Chapter 9 Memory Architectures

1. Consider the function `compute_variance` listed below, which computes the variance of integer numbers stored in the array `data`.

```
1  int data[N];
2
3  int compute_variance() {
4    int sum1 = 0, sum2 = 0, result;
5    int i;
6
7    for(i=0; i < N; i++) {
8      sum1 += data[i];
9    }
10   sum1 /= N;
11
12   for(i=0; i < N; i++) {
13     sum2 += data[i] * data[i];
14   }
15   sum2 /= N;
16
17   result = (sum2 - sum1*sum1);
18
19   return result;
20 }
```

Suppose this program is executing on a 32-bit processor with a direct-mapped cache with parameters $(m,S,E,B) = (32,8,1,8)$. We make the following additional assumptions:

- An `int` is 4 bytes wide.
- `sum1`, `sum2`, `result`, and `i` are all stored in registers.
- `data` is stored in memory starting at address `0x0`.

Answer the following questions:

(a) Consider the case where `N` is 16. How many cache misses will there be?

Solution: First, note that each block stored in the cache is of size 8 bytes. Since each `int` is 4 bytes wide, we note that `data[0]` and `data[1]` will lie in the same cache block, so will `data[2]` and `data[3]`, and so on.

If `N` is 16, then we will suffer a cache miss in the first for loop on reading `data[i]` for every even `i` from 0 to 15. At the end of the first for loop, the entire array `data` will be in the cache. Thus, while executing the second for loop, we will never suffer a cache miss on any read.

Thus, the total number of cache misses is 8.

(b) Now suppose that `N` is 32. Recompute the number of cache misses.

Solution: If `N` is 32, `data[i]` and `data[i+16]` will map to the same cache block. Thus, during the first for loop, each read to `data[i+16]` will evict the block containing `data[i]`.

Thus, when we execute the second for loop, we will suffer a cache miss on each even entry in the array all over again, just as in the first for loop.

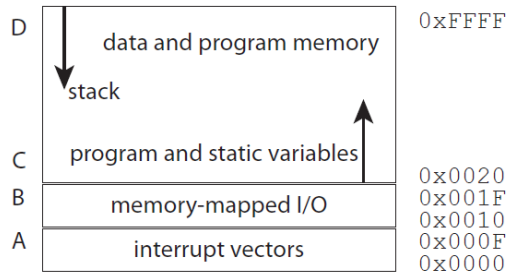
Therefore, the total number of cache misses in this case will be 16*2 = 32.

(c) Now consider executing for `N` = 16 on a 2-way set-associative cache with parameters $(m,S,E,B) = (32,8,2,4)$. In other words, the block size is halved, while there are two cache lines per set. How many cache misses would the code suffer?

Solution: The code would suffer 16 cache misses in the first for loop – one on reading each array entry. The reason the number of misses doubles is the smaller block size. In part (a), reading `data[2*i]` also moves `data[2*i+1]` into the cache, but this does not occur in the present case.

3. Consider the C program and simplified memory map for a 16-bit microcontroller shown below. Assume that the stack grows from the top (area D) and that the program and static variables are stored in the bottom (area C) of the data and program memory region. Also, assume that the entire address space has physical memory associated with it.

```
1  #include <stdio.h>
2  #define FOO 0x0010
3  int n;
4  int* m;
5  void foo(int a) {
6     if (a > 0) {
7       n = n + 1;
8       foo(n);
9     }
10 }
11 int main() {
12    n = 0;
13    m = (int*)FOO;
14    foo(*m);
15    printf("n = %d\n", n);
16 }
```



You may assume that in this system, an `int` is a 16-bit number, that there is no operating system and no memory protection, and that the program has been compiled and loaded into area C of the memory.

(a) For each of the variables `n`, `m`, and `a`, indicate where in memory (region A, B, C, or D) the variable will be stored.

Solution: `n` and `m` will be in C, and `a` will be in D.

(b) Determine what the program will do if the contents at address 0x0010 is 0 upon entry.

Solution: The program will print 0 and exit.

(c) Determine what the program will do if the contents of memory location 0x0010 is 1 upon entry.

Solution: The program will overflow the stack, overwriting the program and static variables region. This will most likely cause it to start executing arbitrary data as if it were a program, which will produce highly unpredictable results.

4. Consider the following program:

```
1  int a = 2;
2  void foo(int b) {
3    printf("%d", b);
4  }
5  int main(void) {
6    foo(a);
7    a = 1;
8  }
```

Is it true or false that the value of `a` passed to `foo` will always be 2? Explain. Assume that this is the entire program, that this program is stored in persistent memory, and that the program is executed on a bare-iron microcontroller each time a reset button is pushed.

Solution: False. The memory for storing `a` is initialized with the value 2 in the program image created by the compiler. Therefore, this memory location will be assigned value 2 when the program is loaded into the persistent memory. After the first run of the program, the memory location will have 1. If the reset button is pushed again, then `foo` will be passed the value 1.