SIMULAÇÃO DE UMA MEMÓRIA CACHE
split, 2 way set associative com *LRU*

NUNO NUNES
PAULO BLUEMEL

3 de Junho de 2001

Conteúdo

Lista de Figuras

Lista de Tabelas

1 Objectivos

Com este trabalho pretende-se fazer uma simulação de uma memória cache.
O processo de escrita em memória principal não deve ser simulado.
A arquitectura da cache a simular será *2-way set-associative*, separada (*split*), política de substituição LRU[1]. Os parâmetros do programa de simulação são o número de linhas e o comprimento de cada linha (em bytes); cada cache deve ser especificada separadamente.

[1] Least Recently Used

2 Explicação sumária das opções efectuadas no nível de programação

Para a implementação do programa de simulação escolhemos a linguagem "*c*", devido à sua portabilidade e ao formato em que os ficheiros *test.trace* e *add.trace* se encontravam, o que simplificou o trabalho de programação.
A implementação foi relativamente fácil e rápida, mas houve pequenos pormenores que nos escaparam (vulgo *bugs*), que foram detectados e corrigidos (*todos, esperamos*).
Utilizamos vectores de estruturas para definir a estrutura da cache, procedendo, em seguida, à leitura do(s) ficheiro(s) de simulação. Mediante os parâmetros lidos, é então calculada a linha da cache onde possivelmente estará o valor, analiza-se o campo *valid*, procede-se ao calculo da *tag*, e compara-se com os valores já existentes na cache.
Caso exista uma situação de *cache hit*[2], incrementa-se o contador de *hits*, e actualiza-se a entrada correspondente ao campo *LRU*.
Caso não exista na cache (*cache miss*), incrementa-se o contador de *misses*, actualizando-se em seguida os campos de *tag*, *valid* e *LRU*.
No fim da leitura do ficheiro, caso a chamada do programa tenha sido efectuada com a opção *-d*, executa-se novamente, mas sem inicializar a cache.
Para o indispensável *debugging*, optámos por incluir um *#define debug N*, que, quando $N \neq 0$ é mostrado no *stdout* o conteúdo da cache a cada acesso assim, como os parâmetros que estão a ser alterados.
Criámos também mais dois ficheiros de simulação *seq.trace* e *rand.trace*, que contêm, respectivamente acessos sequênciais e pseudo-aleatórios à memória, de forma a que nos fosse possível a análise das caracteristicas espaciais num programa e as suas implicações nos parâmetros da cache.
Depois do programa efectuado, para a geração dos gráficos, numa primeira fase, pensámos em utilizar o programa *gnuplot*[3], utilizando como saída os valores de um script em *bashell*, de forma a termos gráficos representativos da *performance* da cache, calculados instantâneamente e facilmente variáveis para outros parâmetros de simulação, mas não conseguimos, em tempo útil, uma maneira rápida e fiável de acertar com os parâmetros do *gnuplot*, pelo que voltámo-nos para a utilização de uma folha de cálculo normal (*starcalc* do *staroffice*) com a possibilidade de gerar gráficos.
Assim, fomos anotando os valores de saída das várias simulações, cujo comando se encontra nas próprias figuras, e preenchendo a folha de cálculo.
Os valores escolhidos para a simulação, e posterior geração dos gráficos, tiveram em conta com uma apresentação mais cuidada do próprio gráfico, de forma a que fosse mais facilmente possível a performance da cache.
Para a realização deste relatório, optámos pela utilização da *macro-package* LaTeX 2_ε assim como da sua documentação[3].

3 Apresentação e Análise de Resultados

Nas figuras seguintes é apresentado o comportamento da cache, para os vários ficheiros de simulação e parâmetros de entrada.
O ficheiro de simulação *seq.trace* (para instruções), é composto por uma simulação de acessos sequênciais desde o endereço *0x00* até *0xFF*, i.e., contém as seguintes entradas :

f 0x00 0x00
f 0x00 0x01
...
f 0x00 0xfe

[2] O campo *valid*, na linha calculada, é *1*, e o valor da *tag* corresponde ao calculado, pelo menos numa das *sub-caches*
[3] http://www.gnuplot.org

f 0x00 0xff

Este ficheiro de simulação, pretende assim simular um programa em que não existem qualquer tipo de saltos.
Para dados, o ficheiro *seq.trace* contém :

r 0xFFFFFFFF 0x00
r 0xFFFFFFFF 0x01
...
r 0xFFFFFFFF 0xfe
r 0xFFFFFFFF 0xff

pretendendo simular um acesso à memória sequêncial, o que é mais realista que o caso anterior, dado que na prática corresponde, a por exemplo, um acesso a um vector.
O ficheiro *rand.trace*, foi criado tendo em vista acessos aleatórios, ignorando assim, as características espaciais da maioria dos programas. O espaço de endereçamento encontra-se desde *0x00* até 0x3FF, portanto 1024 bytes.
Como era esperado, na maioria dos ficheiros de simulação, à medida que era aumentado o tamanho da cache, também era aumentada a sua *taxa de acertos*[4].
Observa-se claramente, que à excepção do programa de teste *rand.trace*, que contém acessos pseudo-aleatórios, e para casos particulares do *seq.trace*, que contém acessos totalmente sequênciais, a utilização de caches é um factor de aumento de performance.
Notou-se no entanto, ao contrário do que era inicialmente esperado, a segunda vez que a cache era simulada, i.e., quando o programa era invocado com a opção *-d*, a taxa de acertos não era aumentada significativamente. Pensando que era um erro no nosso programa, procedemos ao *debugging* e verificámos que a extensão dos programas de teste assim como o seu conteúdo eram suficientes para minorar o efeito de *arranque a frio* da cache.

3.1 Análise das variações na cache de instruções

Observe-se então a figura 1, onde é variado o número de linhas da cache de instruções, tendo cada linha 2 *bytes*.

Observa-se que o desempenho da cache, nomeadamente nos ficheiros de simulação *test.trace* e *add.trace*, mesmo para tamanhos relativamente pequenos da cache de instruções, (*1 linha · 2 bytes por linha · 2 way set associative = 4 bytes*), apresenta um rendimento considerável ($\approx$ *47%*).
À medida que o número de linhas da cache vai aumentando, assim é incrementeada a taxa de acertos, como era esperado.
De notar, no entanto, o comportamento da cache, com o ficheiro de teste *seq.trace*, que apresenta uma taxa de acertos inicial de *50%*, passando, quando o número de linhas é de 32, para uma taxa de acertos de *100%*.
A taxa de acertos inicial de 50% é explicada, devido ao facto de que o número de bytes por linha é de 2. Assim, quando, a cache se encontra vazia e é feito um acesso ao endereço *0x00*, ocorre um *cache miss*, em que a cache se encarrega de ir à memória buscar o conteúdo dos endereços *0x00* e *0x01*. No instante seguinte, quando é solicitado o conteúdo do endereço *0x01*, a cache já o tem, pelo que ocorre um *cache hit*, e assim sucessivamente, numa sequência de *cache misses* e *cache hits*, provocando uma taxa de acertos de 50%.
Para um número de linhas de 64 (e superiores), dado que o ficheiro *seq.trace* simula acessos sequênciais desde o endereço *0x00* até *0xFF* (um espaço de endereçamento de 256 bytes), e a memória cache tem (*64 linha · 2 bytes por linha · 2 way set associative*) 256 bytes, então o espaço de endereçamento cabe completamente na memória cache, conseguindo-se assim, uma taxa de acertos de 100%.

[4] Hit Rate

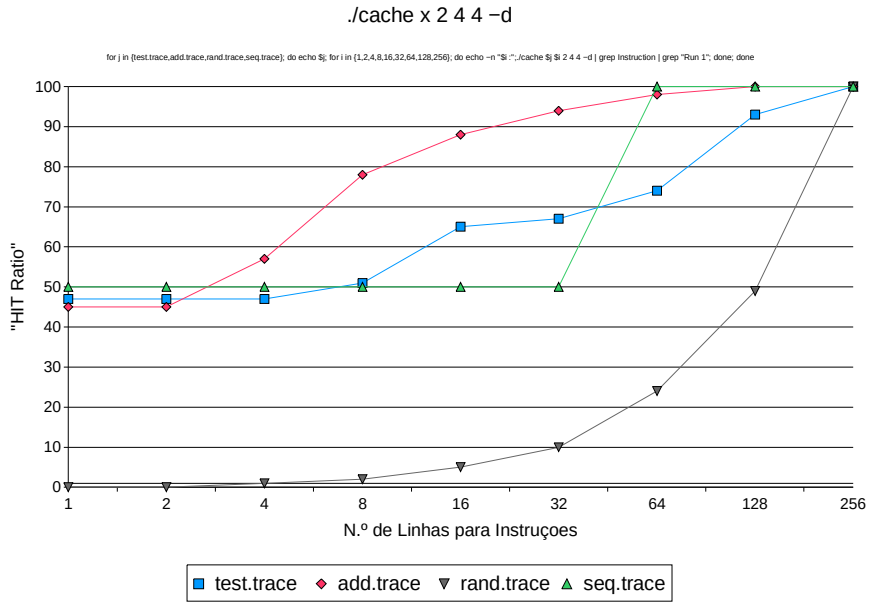

Figura 1: Variação do N.º de linhas da cache de instruções

Para o ficheiro de simulação *rand.trace*, obtém-se uma taxa de acertos muito baixa ($< 10\%$), para um baixo valor de linhas ($1 \ldots 32$), porque os acessos são aleatórios, pelo que a cache não aumenta a performance dos acessos à memória. No extremo, a cache só aumenta a performance, quando é suficientemente grande para albergar a totalidade do espaço de endereçamento da memória. Por exemplo, para um número de linhas de 256, a cache fica com um tamanho de (256 *linha* $\cdot$ 2 *bytes por linha* $\cdot$ 2 *way set associative*) 1024 bytes. O espaço de endereçamento utilizado na criação deste ficheiro foi também de 1024 bytes, pelo que se obtém uma taxa de acertos de 100%. Para a figura 2, foi variado o número de bytes para instruções, por linha.

Obtivemos, tal como na figura anterior, um crescimento da taxa de acertos com o aumento do número de bytes, mais notória nos ficheiros *test.trace*, *rand.rtace* e *seq.trace*, seguindo uma monotonia dada pela equação 1.
Mais uma vez, a simulação do *rand.trace* mostrou a ineficácia da cache em acessos aleatórios.

3.2 Análise das variações na cache de dados

Na cache de dados, para uma variação do número de linhas, a monotonia da taxa de acertos foi muito idêntica à sua variação na cache de instruções, apenas com uma pequena melhoria, que pode ser explicada pelo facto de que é mais comum a existência de *saltos* num programa, do que as variáveis que ele acessa não estarem em posições contíguas.
No entanto, isto depende fortemente do programa, pelo que, apenas pode ser numa situação comum.
Também na variação do número de bytes por linha, obtiveram-se resultados idênticos à simulação da cache de instruções, que mais uma vez, tende a seguir a fórmula 1.

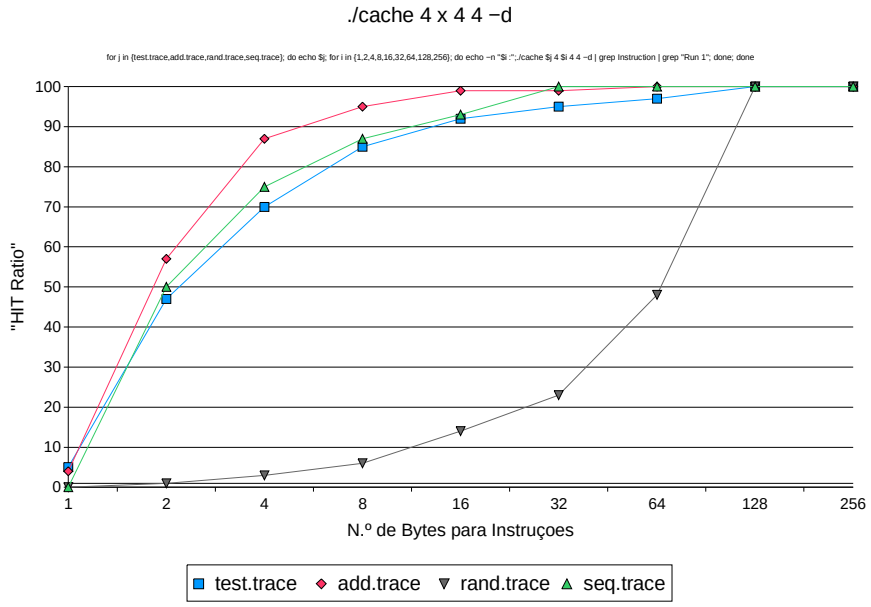

Figura 2: Variação do N.º de bytes por linha da cache de instruções

Para a cache de dados, obtivemos as figuras 3 e 4, que têm uma forma bastante semelhante à das figuras anteriores, como seria de esperar. Aqui, será útil dar especial atênção à variação do ficheiro *seq.trace*.
Como se observa, para um valor razoável de número de linhas, torna-se vantajoso, incrementar o número de bytes, dado a variação da fórmula 1.
Assim, caso estivéssemos na presença de um sistema que utilizasse maioritariamente acessos sequênciais a vectores, seria razoável um aumento no número de bytes por linha, dado que este tem um incremento directo e mais rápido do que um aumento no número de linhas.

4 Análise do trabalho

Como conclusão básica do trabalho, apercebemo-nos que enquanto a cache é *relativamente* pequena um aumento no seu tamanho impoê um aumento significativo na taxa de acertos. Quando o tamanho da cache já é *suficientemente* grande, um aumento no seu tamanho não se traduz num aumento significativo na taxa de acertos.
No limite, quando a cache consegue albergar toda a faixa de endereçamento de um dado programa, a razão de acertos é sempre de 100%.

Para uma variação do número de bytes por linha, analisando a taxa de acertos do ficheiro de simulação *seq.trace*, chegámos à seguinte equação:

$$HitRate = \frac{2^n - 1}{2^n}; n \geqslant 1 \tag{1}$$

que indica a taxa de acertos para a variação do número de bytes por linha.
Os ficheiros de simulação *test.trace* e *add.trace*, também parecem reger-se por uma equação aproxi-

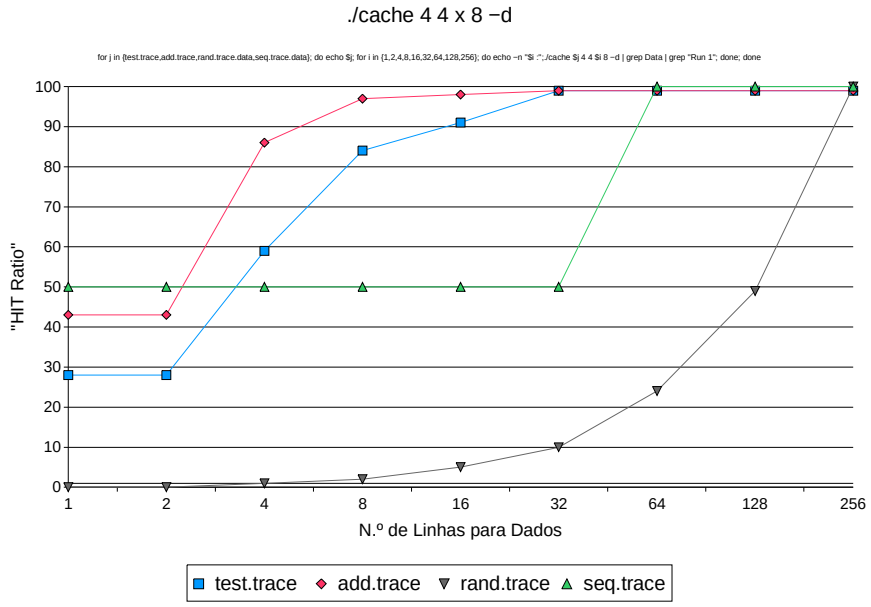

Figura 3: Variação do N.º de linhas da cache de dados

mada, como se pode confirmar confirmar pelas figuras apresentadas (2 e 4). Isto deve-se ao facto de que a maioria dos programas têm uma característica de localização espacial, seguindo assim a monotonia do ficheiro de simulação *seq.trace*. No entanto, além de acessos sequenciais quer a nível de instruções que a nivel de dados, também têm componentes aleatórias, embora em menor parte, logo, podemos concluir que qualquer programa pode ser aproximado por uma combinação com diferentes *pesos* por um programa *seq.trace*, *rand.trace* e um outro que utilize sempre os mesmos dados/instruções.

Observe-se de seguida as tabelas 1 e 2 onde são mostradas a variação necessária no tamanho da cache de instruções e de dados respectivamente, para um incremento na taxa de acertos, para os dois programas de simulação.
Observa-se assim que para altos valores relativos da taxa de acertos, uma tentativa de incremento nessa taxa leva a um muito maior incremento no tamanho da cache.
Cabe a quem dimensiona os parâmetros da cache, tendo em conta o nível de performance exigido e factores como o preço no incremento do tamanho, escolher então a melhor opção.

É ainda de notar, e conforme está expresso nas figuras 1, 2, 3 e 4, que programas que utilizem maioritariamente acessos aleatórios, embora sejam raros na prática, não tiram partido da inclusão da cache, excepto quando o tamanho da cache é uma boa parte do tamanho da memória principal, o que não é viável na prática.

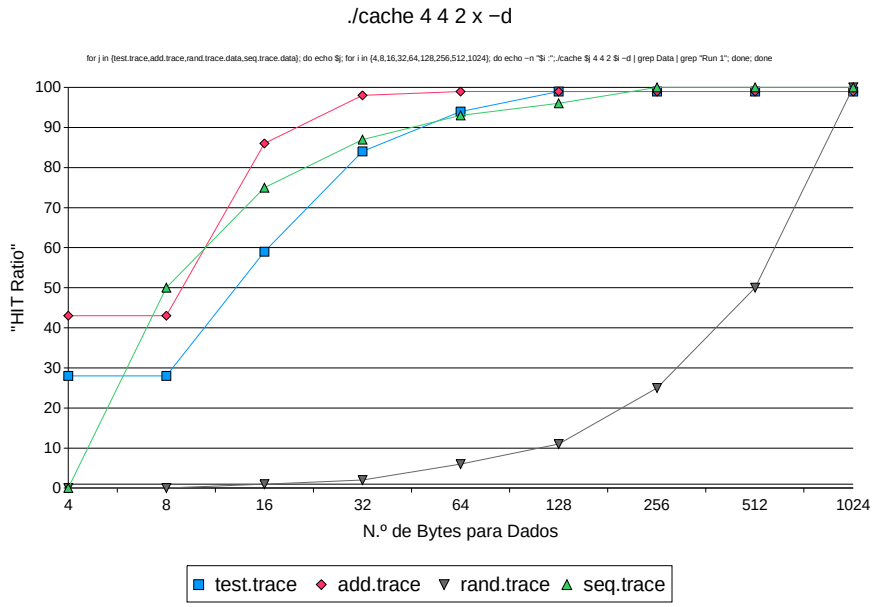

Figura 4: Variação do N.º de bytes por linha da cache de dados

Ficheiro	N.º Linhas	N.º de bytes p. l.	Tamanho *(bytes)*	*Hit Ratio*
test.trace	2	32	128	93%
	2	128	512	98%
	128	4	1024	99%
add.trace	8	4	64	93%
	4	16	128	99%
	64	4	512	100%

Tabela 1: Variação do tamanho da cache de instruções para variações no *Hit Rate*

Ficheiro	N.º Linhas	N.º de bytes p. l.	Tamanho *(bytes)*	*Hit Ratio*
test.trace	4	32	256	94%
	64	4	512	99%
	1024	512	1048576	100 %
add.trace	2	32	128	98%
	4	32	256	99%
	1024	512	1048576	100%

Tabela 2: Variação do tamanho da cache de dados para variações no *Hit Rate*

5 Extensão dos objectivos propostos

De forma a confirmarmos realmente a variação no tipo e tamanho de dados que um programa acessa com a sua taxa de acertos, recorremos ao pacote *CacheProf* [2]
Este programa foi projectado para possibilitar uma percepção e quantificação do comportamento da cache em programas e algoritmos.
Cacheprof corre o programa escolhido, simulando uma cache à escolha, e reporta cada linha do código fonte com o número de referências à memória e o número de *cache misses* causado por linha. Reporta também sumários por procedimento, e para o programa como um todo. Finalmente, *CacheProf* conta também o número de instruções executadas.
Fizemos algumas simulações com o nosso próprio programa de simulação *cache.c*, e confirmámos a utilização do *CacheProf*, dado que, por exemplo, à medida que aumentávamos os parâmetros do nosso programa (*criávamos vectores maiores*), também aumentava o número de misses, como seria de esperar.
Infelizmente, limitações temporais impediram-nos de explorar melhor todas as potencialidades deste programa.

A Programa Implementado

O programa implementado, encontra-se *online*[5], devido ao efeito de quebra de linhas impossível de minorar neste documento, assim como os ficheiros de teste adicionais *seq.trace* e *rand.trace*, e a *folha de cálculo* utilizada para a geração das figuras aqui mostradas.
No programa seguinte, foram eliminadas as instruções de debugging, de forma a torná-lo mais perceptível.

```
#include <stdio.h>
#include <string.h>
# define debug 0
# define max_lines 8096
# define max_bytes 1024
# define nbit_address 32
# define nbit_inst_dado 8
# define nbit_data_dado 32
# define set_associative 2
int main(int *argc, char *argv[]){
  int n_vezes;
  FILE *j;
  if (( (int) argc < 6)){
    printf ("Sintax : %s nome_ficheiro linhas_inst bytes_inst
                              linhas_dados bytes_dados
                              [-d]\n",argv[0]);
    exit (1);
  }
  if ((j=fopen(argv[1],"r")) == NULL){
    perror("fopen");
    printf ("Check %s .",argv[1]);
    exit(1);
  }else{
    fclose (j);
  }
  if ( (atoi(argv[2])) <= 0 ){
    printf ("Erro: Linhas Cache Instrução tem de ser > 0 ...\n");
```

[5] http://www.fe.up.pt/~ee99043/cache/files/

```
    exit(1);
  }
  if ( (atoi(argv[3])) <= 0 ){
    printf ("Erro: Bytes Cache Instrução tem de ser > 0 ...\n");
    exit(1);
  }
  if ( (atoi(argv[4])) <= 0 ){
    printf ("Erro: Linhas Cache Dados tem de ser > 0 ...\n");
    exit(1);
  }
  if ( (atoi(argv[5])) <= 3 ){
    printf ("Erro: Bytes Cache Dados tem de ser >= 4 ...\n");
    exit(1);
  }
  if ( ((atoi(argv[5]))%4) != 0){
    printf ("Erro: Bytes Cache Dados tem de ser divisível por 4
                              ...\n");
    exit(1);
  }
  n_vezes=1;
  if (( (int)  argc == 7 )){
    if ( strcmp(argv[6],"-d") == 0 ){
      n_vezes=2;
    }
  }

                              simulate(argv[1],atoi(argv[2]),atoi(
                              argv[3]),atoi(argv[4]),atoi(argv[5]),
                              n_vezes);
}
int simulate(char filename[], int i_nlines, int i_nbytes, int
                              d_nlines, int d_nbytes, int nvezes){
  typedef struct inst_line_struct{
    unsigned int lru;
    unsigned int valid[set_associative];
    unsigned int tag[set_associative];
  }inst_line_cache;
  typedef struct data_line_struct{
    unsigned int lru;
    unsigned int valid[set_associative];
    unsigned int tag[set_associative];
  }data_line_cache;
  typedef struct cache_struct{
    inst_line_cache instruction[i_nlines];
    unsigned int inst_hit,inst_miss;
    data_line_cache data[d_nlines];
    unsigned int data_hit,data_miss;
  }split_cache;
  split_cache cache;
  int h,i,j,k,l;
  FILE *ficheiro;
  char tipo_acesso;
  unsigned int endereco,dado;
  unsigned int number_of_line,number_of_tag,number_of_byte;
```

```
  unsigned int achei;
  for (h=0;h<set_associative;h++){
    for (i=0;i<i_nlines;i++){  // Initialize Instruction cache ...
      cache.instruction[i].lru=1;
      cache.instruction[i].tag[h]=0;
      cache.instruction[i].valid[h]=0;
    }
    for (i=0;i<d_nlines;i++){  // Initialize Data cache ...
      cache.data[i].lru=1;
      cache.data[i].tag[h]=0;
      cache.data[i].valid[h]=0;
    }
  }
  d_nbytes=d_nbytes/4; // Os dados sao de 32 bits (4 bytes)
  for (l=0;l<nvezes;l++){
    cache.inst_hit=cache.inst_miss=cache.data_hit=cache.data_miss=0;
                              // Clean counters
    ficheiro=fopen(filename,"r");
    while ( fscanf(ficheiro,"%c %x %x\n",&tipo_acesso,&dado,&endereco)
                              == 3){ // Enquanto houver algo para ler
                              ...
      if (tipo_acesso==’f’){ // It’s a fetch
achei=0;
number_of_line=((endereco%(i_nlines*i_nbytes))/i_nbytes);
number_of_tag=endereco/i_nbytes;
for (i=0;i<set_associative;i++){
  if (achei==0){
    if (cache.instruction[number_of_line].valid[i]==1){
      if(cache.instruction[number_of_line].tag[i]==number_of_tag){
                              //HIT
achei=1;
cache.inst_hit++;
cache.instruction[number_of_line].lru=i;
      }
    }
  }
}
if (achei==0){ //MISS
  cache.inst_miss++;
  if (cache.instruction[number_of_line].lru==0){
    i=1;
  }else{
    i=0;
  }
  cache.instruction[number_of_line].lru=i;
  cache.instruction[number_of_line].valid[i]=1;
  cache.instruction[number_of_line].tag[i]=number_of_tag;
}
      }
      if ( (tipo_acesso==’r’) || (tipo_acesso==’w’) ){ // It’s a read
                              or a write
achei=0;
number_of_line=((endereco%(d_nlines*d_nbytes))/d_nbytes);
number_of_tag=endereco/d_nbytes;
```

```
for (i=0;i<set_associative;i++){
  if (achei==0){
    if (cache.data[number_of_line].valid[i]==1){
      if(cache.data[number_of_line].tag[i]==number_of_tag){ //HIT
achei=1;
cache.data_hit++;
cache.data[number_of_line].lru=i;
      }
    }
  }
}
if (achei==0){ //MISS
  cache.data_miss++;
  if (cache.data[number_of_line].lru==0){
    i=1;
  }else{
    i=0;
  }
  cache.data[number_of_line].lru=i;
  cache.data[number_of_line].valid[i]=1;
  cache.data[number_of_line].tag[i]=number_of_tag;
}
      }
    }
    fclose(ficheiro);
    if ( (cache.inst_hit+cache.inst_miss) != 0 ){
      printf ("Run %d :Instruction Hit Ratio =\t%d%% (%d em
                              %d)\n",l+1,cache.inst_hit*100/(cache.
                              inst_hit+cache.inst_miss),cache.
                              inst_hit,(cache.inst_hit+cache.
                              inst_miss));
    }else{
      printf ("No Instruction Cache access.\n");
    }
    if ( (cache.data_hit+cache.data_miss) != 0 ){
      printf ("Run %d : Data Hit Ratio =\t %d%% (%d em
                              %d)\n",l+1,cache.data_hit*100/(cache.
                              data_hit+cache.data_miss),cache.
                              data_hit,(cache.data_hit+cache.
                              data_miss));
    }else{
      printf ("No Data Cache access.\n");
    }
  }
}
```

Referências

[1] Andrew S. Tanenbaum: *Structured Computer Organization*, 4.ed, Prentice Hall

[2] Julian Seward: *The Cacheprof home page - http://www.cacheprof.org*, Cambridge, UK

[3] Tobias Oetiker: *The Not So Short Introduction to LaTeX 2ε*, Dep. Electrical Engineering, Federal Institute of Technology - Swiss