Shared-Memory Multiprocessors

[§5] Symmetric multiprocessors (SMPs) are the most common multiprocessors.

They provide a shared address space, and each processor has its own cache. All processors and memories attach to the same interconnect, usually a shared bus.

SMPs dominate the server market, and are the building blocks for larger systems.

Memory hierarchies in multiprocessors fall into one of the following four categories.

(a) Shared cache

(b) Bus-based shared memory

(c) Dancehall

(d) Distributed-memory
(a) The *shared-cache* approach. The interconnect is located between the processors and the shared first-level cache.

Both the cache and main memory system may be interleaved for greater bandwidth.

Usable for connecting 2–8 processors. Why the limit?

Today, this is viable for a multiprocessor-on-a-chip.

(b) The *bus-based shared-memory* approach. The interconnect is a shared bus located between the processors’ private caches and shared main memory.

Viable for up to 30 processors. Dominant form of parallel machine today.

Modern microprocessors support “cache-coherent” shared-memory configurations (e.g., Pentium Pro can attach to shared bus without any “glue logic”).

Why the limit of 30 processors? Bandwidth limitations of shared bus, or shared memory.

(c) The *dance-hall* approach. Different from (b) because the interconnect is a scalable point-to-point network.

Many separate memory modules can be attached.

*Disadvantage:* Latency of the interconnection network; all references may have to traverse many switches.

(d) The *distributed-memory* approach. Each processor has its own local memory.

Most cache misses can be satisfied in local memory, and not have to traverse the network.
This approach is used for large-scale multiprocessors.

**The cache-coherence problem**

[§5.1] In the diagram below, memory contains a value of 5 for the variable \( u \).

\( P_1 \) reads \( u \) from main memory, thereby copying it into its cache.

\( P_3 \) reads \( u \) from main memory, thereby copying it into its cache.

\( P_3 \) writes 7 to \( u \). What happens if the cache is write-through?

What happens if the cache is write-back?

A further problem is that if the processors write different values for \( u \) to their caches, the value that reaches main memory will be determined by
This is a cache-coherence problem. Two different copies of the same information contain different values.

This problem is not unique to multiprocessors. On a uniprocessor, it can arise during I/O:

A value is read in via DMA and put in a memory location whose value is already cached. When the processor subsequently reads that location, it sees

However, I/O is not that frequent, so it can be dealt with by several ad hoc mechanisms, such as by—

- making some memory uncachable,
- flushing cache lines containing memory that will be involved in input, or
- automatically caching any data that is input.

But these approaches are not viable for multiprocessor cache coherence. Why?

**What does coherence mean?**

Our intuitive notion is that

Each read should return the last value written to that location.

However, this notion is imprecise. Why?

Let’s consider some definitions.

- A *memory operation* means a single read (load), write (store), or read-modify-write access.

  Some instructions perform ≥ one memory operation; the instruction defines the sequence in which these ops occur.
• A memory operation *issues* when it leaves the processor’s internal environment and is presented to the memory system.

• A memory operation is *performed* when it appears to have taken place, as far as the processor can tell from other memory operations it issues.

  A write “performs with respect to the processor” when a subsequent read by that processor returns the value of that write or a later write.

  A read “performs with respect to the processor” when subsequent writes by that processor cannot affect the value returned by the read.

• In a multiprocessor, an operation is *complete* when it is performed with respect to all the processors.

In a system with no caches, when would reads and writes be complete?

In this case, the memory imposes a *total order* on operations to a particular location.

• Operations to the location from a given processor occur in *program order*.

• Operations from different processors are seen with an interleaving that preserves the individual program orders.

“Last” therefore means the most recent in a hypothetical serial order that maintains these properties.

All processors will see writes in the same order (if they bother to look). This is called *write serialization*.

Note that it really doesn’t matter what order the operations *actually* occurred in, just what order they *appear* to happen.

**Bus snooping**

A bus is a good place to arrange for cache coherence.
It is just a set of wires connecting several devices. Each device can observe all transactions.

In particular, a cache can notice whether a memory transaction is relevant to it, i.e., whether it affects a memory block that it has cached.

The simplest case is in a system with

- single-level caches and a
- write-through policy.

Here, every write causes a bus transaction, so all caches can observe all writes.

If a snooping cache has a copy of the block that is written, it either

- invalidates or
- updates

its line. Every read also causes a bus transaction, but the caches can ignore these.

Look again at the diagram on p. 3 of these notes. How does an invalidation-based snooping protocol solve it?
The operation of a cache-coherence policy can be illustrated by a finite-state machine diagram. The diagram below illustrates the policy we have just described for write-through no-allocate caches.

There are two states, \texttt{Invalid} and \texttt{Valid}.

Each transition is marked with

- the input that causes the transition, and
- the output that is generated by the transition.

For example, when a controller sees a read from its processor miss in the cache, a \texttt{BusRd} transaction is generated.

After the bus read completes, the block will be in the Valid state.
If a bus write occurs (because some other processor wrote the block), the line transitions to the Invalid state.

Since this happens for all caches in the system, when one processor writes a block, all other caches that contain the line mark it as invalid.

Initially, all lines are invalid.

Blocks that are not present in the cache may be viewed either as being “invalid,” or in a special “not present” state.

(Write-back caches also need to keep track of whether each line is dirty or not.)

Processors “see” all writes in the order that they hit the bus.

What about reads?

- Read-misses are serialized along with writes on the bus. Therefore, a read-miss will see the value written by the most recent write.

- Read-hits also see the most recent write, either by this processor (the value written to the cache by this processor), or the value fetched from the bus by the most recent read-miss.

In this way, the partial ordering of operations within the individual processors is used to construct a (hypothetical) total order of operations in the system.

Any serial order that preserves these partial orders is coherent.
However, write-through is not viable for most multiprocessors:

*Example:* Suppose a processor runs at 200 MHz and issues two instructions per cycle.

- Its CPI is 1.
- 15% of instructions are stores.
- Each store writes 8 bytes.

How many processors a 1 GB/s bus support without becoming saturated?

*Answer:* How much data does a single processor transfer?

There are ______ cycles per second.
There is _____ instruction per cycle.
There are ____ writes per instruction.
Therefore there are ________ stores per second.

So the amount of data transferred is

Therefore, the maximum # of processors that can be supported is

**Interleaved memory**

When a block is written to/from a cache, it is written to $M$ consecutive words of memory. Can we design a memory that, even in the
absence of buffering, does not take $M$ memory-cycle times to complete the transfer?

Such an organization would also be faster in accessing any sequential information (e.g., instructions, arrays).

*Interleaved memory* places consecutive words of memory in different memory modules:

<table>
<thead>
<tr>
<th>Memory Module 0</th>
<th>Memory Module 1</th>
<th>Memory Module 2</th>
<th>Memory Module 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Words with addresses $= 0 \text{ (mod 4)}$</td>
<td>Words with addresses $= 1 \text{ (mod 4)}$</td>
<td>Words with addresses $= 2 \text{ (mod 4)}$</td>
<td>Words with addresses $= 3 \text{ (mod 4)}$</td>
</tr>
</tbody>
</table>

- Since a read or write to one module can be started before a read/write to another module finishes, reads/writes can be overlapped.
- Only the leading bits of the address are used to determine the address within the module. The least-significant bits (in the diagram above, the two least-significant bits) determine the memory module.
- Thus, by loading a single address into the memory-address register (MAR) and saying “read” or “write”, the processor can read/write $M$ words of memory. We say that memory is $M$-way interleaved.

This method of interleaving is known as “fine interleaving” (or “low-order interleaving”), because it interleaves on the least-significant bits.

Other forms of interleaving are also possible. Suppose we have a 32-word memory and eight memory modules.
• **Fine interleaving** distributes the addresses so that consecutive addresses are located within consecutive modules. For example,

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

• **Coarse interleaving** (high-order interleaving) distributes the addresses so that each module contains consecutive addresses.

If addresses contain \( n \) bits and there are \( M = 2^m \) modules, then module \( i (0 \leq i \leq M - 1) \) contains addresses \( i \times 2^{n-m} \) to \((i + 1) \times 2^{n-m} - 1\), inclusive. E.g.,

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

• A combination of low- and high-order interleaving can also be used. For example,

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

This is called a “grouped four-way interleaved” organization with two groups.

In addition to the way addresses are distributed, interleaved memories can also be classified according to the way concurrency is achieved:

**Interleaved-memory designs:** Interleaved memory divides an address into two portions: one selects the module, and the other selects an address within the module.

Each module has a separate MAR and a separate MDR.
• When an address is presented, a decoder determines which MAR should be loaded with this address. It uses the low-order \( m + \log_2 M \) bits to decide this.

• The high-order \( n-m \) bits are actually loaded into the MAR. They select the proper location within the module.

**S-access**

One address can be issued each (latch-delay) cycle, and one word is returned every (latch-delay) cycle (after a startup time equivalent to the memory cycle time).
Suppose that $M = 8$. Assume that the memory-access time is = 8 cycles (or 8 “latch delay” times).

Answer these questions. Assume that the processor fetches only the words that are needed.

- How long does it take to access word 0?
- How long does it take to access word 5?
- How long does it take to access word 0 and then word 5?
- How long does it take to access word 10?
- How long does it take to access word 5 and then word 10?
- How long does it take to access word 1 and then word 9?

There is a hidden assumption here. Under what circumstances would it not be possible to access word 5 and then word 10 on the next cycle?

**C-access**

Multiple MARs are important for performance when—
• the latch-delay time is much less than the memory-access time, and
• only one word at a time is being read from memory by the processor.

This organization is sometimes called the \( C \)-access configuration (concurrent accesses).

However, when a low-order interleaved memory is being used to load a cache, accesses will always be sequential.

In this case, if the line length is evenly divisible by the degree of interleaving, we can get by with a single MAR.

In this case, the accesses are \textit{simultaneous}, so this is called the \( S \)-access organization.

After waiting for all of the memory accesses to complete (simultaneously), the processor can fetch a different word each (latch-delay) cycle:

When a cache line is loaded, the processor always fetches consecutive words, beginning with the word from module 0 (i.e., if the word from module 6 is needed, the words from modules 0, \ldots, 5 must be fetched first).

• How long does it take to access word 0?
• How long does it take to access word 5?
• How long does it take to access word 0 and then word 5?
• How long does it take to access word 10?
• How long does it take to access word 5 and then word 10?