332 Advanced Computer Architecture

Unassessed tutorial exercise 7

Exercise 7.1. The cache design space

This question is intended to encourage you to think about the cache design space. Talk about it with your friends and with the tutorial assistants:

1. How do you think miss-rate varies with increasing block size
   - if the total number of blocks in the cache remains constant?
   - if the total capacity of the cache remains constant?
2. Some machines have very large cache blocks. How might miss penalty for large blocks be reduced?
3. Should the index be the virtual address or the physical address? Discuss. Is there a consistency problem? How might it be solved?
4. Should the tag be the virtual address or the physical address? Discuss.
5. Should the second-level cache always contain everything held in the first-level cache (the "multi-level inclusion" property)?
6. How could one restructure a program to get better performance with a small cache? Here is an example:

   ```
   for (i = 0; i < 500; i++)
     for (j = 0; j < 500; j++)
       for (k = 0; k < 500; k++)
         C[i][j] += A[i][k] * B[k][j];
   ```

7. Construct an example program would have a very high rate of conflict misses with a two-way set-associative cache, but which would have a high hit rate with a three-way set-associative cache.

Exercise 7.2. Update vs. Invalidate

The performance differences for write-invalidate and write-update schemes can arise from both bandwidth consumption and latency. Assume a memory system with 64-byte cache blocks.

1. Write two parallel loops to illustrate the bandwidth differences between update and invalidate schemes. One loop should make update look much better, the other should make invallidate look much better.

2. Write a parallel loop to illustrate the latency advantage of an update scheme versus an invalidate scheme.

3. Show, by example, that when contention is taken into account, the latency of update may actually be worse. Assume a bus-based machine with a 50-cycle memory and snoop transactions.

Exercise 7.3. Degrees of consistency

Consider the following program fragment:

```
/* processor 1 */
A = 0;
  :
A = 1;
if (B == 0) {
  P();
}
```

```
/* processor 2 */
B = 0;
  :
B = 1;
if (A == 0) {
  P();
}
```

Assume that procedure `P` is not called from elsewhere.

1. Assume that the program is running on an idealised shared-memory machine.

   Can both processors execute `P` simultaneously? Explain your answer carefully.

2. Assume that the two fragments are running on different CPUs in a coherent-cache shared-memory multiprocessor, using an invalidation protocol.

   Can both processors execute `P` simultaneously? Explain your answer carefully.

   What precautions are needed to preserve the expected behaviour?

3. Assume that the two fragments are running on different CPUs in a coherent-cache shared-memory multiprocessor, using an *update* protocol.

   Can both processors execute `P` simultaneously? Explain your answer carefully.

   What precautions are needed to preserve the expected behaviour?

4. Some multiprocessors do not guarantee the expected behaviour for this example, but instead guarantee consistency only when CPUs synchronise (e.g. after passing through a LOCK or BARRIER). Explain why this might offer a performance advantage.

Paul Kelly, Imperial College, December 1999

Notes on the solutions to Exercise 7

Exercise 7.1. The cache design space

This question is intended to encourage you to think about the cache design space. Talk about it with your friends and with the tutorial assistants:

1. How do you think miss-rate varies with increasing block size

   - if the total number of blocks in the cache remains constant?

     A: Provided the cache miss penalty is unchanged, the miss rate should be improved thanks to spatial locality - how much depends on the application. However, increasing the block size means that the amount of data loaded into the cache but not actually used will increase - DRAM-cache bandwidth is wasted. If this bandwidth is fixed, increasing the block size increases the miss penalty and there is a difficult application-dependent tradeoff.

   - if the total capacity of the cache remains constant?

     A: Generally speaking, programs with high spatial locality should benefit, but programs with high temporal locality which is not also spatially-localised will suffer. However if the working set is small so it easily fits within the cache (or the cache is big), we should always see an improvement.

2. Some machines have very large cache blocks. How might miss penalty for large blocks be reduced?

   A: We obviously don't have to wait for the whole cache block to arrive - the load can proceed as soon as the bytes it requests arrive. In fact we could transfer those bytes first, even if the access is to the middle of a block.

3. Should the index be the virtual address or the physical address? Discuss. Is there a consistency problem? How might it be solved?

   A: It's faster to pass the virtual address direct to the cache without having to translate it. We have two consistency problems - (1) different processes commonly map the same virtual address to physical physical addresses, and (2) if two different processes map the same physical page at different virtual addresses (eg using mmap).

4. Should the tag be the virtual address or the physical address? Discuss.

   A: If both index and tag are virtual we have a problem when switching from one process to another due to (1) above. One solution is to flush the cache on context switches. Another is to augment the tag with an "address space identifier", essentially a process id.

   Even if the index is virtual, we can do the translation in parallel with cache access, so we can store the physical address in the tag and check that. This solves problem (1). It does not solve problem (2).

   A popular solution is to notice that if each bank of the cache is at most one page in size, address translation will not change the index - it will only change higher-order bits. So with 4KByte pages, the maximum size for a direct-mapped cache would be only 4KBytes - but increasing the amount of associativity allows a larger cache. Thus, it is common for level-1 caches to be more associative than simple benchmark studies would justify.

5. Should the second-level cache always contain everything held in the first-level cache (the "multi-level inclusion" property)?

   A: The main problems arise from writes, especially since the L1 block size is normally smaller than the L2 block size. Without multi-level inclusion (MLI), we would need a separate path from the L1 cache to main memory. MLI ensures that L1 can always write back/through to hit in the L2 cache, so cache-memory transfers occur only between L2 and memory, and only at L2's block size.

6. How could one restructure a program to get better performance with a small cache? Here is an example:

```
for (i = 0; i < 500; i++)
  for (j = 0; j < 500; j++)
    for (k = 0; k < 500; k++)
      C[i][j] += A[i][k] * B[k][j];
```

   A: See Chapter 5 of the lecture notes.

7. Construct an example program would have a very high rate of conflict misses with a two-way set-associative cache, but which would have a high hit rate with a three-way set-associative cache.

   A: We need a loop which accesses three different locations, which share the same low-order address bits. E.g.

```
for (i = 0; i < 500; i++)
 A[i] = A[i] + A[i+32768] + A[i+65536]
```

Exercise 7.2. Update vs. Invalidate

The performance differences for write-invalidate and write-update schemes can arise from both bandwidth consumption and latency. Assume a memory system with 64-byte cache blocks.

1. Write two parallel loops to illustrate the bandwidth differences between update and invalidate schemes. One loop should make update look much better, the other should make invallidate look much better.

2. Write a parallel loop to illustrate the latency advantage of an update scheme versus an invalidate scheme.

3. Show, by example, that when contention is taken into account, the latency of update may actually be worse. Assume a bus-based machine with a 50-cycle memory and snoop transactions.

Exercise 7.3. Degrees of consistency

Consider the following program fragment:

```
/* processor 1 */
A = 0;
⋮
A = 1;
if (B == 0) {
  P();
}
```

```
/* processor 2 */
B = 0;
⋮
B = 1;
if (A == 0) {
  P();
}
```

Assume that procedure P is not called from elsewhere.

1. Assume that the program is running on an idealised shared-memory machine.

   Can both processors execute P simultaneously? Explain your answer carefully.

   A: See textbook

2. Assume that the two fragments are running on different CPUs in a coherent-cache shared-memory multiprocessor, using an invalidation protocol.

   Can both processors execute P simultaneously? Explain your answer carefully.

   A: This can happen if the processors proceed with the test before waiting for an acknowledgement that the invalidation or update has finished.

   What precautions are needed to preserve the expected behaviour?

   A: The simple answer is to block the processor's subsequent memory operations until the store's acknowledgement has been received. Another approach with some followers, called "weak consistency", is to explain the memory system's semantics in the programmer's manual - explain that consistency is not guaranteed until specific synchronisation instructions are executed.

3. Assume that the two fragments are running on different CPUs in a coherent-cache shared-memory multiprocessor, using an *update* protocol.

   Can both processors execute P simultaneously? Explain your answer carefully.

   What precautions are needed to preserve the expected behaviour?

   A: This is a bit of a trick question - the answer is the same as above.

4. Some multiprocessors do not guarantee the expected behaviour for this example, but instead guarantee consistency only when CPUs synchronise (e.g. after passing through a LOCK or BARRIER). Explain why this might offer a performance advantage.

   This means that computation (including memory accesses) can be overlapped with invalidation and acknowledgement messages.

(Paul Kelly, Imperial College, November 1999)