PMR3309 - 2022
Cache Lab: Compreendendo memórias cache
Data final de entrega: 14/12/2024 às 23:59

Adaptado do Material de Bryant e O'Hallaron para o curso 15-213 2016 da CMU por Thiago Martins

1 Introdução

Esta atividade irá ajudá-lo a compreender o impacto de memórias cache no desempenho de seus programas C.

Ela consiste em duas partes. Na primeira parte você escreverá um pequeno programa (de 200 a 300 linhas) que simula o comportamento de uma memória cache. Na segunda parte, você irá otimizar uma pequena função de transposição de matrizes com o objetivo de minimizar o número de falhas de cache.

2 Preparando o seu trabalho

Recupere o arquivo `cachelab-handout.tar` pelo moodle:

`https://edisciplinas.usp.br/mod/resource/view.php?id=4937726`

Copie o arquivo `cachelab-handout.tar` em um diretório no qual você pretende fazer o seu trabalho. Em seguida use o comando

```
user@pmr3309:~$ tar xvf cachelab-handout.tar
```

Isso irá criar um diretório chamado `cachelab-handout` que contém diversos arquivos. Você modificará dois arquivos:`csim.c` e `trans.c`. Para compilar estes arquivos digite dentro do diretório:

```
user@pmr3309:~/cachelab-handout$ make clean
user@pmr3309:~/cachelab-handout$ make
```

3 Descrição

A atividade tem duas partes. A parte A consiste em implementar um simulador de cache. A parte B consiste em usar este simulador para otimizar uma função de transposição de matrizes em relação ao desempenho de cache.

3.1 Arquivos de Rastreamento de referência

O subdiretório `traces` da atividade contém uma coleção de arquivos de rastreamento (*trace files*) que serão usados para avaliar a correção do simulador de cache escrito na parte A. Os arquivos de rastreamento são registros da operação de um programa, neste caso específico, de atividades de acesso à memória. Estes arquivos de rastreamento foram gerados por um programa chamado `valgrind`. Por exemplo, ao digitar-se

```
user@pmr3309:~$ valgrind --log-fd=1 --tool=lackey -v --trace-mem=yes ls -l
```

na linha de comando executa-se o programa "`ls -l`", capturando um rastreamento de cada um dos seus acessos à memória na ordem que ocorrem. Este rastreamento é exibido em `stdout`.

Os rastreamentos de memória do `valgrind` têm o seguinte formato:

```
I 0400d7d4,8
 M 0421c7f0,4
 L 04f6b868,8
 S 7ff0005c8,8
```

Cada linha indica um ou dois acessos à memória. O formato de cada linha é:

```
[espaço]operação endereço,tamanho
```

O campo *operação* indica o tipo de acesso à memória: "I" indica uma leitura de instrução, "L" uma leitura de dados, "S" uma gravação de dados e "M" uma modificação de dados (i.e., uma leitura de dados seguida imediatamente por uma gravação). Não há nunca um espaço antes de cada"I". Há sempre um espaço antes de cada "M", "L", e "S". O campo *endereço* especifica um endereço de 32 ou 64 bits hexadecimal (ou seja, 8 ou 16 dígitos hexadecimais). O campo *tamanho* especifica o número de bytes acessado pela operação.

3.2 Parte A: Escrevendo um simulador de cache

Na Parte A você escreverá um simulador de cache no arquivo `csim.c` que lê um rastreamento de memória gerado pelo `valgrind`, simula o comportamento de acertos/falhas de uma memória cache e exibe o número total de acertos, falhas e substituições.

Foi fornecido um executável binário de um *simulador de cache de referência*, chamado `csim-ref`, que simula o comportamento de um cache com tamanho e associatividade arbitrários a partir de um arquivo de rastreamento `valgrind`[1]

A política de substituição do cache empregada pelo simulador é a LRU (*Least-Recently Used – O menos recentemente Usado)*. Esta política elege a linha menos recentemente usada para ser substituída por uma nova. *Esta deve ser a política usada pelo seu simulador.*

O cache usa uma estratégia de *write-back + write-allocate* nas gravações, ou seja, no caso de uma falha de cache na gravação, a região da memória alvo é carregada ao cache.

O simulador usa os seguintes argumentos de linha de comando:

[1] Esta ferramenta pode ser instalada em um sistema Debian/Ubuntu com o comando `sudo apt-get install valgrind`.

```
Usage: ./csim-ref [-hv] -s <s> -E <E> -b <b> -t <tracefile>
```

- `-h`: Opção que exibe informações de uso (este texto em inglês).
- `-v`: Opção que exibe informação de depuração.
- `-s <s>`: Número de bits no índice de conjuntos ($S = 2^s$ é o número de conjuntos)
- `-E <E>`: Associatividade (número de linhas de cache por conjunto)
- `-b <b>`: Número de bits do endereçamento de bloco ($B = 2^b$ é o tamanho de cada bloco em bytes)
- `-t <tracefile>`: Nome do rastreamento `valgrind` para a simulação

Os argumentos da linha de comando são baseados na notação (s, E, e b) da página 597 do livro CS:APP 2ed.

Por exemplo:

```
user@pmr3309~$ ./csim-ref -s 4 -E 1 -b 4 -t traces/yi.trace
hits:4 misses:5 evictions:3
```

O mesmo exemplo em modo *verbose*:

```
user@pmr3309~$ ./csim-ref -v -s 4 -E 1 -b 4 -t traces/yi.trace
L 10,1 miss
M 20,1 miss hit
L 22,1 hit
S 18,1 hit
L 110,1 miss eviction
L 210,1 miss eviction
M 12,1 miss eviction hit
hits:4 misses:5 evictions:3
```

Sua tarefa para a parte A é preencher o arquivo `csim.c` de modo a construir um programa que toma os mesmos argumentos de linha de comando e produz uma saída idêntica à do simulador de referência. Note que o arquivo está praticamente vazio (ele basicamente processa a linha de comando).

Regras de Programação para a Parte A

- Inclua seu nome e NUSP nos comentários do cabeçalho do arquivo `csim.c`.
- Seu arquivo `csim.c` deve compilar *sem warnings*.
- Seu simulador deve trabalhar corretamente para valores arbitrários de s, E, e b. Isso significa que você precisará alocar armazenamento para as estruturas de dados do simulador usando a função `malloc` function. Digite "man malloc" para informações sobre esta função.

- Para esta atividade estamos interessados apenas no desempenho do cache de dados, de modo que o seu simulador deve ignorar todos os acessos a instruções (linhas iniciadas por h "I"). Lembre-se que `valgrind` sempre põe "I" na primeira coluna (sem espaço que o anteceda) e "M", "L" e "S" na segunda coluna (precedido de um espaço). Isso pode ajudá-lo a interpretar o arquivo de rastreamento.
- Ao final da simulação, para exibir o resultado na formatação correta, você deve chamar a função `printSummary` com o número total de acertos, falhas e substituições:

```
printSummary(hit_count, miss_count, eviction_count);
```

- Para este laboratório você deve presumir que os acessos de memória estão em um alinhamento favorável, de modo que um acesso de memória *nunca* cruza fronteiras entre blocos. Note que com este pressuposto, pode-se ignorar os tamanhos de requisições nos rastreamentos produzidos por `valgrind`.

3.3 Parte B: Otimizando transposição de Matrizes

Na parte B você escreverá uma função de transposição em `trans.c` que provoca o menor número de falhas de cache.

Seja A uma matriz e A_{ij} o componente na i-gésima linha e j-ésima coluna. A *tranposta* de A, dita A^T, é uma matriz tal que $A_{ij} = A^T_{ji}$.

Para ajudá-lo, foi fornecida uma função de transposição de exemplo em `trans.c` que computa a transposta de uma matriz A $N \times M$ matrix A e armazena o resultado em uma matriz B $M \times N$:

```
char trans_desc[] = "Simple row-wise scan transpose";
void trans(int M, int N, int A[N][M], int B[M][N])
```

A função de exemplo está correta, mas é ineficiente pois o padrão de acesso à memória produz relativamente muitas falhas de cache.

Seu trabalho na parte B é escrever uma função similar, chamada `transpose_submit`, que minimiza o número de falhas de cache com diferentes tamanhos de matrizes:

```
char transpose_submit_desc[] = "Transpose submission";
void transpose_submit(int M, int N, int A[N][M], int B[M][N]);
```

Não altere a string de descrição ("`Transpose submission`") para a sua função `transpose_submit` O scritp de auto-correção procura por essa string para determinar qual função de transposição deve ser avaliada para a sua nota.

Regras de programação para a parte B

- Inclua o seu nome e NUSP nos comentários do cabeçalho do arquivo `trans.c`.

- Seu código em `trans.c` deve compilar sem warnings.
- Você pode definir até 12 variáveis locais do tipo `int` por função de transposição.[2]
- Você não pode contornar a restrição anterior usando variáveis do tipo `long` ou usando qualquer outro truque para armazenar mais do que um valor em uma única variável.
- Sua função de recursão não pode suar recursão.
- Se você escolher usar funções auxiliares, você não deve ter mais do que 12 variáveis locais na pilha em qualquer momento, incluindo o estado das funções auxiliares e suas funções auxiliares. Por exemplo, se sua função principal declara 8 variáveis e chama uma função com mais 4 variáveis, que por sua vez chama outra função com mais duas, você terá 14 variáveis na pilha e estará em violação desta regra.
- Sua função de transposição não deve modificar a matriz A. Você pode, no entanto, alterar como quiser o conteúdo da matriz B.
- Você *não* pode definir nenhuma matriz no seu código ou usar qualquer variação de `malloc`.

4 Avaliação

Esta seção descreve como seu trabalho será avaliado. O trabalho vale um total de 60 pontos:

- Parte A: 27 Pontos
- Parte B: 26 Pontos
- Estilo: 7 Pontos

4.1 Avaliação da parte A

Para a parte A, o seu simulador será executado usando parâmetros distintos de caches e rastreamentos. Há oito casos de teste, cada um valendo 3 pontos, exceto o último caso, que vale 6 pontos:

```
user@pmr3309:~$ ./csim -s 1 -E 1 -b 1 -t traces/yi2.trace
user@pmr3309:~$ ./csim -s 4 -E 2 -b 4 -t traces/yi.trace
user@pmr3309:~$ ./csim -s 2 -E 1 -b 4 -t traces/dave.trace
user@pmr3309:~$ ./csim -s 2 -E 1 -b 3 -t traces/trans.trace
user@pmr3309:~$ ./csim -s 2 -E 2 -b 3 -t traces/trans.trace
user@pmr3309:~$ ./csim -s 2 -E 4 -b 3 -t traces/trans.trace
user@pmr3309:~$ ./csim -s 5 -E 1 -b 5 -t traces/trans.trace
user@pmr3309:~$ ./csim -s 5 -E 1 -b 5 -t traces/long.trace
```

[2] Esta restrição se deve ao fato de que o código de correção não consegue contar referências à pilha. O objetivo é limitar as suas referências e focalizar o seu trabalho em padrões de acesso às matrizes de origem e destino.

Você pode usar o simulador de referência `csim-ref` para obter a resposta correta para cada um dos testes. Você pode usar também a opção `-v` para detalhar a simulação do cache.

Para cada caso de teste, exibir o número correto de acertos, falhas e substituições de cache dará a nota inteira para o caso. Cada um dos números, a saber, acertos, falhas e substituições, vale $1/3$ do caso. Por exemplo, se um caso particular vale 3 pontos e seu simulador acerta o número de acertos e falhas, mas erra o número de substituições, você ganhará 2 pontos por este caso.

4.2 Avaliação da parte B

Para a parte B, será avaliada a correção e desempenho de sua função `transpose_submit` em três matrizes com tamanhos distintos:

- 32×32 ($M = 32$, $N = 32$)
- 64×64 ($M = 64$, $N = 64$)
- 61×67 ($M = 61$, $N = 67$)

4.2.1 Desempenho (26 pontos)

Para cada matriz, o desempenho de sua função `transpose_submit` será avaliado usando `valgrind` para extrair o rastreamento de sua função e então o simulador de referência será usado para simular o seu desempenho em um cache com parâmetros ($s = 5$, $E = 1$, $b = 5$).

Sua pontuação para cada tamanho de matriz escala linearmente com o número de falhas de cache m, até um dado limite:

- 32×32: 8 pontos se $m < 300$, 0 pontos se $m > 600$
- 64×64: 8 pontos if $m < 1,300$, 0 pontos se $m > 2,000$
- 61×67: 10 pontos se $m < 2,000$, 0 pontos se $m > 3,000$

Seu código deve estar correto (ou seja, calcular efetivamente a transposição) para receber qualquer ponto para um caso particular. Seu código só precisa estar correto para estes 3 casos e você pode otimizar especificamente para eles. Em particular, você pode checar na sua função o tamanho da entrada e implementar códigos distintos para cada caso.

4.3 Avaliando o Estilo

Há 7 pontos dados por estilo de codificação. Eles serão atribuídos manualmente pela equipe do curso. Documente corretamente seu código.

5 Trabalhando na Atividade

5.1 Trabalhando na Parte A

É fornecido um programa de autocorreção chamado `test-csim` que avalia o seu simulador de cache nos rastreamentos de referência. Lembre-se de compilar o seu simulador antes de rodar o teste:

```
linux> make
linux> ./test-csim
                        Your simulator     Reference simulator
Points (s,E,b)    Hits  Misses  Evicts    Hits  Misses  Evicts
     3 (1,1,1)       9       8       6       9       8       6  traces/yi2.trace
     3 (4,2,4)       4       5       2       4       5       2  traces/yi.trace
     3 (2,1,4)       2       3       1       2       3       1  traces/dave.trace
     3 (2,1,3)     167      71      67     167      71      67  traces/trans.trace
     3 (2,2,3)     201      37      29     201      37      29  traces/trans.trace
     3 (2,4,3)     212      26      10     212      26      10  traces/trans.trace
     3 (5,1,5)     231       7       0     231       7       0  traces/trans.trace
     6 (5,1,5)  265189   21775   21743  265189   21775   21743  traces/long.trace
    27
```

Para cada teste, ele mostrará o número de pontos que você obteve, os parâmetros do cache, o arquivo de entrada e uma comparação dos resultados do seu simulador com o do simulador de referência.

Aqui seguem algumas sugestões para trabalhar na parte A:

- Faça testes e debugs em rastreamentos pequenos, tais como `traces/dave.trace`.
- O simulador de referência leva um argumento opcional `-v` que habilita a saída de depuração. Esta saída mostra os acertos, erros e substituições a cada acesso de memória. O seu código não precisa implementar esta função, mas é útil se você o fizer. Isso ajudará-o a depurar seu código comparando o seu comportamento diretamente com o do simulador.
- Lembre-se que você deve adotar uma estratégia de *write-allocate* em caso de falhas de cache na escrita, ou seja, considere que a região da memória alvo da operação de escrita é carregada para o cache.
- Cada leitura e gravação de dados pode provocar no máximo uma falha de cache. A operação de modificação deve ser tratada como uma leitura seguida de uma gravação no mesmo endereço. Deste modo, uma operação de modificação pode resultar em dois acertos de cache ou uma falha e um acerto e uma possível substituição.

5.2 Trabalhando na Parte B

Foi fornecido um programa de autocorreção chamado `test-trans.c` que testa a correção e desempenho de cada uma das funções de transposição que você registrou.

Você pode registrar até 100 versões da função de transposição no seu arquivo `trans.c`. Cada função de transposição tem a seguinte forma:

```
/* Header comment */
char trans_simple_desc[] = "A simple transpose";
void trans_simple(int M, int N, int A[N][M], int B[M][N])
{
    /* your transpose code here */
}
```

Registre uma função de transposição com o programa de avaliação com a seguinte chamada de função:

```
registerTransFunction(trans_simple, trans_simple_desc);
```

dentro da função `registerFunctions` em `trans.c`. Durante a sua execução, o programa de avaliação irá avaliar cada função registrada e mostrar os resultados. Naturalmente, uma das funções registradas *deve* ser a função `transpose_submit`

```
registerTransFunction(transpose_submit, transpose_submit_desc);
```

Veja o arquivo `trans.c` original para uma ideia de como isso funciona.

O programa de avaliação toma o tamanho da matriz como entrada. Ele em seguida usa `valgrind` para gerar um rastreamento de cada função registrada. Em seguida ele avalia cada rastreamento rodando o simulador de referência com um cache com parâmetros ($s = 5$, $E = 1$, $b = 5$).

Por exemplo, para testar suas funções de transposição em uma matriz 32×32 recompile `test-trans` e execute-o com valores apropriados para M e N:

```
user@pmr3309:~$ make
user@pmr3309:~$ ./test-trans -M 32 -N 32
Step 1: Evaluating registered transpose funcs for correctness:
func 0 (Transpose submission): correctness: 1
func 1 (Simple row-wise scan transpose): correctness: 1
func 2 (column-wise scan transpose): correctness: 1
func 3 (using a zig-zag access pattern): correctness: 1

Step 2: Generating memory traces for registered transpose funcs.

Step 3: Evaluating performance of registered transpose funcs (s=5, E=1, b=5)
func 0 (Transpose submission): hits:1766, misses:287, evictions:255
func 1 (Simple row-wise scan transpose): hits:870, misses:1183, evictions:1151
func 2 (column-wise scan transpose): hits:870, misses:1183, evictions:1151
func 3 (using a zig-zag access pattern): hits:1076, misses:977, evictions:945

Summary for official submission (func 0): correctness=1 misses=287
```

Neste exemplo, foram registradas quatro funções diferentes de transposição em `trans.c`. O programa `test-trans` testa cada uma destas funções, mostra os resultados para cada uma e extrai os resultados da submissão oficial.

Aqui seguem algumas sugestões para trabalhar na parte B.

- O programa `test-trans` salva o rastreamento para a i-gésima função no arquivo `trace.f`i.[3] Estes arquivos são importantes ferramentas para ajudá-lo a compreender de onde vêm cada acerto e falha de cache para cada função. Para depurar uma função particular, rode-a no simulador de referência com a opção `-v`:

```
user@pmr3309:~$ ./csim-ref -v -s 5 -E 1 -b 5 -t trace.f0
S 68312c,1 miss
L 683140,8 miss
L 683124,4 hit
L 683120,4 hit
L 603124,4 miss eviction
S 6431a0,4 miss
...
```

- Lembre-se que a opção `E 1` significa que o cache tem apenas *uma* linha por conjunto, ou seja, ele é um cache *mapeado diretamente*. Em caches mapeados diretamente, falhas causadas por conflitos (ou seja, endereços distintos que são mantidos na mesma linha de cache) são um grave problema.

- Em relação ao exposto acima, uma fonte importante de conflitos são acessos alternados a dados na matriz A e B. Você pode considerar que a diferença na posição de inicio da matriz A e a matriz B é *sempre* 262144 bytes (ou 0x00040000 em hexadecimal), independentemente dos valores de M e N. Pense na transposição dos elementos diagonais das matrizes e de como ela afeta o cache.

- O acesso às matrizes em blocos, como mencionado em sala, é uma técnica importante para reduzir falhas de cache. Vide

  `http://csapp.cs.cmu.edu/public/waside/waside-blocking.pdf`

  para mais informação.

5.3 Verificando o seu trabalho

É fornecido um programa chamado `./driver.py` que avalia o seu trabalho, tanto no simulador quanto no código de transposição. Este é o mesmo programa empregado pelos instrutores para avaliar as suas submissões. O programa chama `test-csim` para avaliar seu simulador e `test-trans` para avaliar a função submetida para transposição dos três tamanhos de matriz descritos anteriormente. Em seguida, ele mostra um sumário dos seus resultados com a correspondente pontuação.

Para usá-lo, digite:

```
user@pmr3309:~$ ./driver.py
```

[3] Como `valgrind` introduz vários acessos à pilha que não têm nada a ver com o seu código, estes estão removidos do rastreamento. Esta é a razão da proibição de matrizes locais e limites em variáveis locais.

6 Entregando o seu trabalho

A cada vez que você executa o comando `make` no diretório `cachelab-handout` um arquivo chamado `userid-handin.tar` (onde `userid` é o nome do usuário atual). Este arquivo contém os seus arquivos `csim.c` e `trans.c` atuais.

Submeta o arquivo pelo moodle no link:

`https://edisciplinas.usp.br/mod/assign/view.php?id=2135168`