Contagem do número de instruções dos *benchmarks* do SPEC CPU2006 e implementação de novas *pintools*

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 instructions executed by each benchmark in SPEC CPU2006. It was achieved thanks to the Intel's pin application. All results were obtained by running the benchmarks with the ref inputs and a single iteration. Two new pin tools were equally developed, capable of listing the number of executed instructions of every routine of all threads that compose the program and simulating the 1-bit and 2-bit branch predictor models. Five benchmarks, run with inputs belonging to the ref set, were used to test these new tools.*

Resumo. *Este relatório apresenta o número de instruções executadas por cada benchmark presente no SPEC CPU2006, calculado através da utilização da ferramenta pin, desenvolvida pela Intel. Os valores encontrados correspondem às entradas do tipo ref e a uma única iteração. Além disso, desenvolveu-se duas novas pin tools, cujas saídas são uma lista com o número de instruções executas pelas rotinas de um programa, separadas pela sua respectiva thread, e o resultado de uma simulação de dois modelos (1-bit e 2-bit) de branch predictor. Tais ferramentas foram testadas em 5 benchmarks na configuração ref.*

1. Introdução

SPEC, do inglês *Standard Performance Evaluation Corporation*, é uma organização constituída por fabricantes de *hardware*, *software* e instuições de pesquisa, cujo objetivo é definir uma série de testes relevantes e padronizados para a análise da performance de um computador. Tais testes podem ser igualmente denominados de *benchmarks* e visam avalisar um aspecto específico de processamento. Durante a aula, vimos que existe uma grande variedade de *benchmarks* disponíveis na *internet*, sendo que alguns foram implementados, por exemplo, para avaliar o desempenho de aplicações *web*. Um *benchmark* realiza um conjunto de operações definidos, chamado de **workload**, e produz um resultado, ou seja, uma **métrica** que tenta avaliar o desempenho do computador submetido ao respectivo *workload* [SPEC 2011]. Tendo em vista tais conceitos, o SPEC CPU2006 é um conjunto de 31 *benchmarks* que procuram avaliar a performace de três componentes principais, sendo eles o processador, a arquitetura de memória e os compiladores. Os *benchmarks* são distribuídos em dois *suites*, denominados de CINT2006 e CFP2006, que se distinguem quanto à natureza do seu processamento intensivo: o primeiro é focado na performance das operações que utilizam números **inteiros**, sendo que o segundo, de números em **ponto flutuante**. Essencialmente, SPEC CPU2006 oferece duas métricas, **speed** e **rate** (ou **throughput**), medindo, respectivamente, o quão rápido um computador completa uma única tarefa e quantas tarefas tal sistema pode realizar em um pequeno período de tempo. Cada métrica possui, por sua vez, quatro *variações*, que se diferenciam quanto ao *suite* (inteiro ou ponto flutuante) e ao método de compilação. Em relação a este

último, duas opções são disponibilizadas: `base`, que apresenta requisições mais estritas, isto é, as *flags* de compilação devem ser usadas na mesma ordem para todos os *benchmarks* de uma dada linguagem e é exigida para um teste *reportable* (execução que pode ser publicada), e `peak`, opcional e com menos exigências.

`PIN`, por sua vez, é uma ferramenta para instrumentação e análise de programas, à medida que ela permite a inserção de código dinamicamente ao executável. Esta ferramenta é capaz de interceptar a execução da primeira instrução e gerar um novo código a partir dela. Uma `pintool` pode ser definida como uma extensão do processo de geração de código realizada pelo `pin`, já que é capaz de interagir com este último e comunicar quais funções, ou *callbacks*, o aplicativo deve inserir ao código. De maneira geral, uma `pintool` é composta por dois componentes, chamados de *instrumentation code* e *analysis code*. O primeiro deve decidir onde inserir o novo código, isto é, em que locais as rotinas de análise deverão ser lançadas. É nessa fase, portanto, que características **estáticas** do código, tais como, nome de rotinas ou número de instruções que as compõem, devem ser exploradas. O segundo, por sua vez, é chamado à medida que o código é executado e, dessa forma, pode afetar significamente a performance de um executável se o determinado código apresentar complexidade elevada.

Neste relatório, será abordada, na primeira parte, o uso de uma `pintool` para a avaliação do número de instruções executadas por cada um dos *benchmarks* e, em seguida, a implementação de duas novas ferramentas capazes de determinar quantas instruções cada rotina de cada *thread* foram executadas e de simular a performance de um *branch predictor*.

2. Contagem das instruções

A fim de calcular as instruções de cada *benchmark* , dois *scripts bash* foram implementados. O primeiro, chamado de `run-pintool-all-benchmarks.sh` verifica e executa o *runspec*, escrito em *perl*, para cada um dos *benchmarks*, além de compilar a *pintool* que será utilizada. Tal *script* comunica alguns parâmetros importantes ao comando *runspec*, tais como o método de compilação (`base`), o conjunto de entradas que será transmitido aos programas (`ref`, no nossa caso), o número de iterações (1) e, evidentemente, o nome do *benchmark*. A execução deste comando produz um arquivo de extensão `.tmp.log` no diretório do projeto, que é copiado posteriormente a uma pasta, cujo nome é igual ao do *benchmark* que acabou de ser executado. O arquivo de configuração transmitido ao comando `runspec` faz referência ao segundo *script*, `run-spec-command.sh`, responsável por relacionar o aplicativo `pin` com o respectivo *benchmark*. Este *script* recebe como parâmetro o comando que o `SPEC` utiliza e o retransmite para o `pin`, que é responsável por executá-lo efetivamente.

No manual de referência do `pin` [Intel 2012], são indicadas quatro maneiras distintas de se calcular o número de instruções executadas por um programa. A primeira, `inscount0`, insere a rotina de análise antes de cada instrução, produzindo, assim, uma grande perda de performance. A segunda, `inscount1`, é superior à anterior, à medida que utiliza uma outra medida de granularidade, chamado de `BBL` (do inglês, *basic block*) e, portanto, economiza diversas chamadas à função de análise. A terceira rotina, chamada de `inscount2`, usa o mesmo princípio que a anterior, porém apresenta melhor desempenho, visto que faz uso de dois recursos a mais que `inscount1`. O primeiro recurso é a mudança de `IPOINT_BEFORE` para `IPOINT_ANYWHERE`, que autoriza o `pin` escolher em que ordem a função de análise é colocada, permitindo, assim, que ele escolha o ponto que requeira mínimas operações de salvamento e restaturação dos registradores. Além disso, esta ferramenta também usa a opção de *fast call linkage*, que explora o fato de que alguns compiladores podem eliminar o *overhead* que, para funções pequenas como a a função de análise, é comparável ao

próprio conjunto de operações da respectiva função. Esta opção é ativada através do uso de `PIN_FAST_ANALYSIS_CALL`. Por fim, o último programa utiliza, além dos recursos comentados anteriormente, uma unidade de armazenamento rápido, chamado de `TLS`, indexado pelos *indexes* das *threads* para gerar o número de instruções por *thread*.

Tendo em mente estas características, escolhe-se o programa **`inscount2`** para o cálculo do número de instruções, visto que este último apresenta o maior número de otimizações e que, neste contexto, não visa-se encontrar uma contagem por *threads*, mas sim global. Os *benchmarks* foram rodados em apenas um *core* do processador *Intel i3* e levaram, ao todo, 20 horas aproximadamente. Os resultados encontrados estão listados abaixo. Alguns *benchmarks* possuem mais de uma entrada e, portanto, produzem mais de uma saída. Tais saídas estão representadas por barras de cores distintas nas figuras a seguir, sendo que a primeira execução corresponde à barra mais próxima da extremidade esquerda e a última, à mais próxima do canto direito. A figura 1 representa a contagem para os *benchmarks* presentes no conjunto `CINT2006`, enquanto que os da figura 2 ao `CFP2006`.

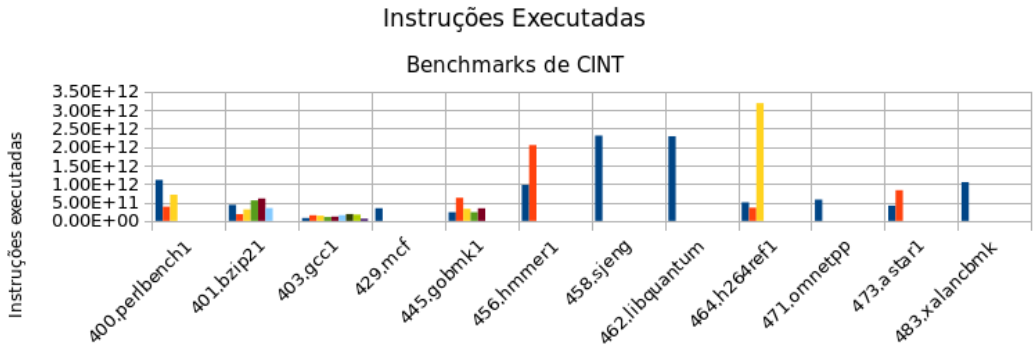

Figura 1. Número de instruções dos *benchmarks* pertencentes ao conjunto `CINT`.

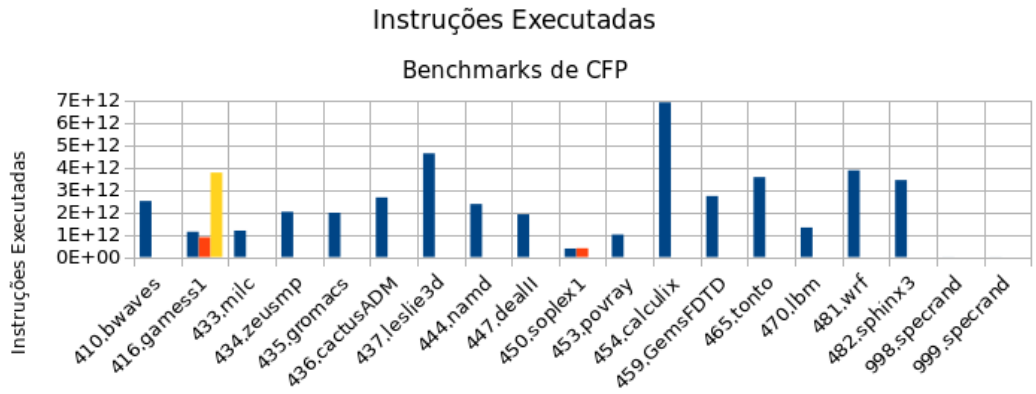

Figura 2. Número de instruções dos *benchmarks* pertencentes ao conjunto `CFP`.

Verifica-se, portanto, que o *benchmark* que utiliza mais entradas em seus testes é `403.gcc`, com 9 entradas, totalizando 1.152.387.847.873 instruções executadas. Os *benchmarks* que rodaram mais e menos instruções foram, respectivamente, `454.calculix` (com 6.894.341.859.256 instruções) e `998.specrand`/`999.specrand` (tendo 536.611.748 instruções cada). Em geral, os *benchmarks* do conjunto inteiro executam menos instruções que os do conjunto de ponto flutuante.

3. Implementação de novas *pin tools*

Nesta seção, serão discutidas as duas novas *pin tools* desenvolvidas.

3.1. Contagem de instruções por *thread* e por rotina

Tendo em vista que os programas apresentados na seção anterior não permitem a contagem de instruções por rotina e por *thread*, propõe-se a implementação de uma nova *pintool* contendo estas duas funcionalidades. Para isso, utiliza-se a API da ferramenta para a adição de funções de *instrumentação* para cada rotina e de *análise* para suas instruções. Além disso, faz-se uso do *TLS* para a armazenagem de informações específicas a uma *thread*.

O registro da função de instrumentação é realizada através de uma chamada a `RTN_AddInstrumentFunction()`, cujo parâmetro é uma *callback function* que é executada uma vez para cada rotina do programa. Dentro desta função, processa-se todo o comportamento estático do programa, isto é, explora-se o seu estado independente da sua execução. Cada rotina é descrita por uma estrutura de dados, cujo nome é `RTN`. Tal estrutura permite que diversas informações sejam extraídas, como por exemplo, o nome da rotina, através de `RTN_Name()`, o número de instruções estáticas dentro dela, com `RTN_NumIns()` e quais instruções pertencem a ela. Este último é realizado através da iteração de uma lista ligada, acessada através de `RTN_InsHead()` ou `RTN_InsTail()`. No entanto, tais funções devem ser chamadas somente depois de `RTN_Open()` ser executada, conforme documentado no manual da ferramenta [Intel 2012].

O programa define duas novas classes, `Thread_node` e `Routine_node`, responsáveis por armazenar informações sobre as *threads* e rotinas executadas. Os objetos instanciados destas classes compõem duas listas ligadas, sendo que cada nó do tipo `Routine_node` contém o início e o fim de uma lista de objetos `Thread_node`. Tal escolha de implementar uma lista de `Thread_node` dentro de cada instância de `Routine_node`, e não o inverso, foi adotada a fim de melhorar a performance da rotina de análise, que será explicada mais adiante. Além de dois ponteiros para o início e fim de uma lista ligada de `Thread_node`s, um objeto da classe `Routine_node` possui o nome da rotina, obtido através de `RTN_Name()`, a referência para o próximo nó da lista e o *id*, que é dado pela função `RTN_Id()`. `Pin` atribui a cada rotina uma identificação única globalmente, isto é, mesmo se uma determinada rotina com mesmo nome aparecer em duas imagens distintas, as duas *cópias* terão *ids* diferentes. Um objeto da classe `Thread_node` possui, por sua vez, além de um ponteiro para o próximo elemento da lista, o *id* da *thread*, que pode ser recuperado através do `THREAD_ID` e o número de instruções que a rotina daquela respectiva *thread* executou.

A criação de nós do tipo `Routine_node` é realizada na rotina de instrumentação por motivos ligados à performance, visto que é um processo extremamente custoso. Tal afirmação reside no fato de que uma rotina pode aparecer mais de uma vez em *threads* distintas e que, por este motivo, é necessário percorrer a lista toda vez a fim de procurá-la. Observa-se também que um programa pode apresentar um número elevado de rotinas - os *benchmarks* testados, por exemplo, apresentam mais de 1000 delas. Dessa forma, caso tais operações fossem inseridas na função de análise, a performance do programa seria afetada muito negativamente. Sendo assim, após a verificação da existência da rotina na lista e a possível criação em caso negativo, inicia-se o processo de posicionamento das funções de análise. Cada estrutura do tipo `RTN` possui uma referência para uma lista das instruções que a compõem, sendo acessada pela função `RTN_InsHead()` e percorrida por `RTN_Next()`. Para cada instrução, executa-se, portanto, a função `INS_InsertCall()` para inserir a *callback* de análise. Essa função

pode receber, além da referência da *callback*, a ordem em que ela é chamada (antes ou depois, por exemplo, da execução da instrução) e um conjunto variável de parâmetros. `PIN` oferece uma série de possibilidades para a passagem de parâmetros à função de análise. No nosso caso, por exemplo, dois argumentos são utilizados, sendo eles o *id* da *thread*, que é passado através de `THREAD_ID`, definido na enumeração `IARG_TYPE`, e o ponteiro do objeto do tipo `Routine_node`, que identifica a qual rotina a instrução pertence. Este último exige a adição de dois parâmetros a `INS_InsertCall()`: um para a especificação do tipo do argumento, isto é, `IARG_PTR`, e o segundo o ponteiro propriamente dito. Para a ordem de execução, quatro opções são disponíveis, sendo elas `IPOINT_BEFORE`, `IPOINT_AFTER`, `IPOINT_ANYWHERE` e `IPOINT_TAKEN_BRANCH`, equivalendo, respectivamente, à execução da rotina de análise antes, depois, em um ponto determinado pelo próprio `PIN` ou se um *branch* foi executado. Como não há nenhuma preferência de ordem e prefere-se aquela que ofereça melhor desempenho, escolhe-se `IPOINT_ANYWHERE`.

Por último, a criação de nós do tipo `Thread_node` é realizada na função de análise juntamente com o incremento do número de instruções executadas. É necessário, porém, a fim de não comprometer siginificamente a execução do programa, implementarmos uma função que execute o mínimo de operações. Neste contexto, acessa-se a lista de `Thread_node`s através do ponteiro do tipo `Routine_node`, passado como parâmetro, e procura-se o nó, cujo *id* seja igual ao segundo argumento da função. Caso ele não seja encontrado, um novo nó é criado. Vale lembrar, portanto, que o desempenho do programa é fortemente prejudicado caso ele possua muitas *threads*. Isso porque a lista ligada de `Thread_node`s é sempre percorrida a cada nova instrução. Por fim, o contador de instruções é incrementado.

Esta `pintool` foi testada em cinco *benchmarks* do `SPEC CPU2006`, sendo eles `400.perlbench`, `401.bzip2`, `403.gcc`, `445.gobmk` e `999.specrand`. A impressão dos resultados é realizada de maneira decrescente. A seguir, são apresentadas algumas considerações importantes a respeito das execuções.

- Todos os *benchmarks* **testados** apresentam apenas uma *thread*, portanto a sua performance não é *extremamente* degradada pela *tool*;
- A execução de `S_regmatch` corresponde a 50% das instruções executadas por `400.perlbench`, com 1.116×10^{12} instruções nas três entradas;
- A routina `BZ2_compressBlock` em `401.bzip2` contribui com 22.2% das instruções executadas, enquanto que `BZ2_decompress` apenas com 12.6%;
- as instruções executadas em `bitmap_operation` valem 9.2% do número de instruções de `403.gcc`;
- A rotina que executa mais instruções em `445.gobmk` é `do_play_move`, com 8.4% do total. Isto indica um grande equilíbrio entre as funções.
- `_GI__printf_fp_` corresponde a 32.6% da execução de `999.specrand`;

3.2. Simulação dos modelos de 1 e 2 *bits* de *branch predictor*

A segunda ferramenta implementada utilizou os recursos do `pin` para avaliar a performance de dois *branch predictors*, sendo eles de 1 e dois *bits*. Para isso, três novas classes foram implementadas. A primeira, `GenericBP`, é abstrata e serve como modelo para as demais classes, que devem implementar o método `update`, que recebe como parâmetro um endereço de 32 *bits* e recalcula os estados do preditor. A segunda classe, `BP1bit`, implementa um preditor de um *1 bit*, isto é, que utiliza apenas o resultado do último evento para a sua atualização. Por fim, a classe `BP2bit` define o modelo de *2 bits* que utiliza uma máquina com 4 estados possíveis. A instrumentação é realizada a cada instrução de pulo ou chamada de função, sendo

implementada através das funções `INS_IsDirectBranchOrCall`, capaz de retornar se uma instrução atende a essa condição, e `INS_InsertCall`, já discutida anteriormente. Três parâmetros são passados à função de análise, sendo eles o endereço PC da instrução de pulo ou chamada, seu endereço alvo e se o pulo foi realizado. Tais parâmetros são especificados através de, respectivamente, `INS_Address`, `INS_DirectBranchOrCallTargetAddress` e `IARG_BRANCH_TAKEN` durante a chamada de `INS_InsertCall`.

A execução desta *pintool* nos mesmos *benchmarks* da subseção anterior permitiu a avaliação dos dois modelos. A tabela 1 mostra os resultados para os dois modelos de *branch predictor*. *E* corresponde ao número de entradas de cada *benchmark*, *Misses* é a somatória do número de *miss-predictions* de todas as entradas, *Hits* a somatória das quantidades em que o preditor acertou e, enfim, % é a razão $\frac{Hits}{Hits+Misses}$. Tais campos são as médias dos resultados de todas as entradas.

Tabela 1. Resultados obtidos para os modelos *1-bit* e *2-bit*.

		Modelo 1			**Modelo 2**		
Benchmark	*E*	*Misses*	*Hits*	*%*	*Misses*	*Hits*	*%*
`400.perlbench`	3	4.0×10^{10}	9.4×10^{10}	0.70	2.7×10^{10}	1.1×10^{11}	0.80
`401.bzip2`	6	9.1×10^{9}	5.4×10^{10}	0.86	1.3×10^{10}	5.0×10^{10}	0. 79
`403.gcc`	9	4.7×10^{9}	2.3×10^{10}	0.83	4.9×10^{9}	2.3×10^{10}	0. 82
`445.gobmk`	5	2.0×10^{10}	4.3×10^{10}	0.68	1.8×10^{10}	4.5×10^{10}	0.72
`999.specrand`	1	3.9×10^{7}	6.4×10^{7}	0.62	4.7×10^{7}	5.9×10^{7}	0.58

Observa-se que:

- O modelo 2 foi superior para os *benchmarks* `400.perlbench` e `445.gobmk`, e inferior para os demais, provando que tais programas não apresentam comportamento desequilibrado, isto é, não permanecem nos estados *fortes* de tal modelo;
- O desempenho foi inferior para o *benchmark* que apresenta menos instruções, isto é, `999.specrand`;

4. Conclusões

Neste relatório, explorou-se algumas funcionalidades do `pin`, aliado com a execução dos *benchmarks* contidos no `SPEC CPU 2006`. Foi possível avaliar, portanto, quais *benchmarks* utilizaram mais ou menos instruções durante sua execução. Além disso, foi possível verificar que alguns deles rodaram mais de uma vez com entradas diferentes, uma vez que o número de instruções variou. Adicionalmente, implementou-se duas *pintools* para extender as funcionalidades dos exemplos dados na página de referência da ferramenta e para explorar os conceitos de *branch prediction* discutidos em aula. Estas novas *pintools* exploraram, diversos recursos da API e são capazes, respectivamente, de imprimir as rotinas de cada *thread* em ordem decrescente, permitindo, assim, o estudo das rotinas que mais impactam a aplicação avaliada, e de contar o número de vezes que os modelos de 1 e 2 *bits* foram bem-sucedidos na predicação na execução dos *benchmarks*.

Referências

Intel (2012). Pin 2.11 user guide.

SPEC (2011). Spec cpu2006: Read me first.