Performance results

[§5.4] What cache line size is performs best? Which protocol is best to use?

Questions like these can be answered by simulation. However, getting the answer right is part art and part science.

Parameters need to be chosen for the simulator. The authors selected a single-level 4-way set-associative 1 MB cache with 64-byte lines.

The simulation assumes an idealized memory model, which assumes that references take constant time. Why is this not realistic?

The simulated workload consists of 6 parallel programs from the SPLASH-2 suite and one multiprogrammed workload, consisting of mainly serial programs.

Effect of coherence protocol

[§5.4.3] Three coherence protocols were compared:

- The Illinois MESI protocol (“Ill”, left bar).
- The three-state invalidation protocol (3St) with bus upgrade for S→M transitions. (This means that instead of rereading data from main memory when a block moves to the M state, we just issue a bus transaction invalidating the other copies.)
- The three-state invalidation protocol without bus upgrade (3St-BusRdX). (This means that when a block moves to the M state, we reread it from main memory.)
In our parallel programs, which protocol seems to be best?

Somewhat surprisingly, the result turns out to be the same for the multiprocessor workload.

The reason for this? The advantage of the four-state protocol is that no bus traffic is generated on E→M transitions. But E→M transitions are very rare (less than 1 per 1K references).
**Effect of cache line size**

[§5.4.4] Recall from Lecture 11 that cache misses can be classified into four categories:

- *Cold misses* (called “compulsory misses” in the previous discussion) occur the first time that a block is referenced.
- *Conflict misses* are misses that would not occur if the cache were fully associative with LRU replacement.
- *Capacity misses* occur when the cache size is not sufficient to hold data between references.
- *Coherence misses* are misses caused by the coherence protocol.

Coherence misses can be divided into those caused by *true sharing* and those caused by *false sharing*. False-sharing misses are those caused by having a line size larger than one word. Can you explain?

True-sharing misses, on the other hand, occur when a processor writes some words into a cache block, invalidating the block in another processor’s cache, after which the other processor reads one of the modified words.

How could we attack each of the four kinds of misses?

- To reduce capacity misses, we could
- To reduce conflict misses, we could
- To reduce cold misses, we could
- To reduce coherence misses, we could

If we increase the line size, the number of coherence misses might go up or down. Why?
Increasing the line size has other disadvantages.

- It increases conflict misses. Why?
- It increases bus traffic. Why?

So it is not clear which line size will work best.

Results for the first three applications seem to show that which line size is best?

For the second set of applications, Radix shows a greatly increasing number of false-sharing misses with increasing block size.
However, this is not the whole story. Larger line sizes also create more bus traffic.
With this in mind, which line size would you say is best?

**Invalidate vs. update**

[§5.4.5] Which is better, an update or an invalidation protocol?

At first glance, it might seem that update schemes would always be superior to write-invalidate schemes.

Why might this be true?

Why might this not be true?

When there are not many “external rereads,”

When there is a high degree of sharing,

For example, in a producer-consumer pattern,

Update and invalidation schemes can be combined (see §5.4.5).

Let’s look at real programs.
Where there are many coherence misses,
If there were many capacity misses,

So let’s look at bus traffic …
Note that in two of the applications, updates in an update protocol are much more prevalent than upgrades in an invalidation protocol.

Each of these operations produces bus traffic; therefore, the update protocol causes more traffic.

The main problem is that one processor tends to write a block multiple times before another processor reads it.

This causes several bus transactions instead of one, as there would be in an invalidation protocol.

In addition, updates cause problems in non-bus-based multiprocessors.

**Extending Cache Coherence**

[§6.6] There are several ways in which cache coherence can be extended. These ways interact with other aspects of the architecture.

Considering these issues yields a good understanding of where caches fit in the overall hardware/software architecture.
Shared-cache designs

Why not let one cache serve multiple processors? This could be a 1st-level cache, a 2nd-level cache, etc.

What would be the advantages?

- It eliminates the need for cache coherence.
- It may decrease the miss ratio. Why?
- It may reduce the need for a large cache. Why?
- It allows more effective use of long cache lines.

Examples of such designs include the Alliant FX-8 (ca. 1985), which shared a cache among 8 processors, and the Encore Multimax (ca. 1987), which shared a cache among two processors.

Today, shared caches are being considered for single-chip multiprocessors, with 4–8 processors sharing an on-chip first-level cache.

What are some disadvantages of shared caches?
- Latency.
• Bandwidth.

• Speed of cache.

• Destructive interference—the converse of overlapping working sets.

Using shared second-level caches moderates both the advantages and disadvantages of shared first-level caches.

**Virtually indexed caches and cache coherence**

[§6.6.2] Why not just store information in the cache by virtual address rather than physical address?

What would be the advantage of this?

[A TLB is like a cache that translates a virtual address of the form

\[(\text{page } #, \text{displacement})\]

to a physical address of the form

\[(\text{frame } #, \text{displacement}).\]

With large caches, it becomes increasingly difficult to do address translation in parallel with cache access. Why?]
Recall this diagram from Lecture 6.
What happens to the set number as the cache increases in size?

Rather than slow down every instruction by serializing address translation and cache access, it makes sense just to use a virtually addressed cache.

But virtually addressed caches have their own problems.

One problem occurs when memory is shared. Two processes may have different virtual names for the same block.

What’s wrong with this?

Why are duplicate copies of information bad? Think about what happens on a process switch. Assume

- The processor has been running Process 1.
- It then switches to Process 2.
- Later it switches back to Process 1.

This is called the *synonym problem*.
Solution:

- For each cache block, keep both a virtual tag and a physical tag. This allows a virtual address to be found, given a physical address, or vice versa.

- When a cache miss occurs, look up the physical address to find its virtual address (if any) within the cache.

- If the block is already in the cache, rename it to a new virtual address rather than reading in the block from memory.

Let's see how this works in practice.

Notes:

Step 4: We are looking for another copy of the same block being used by another process, with a different virtual address.

Step 6: Block at virtual address needs to be written back if it has been modified.

The newly fetched block's physical index may be the same as another block in cache; if so, that block needs to be written back too.
**Step 8:** If another process $q$ was using this block and tries to access it again, it will get a miss on virtual address, but will find it by physical address. What will it therefore do with this block?

Note that only physical addresses are put on the bus. Therefore, they can be snooped, and blocks can be invalidated (or updated) just like in coherence protocols for physically addressed caches.

Figure 6.24 in CS&G illustrates the organization of a virtually addressed cache:

Consider first a virtual address

- “ASID” means “address-space identifier,” which is often called a process identifier (PID) in other books.
- The diagram shows the “V-index” overlapping the (virtual) page number. What is the significance of this?
• Is the V-index actually stored in the cache?

• What is the “state” field in the tag memories?

Now consider the physical address.

• What is the role of the P-index?

• How long is the P-index, relative to the V-index?

Note that each P-tag entry points to a V-tag entry and vice versa; this makes it possible to find a physical address given a virtual address, or vice versa.

In multilevel caches, the $L_1$ cache can be virtually tagged for rapid access, while the $L_2$ cache is physically tagged to avoid synonyms across processors.

Example: Suppose $P_1$’s page 7 (addresses $7000_{16}$ to $7FFF_{16}$) is stored in page frame $12_{16}$. Suppose also that $P_2$ is using the same page, but calls it page 2.

$P_1$ is the first to access block 0 on this page. Assuming cache lines are 16 bytes long, addresses 7000 to 700F are now cached. Let’s assume the cache is direct mapped, and has $1024$ ($= 400_{16}$) lines. Which line will this block be placed in?
Now suppose that $P_2$ accesses this block. What will happen?

What number given above haven’t we used yet?

Where does this number get used or stored?

**TLB coherence**

[§6.6.3] A TLB is effectively just a cache of recently used page-table entries. Each processor has its own TLB.

How can two TLBs come to contain entries for the same page?

Name some times when TLB entries need to be changed.

- 
- 
- 
- 
- 

Which of the above lead to coherence problems?

**Solution 1: Virtually addressed caches**

When a cache is virtually addressed, when is address translation needed?

Given that accesses to page mappings are rare in virtually addressed caches, what table might we be able to dispense with?

If we don’t have a _____, we don’t have a TLB coherence problem.
Well, we still have the problem of keeping the swap-out and protection information coherent. Assuming no TLB, where will this information be found?

This problem will be handled by the cache-coherence hardware when the OS updates the page tables.

**Solution 2: TLB shootdown**

With the TLB-shootdown approach, a processor that changes a TLB entry tells other processors that may be caching the same TLB entries to invalidate their copies.

**Step 1.** A processor $p$ that wants to modify a page table disables interprocessor interrupts and locks the page table. It also clears its active flag, which indicates whether it is actively using any page table.

**Step 2.** Processor $p$ sends an interrupt to other processors that might be using the page table, describing the TLB actions to be performed.

**Step 3.** Each processor that receives the interrupt clears its active flag.

**Step 4.** Processor $p$ busy-waits till the active flags of all interrupted processors are clear, then modifies the page table. Processor $p$ then releases the page-table lock, sets its active flag, and resumes execution.

**Step 5.** Each interrupted processor busy-waits until none of the page tables it is using are locked. After executing the required TLB actions and setting its active flag, it resumes execution.

This solution has an overhead that scales linearly with the number of processors. Why?

**Solution 3: Invalidate instructions**

Some processors, notably the PowerPC, have a “TLB_invalidate_entry” instruction.
This instruction broadcasts the page address on the bus so that the snooping hardware on other processors can automatically invalidate the corresponding TLB entries without interrupting the processor.

A processor simply issues a TLB_invalidate_entry instruction immediately after changing a page-table entry.

This works well on a shared-bus system; if two processors change the same entry at the same time, only one change can be broadcast first on the bus.