Lecture 12

Shared memory architecture

Announcements

• Assignment #4 due on Tuesday
• Midterm return
Shared memory architecture

- Every processor has direct access to all of memory
- The address space is global to all processors
- Hardware automatically performs the global to local mapping using virtual to physical address translation

Cache Coherence

- A central design issue in shared memory architectures
- Processors may cache the same location
- If one processor writes to the location, all others must eventually see the write

\[
\text{X:=1} \\
\text{Memory}
\]
Cache Coherence

- P1 loads X from main memory into its cache

![Diagram showing cache coherence]

Cache Coherence

- P1 loads X from main memory into its cache
- P2 loads X from main memory into its cache

![Diagram showing cache coherence with P2]
Cache Coherence

- P1 stores 2 into X
- We don’t have consistent values for X across the memory hierarchy

![Diagram of cache coherence showing inconsistent values for X]

Cache Coherence Protocols

- Ensure that all processors *eventually* see the same value for x
- One approach is to update all copies on a write: implies a write-through cache

![Diagram of cache coherence protocols showing consistent value for X]
Cache Coherence Protocols

- Ensure that all processors *eventually* see the same value for x
- One approach is to update all copies on a write: implies a write-through cache
- We may also use **invalidate-on-write**

Memory consistency and correctness

- Cache coherence tells us that memory will *eventually* be consistent
- The memory consistency policy tells us *when* this will happen
- Even if memory is consistent, changes don’t propagate instantaneously
- These give rise to correctness issues involving program behavior
Race conditions

- A *Race* condition arises within an application when the timing of accesses to shared memory can affect the outcome
- We say we have a *non-deterministic* computation
- Sometimes we can use non-determinism to advantage, but usually we want to avoid it
- Memory consistency and cache coherence are necessary but not sufficient conditions for ensuring program correctness
- We need to take steps to avoid race conditions through appropriate program synchronization

Two kinds of shared memory architectures

- Distinguished by whether or not memory access time is uniform
- **UMA**: Uniform Memory Access time
  - In the absence of contention, all processors see the same access time to memory (approximates a PRAM)
  - Also called *Symmetric Multiprocessors (SMPs)*
  - Usually bus based: not a scalable solution
- **NUMA**: Non-Uniform Memory Access time
  - Memory access time depends on distance to memory
  - Also called *Distributed Shared Memory (DSM)*
  - Elaborate interconnect structure
**SMP architectures**

- Employ a *snooping protocol* to ensure coherence

- Key extensions to uniprocessor: snooping, invalidating/updating caches, I/O devices, Mem

**Snooping protocol**

- Bus-based scheme
- Processors passively listen for bus activity, updating or invalidating cache entries as necessary
- The scheme is not scalable but is cost-effective for small designs
False sharing

- Consider two processors that write to different locations mapping to different parts of the same cache line

![Diagram showing two processors sharing a cache line](image)

False sharing

- P0 writes a location

![Diagram showing P0 writing a location](image)
False sharing

- P0 writes a location
- Assuming we have a write-through cache, memory is updated

False sharing

- P1 reads the location written by P0
False sharing

- P1 reads the location written by P0
- P1 then writes a different location in the same block of memory

False sharing

- P1’s cache is no longer consistent with P0’s cache and with main memory
False sharing

• P1’s write will update main memory
False sharing

Successive writes by P0 and P1 cause the processors to uselessly invalidate one another’s cache.
Ping pong effect

• Successive writes by P0 and P1 cause the processors to uselessly invalidate one another’s cache
Ping pong effect

- Successive writes by P0 and P1 cause the processors to uselessly invalidate one another’s cache
Eliminating false sharing

- Cleanly separate locations updated by different processors
  - Manually assign scalars to a preallocated region of memory using pointers
  - With a block partitioned array, we want partition boundaries to coincide with a cache line boundary
- Compilers can perform some of these optimizations

False sharing in higher dimension arrays

- Large memory access strides, conflict misses
- Compare with distributed memory solution
NUMA Architectures

- Unlike UMAs, NUMAs do no rely on a broadcasting capability
- They employ point-to-point messages to manage coherence
- Rely on a directory to keep track of sharers

Directory based coherence

- Every block of memory has an associated directory entry
- Stanford Dash; SGI Origin 2000
Some basic terminology

• Every block of memory has an associated **home**: the specific processor that physically holds the associated portion of the global address space
• Every block also has an **owner**: the processor whose memory contains the actual value of the data
• Normally these are the same
• But they can be different if a processor other than the home’s processor writes a block

Inside a directory

• Each processor has a 1-bit “sharer” entry in the directory
• There is also a dirty bit and a PID identifying the owner in the case of a dirt block
**Operation of a directory**

- Assume a 4 processor system (only P0 & P1 shown)
- A is a location with home P1
- Initial directory entry for block containing A is empty

![Diagram showing initial state](image1)

- P0 loads A
- Set directory entry for A (on P1) to indicate that P0 is a sharer

![Diagram showing updated state](image2)
Operation of a directory

- P2, P3 load A (not shown)
- Set directory entry for A (on P1) to indicate that P0 is a sharer

Acquiring ownership of a block

- P0 writes A
- P0 becomes the owner of A
Acquiring ownership of a block

- P0 becomes the owner of A
- P1’s directory entry for A is set to Dirty
- Outstanding sharers are invalidated
- Access to line is blocked until all invalidations are acknowledged

![Diagram of acquiring ownership of a block]

Forwarding

**Store A, #1**

\[\downarrow\]

*(home & (P1) owner)*
Forwarding

\begin{align*}
\text{Store A, #1} & \\
\downarrow & \\
(P1) & \leftarrow \leftarrow (P0) \\
\text{Store A, #2}
\end{align*}

\textit{mark A as dirty}

\begin{align*}
\text{Store A, #1} & \\
\downarrow & \\
(P1) & \leftarrow \leftarrow (P0) \text{owner} \\
\text{Store A, #2}
\end{align*}
Forwarding

Store A, #1
↓
mark A as dirty

(P1)←-----------(P0)owner

Store A, #2

(P3) Load A
Forwarding

Store A, #1
↓
mark A as dirty
(P1) ← owner (P0)

(P3) Load A

Store A, #2

Performance issues with DSMs

• When we allocate a block of memory, which processor(s) is (are) the owner(s)?
• Page allocation policies
  – First touch
  – Round robin
• We can control memory locality with the same kind of data layouts that we use in shared nothing architecture, BLOCK, CYCLIC
• See extensive on-line notes published by SGI:
  Performance Tuning for the Origin 2000
  http://www.ncsa.uiuc.edu/SCD/Hardware/Origin2000/Doc