Processor Cores

ECE/CS 757 Spring 2007
J. E. Smith

Outline
- Conventional Cores (Review)
  - In-order
  - Out-of-order Superscalar
- Historical Multi-threaded Cores
  - 6600, HEP
- In-Order Multi-threaded Cores
- Out-of-Order Multi-threaded Cores
  - Resource Sharing
  - Thread Scheduling
- Case Studies
  - IBM RS64 IV
  - SUN Niagara
  - IBM Power5

In-Order Pipelines
- As used in early RISCs
- Re-merging in throughput-oriented servers
- Complexity in stall/forwarding logic

Superscalar Processors
- Widely used today in high performance microprocessors

Stall/Forward Example

Superscalar Processors
- Issue instructions out-of-order when dependences are resolved
- Reorder buffer keeps track of architected order
Instruction fetch and branch prediction converts static program into dynamic instruction stream

Static Program:
loop: r3 <- mem(r4+r2)
r7 <- mem(r5+r2)
r7 <- r7 * r3
r1 <- r1 - 1
mem(r6+r2)<- r7
r2 <- r2 + 8
P <- loop; r1!=0

Dynamic Stream:
(1) r3 <- mem(r4+r2)
r7 <- mem(r5+r2)
r7 <- r7 * r3
r1 <- r1 - 1
mem(r6+r2)<- r7
r2 <- r2 + 8
P <- loop; r1!=0

Register Renaming
- More physical registers than logical registers
- Physical register names act as tags
  - Mapping changes over time
- Turns instruction stream into "single assignment" form
  - avoids WAW and WAR hazards
- For each instruction
  - Read source register mappings from table
  - Acquire new result physical register from free pool
  - Modify map table for new result register mapping
- Example
  - Assume 8 logical, 24 physical registers
  (to simplify free pool management in example fewer physical regs. can be used in practice)
Example
Renamed Stream
p8 <- mem(p1+p4)
p9 <- mem(p2+p4)
p10 <- p9 * p8
p11 <- p3 - 1
mem(p7+p4) <- p10
p12 <- p4 + 8
P <- loop; p11 != 0
p13 <- mem(p1+p12)
p14 <- mem(p2+p12)
p15 <- p14 * p13
p16 <- p11 - 1
mem(p7+p12) <- p15
p17 <- p12 + 8
P <- loop; p16 != 0
p18 <- mem(p1+p17)
p19 <- mem(p2+p17)
p20 <- p19 * p18
p21 <- p16 - 1
mem(p7+p17) <- p20
p22 <- p17 + 8
track reg. 7:

Example
Renamed Stream
p8 <- mem(p1+p4)
p9 <- mem(p2+p4)
p10 <- p9 * p8
p11 <- p3 - 1
mem(p7+p4) <- p10
p12 <- p4 + 8
P <- loop; p11 != 0
p13 <- mem(p1+p12)
p14 <- mem(p2+p12)
p15 <- p14 * p13
p16 <- p11 - 1
mem(p7+p12) <- p15
p17 <- p12 + 8
P <- loop; p16 != 0
p18 <- mem(p1+p17)
p19 <- mem(p2+p17)
p20 <- p19 * p18
p21 <- p16 - 1
mem(p7+p17) <- p20
p22 <- p17 + 8
track reg. 7:
Example

Renamed Stream
p8 <- mem(p1+p4)
p9 <- mem(p2+p4)
p10 <- p9 * p8
p11 <- p3 - 1
mem(p7+p4) <- p10
p12 <- p4 + 8
P <- loop; p11 != 0
p13 <- mem(p1+p12)
p14 <- mem(p2+p12)
p15 <- p14 * p13
p16 <- p11 - 1
mem(p7+p12) <- p15
p17 <- p12 + 8
P <- loop; p16 != 0
p18 <- mem(p1+p17)
p19 <- mem(p2+p17)
p20 <- p19 * p18
p21 <- p16 - 1
mem(p7+p17) <- p20
p22 <- p17 + 8

Bottlenecks

- Initially
  - 7 instructions per 3 clocks => 2.3 IPC
  - Bottleneck is cache ports (1)
- Add cache port
  - 7 instructions per 2 cycles => 3.5 IPC
  - Bottleneck is I-fetch
- Increase fetch width (F) to 8
  - 4 instructions per cycle
  - (But I didn’t check this completely)

Exceptions and Commit

- Exceptions are reported to ROB entry as they are detected
- When an instruction completes execution, this is marked in ROB
- Instructions are removed from ROB in order only after complete

Example

Renamed Stream dispatch issue complete
p8 <- mem(p1+p4)  0     2      5
p9 <- mem(p2+p4)  0     0      4
p11 <- p3 - 1     0     1      3
mem(p7+p4) <- p10  1     3      10
p12 <- p4 + 8     1     2      3
P <- loop; p11 != 0 1     3      10
p13 <- mem(p1+p12) 2     4      7
p14 <- mem(p2+p12) 2     5      8
p15 <- p14 * p13   2     6      10
p16 <- p11 - 1    2     3      4
mem(p7+p12) <- p15 3     4      8
p17 <- p12 + 8    3     5      8
P <- loop; p16 != 0 3     4      8
p18 <- mem(p1+p17) 4     7      10
p19 <- mem(p2+p17) 4     8      12
p20 <- p19 * p18  4     11     14
p21 <- p16 - 1   4     5      6
mem(p7+p17) <- p20 5     9      12
p22 <- p17 + 8  5     6      7

(C) 2007 J. E. Smith

Performance:
1 loop iteration per 3 cycles
**Example**

- Final load in previous example page faults
  - at PC=5C
- Use ROB to
  - Restore register map to state just prior to 5C
  - Restore PC to 5C

**Load/Store Buffering**

- Important component with respect to Multiprocessor Design
  - For both memory ordering problems and solutions
- Problem: memory address dependences not resolvable until after issue
  - Addresses must first be computed
- Performance advantages if:
  - Loads can pass stores
  - Pending stores forward data to loads

**General Load/Store Implementation**

- Store assigned Store Queue entry at dispatch
- Loads and Stores issue in order
- Stores issue when address registers ready
  - Data forwarded later (Store completes)
  - Stores commit when at head of ROB
- Load addresses compared with pending stores
  - Match when store data available => forward
  - Match when data unavailable => wait for data, then forward
- Coalescing store buffers reduce cache bandwidth demand

**Forwarding: More Detail**

- Relationships between bandwidths and buffer sizes for balanced design
- Most bandwidths linear with respect to issue width
- Most storage quadratic (or more) with respect to issue width

**More aggressive implementation**

- Issue loads and stores out of order
  - Before addresses are known
- Additional buffering/checking
- Detects address match when store addresses are known
  - Save loads in completed load buffer
- On address match, flush and restart using ROB mechanism
- Add-on optimizations (not shown) predict conflicts and defer load issue

**Balanced Superscalar Processors**

- Relationships between bandwidths and buffer sizes for balanced design
  - Most bandwidths linear with respect to issue width
  - Most storage quadratic (or more) with respect to issue width
Balance Relationships
- Empirically derived relationship between Issue width (I) and Window size (W) – based on critical dependence paths
  \[ W = \frac{1}{3} \leq I \leq W \frac{1}{2} \]
- Issue buffer size \( \approx \frac{W}{3} \)

Processor | Issue Buffer Size | Issue Buffer Size | size Width | log(Issue Width) | log(Issue Buffer Size) |
--- | --- | --- | --- | --- | --- |
Intel Core | 64 | 72 | 6 | 3.0 | 5 |
IBM Power | 384 | 36 | 5 | 3.3 | 4 |
SAP R/3 | 312 | 30 | 5 | 3.0 | 4 |
Alpha 21264 | 72 | 24 | 4 | 3.1 | 3 |
HP PA-8000 | 128 | 64 | 4 | 3.9 | 4 |
AMD Opteron | 128 | 64 | 4 | 3.9 | 4 |

Multithreaded Processors
- Motivation: many resources (including internal processor resources) are used intermittently
- Let threads (or processes) share a single hardware processor
- Replicate hardware that holds architected state
  - program counter, logical registers
- Share or partition implementation hardware
  - functional units, caches, physical registers, pipeline hardware
  - implementation hardware may be expanded/enhanced
- Multithreading is implementation, not architecture

Historical Multithreaded Processors
- CDC6600 PPs
  - I/O processing
- Denelcor HEP
  - General purpose scientific

CDC 6600 Peripheral Processors
- Intended to perform OS and I/O functions
- Used “barrel and slot”
  - register state is arranged around a “barrel”
  - one set of ALU and memory hardware accessed through “slot” in barrel
  - slot in barrel rotates one position each cycle
- Could be used as stand-alone “MP”
- Similar method later used in IBM Channels

CDC 6600 Peripheral Processors
- General purpose scientific computer
- Organized as an MP
  - Up to 16 processors
  - Each processor is multithreaded
  - Up to 128 memory modules
  - Up to 4 I/O cache modules
  - Three-input switches and chaotic routing

Denelcor HEP
**Processor Organization**

- Multiple contexts (threads) are supported:
  - 120 threads
    - Each with a PSW (program status word)
- PSWs circulate in a control loop
  - Control and data loops pipelined 8 deep
  - PSW in control loop can circulate no faster than data in data loop
  - PSW at queue head fetches and starts execution of next instruction
- No inter-instruction pipeline forwarding or stalls needed
- Clock period: 100 ns
  - 8 PSWs in control loop => 10 MIPS
  - Maximum perf. per thread => 1.25 MIPS
    (They tried to sell this as a supercomputer)

**Switch**

- Packet switched
  - Every cycle, take in 3 packets, send out 3 packets
- "Hot Potato" routing
  - Form of adaptive routing
  - Do not enqueue on a port conflict
    - Send anyway on another port and raise priority
  - At top priority (15) traverse a circuit through the net

**Modern Day Multi-Threading**

- Apply to superscalar pipelines
  - More resources to share
- Also one-wide in-order processors
  - Provide high efficiency for throughput-oriented servers
- Start with Case Study
  - Intel Pentium 4 Hyperthreading

**Intel Hyperthreading**

- Part of Pentium 4 design (Xeon)
- Two threads per processor
- Goals
  - Low cost – less than 5% overhead for replicated state
  - Assure forward progress of both threads
    - Make sure both threads get some buffer resources through partitioning or budgeting
  - Single thread running alone does not suffer slowdown
Intel Hyperthreading

- Main pipeline
  - Pipeline prior to trace cache not shown (see paper)
- Round-Robin instruction fetching
  - Alternates between threads
  - Avoids dual-ported trace cache
  - BUT trace cache is a shared resource

Trace Caches

- Trace cache captures dynamic traces
- Increases fetch bandwidth
- Help shorten pipeline (if predecoded)

Capacity Resource Sharing

- Append thread identifier (TId) to threads in shared capacity (storage) resource
- Example: cache memory

Frontend Implementation

- Partitioned front-end resources
  - Fetch queue (holds uops)
  - Rename and allocate tables
  - Post-rename queues
- Partitioning assures forward progress if other thread is blocked
  - Round-robin scheduling

Backend Implementation

- Physical registers are pooled (shared)
- Five instruction buffers (schedulers)
  - Shared
  - With an upper limit
- Instruction issue is irrespective of thread ID
- Instruction commit is round-robin
  - From partitioned ROB

Operating Modes and OS Support

- MT-mode – two active logical processors; shared/partitioned resources
- ST-mode (ST0 or ST1) – one logical processor; combined resources
- HALT – privileged instruction => (normally) low power mode
  - In MT mode => transition to ST0 or ST1
  - In ST mode => low power mode
- Interrupt to HALTed thread => transition to MT mode
- OS manages two “processors” (some licensing issues)
  - OS code should HALT rather than idle loop
  - Schedule threads with priority to ST mode
    - (require OS knowledge of hyperthreading)
Performance

- OLTP workload
  - 21% gain in single and dual systems
  - Must be some external bottleneck in 4 processor systems

Intel Hyperthreading Summary

- Mix of partitioned and shared resources
- Mostly round-robin scheduling
- Secondary objective: fairness
- Not a lot of obvious structure/reasons for design decisions

Objectives

- Optimize aggregate performance
  - Possibility at the expense of individual thread performance
- Provide assured performance for one (or more) threads
  - Soft real-time
  - Fast interactive response time is special-case
- Fairness
  - Often not well-defined (e.g., Intel Pentium)
  - Often means the lack of obvious unfairness
  - Some quantitative measures have been proposed (next slide)
- Isolation
  - Don’t let threads interfere with each other
  - Provides dependable performance
  - Example: shared server

Multi-threaded Processors

- Contains both capacity (storage) and bandwidth (processing) resources
- These must be managed in some fashion
  - Can’t be done by OS (as with conventional shared hardware resources)
  - Implementation dependent features

- Objectives
  - Big picture goals
- Policies
  - An algorithmic plan of action
- Mechanisms
  - Implement the policies

Fairness

- Often a secondary goal
- Often defined in terms of performance
  - “Fair” if all threads experience similar slowdowns
  - Compared with running alone

- Harmonic mean speedup
  \[ S = \frac{n}{\sum 1/(S_i)} \]
  - Captures aggregate performance
  - As well as a measure of fairness
  - A slow thread drags down the harmonic mean more than a fast thread lifts it

- Proportionate slowdown
  - Primary fairness goal
  - Make all the speedups (slowdowns) the same
- Other definitions focus on resources
  - Give each thread a fair share of resources
  - May be equal share (in server example)
Policies and Mechanisms

- Separate primitives (mechanisms) from solutions (policies)
  - Generally good computer engineering
  - Allows flexibility in policies (during and after design)
- Example
  - Mechanism: Program counter multiplexer in l1Fetch stage
  - Policy: mux control – round-robin (or priorities)

Mechanisms

- Capacity resources
  - Caches – write once, read many times
  - Buffers – write once, read once
  - Can be partitioned or shared (pooled)
- Bandwidth resources
  - Used for processing
    - E.g., a pipeline stage
  - If not used at a given cycle, then wasted for that cycle
  - Can be partitioned or shared (in time)

Example: Hyperthreading

- Mechanisms (and Policies) in Pentium 4

Other Mechanism Considerations

- Pre-emption
  - If a thread is using a resource, can it be pre-empted by another (higher) priority thread?
  - E.g., cache eviction or pipeline flush
  - Requires a mechanism for pre-emption – policy decides if/when to do it
- Feedback
  - Mechanisms can also monitor resource usage and provide feedback for policies

Policies

- Coordination – policies are usually implemented by local hardware
  - But follow some overall “global” policy
  - This doesn’t imply global hardware
  - Upstream policy (e.g., at l1-fetch) usually drives downstream policies

Policies: Feedback

- Part of Coordination – policies use feedback from downstream
Scheduling Policies
- Applies to allocation of bandwidth resources
  - e.g., pipeline stages
- Granularity
  - When to schedule
- Selection
  - Which to schedule

Scheduling Granularity
- Coarse-grain
  - Re-schedule after multiple clock cycles
  - "Switch-on-event"
- Fine-grain
  - Schedule every cycle
  - On thread gets all of resource for that cycle
- Simultaneous
  - Schedule every cycle
  - Resource is shared during that cycle
- Pentium 4 uses combination of fine-grain and simultaneous

Thread Selection
- Round-Robin
  - Rotate among threads
- LRU
  - Like round-robin, but if thread has been stalled and re-enters selection process, it goes to front of the line
- FCFS (FIFO)
  - First come – first served – typically applies to a queue
- FR-FCFS
  - First ready FCFS – of those that are ready, the first to arrive gets priority
  - Example: issue buffer
- Priority
  - Top priority thread gets resource
  - May be set by software
  - Maybe established through feedback mechanism in hardware

Capacity Policies
- How big should partitions be?
  - If partitions are flexible
- How much of a shared resource should each thread get?
- Many interesting issues arise with caches
  - To be covered later

Sharing Backend Capacity Resources
- Issue Buffer, ROB, Load/Store queues, Physical Registers
  - Grow quadratically with issue width
- Many designs provide resources for high single thread performance
  - To permit single thread mode
  - For good performance when there is only one thread working
  - Implies excess resources when multiple threads

Sharing Backend Capacity Resources
- Divide quadratic-growth resource evenly among n threads
  - pipeline width d
- Total resource requirement: \( n^2(d/n)^2 = d^2/n \)
- More threads \( \Rightarrow \) less total resource
- Partitioning may not hurt performance (much)
**Work Conservation**

- If any thread can use a resource, will it be used?
  - If so, then work conserving
- Work conservation is easier with bandwidth resources than capacity (cache) resources
  - With a cache, it is hard to determine if it is being “used”
- Partitioning leads to non-work conservation
- Work conservation may be paired with pre-emption

**Example: Pentium 4**

- Combination of Policies and Mechanisms

```
<table>
<thead>
<tr>
<th>Instruction Fetch</th>
<th>Instruction Dispatch</th>
<th>Instruction Issue</th>
<th>Read Registers</th>
<th>Execute</th>
<th>Memory Access</th>
<th>Write-Back</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Thread Scheduling: In-Order Processors**

- I-Fetch scheduling drives entire pipeline
- Many small stalls due to register dependences
- Fine granularity scheduling “meshes” the stalls from multiple threads

**Thread Scheduling: Out-Of-Order Processors**

- Miss events tend to dominate
- Out-of-order issue reduces fine-grain stalls
  - In balanced processors
  - Switch-on-event appears to be good policy
  - Problems with switch delay
  - For branch misprediction, new thread ready no sooner than old thread
  - Ditto for short (L1) D-cache miss (miss latency = pipeline fill latency)
  - May be useful only for long latency miss events

**Thread Scheduling: In-Order Processors**

- Coarse granularity can also be successful if miss-events are dominant
  - Switch-on-event policies; “event” == cache miss

```
<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Fine-grain MT</th>
<th>Coarse-grain MT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

---

(c) 2007 J. E. Smith
O-O-O Scheduling: Fetch Unit

- Fetch unit mechanisms
  - Easier if time partitioned
  - Else dual-ported cache
- Easier for narrow fetch widths
  - Due to discontinuities caused by branches
- Wide issue may require dual-ported cache, anyway
  - Or trace cache
- Most current processors are narrow enough that partitioning works
  - Alpha EV-8 (abandoned), 8-wide => two fetches per cycle

Fetch Unit Policies

- Fine-grain
- Selection establishes policy for entire front-end (up to issue)
- Round-robin with feedback is common (later)
- Also can implement thread priorities

O-O-O Scheduling: Issue Unit

- Mechanism allows sharing (no hard partitioning)
  - Back-end capacity resources are expensive
  - May want one thread to have all of resource
  - May have soft partitioning
- Policy is “thread-blind”
  - Otherwise would be more complicated
  - Typically FR-FCFS
- Combination of sharing and thread-blindness could lead to congestion/starvation
  - A thread with a long cache miss could occupy issue buffer slots and slow other thread(s)
- One Solution: Pre-emption on long cache miss
- Another Solution: Feedback to fetch policy

O-O-O Scheduling: Feedback

- I-Fetch Policy
  - Count in-flight, un-issued instructions for each thread
    - Give priority to thread with lowest count
    - => ICOUNT policy

Another possibility

- From Raasch et al.
  - Use soft partition of back-end resources
    - Soft partition allows single thread mode
    - Partition doesn’t hurt performance much due to quadratic growth
    - Remember:
      - Then use round-robin without feedback (works pretty well)
Fairness Policies

- Share the pain (proportionate slowdown)
  - Requires feedback mechanism for estimating "run alone" performance
  - In general, this is not easy
  - Can be done via off-line profiling => coarse granularity policy

- Share the resources (proportionate resources)
  - Good for real-time objectives
  - Good for performance isolation
  - Work conservation becomes an issue
  - Leads to "Virtual Private Machines" (later)

Case Study: IBM RS64 IV

- Two threads per processor
  - 4-way superscalar, in-order processor
  - Foreground thread executes until there is a long latency event (e.g., cache miss)
  - Then switch to the background thread
  - Priorities adjust miss events that cause switches
  - Thread Switch Timeout Register assures fairness when there are few miss event context switches

- Thread state control register (TSC)
  - Software-writeable, sets priorities
  - Priorities determine events that cause switch

- Forward Progress Count
  - Counts number of times a thread was switched in, but made no progress
  - Threshold in TSC
  - If count exceeds threshold, then force forward progress

Fast Thread Switches

- An issue with switch-on-event scheduling
- Add thread switch buffer
  - Use extra latch capacity to fill thread switch buffer with insts. from background thread
  - Also a branch target buffer for quick mispredict recovery
- Leads to three-cycle switch time
  - For D-cache, TLB misses
  - Cache miss not detected until WB stage

Miss Performance

- Multithreading can affect miss rates
  - Long miss (L2) degrades little
  - I-cache misses degrade little
  - In server workload there are a lot of thread switches, anyway
  - I-cache misses already high

<table>
<thead>
<tr>
<th>Directory</th>
<th>Increase in miss rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2 cache</td>
<td>4.6</td>
</tr>
<tr>
<td>L1 I cache</td>
<td>8.5</td>
</tr>
<tr>
<td>L1 D cache</td>
<td>32.0</td>
</tr>
<tr>
<td>TLB</td>
<td>19.0</td>
</tr>
</tbody>
</table>

Thread Switch Causes

- Most thread switches due to L1 misses, then timeouts
Case Study: SUN Niagara

- Used in Sun Servers
  - SPARC ISA
  - Commercial apps, abundant threads
  - => one-wide in-order pipeline (for simplicity, efficiency)
  - Uses fine-grain thread switching
- Each chip contains:
  - Eight cores, four threads each
  - 16K I-cache, 8K D-cache per core
  - Shared floating point unit
  - Shared 3 MByte L2 cache
  - Memory controllers

SUN Niagara

- In-order Core
  - Supports four threads
  - Forwarding logic
  - Load has two cycle latency
  - Separate Load/Store buffers per thread
  - Reduces cross-checks, forwarding

Pipeline Processing

- Coordinated policy at fetch and decode
- Fetch one or two instructions with LRU policy
  - Dual fetch => unused I-cache slots, for line-fills
- Pre-decode identifies long latency instructions
  - Multiplies, divides, branches, loads
- Thread with long latency instruction is temporarily de-selected
  - For multiplies/divides – until operation is complete
  - For branches – until outcome is known
  - (then no need for prediction)
  - For loads – more complex...

Load Instructions

- Deselect at least until time data would be available via forward path
- Lower priority until time hit/miss is known
  - Happens one cycle later than first forward possibility
  - Miss not detected until WB stage
- Example:
  - cycles: 0 1 2 3 4 5 6 7
  - load, add
  - load, add
  - ID ID ID ID ME ME ME WB
  - ID ID ID ID ME WB WB
  - TS* TS* TS* TS* EX EX EX
  - TS TS TS TS EX EX WB

Case Study: IBM Power5

- Used in IBM Servers
  - PowerPC ISA
  - High-end out-of-order superscalar processor
  - Uses simultaneous multi-threading
- Each chip contains:
  - Two cores, two threads each
  - 64K I-cache, 32K D-cache per core
  - Shared 1.875 MByte L2 cache
  - Tag directory for external L3 cache
  - Memory controllers
  - System interconnect

Power5 Policies and Mechanisms

- I-Fetch
  - Round-robin into partitioned 24-entry fetch buffers
- Dispatch Selection
  - Done via priorities and feedback mechanisms (next slide)
- ROB shared via linked list structure
  - Manages groups of instructions (up to 5) for simpler design
Dispatch Policy

- Primary point for resource management
- Uses software-settable priorities
  - Set via software writeable control register
  - Application software can control some of the priority settings
  - Priority 0 => idle, priority 1 => spinning
  - Both threads at level 1 => throttle to save power
- Hardware feedback can adjust priorities

Feedback Priority Adjustment

- Dynamic Resource Balancing
  - Temporarily adjusts priority levels
- GCT (ROB) Occupancy
  - Thread with more than threshold entries gets reduced priority
  - Similar to ICOUNT
- LMQ (load miss queue) Occupancy
  - Thread more than threshold entries gets reduced priority
  - Many outstanding misses => resource blockage

Priority Behavior

- Illustration of typical behavior
- Eight priority levels
- Highest aggregate throughput with equal priorities
- Highest single thread performance at priority 7,0

Spin Lock Optimization

- Spinning on a lock should not steal resources from other threads
- Typical synchronization code around critical sections (TTAS):
  - spin: ld r0,lock (MI)(HitS)(MI)
  - cmpi c0,0,0
  - ldarx r0,lock (HitS)
  - cmpi c0,0,0
  - stdcx r1,lock (DClaim)
  - ...critical section...
  - sync
  - std r0,lock (DClaim)
- Note miss and modified intervention (MI) that precedes falling out of spin loop
- Could thread switch on failure to acquire lock (test-and-test-and-set is not very amenable to that)

SMT “Ugly” Instructions

- Ugly, context-synchronizing instructions
  - System call/return (switch to/from privileged state)
  - Writes to lower-level status registers (translation on/off, etc.)
  - TLB invalidates
  - Cache line invalidates
  - Memory barriers (weak consistency)
- Single-threaded CPU:
  - Stall issue, wait for completion, continue
- SMT:
  - This may not be necessary for functional correctness or acceptable for performance
- Need to consider these case by case
- Lots of headaches, potential corner cases

Alpha 21464 Sleep Instruction

- Alpha 21464 proposed clever new sleep instruction that puts thread to sleep until address changes
- Here’s the new spin lock code:
  - ld r0,lock # test lock
  - cmpi c0,0,0,0
  - test: ldarx r0,lock # do atomic test-and-set
  - cmpi c0,0,0,0
  - stdcx r1,lock # go to spin loop
  - ...critical section...
  - sync
  - std r0,lock # release lock
SMT Consistency Model Issues

- Consistency model restricts when other processors can see your write
  - What does "other processor" mean in an MT processor?
  - Descriptions of consistency models may need to be updated...
  - Probably "other thread" is sufficient

- Weak consistency (e.g. PowerPC)
  - Stores visible anytime, so store forwarding or reading stored lines from cache early is OK
  - Memory barriers (sync) cause problems: store queue flush

- Sequential consistency
  - Stores must be atomically visible to all other processors (write atomicity)
  - Cannot forward store values until preceding stores are done
  - Need logically separate store queues (or thread IDs in store queue entries)
  - Cannot retire stores to L1 until coherence activity is done

- Other issues? Certainly there is no exhaustive treatment in the literature.