Lecture 3 Outline

• Multithreaded processors
• Multicore processors
Multithreaded Cores

• Basic idea:
  – CPU resources are expensive and should not be idle
• 1960’s: Virtual memory and multiprogramming
  – Virtual memory/multiprogramming invented to tolerate latency to secondary storage (disk/tape/etc.)
  – Processor-disk speed mismatch:
    • microseconds to tens of milliseconds (1:10000 or more)
  – OS context switch used to bring in other useful work while waiting for page fault or explicit read/write
  – Cost of context switch must be much less than I/O latency (easy)
Multithreaded Cores

• 1990’s: Memory wall and multithreading
  – Processor-DRAM speed mismatch:
    • nanosecond to fractions of a microsecond (1:500)
  – H/W task switch used to bring in other useful work while waiting for cache miss
  – Cost of context switch must be much less than cache miss latency

• Very attractive for applications with abundant thread-level parallelism
  – Commercial multi-user workloads
Approaches to Multithreading

• Fine-grain multithreading
  – Switch contexts at fixed fine-grain interval (e.g. every cycle)
  – Need enough thread contexts to cover stalls
  – Example: Tera MTA, 128 contexts, no data caches

• Benefits:
  – Conceptually simple, high throughput, deterministic behavior

• Drawback:
  – Very poor single-thread performance
Approaches to Multithreading

• Coarse-grain multithreading
  – Switch contexts on long-latency events (e.g. cache misses)
  – Need a handful of contexts (2-4) for most benefit
• Example: IBM RS64-IV (Northstar), 2 contexts
• Benefits:
  – Simple, improved throughput (~30%), low cost
  – Thread priorities mostly avoid single-thread slowdown
• Drawback:
  – Nondeterministic, conflicts in shared caches
Approaches to Multithreading

• Simultaneous multithreading
  – Multiple concurrent active threads (no notion of thread switching)
  – Need a handful of contexts for most benefit (2-8)

• Examples: Intel Pentium 4, IBM Power 5/6/7, Alpha EV8/21464

• Benefits:
  – Natural fit for OOO superscalar
  – Improved throughput
  – Low incremental cost

• Drawbacks:
  – Additional complexity over OOO superscalar
  – Cache conflicts

Mikko Lipasti-University of Wisconsin
Approaches to Multithreading

• Chip Multithreading (CMT)
  – Similar to CMP
• Share something in the core:
  – Expensive resource, e.g. floating-point unit (FPU)
  – Also share L2, system interconnect (memory and I/O bus)
• Example: Sun Niagara, 8 cores per die, one FPU
• Benefits:
  – Same as CMP
  – Further: amortize cost of expensive resource over multiple cores
• Drawbacks:
  – Shared resource may become bottleneck
  – 2nd generation (Niagara 2) does **not** share FPU
## Multithreaded/Multicore Processors

<table>
<thead>
<tr>
<th>MT Approach</th>
<th>Resources shared between threads</th>
<th>Context 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>Coarse-grained</td>
<td>Everything but I-fetch buffers, register file and control logic/state</td>
<td>Switch on pipeline stall</td>
</tr>
<tr>
<td>SMT</td>
<td>Everything but instruction fetch buffers, return address stack, architected register file, control logic/state, reorder buffer, store queue, etc.</td>
<td>All contexts concurrently active; no switching</td>
</tr>
<tr>
<td>CMT</td>
<td>Various core components (e.g. FPU), secondary cache, system interconnect</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 Power7 CMP+SMT
SMT Microarchitecture [Emer,’01]
SMT Microarchitecture [Emer,’01]
SMT Performance [Emer,’01]

Multiprogrammed workload

![Chart showing performance comparison for different workloads and core counts.]
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

Memory
Latency = 1 Barrel Rotation

SLOT
Time-shared instruction control

I/O Programs in Barrel

PC
reg0
reg1
regn-1

PC
reg0
reg1
regn-1

Latency = 1 Barrel Rotation
Denelcor HEP

- 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
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)
HEP Processor Organization
HEP Processor, contd.

- Address space: 32K to 1Mwords (64 bits)
- 64 bit instructions
- 2048 GP registers + 4096 constants
  - Registers can be shared among threads
- Memory operation
  - Loads and stores performed by scheduler functional unit (SFU)
  - SFU builds network packet and sends it into switch
  - PSW is removed from control loop and placed in SFU queue
  - PSW is placed back into control loop following memory response
- Special operations
  - control instructions to create/terminate threads
  - full/empty bits in memory and registers
    - busy wait on empty/full operands
Switch

• Packet switched, but bufferless
• 3 bi-directional ports per switch
  – 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
  – [Marr reading]
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

• **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)

Instruction Cache

Trace Cache
Capacity Resource Sharing

• Append thread identifier (TId) to threads in shared capacity (storage) resource

• Example: cache memory

![Diagram of 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
  - HALT – privileged instruction => (normally) low power mode
    - IN MT mode => transition to ST0 or ST1
      (depending on the thread that HALTed)
    - In ST mode => low power mode
- ST-mode (ST0 or ST1) – one logical processor; combined resources
- 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
    • Most likely front-side bus (FSB), i.e. memory bandwidth
Performance

- Web server apps
Intel Hyperthreading Summary

- Mix of partitioned and shared resources
- Mostly round-robin scheduling
- Primary objective: performance
- Secondary objective: fairness
- Not a lot of obvious structure/reasons for design decisions
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 IFetch stage
  – Policy: mux control – round-robin (or priorities)
Example: Hyperthreading

- Mechanisms (and Policies) in Pentium 4
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
Multithreading Summary

• Goal: increase throughput  
  – Not latency
• Utilize execution resources by sharing among multiple threads:  
  – Fine-grained, coarse-grained, simultaneous
• Usually some hybrid of fine-grained and SMT  
  – Front-end is FG, core is SMT, back-end is FG
• Resource sharing  
  – I$, D$, ALU, decode, rename, commit – shared  
  – IQ, ROB, LQ, SQ – partitioned vs. shared
• Historic multithreaded machines
• Recent examples
Lecture 3 Outline

• Multithreaded processors
• Multicore processors
Processor Performance

Processor Performance = \frac{\text{Time}}{\text{Program}}

= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}

\begin{align*}
\text{(code size)} & \quad \times \quad \text{(CPI)} & \quad \times \quad \text{(cycle time)}
\end{align*}

- In the 1980’s (decade of pipelining):
  - CPI: 5.0 => 1.15
- In the 1990’s (decade of superscalar):
  - CPI: 1.15 => 0.5 (best case)
- In the 2000’s (decade of multicore):
  - Core CPI unchanged; chip CPI scales with #cores
**Multicore Objectives**

- Use available transistors to add value
  - Provide better perf, perf/cost, perf/watt
- Effectively share expensive resources
  - Socket/pins:
    - DRAM interface
    - Coherence interface
    - I/O interface
  - On-chip area/power
    - Mem controller
    - Cache
    - FPU? (Conjoined cores, e.g. Niagara)
High-Level Design Issues

1. Where to connect cores?
   
   – Time to market:
     * at off-chip bus (Pentium D)
     * at coherence interconnect (Opteron Hypertransport)
   
   – Requires substantial (re)design:
     * at L2 (Power 4, Core Duo, Core 2 Duo, etc.)
     * at L3 (Opteron, Itanium, etc.)
High-Level Design Issues

2. Share caches?
   – yes: all designs that connect at L2 or L3
   – no: initial designs that connected at “bus”

3. Coherence?
   – Private caches? Reuse existing MP/socket coherence
     • Optimize for on-chip sharing?
   – Shared caches?
     • Need new coherence protocol for on-chip caches
     • Often write-through L1 with back-invalidates for other caches
       (mini-directory)
High-Level Design Issues

4. How to connect?
   – Off-chip bus? Time-to-market hack, not scalable
   – Existing pt-to-pt coherence interconnect
     • e.g. AMD’s hypertransport
   – Shared L2/L3:
     • Crossbar, up to 3-4 cores
     • 1D "dancehall“ organization with crossbar
   – On-chip bus? Not very scalable
   – Interconnection network
     • scalable, but high overhead, design complexity
     • E.g. ring, 2D tiled organization, mesh interconnect
Shared vs. Private L2/L3

- Bandwidth issues
  - Data: if shared then banked/interleaved
  - Tags: snoop b/w into L2 (L1 if not inclusive)

- Cache misses: per core vs. per chip
  - Compare same on-chip capacity (e.g. 4MB)
  - When cores share data:
    - Cold/capacity/conflict misses fewer in shared cache
    - Communication misses greater in private cache
  - Conflict misses can increase with shared cache
    - Fairness issues between cores
Shared vs. Private L2/L3

• Access latency: fixed vs. NUCA (interconnect)
  – Classic UMA (dancehall) vs. NUMA
  – Collocate LLC banks with cores
    • Commonly assumed in research literature

• Complexity due to bandwidth:
  – Arbitration
  – Concurrency/interaction

• Coherent vs. non-coherent shared LLC
  – LLC can be "memory cache" below "coherence"
  – Only trust contents after snoop/coherence has determined that no higher-level cache has a dirty copy
Multicore Coherence

• All private caches:
  – reuse existing protocol, if scalable enough

• Some shared cache
  – New LL shared cache is non-coherent (easy)
    • Use existing protocol to find blocks in private L2/L1
    • Serialize L3 access; use as memory cache
  – New shared LLC is coherent (harder)
    • Complexity of multilevel protocols is underappreciated
    • Could flatten (treat as peers) but:
      – Lose opportunity
      – May not be possible (due to inclusion, WB/WT handling)
    • Combinatorial explosion due to multiple protocols interacting
Multicore Coherence

• Shared L2 is coherent via writethru L1
  – Still need sharing list to forward invalidates/writes (or broadcast)
  – Ordering of WT stores and conflicting loads, coherence messages not trivial
  – WT bandwidth is expensive

• Shared L2 with writeback L1
  – Combinatorial explosion of multiple protocols
  – Recent work on fractal coherence (MICRO ‘10), manager-client pairing (MICRO’11) address this
Multicore Interconnects

• Bus/crossbar - dismiss as short-term solutions?
• Point-to-point links, many possible topographies
  – 2D (suitable for planar realization)
    • Ring
    • Mesh
    • 2D torus
  – 3D - may become more interesting with 3D packaging (chip stacks)
    • Hypercube
    • 3D Mesh
    • 3D torus
• More detail in subsequent NoC unit

Mikko Lipasti-University of Wisconsin
Cross-bar (e.g. IBM Power4/5/6/7)
On-Chip Bus/Crossbar

• Used widely (Power4/5/6/7 Piranha, Niagara, etc.)
  – Assumed not scalable
  – Is this really true, given on-chip characteristics?
  – May scale "far enough" : watch out for arguments at the limit
  – e.g. swizzle-switch makes x-bar scalable enough [UMich]

• Simple, straightforward, nice ordering properties
  – Wiring can be a nightmare (for crossbar)
  – Bus bandwidth is weak (even multiple busses)
  – Compare DEC Piranha 8-lane bus (32GB/s) to Power4 crossbar (100+GB/s)
  – Workload demands: commercial vs. scientific
On-Chip Ring (e.g. Intel)
On-Chip Ring

• Point-to-point ring interconnect
  – Simple, easy
  – Nice ordering properties (unidirectional)
  – Every request a broadcast (all nodes can snoop)
  – Scales poorly: $O(n)$ latency, fixed bandwidth

• Optical ring (nanophotonic)
  – HP Labs Corona project
  – Much lower latency (speed of light)
  – Still fixed bandwidth (but lots of it)
On-Chip Mesh

• Widely assumed in academic literature
• Tilera (Wentzlaff reading), Intel 80-core prototype
• Not symmetric, so have to watch out for load imbalance on inner nodes/links
  – 2D torus: wraparound links to create symmetry
    • Not obviously planar
    • Can be laid out in 2D but longer wires, more intersecting links
• Latency, bandwidth scale well
• Lots of recent research in the literature
**CMP Examples**

- Chip Multiprocessors (CMP)
- Becoming very popular

<table>
<thead>
<tr>
<th>Processor</th>
<th>Cores/chip</th>
<th>Multi-threaded?</th>
<th>Resources shared</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM Power 4</td>
<td>2</td>
<td>No</td>
<td>L2/L3, system interface</td>
</tr>
<tr>
<td>IBM Power7</td>
<td>8</td>
<td>Yes (4T)</td>
<td>Core, L2/L3, system interface</td>
</tr>
<tr>
<td>Sun Ultrasparc</td>
<td>2</td>
<td>No</td>
<td>System interface</td>
</tr>
<tr>
<td>Sun Niagara</td>
<td>8</td>
<td>Yes (4T)</td>
<td>Everything</td>
</tr>
<tr>
<td>Intel Pentium D</td>
<td>2</td>
<td>Yes (2T)</td>
<td>Core, nothing else</td>
</tr>
<tr>
<td>AMD Opteron</td>
<td>2</td>
<td>No</td>
<td>System interface (socket)</td>
</tr>
</tbody>
</table>
IBM Power4: Example CMP

Power4 core0
L1 I$  L1 D$

Power4 core1
L1 D$  L1 I$

Crossbar interconnect

Coherent I/O interface

L3 tags & I/F

Address/response data interconnect

~512KB L2 slice

P0 STQ  P1 STQ

L2 tags  L2 data

MSHR  SNP Q  WB Q

~512KB L2 slice

Address/response data interconnect

Address/response data interconnect

Address/response data interconnect

P0 STQ  P1 STQ

L2 tags  L2 data

MSHR  SNP Q  WB Q

~512KB L2 slice

P0 STQ  P1 STQ

L2 tags  L2 data

MSHR  SNP Q  WB Q

~512KB L2 slice

P0 STQ  P1 STQ

L2 tags  L2 data

MSHR  SNP Q  WB Q

~512KB L2 slice

Mikko Lipasti - University of Wisconsin
Niagara Case Study

• Targeted application: web servers
  – Memory intensive (many cache misses)
  – ILP limited by memory behavior
  – TLP: Lots of available threads (one per client)

• Design goal: maximize throughput (/watt)

• Results:
  – Pack many cores on die (8)
  – Keep cores simple to fit 8 on a die, share FPU
  – Use multithreading to cover pipeline stalls
  – Modest frequency target (1.2 GHz)
Niagara Block Diagram [Source: J. Laudon]

- 8 in-order cores, 4 threads each
- 4 L2 banks, 4 DDR2 memory controllers
Ultrasparc T1 Die Photo [Source: J. Laudon]

Features:
- 8 64-bit Multithreaded SPARC Cores
- Shared 3 MB, 12-way 64B line writeback L2 Cache
- 16 KB, 4-way 32B line ICache per Core
- 8 KB, 4-way 16B line write-through DCache per Core
- 4 144-bit DDR-2 channels
- 3.2 GB/sec JBUS I/O

Technology:
- TI's 90nm CMOS Process
- 9LM Cu Interconnect
- 63 Watts @ 1.2GHz/1.2V
- Die Size: 379mm²
- 279M Transistors
- Flip-chip ceramic LGA
Niagara Pipeline [Source: J. Laudon]

- Shallow 6-stage pipeline
- Fine-grained multithreading
T2000 System Power

- 271W running SpecJBB2000
- Processor is only 25% of total
- DRAM & I/O next, then conversion losses
Niagara Summary

• Example of application-specific system optimization
  – Exploit application behavior (e.g. TLP, cache misses, low ILP)
  – Build very efficient solution

• Downsides
  – Loss of general-purpose suitability
  – E.g. poorly suited for software development (parallel make, gcc)
  – Very poor FP performance (fixed in Niagara 2)
CMPs WITH HETEROGENEOUS CORES

- Workloads have different characteristics
  - Large number of small cores (applications with high thread count)
  - Small number of large cores (applications with single thread or limited thread count)
  - Mix of workloads
  - Most parallel applications have both serial and parallel sections (Amdahl’s Law)

- Hence, heterogeneity
  - Temporal: EPI throttling via DVFS
  - Spatial: Each core can differ either in performance or functionality

- Performance asymmetry
  - Using homogeneous cores and DVFS, or processor with mixed cores (ARM BIG.little)
  - Variable resources: e.g., adapt size of cache via power gating of cache banks
  - Speculation control (unpredictable branches): throttle in-flight instructions (reduces activity factor)

<table>
<thead>
<tr>
<th>Method</th>
<th>EPI Range</th>
<th>Time to vary EPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>DVFS</td>
<td>1:2 to 1:4</td>
<td>100 us, ramp $V_{cc}$</td>
</tr>
<tr>
<td>Variable Resources</td>
<td>1:1 to 1:2</td>
<td>1 us, Fill L1</td>
</tr>
<tr>
<td>Speculation Control</td>
<td>1:1 to 1:1.4</td>
<td>10 ns, Pipe flush</td>
</tr>
<tr>
<td>Mixed Cores</td>
<td>1:6 to 1:11</td>
<td>10 us, Migrate L2</td>
</tr>
</tbody>
</table>
CMPs WITH HETEROGENEOUS CORES (Functional Asymmetry)

• Use heterogeneous cores
  – E.g., GP cores, GPUs, cryptography, vector cores, floating point coprocessors
  – Heterogeneous cores may be programmed differently
  – Mechanisms must exist to transfer activity from one core to another
    • Fine-grained: e.g. FP co-processor, use ISA
    • Coarse-grained: transfer computation using APIs

• Examples:
  – Cores with different ISAs
  – Cores with different cache sizes, different issue width, different branch predictors
  – Cores with different micro-architectures (in-order vs. out-of-order)
  – Different types of cores (GP and SIMD)

• Goals:
  – Save area (more cores)
  – Save power by using cores with different power/performance characteristics for different phases of execution
CMPs WITH HETEROGENEOUS CORES

- Different applications may have better performance/power characteristics on some types of core (static)
- Same application goes through different phases that can use different cores more efficiently (dynamic)
  - Execution moves from core to core dynamically
  - Most interesting case (dynamic)
  - Cost of switching cores (must be infrequent: such as O/S time slice)
- Assume cores with same ISA but different performance/energy ratio
  - Need ability to track performance and energy to make decisions
  - Goal: minimize energy-delay product (EDP)
  - Periodically sample performance and energy spent
    - Run application on one or multiple cores in small intervals
  - Possible heuristics
    - Neighbor: pick one of the two neighbors at random, sample, switch if better
    - Random: select a core at random and sample, switch if better
    - All: sample all cores and select the best
    - Consider the overhead of sampling
IBM CELL PROCESSOR

- ONE PowerPC processing element (PPE)
  - 2-WAY SMT Power CORE
- 8 Synergistic processing elements (SPEs)
  - SPE is 2-issue in-order processor
  - Two SIMD instructions can be issued in each cycle (vectors)
  - No coherence support between SPE and PPE (software-managed scratchpad memory in SPE)
Multicore Summary

• Objective: resource sharing, power efficiency
  – Where to connect
  – Cache sharing
  – Coherence
  – How to connect

• Examples/case studies

• Heterogeneous CMPs
Lecture 3 Summary

• Multithreaded processors
• Multicore processors