Computer Architecture: Intro to Thread-level Parallelism

Berk Sunar and Thomas Eisenbarth

ECE 505

WPI

Thread Level Parallelism

- About 10 years ago, ILP improvements (single core performance) reaches limit

Alternative approach: Multiprocessors

- Several cores on same die
- Cores share resources, but can execute independent threads (MIMD):
  - Parallel Processing: tightly coupled threads collaborate on single task
    - Supercomputing, research computing, e.g. simulations
  - Request-Level Parallelism/multiprocessing: independent execution of independent threads on same platform
    - Cloud computing / data centers

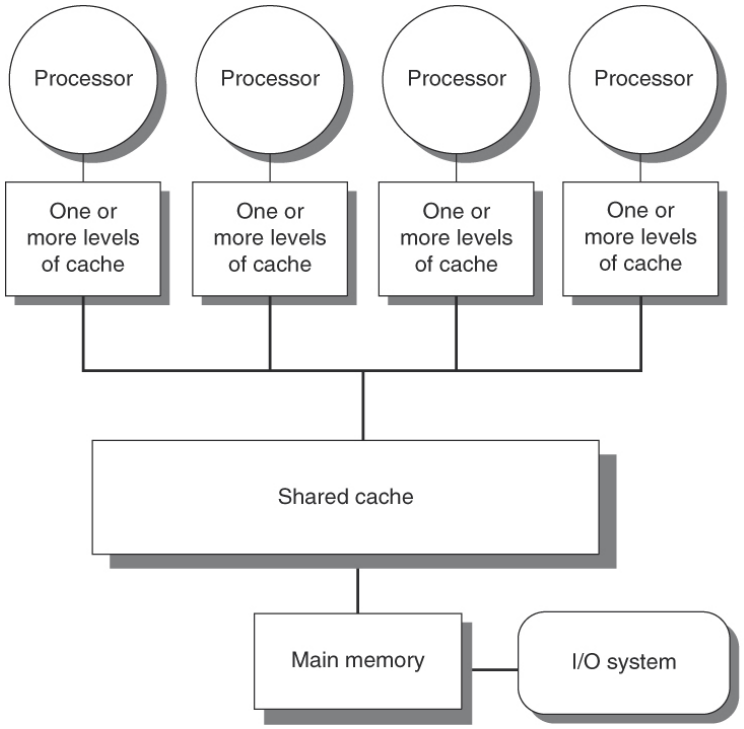
Towards Multiprocessors

- MIMD multiprocessor with n processors needs *at least* n threads to exploit full performance
- TLP vs ILP: parallelism is identified at high level →thread can execute independently for a while
  - Parallelism identified by programmer (not automatic as ILP)
- Multiprocessors can also exploit Data level parallelism
  - but SIMD processors (e.g. GPUs) are more efficient for that

Multiprocessor Architectures

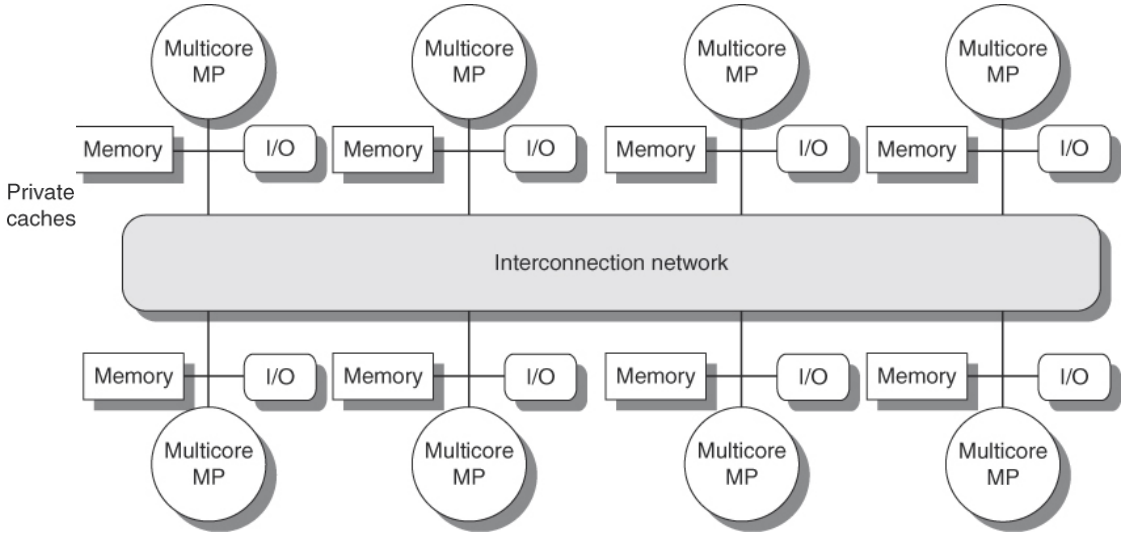
- Symmetric (shared memory) Multiprocessors (SMP):
  - Small number of cores, usually 2 – 8
  - Processors share a single centralized memory
  - Processors have equal *symmetric* access to mem (uniform memory access (UMA))
- Distributed Shared memory Multiprocessor (DSM)
  - Memory distributed to support higher core count
  - Memory and address space shared in both architectures
- Problems:
  - Increasing memory bandwidth (more cores) incurs access latency
  - More cores raises need for high speed interconnect

SMP vs DSM



Symmetric Multiprocessor

- Shared Memory/IO
- Uniform access times/rights



Distributed Shared Multiprocessor

- More cores with local memories
- Interconnect allows access to other processors and resources
  - Each core can still access all memory, but access times vary

Parallel Processing Challenges

Two main challenges

- Limited parallelism in most programs
- High cost of communication between threads

Example:

Goal: speedup of 80 with 100 cores

Q: What fraction of original task can be sequential?

Recall: (*s*: speedup for *f*: fraction)

Amdahl's Law: Speedup = $$\frac{t_{old}}{(1-f)\cdot t_{old}+\frac{f}{s}\cdot t_{old}} = \frac{1}{1-f+f/s}$$

Find f to achieve speedup of 80 with 100 cores

- $\frac{1}{1-f+f/s} = 80$, with s=100
- $\Rightarrow \frac{1}{80} = 1 + f\left(-1 + \frac{1}{100}\right)$,solve for f
- $f = \frac{\frac{1}{80}-1}{\frac{1}{100}-1}$ = .9975
- Hence 99.75% of original task must be parallelizable (good luck!)

Communication between Cores

Communication delays:

- With shared memory: L3 $ access: 35 – 50 cycles
- Between chips: 100 – 500 cycles or more

Example: 32 core chip, at 3.3GHz, 0.5 CPI base; 200ns reference time to remote memory; cores stall only on remote memory reference

- Compare base performance to case w\0.2% instructions have remote reference

Communication Cost Example

- Base CPI= 0.5
- CPI with references =
  base CPI + reference rate x reference cost
  reference cost= 200ns x 3.3GHz (or /.3ns)
  = 660 cycles
- Hence CPI = .5 + .2% x 660 = .5 + 1.3 = 1.8
- Hence, the all local references task executes
  1.8/.5 = 3.6 times faster than the non-local
  - Reducing remote data access for speedup:
    - In HW: using Caches In SW: restructuring data

Memory Access fur Multiprocessors

- CPU connects directly to dedicated memory chip through *memory bus*
  - Cores of same chip have symmetric access
  - Cross-chip memory access goes through *owner* chip (asymmetric)
- Data types:
  - Private data: only used by one core
  - Shared data: used by many cores, for communication
- Caching: private data is easy (lives in cache)
  - Access time reduced *and* memory bandwidth reduced
- Caching shared data: data may be replicated in several caches
  - Reduces access time, memory bandwidth and contention (several processors trying to read same data at same time)
  - BUT: New Problem: Cache coherence

Multiprocessor Cache Coherence

Cache Coherence Problem: processors see individual caches→ values might differ

Ex:

Time	Event	Cache of Core A	Cache of Core B	Memory Content X
0				1
1	A_Ld X	1		1
2	B_Ld X	1	1	1
3	A_St X, 0	0	1	0

A read should return the most recently written value (ideally)

Multiprocessor Cache Coherence

- Coherence: what values can be returned by a read:
  1. A read of X by core P, following a write of P to X and no other writes to X, returns value written by P
  2. A read to X by P following a write by Q to X returns written value if (i) enough time has passed and (ii) no other writes occur during that time
  3. Writes to same location are *serialized:* subsequent writes to X are seen by all processors in same order

  ➢ Reads can be reordered, but writes must be executed in program order

Cache Coherence Protocols: Basics

In coherent processor, cache provides

- Migration: value is moved to local cache and used in *transparent* way (reducing latency/BW)
- Replication: copies of data in several caches for multiple reads (reducing latency / contention)

Provided by two classes of protocols:

- Directory based: sharing status of memory block is stored in *directory*: Centralized for SMP (in outermost cache); Distributed for DSM (centralized would result in contention) (later)
- Snooping: every cache keeps track of sharing state of its content. Each cache snoops memory bus to see changes (SMP); for DMP, snooping might be combined at top level with directory within each multicore

Snooping Coherence Protocols

Write invalidate Protocol:

- Invalidates other copies on write
- Ensures exclusive access on write
  - Enforces write serialization

Event	Bus Activity	Cache of Core A	Cache of Core B	Memory Content X
				1
A_Ld X	$ miss for X	1		1
B_Ld X	$ miss for X	1	1	1
A_St X, 0	Invalidation for X	0		0
B_Ld X	$ miss for X	0	0	0

Additional *owner* state for data is needed (owner cache is in charge of updating memory and other caches)

Snooping Coherence Protocols

Write Update (or broadcast) Protocol:

- Updates all cached copies of data on write
- Consumes considerably more bandwidth

➢ Write invalidate Protocol is the much more common choice and widely used

Write invalidate Protocol Implementation

- Invalidation via Shared memory access bus (between private L1/L2 $ and shared L3 $ on i7)
  - Core gets bus access and broadcasts address to invalidate → other cores are snooping If two cores attempt parallel broadcast, serialization is done via bus access -> implicit write serialization
- Locating data on cache misses:
  - Write-through cache: most recent value in memory: fetch from memory
  - Write-back cache: apply same snooping technique for fetches: snoop for reads and provide value if 'dirty' (→ this is the common way to do last level caches)

Write invalidate Protocol Implementation

- Tracking of block state via state bits:
  - Valid bit, dirty bit, new shared bit indicates if invalidate broadcast is necessary (or data is private)
  - After invalidation, core gets exclusive access to block (shared becomes private)
  - Once other core requests access (seen by snoop), the block is made shared again

Snoopy Coherence Protocol Implementation: Example

- Each core has a FSM to implement snooping
  - Interacts with core and bus
- Both regular cache and snoop need to check cache address tags: duplication of checking HW
- Protocol with three states: invalid, shared, modified (latter implies private/exclusive)

Snoopy Coherence Protocol Implementation: Example

Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read hit	Processor	Shared or modified	Normal hit	Read data in local cache.
Read miss	Processor	Invalid	Normal miss	Place read miss on bus.
Read miss	Processor	Shared	Replacement	Address conflict miss: place read miss on bus.
Read miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place read miss on bus.
Write hit	Processor	Modified	Normal hit	Write data in local cache.
Write hit	Processor	Shared	Coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, since they do not fetch the data but only change the state.
Write miss	Processor	Invalid	Normal miss	Place write miss on bus.
Write miss	Processor	Shared	Replacement	Address conflict miss: place write miss on bus.
Write miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place write miss on bus.
Read miss	Bus	Shared	No action	Allow shared cache or memory to service read miss.
Read miss	Bus	Modified	Coherence	Attempt to share data: place cache block on bus and change state to shared.
Invalidate	Bus	Shared	Coherence	Attempt to write shared block; invalidate the block.
Write miss	Bus	Shared	Coherence	Attempt to write shared block; invalidate the cache block.
Write miss	Bus	Modified	Coherence	Attempt to write block that is exclusive elsewhere; write-back the cache block and make its state invalid in the local cache.

Only if address matches valid block in local cache

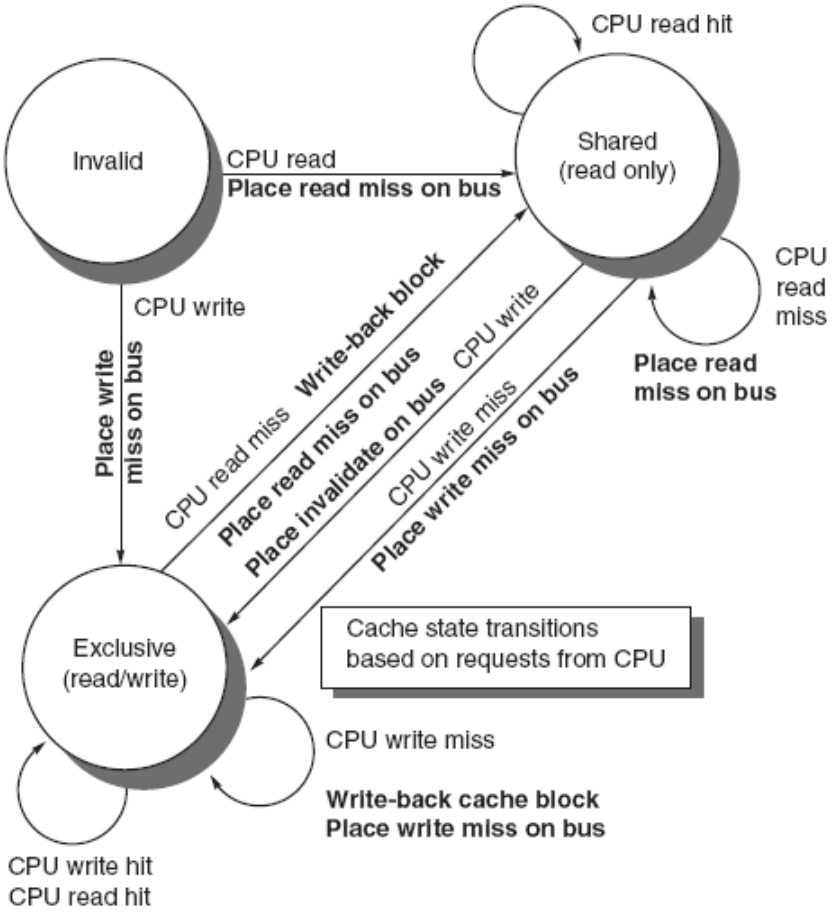
Snoopy Coherence Protocol Implementation: Example

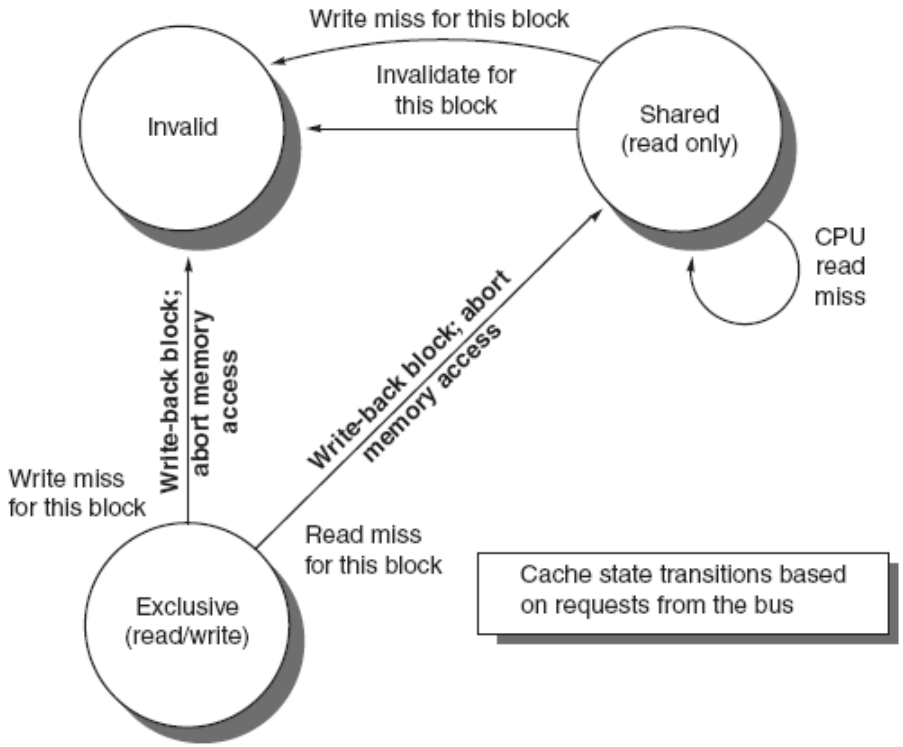
- Each core has a FSM to implement snooping
  - Interacts with core and bus
- Protocol with three states: invalid, shared, modified (latter implies private)

How it works:

- Block is either in shared state in one or many caches or in exclusive state in one cache only
- Transition to exclusive (requires write) invalidates copies of block in other caches (might force write-back on other core if exclusive on that core)
- Read miss on exclusive block triggers writeback

Snoopy Coherence Protocol Implementation: Example





Simplifying assumptions

- Protocol is *atomic*: no intervening operations can occur during operation execution
  - In reality: write miss detected, then acquire bus, then receive response
  - Nonatomic operations can result in deadlock:
    - Dining Philosophers problem
- Protocol only solves on-chip communication
  - Many modern chips support connection of several multiprocessor chips. That connection uses different protocol