Questões técnicas envolvidas no simulador de *cache*

Nicolas Zachow Coelho[1], **Romeu Zimmer Luz**[1]

[1]Universidade Federal de Ciências da Saúde de Porto Alegre (UFCSPA)
Porto Alegre – RS – Brasil

{nicolasco,romeulu}@ufcspa.edu.br

Resumo. *Este artigo é resultado de um trabalho proposto na disciplina de Arquitetura de Computadores, o trabalho tinha como objetivo desenvolver um simulador de memória* cache*. Com isto em mente, neste artigo abordamos superficialmente a teoria que dá suporte à* cache *e voltamos nossa atenção mais para as questões técnicas envolvidas no desenvolvimento deste simulador. Além disto no final nós discutimos brevemente sobre as questões relativas ao desempenho e deixamos algumas ideias para trabalhos futuros nesta mesma linha.*

1. Introdução

Este é um trabalho que foi desenvolvido no contexto da disciplina de Arquitetura de Computadores, durante o primeiro semestre de 2016. Ele tinha por objetivo uma simulação rudimentar do funcionamento de uma memória *cache* que utiliza o LRU como algoritmo de substituição. Neste artigo a nossa intenção é apresentar uma breve revisão teórica sobre memória *cache* na seção 2. Na seção 3 discutiremos sobre detalhes da implementação do simulador. Em sequência, na seção 4, faremos alguns testes comparativos para analisar de que forma a *cache* pode afetar o desempenho. E finalmente, apresentaremos algumas conclusões e ideias para novos estudos.

2. Memória cache

Cache é definida como uma memória pequena e extremamente rápida, que tem como objetivo armazenar parte do conteúdo de uma memória maior e mais lenta. O raciocínio que motiva a utilização desta arquitetura é que ao evitar o acesso à memória principal se ganha muito desempenho. Os dois principais fatores que contribuem para o sucesso da cache são as localidades *temporal* e *espacial*. A localidade *temporal* nos diz que informações que foram utilizadas recentemente têm uma alta probabilidade de serem utilizadas novamente. Já a localidade *espacial* diz que referências futuras têm alta probabilidade de estarem próximas das últimas referências.

A memória *cache* já foi extensivamente discutida na literatura técnica durante o passar dos anos e por isso acreditamos que para quem deseja ter um bom embasamento teórico neste assunto a melhor alternativa é buscar em outras fontes. Para facilitar a vida de quem estiver interessado nisso deixamos aqui, sem qualquer ordem de relevância ou preferência, alguns recursos que nós consideramos de grande utilidade: [Hill 1987], [Peir et al. 1998], [Di Carlo et al. 2011] e [Mano 1982].

3. Desenvolvimento

3.1. Definição da simulação

O nosso primeiro grande desafio neste trabalho foi justamente definir exatamente qual era o nível de detalhamento que estávamos buscando. O primeiro passo foi definir a quantidade de unidades de processamento que seria possível simular com o nosso software, como na definição do trabalho não havia nenhuma restrição neste aspecto, escolhemos a saída mais fácil e definimos que nossa simulação teria apenas um único *core*. Isso eliminou a preocupação com problemas de concorrência entre processadores, caso tivéssemos escolhido tratar este problema, o nosso software apresentaria uma complexidade muito maior devido às questões de controle envolvidas.

Depois de termos definido que teríamos apenas um processador, ficou claro para nós que não havia sentido criar uma simulação com múltiplos níveis de *cache* e com *write-through* como política de escrita à memória principal. Assim ficou definido que nossa simulação teria apenas um nível de *cache* e utilizaria *write-back* como política de escrita à memória. Além disto, havia a imposição de que o algoritmo de substituição deveria ser o *LRU* e com isso estavam descartas as possibilidades de se utilizar outros métodos como o aleatório ou um algoritmo adaptativo. Com tudo isto definido, a última decisão importante a ser tomada era a respeito da associatividade que a simulação teria. Como não havia nenhuma imposição de perfomance na definição do trabalho, escolhemos simular um modelo completamente associativo.

Outra questão importante sobre o funcionamento do simulador que não foi definida na especificação do trabalho é sobre a forma de simular. Basicamente existem duas opções quando se fala neste tipo de simulação. A primeira, e mais simplista, seria gerar requisições de acesso à memória de maneira aleatória. A segunda, um pouco mais realista, é utilizar um *trace* de execução de um programa. Este *trace* é um arquivo que contém todas as referência feitas a memória enquanto um determinado código estava executando. O problema é que esta técnica não resolve todos os problemas de falta de realismo, já que segundo [Smith 1994] a maior parte dos *cache misses* acontece na troca de contexto do sistema operacional. E para simular este tipo de comportamento deveriam ser utilizados vários *traces* simultaneamente com um algoritmo de escalonamento mas isto não foi feito. Por este motivo o nosso simulador pode apenas ser comparado a um sistema monoprocessado.

Além disso existe a questão do *cacheline*, que pode ser definido grosseiramente como a quantidade de endereços que são trazidos a *cache* por operação. Isto é feito porque computacionalmente é o mesmo custo trazer apenas um endereço ou um bloco de endereços. Assim isso é utilizado para colocar informação na *cache* antes que ela seja de fato necessária. Como qualquer coisa isso possui prós e contras. Por um lado você pode antecipar a demanda de endereços mas, por outro, você pode acabar tirando da cache informação que seria útil.

3.2. Implementação da simulação

Uma vez que o escopo de nossa simulação estava totalmente definido o próximo passo foi nos debruçarmos sobre as técnicas da implementação, como por exemplo e escolha da linguagem de programação. A escolhida foi *Python*, e esta escolha foi relativamente fácil

de fazer já que esta linguagem apresenta várias facilidades quando comparada, por exemplo ao *C*. A vantagem desta escolha se torna aparente quando conseguimos implementar o algoritmo de *LRU* utilizando um tipo nativo da linguagem em poucas linhas. Para fazer o mesmo em *C*, por exemplo, seria necessário mais astúcia do que nós tínhamos para oferecer.

Depois de definir a linguagem a ser utilizada partimos para a modelagem da nossa simulação. Como *Python* é uma linguagem que suporta mais de um paradigma de programação isso nos deu a liberdade necessária para modelar uma solução que fosse altamente fiel ao que havíamos nos proposto mas que ao mesmo tempo o código não se tornasse complexo demais. Por fim chegamos à conclusão de que a melhor forma de modelar este problema seria com a utilização de conceitos de orientação a objetos. Desta forma definimos que haveriam três classes. *CPU*, *MMU* e *MemoryUnit*, todas elas relacionadas de alguma forma mas sem nenhuma relação de herança. A classe *CPU* é a responsável por abrir o arquivo de *trace* e passar as referências para a classe *MMU*, que por sua vez é responsável por repassar as operações para a cache de acordo com a necessidade. Caso tivéssemos optado por uma simulação com vários níveis de cache, é nesta classe que toda lógica para o gerenciamento destes diversos níveis deveria ficar. E, finalmente, temos a classe *MemoryUnit* que foi criada para abstrair a implementação da *cache*. Ela, de maneira automática, toma conta de contar todas as referências feitas e também de manter a pilha do *LRU* atualizada corretamente.

Sobre os detalhes de implementação acreditamos que exitem alguns que valem a pena serem discutidos, e começaremos discutindo sobre algumas *features* que a linguagem nos proporciona e que são fundamentais para esta implementação. A mais importante de todas, provavelmente, é a possibilidade de sobrecarregar operadores(*operator overload*). É isto que nos permite contar os acertos e manter a pilha do *LRU* sempre atualizada sem a necessidade de grande esforço. Para fazer isto precisamos apenas escrever um método chamado *__contains__()* que deverá conter toda a lógica necessária. Assim, sempre que perguntarmos se um determinado endereço está em *cache*, utilizando o operador *in*.

Já para manter um registro correto dos endereços utilizados recentemente foi criada uma pilha onde os endereços recentemente utilizados são colocados no topo e sempre que é necessário abrir espaço para novos endereços o primeiro elemento é retirado. Na prática este método é difícil de implementar, principalmente porque toda vez que acontece um *hit* é necessário fazer uma busca na pilha, encontrar o endereço utilizado e colocá-lo novamente no topo da pilha. Nesta implementação usamos uma combinação de três métodos da lista(*pop()*, *index()* e *append()*). Outra questão que consideramos importante mencionar é que as informações sobre os endereços ocupados e também o bit de validade são armazenadas em um dicionário(*dict*). Nós adotamos esta solução porque com esta estrutura nós podemos armazenar o endereço e o seu bit de validade "no mesmo lugar" e, além de diminuir a complexidade, isso acaba gerando uma melhora de desempenho *i.e.* menos código sendo executado.

Como não desejamos tornar este texto desnecessariamente longo, optamos por não abordar os demais detalhes da implementação. Entretanto todo o código fonte deste simulador está disponível para acesso em um repositório do GitHub(https://github.com/nikozc/cachesimpy) para que as questões que não foram abor-

dadas aqui também possam ser esclarecidas por quem assim desejar.

4. Resultados das simulações

Discutiremos agora os resultados de algumas simulações realizadas utilizando o simulador descrito neste artigo e de que maneira os resultados estão de acordo com a nossa intuição inicial sobre o desempenho da memória cache. Em tempo, gostaríamos de fazer uma observação sobre a apresentação dos resultas a seguir, em virtude do espaço limitado que nós temos disponível optamos por não colocar nenhum gráfico. Entretanto nós acreditamos que os gráficos são uma ferramenta muito boa para visualização dos resultados e é por isso que no mesmo repositório onde se encontra o código deste simulador, também estão os gráficos com as resultados das simulações aqui mencionadas.

4.1. Tamanho da *cache*

Esta simulação foi feita para avaliar os efeitos na performance da cache de acordo com o tamanho da *cache*. Para verificar a importância do tamanho da *cache* nós vamos comparar os arquivos de *trace* 4 e 5. O primeiro possui uma média de uma referência por endereço, enquanto o segundo possui uma média de 65 referências por endereço. Em ambos os casos testamos com a capacidade de *cache* variando de 32 até 16K. Com o primeiro arquivo(trace4) notamos um aumento de *cache hits* de 36% para 98% com 1K. Depois disso o aumento na taxa de acerto passa a ser desprezível, indicando que um aumento de *cache* indiscriminado não necessariamente gera um ganho em performance. Isto parece um pouco contra-intuitivo uma vez que a *cache* muitas vezes é vista como uma solução "one size fits all", o que não poderia estar mais longe da realidade. Por outro lado realizando o mesmo procedimento com o segundo arquivo(trace5), quando nós aumentamos a capacidade da *cache* de 32 para 16K e taxa de acerto foi de 23% para 96%, e nada indica que a taxa de acerto deixaria de aumentar com o crescimento da *cache*.

4.2. Tamanho da *cache line*

Outro aspecto importante da *cache* é o tamanho da linha de *cache* (*cache line*) que se aproveita do princípio da localidade e traz para a cache um número de endereços maior do que apenas aquele que foi solicitado. Em casos em que os endereços referenciados estão em posições próximas, tê-los já no processador eleva bastante a taxa de acertos. O tamanho da linha de cache pode influenciar bastante no tempo de processamento, dependendo da tarefa que está em curso, pois se a distância média para o próximo endereço referenciado for maior do que a quantidade de endereços que é trazida para a cache por vez, para esta tarefa, sempre será necessário um novo acesso à memória, pois o próximo endereço muito provavelmente não estará na cache. Se para a mesma situação for utilizada uma linha de cache maior, e que traga mais de um endereço referenciado por vez, a taxa de acertos pode aumentar consideravelmente. Um exemplo claro disso é o arquivo de trace3 que ao aumentarmos a cache line de 16 para 32 endereços a taxa de acertos passou de 50% para 75%.

5. Conclusão e Trabalhos futuros

As conclusões que nós pudemos tirar deste trabalho é, em primeiro lugar, que a o funcionamento da *cache*, apesar de ser transparente para os usuários, é extremamente complicado e que realizar otimizações a nível de hardware são operações complexas e multifacetadas que exigem muito conhecimento específico. Em segundo lugar percebemos que

a *cache*, apesar de ser uma das melhores ideias da computação, está longe de ser uma solução aplicável, da mesma forma, a todos os problemas.

Por fim, algumas coisas ficaram fora do escopo deste trabalho, por uma série de motivos, entretanto acreditamos que em trabalhos futuros estas questões possam ser resolvidas. Um exemplo seria realizar uma pequena generalização para que o simulador tivesse suporte a mais de um núcleo e mais de um nível de cache. Outra expansão interessante seria a adição de outros algoritmos de substituição. Outra ideia, que levaria mais tempo para implementar, seria a adição de um *scheduler* para simular várias tarefas sendo executadas ao mesmo tempo.

References

Di Carlo, S., Prinetto, P., and Savino, A. (2011). Software-based self-test of set-associative cache memories. *IEEE Transactions on Computers*, 60(7):1030–1044.

Hill, M. D. (1987). Aspects of cache memory and instruction buffer performance. Technical report, DTIC Document.

Mano, M. M. (1982). *Computer System Architecture*.

Peir, J.-K., Hsu, W. W., and Smith, A. J. (1998). *Implementation issues in modern cache memory*. University of California, Berkeley, Computer Science Division.

Smith, A. J. (1994). Trace driven simulation in research on computer architecture and operating systems. In *Proceedings of the Conference on New Directions in Simulation for Manufacturing and Communications*, pages 43–49.