Reprodução de um elemento de artigo

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 curves* ***single thread*** *and* ***ideal*** *from figure 3 in the article Accelerating Decoupled Look-ahead via Weak Dependence Removal: A Metaheuristic Approach [Parihar and Huang 2014]. We used Sniper micro-architecture simulator and pinballs of SPEC CPU2006 benchmarks with 1 billion instruction detailed regions and no warm up. We compared the effects of ideal caches and branch predictors on IPC measures. A gshare predictor was implemented and added to the simulator. Some difficulties met during the project will be also described.*

Resumo. *Este relatório reproduz as curvas* ***single thread*** *e* ***ideal*** *contidos na figura 3 do artigo Accelerating Decoupled Look-ahead via Weak Dependence Removal: A Metaheuristic Approach [Parihar and Huang 2014]. Foi utilizado o simulador de micro-arquiteturas Sniper e os pinballs dos benchmarks do SPEC CPU 2006 com regiões detalhadas contendo 1 bilhão de instruções e sem quaisquer intruções de warmup. Comparou-se os efeitos de caches e preditores ideais no IPC. Um preditor gshare foi implementado para o simulador. Serão discutidos também algumas dificuldades encontradas durante o projeto.*

1. Introdução

O artigo escolhido para este projeto foi *Accelerating Decoupled Look-ahead via Weak Dependence Removal: A Metaheuristic Approach* [Parihar and Huang 2014], cujo objetivo foi propor um método para melhorar o desempenho de *look ahead threads* a partir de algoritmos genéticos. Neste artigo, os autores argumentam que o IPC de uma aplicação com apenas uma *thread* é limitado justamente pelo IPC desta *thread* auxiliar, que é executada em um outro *core*.

A ideia principal deste tipo de arquitetura é gerar uma versão reduzida da programa principal, de maneira a não obter um grande *overhead* e redundância (executar o mesmo programa duas vezes), e a diminuir os efeitos causados por *misses* nos *caches* e nos preditores de salto, à medida que evita a perda de ciclos em casos em que o preditor previu erroneamente o salto ou quando o dado procurado não se encontra no primeiros níveis de *cache*. Tal versão reduzida é executada, conforme já dito no parágrafo anterior, em outro *core* do processador, paralelamente à *thread* principal. O desafio reside, portanto, em como obter uma *lookup thread* representativa, mas, ao mesmo tempo, a mais simples possível, já que, de acordo com [Parihar and Huang 2014], tal *thread* torna-se o novo limite de velocidade do sistema.

Nestes sistemas, um *parser* é utilizado para analisar o binário do programa e criar uma versão *esqueleto*, que consistirá na *lookup thread*. Durante a execução, esta *thread*

envia os resultados de saltos condicionais já comitados através de uma fila para o *core* principal, que, por sua vez, pode utilizar tais resultados no seu processamento. Além disso, dados são diretamente enviados ao *caches* compartilhados. Os autores destacam a existência de *dependências fracas*, isto é, instruções que contribuem o mínimo para os propósitos da *lookup ahead thread*. Exemplos destes tipos de instruções podem ser instruções aritméticas e lógicas que não mudam o resultado de um registrador na maioria do tempo, ajustes inúteis em registradores (realizar uma operação em um registrador sendo que ele será o alvo de um *store* em seguida) e instruções de *store/load* que sempre escrevem ou carregam o mesmo valor. Infelizmente, sua análise torna-se muito difícil caso seja realizada estaticamente, uma vez que uma instrução se torna uma dependência fraco de acordo com o contexto do programa. Os autores ainda afirmam que não conseguiram encontrar nenhuma característica especial em comum que pudesse identificá-las rapidamente no momento de geração dos esqueletos. É importante lembrar que a identificação de uma instrução errada pode de fato piorar o desempenho do sistema, dado que novos *misses* podem ser inseridos e que não ocorreriam naturalmente.

Sendo assim, dada a natureza dinâmica das dependências, a não-linearidade dos efeitos de performance e a interdependência das cadeias de intruções, a melhor solução encontradas pelos autores foi recorrer a heurísticas e análise de código para predizer a fraca dependência. No caso, eles utilizaram algoritmos genéticos e propuseram um novo método para encontrar instruções com grande probabilidade de serem fracas dependências. O método proposto foi capaz de aumentar a performance total dos sistemas testados em até 1.48x (1.14x média geométrica).

Apesar de toda a explicação, os elementos da figura escolhida para a reprodução não abrange diretamente as ideias anteriores. O objetivo neste relatório é replicar as curvas *ideal* e *single thread* da figura 1 (figura 3 de [Parihar and Huang 2014], página 3). A curva relativa a *single thread* corresponde à execução *normal* dos *benchmarks*, enquanto que a *ideal* supõe que nunca ocorrerão *mispredictions* nos preditores ou *cache misses*, ou seja, supões-se que todos os dados estão disponíveis nos *caches* e que o preditor sempre acertará suas previsões. As duas outras curvas, por exigirem uma grande alteração na estrutura dos simuladores, serão deixadas para uma outra oportunidade. Destaca-se, ainda, que os autores utilizaram os *benchmarks* do `SPEC CPU2000`. Por dificuldades de licença e por razões de simplicidade (já possuímos os *pinballs*), usamos o `SPEC CPU2006`.

Em um primeiro contato com os autores, fui aconselhado a utilizar o *gem5* ou o *SimpleScalar* como simuladores. Entretanto, escolhemos o *sniper*, dado experiências anteriores com a ferramenta *pin*, no qual tal simulador é baseado. O *sniper* implementa o modelo de processamento chamado *interval core model*, que aumenta o grau de abstração da simulação, permitindo tempos menores de avaliação e desenvolvimento [Carison and Heirman 2013]. Contrariamente ao *gem5*, por exemplo, que modela todos os estágios de *pipeline*, o *sniper* divide a execução em intervalos, delimitados pelos eventos denominados de *miss events*, tais como *branch mispredictions*, *cache* e TLB *misses*, etc. O simulador mantém um registro interno do tempo de simulação em cada *core* e, a cada *miss event*, esse contador é aumentado com o tempo do respectivo evento. Dessa forma, o *sniper* é capaz de calcular uma série de estátisticas, como o IPC, que foi utilizado neste relatório como métrica.

Enfim, neste relatório serão apresentados os resultados das curvas *ideal* e *single*

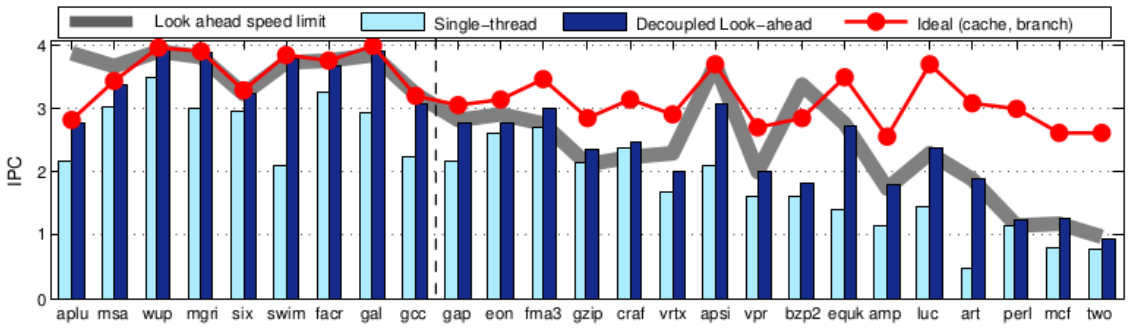


Figure 3. Performance comparison of 4 configurations. Shown in the bars are baseline single core (left) and a decoupled look-ahead system (right). Two upper-bounds are shown: the performance of a single core with idealized branch predictions and perfect cache accesses (curve with circles), and the the approximate speed limit of the look-ahead thread (gray wide curve indicating approximation). The applications are sorted with increasing performance gap between the decoupled look-ahead system and the prediction- and accesses-idealized single-core system.

Figura 1. Figura escolhida para reprodução com legenda original. Extraída de [Parihar and Huang 2014].

thread da figura 1 para a configuração do artigo (consultar próximas seções) e para uma configuração padrão do *sniper*, que modela a microarquitetura *Gainestown* da *Intel*.

2. Implementação

Nesta seção, serão discutidos detalhes da implementação e algumas dificuldades encontradas na configuração e instalação do *sniper*.

2.1. Configuração do *sniper*

O *sniper* foi executado em uma máquina virtual de 64 *bits Ubuntu 14.04 LTS* e foi compilado com o *gcc 4.4*. A compilação não apresentou nenhuma grande dificuldade em específico, diferentemente da execução dos *pinballs*, que exigiu um pouco mais de estudo da ferramenta. Dois tipos de *pinballs* foram utilizados: aqueles com 100M instruções de *warmup* e 30M instruções na região detalhada, e, posteriormente, os de 1B de instruções na região detalhada e sem *warmup*. Para os últimos, a integração com o *sniper* foi bem simples, visto que foi necessário apenas o uso da opção `--pinballs` na comando `run-sniper`. Para os primeiros, entretanto, o desafio foi dizer ao simulador para executar a região de *warm up* no modo correto, isto é, `CACHE_ONLY`. As seguintes propostas foram implementadas:

- Utilizar a opção `--roi`: o uso dessa opção fazia com que todo o *pinball* fosse executado em modo `CACHE_ONLY`, gerando nenhum arquivo de saída e nenhuma estatística.
- Implementação de *pintool*: uma nova *pin tool* foi implementada a fim de contar as intruções e chamar a função `SimRoiStart()` explicitamente. Essa função e várias outras são fornecidas pelo arquivo `sniper/include/sim_api.h` e podem ser adicionadas aos programas simulados. Infelizmente, o *sniper* não é capaz de executar o binário do *pin*, uma vez que este último foi pré-compilado em uma máquina de 32 *bits*.
- Uso de uma máquina virtual de 32 *bits*: já que o *pin* não pode ser rodado na compilação em máquinas de 64 *bits*, compilei o *sniper* novamente, mas agora em

uma máquina virtual de 32 *bits*. Nesta oportunidade, não pude executar corretamente, uma vez que o *pinballs* haviam sido criados em máquinas de 64 *bits* e não poderiam rodar em máquinas de 32 *bits*.

- Utilizar um *script python* com a opção `-s`: depois de perguntar do *Google group* do simulador, recebi a resposta de que um *script python* capaz de contar as instruções é disponível no diretório `sniper/scripts`. Tal *script* se chama `roi-icount.py` e, no nosso caso, deve ser chamado com o parâmetro 0:100000000:30000000, indicando a largura de instruções de cada região.

Apesar de todo o trabalho de configuração para o uso dos *pinballs* com 100M de instruções de *warm up*, eles não foram utilizados, uma vez que os resultados encontrados foram muito distantes daqueles encontrados no artigo. Ao invés deles, usamos os *pinballs* com 1B de instruções na região de interesse. Maiores detalhes podem ser encontrados na seção **Resultados**.

2.2. Ambiente de testes do artigo

A tabela 1 resume a configuração em que os resultados do artigo foram encontrados. As linhas indicadas por um ✓ puderam ser corretamente configuradas no *sniper*, enquanto que as indicadas por um ✗ não puderam ser atribuídas, uma vez que modificações mais profundas no modelo de intervalos, comentado na seção **Introdução**, deveriam ser realizadas. Dado o prazo e os problemas enfrentados durante o projeto, preferimos deixar tais requisitos para o projeto 4.

Tabela 1. Configuração utilizada no artigo. Extraído de [Parihar and Huang 2014].

Baseline core		
Fetch/Decode/Issue/Commit	8/4/6/6	✗
ROB	128	✓
Functional units	INT 2+1 mul +1 div, FP 2+1 mul +1 div	✗
Fetch Q/ Issue Q / Reg. (int,fp)	(32, 32) / (32, 32) / (80, 80)	✗
LSQ(LQ,SQ)	64 (32,32) 2 search ports	✗
Branch predictor	Gshare – 8K entries, 13 bit history	✓
Br. mispred. penalty	at least 7 cycles	✓
L1 data cache (private)	32KB, 4-way, 64B line, 2 cycles, 2 ports	✓
L1 inst cache (private)	64KB, 2-way, 128B, 2 cycles	✓
L2 cache (shared)	1MB, 8-way, 128B, 15 cycles	✓
Memory access latency	200 cycles	✓

As configurações da tabela 1 podem ser encontradas no arquivo `article.cfg`. Para o caso ideal, utilizamos o arquivo `ideal-article.cfg`, que extende o primeiro, mas modifica algumas propriedades de forma a obter o valor máximo de IPC. Para o preditor ideal, mudamos a propriedade `mispredict_penalty` para 0 e para todos os *caches*, a propriedade `perfect` para `true`. O ajuste da latência da memória, por sua vez, é realizado pelo parâmetro `perf_model/dram/latency`. O ajuste do tamanho do *reorder buffer* é feito por `window_size`, já que o simulador mantém uma janela de instruções para cada *core* simulado [Carison 2012], o que equivale justamente à função do *reorder buffer* em um processador super-escalar *out-of-order*. Além disso, implementamos um preditor do tipo *gshare*, a partir das classes abstratas fornecidas pelo simulador com as características

apresentadas acima. Enfim, utilizamos o modelo do processador padrão *Nehalem* para modelar as intruções e micro-instruções. Uma das metas do projeto 4 é implementar um modelo mais parecido ao POWER5, que foi utilizado no artigo.

2.3. Implementação de preditor *Gshare*

A implementação de um modelo de preditor no *sniper* é realizado a partir da definição de uma classe abstrata chamada `BranchPredictor`, que possui 2 métodos virtuais, sendo eles `predict()` e `update()`. Todos recebem como parâmetro o endereço atual de *PC*. O primeiro retorna um valor *booleano*, indicando se a operação de salto deve ou não ser realizada e o segundo atualiza o estado preditor e o histório de saltos de acordo com o resultado do salto após que tal operação tenha sido *comitada*. Implementou-se, portanto, a classe `GShareBranchPredictor` que redefine estes dois métodos e mantém uma variável contendo o estado dos 13 últimos saltos (13 *bits*). Tal classe possui também um vetor de preditores que implementam a máquina de estados de 2 *bits* (classe `SaturatingPredictor<2>`) e, a cada operação, o índice desse vetor é obtido através de um *ou exclusivo* entre o endereço PC e a variável de histórico. Por fim, é necessário adicionar essa classe no método `create()` do arquivo `branch_predictor.cc` e compilar o *sniper* novamente.

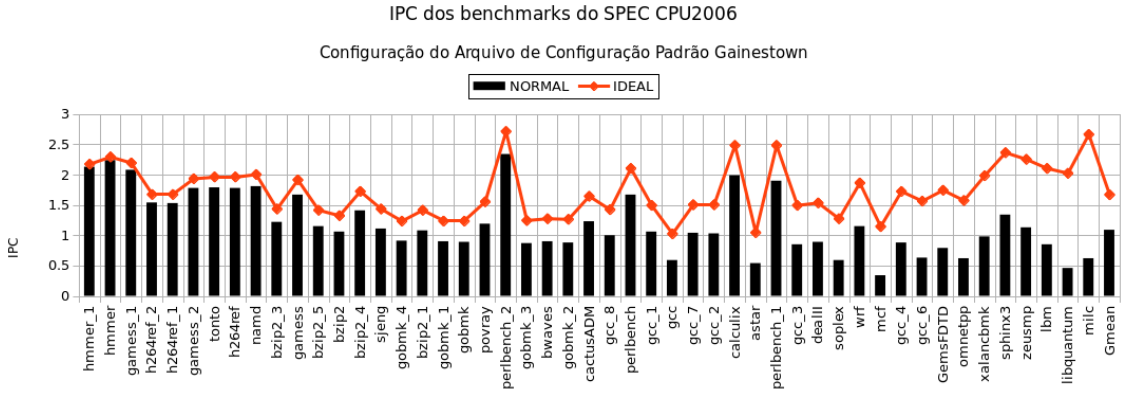
3. Resultados

Contrariamente aos resultados da figura 1 que foram obtidos para os *benchmarks* do `SPEC CPU2000`, os testes produzidos neste relatório foram gerados a partir dos *pinballs* do `SPEC CPU2006`.

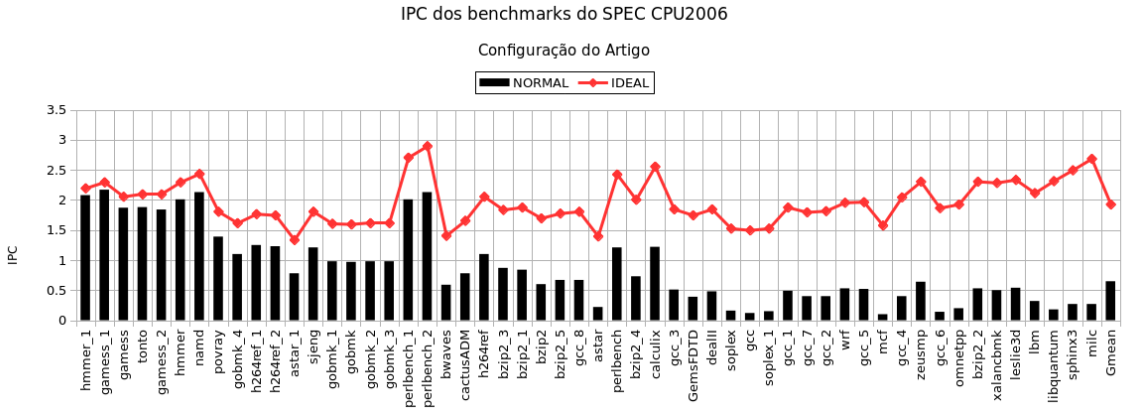
No total, 4 arquivos de configuração foram utilizado, dois para a micro-arquitetura *Gainestown* e duas para a micro-arquitetura do artigo. É necessário observar que o caso referente ao *Gainestown* não possui alguma ligação com o artigo, porém ele foi executado mesmo assim com o intuito de testar completamente o ambiente e de comparar os resultados finais. Conforme comentado anteriormente, utilizamos os *pinballs* com 1B de instruções na região detalhada, gastando, em média, aproximadamente 30 minutos para cada entrada dos *benchmarks*. A figura 2(a) representa os resultados para a micro-arquitetura *Gainestown* e a 2(b), para a configuração da tabela 1. Os *benchmarks* foram posicionados em ordem crescente de diferença entre o desempenho ideal e o normal.

Observa-se que os desempenhos ideais das duas execuções apresentam IPCs muito parecidos entre si, valendo 1.93 e 1.68 em média geométrica para a configuração do artigo e *Gainestown*, respectivamente. As medidas de IPC para os casos normais variam mais entre si em média geométrica, valendo de 0.65 para a configuração do artigo e 1.09 para o *Gainestown* (variação de 1.68x).

Em relação aos resultados encontrados no artigo, nota-se que, em geral, eles são maiores que os encontrados nos experimentos encontrados neste relatório. Uma possível explicação reside no fato de que os testes produzidos em [Parihar and Huang 2014] utilizaram os programas completos para as estatísticas. Durante os testes com os *pinballs* com 130M de instruções, constatou-se que os resultados obtidos eram muitos inferiores àqueles dos *pinballs* de 1B instruções. É evidente que quanto mais instruções executarmos, mais parecidos serão os resultados encontrados. Entretanto, o tempo de execução também cresce muito: se assumirmos que o tempo de execução varia linearmente com o número de



(a) Resultados da simulação dos *benchmarks* do `SPEC 2006` para o arquivo de configuração *Gainestown*.



(b) Resultados da simulação dos *benchmarks* do `SPEC 2006` para o arquivo de configuração do artigo.

Figura 2. Resultados da simulação.

instruções, a execução completa de um *benchmark* com um 1 trilhão de instruções levaria aproximadamente 500 horas!

4. Conclusões

Foi possível reproduzir, através do *sniper*, resultados semelhantes àqueles encontrados no artigo para as curvas *ideal* e *single thread*, apesar de que diferentes *benchmarks* foram utilizados. Duas configurações foram testadas: uma para a microarquitetura *Gainestown* e a outra para o ambiente configurado no artigo. Notou-se que, em geral, a primeira configuração apresentou resultados melhores de IPC, em média geométrica. Implementou-se, ainda, um preditor to tipo *gshare* a partir das classes abstratas fornecidas pelo *sniper*.

Referências

Carison, T. E. (2012). Interval simulation. `http://snipersim.org/w/Interval_Simulation`.

Carison, T. E. and Heirman, W. (2013). *The Sniper User Manual*.

Parihar, R. and Huang, M. C. (2014). Accelerating decoupled look-ahead via weak dependence removal: A metaheuristic approach. *International Symposium on High Performance Computer Architecture*.