

CSC 631: High-Performance Computer Architecture

Spring 2017
Lecture 11: Multiprocessors

Multiprocessing

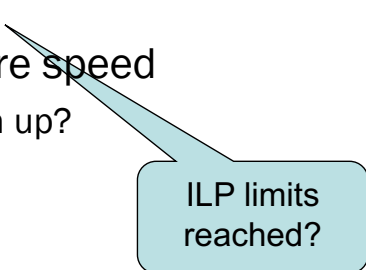
- Flynn's Taxonomy of Parallel Machines
 - How many Instruction streams?
 - How many Data streams?
- SISD: Single I Stream, Single D Stream
 - A uniprocessor
- SIMD: Single I, Multiple D Streams
 - Each “processor” works on its own data
 - But all execute the same instrs in lockstep
 - E.g. a vector processor or MMX

Flynn's Taxonomy

- MISD: Multiple I, Single D Stream
 - Not used much
 - Stream processors are closest to MISD
- MIMD: Multiple I, Multiple D Streams
 - Each processor executes its own instructions and operates on its own data
 - This is your typical off-the-shelf multiprocessor (made using a bunch of “normal” processors)
 - Includes multi-core processors

Multiprocessors

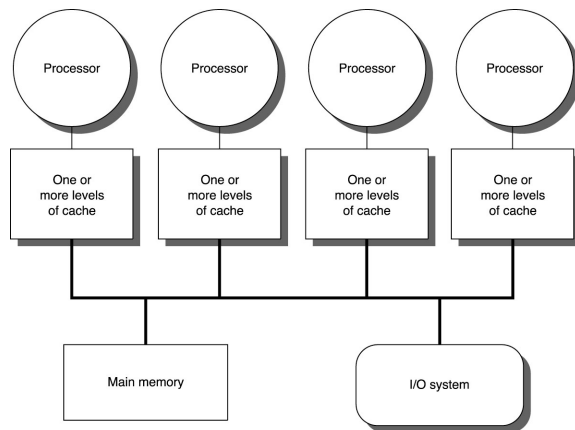
- Why do we need multiprocessors?
 - Uniprocessor speed keeps improving
 - But there are things that need even more speed
 - Wait for a few years for Moore's law to catch up?
 - Or use multiple processors and do it now?
- Multiprocessor software problem
 - Most code is sequential (for uniprocessors)
 - MUCH easier to write and debug
 - Correct parallel code very, very difficult to write
 - *Efficient* and correct is even harder
 - Debugging even more difficult (Heisenbugs)



ILP limits reached?

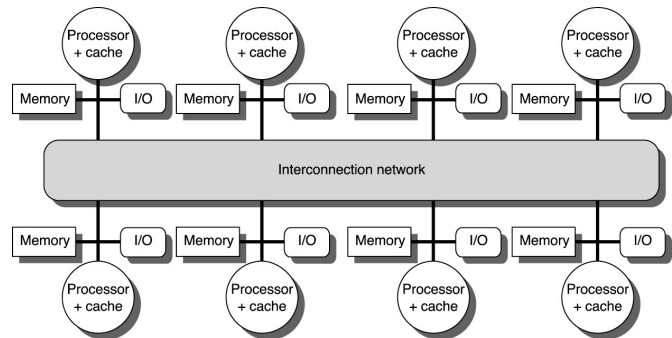
MIMD Multiprocessors

Centralized Shared Memory



© 2003 Elsevier Science (USA). All rights reserved.

Distributed Memory



© 2003 Elsevier Science (USA). All rights reserved.

Centralized-Memory Machines

- Also “Symmetric Multiprocessors” (SMP)
- “Uniform Memory Access” (UMA)
 - All memory locations have similar latencies
 - Data sharing through memory reads/writes
 - P1 can write data to a physical address A, P2 can then read physical address A to get that data
- Problem: Memory Contention
 - All processor share the one memory
 - Memory bandwidth becomes bottleneck
 - Used only for smaller machines
 - Most often 2,4, or 8 processors

Distributed-Memory Machines

- Two kinds
 - Distributed Shared-Memory (DSM)
 - All processors can address all memory locations
 - Data sharing like in SMP
 - Also called NUMA (non-uniform memory access)
 - Latencies of different memory locations can differ (local access faster than remote access)
 - Message-Passing
 - A processor can directly address only local memory
 - To communicate with other processors, must explicitly send/receive messages
 - Also called multicomputers or clusters
- Most accesses local, so less memory contention (can scale to well over 1000 processors)

Message-Passing Machines

- A cluster of computers
 - Each with its own processor and memory
 - An interconnect to pass messages between them
 - Producer-Consumer Scenario:
 - P1 produces data D, uses a SEND to send it to P2
 - The network routes the message to P2
 - P2 then calls a RECEIVE to get the message
 - Two types of send primitives
 - Synchronous: P1 stops until P2 confirms receipt of message
 - Asynchronous: P1 sends its message and continues
 - Standard libraries for message passing:
Most common is MPI – Message Passing Interface

Communication Performance

- Metrics for Communication Performance
 - Communication Bandwidth
 - Communication Latency
 - Sender overhead + transfer time + receiver overhead
 - Communication latency hiding
- Characterizing Applications
 - Communication to Computation Ratio
 - Work done vs. bytes sent over network
 - Example: 146 bytes per 1000 instructions

Parallel Performance

- Serial sections
 - Very difficult to parallelize the entire app
 - Amdahl's law

$$\text{Speedup}_{\text{Overall}} = \frac{1}{(1 - F_{\text{Parallel}}) + \frac{F_{\text{Parallel}}}{\text{Speedup}_{\text{Parallel}}}}$$

\Rightarrow

$\frac{\text{Speedup}_{\text{Parallel}} = 1024}{F_{\text{Parallel}} = 0.5}$
 $\text{Speedup}_{\text{Overall}} = 1.998$

$\frac{\text{Speedup}_{\text{Parallel}} = 1024}{F_{\text{Parallel}} = 0.99}$
 $\text{Speedup}_{\text{Overall}} = 91.2$

- Large remote access latency (100s of ns)
 - Overall IPC goes down

$$\text{CPI} = \text{CPI}_{\text{Base}} + \text{RemoteRequestRate} \times \text{RemoteRequestCost}$$

$$\text{CPI}_{\text{Base}} = 0.4 \quad \text{RemoteRequestCost} = \frac{400\text{ns}}{0.33\text{ns/Cycle}} = 1200 \text{ Cycles} \quad \text{RemoteRequestRate} = 0.002$$

$$\text{CPI} = 2.8$$

We need at least 7 processors just to break even!

This cost reduced with CMP/multi-core

Message Passing Pros and Cons

- Pros
 - Simpler and cheaper hardware
 - Explicit communication makes programmers aware of costly (communication) operations
- Cons
 - Explicit communication is painful to program
 - Requires manual optimization
 - If you want a variable to be local and accessible via LD/ST, you must declare it as such
 - If other processes need to read or write this variable, you must explicitly code the needed sends and receives to do this

Message Passing: A Program

- Calculating the sum of array elements

```
#define ASIZE 1024
#define NUMPROC 4
double myArray[ASIZE/NUMPROC];
double mySum=0;
for(int i=0;i<ASIZE/NUMPROC;i++)
    mySum+=myArray[i];
if(myPID=0) {
    for(int p=1;p<NUMPROC;p++){
        int pSum;
        recv(p,pSum);
        mySum+=pSum;
    }
    printf("Sum: %lf\n",mySum);
}else
    send(0,mySum);
```

Must manually split the array

“Master” processor adds up partial sums and prints the result

“Slave” processors send their partial results to master

Shared Memory Pros and Cons

- Pros
 - Communication happens automatically
 - More natural way of programming
 - Easier to write correct programs and gradually optimize them
 - No need to manually distribute data (but can help if you do)
- Cons
 - Needs more hardware support
 - Easy to write correct, but inefficient programs (remote accesses look the same as local ones)

Shared Memory: A Program

- Calculating the sum of array elements

```
#define ASIZE 1024
#define NUMPROC 4
shared double array[ASIZE];
shared double allSum=0;
shared mutex sumLock;
double mySum=0;
for(int i=myPID*ASIZE/NUMPROC; i<(myPID+1)*ASIZE/NUMPROC; i++)
    mySum+=array[i];
lock(sumLock);
allSum+=mySum;
unlock(sumLock);
if(myPID==0)
    printf("Sum: %lf\n",allSum);
```

Array is shared

Each processor sums up "its" part of the array

Each processor adds its partial sums to the final result

"Master" processor prints the result

Cache Coherence Problem

- Shared memory easy with no caches
 - P1 writes, P2 can read
 - Only one copy of data exists (in memory)
- Caches store their own copies of the data
 - Those copies can easily get inconsistent
 - Classic example: adding to a sum
 - P1 loads allSum, adds its mySum, stores new allSum
 - P1's cache now has dirty data, but memory not updated
 - P2 loads allSum from memory, adds its mySum, stores allSum
 - P2's cache also has dirty data
 - Eventually P1 and P2's cached data will go to memory
 - Regardless of write-back order, the final value ends up wrong

Cache Coherence Definition

- A memory system is coherent if
 1. A read R from address X on processor P1 returns the value written by the most recent write W to X on P1 if no other processor has written to X between W and R.
 2. If P1 writes to X and P2 reads X after a sufficient time, and there are no other writes to X in between, P2's read returns the value written by P1's write.
 3. Writes to the same location are serialized: two writes to location X are seen in the same order by all processors.

Cache Coherence Definition

- Property 1. preserves program order
 - It says that in the absence of sharing, each processor behaves as a uniprocessor would
- Property 2. says that any write to an address must eventually be seen by all processors
 - If P1 writes to X and P2 keeps reading X, P2 must eventually see the new value
- Property 3. preserves causality
 - Suppose X starts at 0. Processor P1 increments X and processor P2 waits until X is 1 and then increments it to 2. Processor P3 must eventually see that X becomes 2.
 - If different processors could see writes in different order, P2 can see P1's write and do its own write, while P3 first sees the write by P2 and then the write by P1. Now we have two processors that will forever disagree about the value of A.

Maintaining Cache Coherence

- Hardware schemes
 - Shared Caches
 - Trivially enforces coherence
 - Not scalable (L1 cache quickly becomes a bottleneck)
 - Snooping
 - Needs a broadcast network (like a bus) to enforce coherence
 - Each cache that has a block tracks its sharing state on its own
 - Directory
 - Can enforce coherence even with a point-to-point network
 - A block has just one place where its full sharing state is kept

Snooping

- Typically used for bus-based (SMP) multiprocessors
 - Serialization on the bus used to maintain coherence property 3
- Two flavors
 - Write-update (write broadcast)
 - A write to shared data is broadcast to update all copies
 - All subsequent reads will return the new written value (property 2)
 - All see the writes in the order of broadcasts
One bus == one order seen by all (property 3)
 - Write-invalidate
 - Write to shared data forces invalidation of all other cached copies
 - Subsequent reads miss and fetch new value (property 2)
 - Writes ordered by invalidations on the bus (property 3)

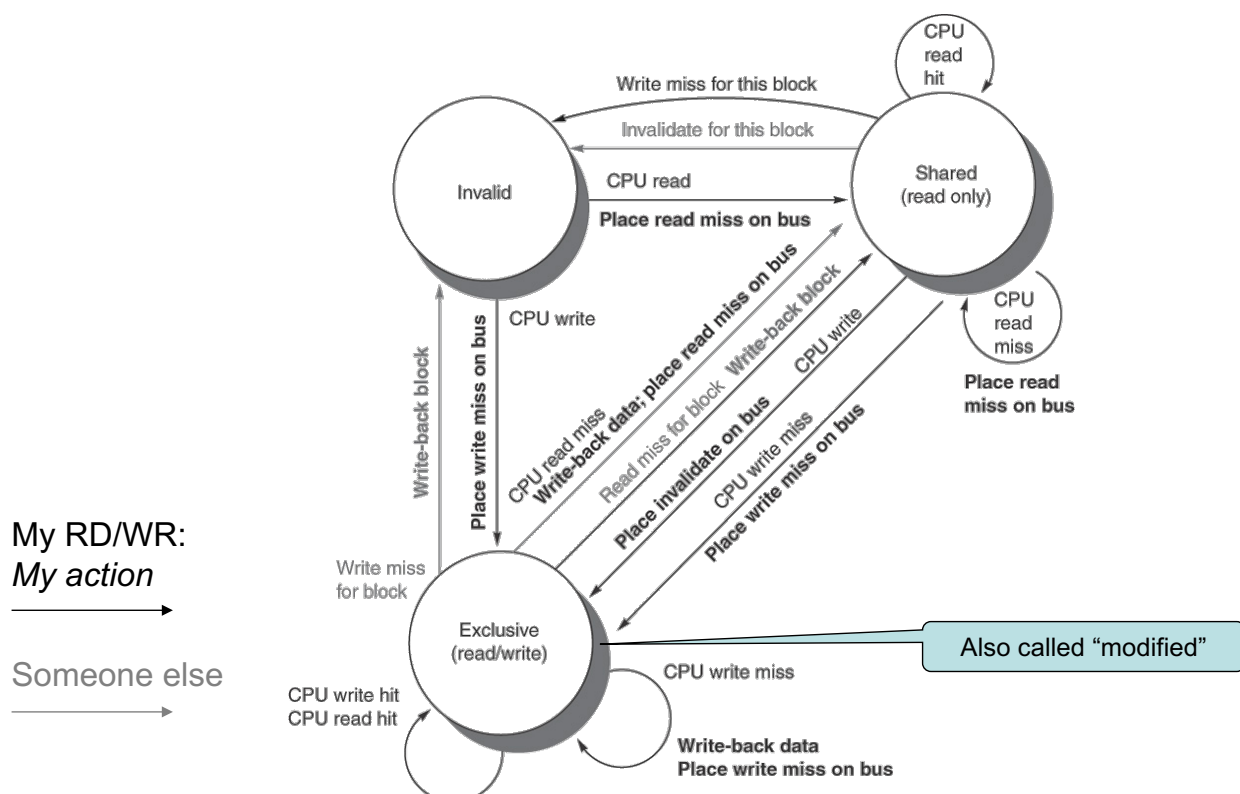
Update vs. Invalidate

- A burst of writes by a processor to one addr
 - Update: each sends an update
 - Invalidate: possibly only the first invalidation is sent
- Writes to different words of a block
 - Update: update sent for each word
 - Invalidate: possibly only the first invalidation is sent
- Producer-consumer communication latency
 - Update: producer sends an update, consumer reads new value from its cache
 - Invalidate: producer invalidates consumer's copy, consumer's read misses and has to request the block
- Which is better depends on application
 - But write-invalidate is simpler and implemented in most MP-capable processors today

MSI Snoopy Protocol

- State of block B in cache C can be
 - Invalid: B is not cached in C
 - To read or write, must make a request on the bus
 - Modified: B is dirty in C
 - C has the block, no other cache has the block, and C must update memory when it displaces B
 - Can read or write B without going to the bus
 - Shared: B is clean in C
 - C has the block, other caches have the block, and C need not update memory when it displaces B
 - Can read B without going to bus
 - *To write, must send an upgrade request to the bus*

MSI Snoopy Protocol



Cache to Cache transfers

- Problem
 - P1 has block B in M state
 - P2 wants to read B, puts a RdReq on bus
 - If P1 does nothing, memory will supply the data to P2
 - What does P1 do?
- Solution 1: abort/retry
 - P1 cancels P2's request, issues a write back
 - P2 later retries RdReq and gets data from memory
 - Too slow (two memory latencies to move data from P1 to P2)
- Solution 2: intervention
 - P1 indicates it will supply the data ("intervention" bus signal)
 - Memory sees that, does not supply the data, and waits for P1's data
 - P1 starts sending the data on the bus, memory is updated
 - P2 snoops the transfer during the write-back and gets the block

Cache-To-Cache Transfers

- Intervention works if some cache has data in M state
 - Nobody else has the correct data, clear who supplies the data
- What if a cache has requested data in S state
 - There might be others who have it, who should supply the data?
 - Solution 1: let memory supply the data
 - Solution 2: whoever wins arbitration supplies the data
 - Solution 3: A separate state similar to S that indicates there are maybe others who have the block in S state, but if anybody asks for the data we should supply it

MSI Protocol

- Three states:
 - Invalid
 - Shared (clean)
 - Modified (dirty)

MESI Protocol

- New state: exclusive
 - data is clean
 - but I have the only copy (except memory)
- Benefit: bandwidth reduction

Detecting Other Sharers

- Problem
 - P1 wants to read B, puts a RdReq on bus, receives data
 - How does P1 know whether to cache B in S or E state?
- Solution: “Share” bus signal
 - Always low, except when somebody pulls it to high
 - When P2 snoops P1’s request, it pulls “Share” to high
 - P1 goes to S if “Share” high, to E if “Share” low

Directory-Based Coherence

- Typically in distributed shared memory
- For every local memory block, local directory has an entry
- Directory entry indicates
 - Who has cached copies of the block
 - In what state do they have the block

Basic Directory Scheme

- Each entry has
 - One dirty bit (1 if there is a dirty cached copy)
 - A presence vector (1 bit for each node)
Tells which nodes may have cached copies
- All misses sent to block's home
- Directory performs needed coherence actions
- Eventually, directory responds with data

Read Miss

- Processor P_k has a read miss on block B, sends request to home node of the block
- Directory controller
 - Finds entry for B, checks D bit
 - If $D=0$
 - Read memory and send data back, set $P[k]$
 - If $D=1$
 - Request block from processor whose P bit is 1
 - When block arrives, update memory, clear D bit, send block to P_k and set $P[k]$

Directory Operation

- Network controller connected to each bus
 - A proxy for remote caches and memories
 - Requests for remote addresses forwarded to home, responses from home placed on the bus
 - Requests from home placed on the bus, cache responses sent back to home node
- Each cache still has its own coherence state
 - Directory is there just to avoid broadcasts and order accesses to each location
 - Simplest scheme:
If access A1 to block B still not fully processed by directory when A2 arrives, A2 waits in a queue until A1 is done

Shared Memory Performance

- Another “C” for cache misses
 - Still have Compulsory, Capacity, Conflict
 - Now have Coherence, too
 - We had it in our cache and it was invalidated
- Two sources for coherence misses
 - True sharing
 - Different processors access the same data
 - False sharing
 - Different processors access different data, *but they happen to be in the same block*

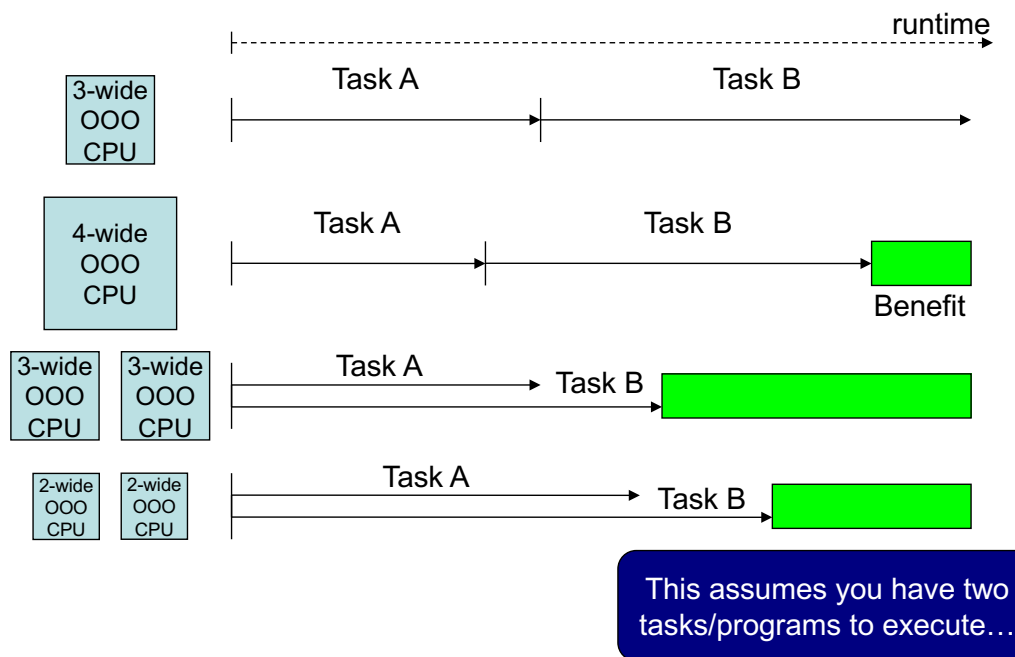
User Visible/Invisible

- All microarchitecture performance gains up to this point were “free”
 - in that no user intervention required beyond buying the new processor/system
 - recompilation/rewriting could provide even more benefit, but you get some even if you do nothing
- Multi-processing pushes the problem of finding the parallelism to above the ISA interface

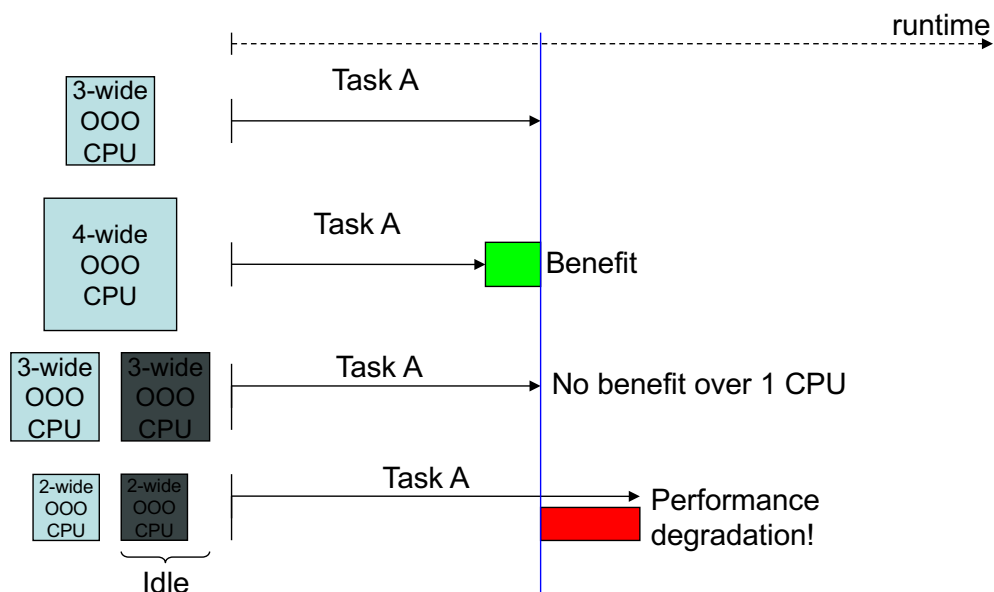
Multi-thread/Multi-core HW

- Benefits of going parallel
- Different HW organizations, tradeoffs

Workload Benefits



... If Only One Task

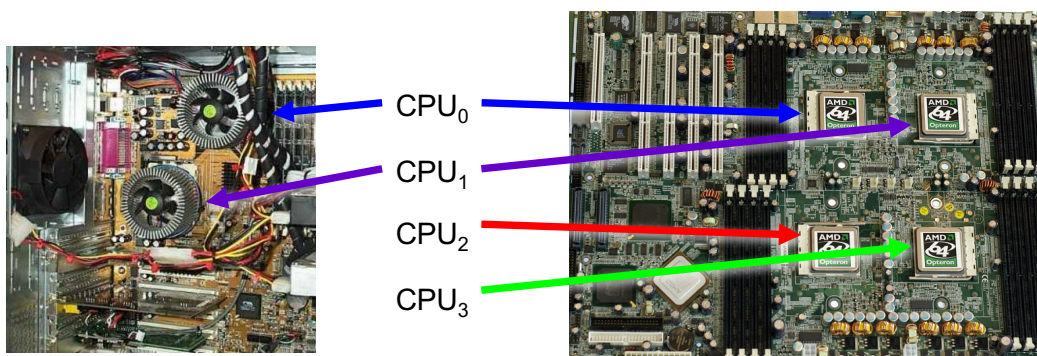


(Execution) Latency vs. Bandwidth

- Desktop processing
 - typically want an application to execute as quickly as possible (minimize latency)
- Server/Enterprise processing
 - often throughput oriented (maximize bandwidth)
 - latency of individual task less important
 - ex. Amazon processing thousands of requests per minute: it's ok if an individual request takes a few seconds more so long as total number of requests are processed in time

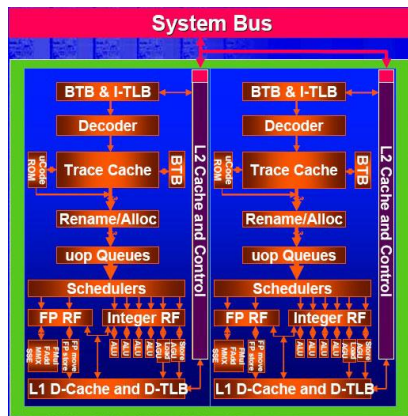
Implementing MP Machines

- One approach: add sockets to your MOBO
 - minimal changes to existing CPUs
 - power delivery, heat removal and I/O not too bad since each chip has own set of pins and cooling

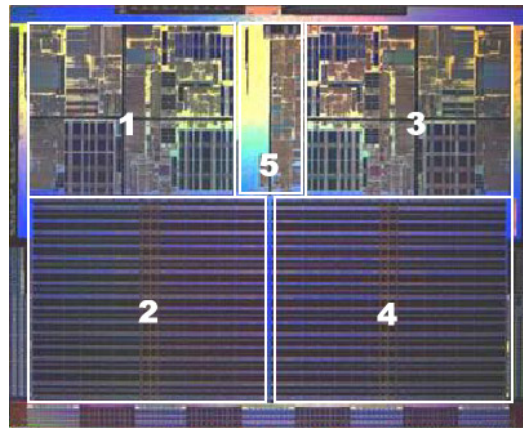


Chip-Multiprocessing

- Simple SMP on the same chip



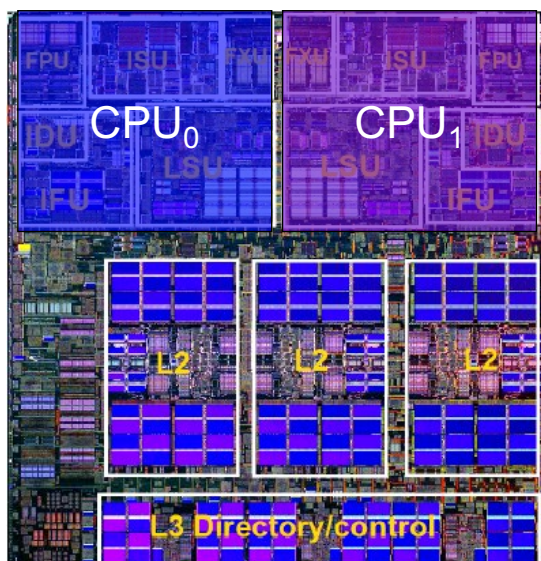
Intel "Smithfield" Block Diagram



AMD Dual-Core Athlon FX

Shared Caches

- Resources can be shared between CPUs
 - ex. IBM Power 5



L2 cache shared between both CPUs (no need to keep two copies coherent)

L3 cache is also shared (only tags are on-chip; data are off-chip)

Benefits?

- Cheaper than mobo-based SMP
 - all/most interface logic integrated on to main chip (fewer total chips, single CPU socket, single interface to main memory)
 - less power than mobo-based SMP as well (communication on-die is more power-efficient than chip-to-chip communication)
- Performance
 - on-chip communication is faster
- Efficiency
 - potentially better use of hardware resources than trying to make wider/more OOO single-threaded CPU

Performance vs. Power

- 2x CPUs not necessarily equal to 2x performance
- 2x CPUs \rightarrow $\frac{1}{2}$ power for each
 - maybe a little better than $\frac{1}{2}$ if resources can be shared
- Back-of-the-Envelope calculation:
 - 3.8 GHz CPU at 100W
 - Dual-core: 50W per CPU
 - $P \propto V^3$: $V_{\text{orig}}^3 / V_{\text{CMP}}^3 = 100\text{W} / 50\text{W} \rightarrow V_{\text{CMP}} = 0.8 V_{\text{orig}}$
 - $f \propto V$: $f_{\text{CMP}} = 3.0\text{GHz}$

Benefit of SMP: Full power budget per socket!

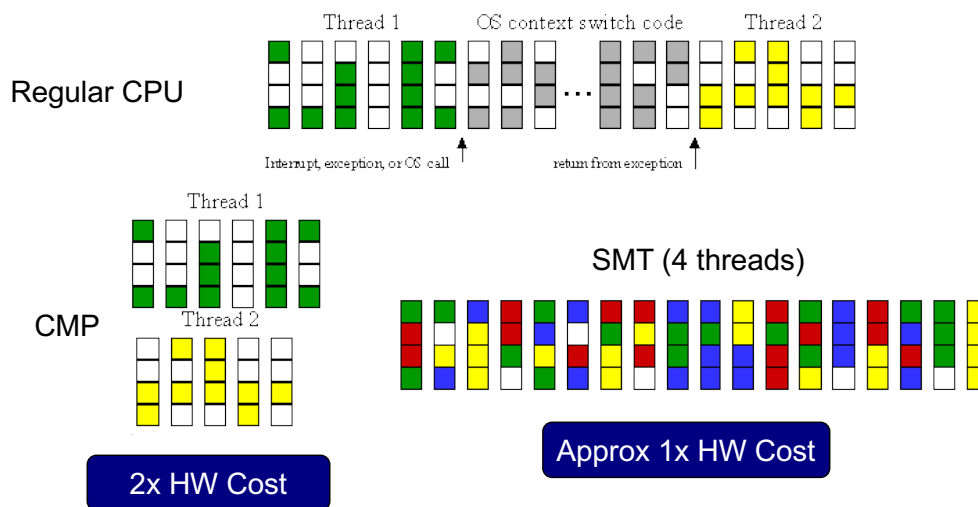
Multithreaded Processors

- Single thread in superscalar execution: dependences cause most of stalls
- Idea: when one thread stalled, other can go
- Different granularities of multithreading
 - Coarse MT: can change thread every few cycles
 - Fine MT: can change thread every cycle
 - Simultaneous Multithreading (SMT)
 - Instrs from different threads even in the same cycle
 - AKA Hyperthreading

Simultaneous Multi-Threading

- Uni-Processor: 4-6 wide, lucky if you get 1-2 IPC
 - poor utilization
- SMP: 2-4 CPUs, but need independent tasks
 - else poor utilization as well
- SMT: Idea is to use a single large uni-processor as a multi-processor

SMT (2)

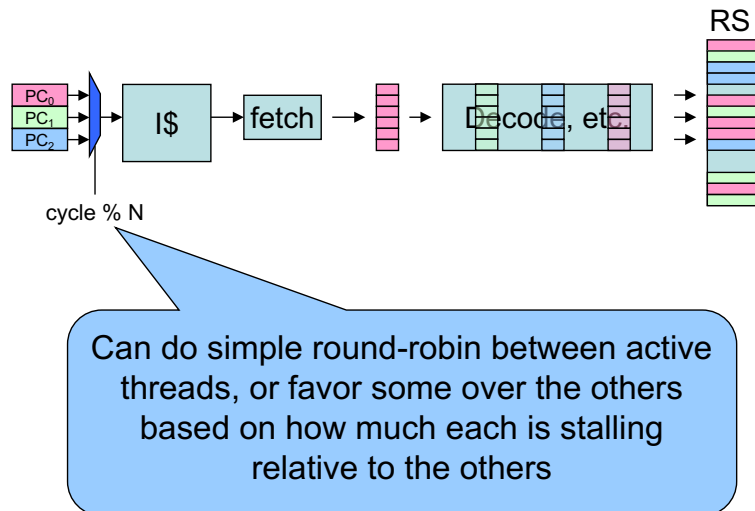


Overview of SMT Hardware Changes

- For an N-way (N threads) SMT, we need:
 - Ability to fetch from N threads
 - N sets of registers (including PCs)
 - N rename tables (RATs)
 - N virtual memory spaces
- But we don't need to replicate the entire OOO execution engine (schedulers, execution units, bypass networks, ROBs, etc.)

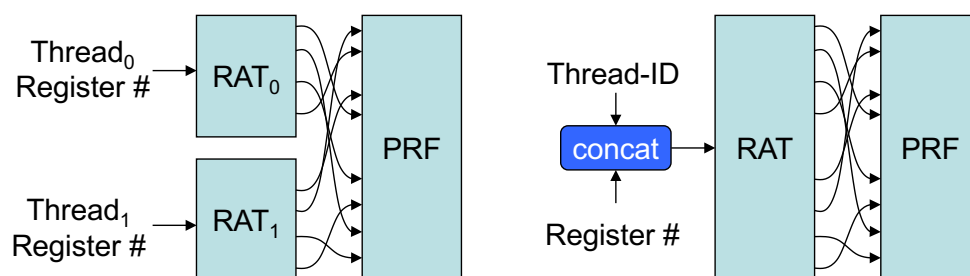
SMT Fetch

- Multiplex the Fetch Logic



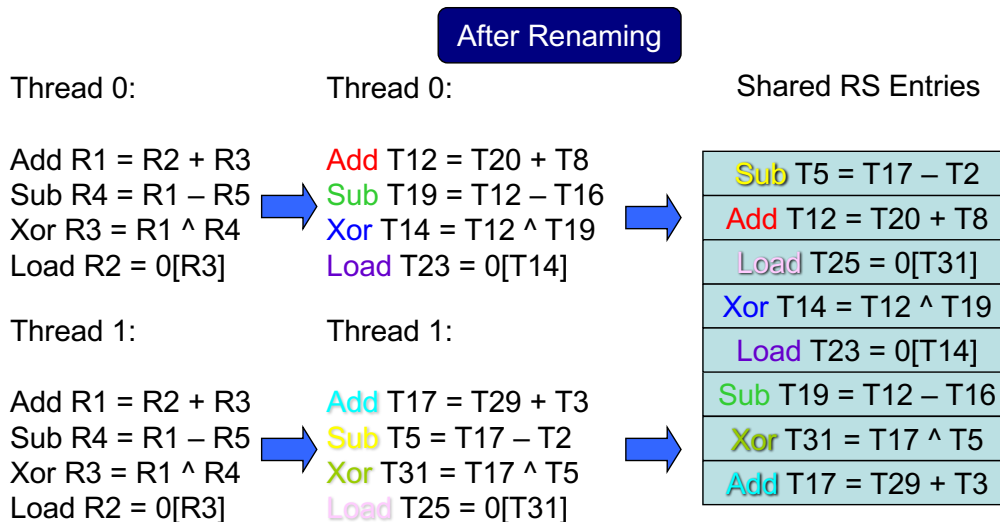
SMT Rename

- Thread #1's R12 \neq Thread #2's R12
 - separate name spaces
 - need to disambiguate



SMT Issue, Exec, Bypass, ...

- No change needed



SMT Cache

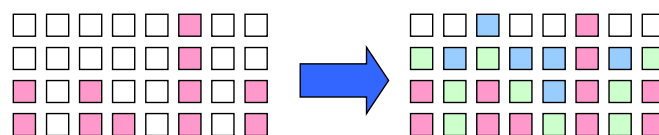
- Each process has own virtual address space
 - TLB must be thread-aware
 - translate (thread-id, virtual page) → physical page
 - Virtual portion of caches must also be thread-aware
 - VIVT cache must now be (virtual addr, thread-id)-indexed, (virtual addr, thread-id)-tagged
 - Similar for VIPT cache
 - No changes needed if using PIPT cache (like L2)

SMT Commit

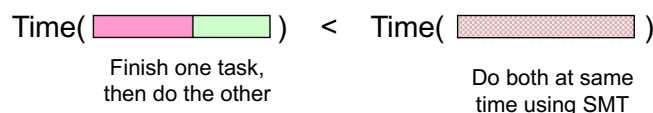
- Register File Management
 - ARF/PRF organization
 - need one ARF per thread
- Need to maintain interrupts, exceptions, faults on a per-thread basis
 - like OOO needs to appear to outside world that it is in-order, SMT needs to appear as if it is actually N CPUs

SMT Performance

- When it works, it fills idle “issue slots” with work from other threads; throughput improves

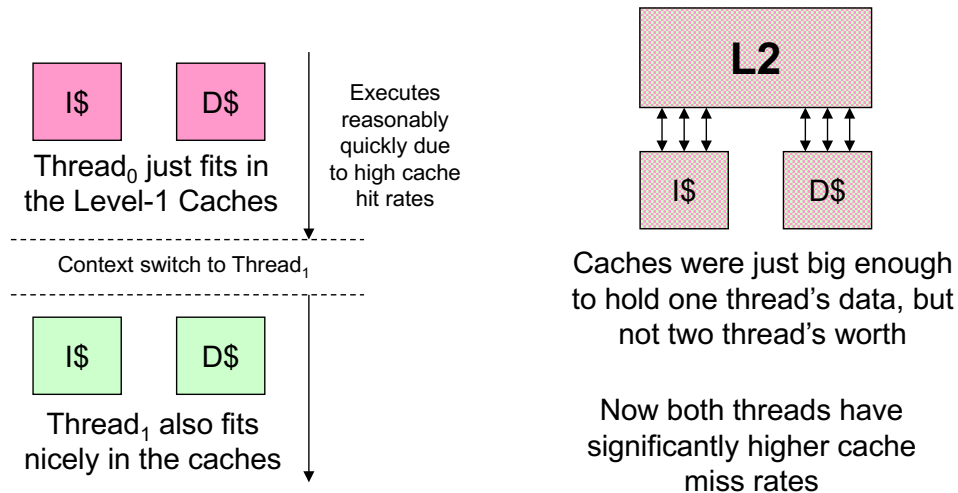


- But sometimes it can cause performance degradation!



How?

- Cache thrashing



This is all combinable

- Can have a system that supports SMP, CMP and SMT at the same time
 - Take a dual-socket SMP motherboard...
 - Insert two chips, each with a dual-core CMP...
 - Where each core supports two-way SMT
- This example provides 8 threads worth of execution, shared on 4 actual “cores”, split across two physical packages

OS Confusion

- SMT/CMP is supposed to look like multiple CPUs to the software/OS

