# ECE/CS 552: Parallel Processors

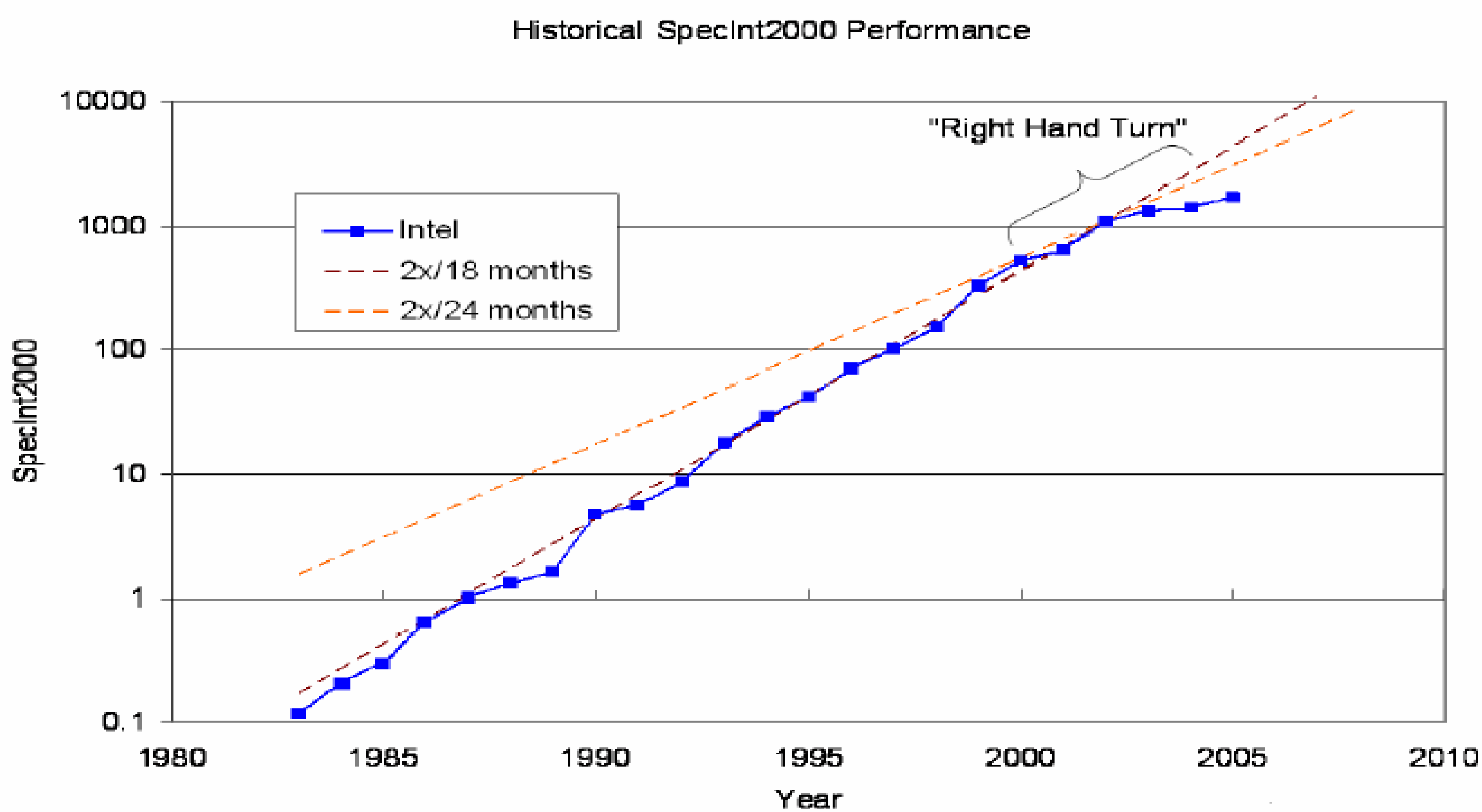


Lecture notes based in part on slides created by Mark Hill, David Wood, Guri Sohi, John Shen and Jim Smith

# Parallel Processors

- Why multicore?
- Static and dynamic power consumption
- Thread-level parallelism
- Parallel processing systems

# Why Multicore Now?

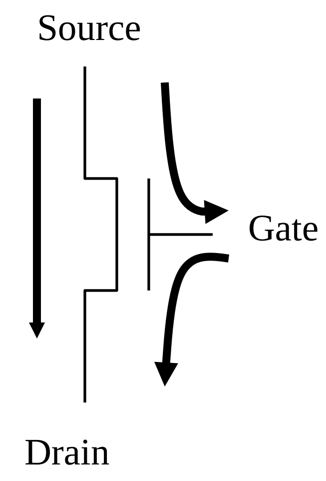



- Moore's Law for device integration
- Chip power consumption
- Single-thread performance trend

[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
- Leakage compounded by
  - Low threshold voltage
    - Low $V_{th}$ => fast switching, more leakage
    - High $V_{th}$ => slow switching, less leakage
  - Higher temperature
    - Temperature increases with power – positive feedback




- Rough approximation: leakage proportional to area
  - Transistors aren’t free, unless they’re turned off
- Controlling leakage
  - Power gating (turn off unused blocks)

# Dynamic Power

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

- 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

- Voltage scaling ended ~2005

# 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)
    - AMD PowerNow, Intel Speedstep

# Frequency/Voltage relationship

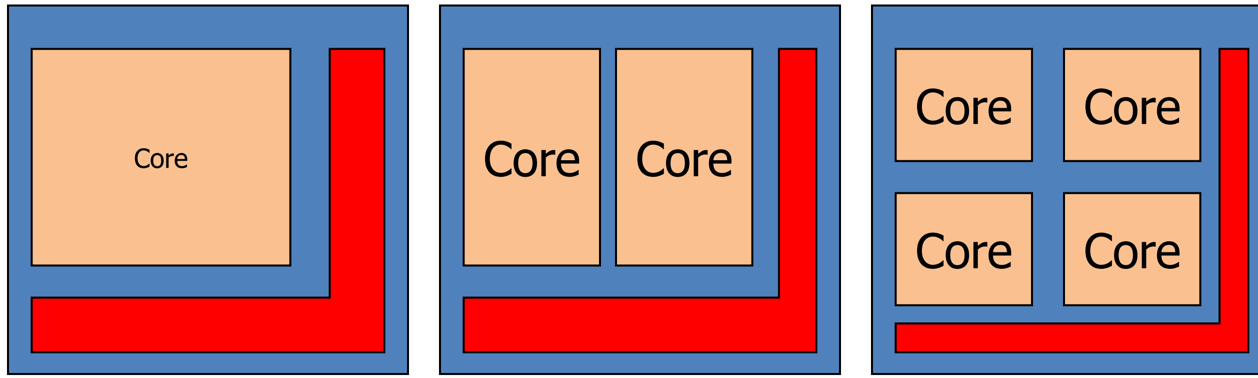
- Lower voltage implies lower frequency
  - Lower $V_{th}$ increases delay to sense/latch 0/1
- Conversely, higher voltage enables higher frequency
  - Overclocking
- Sorting/binning and setting various $V_{dd}$ & $V_{th}$
  - Characterize device, circuit, chip under varying stress conditions

- Design for near-threshold operation
  - Optimize for lower voltage, lower frequency
  - Reap performance via hardware parallelism

# Multicore Mania

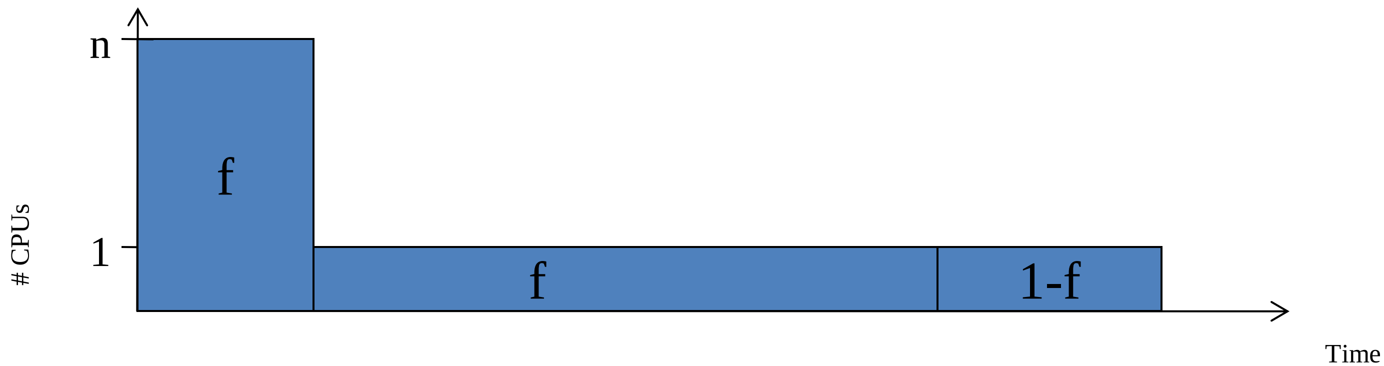
- First, servers
  - IBM Power4, 2001
- Then desktops
  - AMD Athlon X2, 2005
- Then laptops
  - Intel Core Duo, 2006
- Your cellphone
  - Baseband/DSP/GPU
  - Multicore application processors

# Why Multicore




| | Single Core | Dual Core | Quad Core |
|---|---|---|---|
| Core area | A | ~A/2 | ~A/4 |
| Core power | W | < W/2 | < W/4 |
| Chip power | W + O | W' + O' | W'' + O'' |
| Core performance | P | 0.9P | 0.8P |
| Chip performance | P | 1.8P | 3.2P |

# Amdahl's Law



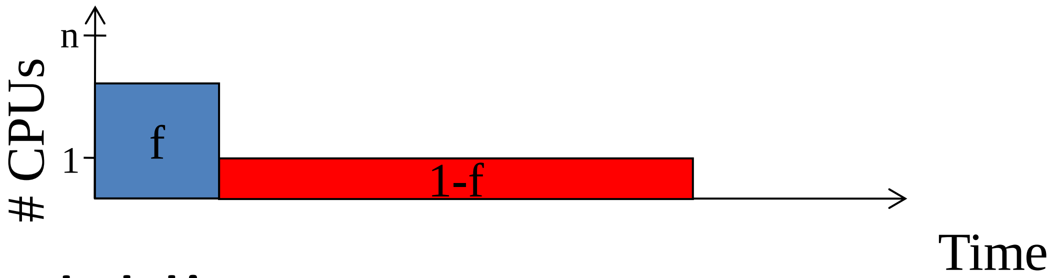

f – fraction that can run in parallel

1-f – fraction that must run serially

$$Speedup = \frac{1}{(1-f)+\frac{f}{n}}$$

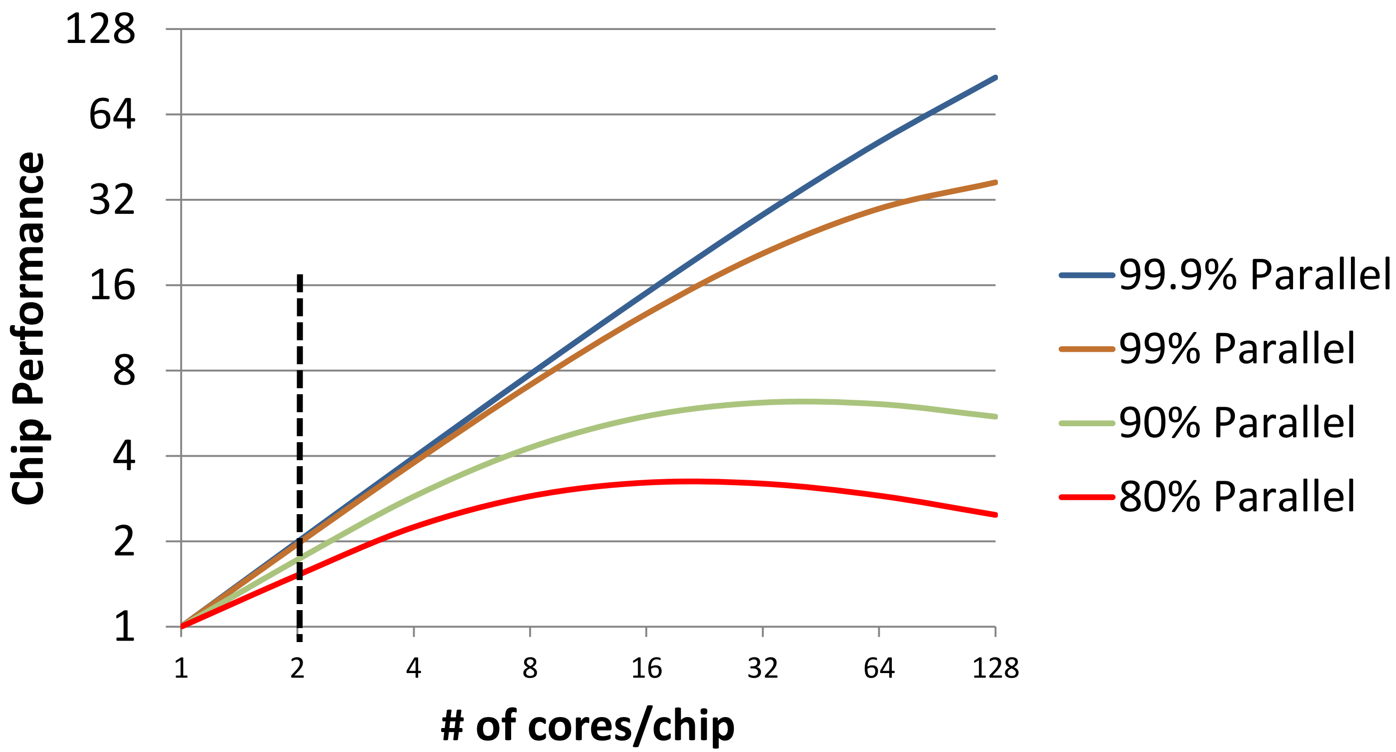
$$\lim_{n\to\infty} \frac{1}{1-f+\frac{f}{n}} = \frac{1}{1-f}$$

# Fixed Chip Power Budget




- Amdahl's Law
  - Ignores (power) cost of n cores
- Revised Amdahl's Law
  - More cores → 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)

# Parallel Processing Systems

- Shared-memory symmetric multiprocessors
  - Key attributes are:
    - Shared memory: all physical memory is accessible to all CPUs
    - Symmetric processors: all CPUs are alike
  - Following lecture covers shared memory design

- Other parallel processors may:
  - Share nothing: compute clusters
  - Share disks: distributed file or web servers
  - Share some memory: GPUs (via PCIx)
  - Have asymmetric processing units: ARM Big/Little
  - Contain noncoherent caches: APUs (AMD Fusion)

# Summary

- Why multicore?
- Static and dynamic power consumption
- Thread-level parallelism
- Parallel processing systems

- Broader and deeper coverage in ECE 757

# ECE/CS 552: Shared Memory

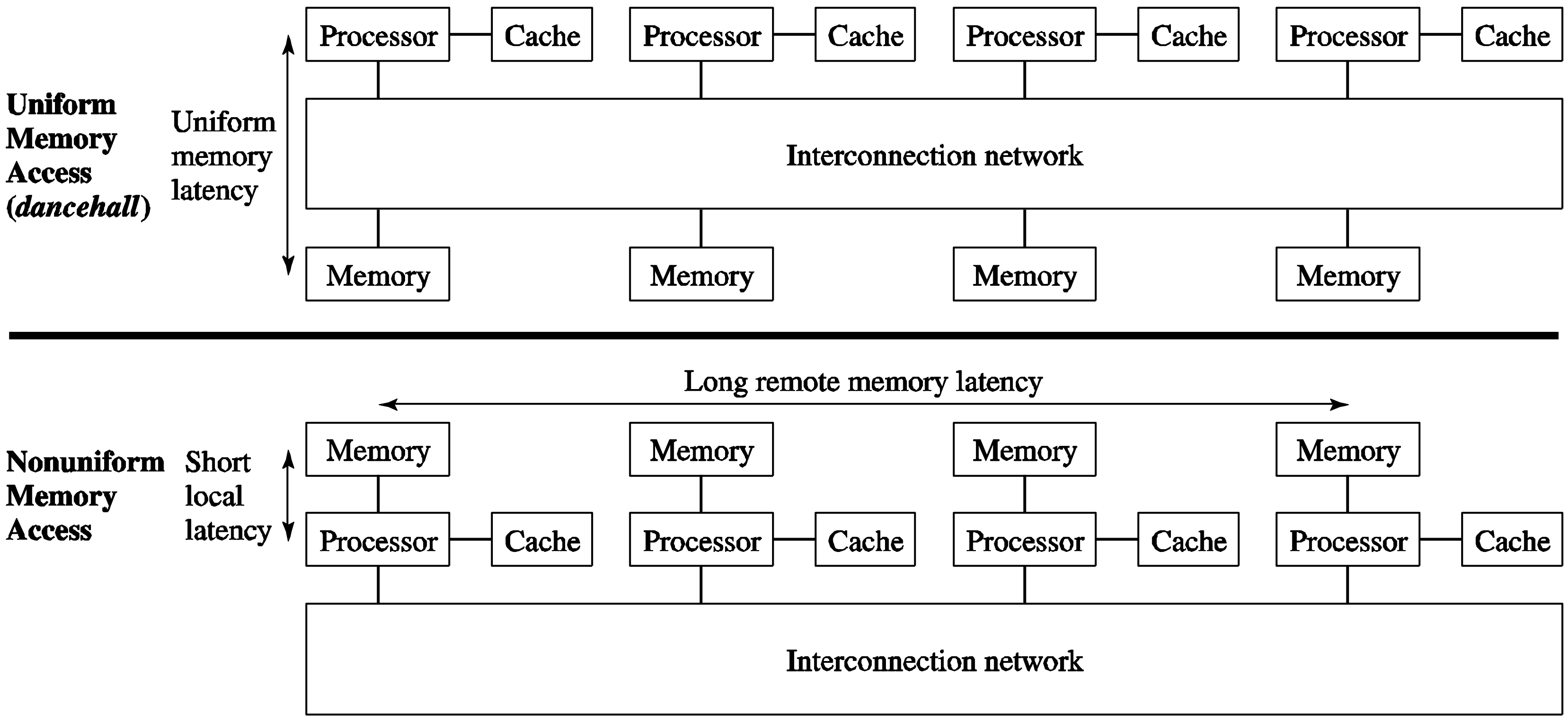
© Prof. Mikko Lipasti

Lecture notes based in part on slides created by Mark Hill, David Wood, Guri Sohi, John Shen and Jim Smith

# Multicore and Multiprocessor Systems

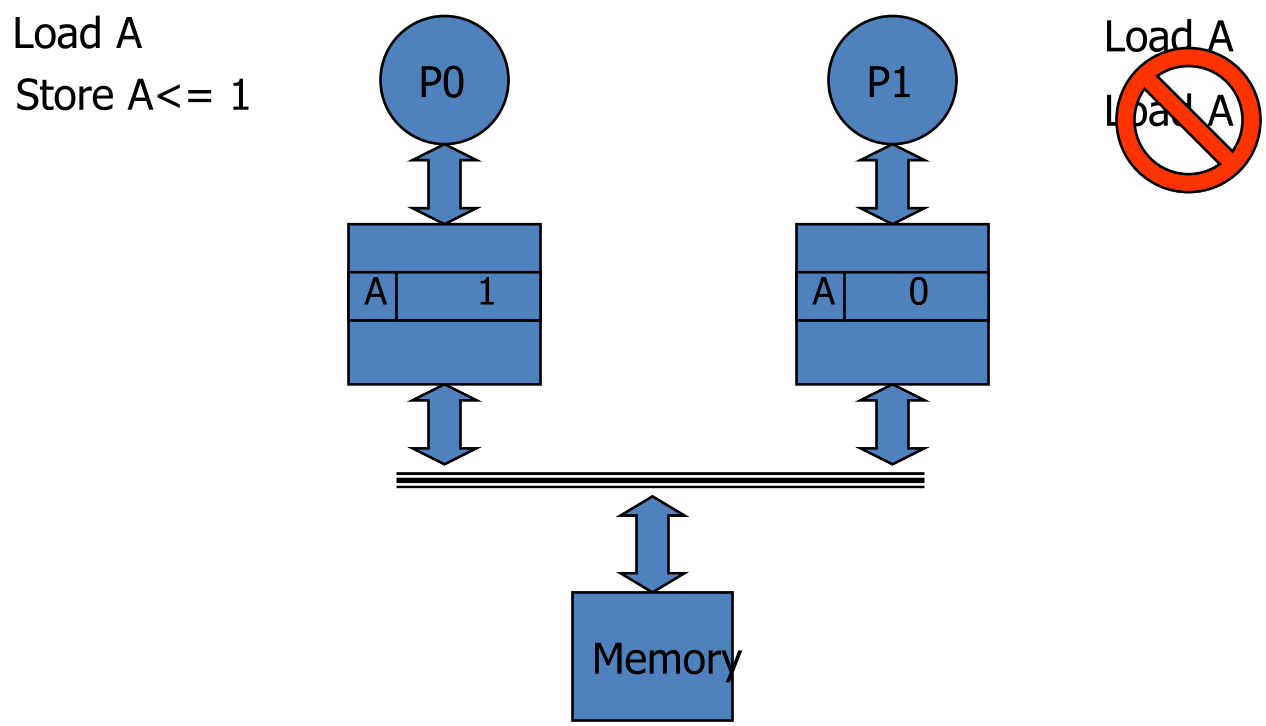
- Focus on shared-memory symmetric multiprocessors
  - Many other types of parallel processor systems have been proposed and built
  - Key attributes are:
    - Shared memory: all physical memory is accessible to all CPUs
    - Symmetric processors: all CPUs are alike
  - Other parallel processors may:
    - Share some memory, share disks, share nothing
      - E.g. GPGPU unit in the textbook
    - 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

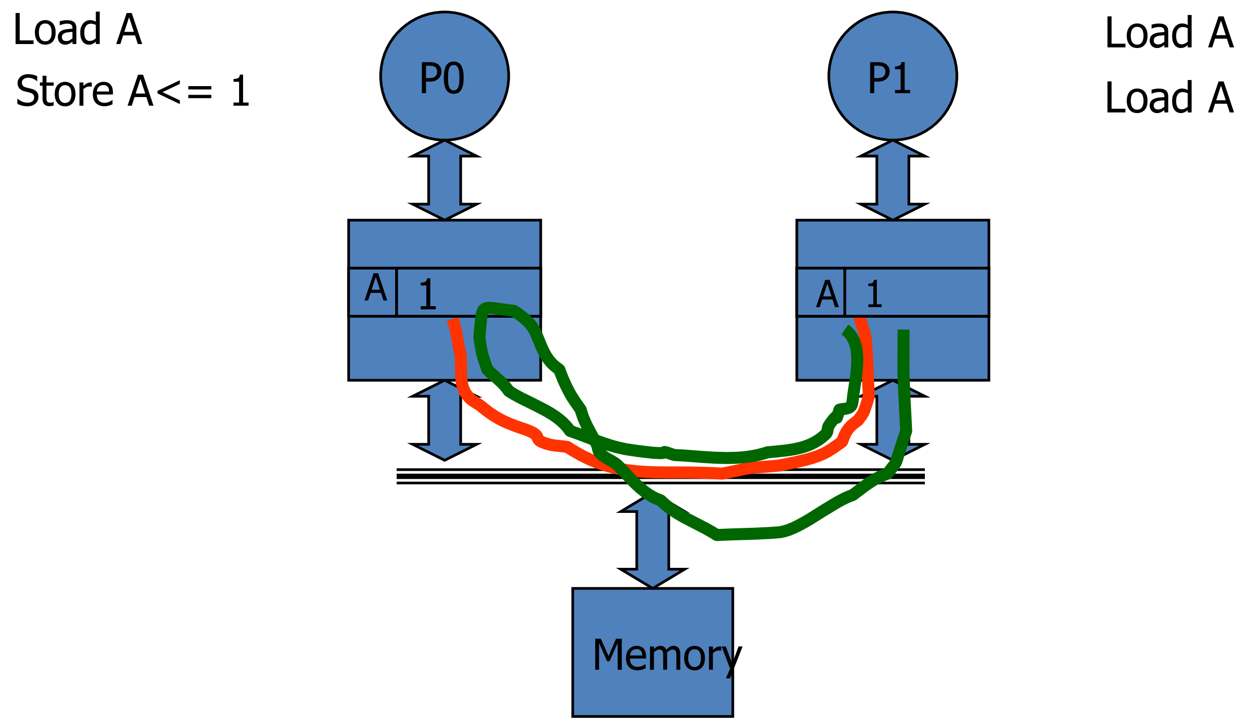


Uniform Memory Access (*dancehall*)
Uniform memory latency
Processor
Cache
Processor
Cache
Processor
Cache
Processor
Cache
Interconnection network
Memory
Memory
Memory
Memory
Long remote memory latency
Nonuniform Memory Access
Short local latency
Memory
Memory
Memory
Memory
Processor
Cache
Processor
Cache
Processor
Cache
Processor
Cache
Interconnection network

# Cache Coherence Problem



# Cache Coherence Problem



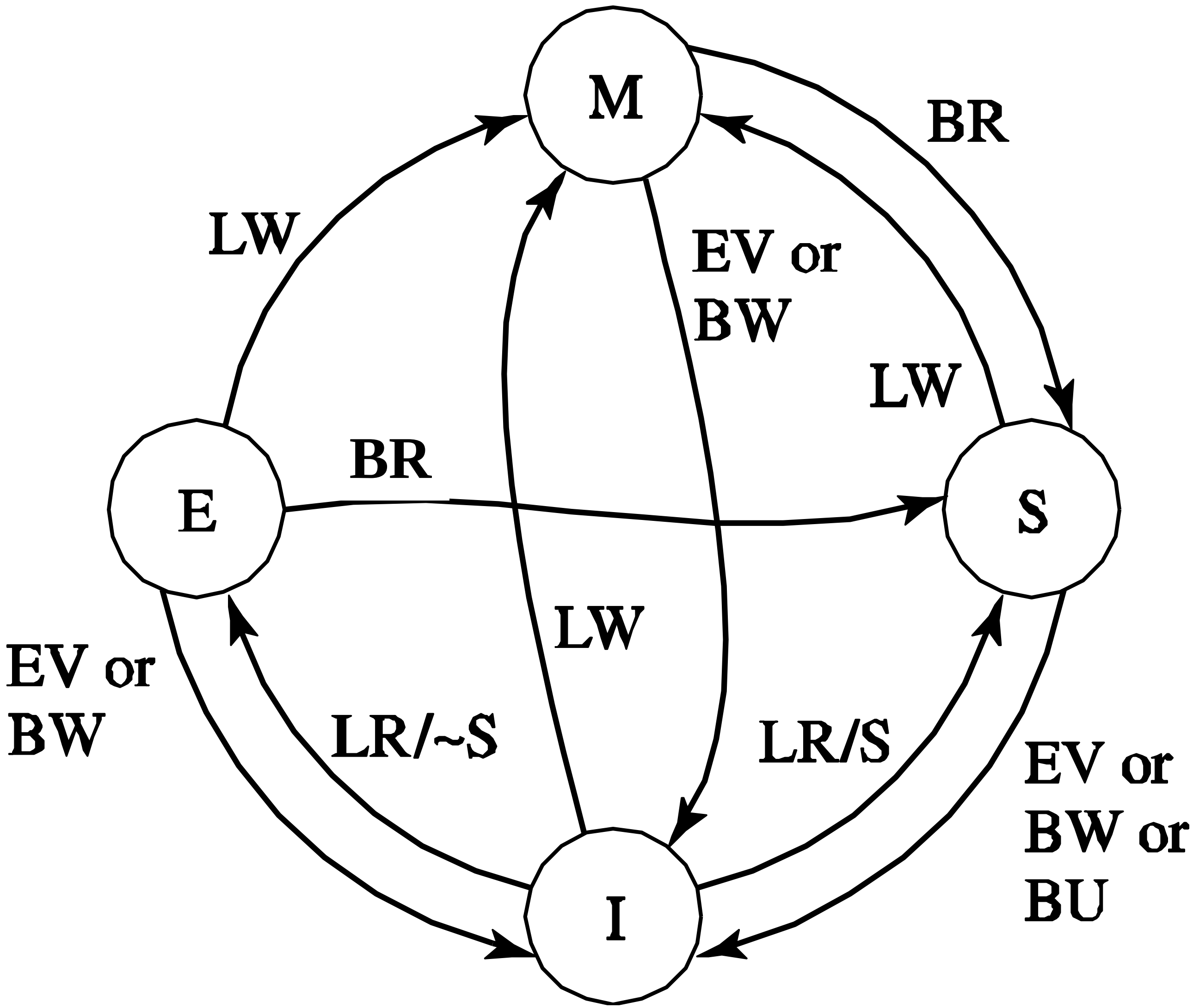
# Invalidate Protocol

- Basic idea: maintain **single writer** property
  - Only one processor has write permission at any point in time
- 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 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

# Sample Invalidate Protocol (MESI)



# Sample Invalidate Protocol (MESI)

| Current State s | Event and Local Coherence Controller Responses and Actions (s' refers to next state) | | | | | |
|---|---|---|---|---|---|---|
| | **Local Read (LR)** | **Local Write (LW)** | **Local Eviction (EV)** | **Bus Read (BR)** | **Bus Write (BW)** | **Bus Upgrade (BU)** |
| **Invalid (I)** | Issue bus read if no sharers then s' = E else s' = S | Issue bus write s' = M | s' = I | Do nothing | Do nothing | Do nothing |
| **Shared (S)** | Do nothing | Issue bus upgrade s' = M | s' = I | Respond shared | s' = I | s' = I |
| **Exclusive (E)** | Do nothing | s' = M | s' = I | Respond shared s' = S | s' = I | Error |
| **Modified (M)** | Do nothing | Do nothing | Write data back; s' = I | Respond dirty; Write data back; s' = S | Respond dirty; Write data back; s' = I | Error |

# Snoopy Cache Coherence

- 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)

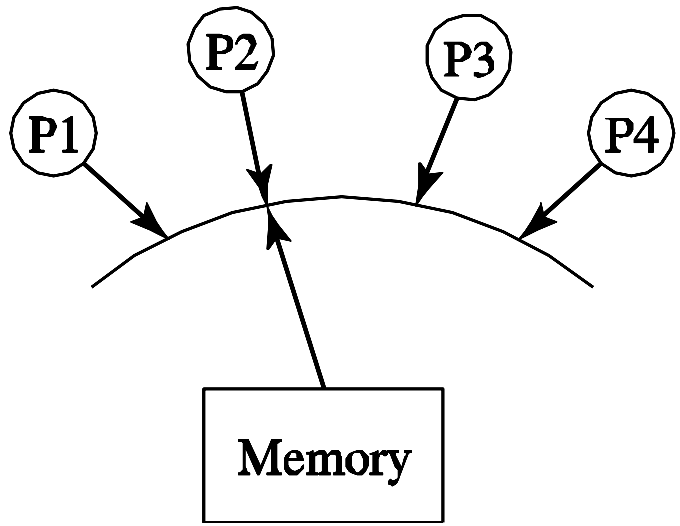
- Can provide both snooping & directory
  - AMD Opteron switches based on socket count

# Memory Consistency

```
Reorder       Proc0                          Proc1
load
before        st A=1                         st B=1
store         if (load B==0) {               if (load A==0) {
                ...critical section            ...critical section
              }                              }
```

- 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
- 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]

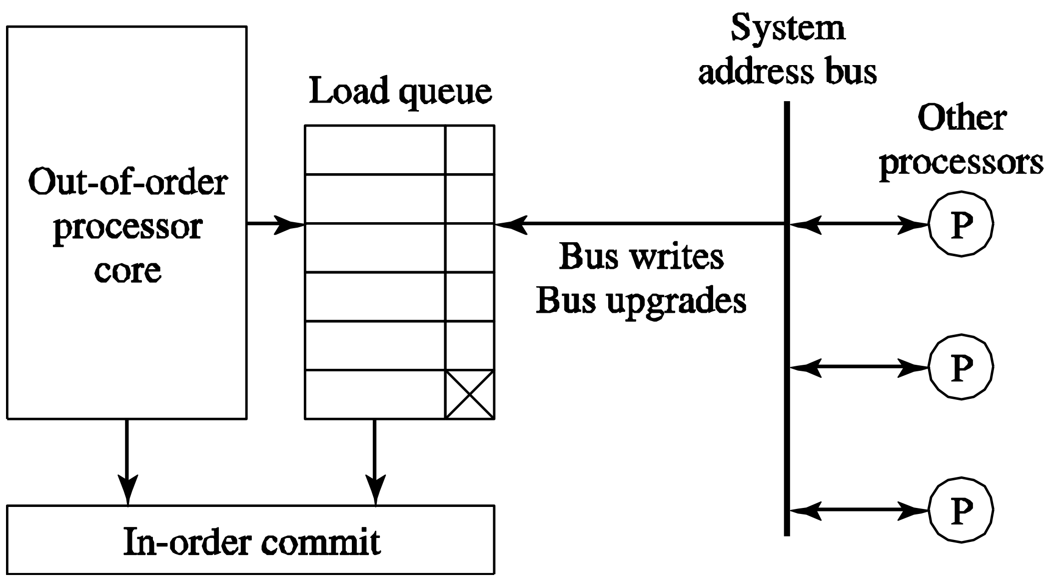



- Processors treated as if they are interleaved processes on a single time-shared CPU
- All references must fit into a total global order or interleaving that does not violate any CPUs program order
  - Otherwise sequential consistency not maintained
- Now Dekker’s algorithm will work
- Appears to preclude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

# High-Performance Sequential Consistency




1. Load queue records all speculative (early) loads
2. Bus writes/upgrades (remote stores) are checked against LQ
3. Any matching load will be replayed and get new value via coherence

Proper order for loads/stores is maintained as long as all events show up on the bus in the proper order

# Summary

- 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
- Shared memory implementation
  - Coherence – snooping vs. directory
  - Memory ordering
- Much more in ECE 757