# Shared Memory Multiprocessors

Ashok Singhal

Sun Microsystems

# Shared Memory Multiprocessors

**Definition: What are shared memory multiprocessors?**

Two or more CPUs (and IO devices) that can access (read and write) a shared memory area.

- Networks and Clusters are NOT shared memory multiprocessors.

**Motivation: Why are we building these machines?**

- Why is shared memory useful?
- Why is shared memory multiprocessor design challenging?

**Coherence and Synchronization: The fundamental problems.**

- What is synchronization?
- What is cache coherence?

**Coherence protocols :How do we solve the problem?**

- Data write problem: update vs. invalidate protocols.
- Writeback cache problem: ownership protocols.
- Directory vs Snooping protocols.
- A Simple Snooping Protocol with a Circuit-Switched Bus.
- Performance Improvement: Packet-Switched Bus.
- Performance Improvement: MOESI Protocol.
- Performance Improvement: Dual Tags.

**Implementation: Details of a high performance snooping implementation.**

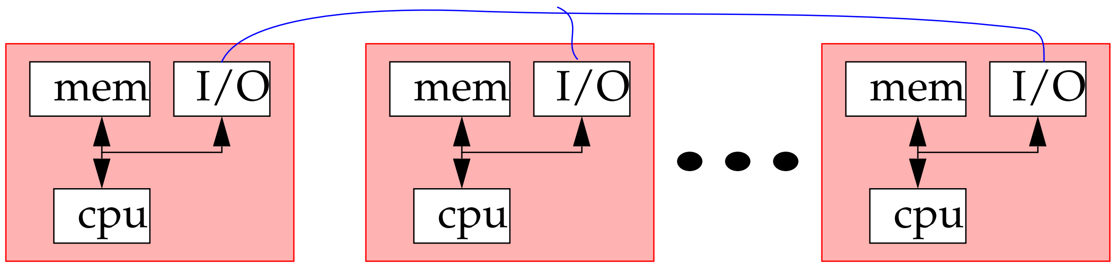
- Example: SUN Ultra Enterprise (UPA and Gigaplane).
- Memory, Input/Output, Interrupts and Errors.

**Evaluation: Measuring performance.**

- Comparing a few large commercial bus-based shared memory multiprocessors.

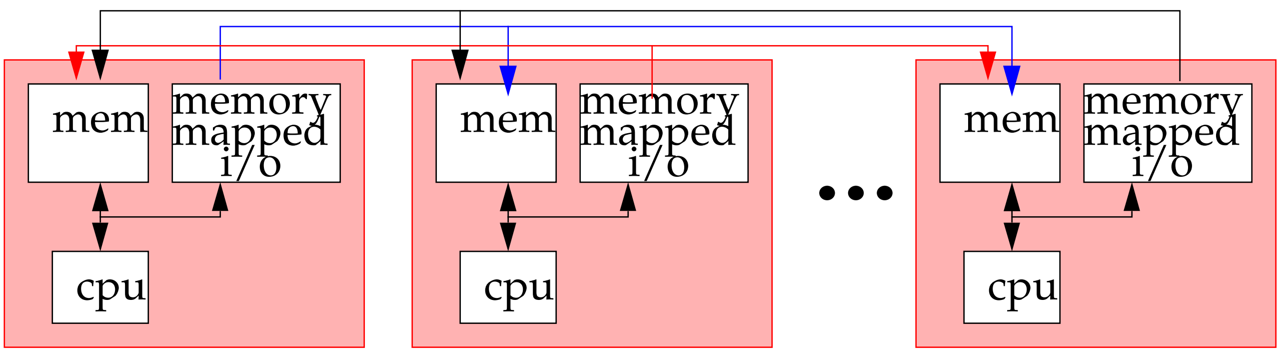
# What are Shared Memory Multiprocessors?

**Computer networks are NOT shared memory multiprocessors**



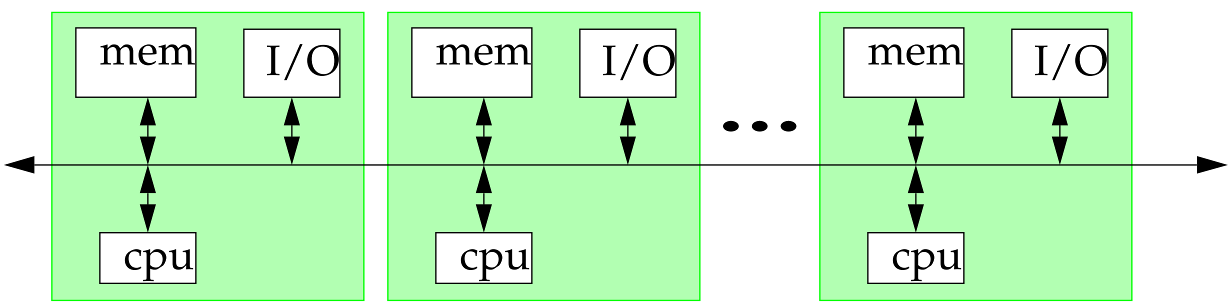

- A memory location on one computer cannot be directly read or written on the other comuters (examples: Ethernet)

**Computer clusters are NOT shared memory multiprocessors**




- Writes (and reads) to I/O locations on one computer appear as writes (and reads) to memory locations on other computers (examples: DEC TruCluster(only writes))

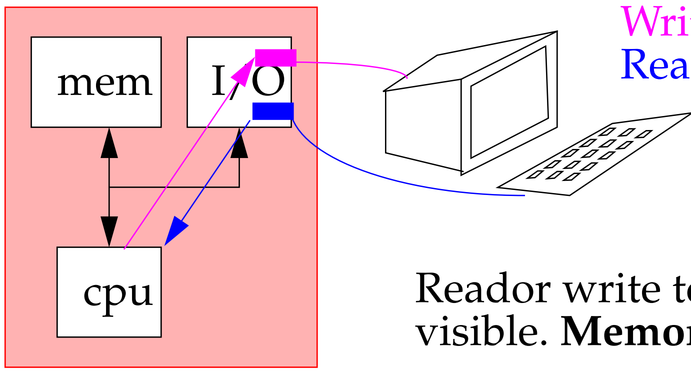
**Shared Memory Multriprocessors really, truly SHARE memory**




- memory read/written by one CPU can be directly read/written by another (examples: SUN Ultra Enterprise 6000, DEC AlphaServer 8400, SGI Origin 2000).

# DIfference between IO and Memory

- IO has Side Effects, Memory does not.

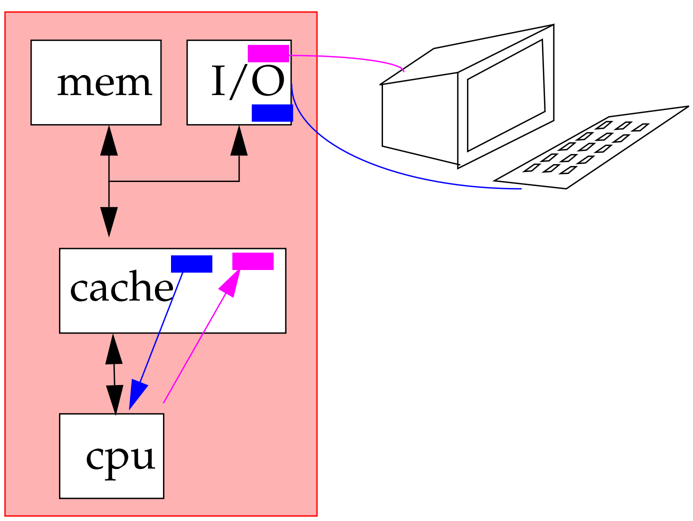



Write to IO: changes screen
Read from IO: read different key

Readorwrite to memory is not externally visible. **Memory is pure storage**

- Memory can be safely cached, IO (in general) can not

Write to cached IO: NO changes screen
Read from cached IO: read same key everytime




- Memory is usually cached
- IO is usually non-cached

# Why is Shared Memory Useful?

**Easier to program.**

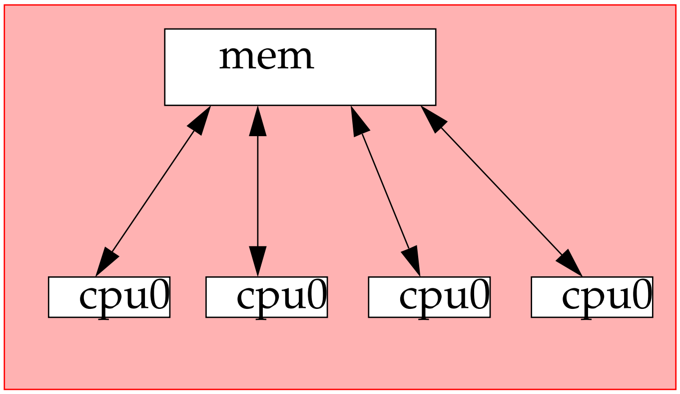
- Single-threaded program is identical to uniprocesor
  - Faster time-sharing (multiple processes at same time).
  - More efficient use of resources (share memory, IO devices).
- Parallel (multi-threaded) programs are simpler to write
  - Communication can be implicit (just read or write shared memory).
  - Explicit communication (messages) possible but not required.
  - Communication model is basically same as for processes/threads on uniprocesors.

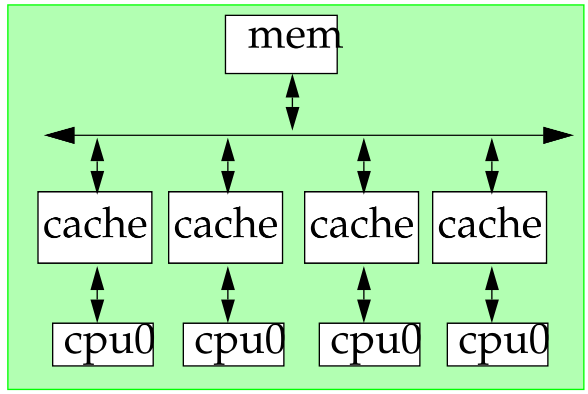
**Higher performance communication.**

- Pass “message” by writing to memory area and passiing pointer to thread on another CPU
  - Single-threaded program can be moved by OS to another CPU without explicitly copying memory.
- Communication uses paths that are highly optimized
  - Latency ~ cache miss latency
  - Bandwidth ~memory bandwidth

# Why is Shared Memory Multiprocessor Design Challenging?

- Mult-port memory directly accessed by CPU is
  - Too slow
  - Requires **very** high bandwidth memory (too expensive)

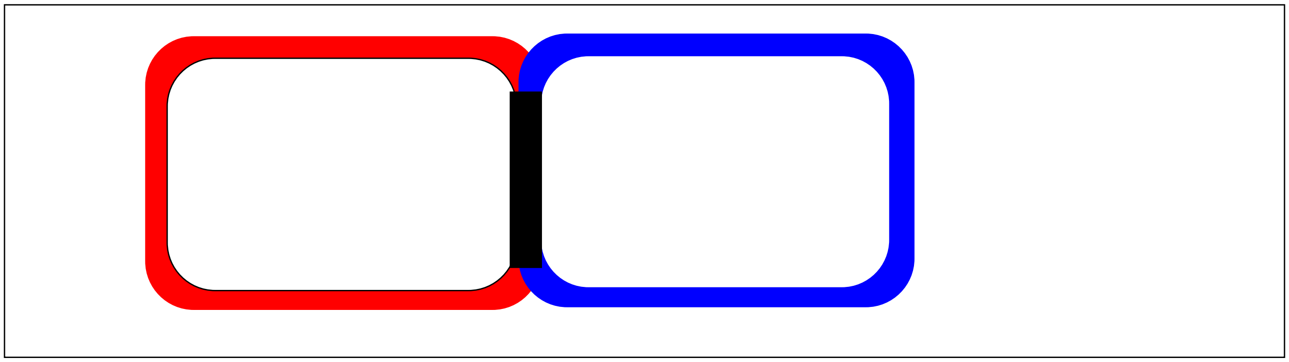



- Caches complicate matters greatly (Cache coherence problem)
- Hard to get low latency to shared memory (compared to uniprocessor)
  - Sun SparcCenter 2000, SGI Challenge: ~ 1200 ns
  - Sun SparcStation 5: ~ 145 ns
- Need large, high bandwidth memory
  - Multiple interleaved banks
- Need high IO bandwidth and connectivity
  - Multiple IO controllers
- Hard to get the Operating System software right
  - Synchronization
  - Many more cases to test

# Why is Synchronization Necessary?

**Two trains sharing a section of track**



**Control programs for train**

```
while (in service)                    while (in service)
{                                     {
    while (track == busy)                 while (track == busy)
    {                                     {
        ; /* wait */                          ; /* wait */
    }                                     }
    track = busy;                         track = busy;
    use track;                            use track;
    track = free;                         track = free;
}                                     }
```

**But trains crash!**

**track = free**

**Both trains see track != busy**
**Both trains set track = busy**
**Both trains use track ===> CRASH !!**

**REASON: Between read and write of track status, other train reads track status**

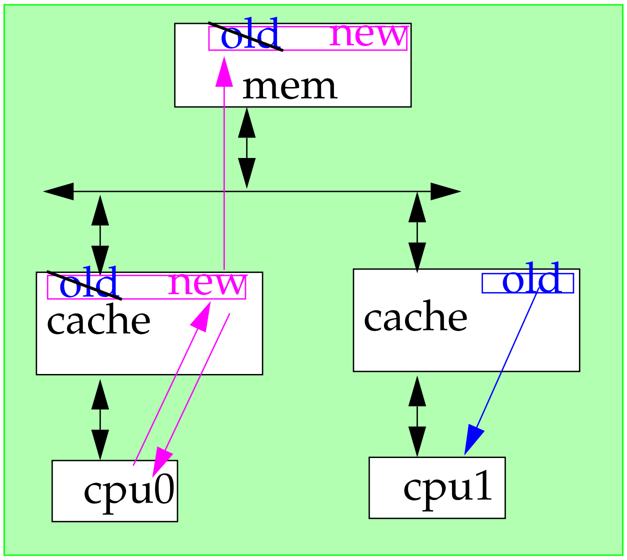
**Synchronization => Make the read and write of track status ATOMIC**

- Trickier on a multiprocessor

# What is Cache Coherence?

**Memory: Read value = most recent write value**

**"Stale" data problem due to cpu writes to shared cached copy**



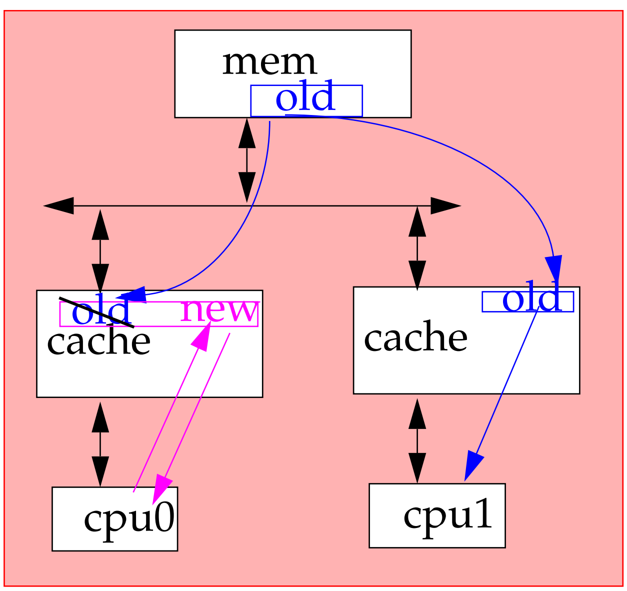

cpu 0 and 1 both cache the same data (value = old).

cpu 0 writes to its copy of cached data (value = new)

cpu 0 reads value = new

cpu 1 reads value = old
-> "stale" data

**"Stale" data problem due to writeback caches**




cpu 0 caches data (value = old)

cpu 0 writes to its copy of cached data (value = new)

cpu 0 reads value = new

cpu 1 gets copy from memory

cpu 1 reads value = old
-> "stale" data

**Cache Coherence**

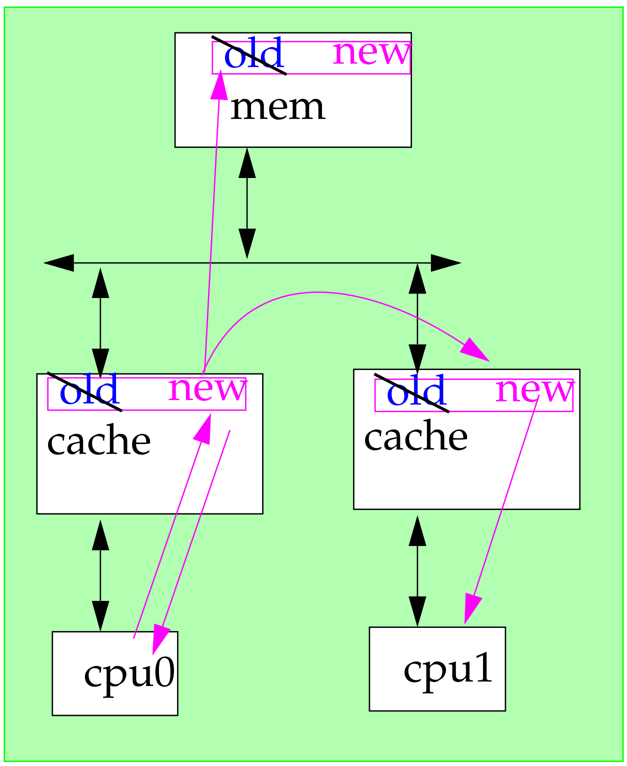
- Method that solves the "stale" data problems

# Update Coherence Protocols

## Solving the stale data problem due to cpu writes

**Update solution: when cpu writes to cache:**

- update other caches (if they have a copy. How does cpu know which caches have a copy?)
- update memory (if write-through)




cpu 0 and 1 both cache the same data (value = old).

cpu 0 writes to its copy of cached data (value = new) and updates cpu 1

cpu 0 reads value = new

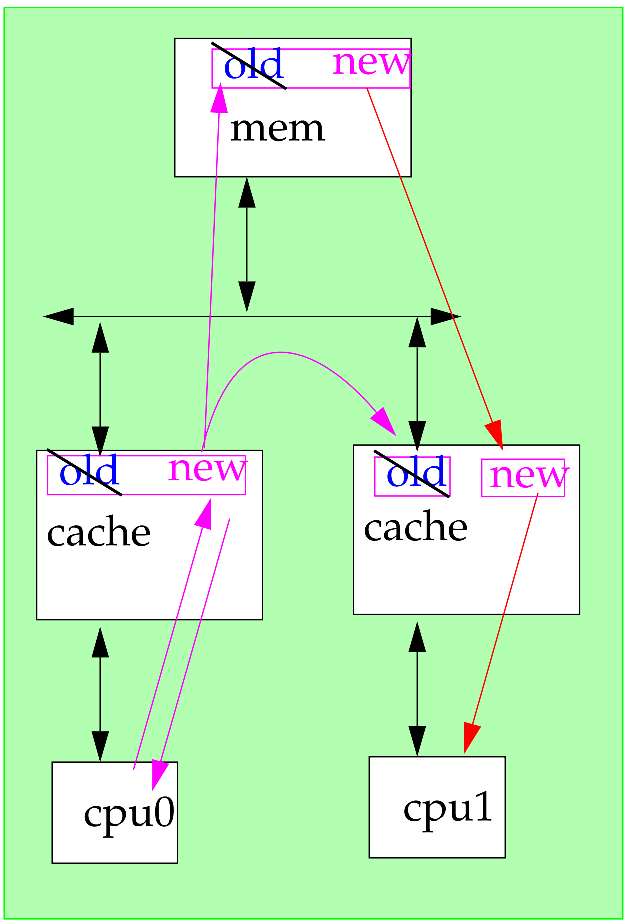
cpu 1 reads value = new
-> NO “stale” data

# Invalidate Coherence Protocols

## Solving the stale data problem due to cpu writes

**Invalidate solution: when cpu writes to cache:**

- invalidate other caches (if they have a copy. How does cpu know which caches have a copy?)
- update memory (if write-through)




cpu 0 and 1 both cache the same data (value = old).

cpu 0 writes to its copy of cached data (value = new) and invalidates dates cpu 1

cpu 0 reads value = new

cpu 1 gets cache miss reads new value

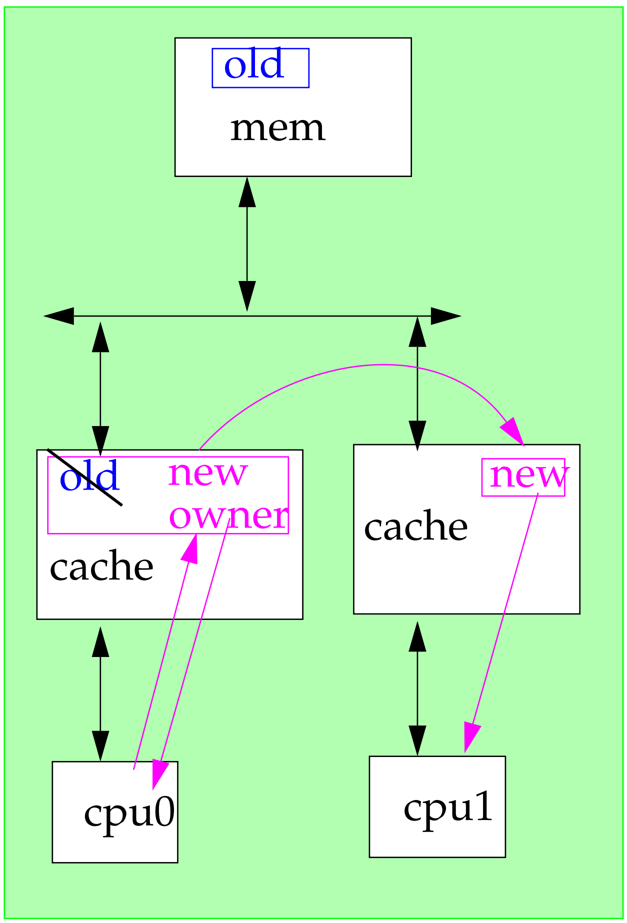
-> NO “stale” data

# Ownership Protocols

## Solving the stale data problem due to writeback caches

**Ownership solution: when cpu writes to cache:**

- do not update memory (writeback cache)
- cache becomes OWNER of line => responds to other cpu's cache miss instead of memory (how does OWNER know to respond to the other CPU's cache miss? How does memory know not to respond?).




cpu 0 caches the data (value = old).

cpu 0 writes to its copy of cached data (value = new)

cpu 1 gets cache miss
cpu 0 cache responds new value
(memory does not respond)

-> NO "stale" data

**OPTIONAL: update memory when cpu 0 transfers data to cpu 1.**

# Directory vs Snooping Protocols

**Two questions arose in the update/invalidate/ownership protocol discussion:**

**How does cpu know which caches have a copy to invalidate/update?**

**How do we know when OWNER and not memory should respond?**

**Two solutions:**

**1. Directory Protocols**

- Each line of memory has extra storage (DIRECTORY) to keep track of
    - which cpus have shared copies
    - which cpu is owner
- Check directory in memory for OWNER and shared copies to update/invalidate

**2. Snooping Protocols**

- Each line of cache has extra storage (TAG) to keep track of
    - whether line is shared/exclusive
    - whether the cpu is owner or not
- Broadcast request to all other caches which look up their tags ("snoop") to check if they are OWNER or have a shared copy to update/invalidate

**Main Advantage for Directory:**

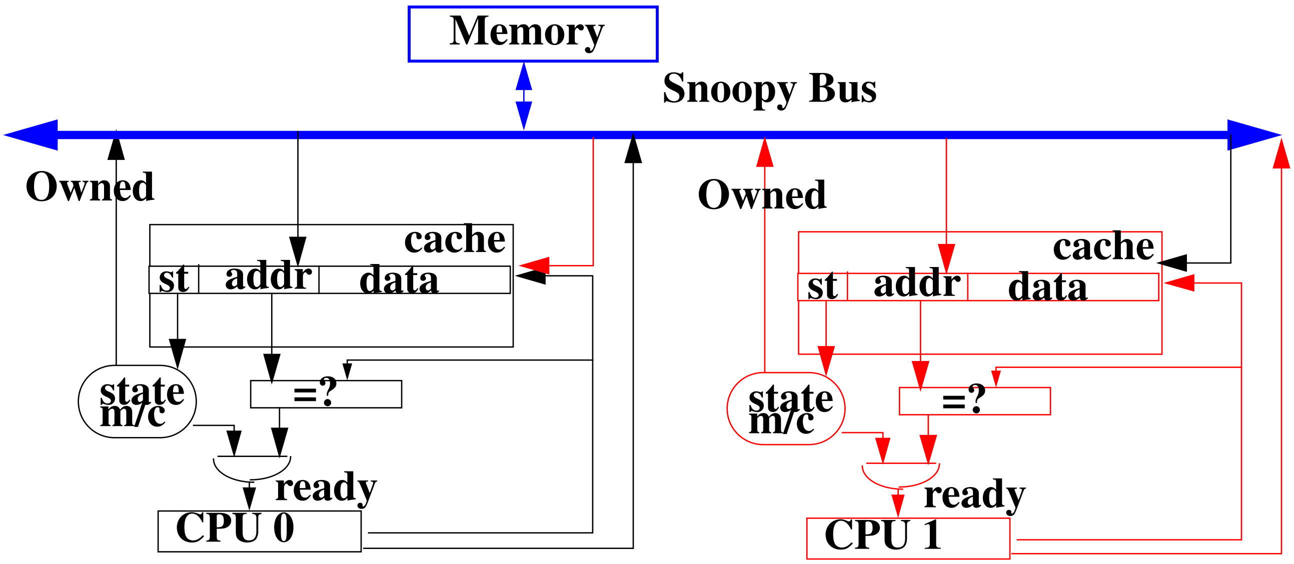
- No broadcast since directory distributed among memory controllers (so better for really large systems).

**Main Advantage for Snooping**

- Tag size proportional to cache size, not (memory size * number of cpus).
- Simpler, use broadcast on bus.

# A Simple Invalidate Snooping Protocol

## Simplified Snooping Structure

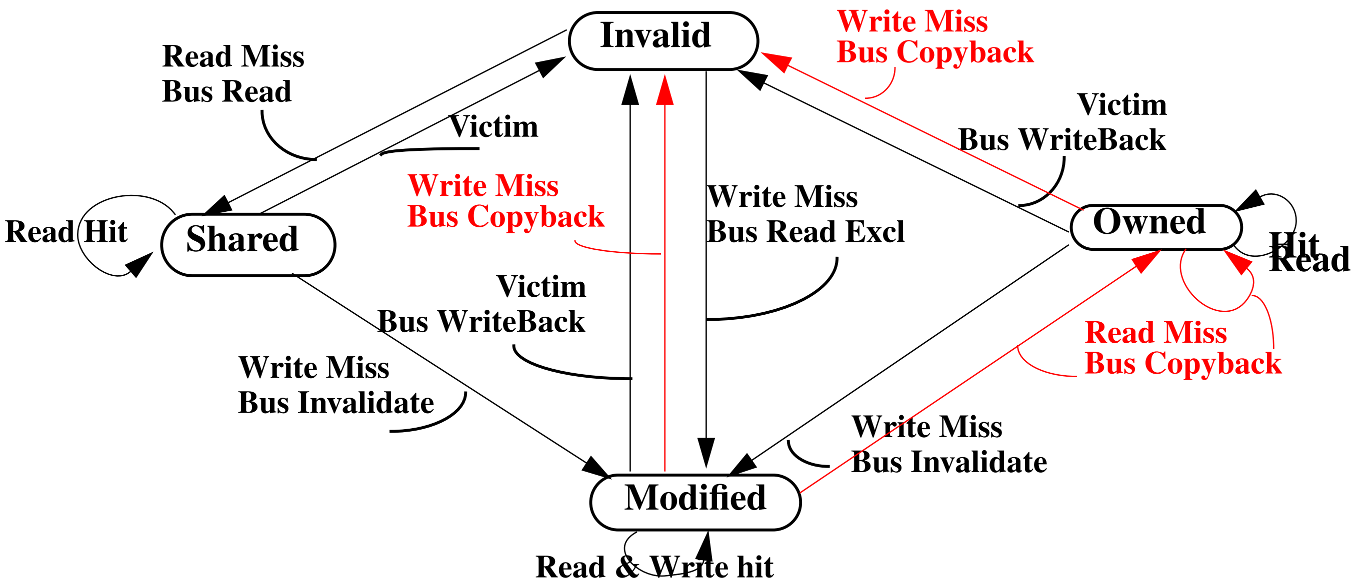



### "MOSI" protocol States

| State | Name | Meaning |
|---|---|---|
| I | Invalid | Read -> Miss |
| | | Write -> Miss |
| | | Victim -> None |
| | | Not owner (no copyback required) |
| S | Shared | Read -> Hit |
| | | Write -> Miss (need to invalidate copies) |
| | | Victim -> Clean (no writeback required) |
| | | Not owner (no copyback required) |
| M | Modified | Read -> Hit |
| | | Write -> Hit (exclusive copy) |
| | | Victim -> Dirty (writeback required) |
| | | Owner (copyback required) |
| O | Owned | Read -> Hit |
| | | Write -> Miss (need to invalidate copies) |
| | | Victim -> Dirty(writeback required) |
| | | Owner (copyback required) |

# MOSI Snooping Protocol State Machine

**Snooping State Machine for CPU 0 (Red = CPU 1 operations)**




MOSI Protocol Example

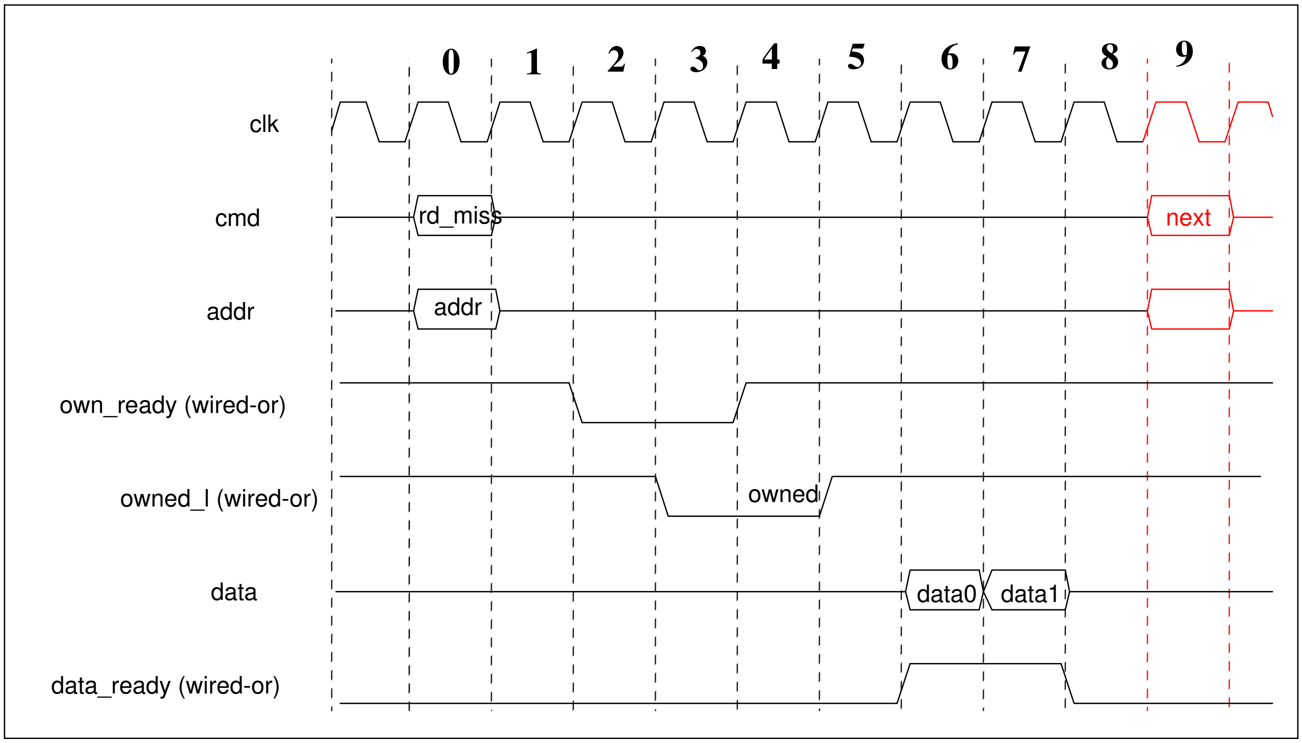
| Seq. | Action | CPU 0 state | CPU 1 state | Bus transaction |
|---|---|---|---|---|
| 0 | (initial) | I | I | |
| 1 | CPU 0 read | S | I | Read Miss |
| 2 | CPU 1 read | S | S | Read Miss |
| 3 | CPU 0 write | M | I | Write Miss Invalidate |
| 4 | CPU 0 write | M | I | (none) |
| 5 | CPU 1 read | O | S | Read Miss Copyback |
| 6 | CPU 0 victim | (I) | S | Writeback |
| 7 | CPU 1 write | (I) | M | Write Miss Invalidate |
| 8 | CPU 1 read | (I) | M | (none) |
| 9 | CPU 1 victim | (I) | (I) | Writeback |

**Invariance: If a line is in state M in a cache, it must be invalid in all other caches.**

**Invariance: A line can be in state O in at most one cache.**

# A Simple Circuit-Switched Snoopy Bus

**An Example of a Circuit Switched Bus Timing Diagram**




**Owned signal**

- owned_l qualified by own_ready (own_ready goes high when all devices have completed snoop).
- wire-or owned_l (active-low) and own_ready.

**Circuit-switched: Bus transaction completes from start to finish before next one starts.**

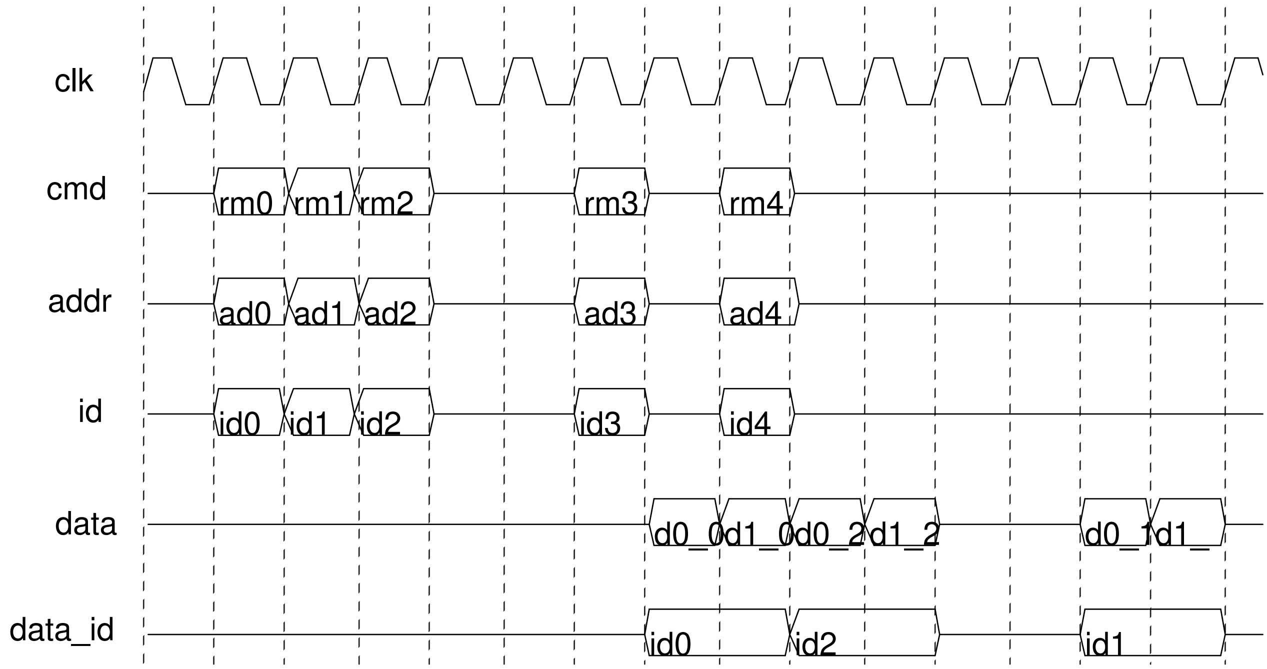
- Simple (no interleaved transactions)

**Main Disadvantage: Low bandwidth (one read miss every 9 clocks)**

**Performance Improvement: Packet-Switched (Split Transaction) Bus.**

# Packet-Switched (Split Transaction) Bus

**An Example of a Split Transaction Bus Timing Diagram**




**Split Transactions:**

- Don't wait for one transaction to complete before making additional requests => Much higher bandwidth.
- Data (responses) can be in different order than requests => better performance during contention.
  - Some systems only: Examples: HP K-class, Sun Ultra Enterprise (Gigaplane) , SGI Challenge.
  - Not in: DEC AlphaServer 8400, Intel Pentium Pro.
- Number of outstanding transactions varies
  - DEC AlphaServer 8400, SGI Challenge = 8
  - Sun Ultra Enterprise (Gigaplane) = 112

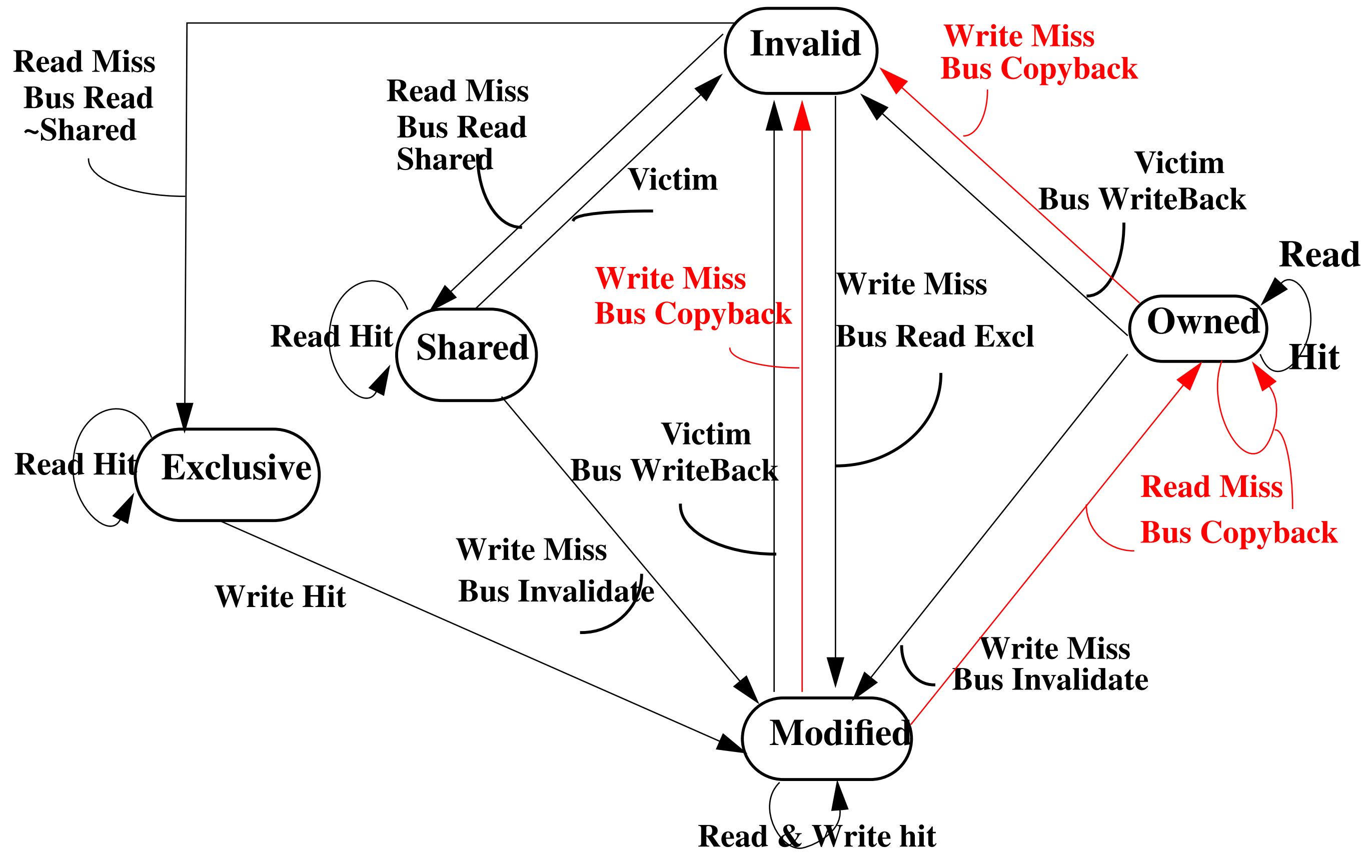
**Owned and Shared signals?**

- Handled in many different ways

**Main Disadvantage: More complex design (transactions no longer atomic)**

# MOESI Snooping Protocol State Machine

**Snooping State Machine for CPU 0 (Red = CPU 1 operations)**




**Additional State: Exclusive (E): write allowed without bus transaction**

**Additional Snoop Signal: Shared**

**Additional Invalriance: If a line is in state E, it must be invalid in all other caches.**

**Invariance: If a line is in state M in a cache, it must be invalid in all other caches.**

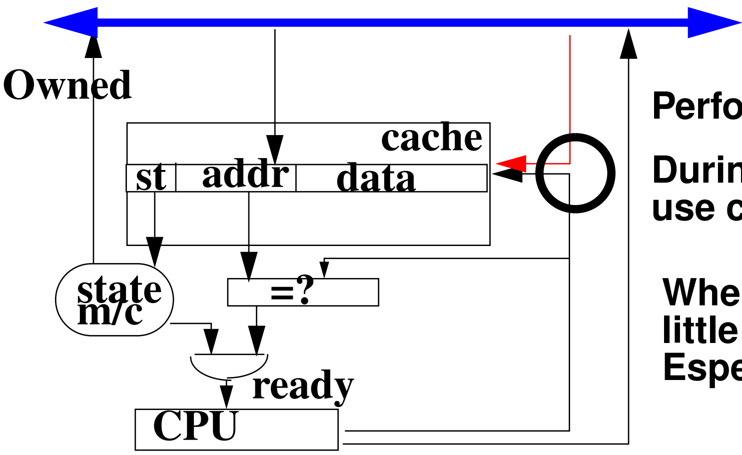
**Invariance: A line can be in state O in at most one cache.**

**Complexity:**

- Write and State transition (E to M) must be atomic, OR
- Exclusive state must be treated as OWNED.

# Performance Improvement: Dual Tags

**Perfomance probelm due to single tag copy**



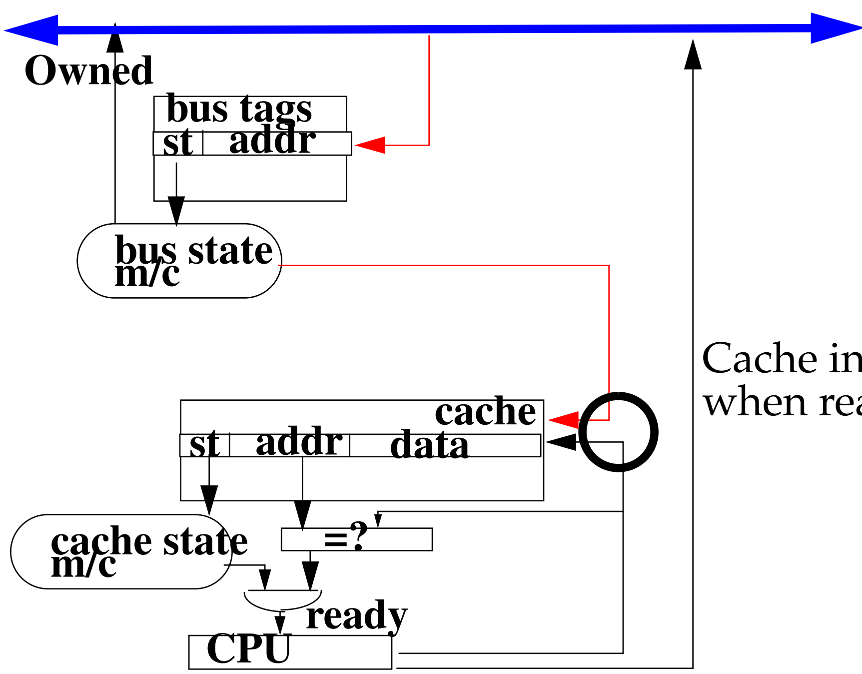

**Performance problem:**

**During bus snoops, CPU cannot use cache.**

**When Bus is busy CPU gets very little cache bandwidth. Especially with Split Bus**

**Solution:**

- Two tag copies: Bus Tags for snooping, separate tags for CPU.
- Bus Tags filter out unrelated transactions (CPU Tags get used only when state change really required).

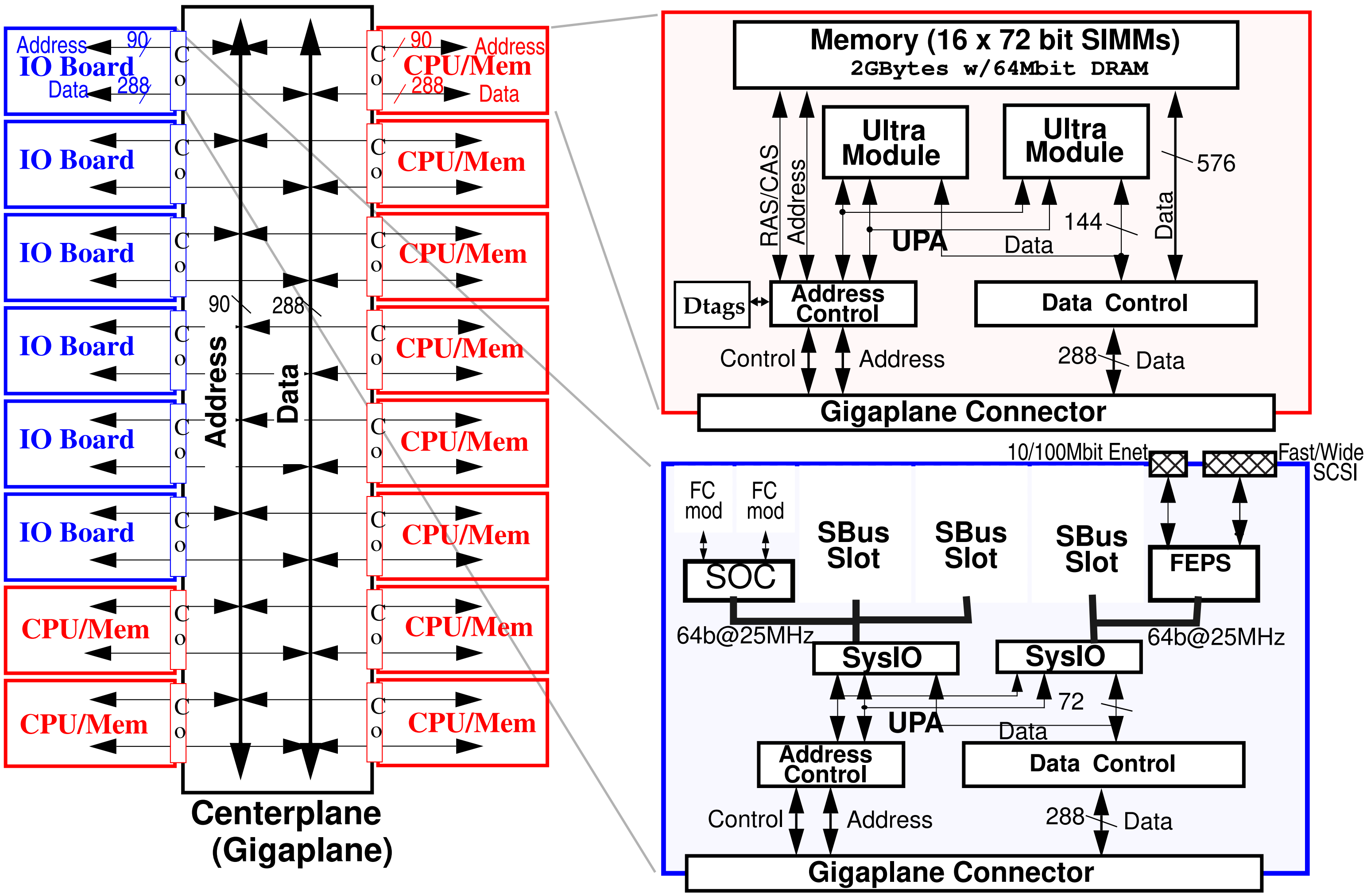



Cache intervention only when really necessary

**New problems with this:**

- Bus Tags and CPU Tags must be consistent with each other.

# Sun Ultra Enterprise System Architecture



# Memory Interleaving: why it's useful

**Non-Interleaved**

bank0: 0-15

bank1: 16-32

access sequence: 0, 1, 2, 3, 16, 17, 18, 19

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| bank 0: | 0 | 1 | 2 | 3 | | | | |
| bank 1: | | | | | 16 | 17 | 18 | 19 |

**Interleaved**

bank0: 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30

bank1: 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31

access sequence: 0, 1, 2, 3, 16, 17, 18, 19

| | | | | |
|---|---|---|---|---|
| bank 0: | 0 | 2 | 16 | 18 |
| bank 1: | 1 | 3 | 17 | 19 |

- Spatial and temporal locality causes access in bursts to a memory address range
- Interleaving helps to spread these accesses to multiple banks

# IO Systems

**DMA: Direct Memory Access**

- IO Controller can read/write memory
  - Like CPU, can cache data.
  - IO device needs to participate in snooping or coherence maintained by software.
  - Many high performance IO devices use DMA.
  - Happens in parallel with CPU computation.
- DVMA: Direct Virtual Memory Access
  - IO Controller has its own IO MMU (Memory Management Unit).
  - Roughly equivalent to “gather/scatter” of channels.
  - Translates between IO Bus address (e.g. SBus/PCI address) to system (physical) address.

**PIO: Programmed Input Output**

- CPU reads/writes IO locations
  - Can be high performance as well (eg. graphics frame buffers) but CPU tied up doing IO.
  - Usually uncached.

**Interrupts**

- IO-to-CPU interrupts (many reasons, often for completion of DMA operation).
- CPU-to-CPU interrupts (usually for page demap in MMU).

**IO looks simple, BUT IT ISN’T (especially for shared memory multiprocessors) !!**

- Wide variety of IO devices and bridges.
- Ordering rules are complicated.
- Deadlock/starvation opportunities abound.
- Needs careful cooperation in design between hardware and OS.

# Comparison of some Large SMP Buses

| System/Bus/CPU | Max # CPUs | Sustainable System Data B/W (MB/s) | System Transaction throughput (M trans/s) | Read Latency (ns) |
|---|---|---|---|---|
| Sun Ultra Enterprise 6000/Gigaplane/Ultra SPARC-1 @167 MHz | 30 | 2670 | 41.75 | 306 |
| DEC AlphaServer 8400 5/300/ ? /Alpha 21164 @300 MHz | 12 | 1600 | 25 | 400 |
| SGI Challenge/ POWERpath-2 /MIPS R10K @200 MHz | 36 | 1200 | 9.5 | 1115 |
| HP T 520/?/PA-7150 @ 120 MHz | 14 | 960 | 15 | ?? |
| Sun SC2000E/2xXDBus /SuperSPARC-2 @85 MHz | 20 | 625 | 9 | 1200 |