Análise de padrões de acesso à memória em aplicações single threaded *

Gabriel d A. S. Evaristo[1], **Sairo R. dos Santos**[1], **Marco A. Z. Alves**[1]

[1]Departamento de Informática – Universidade Federal do Paraná (UFPR)

{almeidasalesgabriel,sairoraoni}@gmail.com, mazalves@inf.ufpr.br

Resumo. *Esse trabalho busca avaliar os padrões de acesso à memória utilizando as aplicações do SPEC CPU-2017, buscando especificamente observar a regularidade dos acessos quanto aos endereços e seus saltos. O objetivo é entender o comportamento dominante nas aplicações para propor melhorias no simulador de arquiteturas e nos componentes arquiteturais, tendo como simulador alvo o Ordinary Computer Simulator (OrCS).*

1. Introdução

Em pesquisas de arquiteturas de computadores, os simuladores baseados em traços são bastante utilizados. Tais simuladores utilizam traços de execução de aplicações para estimar o comportamento de uma determinada arquitetura ao executar a aplicação em questão. Tais traços de simulação são gerados durante a execução real de uma aplicação, onde são colhidas informações como instruções executadas, endereços dos acessos à memória, resultados de instruções de salto, etc. Devido ao grande número de instruções e informações associadas, traços de simulação podem se tornar arquivos na ordem de *Giga* ou *Terabytes*. Assim, os traços em si se tornam um problema de pesquisa.

O OrCS é um simulador inspirado no *Simulator of Non-Uniform Caches* (Si-NUCA) [Alves et al. 2015], baseado em traços. Os traços de simulação do OrCS são compostos por três arquivos: **(i) O traço estático** tem tamanho fixo e trás uma tradução das instruções (código *assembly*) de um programa dividida em blocos básicos. **(ii) O traço dinâmico** tem tamanho variável, pois depende dos parâmetros de entrada. Ele indica a ordem dos blocos básicos na execução real, consumindo assim, pouco espaço de armazenamento. **(iii) O traço de memória** é o maior pois além de ter tamanho variável, conforme o parâmetro de entrada, ele descreve cada acesso à memória feito pela aplicação. A descrição dos acessos à memória inclui mais informações, como o tipo de acesso (R/W), o tamanho, o endereço de memória e o bloco básico que gerou tal acesso.

Este trabalho busca caracterizar o comportamento dos acessos à memória de aplicações recentes explorando alternativas para diminuir o tamanho dos arquivos de traço de memória, utilizando para isso a carga de trabalho SPEC CPU-2017.

2. Trabalhos correlatos

O trabalho de A. R. Pleszkun [Pleszkun 1994] apresenta uma técnica de compressão que explora a localidade espacial e temporal dos acessos. No entanto, os autores relacionam os endereços observando os possíveis sucessores de um acesso, visando encontrar padrões e assim substituí-los por alguma representação traduzível para uma sequência de endereços.

*Este trabalho foi parcialmente suportado pelo Instituto Serrapilheira (grant número Serra-1709-16621).

A. Milenkovic et al. [Milenkovic and Milenkovic 2003] apresentam o algoritmo *Stream-Based Compression* (SBC), que cria um dicionário com os blocos básicos encontrados no traço. A compressão se dá pela troca dos blocos observados pelo seu índice no dicionário. Cada bloco contém informações de seu endereço inicial, tamanho e instruções. Os endereços dos acessos à memória são substituídos pelos *strides*, a diferença entre os endereços de dois acessos consecutivos feitos pela mesma instrução. Um contador é utilizado para agrupar os *strides* iguais.

3. Estratégia de caracterização

Em geral, acessos à memória por parte de aplicações seguem determinados padrões que podem ser observados em várias instâncias. Se identificamos acessos à memória idênticos ou que sigam um comportamento bem definido, temos o potencial de omitir essas linhas do traço de memória, economizando assim espaço de armazenamento.

Para uma determinada instrução que acessa a memória, a única variação possível de ocorrer, para cada repetição dela, é o endereço de memória acessado. Assim, ao determinar como o endereço de memória de uma determinada instrução evolui durante a execução do programa, temos potencial de compactar o traço de memória. Podemos por exemplo, estabelecer uma regra para a produção dos endereços subsequentes à um endereço inicial observando os saltos (*strides*) entre os endereços acessados.

4. Mecanismo de monitoramento

Para calcular os *strides* dos acessos à memória, precisamos considerar que o OrCS presume instruções CISC, assim, cada instrução pode gerar até três acessos à memória, sendo estes duas leituras e uma escrita (chamadas respectivamente de *read*, *read2* e *write*). Dessa maneira, consideramos as seguintes informações sobre cada instrução do traço: o endereço da instrução, o primeiro e último endereços de memória acessados, o número de ocorrências da instrução durante a execução e status final da instrução.

Para esse cálculo, criamos um mecanismo inspirado no design do *stride prefetch* [Fu et al. 1992]. O mecanismo consiste basicamente em uma tabela, preenchida durante a execução dos traços de simulação de cada programa do *benchmark*. Para cada instrução distinta no traço criamos uma linha nessa tabela cujos campos são: o endereço da instrução, uma sub-tabela com informações do primeiro acesso possível que a instrução pode fazer (*read*), uma com informações do segundo possível (*read2*), uma do terceiro acesso possível (*write*), outra com informação dos acessos sem fazer distinção, o número de vezes que a instrução apareceu durante a execução e o status final da instrução. As subtabelas possuem os seguintes campos: o primeiro endereço acessado, o último endereço acessado, o último *stride* visto e o status final do acesso. O status é usado para caracterizar o acesso, ele pode ser linear ou não-linear. Um acesso linear possui um *stride* constante.

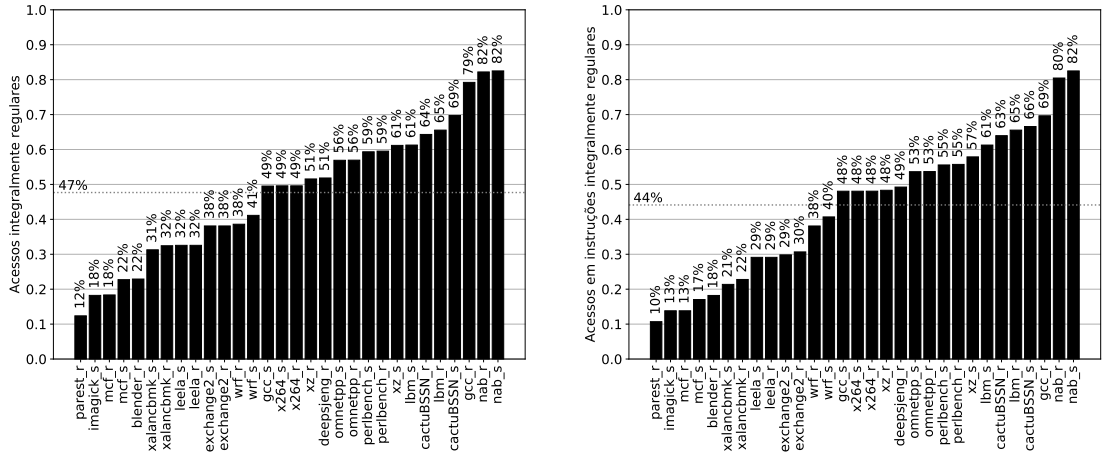
Durante a simulação, a cada instrução a tabela é atualizada, assim, os status associados às instruções variam durante a simulação, no final da simulação nos interessa observar quais instruções/acessos possuem *stride* linear. Porém, podemos caracterizá-los de duas maneiras: instruções/acessos que estão com status linear ao final da simulação e instruções/acessos que estiveram com status linear durante um período da execução. Apresentamos resultados apenas para a primeira caracterização, essa escolha é suficiente para uma análise inicial, visto que estamos utilizando traços derivados de *pinpoints*, os

acessos contidos não são provenientes de fases da aplicação que apresentam um comportamento regular que não seja representativo da totalidade do programa.

Considerando que algumas instruções complexas podem gerar até três acessos de memória (*read*, *read2* e *write*) pensamos em duas maneiras de calcular os *strides*: na primeira, calculamos *strides* entre os mesmos acessos de uma instrução (*read* da instrução i com o próximo *read* da instrução i, *read2* com *read2* e *write* com *write*); na segunda, continuamos calculando *strides* entre os acessos da mesma instrução, porém, não fazemos distinção entre *read*, *read2* e *write* (acesso qualquer da instrução i com o próximo acesso qualquer da instrução i).

5. Experimento e resultados

Para nossos experimentos utilizamos as aplicações do SPEC CPU-2017. Cada aplicação foi analisada durante a execução de uma fatia de dois bilhões de instruções mais representativas que foi selecionada utilizando *Pinpoints* [Patil et al. 2004]. Todos os *benchmarks* foram compilados para x86-64 usando GCC versão 7.5.0 com a opções *O3* e *march=native*.



(a) Acessos integralmente regulares. (b) Acessos em instruções integralmente regulares.

Figura 1. Porcentagens de acessos integralmente regulares.

A Figura 1 (a) apresenta no eixo vertical a porcentagem de acessos integralmente regulares, isto é, que mantêm *stride* constante até o fim da simulação e no eixo horizontal as aplicações da carga de trabalho de teste. Os dados dessa figura consideram o mecanismo calculando os *strides* de cada tipo de acesso feitos pela mesma instrução independentemente. Como apontado pela linha tracejada, em média 47% dos acessos têm *stride* constante, ou seja, são bem comportados ou redundantes e podem ser descritos de maneira mais compacta. A Figura 1 (b) ilustra os resultados do mecanismo calculando os *strides* entre todos os tipos de acesso, *read*, *read2* e *write*, de cada instrução. O eixo vertical representa a porcentagem de acessos em instruções integralmente regulares. Como esperado, misturar os tipos de acesso de uma mesma instrução reduz a regularidade dos acessos (para 44% em média). Sendo que a grande maioria dos acessos regulares foram feitos por instruções que sempre acessavam o mesmo endereço.

A Figura 2 apresenta o tamanho dos traços de memória de algumas aplicações do *benchmark* SPEC CPU-2017. A altura total das barras representa o tamanho atual dos traços e o tamanho das porções mais escuras representam o tamanho esperado após a compressão. Obtemos essa estimativa de redução assumindo que substituindo todas as ocorrências de endereços com *strides* constantes por uma regra de formação de endereços subsequentes à partir de um endereço inicial, eliminamos todos os bytes de endereços de acessos redundantes, ou seja, um traço de memória com 80% de acessos regulares sofreria uma redução de cerca de 80%. Como o ato de descomprimir se dará durante a simulação, criamos uma troca entre o tamanho da compressão e o tempo de simulação.

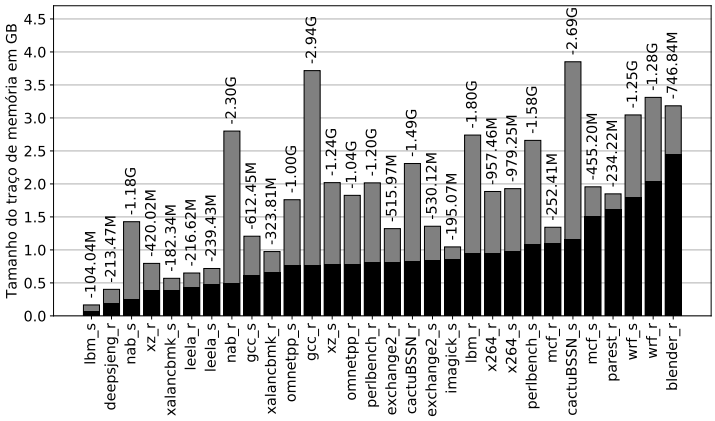


Figura 2. Tamanhos dos traços de memória e reduções esperadas

6. Conclusões e trabalhos futuros

Este trabalho apresentou o comportamento das 30 aplicações do SPEC CPU-2017, mostrando que quase 50% das instruções de memória dessas aplicações possuem um comportamento repetitivo de *stride* nos acessos à memória. Como próximo passo pretendemos implementar o mecanismo de compressão dos traços de memória e o de descompressão. Com os resultados que obtivemos podemos estimar a redução no tamanho dos traços de memória. Uma comparação sobre o *tradeoff* entre tempo de compressão/descompressão e espaço de armazenamento também está prevista entre os trabalhos futuros.

Referências

Alves, M. A. Z., Villavieja, C., et al. (2015). Sinuca: A validated micro-architecture simulator. In *Int. Conf. on High Performance Computing and Communications*.

Fu, J. W., Patel, J. H., and Janssens, B. L. (1992). Stride directed prefetching in scalar processors. *ACM SIGMICRO Newsletter*, 23.

Milenkovic, A. and Milenkovic, M. (2003). Stream-based trace compression. *IEEE Computer Architecture Letters*, 2(1):4–4.

Patil, H., Cohn, R., et al. (2004). Pinpointing representative portions of large intel® itanium® programs with dynamic instrumentation. In *Int. Symp. on Microarchitecture*.

Pleszkun, A. R. (1994). Techniques for compressing program address traces. In *Int. Symp. on Microarchitecture*.