Why Multicore Now?

- Moore’s Law for device integration
- Chip power consumption
- Single-thread performance trend

[Source: Intel]

Leakage Power (Static/DC)

- Transistors aren’t perfect on/off switches
- Even in static CMOS, transistors leak
  - Channel (source/drain) leakage
  - Gate leakage through insulator
    - High-K dielectric replacing SiO2 helps
- Leakage compounded by
  - Low threshold voltage
    - Low $V_{th}$ => fast switching, more leakage
    - High $V_{th}$ => slow switching, less leakage
  - Higher temperature
    - Power increases with $C V^2 A f$
- Rough approximation: leakage proportional to area
  - Transistors aren’t free, unless they’re turned off
  - Controlling leakage
    - Power gating (turn off unused blocks)

Reducing Dynamic Power

- Reduce capacitance
  - Simpler, smaller design
  - Reduced IPC
- Reduce activity
  - Smarter design
  - Reduced IPC
- Reduce frequency
  - Often in conjunction with reduced voltage
- Reduce voltage
  - Biggest hammer due to quadratic effect, widely employed
  - However, reduces max frequency, hence performance
  - Dynamic (power modes)
    - E.g. Transmeta Long Run, AMD PowerNow, Intel SpeedStep

Dynamic Power

$$P_{dyn} \approx kCV^2 Af$$

- Aka AC power, switching power
- Static CMOS: current flows when transistors turn on/off
  - Combinational logic evaluates
  - Sequential logic (flip-flop, latch) captures new value (clock edge)
- Terms
  - $C$: capacitance of circuit (wire length, no. & size of transistors)
  - $V$: supply voltage
  - $A$: activity factor
  - $f$: frequency
- Moore’s Law: which terms increase, which decrease?
  - Historically voltage scaling has saved us, but not any more
Frequency/Voltage relationship

- Lower voltage implies lower frequency
  - Lower $V_{th}$ increases delay to sense/latch 0/1
- Conversely, higher voltage enables higher frequency
  - Overclocking
- Sorting/binning and setting various $V_{dd}$ & $V_{th}$
  - Characterize device, circuit, chip under varying stress conditions
  - Black art – very empirical & closely guarded trade secret
  - Implications on reliability
    - Safety margins, product lifetime
    - This is why overclocking is possible

Frequency/Voltage Scaling

- Voltage/frequency scaling rule of thumb:
  - +/- 1% performance buys -/+ 3% power (3:1 rule)
- Hence, any power-saving technique that saves less than 3x power over performance loss is uninteresting
- Example 1:
  - New technique saves 12% power
  - However, performance degrades 5%
  - Useless, since $12 < 3 \times 5$
  - Instead, reduce f by 5% (also V), and get 15% power savings
- Example 2:
  - New technique saves 5% power
  - Performance degrades 1%
  - Useful, since $5 > 3 \times 1$
- Does this rule always hold?

Multicore Mania

- First, servers
  - IBM Power4, 2001
- Then desktops
  - AMD Athlon X2, 2005
- Then laptops
  - Intel Core Duo, 2006
- Your cellphone
  - Baseband/DSP/application/graphics

Why Multicore

Fixed Chip Power Budget

- Amdahl’s Law
  - Ignores (power) cost of n cores
- Revised Amdahl’s Law
  - More cores $\rightarrow$ each core is slower
  - Parallel speedup $< n$
  - Serial portion (1-f) takes longer
  - Also, interconnect and scaling overhead
Fixed Power Scaling

- Fixed power budget forces slow cores
- Serial code quickly dominates

Multicores Exploit 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

- 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
    - Fine-grained multithreading
    - Simultaneous multithreading
  - Multiple processor cores per die
    - Chip multiprocessors (CMP)
    - Chip multithreading (CMT)

Multicore and Multiprocessor Systems

- Focus 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
      - e.g., GPGPU unit described book
    - May have asymmetric processing units or noncoherent caches
- Shared memory idealisms
  - Fully shared memory: usually nonuniform latency
  - Unit latency: approximate with caches
  - Lack of contention: approximate with caches
  - Instantaneous propagation of writes: coherence required

UMA vs. NUMA

- Uniform Memory Access (UMA)
  - Uniform memory access latency
  - Short local latency
  - Unit locality
  - Instantaneous propagation of writes
  - Coherence required
- Nonuniform Memory Access (NUMA)
  - Local memory access
  - Remote memory access
  - Long remote access latency
  - Lack of coherence
Cache Coherence Problem

Load A
Store A <= 1

Memory

P0
P1

Load A Load A

A0
A0

Invalidate Protocol

- Basic idea: maintain single writer property
- Write handling
  - On write, invalidate all other copies of data
  - Make data private to the writer
  - Allow writes to occur until data is requested
  - Supply modified data to requestor directly or through memory
- Minimal set of states per cache line:
  - Invalid (not present)
  - Modified (private to this cache)
- State transitions:
  - Local read or write: I->M, fetch modified
  - Remote read or write: M->I, transmit data (directly or through memory)
  - Writeback: M->I, write data to memory

Invalidate Protocol

- Observation: data can be read-shared
  - Add S (shared) state to protocol: MSI
- State transitions:
  - Local read: I->S, fetch shared
  - Local write: I->M, fetch modified; S->M, invalidate other copies
  - Remote read: M->S, supply data
  - Remote write: M->I, supply data; S->I, invalidate local copy
- Observation: data can be write-private (e.g. stack frame)
  - Avoid invalidate messages in that case
  - Add E (exclusive) state to protocol: MESI
- State transitions:
  - Local read: I->E if only copy, I->S if other copies exist
  - Local write: E->M silently, S->M, invalidate other copies

Sample Invalidate Protocol (MESI)

Current State: I, Event: LW

Local Read: 0, Local Write: 1, Bus Read: 0, Bus Write: 0, Bus Upgrade: 0

I -> S: Lease valid: 0, lease shared: 0
S -> I: Lease valid: 0, lease shared: 0
I -> E: Lease valid: 0, lease shared: 0
E -> M: Lease valid: 0, lease shared: 0
M -> I: Lease valid: 0, lease shared: 0
M -> S: Lease valid: 0, lease shared: 0
M -> E: Lease valid: 0, lease shared: 0
S -> M: Lease valid: 0, lease shared: 0
S -> I: Lease valid: 0, lease shared: 0
S -> E: Lease valid: 0, lease shared: 0
E -> S: Lease valid: 0, lease shared: 0
E -> L: Lease valid: 0, lease shared: 0
L -> E: Lease valid: 0, lease shared: 0
L -> S: Lease valid: 0, lease shared: 0
S -> L: Lease valid: 0, lease shared: 0
L -> M: Lease valid: 0, lease shared: 0
M -> L: Lease valid: 0, lease shared: 0
M -> BU: Lease valid: 0, lease shared: 0
BU -> M: Lease valid: 0, lease shared: 0

Sample Invalidate Protocol (MESI)

Current State: I, Event: LW

Local Read: 0, Local Write: 1, Bus Read: 0, Bus Write: 0, Bus Upgrade: 0

I -> S: Lease valid: 0, lease shared: 0
S -> I: Lease valid: 0, lease shared: 0
I -> E: Lease valid: 0, lease shared: 0
E -> M: Lease valid: 0, lease shared: 0
M -> I: Lease valid: 0, lease shared: 0
M -> S: Lease valid: 0, lease shared: 0
M -> E: Lease valid: 0, lease shared: 0
S -> M: Lease valid: 0, lease shared: 0
S -> I: Lease valid: 0, lease shared: 0
S -> E: Lease valid: 0, lease shared: 0
E -> S: Lease valid: 0, lease shared: 0
E -> L: Lease valid: 0, lease shared: 0
L -> E: Lease valid: 0, lease shared: 0
L -> S: Lease valid: 0, lease shared: 0
S -> L: Lease valid: 0, lease shared: 0
L -> M: Lease valid: 0, lease shared: 0
M -> L: Lease valid: 0, lease shared: 0
M -> BU: Lease valid: 0, lease shared: 0
BU -> M: Lease valid: 0, lease shared: 0

Optimizations

- Observation: data can be read-shared
  - Add S (shared) state to protocol: MSI
- State transitions:
  - Local read: I->S, fetch shared
  - Local write: I->M, fetch modified; S->M, invalidate other copies
  - Remote read: M->S, supply data
  - Remote write: M->I, supply data; S->I, invalidate local copy
- Observation: data can be write-private (e.g. stack frame)
  - Avoid invalidate messages in that case
  - Add E (exclusive) state to protocol: MESI
- State transitions:
  - Local read: I->E if only copy, I->S if other copies exist
  - Local write: E->M silently, S->M, invalidate other copies
Snoopy Cache Coherence
- Snooping implementation
  - Origins in shared-memory-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)

Directory Cache Coherence
- Directory implementation
  - Extra bits stored in memory (directory) record MSI state of line
  - Memory controller maintains coherence based on the current state
  - Other CPUs’ commands are not snooped, instead:
    - Directory forwards relevant commands
    - Ideal filtering: 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 (100s to 1000s of CPUs)
- Can provide both snooping & directory
  - AMD Opteron switches based on socket count

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 an OOO CPU)
    - Both Proc0 and Proc1 enter critical section, since both observe that other’s lock variable (A/B) is not set
  - 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)
Multithreading

- Basic idea: CPU resources are expensive and should not be left idle
- 1960's: Virtual memory and multiprogramming
  - VM/MP invented to tolerate latency to secondary storage
  - Processor-secondary storage cycle-time ratio: microseconds to tens of milliseconds (1:1000 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)
- 1990's: Memory wall and multithreading
  - Processor: non-cache storage cycle-time ratio: nanosecond to fractions of a microsecond (1:500 or worse)
  - HW 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-grained 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

- Coarse-grained multithreading
  - Switch contexts on long-latency events (e.g. cache misses)
  - Need a handful of contexts (2-4) for most benefit
  - Example: IBM Northstar, 2 contexts
  - Benefits: Simple, improved throughput (~30%), low cost
    - Thread priorities mostly avoid single-thread slowdown
  - Drawback: Nondeterministic, conflicts in shared caches
  - Not suitable for out-of-order processors

- Simultaneous multithreading
  - Multiple concurrent active threads (no notion of thread switching)
  - Need a handful of contexts for most benefit (2-8)
  - Example: Intel P4, Core i7, IBM Power 5/6/7
  - Benefits: Natural fit for OOO superscalar
    - Improved throughput
    - Low incremental cost
  - Drawbacks: Additional complexity over OOO superscalar
    - Cache conflicts

SMT Microarchitecture (from Emer, PACT '01)

Multithreading with Multicore

- Chip Multiprocessors (CMP)
  - Share nothing in the core:
    - Implement multiple cores on die
    - Perhaps share L2, system interconnect (memory and I/O bus)
  - Example: IBM Power4, 2 cores per die, shared L2
  - Benefits: Simple replication
    - Packaging density
    - Low interprocessor latency
    - ~2x throughput
  - Drawbacks: L2 shared – conflicts
    - Mem bandwidth shared – could become bottleneck
**Approaches to Multithreading**

- **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 Power 5</td>
<td>2</td>
<td>Yes (2T)</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>Intel Core i7</td>
<td>4</td>
<td>Yes</td>
<td>L3, DRAM, system interface</td>
</tr>
<tr>
<td>AMD Opteron</td>
<td>2, 4, 6, 12</td>
<td>No</td>
<td>L3, DRAM, system interface</td>
</tr>
</tbody>
</table>

**Multithreaded 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>Sack every cycle</td>
</tr>
<tr>
<td>Coarse-grained</td>
<td>Everything but branch buffers, register file and control logic/state</td>
<td>Sack on pipeline stall</td>
</tr>
<tr>
<td>SMT</td>
<td>Everything but instruction fetch buffers, start address stack, architectural register file, control logic/state, reorder buffer, store queue, etc.</td>
<td>All contexts, consistently active, no switching</td>
</tr>
<tr>
<td>CMT</td>
<td>Various core components (e.g. FPU), secondary cache, system interconnect.</td>
<td>All contexts, consistently active, no switching</td>
</tr>
<tr>
<td>CMP</td>
<td>Secondary cache, system interconnect.</td>
<td>All contexts, consistently active, no switching</td>
</tr>
</tbody>
</table>

- Many approaches for executing multiple threads on a single die
  - Mix-and-match: IBM Power7 8-core CMP x 4-way SMT

**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

- Multithreaded
- Shared 3 MB, 12-way 4KB fine-grained cache
- 16 KB, 4-way 32B fine-grained cache per core
- 8 KB, 4-way 16B fine-grained cache per core
- 4 M 64-bit DDR-2 channels
- 1.2 GB/sec, JBUS IO
- Technology:
  - 77A 50nm CMOS Process
  - 4M Cu Interconnect
  - 63 Watts @ 1.25GHz/1.27GHz
  - 100K Transistors
  - Flip-chip ceramic LOA

Niagara Pipeline

- 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)

Summary

- Why multicore now?
- Thread-level parallelism
- Shared-memory multiprocessors
  - Coherence
  - Memory ordering
  - Split-transaction buses
- Multithreading
- Multicore processors