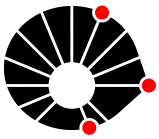

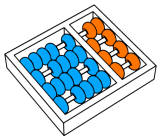

SimPoints aplicado a múltiplas entradas

Luis Fernando Antonioli *Rodolfo Azevedo*

Relatório Técnico - IC-PFG-17-05
Projeto Final de Graduação
2017 - Julho

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

SimPoints aplicado a múltiplas entradas

Luis Fernando Antonioli* Rodolfo Azevedo*

Resumo

Compreender o comportamento a nível de ciclos de um processador executando um programa é vital para a pesquisa moderna de arquitetura de computadores. Visando obter essa informação, simuladores detalhados geralmente são utilizados. A simulação completa de um *benchmark* padrão pode demorar semanas ou até meses para ser concluída. Para endereçar esses problemas, técnicas estatísticas tais como a metodologia *SimPoint* foram propostas. A metodologia *SimPoint* tenta identificar fases repetitivas e encontrar um conjunto pequeno de amostras da execução do programa que representa a maior parte da execução do programa, ou seja, busca prever alguma propriedade da arquitetura baseando-se na execução individual de amostras das fases do programa. Arquiteturas podem ser comparadas simulando o seu comportamento nas amostras de código selecionadas pelo *SimPoint* para rapidamente determinar que arquitetura tem o melhor desempenho. A metodologia *SimPoint* realiza a análise das fases de cada par programa-entrada separadamente. Neste trabalho estudamos a metodologia *SimPoint*, propomos e implementamos uma extensão dela para que permita a análise de fases de um programa para várias entradas visando assim tentar diminuir o número total de *SimPoints* necessasários para simular um *benchmark* inteiro.

1 Introdução

Pesquisas na área de Arquitetura de computadores geralmente requerem o entendimento detalhado do comportamento de um processador durante a execução de um programa.

Muitos programas tem comportamentos bem distintos durante partes de sua execução que denominamos fases. Durante um momento podem usar intensamente a memória, em outros podem sofrer bastante com erros no preditor de desvios.

Para obter esse nível de informação, pesquisadores geralmente utilizam simuladores que modelam cada ciclo executado. Infelizmente esse detalhamento da simulação trás consigo penalidades no tempo de simulação o que acarreta que *benchmarks* utilizados pela indústria demorem meses para serem executados completamente.

Agravando ainda mais a situação do tempo de simulação, geralmente é necessário simular um mesmo *benchmark* inúmeras vezes até que os pesquisadores encontrem uma configuração

*Instituto de Computação, Universidade Estadual de Campinas (UNICAMP), Campinas, SP

de arquitetura que tenha um bom equilíbrio entre consumo, desempenho, complexidade e área, ou seja, muitas vezes um mesmo par de programa-entrada é simulado várias vezes para que possa ser examinado a diferença no desempenho que a mudança no tamanho de uma *cache* pode trazer para uma determinada arquitetura.

Este problema não deixou de ser notado pela comunidade acadêmica, e muitos pesquisadores desenvolveram técnicas que buscam reduzir o tempo de simulação [1]. Uma das técnicas propostas para resolver esse problema é chamada *SimPoint* [2, 3, 4] que inteligentemente escolhe um conjunto de amostras do programa chamada de *Simulation Points* para realizar a simulação do programa, provendo um desempenho preciso da execução completa do programa.

A metodologia *SimPoint* utiliza algoritmos de agrupamento para automaticamente encontrar padrões repetitivos na execução de um programa. Simulando apenas um representante de cada fase do programa, o tempo de simulação pode ser reduzido a minutos ao invés de semanas implicando apenas em um perda pequena de precisão.

Um ponto chave da metodologia *SimPoint* é que os *Simulation Points* escolhidos pela técnica são independentes da arquitetura utilizada para a simulação permitindo assim que o mesmo conjunto de *Simulation Points* possa ser utilizado para a simulação de diversas configurações de arquitetura.

Para que a escolha dos *Simulation Points* seja independente da arquitetura, foi proposto em [5] o conceito do perfilamento do programa utilizando uma estrutura chamada BBV (*Basic Block Vectors*) para permitir uma maneira de capturar comportamentos importantes de um programa durante sua execução.

O BBV é um vetor onde cada posição representa um bloco básico do programa. Cada elemento do vetor guarda a quantidade de vezes que um determinado bloco básico foi executado durante um intervalo e como não estamos interessados no valor absoluto de cada posição do vetor e sim na proporção da execução de cada bloco básico, normalmente o BBV é normalizado dividindo cada elemento pela soma de todos elementos do vetor. Essa normalização garante que a soma de todos os elementos do vetor seja 1, fazendo com que a comparação de BBV de intervalos de tamanhos diferentes seja possível.

Dessa forma se guardarmos o BBV de cada trecho do programa, podemos capturar a frequência relativa dos blocos de código executados durante uma dada parte da execução de um programa.

Encontramos muitos trabalhos na literatura acadêmica propondo técnicas na análise de fases. Esses trabalhos aplicam a análise de fases para cada par programa-entrada separadamente. Neste trabalho temos por objetivo estender a técnica do *SimPoints* para que possa fazer essa análise para um conjunto de entradas de um mesmo programa visando tentar otimizar o caso de simulação de um *benchmark* inteiro.

2 Trabalhos Relacionados

Dentre os trabalhos relacionados existem aqueles que buscam reduzir o tempo de simulação utilizando ou não análise de fases e aqueles que utilizam a análise de fases para outros fins que não são necessariamente a redução do tempo de simulação.

Em [6] Wunderlich et al. propõem uma abordagem chamada SMARTS (*Sampling Microarchitecture Simulation*) que aplica a teoria de amostragem estatística para endereçar problemas na simulação de amostragens. Diferentemente de outras abordagens antes dele, ele descreve um procedimento exato e construtivo para selecionar um subconjunto minimal do fluxo de execução de instruções de um *benchmark* para se atingir um determinado intervalo de confiança. Dessa forma é possível saber qual o número de amostras necessárias para se obter o intervalo de confiança desejado.

Em [7] Girbal et al. apresentam um esquema distribuído de simulação onde para diminuir o tempo de simulação, é realizada distribuição de partes da simulação para N processadores diferentes. É interessante notar que, no esquema proposto, todas as instruções do programa são executadas e para mitigar os problemas de acurácia vindos da divisão da simulação para vários processadores é utilizado um mecanismo automático e dinâmico pra ajustar o tamanho do intervalo de *warm-up*. Nos resultados mostrados pelos autores é visto que a aceleração da execução é escalável em respeito a quantidade de recursos de computação disponíveis trazendo em média um ganho de velocidade de 7,35 vezes utilizando 10 máquinas com um erro médio do CPI de 1,81%

Análise de fases nem sempre tem por objetivo a redução de tempo de simulação. Em [8] os autores discutem a caracterização das fazem de um programa com foco no consumo de energia.

Existem vários trabalhos na academia relacionados a metodologia *SimPoints*. Dentre eles podemos destacar [2] onde os autores descrevem a metodologia em alto nível, [9] onde é discutido uma modificação na técnica dos *SimPoints* para que exista um ganho de desempenho através da seleção de *Simulation Points* que aparecem mais cedo na execução do programa e assim diminuem o tempo de emulação que é necessário até chegar nos *Simulation Points*. Em [10] os autores validam a técnica do *SimPoints* utilizando o *benchmark SPEC CPU 2006* e em [11] os autores discutem a técnica quando aplicada em múltiplos binários para um mesmo programa.

3 Objetivos

A metodologia *SimPoint* encontra automaticamente um pequeno conjunto de *Simulation Points* que representam a completa execução de um programa para uma dada entrada para uma simulação precisa e eficiente. Dessa forma a análise realizada pela metodologia é feita para cada par programa-entrada separadamente.

Neste trabalho buscamos estender a metologia para que realize as análises de múltiplas entradas de um mesmo programa ao mesmo tempo, buscando assim encontrar *Simulation Points* que sejam representativos para mais de uma entrada.

Se um mesmo *Simulation Point* é utilizado para caracterizar mais de uma entrada, diminuímos o número total de *Simulation Points* necessários para simular um conjunto de entradas de um programa sem perdermos a capacidade de simularmos cada entrada individualmente. Dessa forma obtemos as métricas de *hardware* como CPI, *cache miss*, *branch misprediction* e etc, para cada entrada individualmente. Ao reduzirmos o número de Simulation Points necessarios para simular um conjunto de entradas reduzimos também

o tempo de simulação necessário para simularmos todas as entradas.

4 Ferramentas

Apresentaremos em seguida algumas ferramentas e métodos que são de extrema importância para o esse trabalho.

4.1 Análises de fases do programa

A maneira como os programas se comportam durante sua execução geralmente não são aleatórias. Muitos estudos [12, 8] mostraram que geralmente os programas entram em comportamentos repetitivos chamados de fases.

Os autores de [12] definem fases como o conjunto de intervalos (ou fatias no tempo) dentro da execução de um programa que tem comportamento similar, independentemente de adjacência temporal.

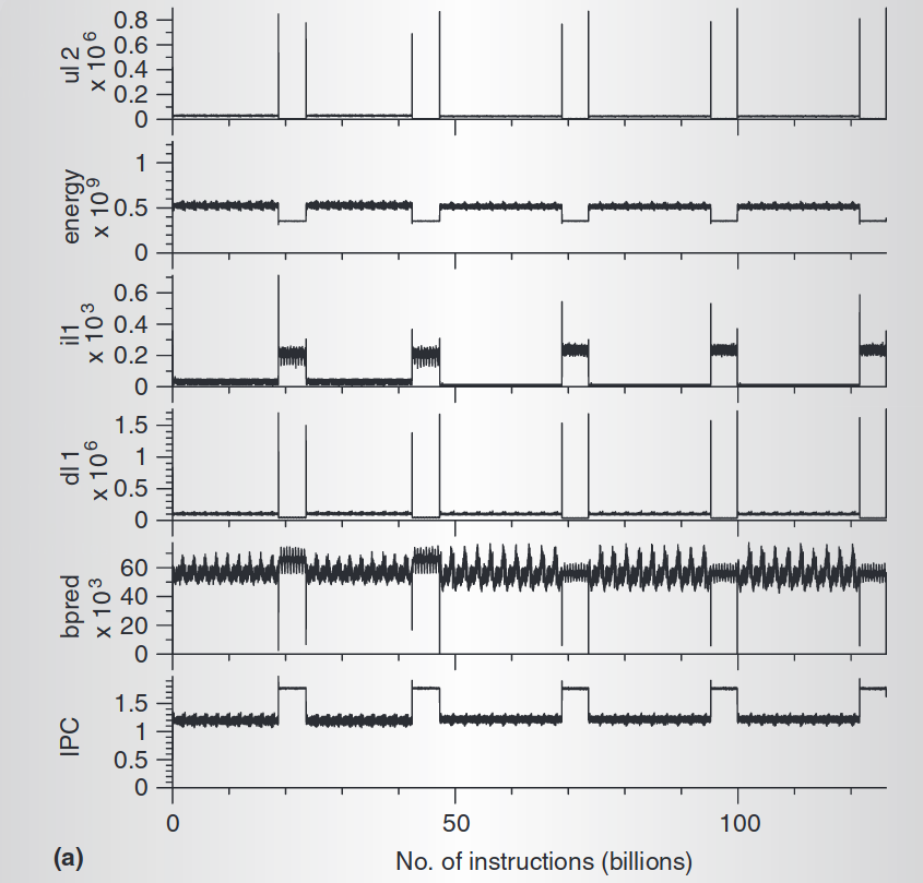

Figura 1: Gráfico de algumas métricas sobre bilhões de instruções executadas pelo programa gzip com uma entrada gráfica. Imagem retirada de [12]

A figura 1 mostra a variação de algumas métricas como CPI, gasto de energia e outras durante a execução programa BZIP. Nela podemos ver claramente um comportamento de fases no programa que inclusive se repete ao longo do tempo.

Uma observação chave que torna o estudo de fases de um programa importante é que qualquer métrica de um programa é função direta da forma que um programa passeia pelo código durante a execução [12]. Dessa forma, é possível encontrar fases de um programa examinando apenas proporções de que regiões do código estão sendo executadas através do tempo.

Uma maneira fácil de coletar esse tipo de informação é através da utilização de *basic block vectors* e posteriormente aplicar algum algorítimo de agrupamento para identificar *basic block vectors* semelhantes que correspondem a intervalos que gastaram quase a mesma proporção de seu tempo em regiões iguais e portanto deveriam pertencer a mesma fase.

4.2 SimPoint

SimPoint é uma metodologia para identificar porções representativas de um programa. Na metodologia, a execução de um programa é dividida em intervalos de iguais números de instruções. Uma fase de um programa é o conjunto de intervalos que possuem um comportamento semelhante. O objetivo é encontrar um intervalo representativo ou um *Simulation Point* de cada uma das fases.

Para cada intervalo, um BBV é coletado. O algoritmo do *SimPoint* então agrupa os vetores de blocos basicos (BBV) utilizando o algoritmo K-means [13]. Em seguida são encontrados os intervalos que são os centroides de cada grupo encontrado pelo algoritmo de agrupamento e esses intervalos centroides são chamados de *Simulation Points*.

Como pode se perceber as fases de um programa tem tamanhos diferentes e como cada um desses *Simulation Points* encontrados representam uma fase do programa, naturalmente cada *Simulation Point* tem uma parcela diferente na composição do comportamento do programa inteiro.

Dessa maneira, caso se deseje prever qual o valor de CPI do programa inteiro simulando-se apenas os *Simulation Points* é necessário que o valor de CPI de cada *Simulation Point* seja ponderado por um peso que seja a proporcional ao tamanho da fase que ele representa. No *SimPoint* esse peso associado a cada *Simulation Point* é calculado como sendo a razão entre o número de instruções pertencentes àquele agrupamento de onde ele veio e o número total de instruções do programa.

4.3 Pin

Pin [14] é uma ferramenta de instrumentação de binários dinâmica para os conjuntos de instruções IA-32, x86-64 e MIC que permite a criação e ferramenta de análise de programas.

Ele é um software proprietário desenvolvido pela Intel e é disponibilizado gratuitamente para fins não comerciais. O Pin é especialmente importante para nosso trabalho pois ele é a base do PinPlay, que será descrito mais adiante e, através de sua API, permite construirmos diversas ferramentas para a extração de dados importantes na execução de um programa, como por exemplo a construção dos *basic block vectors* discutidos anteriormente.

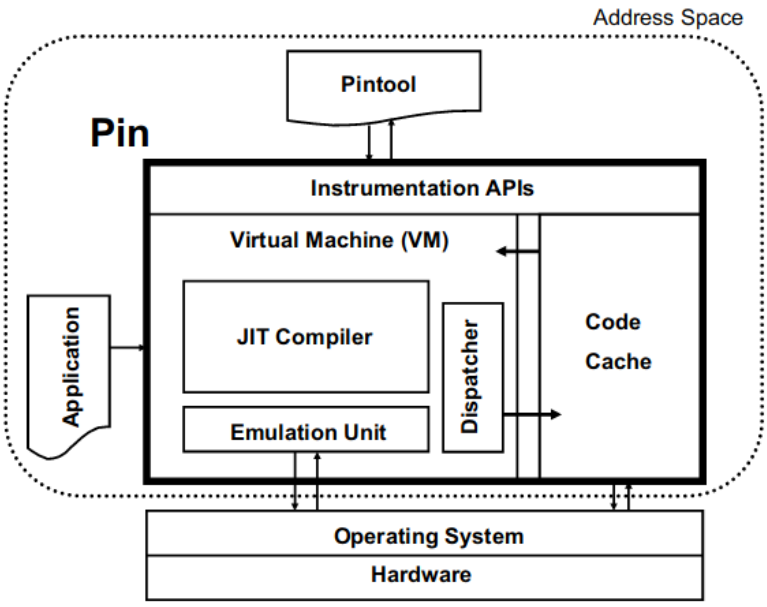

Figura 2: Arquitetura de *software* do Pin. Imagem retirada de [15]

Na figura 2 vemos a arquitetura de software do Pin. Note que ele é composto por um compilador JIT (*Just-In-Time compiler*) e uma máquina virtual para permitir a instrumentação dinâmica do programa. A *Pintool* é a ferramenta que deve ser implementada para permitir a instrumentação desejada do código.

4.4 PinPlay

PinPlay [16] é uma ferramenta que permite a captura e repetição determinística da execução de um programa que permite a análise de grandes programas em um tempo razoável. Ele é baseado no Pin e estende as funcionalidades do Pin através da disponibilização de uma pintool para a captura da execução de um programa (gerando arquivos de *log* chamados de *pinballs* e disponibilizando outra pintool que permite que outras ferramentas baseadas no Pin executem essas *pinballs* como se estivessem executando o binário original. Como a execução é registrada em *pinballs*, execuções das *pinballs* são determinísticas e assim garantem a reprodutibilidade da execução do programa. Essa funcionalidade é muito importante pois permite que as ferramentas de análise executem um programa diversas vezes e tenham sempre a mesma execução, mesmo no caso de programas com múltiplas *threads* e chamadas ao sistema operacional.

Na figura 3 temos ilustrado a interação do PinPlay com o Pin. Note que a pinball é autocontida e ela está atrelada a uma execução específica do programa e não ao binário do programa. Esse é o motivo de não precisarmos das entradas do programa para repetirmos sua execução

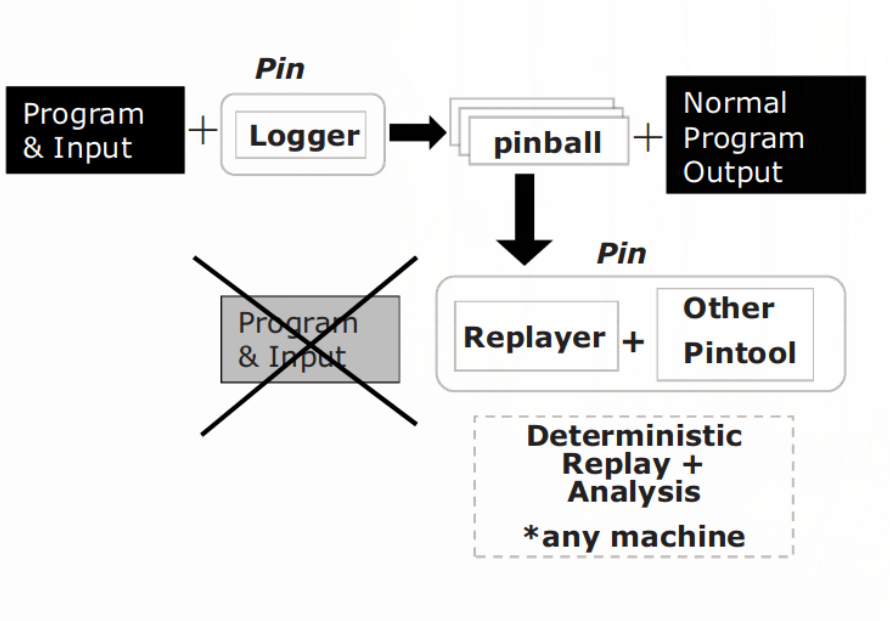

Figura 3: Interação entre o PinPlay e o Pin. Imagem retirada de [16]

4.5 PinPoints

PinPoints é o resultado da junção da técnica *SimPoint* com a ferramenta PIN. Em [17] os autores descrevem como as combinaram. A metodologia *SimPoint* utiliza um perfil de execução para identificar regiões representativas de uma aplicação. Essas regiões, também chamadas de *Simulation Points* são validadas por sua vez contra o comportamento do programa inteiro. Os autores descrevem que a metodologia é composta dos seguintes passos:

1. Coleta do perfil do programa utilizando uma ferramenta baseada no Pin
2. Análise do perfil do programa utilizando a metodologia *SimPoint* para encontrar regiões representativas. Essas regiões são denominadas *PinPoints* pelos autores.
3. Comparação do comportamento dos PinPoints em relação ao comportamento do programa inteiro utilizando o Pin e o *pfmon* [18]

Nosso objetivo é construir uma ferramenta com funcionamento semelhante ao *PinPoints*, entretanto que permita o uso de nosso método de *SimPoints* que analisa múltiplas entradas ao mesmo tempo.

4.6 Sniper

Sniper [19] é um simulador de x86 paralelo e de alto desempenho. Sua importância para esse trabalho é que ele nos permite simular programas inteiros e regiões (*Simulation Points*) para assim podermos comparar os erros de predição realizados por nosso método nas métricas de arquitetura (como o CPI por exemplo).

A escolha do simulador Sniper for feita por ele ser construído utilizando o Pin.

5 Desenvolvimento do trabalho

Nesse trabalho implementamos uma extensão da metodologia *SimPoints* onde nosso objetivo é analisar múltiplas entradas de um mesmo programa ao mesmo tempo para permitirmos agrupar fases equivalentes na execução dessas entradas a fim de diminuir o número de intervalos representativos, i.e. *Simulation Points*.

Para tanto, implementamos uma infraestrutura para testar nossa proposta. Abaixo listamos os passos executados por nossa infraestrutura para a execução da técnica proposta.

1. Para cada entrada do programa, é utilizado o PinPlay para registrar sua execução. Ao final dessa etapa é gerado um *Pinball* para cada entrada.

2. Cada *Pinball* gerado na etapa anterior é executado utilizando o PinPlay para garantir que não houveram falhas no processo de registro da execução

3. Através do uso do PinPlay junto com o Pin, a execução das *Pinballs* da primeira etapa é instrumentada para produzir um arquivo para cada entrada contendo os BBVs dos intervalos daquela entrada. Esses arquivos são nomeados $t_n.bb$ onde n é o número da entrada

4. Os arquivos arquivos $t_n.bb$ gerados no passo anterior são unificados em um único arquivo $t.bb$ apenas. Nesse passo, o desafio é conseguir indexar o BBV de forma que uma determinada posição no vetor de blocos básicos represente o mesmo bloco básico do programa para todas as entradas

   Note que esse problema não existe na metodologia original, visto que na metodologia original só é necessário que exista a consistência de atribuição de identificadores para os blocos básicos entre intervalos do mesmo programa enquanto que agora essa consistência também tem que ocorrer entre múltiplas entradas.

   Outro fator que agrava o problema é que o endereço inicial do bloco básico não pode ser utilizado para identifica-lo unicamente, visto que em cada execução o programa pode ser carregado em lugares diferentes da memória, bem como as bibliotecas dinâmicas podem estar em lugares diferentes.

   Dessa maneira, para identificar um bloco básico unicamente, é utilizado a distância do endereço inicial do bloco básico e o inicio do binário no qual ele se encontra, acompanhado de qual binário aquele bloco básico esta contido, já que é possível a utilização de vários binários e bibliotecas num programa.

5. Depois de unificados os arquivos contendo os BBVs de todas as entradas, é utilizado uma projeção aleatória para reduzir para 15 dimensões os vetores de blocos básicos. Esse passo é necessário pois os vetores de blocos básicos possuem milhares de dimensões e caso não fosse feito iria acarretar num aumento significativo no tempo de execução do algoritmo de agrupamento do passo seguinte.

6. Nessa etapa é utilizado o algoritmo de agrupamento K-means para agrupar os intervalos semelhantes (lembre-se que cada BBV representa um intervalo de execução de uma entrada). Como o algoritmo recebe como parâmetro o K, que é quantos grupos ele vai retornar, é realizado uma busca binária para encontrar o menor K que tenha ao menos 90% da qualidade do K máximo, que por sua vez é uma entrada do nosso método.

7. Após os intervalos serem agrupados, o intervalo mais próximo do centro do grupo (o centroide) é escolhido como representante daquele grupo. Além disso também é necessário saber o quão importante aquele grupo é para a execução de cada entrada. Seja $w_{i,j}$ a importância do grupo i para a execução da entrada j, calculamos $w_{i,j}$ como a razão entre o número de intervalos da entrada j dentro do grupo i pelo número total de intervalos da entrada j. Dessa forma ao final dessa etapa sabemos quais intervalos representam cada grupo i e qual importância $w_{i,j}$ o grupo i tem para a execução da entrada j.

8. Com a informação de quais intervalos são os representantes de cada grupo, é utilizado novamente o *PinPlay* para repetir a execução gravada de cada entrada e registrar cada intervalo representante em uma *Pinball* diferente para permitir que cada intervalo seja executado individualmente quando necessário.

9. Esses últimos passos são opcionais mas foram implementados pois permitem a avaliação da qualidade dos *Simulation Points* escolhidos. Nesse passo, para cada entrada, é utilizado o simulador *Sniper* para simular os *Simulation Points* (*Pinballs* geradas no passo anterior) e, através da utilização dos valores $w_{i,j}$ citados anteriormente, é possível realizar a predição do comportamento das métricas do programa. Por exemplo, seja J o conjunto de todas as entradas e I o conjunto de todos os *Simulation Points*, podemos calcular o CPI para a entrada $j \in J$ como:

$$CPI_{preditoj} = \sum_{i \in I} w_{i,j} * CPI_i$$

10. Novamente utilizando o *Sniper* é simulada a execução completa de cada entrada e então assim é possível saber qual é o valor real de cada métrica desejada.

11. Por fim é feito uma comparação entre o valor real e o predito pela técnica a fim de comparar-se desempenho.

6 Resultados

Para testarmos nossa técnica realizamos uma série de experimentos utilizando programas do *benchmark SPEC CPU 2006* [20] [21]. Abaixo é mostrado o resultado de alguns experimentos.

6.1 Variação no erro da predição do CPI em função escolha do representante de cada fase

Neste experimento é comparado o erro na predição do CPI quando escolhemos como representante o intervalo mais próximo do centro do grupo (centroide) e quando escolhemos como representante o intervalo que foi executado mais cedo, ou seja um intervalo que pode estar em qualquer ponto do grupo. Com esse experimento queremos saber se o erro na predição é muito dependente da escolha do representante de um grupo ou se a variação entre a escolha de um representante causa um impacto pequeno.

Para realizar o teste, é realizado a predição do CPI para as 8 entradas do programa GCC do *SPEC CPU 2006* e os resultados são ilustrados na figura 4.

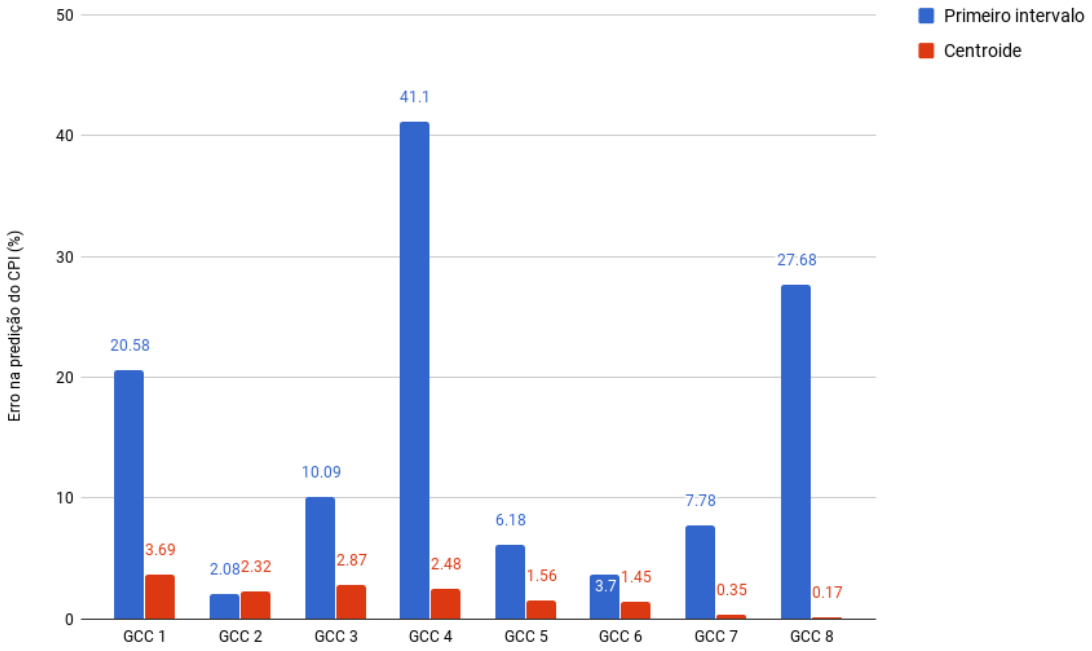

Figura 4: Variação do erro da predição do CPI: centroide vs primeiro intervalo do grupo

Como podemos ver, as os erros mudam muito com a troca do representante no grupo, indicando que a metologia *SimPoints* é muito sensível a escolha do representante dentro do grupo.

6.2 Comparação entre o número de Simulation Points do método de múltiplas entradas e o original

Nesse experimento é realizada a execução do método SimPoints de múltiplas entradas e do original para cinco programas do SPEC CPU 2006. Ao final é sumarizado o número de *Simulation Points* necessários para simular todas as entradas de cada programa. Na figura

5 temos ilustrados os resultados

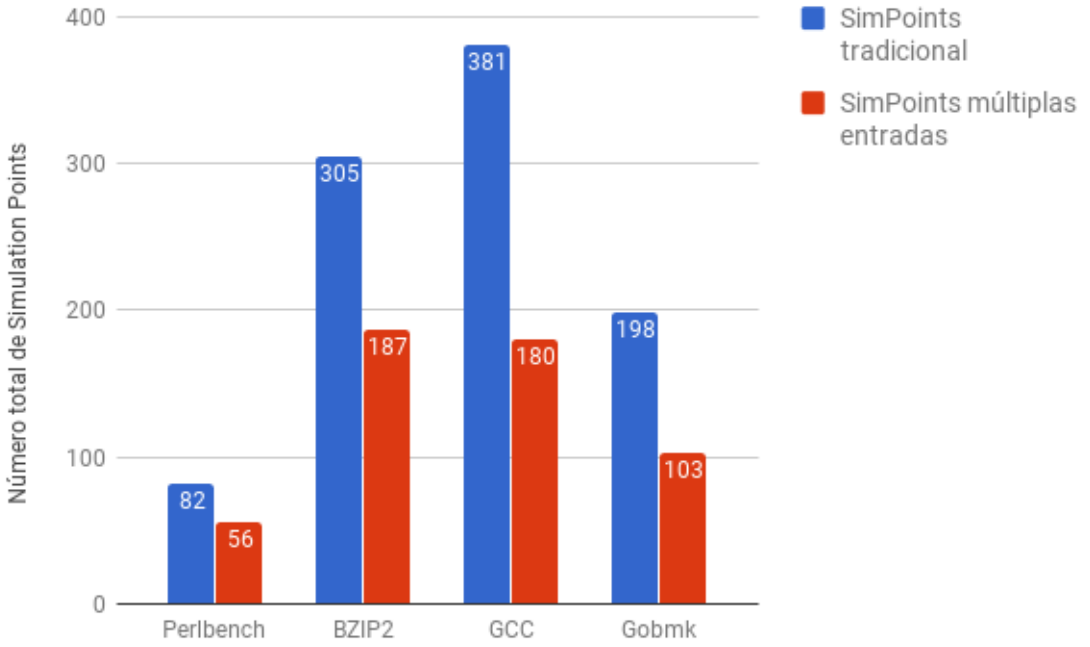

Figura 5: Comparação entre o número de Simulation Points gerados pela técnica SimPoint de múltiplas entradas e a original

É possível ver que o número de *Simulation Points* necessários é bem menor na técnica proposta. Isso acarreta numa diminuição de tempo de execução de todas as entradas de um determinado programa em uma mesma proporção que a diminuição de *Simulation Points*.

6.3 Comparação entre os erros dos método

O sucesso da técnica *SimPoint* não vem apenas da diminuição do tempo de execução, mas também do fato dos erros gerados por essa diminuição serem pequenos. Nas figuras 6, 7, 8 e 9 temos a comparação dos erros para cada entrada dos programas *perlbench*, *bzip*, *gcc* e *gobmk* respectivamente.

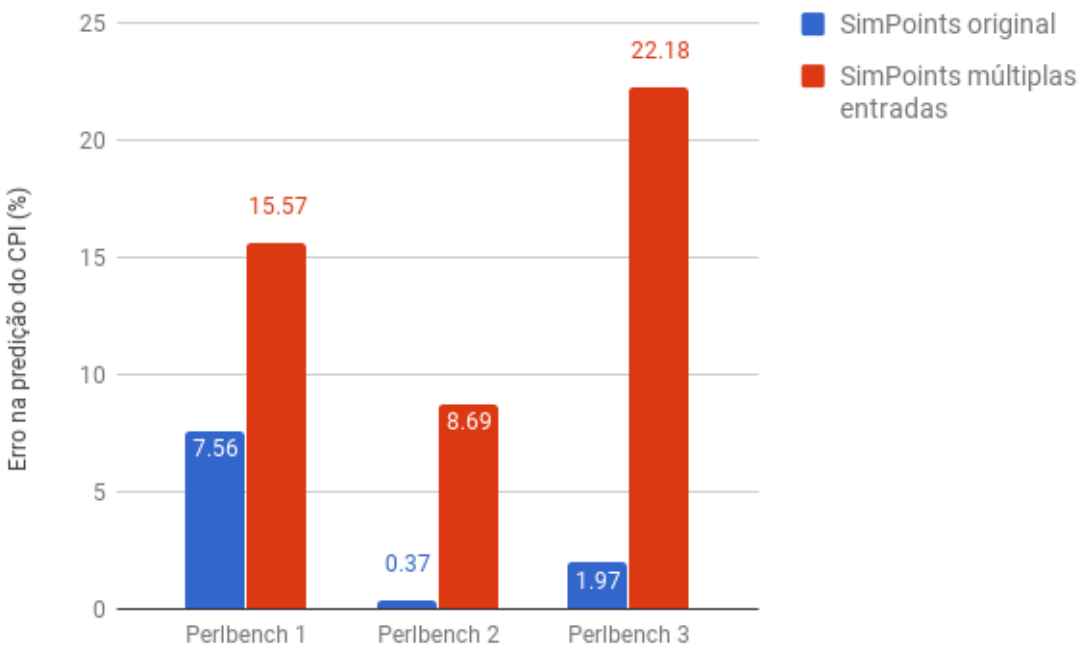

Figura 6: Variação do erro da predição do CPI para diferentes entradas do programa perlbench

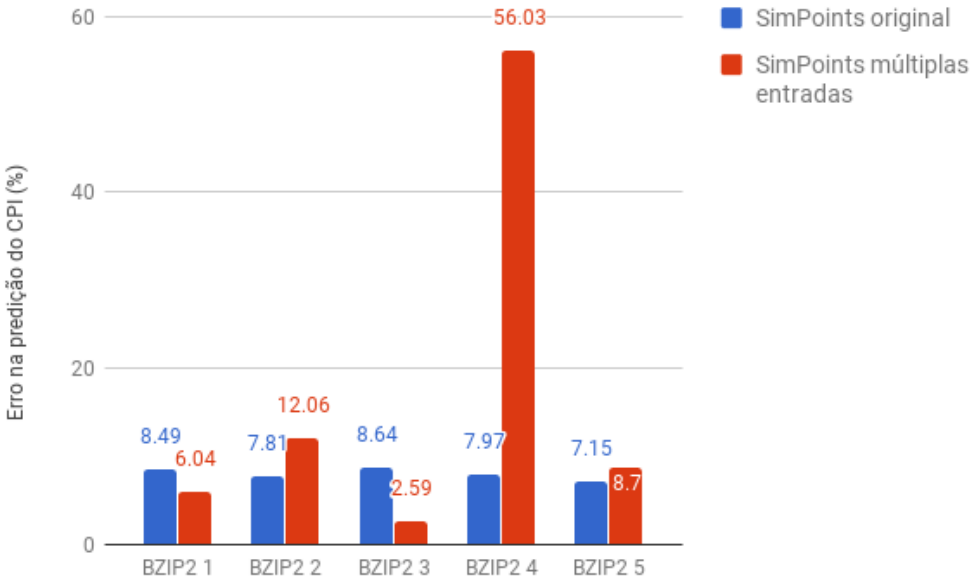

Figura 7: Variação do erro da predição do CPI para diferentes entradas do programa bzip

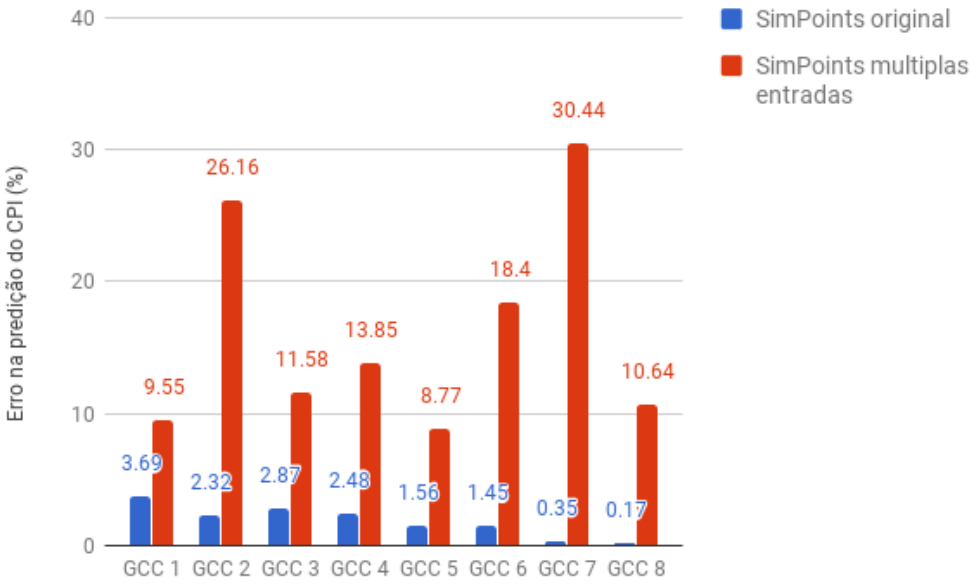

Figura 8: Variação do erro da predição do CPI para diferentes entradas do programa gcc

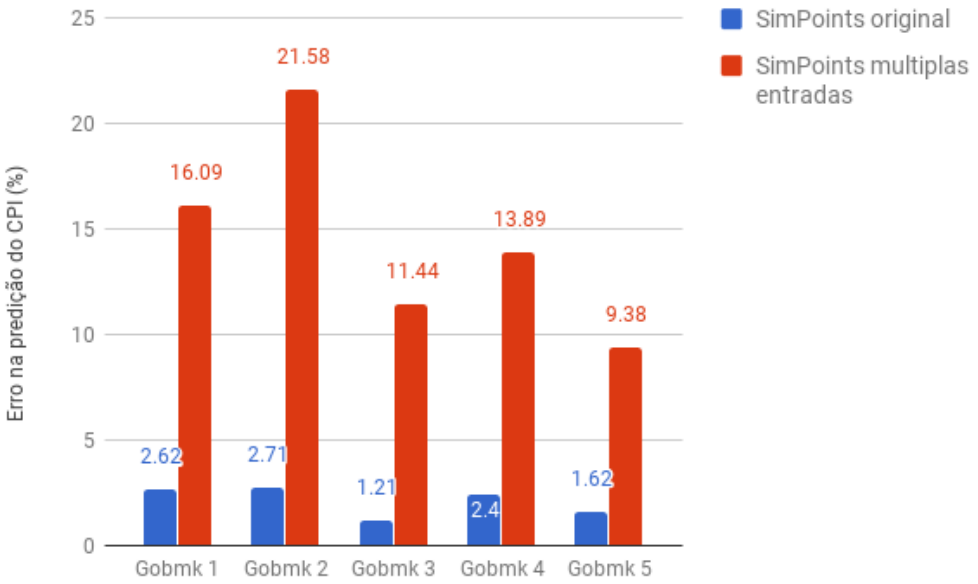

Figura 9: Variação do erro da predição do CPI para diferentes entradas do programa gobmk

Essas figuras mostram um erro em média maior para nosso método do que para o origi-

nal. Acreditamos que isso possa ter ocorrido pela distribuição de intervalos nos grupos não ocorrer de forma homogênea fazendo com que os representantes de cada grupo não correspondessem aos centroides dos grupos quando se é analisado cada entrada separadamente. Como vimos anteriormente, a escolha dos representantes é muito sensível a distância do representante ao centro do grupo.

7 Conclusões

Nesse trabalho foi apresentados uma extensão da técnica dos *SimPoints* com o intuito de explorar fases semelhantes em execuções de entradas distintas. Também pudemos implementar e testar a extensão proposta, comparando seus resultados com a técnica original.

Através desse trabalho concluímos que apesar de existirem redundâncias de fases entre diferentes execuções de entradas de um mesmo programa, elas não necessariamente estão agrupadas de forma homogêneas nos grupos encontrados quando analisadas sob a ótica dos vetores de blocos básico. Para explorar essa redundância, outras ideias ligadas ao algoritmo de agrupamento ou a caracterização dos intervalos possivelmente são necessárias para diminuir os erros de predição e tornar o método mais útil.

Os resultados na diminuição de *Simulation Points* também confirmaram nossa motivação de que existe bastante similaridade entre fases de execuções de um mesmo programa com entradas distintas e que essas similaridades podem realmente diminuir o tempo de simulação de um programa e um *benchmark* inteiro.

Referências

[1] Lieven Eeckhout. Computer architecture performance evaluation methods. *Synthesis Lectures on Computer Architecture*, 5(1):1–145, 2010.

[2] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using simpoint for accurate and efficient simulation. In *ACM SIGMETRICS Performance Evaluation Review*, volume 31, pages 318–319. ACM, 2003.

[3] Greg Hamerly, Erez Perelman, and Brad Calder. How to use simpoint to pick simulation points. *ACM SIGMETRICS Performance Evaluation Review*, 31(4):25–30, 2004.

[4] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism*, 7(4):1–28, 2005.

[5] Timothy Sherwood, Erez Perelman, and Brad Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*, pages 3–14. IEEE, 2001.

[6] Roland E Wunderlich, Thomas F Wenisch, Babak Falsafi, and James C Hoe. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 84–95. IEEE, 2003.

[7] Sylvain Girbal, Gilles Mouchard, Albert Cohen, and Olivier Temam. Dist: A simple, reliable and scalable method to significantly reduce processor architecture simulation time. In *ACM SIGMETRICS Performance Evaluation Review*, volume 31, pages 1–12. ACM, 2003.

[8] Canturk Isci and Margaret Martonosi. Phase characterization for power: evaluating control-flow-based and event-counter-based techniques. In *HPCA*, volume 3, pages 121–132, 2006.

[9] Erez Perelman, Greg Hamerly, and Brad Calder. Picking statistically valid and early simulation points. In *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on*, pages 244–255. IEEE, 2003.

[10] Arun A Nair and Lizy K John. Simulation points for spec cpu 2006. In *Computer Design, 2008. ICCD 2008. IEEE International Conference on*, pages 397–403. IEEE, 2008.

[11] Erez Perelman, Jeremy Lau, Harish Patil, Aamer Jaleel, Greg Hamerly, and Brad Calder. Cross binary simulation points. In *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 179–189. IEEE, 2007.

[12] Timothy Sherwood, Erez Perelman, Greg Hamerly, Suleyman Sair, and Brad Calder. Discovering and exploiting program phases. *IEEE micro*, 23(6):84–93, 2003.

[13] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[14] Vijay Janapa Reddi, Alex Settle, Daniel A Connors, and Robert S Cohn. Pin: a binary instrumentation tool for computer architecture research and education. In *Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture*, page 22. ACM, 2004.

[15] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

[16] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 2–11. ACM, 2010.

[17] Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi. Pinpointing representative portions of large intel® itanium® programs with dynamic instrumentation. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, pages 81–92. IEEE, 2004.

[18] David Mosberger and Stephane Eranian. *IA-64 Linux kernel: design and implementation*. Prentice Hall PTR, 2001.

[19] Trevor E Carlson, Wim Heirmant, and Lieven Eeckhout. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12. IEEE, 2011.

[20] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[21] Arun A Nair and Lizy K John. Simulation points for spec cpu 2006. In *Computer Design, 2008. ICCD 2008. IEEE International Conference on*, pages 397–403. IEEE, 2008.