Solutions to exercises on Memory Hierarchy

J. Daniel García Sánchez (coordinator)
David Expósito Singh
Javier García Blas

Computer Architecture
ARCOS Group
Computer Science and Engineering Department
University Carlos III of Madrid

1. Exam exercises

Exercise 1 *(June 2015). Assume that you have a computer with 1 clock cycle per instruction (1 CPI) when all accesses to memory are in cache. The only accesses to data come from load and store instructions. Those accesses account for 25 % of the total number of instructions. Miss penalty is 50 clock cycles and miss rate is 5 %.*

Determine the speedup obtained when there is no cache miss compared to the case when there are cache misses.

Solution 1

Given:

- c_p: Processor cycles.
- c_w: Wait cycles.
- T: Clock period.
- IC: Amount of instructions o *Instruction Count*.
- CPI: Cycles per instruction.
- a_i: Amount of accesses produced by instructions.
- a_d: Amount of accesses produced by data.
- m: Miss rate.
- p_m: Miss penalty.

In case of not considering misses:

$$T_{cpu} = (c_p + c_w) \cdot T = (IC \cdot CPI + 0) \cdot T = IC \cdot 1 \cdot T = IC \cdot T$$

Including misses:

$$c_w = IC \cdot (a_i + a_d) \cdot m \cdot p_m = IC(1 + 1 \cdot 0{,}25) \cdot 0{,}05 \cdot 50 = IC \cdot 1{,}25 \cdot 0{,}05 \cdot 50 = 3{,}125$$

ARCOS@uc3m

The CPU time will be:

$$T_{cpu} = (IC \cdot 1 + IC \cdot 3{,}125) \cdot T = 4{,}125 \cdot IC \cdot T$$

The speedup will be:

$$S = \frac{4{,}125 \cdot IC \cdot T}{IC \cdot T} = 4{,}125$$

Exercise 2 *(October 2014). Consider the following global variables definition:*

```
const unsigned int max = 1024 * 1024;
double x[max];
double y[max];
double z[max];
double vx[max];
double vy[max];
double vz[max];
```

And the following function:

```
void update_positions(double dt) {
  for (unsinged int i=0; i<max; ++i) {
    x[i] = vx[i] * dt + x[i];
    y[i] = vy[i] * dt + y[i];
    z[i] = vz[i] * dt + z[i];
  }
}
```

Consider a system with a 4 ways set associative L1 cache with a size of 32 KB and a line size of 64 bytes. L2 cache is 8 ways set associative with a size of 1 MB and a line size of 64 bytes. Replacement policy is LRU.

Arrays are consecutively stored in memory and first of them starts at 1024 multiple address.

1. *Determine the hit rate for L1 and L2 caches for running function* ***update_positions()****. What is the global hit rate?*
2. *Modify code applying the array merging optimization.*
3. *Repeat computations from the first question for the code resulting from the second question.*
4. *Assume that a hit at L1 cache requires 4 cycles and that L2 cache requires 14 cycles. Also assume that penalty associated to bringing a block from main memory to L2 cache is 80 cycles. Which is the average access time in each case?*

Solution 2

Point 1 The loop access pattern is:

```
vx[i], x[i], x[i], vy[i], y[i], y[i], vz[i], z[i], z[i], ...
```

Each of the arrays has 2^{20} positions of 8 bytes each. This results in a total of 8 MB per array. Each line of the cache has 64 bytes, which allows hosting 8 elements of the array.

The L1 cache has 4 ways, so each way is 8KB (2^{13} bytes), which results in $\frac{2^{13}}{2^6} = 2^7$ sets.

Due to each array has 2^{23} bytes (which is multiple of the way size), each **i** index of all arrays corresponds to the same cache set. The sequence of hits and misses in the cache will be:

- **VX[0]** → Miss. Selecting set 0.

- **X[0]** → Miss. Selecting set 1.
- **X[0]** → Hit.
- **VY[0]** → Miss. Selecting set 2.
- **Y[0]** → Miss. Selecting set 3.
- **Y[0]** → Hit.
- **VZ[0]** → Miss. Selecting set 0. Evicted **vx**.
- **Z[0]** → Miss. Selecting set 1. Evicted **x**.
- **Z[0]** → Hit.

When you move to position 1, the same sequence of misses and hits is repeated. So in total you have:

$$h_{L1} = \frac{3}{9} = \frac{1}{3}$$

For the L2 cache, only accesses occur in cases where there is a miss in the L1 cache. For position 0, 6 failures occur, but for positions 1 through 7, 6 hits are produced for each element.

$$h(L2) = \frac{42}{48} = \frac{7}{8}$$

The hit rate will be:

$$H = 1 - M$$

Where:

$$M = m_{h1} \cdot m_{h2} = \frac{2}{3} \cdot \frac{1}{8} = \frac{1}{12}$$

$$H = \frac{11}{12}$$

Point 2 Applying the required optimization, the code is:

```
const unsigned int max = 1024 * 1024;
struct part {
  double x, y, z, vx, vy, vz;
};
part vec[max];

void update_postions(double dt) {
  for (unsinged int i=0; i<max; ++i) {
    vec[i].x = vec[i].vx * dt + vec[i].x;
    vec[i].y = vec[i].vy * dt + vec[i].y;
    vec[i].z = vec[i].vz * dt + vec[i].z;
  }
}
```

Point 3 The access pattern in this case is:

`vec[i].vx, vec[i].x, vec[i].x, vec[i].vy, vec[i].y, vec[i].y, vec[i]. vz, vec[i].z, vec[i]. z, ...`

Since each position of the array needs 6 positions and a cache entry occupies 8 values, in 3 lines of cache will fit 4 complete positions of the array.

- Index 0 produces 1 miss and 8 hits.
- Index 1 produces 2 hits, 1 miss and 6 hits.
- Index 2 produces 4 hits, 1 miss and 4 hits.
- Index 3 produces 9 hits.

This pattern is repeated throughout the array. Thus:

$$h_{L1} = \frac{33}{36} = \frac{11}{12}$$

For L2 cache, all misses in L1 cache are also misses in the L2 cache. Therefore:

$$h_{L2} = 0$$

And the global rate:

$$H = 1 - M$$

$$M = m_{h1} \cdot m_{h2} = \frac{1}{12}$$

$$H = \frac{11}{12}$$

Point 4 For the original case:

$$T = 4 + (\frac{2}{3}) \cdot (14 + (\frac{1}{8}) \cdot 80) = 4 + (\frac{2}{3}) \cdot 24 = 4 + 16 = 20$$

For the second case:

$$T = 4 + (\frac{1}{12}) \cdot (14 + 1 \cdot 80) = 4 + (\frac{1}{12}) \cdot 94 = 4 + 7{,}83 = 11{,}83$$

Exercise 3 *(October 2014) Consider the following global variables definition:*

```
const unsigned int max = 1024 * 1024;
double x[max];
double y[max];
double z[max];
double vx[max];
double vy[max];
double vz[max];
```

And the following function:

```
void update_positions(double dt) {
  for (unsinged int i=0; i<max; ++i) {
    x[i] = vx[i] * dt + x[i];
  }
  for (unsinged int i=0; i<max; ++i) {
    y[i] = vy[i] * dt + y[i];
  }
  for (unsinged int i=0; i<max; ++i) {
    z[i] = vz[i] * dt + z[i];
  }
}
```

Consider a system with a 4 ways set associative L1 cache with a size of 32 KB and a line size of 64 bytes. L2 cache is 8 ways set associative with a size of 1 MB and a line size of 64 bytes. Replacement policy is LRU.

Arrays are consecutively stored in memory and first of them starts at 1024 multiple address.

1. *Determine the hit rate for L1 and L2 caches for running function* ***update_positions()****. What is the global hit rate?*
2. *Modify code applying the loop merging optimization.*
3. *Repeat computations from the first question for the code resulting from the second question.*
4. *Assume that a hit at L1 cache requires 4 cycles and that L2 cache requires 16 cycles. Also assume that penalty associated to bringing a block from main memory to L2 cache is 80 cycles. Which is the average access time in each case?*

Solution 3

Point 1 Three loops are executed consecutively.

The first loop has an access pattern:

`vx[i], x[i], x[i]`

The other two loops have similar access patterns.

Each of the arrays has 2^{20} positions of 8 bytes each. This gives a total of 8 MB per array. Each line of the cache has 64 bytes, which allows hosting 8 elements of the array.

L1 cache has 4 ways, so each way is 8KB (2^{13} bytes), providing $\frac{2^{13}}{2^6} = 2^7$ sets.

Given that each array has 2^{23} bytes (which is multiple of the way size), each **i** position of all arrays corresponds to the same cache set. The sequence of hits and misses in the cache will be:

- **VX[0]** → Miss. Selecting set 0.
- **X[0]** → Miss. Selecting set 1.
- **X[0]** → Hit.

The next 7 elements of the vectors will each produces 3 hits. The same will happen with the other two loops.

Therefore the hit rate for level 1 cache will be:

$$h_{L1} = \frac{22}{24} = \frac{11}{12}$$

At L2 cache, only the L1 cache accesses will be failures. In this case, all the accesses will produce failures.

$$h_{L2} = 0$$

The global hit rate is:

$$H = 1 - M$$

Where:

$$M = m_{L1} \cdot m_{L2} = \frac{1}{12}$$

$$H = \frac{11}{12}$$

Point 2 Applying the required optimization, the code looks like:

```
void update_positions(double dt) {
  for (unsigned int i=0; i<max; ++i) {
    x[i] = vx[i] * dt + x[i];
    y[i] = vy[i] * dt + y[i];
    z[i] = vz[i] * dt + z[i];
  }
}
```

Point 3 The loop access pattern is:

```
vx[i], x[i], x[i], vy[i], y[i], y[i], vz[i], z[i], z[i], ...
```

Each of the arrays has 2^{20} positions of 8 bytes each. This results in a total of 8 MB per array. Each line of the cache has 64 bytes, which allows hosting 8 elements of the array.

L1 cache has 4 ways, so each way is 8KB (2^{13} bytes), which results in $\frac{2^{13}}{2^6} = 2^7$ sets .

Since each array has 2^{23} bytes, each position i of all arrays corresponds to the same cache set. The sequence of hits and misses in the cache will be:

- **VX[0]** → Miss. Selecting set 0.
- **X[0]** → Miss. Selecting set 1.
- **X[0]** → Hit.
- **VY[0]** → Miss. Selecting set 2.
- **Y[0]** → Miss. Selecting set 3.
- **Y[0]** → Hit.
- **VZ[0]** → Miss. Selecting set 0. Evicted **vx**.
- **Z[0]** → Miss. Selecting set 1. Evicted **x**.
- **Z[0]** → Hit.

When we move to position 1, the same sequence of misses and hits is repeated. So in total we have:

$$h_{L1} = \frac{3}{9} = \frac{1}{3}$$

For the L2 cache, only accesses occur in cases where there is a miss in the L1 cache. For position 0, 6 misses occur, but for positions 1 through 7, 6 hits are produced for each position.

$$h_{L2} = \frac{42}{48} = \frac{7}{8}$$

The global hit rate is:

$$H = 1 - M$$

where:

$$M = m_{L1} \cdot m_{L2} = \frac{2}{3} \cdot \frac{1}{8} = \frac{1}{12}$$

$$H = \frac{11}{12}$$

Point 4 In the original case, the average access time is:

$$T = 4 + \frac{1}{12} \cdot (16 + 1 \cdot 80) = 4 + \frac{96}{12} = 4 + 8 = 12$$

In the modified case, the average access time is:

$$T = 4 + \frac{2}{3} \cdot (16 + 80 \cdot \frac{1}{8} = 4 + \frac{2}{3} \cdot 26 = 4 + 17{,}33 = 21{,}33$$

Exercise 4 *(October 2013) Given the following code fragment:*

```
struct particle {
  double x, y;
  double ax, ay;
  double vx, vy;
  double mass;
  double load;
};

void move(particle p[], int n, double t) {
  for (int i=0; i<n; ++i) {
    p[i].x += p[i].vx * t + 0.5 *p[i].ax * t * t;
  }
  for (int i=0; i<n; ++i) {
    p[i].y += p[i].vy * t + 0.5 * p[i].ay * t * t;
  }
}
```

Function ***move()*** *is executed for an array of 1000 particles in a system with a L1 data cache with a size of 32 KB and 4 ways set associative. The block size is 64 bytes. Assume that array* ***p*** *is aligned to an address which is 64 multiple.*

You are asked to determine:

1. *The number of cache misses.*
2. *The average miss rate.*
3. *Propose an alternative code using loop merging. What is the new miss rate?*
4. *Propose an alternative code, reorganizing the code in multiple parallel arrays and without loop merging. What is the new miss rate?*
5. *Do you think that it is convenient to apply loop merging in the previous section? Reason your answer.*

Solution 4

Point 1 In each cache block, 8 double values can be stored. On the other hand, each particle also requires 64 bytes. In addition, the complete array has 1000 elements so it does not fit completely in the cache.

During the first loop, each iteration for reading **p[i].x**, the code produces one miss and 3 hits, reading **p[i].ax** and **p[i].vx**, and writing **p[i].x**.

During the second loop exactly the same thing occurs.

Therefore, a total of 2000 misses are generated.

Point 2 The miss rate is:

$$m = \frac{2000}{8000} = 0{,}25$$

Point 3 If we apply the loop-merge technique, we obtain:

```
void move(particle p[], int n, double t) {
  for (int i=0; i<n; ++i) {
    p[i].x += p[i].vx * t + 0.5 *p[i].ax * t * t;
    p[i].y += p[i].vy * t + 0.5 * p[i].ay * t * t;
  }
}
```

In each iteration of the loop, access to **p[i].x** generates a miss but the other accesses generate hits, reaching a miss rate of:

$$m = \frac{1000}{8000} = 0{,}125$$

Point 4 We obtain the following code:

```
struct particle {
  double x[1000], y[1000];
  double ax[1000], ay[1000];
  double vx[1000], vy[1000];
  double mass[1000];
  double weight[1000];
};

void move(particle * p, int n, double t) {
  for (int i=0; i<n; ++i) {
    x[i] += vx[i] * t + 0.5 *ax[i] * t * t;
  }
  for (int i=0; i<n; ++i) {
    y[i] += vy[i] * t + 0.5 * ay[i] * t * t;
  }
}
```

Now we have independent arrays. There are 8 positions of an array in each cache block.

In the first loop, the first iteration produces 3 misses and one hit. The following 7 iterations produce 4 hits per iteration (28 hits in total). This pattern is repeated 125 times in each loop.

Hit rate:

$$h = 3 \cdot 125 \cdot \frac{2}{8000} = \frac{750}{8000} = 0{,}094$$

Point 5 It is not convenient because the number of parallel arrays is greater than the number of ways.

Exercise 5 *(October 2013) Consider the following code:*

Access to **v**	Misses	Access to **b**	Misses
0,0	YES	0	YES
0,2	No because it is on the same cache line as **v[0,0]**	0	No because it is on the same cache line as **b[0,0]**
...		0	No
0,7	NO	0	No
0,8	SI	0	No
0,10	NO	0	No
... 0,16	YES	0	No
... 0,1022	NO	0	No
1,0	YES	1	NO
1,1	No, Because it is on the same cache line as **v[1,0]**	1	NO
... 2,0	YES	2	NO
... 8,0	YES	8	YES
... 31,0	YES	31	NO

Cuadro 1: Access pattern of the Exercise 5.

```
for (i=0; i<64; i++)
  for (j=0; j<1024; j=j+2 )
    v[i][j] = v[i][j] * v[i][j+1] + b[i]
```

Assume an architecture with 256 KB size cache and a block size of 64 bytes and word size of 8 bytes. Cache memory is fully associative and uses an LRU replacement policy. Let ***v*** *and* ***b*** *be 8 byte real number matrices stored by rows (C programming language style) and sizes* 64×1024 *(****v****),* 64 *(****b****). Assume that index variables are stored in processor registers and cache is initially empty.*

Your are asked to determine:

1. *The number of cache misses.*
2. *The average miss rate.*
3. *Reason whether there is some time a replacement of a previously loaded cache line. It is not needed to identify the replacements but it is enough to reason whether they could happen or not.*
4. *Reason if the loop exchanges optimization technique would improve the average miss rate for the cache (it is not needed to compute it).*

Solution 5

Point 1 Cache of 256 KB $\rightarrow 2^{15}$ elements $= 2^{12}$ blocks
Block if 64B = 8 elements
Table 1 shows the access pattern.

Cache misses occur when accessing **v[0,0]**, **v[0,8]**, **v[0,16]**, ... **v[1,0]**, **v[1,8]**, **v[0,16]**, ... **[2,0]**, ... **v[31,0]**, ... and there are hits in all the other cases. Since the array has 64×1024 elements and there is only one miss in 8, the total number of misses is $\frac{64 \cdot 1024}{8} = 8192$.

In **b**, there are misses at 1, 8, 16, 24, 32, 40, 48 and 56. There are 8 misses in total.

The total number of misses is $8192 + 8 = 8200$.

Point 2 Total accesses.

In each instruction, there are 4 accesses and the calculation instruction is done $\frac{64 \cdot 1024}{2}$ times. Therefore, we have $\frac{4 \cdot 64 \cdot 1024}{2} = 131072$ accesses and the miss rate is:

$$m = \frac{8200}{131072} = 0{,}0625 \Rightarrow 6,25\,\%$$

Point 3 Cache size 256 KB = 2^{15} elements = 2^{12} and the matrix v has $64 \cdot 1024 = 2^{16}$ elements = 2^{13} blocks.

Therefore, the **v** array does not fit in the cache. There will be a replacement of the line containing the **v [0,0]** position when we get approximately to the middle of the **v** array.

If the matrix **b** does not exist, the replacement would occur on reaching the middle of the matrix (because only half of the matrix fits in the cache), that is, in the position **v [32,0]** .

Given that the array **b** also occupies cache and when we reach the element **v [32,0]** we have occupied 4 lines of cache with the array **b** (elements from **b[0]** to **b[32]**), we can conclude that the replacement will produce 4 lines of cache before reaching **v[32,0]**. This is in the position **v**$[31, 8 \cdot \frac{1024}{8} - 4]$ = **v[31,992]**. This number is not explicitly asked in the exercise.

Similarly, a replacement of the **b [0]** cache line will occur at approximately the same time.

There will also be successive replacements of **v [0,8]**, **v [0,16]**, ... as space is needed for positions of **v**.

Point 4 The loops exchange would decrease the spatial locality in the accesses to **v**, increasing the miss rate.

Exercise 6 *(October 2013) Given the following code fragment:*

```
int a[100];
int b[100];
for (i=0;i<100;i++)
  a[i]=a[i]+5;
for (i=0;i<100;i++){
  if (i>90)
    c=c+1;
  else
    b[i]=a[i]*3;
}
```

You are asked:

1. *Describe compiler cache optimizations that allow to improve memory access time for this code. Rewrite the code accordingly.*

2. *Consider a computer with cache memory with a line size of 32 bytes. Initially, cache is empty. Which is the cache miss rate when executing the optimized program obtained from the previous question? Consider only compulsory misses for vectors **a** and **b** (there are no capacity of conflict misses).*

3. *Consider now a computer with a two-level cache. Both L1 and L2 are unified caches. Clock cycle is 1ns and CPI=1.3. For L1 cache, assume a miss rate of 10 % and a miss penalty of 10ns. For L2 cache hit rate is 95 % and penalty is 80 ns. Assume access time to L1 cache is 1ns. You are requested to compute program execution time for a program with IC instructions where 50 % are read/write instructions.*

Solution 6

Point 1 The compiler cache optimizations that can be applied are as follows:

- **Array merge**: arrays **a** and **b** being of the same size, can be merged into one in such a way that a consecutive memory space is defined to store the two elements of the array, improving the spatial locality and thus reducing cache failures.
- **Loop merge**: you can merge the loops in lines 1 and 3 that iterate over the arrays **a** and **b** respectively in a single loop, so that the accesses to vector **a** perform all operations, without having a new cache failure.
- **Branch linearization**: the compiler can change the sense of the condition if it detects that the branch prediction strategy will generate many misses since it does not match the result of the condition.

Once the cache optimizations were applied, a possible resulting code would be as follows:

```
struct mi_array {
  int a;
  int b;
};

struct my_array array[100];
for (i=0;i<100;i++) {
  array[i].a= array[i].a+5;
  if (i<=90)
    array[i].b=array[i].a*3;
  else
    c=c+1;
}
```

Point 2 Since the cache line size is 32 B on each cache line, they fit $\frac{32}{4} = 8$ elements of type **int**.

- Accesses **a**
  - **array[0].a** → miss, transferred 32 B. **array[0].a, array[0].b, array[1].a, array[1].b, array[2].a, array[2].b, array[3].a, array[3].b**
  - **array[1].a** ... **array[3].a** → hit.
  - **array[4].a** → miss, transferred 32 B. **array[4].a,array[4].b,array[5].a,array[5].b,array[6].a,array[6]**
    array[7].a, array[7].b
  - **array[5].a** ... **array[7].a** → hit.
  - ...
  - **array[96].a** → miss, transferred 32 B. **array[96].a, array[96].b, array[97].a, array[97].b, array[98].a, array[98].b, array[99].a, array[99].b**
- Accesses **b**

- **array[0].b** → There is no cache miss since the access is made immediately after the read miss of **array [0]**.
- **array[1].b** → There is no cache miss since the access is made immediately after the read miss of **array [0]**.
- ...
- **array[99].b** → There is no cache miss since the access is made immediately after the read miss of **array [0]**.

No cache misses of **b**. There are 90 accesses to **b** and no misses.

The number of total accesses is $100 + 100 + 90 + 90 = 380$. 1 of every 4 accesses to the vector **a** is a cache miss. Since there are 100 accesses, 25 are misses.

Miss rate.

$$m = \frac{25}{380} = 6{,}57\,\%$$

Point 3

$$t = t_{cpu} + t_{mem}$$

$$t_{cpu} = CI \cdot CPI \cdot T = CI \cdot 1{,}3 \cdot 1 = 1{,}3 \cdot CI$$

$$t_{mem} = t_{fetch} + t_{data}$$

$$t_{fetch} = t_{access} + t_{missesL1} \cdot (penalty_{L1} + (t_{missesL2} \cdot penalty_{L2})) = 1 + (0{,}1 \cdot (10 + (0{,}05 \cdot 80)) = 2{,}4$$

The average number of cycles due to the fetch is:

$$t_{fetch} = CI \cdot 2{,}4 \cdot T = 2{,}4 \cdot CI$$

$$t_{data} = t_{access} + (t_{missesL1} \cdot (penalty_{L1} + (t_{missesL2} \cdot penalty_{L2})) = 1 + (0{,}1 \cdot (10 + (0{,}05 \cdot 80)) = 2{,}4$$

$$t_{data} = CI \cdot \frac{num\_ref\_memory}{IC} \cdot 2{,}4 \cdot T = 2{,}4 \cdot CI \cdot 0{,}5 \cdot 1 = 1{,}2 \cdot CI$$

$$t_{mem} = 2{,}4 \cdot CI + \cdot 1{,}2 \cdot CI = 3{,}6 \cdot CI$$

$$Total = 1{,}3 \cdot CI + 3{,}6 \cdot CI = 4{,}9 \cdot CI$$

Exercise 7 *(June 2011) Consider the following code fragment:*

```
double a[256][256], b[256][256], c[256][256], d[256][256];
// ...
for (int i=0;i<256;++i)
  for (int j=0;j<256;++j) {
    a[i][j] = b[i][j] + c[i][j]
  }
}
for (int i=0;i<256;++i)
  for (int j=0;j<256;++j) {
    d[i][j] = b[i][j] - c[i][j]
  }
}
```

We want to run this code in a computer with fully associative 16 KB level 1 cache. Replacement policy is LRU with a 64 bytes line size. Level 1 cache misses require 16 clock cycles. Besides, level 2 cache always gives hits for this code.

Assume that write misses in level 1 cache are directly sent to a write buffer and do not generate stall cycles.

You are asked:

1. *Determine the hit rate for the code segment, assuming that variables* ***i*** *and* ***j*** *are assigned to processor registers and that matrices* ***a***, ***b***, ***c*** *and* ***d*** *fully reside at level 2 cache.*
2. *Determine the memory access time assuming that accesses to level 1 cache require a clock cycle.*
3. *Propose a code transformation that can be generated by a compiler to improve hit rate, showing the resulting code in the C programming language.*
4. *Determine the new hit rate and the resulting average access time to memory.*

Solution 7

Point 1 Due to the line size, 64 bytes, and that all arrays contain values of type **double**, each line of the cache can store 8 values. Since the L1 cache has a size of 16 KB, we can store $\frac{2^{14}}{2^6} = 2^8$ lines.

Each array stores $2^8 \dot{2}^8 = 2^{16}$ elements, which requires $\frac{2^{16}}{2^3} = 2^{13}$ entries in the cache.

In the first loop, the access pattern is:

- **b[0][0]**, **c[0][0]**, **a[0][0]**
- **b[0][1]**, **c[0][1]**, **a[0][1]**
- ...
- **b[0][7]**, **c[0][7]**, **a[0][7]**
- **b[0][8]**, **c[0][8]**, **a[0][8]**
- ...
- **b[0][255]**, **c[0][255]**, **a[0][255]**
- **b[1][0]**, **c[1][0]**, **a[1][0]**
- ...

Therefore, for every 8 iterations of the inner loop:

- 16 reads (8 for **b** and 8 for **c**). Of these, 2 are misses and the other 14 are hits.
- 8 writes in **a**. However, when writes are sent to the buffer, they do not generate misses.

When the second loop is started, the cache entries containing the values of the arrays **b** and **c** have been flushed from the cache, so the miss and hit ratios are the same.

In the case of writes, all writes generate cache misses. So:

$$h_{write} = 0$$

In case of writes, the hit rate is:

$$h_{read} = \frac{14}{16} = \frac{7}{8}$$

Point 2 For write accesses, all accesses are a write misses and are handled in the write buffer, requiring a clock cycle. In total, $2^8 \cdot 2^8$ accesses are made in each of the two loops, resulting in a total of $2 \cdot 2^{16} = 2^{17}$ write accesses.

For read access, the average access time is represented by:

$$t_{read} = t_a + (1 - h_{read}) \cdot t_f$$

The access time t_a is 1 cycle. The penalty for reading failures is 16 cycles. Therefore, we have that the average reading time is:

$$t_{read} = 1 + \frac{1}{8} \cdot 16 = 3$$

Point 3 The merging-loop technique can be applied:

```
for (int i=0;i<256;i++) {
  for (int j=0;j<256;j++) {
    a[i][j] = b[i][j] + c[i][j];
    d[i][j] = b[i][j] - c[i][j];
  }
}
```

Point 4 In this case, the second accesses to **b** and **c** always generate hits.

Now, for every 8 iterations of the inner loop:

- 32 read. From them, 30 are hits and two are read misses.
- 16 writes.

As in the previous case, the writing hit rate (h_{escr}) remains 0. In this case, the reading hit rate is:

$$h_{read} = \frac{30}{32} = \frac{15}{16}$$

For the case of reads, using the new hit rate, we obtain that:

$$t_{read} = 1 + \frac{1}{16} \cdot 16 = 2$$

Exercise 8 *(May 2011) Consider an architecture with two cache levels and the following characteristics:*

Memory	*Access time (ns)*	*Hit rate*
L1	*2*	*0.8*
L2	*8*	*0.9*
RAM	*100*	*1*

The computer runs a program fully residing in memory (no disk access). You are asked:

1. *Assuming that 100 % of memory accesses are write operations, calculate and justify the average access time for (a) a write through policy, and (b) a write back policy, both for L1 and L2 caches.*
2. *Consider both cache levels (L1 and L2) as a single global cache. Calculate the average access time and hit rate for this global cache.*
3. *Given the following code:*

```
for (i=0;i<1000;i=i+32){
  a[i] = a[i+8] + a[i+16];
}
```

Size for each entry in array a is 8 bytes and block size is 64 bytes. The loop index is stored in a processor register. You are asked to comment about the effect in performance of using a multi-bank cache with 4 banks. What effect could this approach have on time for each access and cache bandwidth for the given code?

Solution 8

Point 1 In the case of write-thought, since 100 % of the accesses are write operations, all of them are performed in main memory. For this reason, the access time will be in all cases:

$$t = 2 + 8 + 100 = 110ns$$

In the case of write-back, if a hit occurs, it is not necessary to write directly to memory. In this way, the access time will be:

$$t = 2 + (1 - 0{,}8) \cdot (8 + (1 - 0{,}9) \cdot 100) = 2 + 0{,}2 \cdot (8 + 0{,}1 \cdot 100) = 2 + 0{,}2 \cdot 18 = 2 + 3{,}6 = 5{,}6ns$$

Point 2 The average access time will be:

$$t = 2 + (1 - 0{,}8) \cdot 8 = 2 + 0{,}2 \cdot 8 = 2 + 1{,}6 = 3{,}6ns$$

And the hit rate:

$$h = 1 - (1 - 0{,}8) \cdot (1 - 0{,}9) = 1 - 0{,}2 \cdot 0{,}1 = 1 - 0{,}02 = 0{,}98.$$

Point 3 At each iteration of the loop, three different blocks are accessed, located in consecutive memory locations. By using a non-multibank cache, there would be 3 cache misses that would have to be dealt with sequentially. In the case of a multi-bank cache, it would be possible to access these three blocks in parallel.

The access time would be the same as before because this optimization would not Improves the access time in the cache.

The bandwidth would be the aggregate, since they would be making accesses in parallel. In this case, it would be 3 simultaneous accesses, so the bandwidth would be triple.