Branch Target Buffer, paginação, pontos flutuantes e chamadas ao sistema

Matheus de Moraes Cavazotti, Stephanie Briere Americo, Talita Halboth Cunha Fernandes

I. Introdução

Inicialmente foi realizada a leitura do artigo de [1] por todos os integrantes do grupo. O fato do material ser em inglês foi uma barreira, e o grupo levou cerca de 18h (3 tardes) para ser ultrapassá-la. Muitas consultas a outros materiais foram necessárias para compreender o conteúdo utilizado nos testes do autor (como [2]), e o grupo teve várias ideias no decorrer do processo. Após discutir quais testes deveriam ser realizados, procurando diversas vezes a ajuda do monitor da disciplina, decidimos nos focar no tamanho e organização do *Branch Target Buffer* (BTB). Entretanto, fora do escopo do artigo, alguns temas foram de extremo interesse para o grupo, e alguns testes extras foram realizados.

II. Experimentação

Num primeiro momento, o objetivo do grupo era reproduzir todos os experimentos que Milenkovic et al. [1] realizaram no artigo, além de alguns experimentos adicionais sobre paginação. Rapidamente o grupo percebeu que não seria fácil, pois a complexidade dos testes se mostrou desafiante. Decidimos, então, nos focar nas experimentações acerca do BTB. Escolhemos o processador Intel® Core™ i7-6700, porém, diferente da arquitetura utilizada pelos autores (P6), não há nenhuma informação sobre o número de entradas do BTB para esta arquitetura.

Os testes foram feitos segundo a seção 5.5.1 do artigo, cujo objetivo é encontrar o tamanho e a organização (mapeamento direto, associativo ou associativo por conjuntos) do BTB do processador. Para medir o *branch misprediction rate* (taxa de falhas na previsão) foi utilizado o *Likwid*, ao invés do *VTune*. O *Likwid* é um pacote de ferramentas simples de ser utilizado por linhas de comando em um terminal, e pode ser usado para medir a performance dos processadores Intel e AMD por meio dos registradores conhecidos como *performance counters*.

A. Branch Target Buffer (BTB)

O BTB é o *buffer* responsável pela previsão de *branches*. O experimento realizado consiste em analisar o comportamento de N_{BTB} - 1 *branches* dentro de um *loop*, variando a distância D entre elas. Essa distância, como podemos ver na Figura 1, é o número de instruções entre uma *branch* e outra. As *branches* no *loop* são sempre tomadas, então elas serão previstas incorretamente (*mispredicted*) se não estiverem presentes no BTB. Nós variamos a distância entre as *branches* a cada experimento, de modo que a posição dos bits DM_Index_T difere para distâncias variadas. Encontramos a distância D adequada quando o *branch misprediction rate* é próximo de 0 para essa distância. Deste experimento, é possível tirar conclusões acerca do tamanho e configuração do BTB.

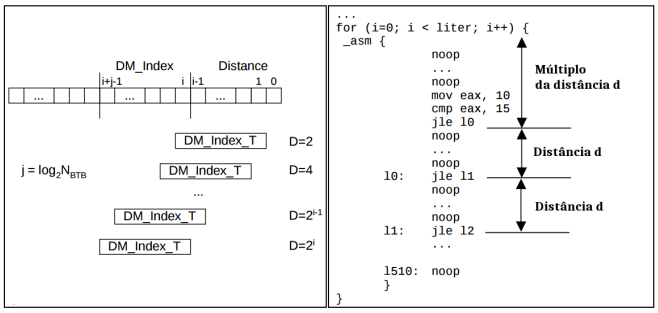

Figura 1. À esquerda, posição dos bits DM_Index_T para distâncias variadas. À direita, a distância D entre instruções. Adaptado de Milenkovic et al. [1].

Se para um N_{BTB} há i distâncias com bons resultados, dobramos o valor de N_{BTB}. Duas situações podem ocorrer: (a) haverão i-1 distâncias boas, então ainda não encontramos o número de entradas da BTB e devemos repetir o procedimento com o dobro de N_{BTB}. Ou (b) nenhuma distância terá *branch misprediction rate* próximo a 0; neste caso, o N_{BTB} anterior é o número de entradas no BTB. Além disso, se há apenas uma distância D com *branch misprediction rate* próximo a 0, podemos concluir que o BTB usa mapeamento direto. Se há duas distâncias D, concluímos que temos uma organização com 2 vias. Da mesma forma, se há exatamente 3 distâncias, há 4 vias. No geral, se há m distâncias, o BTB tem $2^m - 1$ vias.

B. Conclusões sobre o BTB

Fizemos testes exaustivos utilizando uma faixa de valor superior aos testes originais do artigo. Geramos de forma automática (*scripts* Bash) arquivos que testam distâncias entre 2^1 e 2^{10} para cada suposição de N_{BTB} entre 2^{10} e 2^{14} (a execução de todos os testes dura cerca de 3h). Um gráfico com os resultados da variação de distância para cada N_{BTB} pode ser encontrado nos Anexos deste relatório.

Os resultados que obtivemos divergiram dos apresentados no artigo, como é demonstrado no gráfico da Figura 2. Diferente do relatado, o *branch misprediction rate* jamais se aproxima de 100% - a pior taxa que conseguimos por 0.45%. Ficou bastante claro para o grupo que a organização do BTB foi muito aprimorada no decorrer dos anos, e os testes relatados no artigo se tornaram ineficientes para as técnicas mais dinâmicas de previsão.

Encontramos estudos apresentando técnicas cada vez mais elaboradas - por exemplo, utiliza-se redes neurais para previsão

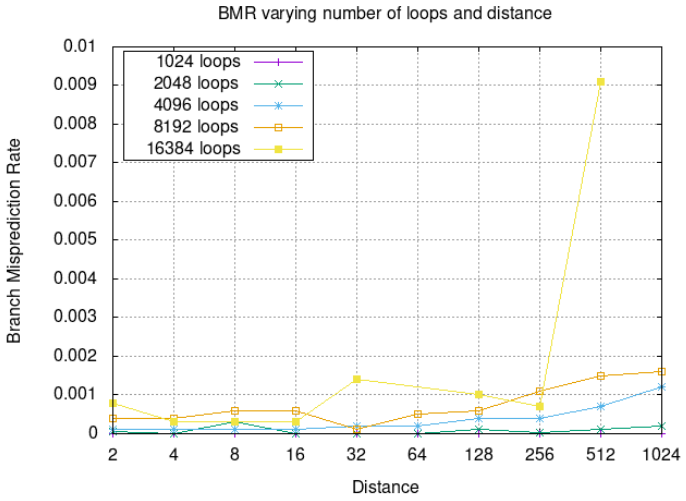

Figura 2. Compilado de testes variando os valores de N_{BTB} e distância D entre as instruções

de *branches* [3]. As técnicas recentes são capazes de detectar loops extensos (que toma a decisão nas primeiras milésimas interações e não toma na última), tornando-os muito mais acertivos. Este fato, por si só, já torna inviável deduzir algo sobre o BTB a partir dos testes de Milenkovic et al. [1]. Isto porque o principio do teste que realizamos é induzir o previsor ao erro por mudar repentinamente a decisão de uma *branch*.

Iniciamos, então, uma busca por informações sobre as arquiteturas mais modernas, principalmente dos chips *Skylake* (Intel i7-6700, mais especificamente). Os resultados foram frustrante, pois as informações são praticamente nulas. Uma das maiores referências da área, Fog [4] descreveu que "a organização do BTB [do *Skylake*] é desconhecida, mas seu tamanho parece ser grande". Já no próprio manual da Intel ([5]), tudo o que sabemos sobre os a previsão de *branches* na arquitetura *Skylake* é que "a previsão foi melhorada".

Analisando os gráficos resultantes de nossos testes (Apêndice A, Figura 8), é possível observar uma maior estabilidade com N_{BTB} = 4096. As distâncias com menor *branch misprediction rate* são 4, 8, 32 e 64. É possível, então, que o Intel i7-6400 possua um BTB com 4096 entradas e 8-vias. Entretanto, esta é apenas uma observação e é impossível afirmar com certeza, pois os resultados foram inconclusivos e nos falta informação sobre a arquitetura.

III. Testes extras

Alguns dos testes realizados foram um pouco fora do tema do artigo (previsão de desvios). Achamos interessante incluir esses testes em um anexo do relatório, pois excederia o tamanho limite de 4 páginas. São pequenos experimentos abordando diversos temas discutidos em sala de aula, o que foi de grande aprendizagem para o grupo.

A. Paginação

Testamos o funcionamento/desempenho da paginação do *Linux Ubuntu* (páginas de 4096Kb) utilizando matrizes. Nos

```
for (int i = 0; i < num; ++i) {
    for (int j = 0; j < num; ++j)
        matriz[i][j] = i;
    }
}
```

```
    for (int j = 0; j < num; ++j) {
{       for (int i = 0; i < num; ++i) {
            matriz[i][j] = i;
        }
    }
```

Figura 3. À esquerda, percorrendo a matriz por colunas. À direita, percorrendo por linhas.

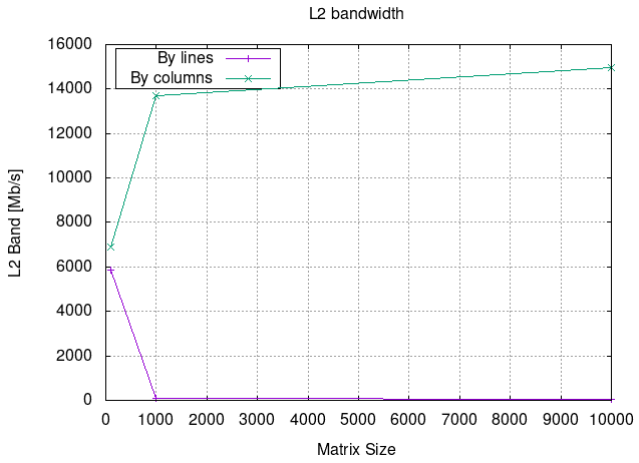

Figura 4. Acessos ao nível L2 da cache.

baseamos em uma das questões da prova, onde uma matriz era percorrida de duas formas: a primeira percorria a matriz por linhas, e a segunda percorria a matriz por colunas (Figura 3). Realizamos esse experimento variando o tamanho de matrizes quadradas e obtendo informação sobre a relação entre o tamanho das matrizes e o tempo de execução do programa. Para evitar otimizações do compilador os programas foram compilados com a opção -O0. Foi usada a função *clock* da biblioteca time.h para medir a quantidade de ciclos do relógio gastos na execução do programa. No primeiro, a execução levou cerca de 48000 ciclos de *clock* para percorrer uma matriz 10000x10000, enquanto no segundo caso, os ciclos foram por volta de 2100000. Ou seja, o percorrer a matriz por colunas é cerca de 44 vezes mais rápido (Figura 4). O motivo é a forma como a matriz é armazenada na memória, como uma sequência de vetores (linhas). Desta forma, percorrer a matriz por colunas acessa posições em sequência na memória e é a forma mais eficiente, pois esgotamos uma página antes de precisar carregar outra. Percorrer por linhas, por sua vez, faz com que a troca de páginas seja muito maior, o que é custoso.

B. Operações com Pontos Flutuantes

Outro teste realizado para a análise do desempenho do processador foi com relação a pontos flutuantes. Variamos o número de iterações em um *loop* que realiza somas de inteiros e pontos flutuantes. Percebemos que variáveis de ponto flutuante levam quase o dobro de ciclos de *clock* para realizar a mesma operação quando comparadas com variáveis inteiras. Como pode ser observado no gráfico da Figura 5, a execução de uma soma de inteiros é cerca de 1.8x mais rápida que

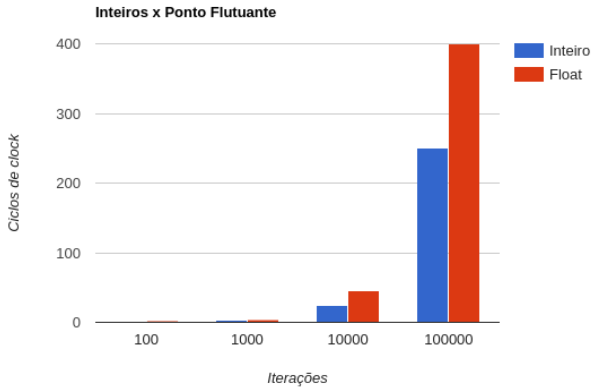

Figura 5. Gráfico mostrando a diferença de ciclos de clock gastos em operações de soma utilizando variáveis do tipo inteiro e de ponto flutuante.

a soma de pontos flutuantes. Parte da razão se deve ao fato de que operações com ponto flutuante necessitam de um coprocessador, então o processador precisa aguardar o retorno do coprocessador responsável para obter o resultado.

C. Chamadas ao Sistema

Nesse experimento, foi usado uma implementação da função *malloc* feita como trabalho da disciplina de *Software* Básico, ministrada pelo professor Bruno Müller Junior. Nesta implementação, o espaço da *heap* (valor da *BRK*) é incrementado em 4096 *bytes* apenas quando necessário, ao invés de em cada chamada da função. O objetivo dos testes foi avaliar a diferença de desempenho a entre a implementação usada no trabalho (*M-4096*) e uma variação que soma à *BRK* a quantidade de bytes necessárias para alocar um bloco de tamanho t (*M-t*).

Para coletar os dados, foram realizadas 3 baterias de testes, todas executando 100 vezes dois programas que faziam, cada um, 1.000 alocações usando *M-4096* e *M-t*, respecitivamente. A primeira bateria alocava blocos de 100 *bytes* (*B-100*), a segunda, 1.000 *bytes* (*B-1k*) e a terceira, 10.000 *bytes* (*B-10k*).

Os resultados obtidos estão resumidos na Tabela 1. A primeira coluna indica a bateria de testes no qual os dados foram coletados. A segunda coluna mostra a porcentagem das alocações feitas em *M-t* que foram mais rápidas que *M-4096*. A última coluna mostra a diferença de tempo entre as chamadas de *M-t* e *M-4096*. Nos experimentos também foi possível ver a ocorrência de troca de processos (dados anômalos).

Tabela 1

Bateria	$T_{M-t} < T_{M-4096}$	$\lvert T_{M-t} - T_{M-4096} \rvert$
B - 100	29,1%	61,2 ns
B - 1K	72,1%	1047 ns
B - 10K	99,8%	69.866 ns

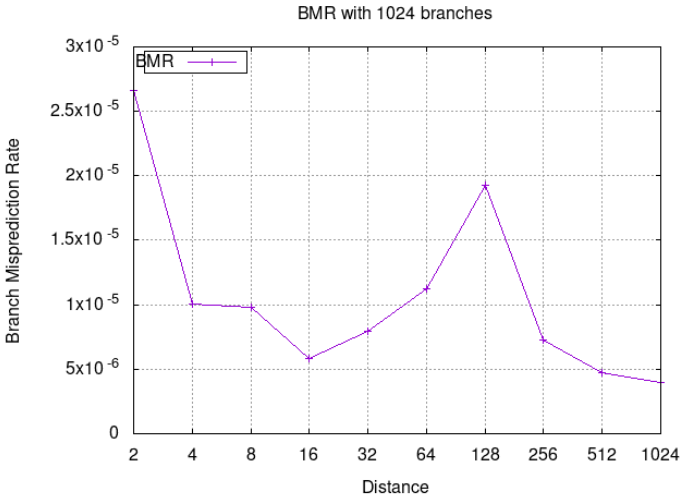

Figura 6. N_{BTB} = 1024

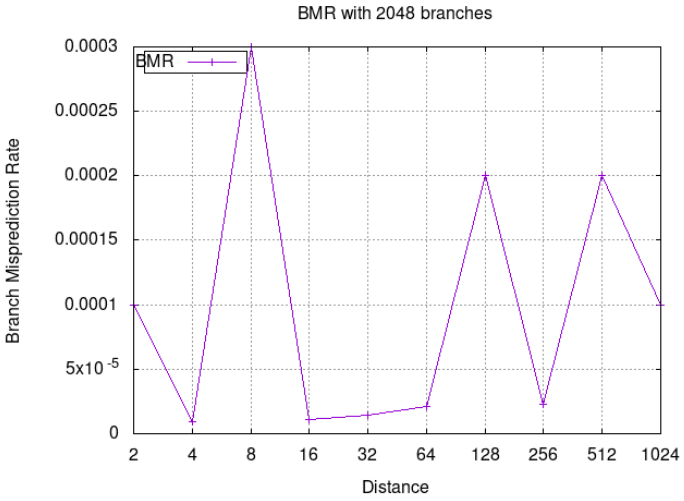

Figura 7. N_{BTB} = 2048

APÊNDICE A
GRÁFICOS RESULTANTES DO EXPERIMENTO COM BTB

Geramos uma sequência de gráficos para analisar a variação de distância D entre 2^1 e 2^{10} para cada suposição de N_{BTB} entre 2^{10} e 2^{14}. O resultado pode ser observado nas Figuras 6 a 10.

REFERÊNCIAS

[1] M. Milenkovic, A. Milenkovic, and J. Kulick, "Demystifying intel branch predictors," in *IN WORKSHOP ON DUPLICATING, DECONSTRUCTING AND DEBUNKING*, pp. 52–61, John Wiley & Sons, 2002.

[2] M. Godbolt, "The btb in contemporary intel chips," 2016. http://xania.org/201602/bpu-part-three. Acessado em 20/06/2017.

[3] D. A. Jiménez and C. Lin, "Neural methods for dynamic branch prediction," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 4, pp. 369–397, 2002.

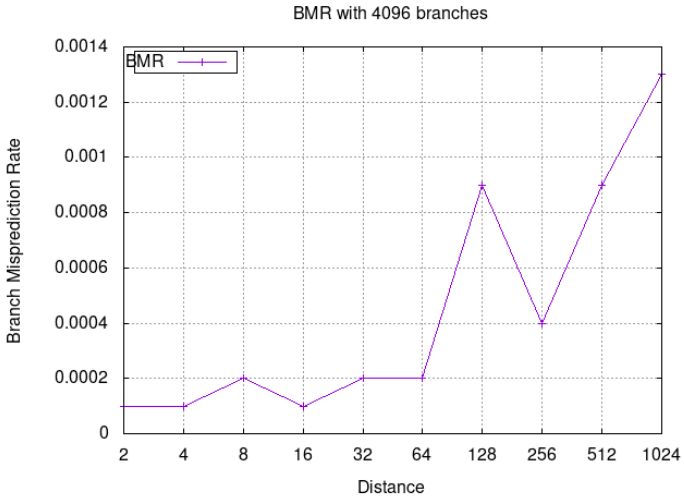

Figura 8. N_{BTB} = 4096

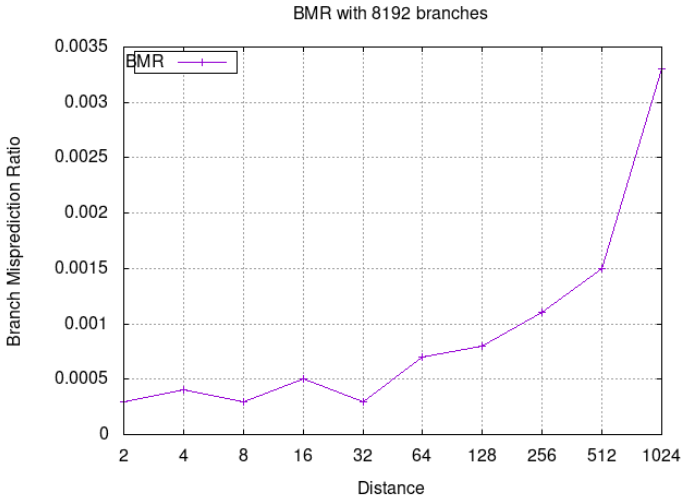

Figura 9. N_{BTB} = 8192

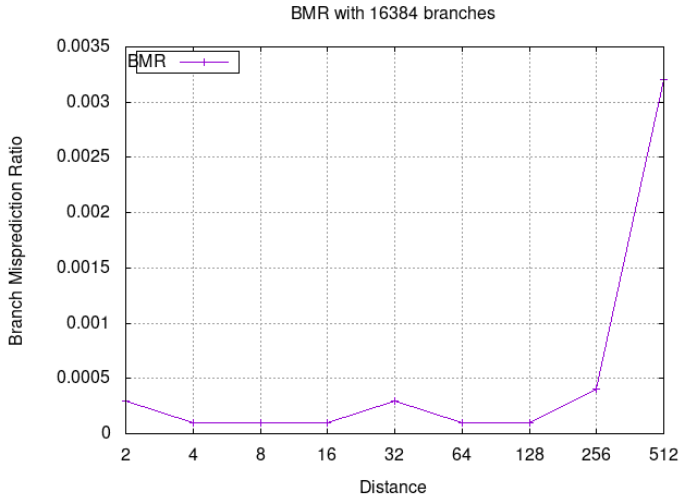

Figura 10. N_{BTB} = 16384

[4] A. Fog, “The microarchitecture of intel, amd and via cpus,” 2017. http://agner.org/optimize/microarchitecture.pdf.

[5] Intel, “Intel® 64 and ia-32 architectures optimization reference manual,” 2016. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.