The MESI protocol

- As described earlier, in MSI, a cache block can be in one of three states
  - *Invalid (uncached) :* not in the cache (not valid in any cache)
  - *Shared/clean:* cached in one or more processors and memory is up-to-date
  - *Modified/dirty/exclusive*: one processor (owner) has data; memory out-of-date
- The MESI protocol regroup the Shared and Modified states into three states:
  - *Invalid (uncached):* same as in MSI
  - *Shared:* cached in more than one processors and memory is up-to-date
  - *Exclusive*: one processor (owner) has data and it is clean (clean but not shared)
  - Modified: one processor (owner) has data, but it is dirty
- If MESI is implemented using a directory, then the information kept for each block in the directory is the same as the three state protocol:
  - Shared in MESI = shared/clean but more than one sharer
  - Exclusive in MESI = shared/clean but only one sharer
  - Modified in MESI = Exclusive/Modified/dirty
- However, at each cached copy, a distinction is made between shared, exclusive and modified (rather than only shared and modified).

The MESI protocol

- **On a read miss** (local block is invalid), load the block and change its state to
  - "exclusive" if it was uncached in memory
  - "shared" if it was already shared, modified or exclusive
    - if it was modified, the owner will send you a clean copy
    - if was modified or exclusive, the previous owner will change the state of the block to "shared" in its cache.
- **On a write miss**: same as read miss, except set the state to "modified"
  - ➢ copies in other caches (if any) are invalidated
- **On a write hit** to a "modified" block, do nothing
- **On a write hit** to an "exclusive" block change the block to "modified"
  - ➢ no need for invalidation. ← this is the main advantage of MESI over MSI
- **On a write hit** to a "shared" block change the block to "modified" and invalidate the other cached copies.
- When a modified block is evicted, write it back .

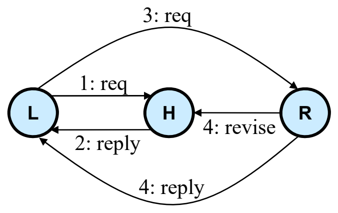
- In snooping bus implementations of MESI, on a read miss, we need to set its state correctly to "shared" (if the block is in some other cache(s)) or "exclusive" (if not).
- To take full advantage of MESI, should know when a block is to be changed from shared to exclusive

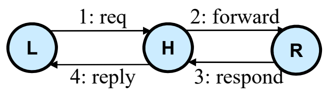
The MESI protocol

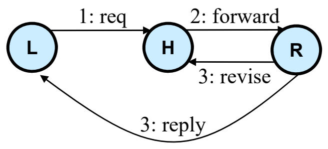
- If MESI is implemented as a snooping protocol, then the main advantage over the three state protocol is when a read to an uncached block is followed by a write to that block.
  - After the uncached block is read, it is marked "exclusive" – (need a scheme to know that it was uncached).
  - Note that, when writing to a shared block, the transaction has to be posted on the bus so that other sharers invalidate their copies.
  - But when writing to an exclusive block, there is no need to post the transaction on the bus. Hence, by distinguishing between shared and exclusive states, we can avoid bus transactions when writing on an exclusive block.
  - However, now a cache that has an "exclusive" block has to monitor the bus for any read to that block. Such a read will change the state to "shared".
  - What if a block was shared in two caches and is evicted from one of them. Should we detect this case and set the block to Exclusive in the other cache?
- This advantage disappears in a directory protocol since after a write onto an exclusive block , the directory has to be notified to change the state to modified.

Latency optimization

1) Forwarding requests







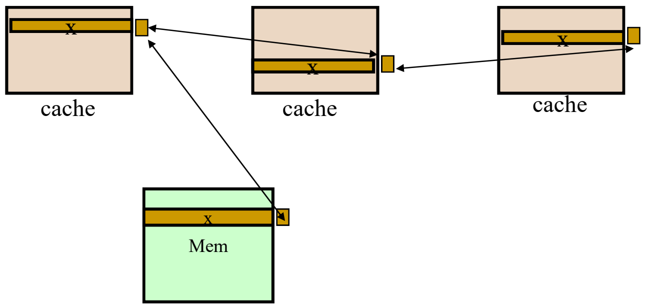
2) Use SRAM for directories (hardware optimization)

3) Overlap activities on the critical path
  - parallel multiple invalidation
  - parallel lookup of directory and memory at home node.

Storage overhead

- In the simplest representation of a directory entry, a *full bit vector* is used for each entry (one bit used to indicate presence in each node.)
  - storage overhead doesn't scale well with number of nodes.
- Larger blocks (cache lines) means lower overhead
- For very large number of nodes, may use a list of sharers instead of a bit vector
  - Lower overhead if only few sharers
  - Example; for 1024 processors, overhead is reduced if fewer than 100 sharers
- May reduce overhead further by keeping only directory entries for the blocks that are cached (uncached blocks do not need an entry)
  - Can keep the directory entries for the cached blocks in a hash table (associative cache structure) – should invalidate cached copies when the directory entry is removed (evicted) from the hash table.

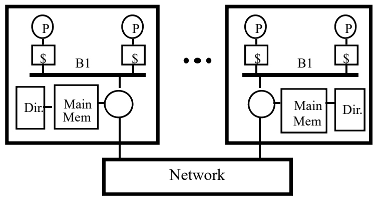
Cache-based Directory Schemes



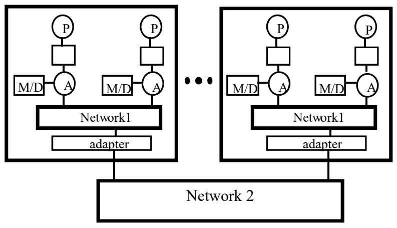
- Keep the information about the sharers of a cached block in the cache by linking the replicated cached entries in a linked list rather than storing a list of sharers with the block in the main memory.
- When a processor caches a block, it inserts itself at the front of the linked list
- To invalidate a cache block in the other caches, follow the link list (easier if a doubly link list)
- Scalable Coherent Interface (SCI) IEEE Standard

Hierarchical approaches to coherence

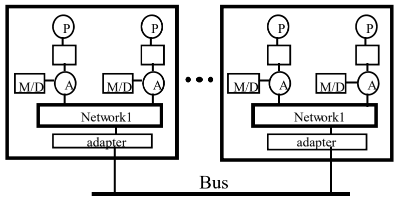
- Multi-levels - especially useful for multi-node systems, when each node is a multiprocessor (example: multi SMPs)
- Examples of two-level systems:



Snooping-directory

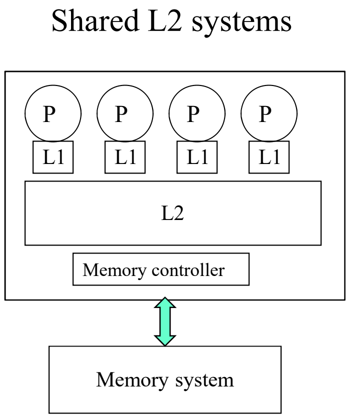


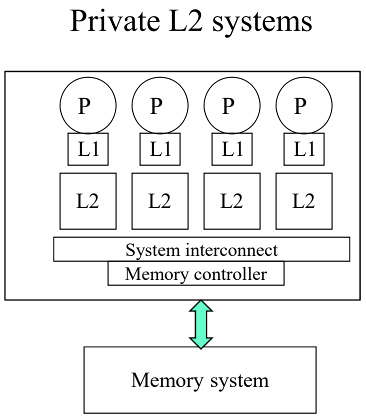
Directory-directory



Directory-snooping

Cache organization in multicore systems

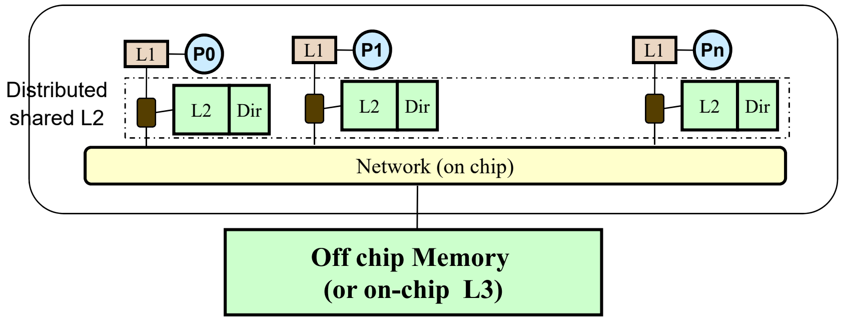




- Examples: Intel Core Duo Pentium
- Uses MESI (Modified, Exclusive, Shared, Invalid) cache coherence protocol to keep the L1 data coherent

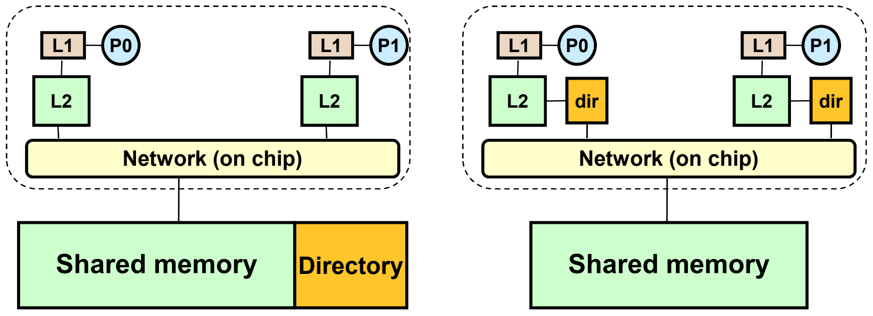
- Examples: AMD Dual Core Opteron
- Uses MOESI (M + Owned + ESI) cache coherence protocol to keep the L2 data coherent (L1 is inclusive to L2)

Example of distributed directories in CMPs



- Directories are used to keep track of the state of shared entities that are cached in multiple private caches.
- If the L2 modules form a shared cache space, then the directories perform a role very similar to their roles in distributed shared memory systems.
  - Preserve coherence in the private L1 caches
  - One directory entry for each entry in L2
  - Location of a cache line in L2 is determine by address of cache entry

Example of distributed directories in CMPs



- If each L2 module is private to the corresponding core, then on chip directories may be used as replacement for a centralized directory
- Each cache block is associated with a directory entry.
  - Only cache blocks that are on chip need to have directory entries
- How do you organize and distribute the directory entries among tiles?
  - location of directory entry (called its home) is determined by the address.

Handling atomic operations ($5.5)

Example: “Atomic Swap” interchanges a value in a register for a value in memory

- loads the value from a memory location into the register
- stores the value in register into the memory location

Atomic swap can be used to implement locks:

- The lock is represented by a memory location, L
  - L=1 → locked
  - L=0 → not locked

```
Lock (L):
         Put 1 in Register, R
   Loop: Atomic Swap (R, L)
         BNEZ  R, Loop
```

```
Unlock(L):
         Store 0 into memory location L
```

49

Possible implementations of Atomic Swap

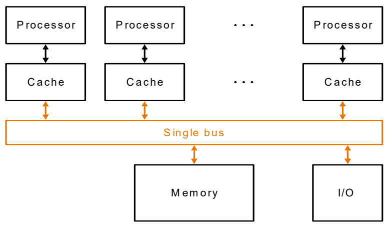
May use two special load/store instructions and guarantee that no other instruction will change the loaded/stored memory location in between.

- Load linked: LL R2, xx(R1)   /* load from memory location L=xx(R1)
- Store conditional: SC R3, xx(R1)

If a single CPU/memory: OS does not allow context switching between LL and LC

If shared memory system:

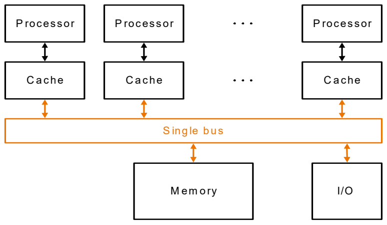
1. The memory controller may keep a record that an LL was issued to location L and does not allow a store or another LL to L before an SC is executed on L.
2. A system with no cache coherence can get the bus to perform LL and keep it until it performs an LC.
3. A cache in an MSI coherent system executing LL can set L to Modified (a miss on L generates a “request to write”) and ignores requests for L on the bus until the LC is executed.



50

A more efficient implementation of "Lock"

In a system that applies MSI coherence, using atomic swaps to implement locks (as described in the last slide) will cause excessive bus traffic if multiple processors compete for a lock.



Example:

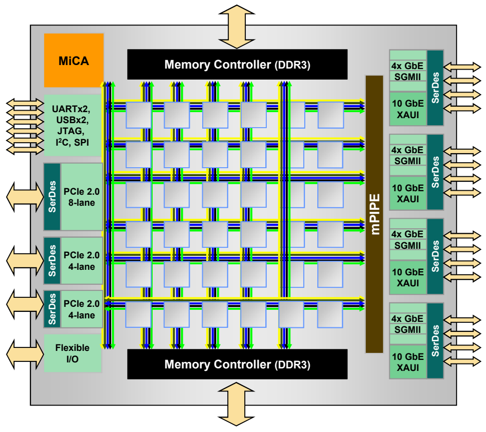
1) P1, P2 and P3 are competing for a lock, L.
2) P1 gets L (is exclusive in P1)
3) P2 executes a swap to get the lock (gets it in exclusive) but fails
4) P3 executes a swap to get the lock (gets it in exclusive) but fails
5) ….

```
Optimized Lock (L):
  Loop: load R, L          /* load L into R
        BNEZ R, Loop       /* loop on local copy
        Put 1 in Register, R
        Atomic Swap (R,L) /* LL R1,L; SC R,L; R←R1
        BNEZ  R, Loop
```

Barrier synchronization

- A barrier synchronization between N threads can be implemented using a shared variable initialized to N.
- When a processor reaches the barrier, it decrements the shared variable by 1 and waits (in a busy wait loop) until the value of the variable is equal to zero before it leaves the barrier.
- Need locks???
- What if there is no shared variables (distributed memory machines)?
- Can you synchronize using special hardware?

The Tilera TILE-Gx36™ Architecture:



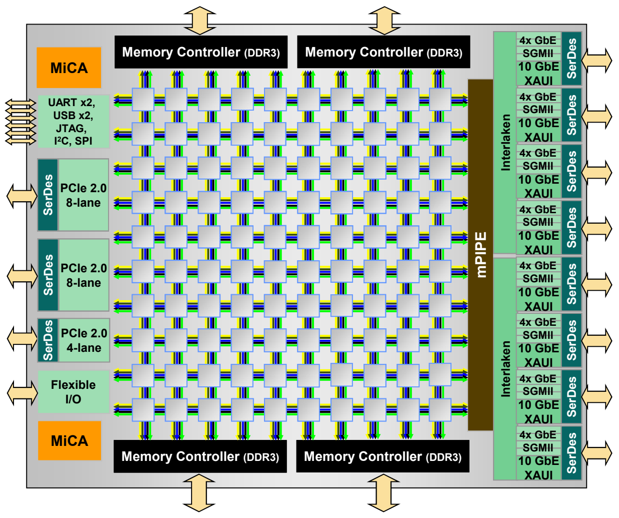
- 36 Processor Cores
- 866M, 1.2GHz, 1.5GHz clk
- 12 MBytes total cache

- 40 Gbps total packet I/O
  - 4 ports 10GbE (XAUI)
  - 16 ports 1GbE (SGMII)
- 48 Gbps PCIe I/O
  - 2 16Gbps Stream IO ports
- Wire-speed packet engine
  - 60Mpps

- MiCA engine:
  - 20 Gbps crypto
  - Compress & decompress

TILE-Gx100™:

Complete System-on-a-Chip with 100 64-bit cores

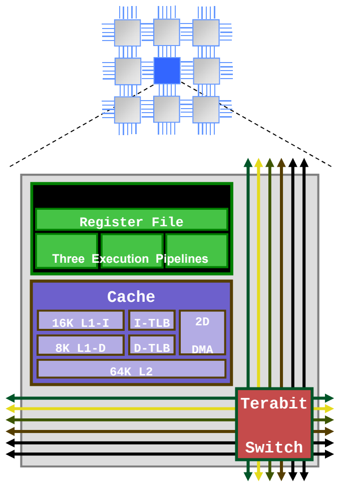


- 1.2GHz – 1.5GHz
- 32 MBytes total cache
- 546 Gbps peak mem BW
- 200 Tbps iMesh BW
- 80-120 Gbps packet I/O
  - 8 ports XAUI / 2 XAUI
  - 2 40Gb Interlaken
  - 32 ports 1GbE (SGMII)
- 80 Gbps PCIe I/O
  - 3 StreamIO ports (20Gb)
- Wire-speed packet eng.
  - 120Mpps
- MiCA engines:
  - 40 Gbps crypto
  - compress & decompress

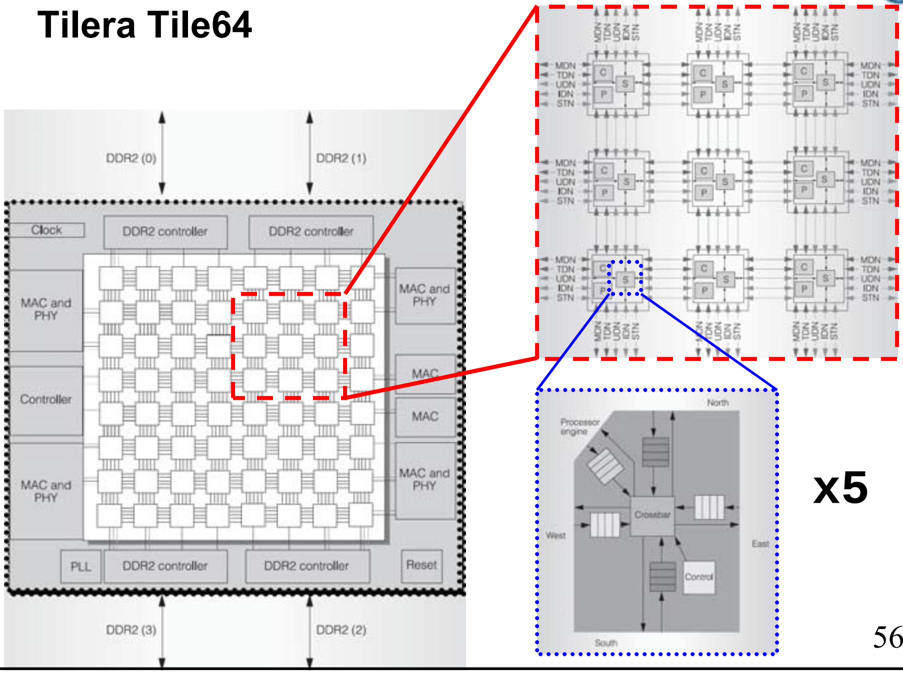
The Tilera core

- Processor
  - Each core is a complete computer
  - 3-way VLIW CPU
  - Protection and interrupts
- Memory
  - L1 cache and L2 Cache
  - Virtual and physical address space
  - Instruction and data TLBs
  - Cache integrated 2D DMA engine

- Runs SMP Linux
- Runs off-the-shelf C/C++ programs
- Signal processing and general apps



Tilera Tile64


DDR2 (0)
DDR2 (1)
Clock
DDR2 controller
DDR2 controller
MAC and PHY
MAC and PHY
MAC
MAC
Controller
MAC and PHY
MAC and PHY
PLL
DDR2 controller
DDR2 controller
Reset
DDR2 (3)
DDR2 (2)
MDN TDN UDN IDN STN
C S P
Processor engine
North
Crossbar
West
East
Control
South
x5