Hardware/Software Tradeoffs

Hardware support for fast non-local references is expensive, so expensive that considerable research has been directed to having software assume some of the burden.

Today we will study two schemes that combine hardware and software to produce scalable architectures that provide a good balance between price and performance.

Cache-only memory architectures (COMAs)

Suppose blocks are shared frequently, and therefore migrate between caches frequently.

There may not be enough room to hold them in the caches at the destination processor, leading to a large number of capacity misses. What can we do about this?

One approach is to dedicate a portion of memory at each node to act as a cache.

Main memory of a node

<table>
<thead>
<tr>
<th>Home memory</th>
<th>Tertiary cache</th>
</tr>
</thead>
</table>

The tertiary cache is managed by software to act just like a cache. Some words of memory are treated as tags. It may use direct or set-associative mapping.

For example, on the Sequent NUMA-Q, the tertiary cache is \( \geq 32\text{MB} \), compared to no more than 4 MB in secondary caches per node.

What are some problems with tertiary caches?

•
With tertiary caches, blocks used remotely are present both at their home memory, and in a tertiary cache(s) elsewhere in the system. Of these, the copy in the tertiary cache is closer to where the data will be used.

So, how about getting rid of the home memory? If we do, we end up with a cache-only memory architecture (COMA).

In a COMA, every block of memory in the entire system has a (hardware) tag associated with it.

Data migrates to, or is copied to, the nodes where it is being used. Thus, main memory is called attraction memory.

When a process at a remote node B references a block in Node A’s attraction memory, a copy is made in B's memory.

The original copy at A may be purged or replaced later.

The block has effectively migrated to B.

So, less memory is wasted in a COMA.

It also frees the programmer from worrying about data placement in memory.

But (s)he still needs to worry about the cost of _________ and also about _________ sharing.
A COMA needs additional hardware to manage the tags and do lookup.

Also, the replacement policy is more complicated. Why?

- When there is a miss, we must determine

- We must be sure not to

**Performance**

How can COMAs help—

- in terms of execution time?

- in terms of space?

How can COMAs hurt performance?

- It takes longer to find the block that is needed to satisfy a miss.
  - Main-memory accesses are a little slower. Why?

What kind of applications are COMAs likely to be useful for?

- Those that have high
- Those where the access pattern is

What kind of applications are they likely to be bad for?

- Those that have high
- Those where the access pattern is

**Flat or hierarchical directory?**
A flat directory scheme has a fixed location for the directory (only).

The location of the directory is determined from a unique ID, which may be the physical address.

What happens on a miss?

- The home is consulted to find the
- The directory keeps track of where
- One of the copies is fetched or replicated,

In a hierarchical directory scheme, a miss proceeds up the hierarchy until a node is found that has the block in the appropriate state.

The request then proceeds down the hierarchy to the appropriate node.

How do these schemes solve the last-copy replacement problem?

- If the block being replaced is in shared state (and another copy really exists),
- If the block being replaced is in the exclusive state (or if we’re not sure another copy exists),

In a hierarchical scheme, for a shared block, we proceed up the hierarchy, updating states along the way, until we find a node that has a copy of the block somewhere in its subtree. Then we can

In a hierarchical scheme, for an exclusive block, we

In a flat scheme, we designate one copy of a block the master copy.

- When memory is allocated for data, the block where it is located is made the master copy.
- When a copy of the block is made, the copy is not a master.
- When a shared copy is replaced,
  - When a master copy is replaced, a replacement message is sent to the home.
    - The home sends the block to another node.
    - If the other node doesn’t have any non-master blocks, it sends the request to a third node, etc.

Clearly, for good performance, there has to exist an ample supply of free memory.

Shared virtual memory

A COMA can be effective in supporting replication and migration of data under certain circumstances. But it requires a lot of hardware support.

We can also support replication and migration in software, with the aid of an ordinary MMU.

However, the unit of sharing and migration must be

Also, because software controls migration, it will be slower than in a COMA.
Here is how sharing and migration occur.

1. Processor $P_0$ encounters a page fault on a read reference and fetches a read-only copy of the page from secondary memory.

2. Processor $P_1$ encounters a page fault on a read reference and fetches a read-only copy of the page from $P_0$.

3. Now $P_0$ writes the page. What happens?

4. The OS gains control and invokes the SVM library to invalidate $P_1$’s copy of the page.

5. The SVM library gives $P_0$ write-access to the page.

6. Now if $P_1$ tries to read the page again, it needs to fetch a new copy from $P_0$ (through operations of the SVM library).

To find copies of pages, the SVM library uses a directory mechanism as described in Lecture 19.

What are some reasons why an SVM is slower than a COMA or other hardware-supported approach?

- High overhead of invoking SVM operations. Why is it high?
- High overhead of communication. Why is it high?
- The processor is interrupted by incoming

$\Rightarrow$ Cost of satisfying a remote page fault is $< 1 \mu s.$ on a hardware-coherent system, but 100s of $\mu s.$ to $> 1$ ms. on an SVM system.

How does false sharing come into the picture?
Think of what SC requires. Each write may cause invalidations throughout the system. This is especially serious under false sharing.

*Relaxed memory consistency*

If we use a model like release consistency, we don’t have to propagate invalidations immediately. Why?

So we delay propagating the invalidations until a synchronization point. How is this different from a hardware-coherent system?

Fortunately, delaying invalidations means fewer invalidations overall.

Now, at which synchronization point should we propagate invalidations?

Under release consistency, a process does not really need to see a write (or an invalidation) until it does an acquire!

- If invalidations are propagated at a release, we have *eager release consistency*.
- If they are propagated at an acquire, we have *lazy release consistency*.

However, LRC requires greater attention to inserting synchronization operations correctly.

Here is a piece of code that works under ERC (and ordinary RC), but gets in an infinite loop under LRC:

```
P_1
lock L1;
ptr = non_null_ptr_val;
unlock L1;
while (ptr == null) {};
lock L1;
a = ptr;
unlock L1;
P_2
```
What is the solution?