual e devolve, em caso de sucesso, o respectivo número da página física e algumas informações adicionais, como permissão de escrita e processo que a detém. Para a nossa aplicação, modelamos dois caches TLB completamente associativos, um para as instruções e outro para os dados, com 512 entradas cada. Além disso, modelos três níveis de cache, sendo eles L1, L2 e L3. O primeiro nível contém dois caches dedicados para instruções e dados, separadamente, com 64KB cada e 32-way associative. O segundo nível, L2, é unificado para instruções e dados, possuindo 4MB e implementando a política de alocação *directed mapped*. Por fim, o cache L3, também unificado e *direct mapped*, possui 32MB.

Neste relatório, serão avaliados dois modelos com páginas de 4KB e 4MB. Para tal, uma nova *pin tool* foi escrita com base nas implementações de cache disponíveis no próprio diretório do *pin*. Os testes foram realizados nos *benchmarks* com maior *memory footprint* através da ferramenta *pinplay*, cujos *pinballs* foram retirados da Internet. Foram escolhidos os *pinballs* com 100M de instruções executadas como *warmup*, regiões de 30M instruções e máximo do parâmetro K do algoritmo de clusterização igual a 10 [SniperSim 2008].

2. Implementação da *pin tool*

O arquivo `pin.cache.H`, disponível no diretório de instalação do *pin*, fornece algumas classes que modelam o funcionamento de um cache. As classes `ROUND_ROBIN` e `DIRECT_MAPPED` modelam e implementam duas políticas de substituição. A primeira consiste na implementação de uma lista circular através de um vetor, em que elementos são inseridos fim da lista e retirados

do começo. Neste caso, diferentemente da política LRU, não há quaisquer registros sobre a última utilização de uma entrada: tende-se a acreditar que elementos mais próximos do início da fila foram usados há mais tempo e, por isso, a probabilidade de serem novamente acessados é menor. A política `DIRECT_MAPPED`, por sua vez, é trivial, já que representa um conjunto com apenas um elemento (associatividade igual a 1). A estrutura do cache é representada pela classe `CACHE_BASE`, que fornece uma série de métodos permitindo o controle das estatísticas dos acessos ao cache. Tal classe é estendida por `CACHE`, que acrescenta métodos de modificação do conteúdo armazenado. Os dois principais métodos são `AccessSingleLine` e `Access`, responsáveis por acessar uma ou múltiplas linhas do cache, respectivamente, retornando *true* em caso de *hit* e *false*, caso contrário. Estes dois métodos também se encarregam da substituição, em caso de *miss*, de uma das entradas pelo novo elemento.

A *pintool* instancia 7 caches distintos: 2 para páginas de 4KB, 2 para páginas de 4MB e três para os níveis L1, L2 e L3. Não há a necessidade de criarmos instâncias distintas para os 3 últimos, visto que os endereços procurados são os mesmos, tanto no caso de páginas de 4KB como no de 4MB. Cada um dos pares de TLB é constituído por um TLB para os dados e outro para as instruções. Todos eles são completamente associativos, utilizam a política de substituição *round robin* e possuem 512 entradas. O primeiro nível L1 contém dois caches dedicados para instruções e dados, separadamente, com 64KB cada e 32-way associativo. O segundo nível, L2, é unificado para instruções e dados, possuindo 4MB e implementando a política de alocação *directed mapped*. Por fim, o cache L3, também unificado e *direct mapped*, possui 32MB. As rotinas de instrumentação são colocadas a cada *trace* através da chamada da função `TRACE_AddInstrumentFunction()` e as de análise, antes de cada bloco funcional (também chamado de *basic block*) ou instrução, dependendo da sua natureza. Dentro da função de instrumentação, itera-se sobre todos os *basic blocks* a partir das funções `TRACE_BblHead()` e `BBL_Next()` e, para cada um deles, uma função de análise é registrada, cujo propósito é verificar o número de acessos à memória referente à fase de **instruction fetch**. Nesta etapa, o processador busca na memória as instruções a serem executadas. Esta função de análise recebe, além do endereço inicial do bloco, o seu tamanho e realiza a chamada dos métodos `Access()` de cada um dos TLBs de instrução. Ainda na função de instrumentação, itera-se também sobre as instruções de cada *basic block* em busca de operações de *load* ou *store*. Caso uma delas seja encontrada, então uma função de análise é registrada, recebendo como parâmetro o endereço de escrita ou leitura. A finalidade desta última é chamar um dos dois métodos de acesso (`AccessSingleLine()` ou `Access()`) dos TLBs de dados, dependendo do tamanho dos operadores.

Diferentemente do projeto 1, a degradação no desempenho dos *benchmarks* neste caso é proibitivo, exigindo um tempo muito elevado para as execuções do tipo *ref*. Uma ferramenta proposta para minimizar este problema é o *Program Record/Replay Toolkit*, que permite salvar ou carregar estados inteiros de uma execução, chamados também de *pinballs*. Aliado a este *toolkit*, pesquisadores da faculdade da Califórnia desenvolveram uma ferramenta capaz de detectar as regiões mais representativas, chamadas de *SimPoints*, a partir de análise de fases em execuções de grandes programas. Eles mostraram que a execução de apenas estas regiões é capaz de revelar com uma precisão muito grande o desempenho real do programa. Ainda com base nestes conceitos, os pesquisadores da Intel estenderam a ideia para criar *PinPoints*, isto é, regiões representativas de análise compatíveis com o *pin*. Enfim, pesquisadores da Universidade de Ghent disponibilizaram os *pinballs* relativos às execuções dos *benchmarks* do SPEC CPU2006 para as entradas do conjunto *ref*. A seção *Resultados*, a seguir, representa, portanto, os resultados encontrados executando-se o *Program Record/Replay Toolkit* com tais *pinballs*.

3. Implementação de um *toy benchmark*

A fim de testarmos exclusivamente o desempenho da memória, um *benchmark* foi escrito. De maneira geral, este *benchmark* aloca vetores de inteiros e pontos flutuantes e realiza acessos aleatórios a eles. Espera-se que os acessos aleatórios produzam muitas *cache misses*, forçando, assim, mais acessos à memória.

A fim de aumentar a quantidade de memória utilizada, o programa aloca dois vetores, um de ponteiros para inteiro (`int*`) e outro de ponteiros para ponto flutuante de precisão dupla (`double*`). Para cada um dos ponteiros presentes nestes vetores, são alocados novos vetores. Neste relatório, chamamos de primeiro nível a alocação de vetores dos tipos `int*` ou `double*` e de segundo nível a alocação de vetores dos tipos primitivos `int` ou `double`. É necessário observar que a alocação sempre ocupa páginas inteiras de memória, isto é, por exemplo, o vetor `int*` de primeiro nível terá `NUMBER_OF_PAGES_LEVEL_1 * PAGE_SIZE / sizeof(int*)`. As constantes abaixo são definidas, a fim de facilitar o entendimento:

- `PAGE_SIZE`: tamanho de cada página de memória. Definida como 4KB.
- `NUMBER_OF_PAGES_LEVEL_1`: número de páginas ocupadas no primeiro nível para cada vetor. Por padrão, 1024.
- `NUMBER_OF_PAGES_LEVEL_2`: número de páginas ocupadas no segundo nível para cada vetor. Por padrão, 128.
- `RANDOM_ACCESSES`: número de acessos aleatórios aos vetores. Definida inicialmente como 15000.

Com o teste deste *benchmark*, espera-se um desempenho muito ruim da TLB de 4KB nos acessos aleatórios, já que produzirão muitos *TLB misses*, mas, ao contrário, bom com o de 4MB, visto que 1024 páginas de 4KB podem ser contidas dentro de uma de 4MB. Observa-se que são necessárias apenas duas páginas de 4MB para conter os dois níveis, sendo assim, poucos *misses* de dados são esperados. Além disso, considerando que o código não é extenso, espera-se também que não haja muito *misses* no cache de instruções.

4. Resultados

Esta seção visa a análise dos resultados obtidos tanto para os *benchmarks* selecionados do SPEC, quanto para o *toy benchmark* criado.

4.1. Resultados para os *benchmarks* do SPEC2006

Visamos escolher os 10 *benchmarks* com maiores *footprint* e, para tal, baseamos-nos em [Henning 2007]. Este trabalho apresenta os atributos *rss*, *vsz* dos *benchmarks*, assim como as suas estabilidades. O primeiro atributo, *rss*, é uma abreviação para *resident set size* e representa a memória física alocada, enquanto que a segundo, *vsz* ou *virtual size*, indica o tamanho total e memória virtual utilizada. Enfim, um *benchmark* é considerado estável se o consumo de memória cresce rapidamente aos valores medidos e se mantém neste nível. Observa-se que a maioria dos *benchmarks* do conjunto inteiro não são estáveis, contrariamente aos do conjunto de ponto flutuante, e, de maneira geral, os *benchmarks* deste último consomem mais memória (dos dois *tipos*, inclusive).

Os caches L1, L2 e L3 apresentam os mesmos resultados para *misses* e *hits* considerando os dois tamanhos de páginas. Isso ocorre devido ao fato de que os endereços procurados são os mesmos nos dois casos e, portanto, em termos de acessos a tais caches não há alguma diferença. Por outro lado, os resultados das conversões realizadas pelas TLBs sofrem modificações, visto que, com páginas de tamanhos distintos, a página física alocada para determinado dado ou

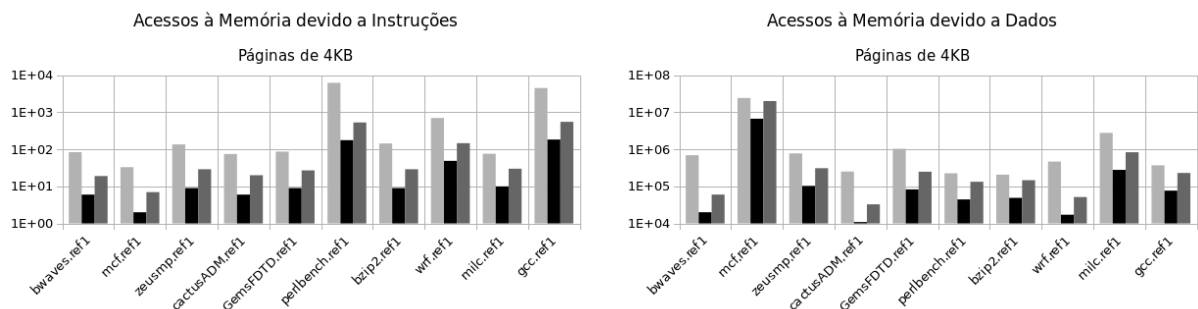
instrução pode mudar. Para páginas maiores, por exemplo, a chance que dados acessados dentro de intervalos de tempo muito curtos estejam na mesma página é maior que para páginas de tamanhos pequenos. Por efeitos de simplicidade, a tabela 1 representa apenas os dados referentes ao último nível L3, obtidos para as execuções:

Tabela 1. Misses e Hits obtidos no nível L3.

SPECint2006			SPECfp2006		
Benchmark	Hits	Misses	Benchmark	Hits	Misses
perlbench	36644	61464	bwaves	41399	594571
bzip2	20808	40065	zeusmp	76187	392976
gcc	71744	72212	cactusADM	1248	217534
mcf	2174881	1981846	GemsFDTD	70754	715526
			wrf	97695	318550
			milc	196654	1752083

Cada *miss* na tabela acima corresponde ao um acesso à memória. As figuras 1(a) e 1(b) representam o total de acessos devido a instruções e a dados para páginas de 4KB, respectivamente. As figuras 2(a) e 2(b), por sua vez, representam os mesmos números, só que para páginas de 4MB. Para cada *benchmark*, a barra na extremidade esquerda representa o total número de acessos, a do meio, o número de *misses* na TLB e, a da direita, o número de acessos à tabela de páginas ocasionados por ela. Todos os gráficos estão em escala logarítmica. Observa-se que:

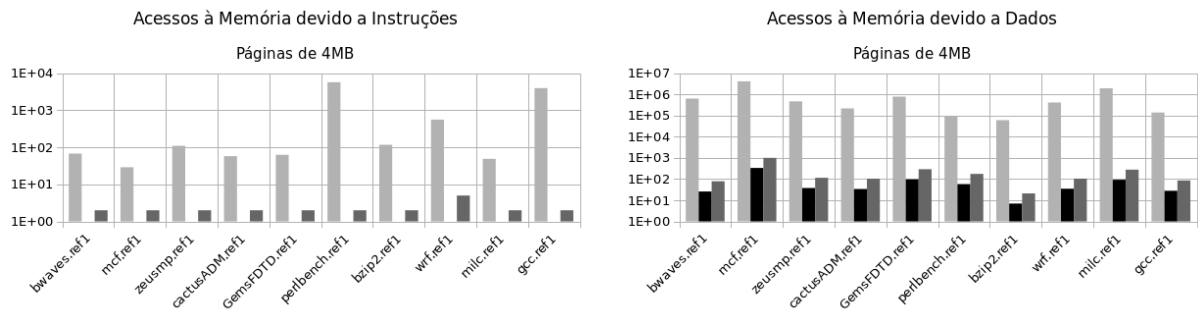
- O número de acessos adicionais causados por *misses* na TLB de instruções em páginas de 4KB corresponde a uma fração importante do número de acessos total. O mesmo ocorre para acessos ocasionados por leitura e escrita de dados.
- Para páginas de 4MB, o número de *misses* na TLB de instruções é praticamente nulo. Os acessos à memória são gerados exclusivamente por *misses* em L3. Se comparado com o caso anterior, este caso apresenta uma grande economia de tempo, porém utiliza uma maior área para a TLB. Dependendo da área e *design* interno do processador, é necessário avaliar corretamente quais benefícios são mais importantes.
- Para os dados em páginas de 4MB, apesar do número de acessos à tabela de páginas ser muito inferior àquele de 4KB, ele ainda apresenta uma fração considerável no total de acessos. Isto indica que, mesmo com páginas de grande comprimento, é necessário que o sistema operacional implemente algoritmos eficientes de escrita e leitura de entradas na tabela de páginas.



(a) Acessos à memória devido a instruções. Páginas de 4KB.

(b) Acessos à memória devido a dados. Páginas de 4KB.

Figura 1. Acessos à memória devido a instruções e dados para páginas de 4KB.

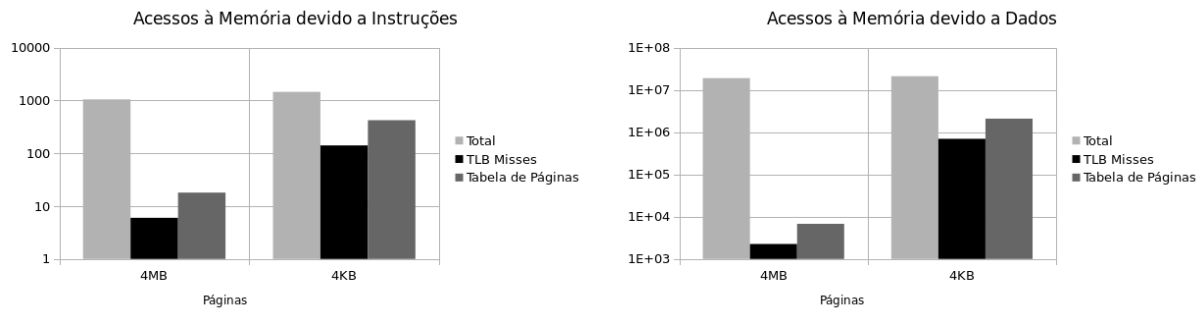


(a) Acessos à memória devido a instruções. Páginas de 4MB. (b) Acessos à memória devido a dados. Páginas de 4MB.

Figura 2. Acessos à memória devido a instruções e dados para páginas de 4MB.

4.2. Resultados para o toy benchmark

As figuras 3(a) e 3(b) representam o resultado da execução deste *benchmark*. Verifica-se, conforme esperado, que a quantidade de *misses* nas TLBs é muito superior para o caso de páginas de 4KB. Para as instruções, a quantidade de misses para tal caso é praticamente 10 vezes maior que o caso de páginas de 4MB. Para dados, esta diferença é ainda maior: cerca de 100 vezes maior. Tais fatos indicam que, para programas adaptados para determinado tamanho de página, seu desempenho pode ser altamente degradado caso ele seja executado em outro ambiente.



(a) Acessos à memória devido a instruções.

(b) Acessos à memória devido a dados.

Figura 3. Acessos à memória devido a instruções e dados para páginas de 4KB e 4MB.

5. Conclusões

Neste projeto, foi possível avaliar o impacto nos caches de TLB, L1, L2 e L3 de páginas de tamanhos distintos. Páginas de 4KB produzem mais *misses* nas TLBs de dados e instruções, gerando assim mais acessos à memória. Por outro lado, tais caches são menores, ocupando, assim, uma área menor dentro do processador. Páginas de 4MB, por sua vez, diminuem significativamente o número de acessos à tabela de páginas, mas ocupam uma área muito superior. É preciso, portanto, avaliar os *tradeoffs* das duas soluções e avaliar quais benefícios serão priorizados no *design* de um processador.

Referências

- Henning, J. L. (2007). Spec cpu2006 memory footprint. *ACM SIGARCH Computer Architecture News*, 35.
- SniperSim (2008). Pinballs.