T2 Org Arq II

Guido Mainardi
Lucas Félix
Pedro Wagner

[1]Pontifícia Universidade Católica de Rio Grande do Sul

Abstract. *O trabalho tem por objetivo explorar conceitos discutidos em sala de aula sobre hierarquia de memória desenvolvendo uma ferramenta através da qual poderá caracterizar programas a serem executados, a configuração da cache L1 e as características do restante da hierarquia de memória.*

1. Apresentação do problema

O problema consiste em conseguir observar o gerenciamento de dados vindo de um pseudocódigo em uma memória cache explorando juntamente a hierarquia de memória. Esse pseudocódigo tem três comandos básicos:

- Números de instruções (deve ser obrigatoriamente a primeira linha);
- Saltos condicionais de uma instrução X para uma instrução Y, junto com a probabilidade do salto ocorrer;
- Saltos incondicionais de uma instrução X para uma instrução Y.

Sobre a hierarquia de memória ela será configurada por um arquivo de texto. Por padrão ela terá quatro níveis: Cache L2, Cache L3, Memória RAM e Hard Disk (HD), essa hierarquia será feita do mesmo jeito que as instruções da memória, cada nível terá uma probabilidade de possuir a informação que a cache L1 necessita, ou seja, sempre que a memória cache L1 der miss ela procurará no primeiro nível da hierarquia e será feito um cálculo de "sorte"para saber se a informação está, caso não esteja ela descerá mais um nível e fará o mesmo processo até achar o que precisa, obviamente que a probabilidade da informação estar no último nível é de cem por cento. Cada nível também tem um certo custo que será computado no final do algoritmo. Quanto mais a cache L1 desce na hierarquia, mais custoso será a informação.

Para resolver o problema será criado dois algoritmos principais: O primeiro serve para consumir as informações dos arquivos textos de hierarquia e do pseudocódigo, e criar uma memória pronta para ser usada no segundo algoritmo, além de gerar outro arquivo texto com o trace (que é basicamente a memória após ser carregada).

O segundo algoritmo será a memória cache L1 em si, onde será possível ver com detalhes aonde as informações estão alocadas em cada instrução. Nesse algoritmo há outras sub-rotinas como a política selecionada pelo usuário, por exemplo.

2. Solução do problema

2.1. Solução do Trace

O trace gerado como um arquivo texto é apenas uma cópia da memória após ter sido carregada pelo "Memoria.java". A memória do algoritmo é um map de inteiro para uma tupla, na tupla o primeiro valor é a instrução, em decimal, que será avançada e o segundo

valor é a probabilidade dessa instrução ser executada, para todos os casos essa chance é cem por cento, exceto para os saltos condicionais que tem um valor menor. Certamente que o tamanho da memória é o "ep"passado na primeira linha no pseudo código.

2.2. Solução da cache

Com a solução do trace devidamente funcional há uma memória pronta para uso da cache. O usuário pode passar os parâmetros para a criação da cache, assim como a seleção da política a ser usada caso a cache esteja cheia. A duas políticas disponíveis para uso:

- Política Aleatória: Caso seja necessário colocar novas informações na cache será escolhido aleatoriamente uma linha de palavras já existentes na cache para ser substituída por essas novas informações.
- Política LFU: Caso seja necessário colocar novas informações na cache será verificado qual das linhas da cache foi menos utilizada, essa linha de palavras será substituída por essas novas informações.

Uma vez configurada a cache, a classe "Processador" irá ler a memória para executar o programa. Por o pseudocódigo ter a possibilidade de haver algum laço com uma probabilidade muito baixa de sair, o processador lerá no máximo 5000 instruções por execução do programa.

Para cada instrução o processador gera um número aleatório para a probabilidade de um possível salto condicional; lê a instrução; recebe um hit ou um miss da cache L1; armazena os custos em variáveis para a exibição final para o usuário.

Para cada instrução da memória o processador chama o método de consumo de informação da classe "Cache", esse método inicia separando os bits de tag, conjunto e palavra do endereço com operações de bitwise; em seguida ele calcula em qual conjunto é necessário realizar a busca e percorre esse conjunto atrás da tag desejada, caso ele ache a tag e seu bit de validade seja 1 ele enviará as informações para o processador, acusando a operação como um hit; caso ele não consiga realizar essa operação ele acusará um miss e a própria cache acionará o método de busca da instrução.

Quando não há a informação na cache o método miss começa utilizando a subrotina "montaLinha" para saber quais são as palavras próximas que devem ser levadas para a cache junto com a informação necessária (tentando prever que provavelmente o processador irá utilizá-las); sabendo quais são as palavras a cache puxa-as pela hierarquia já guardando junto o custo que foi levado para encontrá-las (pois a variável "enderecos_carregados" é uma tupla, onde o primeiro elemento é o custo e o segundo elemento são as palavras em si); em seguida novamente é feito com operações de bitwise a definição da tag e conjunto que serão associados às palavras; a partir dai será feita uma busca por algum bit de validade em 0, caso tenha algum todas as informações serão postas naquele endereço. No mesmo laço será calculado quais são as instruções menos utilizadas, caso não haja bit de validade em 0 significa que a cache está cheia e as instruções precisarão ser salvas com o auxílio de uma política. Caso a política seja a aleatória será, obviamente, escolhida uma linha ao acaso daquele conjunto para ser substituída pelas informações novas e caso seja a LFU será utilizada a linha menos utilizada; sabendo onde colocar as informações o bit de validade e o número de vezes que essas palavras foram utilizadas serão atribuídos com 1.

3. Resultados e Conclusões

Quando as soluções estavam devidamente completas e corretas começamos a realizar casos de testes com cada política para observar qual seria mais otimizada. Nos primeiros casos criamos algo bem simples como no pseudocódigo [1], onde há apenas dois saltos, sendo eles um condicional com uma probabilidade bem baixa de acontecer e o salto sendo bem próximo para que a chance dos dados necessários para o salto ainda estivessem na cache.

```
ep:100
ji:50:40
bi:45:51:10
```
(1)

Para nossa surpresa a política aleatória foi bem mais eficaz do que a LFU, o que fez com que fosse criada dúvidas sobre a veracidade do código. O gráfico da figura [1] mostra a performance da cache com as diferentes políticas. Cada ponto desse e todos os gráficos abaixo é o custo total por instrução de uma execução, ou seja, pontos altos em uma grande quantidade indicam uma política muito custosa e portanto menos vantajosa; o tamanho de cada amostra é dez mil (assim havendo vinte mil pontos em cada gráfico). Também em todos os gráficos usamos a cor vermelha para se referir a política LFU e o rosa para a aleatória.

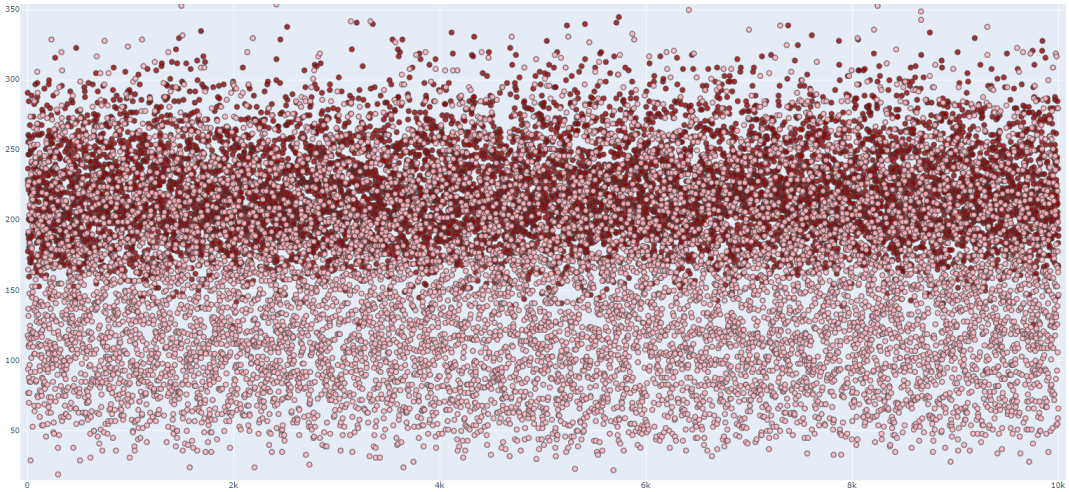

Figura 1. Resultados da amostra 1

O segundo caso de teste foi com o mesmo pseudocódigo citado anteriormente, porém com os atributos da cache modificados, nós aumentamos o número de palavras por linha com o intuito de melhorar a performance da política LFU. Os resultados apontam que nossa hipótese estava correta, porém mesmo com a melhora da política LFU, para esse caso a performance com a política aleatória ainda foi ligeiramente superior como foi analisado na figura [2].

O terceiro e último teste foi utilizado o pseudocódigo [2]. O intuito dele era ter um laço guardado na memória que fosse reutilizado durante todo o programa, assim estimulando uma boa performance com a política LFU.

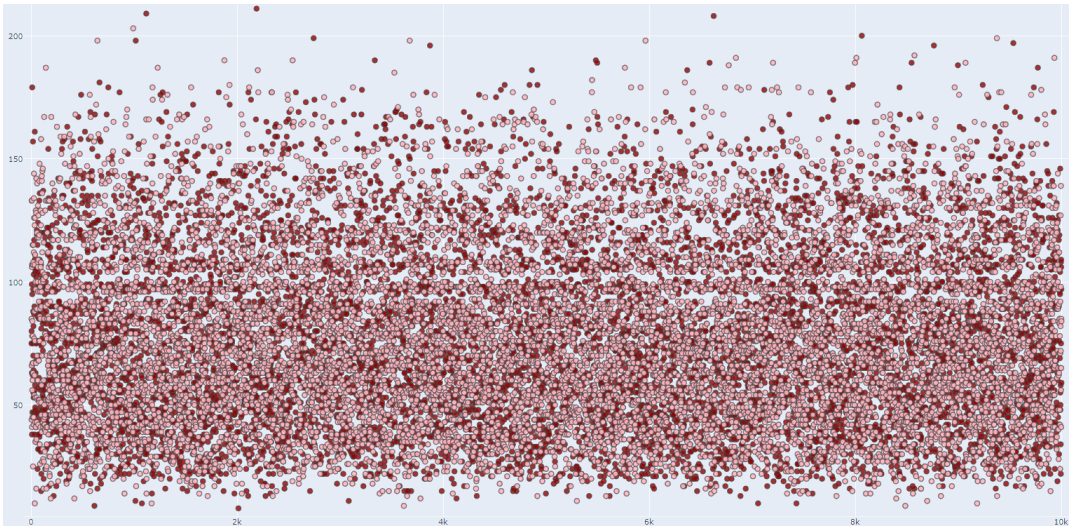

Figura 2. Resultados da amostra 2

```
ep:100
bi:10:3:60
bi:75:2:95
```
(2)

Os resultados na figura [3] apontam que nossa nova hipótese estava correta mesmo que com a política aleatória ainda tenha uma boa performance, também pode-se afirmar que o LFU é aparentemente mais estável nesse caso uma vez que há muito mais pontos "soltos" na nuvem da política aleatória, além de que sua nuvem é mais espessa ao contrário da nuvem de pontos do LFU que se encontra mais uniforme e compacta.

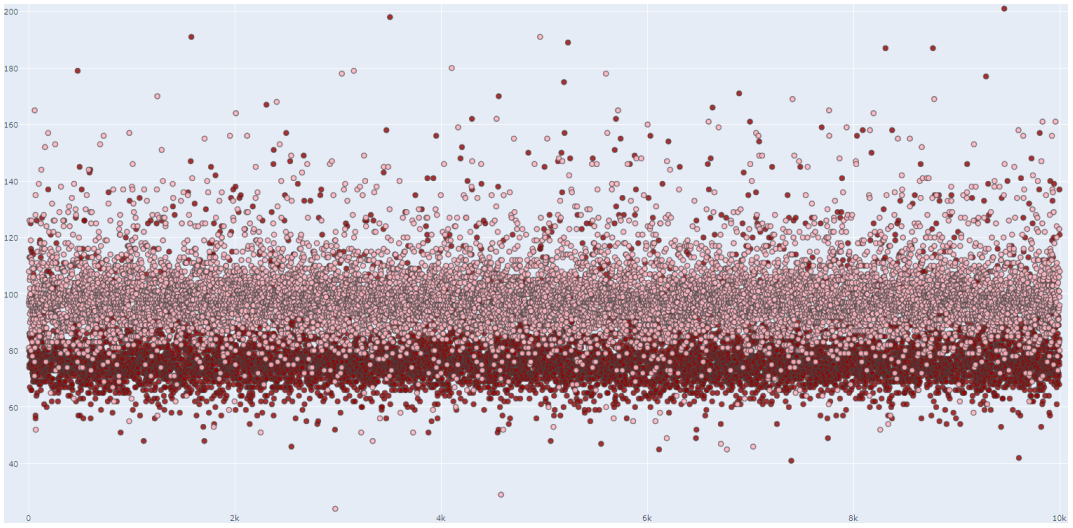

Figura 3. Resultados da amostra 3

Portanto, é possível inferir que a política aleatória normalmente possui uma melhor performance para casos simples com poucos e pequenos laços. Caso haja códigos mais extensos e complexos a política de LFU aparentemente seria mais interessante. Também pode-se afirmar que as configurações gerais da cache influenciam em um ga-

nho de desempenho, não adiantando ter uma memória cache muito grande e ter uma taxa relativamente grande de miss, por exemplo; confirmando o que Monteiro[Monteiro 2012] e Stallings[Stallings 2010] comentam em seus livros sobre tamanho de memórias cache.

Referências

[Monteiro 2012] Monteiro, M. A. (2012). *Introdução à Organização de Computadores*. Livros Tecnicos e Cientificos.

[Stallings 2010] Stallings, W. (2010). *Arquitetura e Organização de Computadores*. Pearson Pratice Hall.