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

Gustavo Ciotto Pinton[1]

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

ra117136@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 not considered L1, L2 and L3 caches for simplicity reasons. 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, não foram considerados os níveis L1, L2 e L3 de cache. 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 física, 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 2^{14} *bytes*. Por outro lado, se 2 níveis fossem utilizados e se 10 *bits* fossem usados para indexar cada um dos dois níveis, teríamos $4 * 2^{10}$ = 4KB para o primeiro nível e outros 4KB para o segundo. Assim, neste caso, precissaríamos 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 possuia 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 associtativos, um para as instruções e outro para os dados, com 512 entradas cada.

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.

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 politícas 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 politíca 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 probalidade 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 método permitindo o controle das estatísticas dos acessos ao cache. Tal classe é extendida 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 4 caches distintos: 2 para páginas de 4KB e outros 2 para páginas de 4MB. Cada um dos pares 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. 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 California 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 extenderam 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

5. Conclusões