Chapter 11: Executing Multiple Threads
Modern Processor Design: Fundamentals of Superscalar Processors

Executing Multiple Threads

- Thread-level parallelism
- Synchronization
- Multiprocessors
- Explicit multithreading
- Implicit multithreading
- Redundant multithreading
- Summary

Thread-level Parallelism

- Instruction-level parallelism
  - Reaps performance by finding independent work in a single thread
- Thread-level parallelism
  - Reaps performance by finding independent work across multiple threads
- Historically, requires explicitly parallel workloads
  - Originate from mainframe time-sharing workloads
  - Even then, CPU speed >> I/O speed
  - Had to overlap I/O latency with “something else” for the CPU to do
  - Hence, operating system would schedule other tasks/processes/threads that were “time-sharing” the CPU

Thread-level Parallelism

Increases in number of active threads reduces effectiveness of spatial locality by increasing working set.

Time dilation of each thread reduces effectiveness of temporal locality.

- Reduces effectiveness of temporal and spatial locality
Thread-level Parallelism
- Initially motivated by time-sharing of single CPU
  - OS, applications written to be multithreaded
- Quickly led to adoption of multiple CPUs in a single system
  - Enabled scalable product line from entry-level single-CPU systems to high-end multiple-CPU systems
  - Same applications, OS, run seamlessly
- Adding CPUs increases throughput (performance)
- More recently:
  - Multiple threads per processor core
    - Coarse-grained multithreading (aka “switch-on-event”)
    - Fine-grained multithreading
    - Simultaneous multithreading
  - Multiple processor cores per die
    - Chip multiprocessors (CMP)

Thread-level Parallelism
- Parallelism limited by sharing
  - Amdahl’s law:
    - Access to shared state must be serialized
    - Serial portion limits parallel speedup
  - Many important applications share (lots of) state
  - Even completely independent processes “share” virtualized hardware, hence must synchronize access
- Access to shared state/shared variables
  - Must occur in a predictable, repeatable manner
  - Otherwise, chaos results
- Architecture must provide primitives for serializing access to shared state

Some Synchronization Primitives

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Semantic</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch-and-add</td>
<td>Atomic load-&gt;add-&gt;store</td>
<td>Permits atomic increment, can be used to synthesize locks for mutual exclusion</td>
</tr>
<tr>
<td>Compare-and-swap</td>
<td>Atomic load-&gt;compare-&gt;conditional store</td>
<td>Stores only if load returns an expected value</td>
</tr>
<tr>
<td>Load-linked/store-conditional</td>
<td>Atomic load-&gt;conditional store</td>
<td>Stores only if load/store pair is atomic; that is, there is no intervening store</td>
</tr>
</tbody>
</table>

- Only one is necessary
  - Itanium provides them all!
Synchronization Examples

Thread 0
fetchadd A, 1
spin:
cmpswp AL, 1
bfail spin
load rl, A
addi rl, rl, 1
store rl, A
store 0, AL
Thread 1
fetchadd A, 3
spin:
cmpswp AL, 1
bfail spin
load rl, A
addi rl, rl, 3
store rl, A
store 0, AL

(b) (c)

Thread 0
spin:
ll r1, A
addi r1, r1, 1
stc r1, A
bfail spin
Thread 1
spin:
ll r1, A
addi r1, r1, 3
stc r1, A
bfail spin

(a)

All three provide guarantee same semantic:
- Initial value of A: 0
- Final value of A: 4

b uses additional lock variable AL to protect critical section with a spin lock
- This is the most common synchronization method in modern multithreaded applications

Multiprocessor Systems

- Focus here is only on shared-memory symmetric multiprocessors
  - Many other types of parallel processor systems have been proposed and built
  - Key attributes are:
    - Shared memory: all physical memory is accessible to all CPUs
    - Symmetric processors: all CPUs are alike
    - Other parallel processors may:
      - Share some memory, share disks, share nothing
      - Have asymmetric processing units

- Shared memory idealisms
  - Fully shared memory
  - Unit latency
  - Lack of contention
  - Instantaneous propagation of writes

UMA vs. NUMA

Uniform Memory Access (shared)

Nonuniform Memory Access (mosaic)

Update vs. Invalidation Protocols

- Coherent Shared Memory
  - All processors see the effects of others’ writes
- How/when writes are propagated
  - Determine by coherence protocol
### Sample Invalidate Protocol (MESI)

<table>
<thead>
<tr>
<th>Current State</th>
<th>Local Read (LR)</th>
<th>Local Write (LW)</th>
<th>Local Eviction (EV)</th>
<th>Bus Read (BR)</th>
<th>Bus Write (BW)</th>
<th>Bus Upgrade (BU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid (I)</td>
<td>Issue bus read if no sharers then $s' = E$ or $s' = S$</td>
<td>Do nothing</td>
<td>Do nothing</td>
<td>Do nothing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shared (S)</td>
<td>Do nothing</td>
<td>Issue bus upgrade $s' = M$</td>
<td>Respond shared $s' = S$</td>
<td>$s' = I$</td>
<td>$s' = I$</td>
<td></td>
</tr>
<tr>
<td>Exclusive (E)</td>
<td>Do nothing</td>
<td>$s' = M$</td>
<td>Respond shared $s' = S$</td>
<td>$s' = I$</td>
<td>Error</td>
<td></td>
</tr>
<tr>
<td>Modified (M)</td>
<td>Do nothing</td>
<td>Do nothing</td>
<td>Write data back $s' = I$</td>
<td>Respond dirty; Write data back $s' = S$</td>
<td>Error</td>
<td></td>
</tr>
</tbody>
</table>

---

**Implementing Cache Coherence**

- **Snooping implementation**
  - Origins in shared-bus systems
  - All CPUs could observe all other CPUs requests on the bus; hence “snooping”
    - Bus Read, Bus Write, Bus Upgrade
  - React appropriately to snooped commands
    - Invalidate shared copies
    - Provide up-to-date copies of dirty lines
      - Flush (writeback) to memory, or
      - Direct intervention (modified intervention or dirty miss)

- **Snooping suffers from:**
  - Scalability: shared buses not practical
  - Ordering of requests without a shared bus
  - Lots of recent and on-going work on scaling snoop-based systems

---

**Directory implementation**

- Extra bits stored in memory (directory) record state of line
- Memory controller maintains coherence based on the current state
- Other CPUs’ commands are not snooped, instead:
  - Directory forwards relevant commands
  - Powerful filtering effect: only observe commands that you need to observe
- Meanwhile, bandwidth at directory scales by adding memory controllers as you increase size of the system
  - Leads to very scalable designs (10s to 1000s of CPUs)

**Directory shortcomings**

- Indirection through directory has latency penalty
  - If shared line is dirty in other CPU’s cache, directory must forward request, adding latency
  - This can severely impact performance of applications with heavy sharing (e.g. relational databases)
Memory Consistency

- How are memory references from different processors interleaved?
- If this is not well-specified, synchronization becomes difficult or even impossible
  - ISA must specify consistency model
  - Common example using Dekker’s algorithm for synchronization
    - If load reordered ahead of store (as we assume for a baseline OOO CPU)
    - Both Proc0 and Proc1 enter critical section, since both observe that other’s lock variable (A/B) is not set
  - If consistency model allows loads to execute ahead of stores, Dekker’s algorithm no longer works
    - Common ISAs allow this: IA-32, PowerPC, SPARC, Alpha

Sequential Consistency [Lamport 1979]

- Processors treated as if they are interleaved processes on a single time-shared CPU
- All references must fit into a total global order or interleaving that does not violate any CPUs program order
  - Otherwise sequential consistency not maintained
- Now Dekker’s algorithm will work
- Appears to preclude any OOO memory references
  - Hence precludes any real benefit from OOO CPUs

High-Performance Sequential Consistency

- Coherent caches isolate CPUs if no sharing is occurring
  - Absence of coherence activity means CPU is free to reorder references
- Still have to order references with respect to misses and other coherence activity (snoops)
- Key: use speculation
  - Reorder references speculatively
  - Track which addresses were touched speculatively
  - Force replay (in order execution) of such references that collide with coherence activity (snoops)
**Relaxed Consistency Models**

- Key insight: only synchronization references need to be ordered
- Hence, relax memory for all references
  - Enable high-performance OOO implementation
- Require programmer to label synchronization references
  - Hardware must carefully order these labeled references
  - All other references can be performed out of order
- Labeling schemes:
  - Explicit synchronization ops (acquire/release)
  - Memory fence or memory barrier ops:
    - All preceding ops must finish before following ones begin
- Often: fence ops cause pipeline drain in modern OOO machine

**Coherent Memory Interface**

**Explicitly Multithreaded Processors**

<table>
<thead>
<tr>
<th>MT Approach</th>
<th>Resources shared between threads</th>
<th>Control Switch Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Everything</td>
<td>Explicit operating system context switch</td>
</tr>
<tr>
<td>Fine-grained</td>
<td>Everything but register file and control logic/state</td>
<td>Switch every cycle</td>
</tr>
<tr>
<td>Course-grained</td>
<td>Everything but instruction buffers, register file and control logic/state</td>
<td>Switch on pipeline stall</td>
</tr>
<tr>
<td>SMT</td>
<td>Everything but instruction buffers, register file and control logic/state, reorder buffer, store queue, etc.</td>
<td>All contexts concurrently active; no switching</td>
</tr>
<tr>
<td>CMP</td>
<td>Secondary cache, system interconnect</td>
<td>All contexts concurrently active; no switching</td>
</tr>
</tbody>
</table>

- Many approaches for executing multiple threads on a single die
  - Mix-and-match: IBM Power5 CMP+SMT
IBM Power4: Example CMP

- Crossbar interconnect
- Power4 core 0: L1 I$, L1 D$
- Power4 core 1: L1 I$, L1 D$
- Coherent I/O interface
- P0 STQ, P1 STQ
- L2 tags & I/F
- L2 data
- ~512KB L2 slice
- Address/response data interconnect

Coarse-grained Multithreading

- Low-overhead approach for improving processor throughput
  - Also known as "switch-on-event"
- Long history: Denelcor HEP
- Commercialized in IBM Northstar, Pulsar
- Rumored in Sun Rock, Niagara

SMT Resource Sharing

- Decode
- Rename
- Fetch
- Issue
- Retire

Implicitly Multithreaded Processors

- Goal: speed up execution of a single thread
- Implicitly break program up into multiple smaller threads, execute them in parallel
- Parallelize loop iterations across multiple processing units
- Usually, exploit control independence in some fashion
- Many challenges:
  - Maintain data dependences (RAW, WAR, WAW) for registers
  - Maintain precise state for exception handling
  - Maintain memory dependences (RAW/WAR/WAW)
  - Maintain memory consistency model
- Active area of research
  - Only a subset is covered here, in a superficial manner
Sources of Control Independence

- (a) Loop-closing
- (b) Control-flow convergence
- (c) Call/return

Implicit Multithreading Proposals

<table>
<thead>
<tr>
<th>Control Flow Attributes Exploited</th>
<th>Multiscalar</th>
<th>H/D/E/E</th>
<th>Dynamic Multi-threading (DNT)</th>
<th>Thread-level Speculation (TLS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop bodies</td>
<td>Control Independence</td>
<td>Control Independence</td>
<td>Control independence</td>
<td>Control independence</td>
</tr>
<tr>
<td>Control-flow joins</td>
<td>Control Independence</td>
<td>Control Independence</td>
<td>Control independence</td>
<td>Control independence</td>
</tr>
<tr>
<td>Call/return</td>
<td>Loop bodies</td>
<td>Control-flow joins</td>
<td>Control-flow joins</td>
<td>Control-flow joins</td>
</tr>
<tr>
<td></td>
<td>Control bodies</td>
<td>Control-flow joins</td>
<td>Control-flow joins</td>
<td>Control-flow joins</td>
</tr>
<tr>
<td>Loop exits</td>
<td>Loop exits</td>
<td>Subroutine returns</td>
<td>Subroutine returns</td>
<td>Loop bodies</td>
</tr>
<tr>
<td>Source of implicit threads</td>
<td>Loop bodies</td>
<td>Loop bodies</td>
<td>Loop bodies</td>
<td>Loop bodies</td>
</tr>
<tr>
<td>Thread-creation mechanism</td>
<td>Software/compiler</td>
<td>Implicit thread</td>
<td>Implicit thread</td>
<td>Software/compiler</td>
</tr>
<tr>
<td>Thread creation and sequencing</td>
<td>Program order</td>
<td>Out of program order</td>
<td>Out of program order</td>
<td>Program order</td>
</tr>
<tr>
<td>Thread execution</td>
<td>Out of order</td>
<td>Out of order</td>
<td>Out of order</td>
<td>Out of order</td>
</tr>
<tr>
<td>Register data dependencies</td>
<td>Software with hardware speculation</td>
<td>Software, no speculation</td>
<td>Software, independent of speculation and specifications</td>
<td>Software, independent of speculation and specifications</td>
</tr>
<tr>
<td>Memory data dependencies</td>
<td>Hardware-supported speculation</td>
<td>Software</td>
<td>Software, independent of speculation</td>
<td>Hardware-supported speculation, checked with simple algorithms in MESI coherence</td>
</tr>
</tbody>
</table>

Executing The Same Thread

- Main thread: Detect faults by comparing results; Redundant thread
- Redundant thread: Prefetch into caches, resolve branches; Main thread

- Why execute the same thread twice?
  - Detect faults
  - Better performance
  - Prefetch, resolve branches

Fault Detection

- AR/SMT [Rotenberg 1999]
  - Use second SMT thread to execute program twice
  - Compare results to check for hard and soft errors (faults)

- DIVA [Austin 1999]
  - Use simple check-processor at commit
  - Re-execute all ops in order
  - Possibly relax main processor’s correctness constraints and safety margins to improve performance
    - Lower voltage, higher frequency, etc.

- Lots of other variations proposed in more recent work
**Speculative Pre-execution**

- Idea: create a *runahead* or *future thread* that helps the main trailing thread
- Advantage: speculative *future thread* has no correctness requirement
- Slipstream processors [Rotenberg 2000]
  - Construct speculative, stripped-down version ("future thread")
  - Let it run ahead and prefetch
  - Construct backward dataflow slice for problematic instructions (mispredicted branches, cache misses)
  - Pre-execute this slice of the program
  - Resolve branches, prefetch data
  - Implemented in Intel production compiler, reflected in Intel Pentium 4 SPEC results

**Summary**

- Thread-level parallelism
  - Abundant, esp. in server applications
- Multiprocessors, cache coherence
- CMP, coarse-grained MT, SMT
- Implicit multithreading
  - Multiscalar, DEE, DMT, TLS
- Redundant multithreading