Shared Memory Multiprocessors and Cache Coherence

Kai Shen

Multiprocessors
- Limitation of instruction-level parallelism
  - Dependences
  - Complexity to support high-degree instruction-level parallelism
- Why so challenging?
  - Hardware has to extract parallelism from software that doesn't explicitly expose parallelism
- Multiprocessors
  - A machine that contains multiple CPUs
  - Software explicitly exposes parallelism and runs multiple tasks simultaneously on the CPUs

Shared Memory Multiprocessors
- Multiple processors sharing memory
  - Does not require data partitioning; easy data access in programming
  
  ![Diagram of shared memory multiprocessors]

  - Traditional shared memory multiprocessors
    - Processors and shared memory connected by a bus
    - Each processor may contain private cache

Multiprocessor Architecture: Hardware Multithreading
- The center of a processor contains
  - The instruction fetching, decoding, pipelining units etc. (with registers)
- Other peripheral things on processor
  - L1/L2 cache, floating-point units, etc.
- Hardware multithreading
  - Multiple processors share the same set of peripheral things so they can be manufactured on the same silicon die

![Diagram of hardware multithreading]
### Hardware Multithreading

- **Benefits:**
  - Less manufacturing cost → more processors on one silicon die
  - Less power consumption per processor
  - Faster processor-to-processor coordination and data sharing

- **Problems:**
  - Resource contention diminish benefits, leads to unpredictable performance

### Multicore

- Last-level cache (LLC) is most significant part of the chip
- Multicore: just sharing the LLC allows multiple processors on the same silicon die

- Benefits:
  - Between traditional multiprocessor and multithreading
  - Inherit much of the benefits for hardware multithreading
  - Only resource contention is on the shared LLC space

### NUMA

- Memory bandwidth is a big problem for large-scale multiprocessor
- Non-Uniform Memory Access
  - Each processor can still access all memory, but accesses are faster to “local memory”

- Data placement has significant performance impact (controllable by software)
- Memory is not connected through a single bus – better scalability, but implication to the cache coherence problem we will discuss

### Our Experimental Machines

- **node2x12x1a**
  - two CPU chips (sockets), each containing 12 cores

- **node4x2a**
  - four CPU chips, each containing two cores, each further containing two hardware threads

- **cycle2 / cycle3**
  - you may find out by reading /proc/cpuinfo

- **node17 – node28**
  - two CPU chips, each containing two hardware threads
Cache Coherence Problem

- In shared memory multiprocessors (except hardware multithreading), each processor has a local cache

- For each data item in memory, additional copies may exist in processor local caches
  - after one processor updates the data, another processor’s local copy may be incoherent
  - What is wrong about it?

Cache Coherence

- Coherence means the system semantics is the same as that of a system without processor-local caches

- Multiprocessor cache coherent if there exists an equivalent sequential ordering of all operations on a data location:
  - returned value in the read operation is that written by last write in the sequential order
  - the sequential order matches the order of operations from each processor

Cache Coherence Through Bus Snooping

- All caches and memory connected by a shared bus

- Bus snooping
  - Each processor can monitor the bus for activities

- Not always the case (particularly for NUMA)

Bus Snooping For Write-Through Caches

- Cache can be write-through or write-back
  - Assume write-through cache – every write goes to the bus

- Bus snooping
  - Each processor monitors the bus for writes
  - If there is a local cached copy of the write target, it is invalidated or updated.
    - Tradeoff between invalidation vs. update?
  - How does it ensure cache coherence?
    - Construct an equivalent sequential ordering of memory accesses
Coherence on Write-Back Caches

- Write-back caches are much more favored in practice
- Writes do not necessarily go to bus, so we may not directly snoop them

MSI Write-Back Invalidation

- Three states for a cache entry
  - Modified (M) – I modified it, I am the only one who has a copy
  - Shared (S) – I have a clean copy, possibly shared by others
  - Invalid (I)
- Write to a modified cache entry?
- Write to a shared/invalid cache entry?
  - Local write preceded by a read-exclusive (must go to bus, invalidate other caches’ copies)
- How to handle a read?
- State transition diagram
  - [Link](http://wiki.expertiza.ncsu.edu/images/e/e4/MSI.jpg)

MESI Write-Back Invalidation

- Three states for a cache entry
  - Modified (M) – I modified it, I am the only one who has a copy
  - Shared (S) – I have a clean copy, possibly shared by others
  - Invalid (I)
- When does it matter?
  - I read an entry into cache and then write to it
  - With S state, write must be preceded by a read-exclusive
- Add a new state
  - Exclusive clean (E)
- When to assign E to an entry?
  - First read, no other cache has a copy
Directory-based Cache Coherence

- No all multiprocessors use shared bus for memory access
  - It does not scale!
- Large multiprocessors with NUMA
  - Many local memory accesses
  - Without the ability of bus snoop, an explicit directory about memory block's caching state can be used
  - May need to check the directory for cache coherence operations