18-447 Lecture 18: Multithreading and Multicores

James C. Hoe
Dept of ECE, CMU
April 1, 2009

Announcements:

Handouts: Handout #13 Project 4 (On Blackboard)
"Design Challenges of Technology Scaling", Shekhar Borkar, IEEE Micro, 1999 (on Blackboard)

Instruction-Level Parallelism

- When executing a program, how many "independent" instructions can be performed in parallel
- How to take advantage of ILP
  - Pipelining (including superpipelining)
    - overlap different stages from different instructions
    - limited by divisibility of an instruction and ILP
  - Superscalar (including VLIW)
    - overlap processing of different instructions in all stages
    - limited by ILP
- How to increase ILP
  - dynamic/static register renaming ⇒ reduce WAW and WAR
  - dynamic/static instruction scheduling ⇒ reduce RAW hazards
  - use predictions to optimistically break dependence

Thread-Level Parallelism

- The average processor actually executes several "programs" (a.k.a. processes, threads of control, etc) at the same time (time multiplexing)
- The instructions from these different threads have lots of parallelism
- Taking advantage of "thread-level" parallelism, i.e. by concurrent execution, can improve the overall throughput of the processor (but not turn-around time of any one thread)
- Assumption: a single thread cannot use the full performance potential of the processor
  - peak performance is always higher than average
  - must overprovision to achieve your average perf. target

Classic Time-multiplex Multiprocessing

- Time-multiplex multiprocessing on uniprocessors started back in 1962 to enable sharing
- Even concurrent execution by time-multiplexing improves throughput
  - a single thread would effectively idle the processor when spin-waiting for new event or for I/O to complete
  - can spin-wait for thousands to millions of cycles at a time

  - a thread should just go to "sleep" when waiting and let other threads use the processor,

  - keep in mind, this is very coarse-grain interleaving

Classic Context Switching

- A "context" is all of the processor (plus machine) states associated with a particular process
  - programmer visible states: program counter, register file contents, memory contents
  - and some invisible states: control and status reg, page table base pointers, page tables

    What about cache, BTB and TLB entries?
- Classic Context Switching
  - interrupt stops a program mid-execution (precise)

    (a thread can also voluntarily give up control by a syscall
  - OS saves away the context of the stopped thread
  - OS restores the context of a previously stopped thread
  - OS uses a "return from exception" to jump to the restarting PC

    The restored thread has no idea it was interrupted, removed, later restored and restarted

Saving and Restoring Context

- Saving
  - "Context" information that occupy unique resources must be copied and saved to memory by the OS
    - e.g. PC, GPR, cntrl/status reg
  - "Context" information the occupy commodity resources just needs to be hidden from the other threads
    - e.g. active pages in memory can be left in place but hidden via address translation (more on this when we talk about VM protection)
- Restoring is the opposite of saving
- The act of saving and restoring is performed by the OS in software

  $\Rightarrow$ can take a few hundred cycles per switch, but the cost is amortize over a long execution "quantum"

(If you want the full story, take a real OS course!)

Fast Context Switches

- A processor becomes idle when a thread runs into a cache miss Why not switch to another thread?
- Cache miss lasts only tens of cycles, but it costs at least 64 inst just to save and restore the 32 GPRs
- Solution: fast context switch in hardware
  - replicate hardware context registers: PC, GPRs, cntrl/status, PT base ptr eliminates copying
  - allow multiple context to share some resources, i.e. include process ID as cache, BTB and TLB match tags
    eliminates cold starts
  - hardware context switch takes only a few cycles
    - set the PID register to the next process ID
    - select the corresponding set of hardware context registers to be active

Really Fast Context Switches

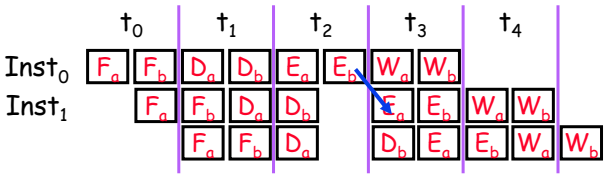
- When pipelined processor stalls due to RAW dependence between instructions, the execution stage is idling
  Why not switch to another thread?
- Not only do you need hardware contexts, switching between contexts must be instantaneous to have any advantage!!
- If this can be done,
  - don't need complicated forwarding logic to avoid stalls
  - RAW dependence and long latency operations (multiply, cache misses) do not cause throughput performance loss

Multithreading is a "latency hiding" technique

Fine-grain Multithreading
(hide latency)

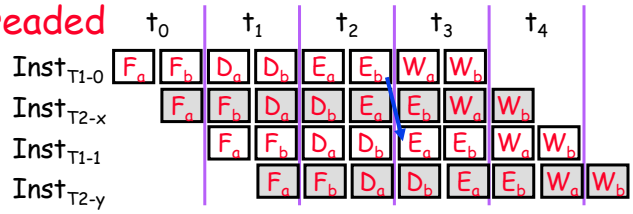
- Superpipeline revisited
  - deeper pipeline to increase frequency and performance
  - but back-to-back dependencies cannot be forwarded
  - no performance gain without ILP

superpipelined



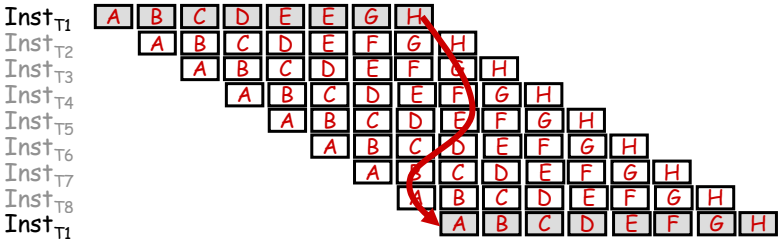
- What about

2-way multithreaded superpipelined

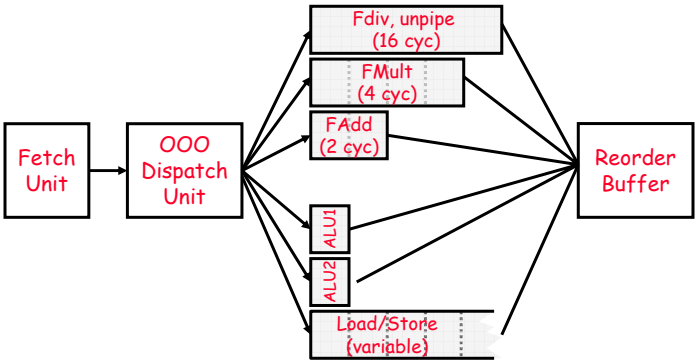


Barrel Instruction Pipeline
(hide latency)

- On each cycle, select a "ready" thread from scheduling pool [HEP, Smith]
  - only one instruction per context in flight at once
  - on a long latency stall, remove the context from scheduling
- Actually make pipelining simpler
  - no data dependence, hence no stall or forwarding
  - no penalty in making pipeline deeper for frequency or complexity
  - assume there are many threads waiting to run



Multithreading and Out-of-order (better utilization)

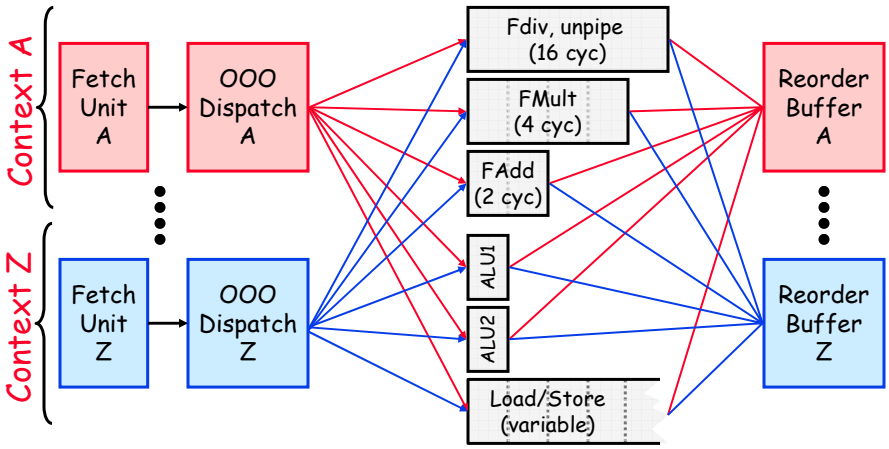


- Superscalar processor datapath must be over-resourced
  - has more functional units than ILP because the units are not universal
  - current 4 to 8 way designs only achieves IPC of 2 to 3
- Some units must be idling in each cycle

Why not switch to another thread?

Simultaneous Multi-Threading

[Eggers, et al.]

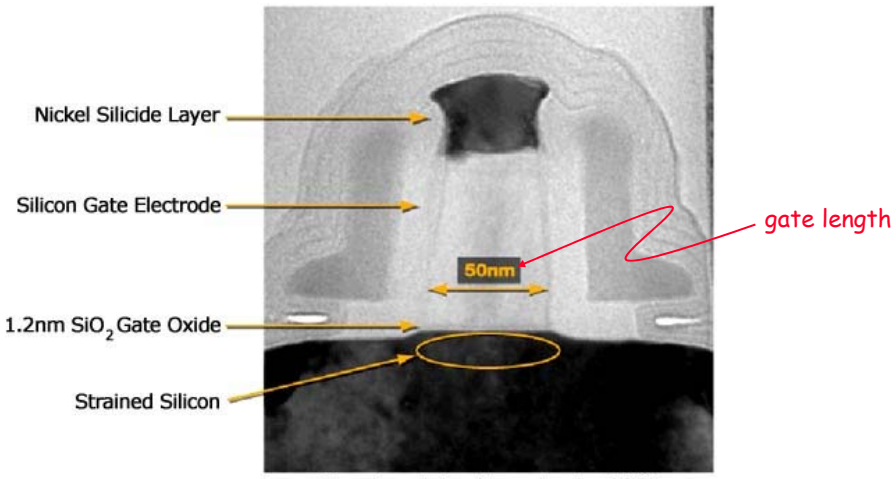


- Dynamic and flexible sharing of functional units between multiple threads

⇒ increases utilization ⇒ increases throughput

The rise of multicores

Transistor Scaling



50nm transistor dimension is ~2000x smaller than diameter of human hair

[http://www.intel.com/museum/online/circuits.htm]

distance between silicon atoms ~ 500 pm

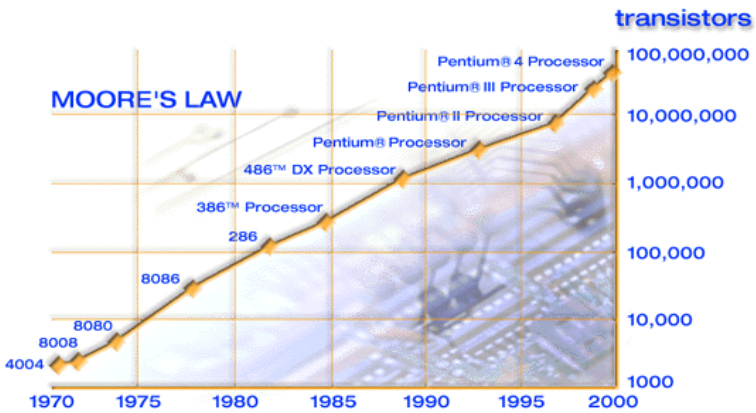
Basic Scaling Theory

- Planned scaling occurs in discrete "nodes" where each is ~0.7x of the previous in linear dimension
  e.g., 90nm, 65nm, 45nm, 32nm, 22nm, 15nm, "The End"
- Take the same design, reducing the linear dimensions by 0.7x (aka "gate shrink") leads to
  - delay = 0.7x, frequency=1.43x
  - capacitance = 0.7x
  - die area = 0.5x
  - Vdd=0.7x (if constant field) or Vdd=1x (if constant voltage)
  - power = $C \times V^2 \times f$ = 0.5x (if constant field)
  - BUT power = 1x (if constant voltage)
- Take the same area, then
  - transistor count = 2x
  - power = 1x (constant field), power = 2x (constant voltage)

[refer to the Shekhar article for the more complete story]

Moore's Law

- The number of transistors that can be <u>economically</u> integrated shall double every 24 m



[http://www.intel.com/research/silicon/mooreslaw.htm]

Moore's Law → Performance

- Microprocessor performance has been doubling about every 24 months
  - is this Moore's Law?
  - are we doing well?
- According to scaling theory, we should get

  constant complexity: 1x transistor at 1.43x frequency

  ⇒ 1.43x performance at 0.5x power

  max complexity: 2x transistor at 1.43x frequency

  ⇒ 2.8x performance at constant power
- Instead, we have been getting (for high-perf CPUs)
  - ~2x transistor?
  - ~2x frequency (how
  - we get about ~2x performance at ~2x power

Performance (In)efficiency

- To hit the "expected" performance target on single-thread microprocessors
  - we had been pushing frequency harder by deepening pipelines
  - we used the 2x transistors to build more complicated microarchitectures so the fast/deep pipelines don't stall (i.e., caches, BP, superscalar, out-of-order)
- The consequence of performance inefficiency

limit of economical PC cooling [ITRS]

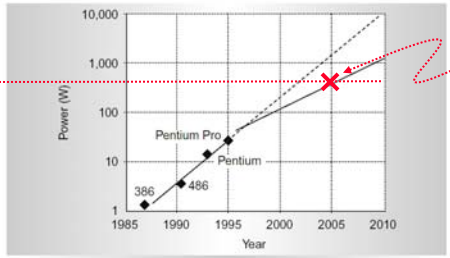


Figure 8. Power dissipation projections.

[from Shekhar Borkar, IEEE Micro, July 1999]

Intel Tehas 150W (Guess what happened next)

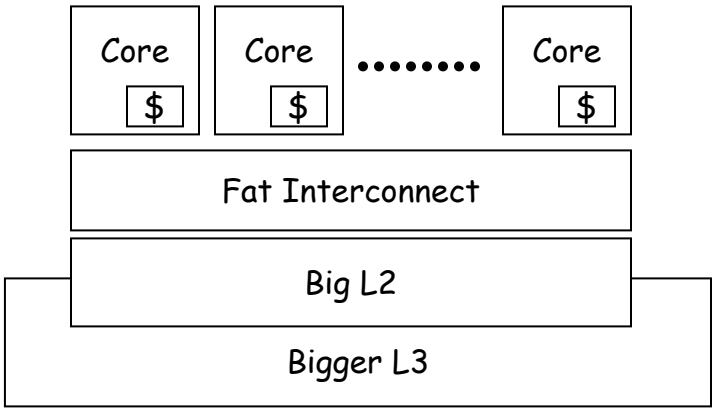
Multicore Performance Efficiency

- PC-class cooling and packaging technologies cap per-die CPU power to ~150W
- Going forward,
  - still deliver 2x performance every 24 months
  - but do it without increasing power
  - in other words, must go faster and at the same time use less energy per instruction How do you do that?
- How about just use the 2x transistor per technology node to 2x the number of cores?
  - 2x the "aggregate" performance without even having to increase frequency
  - slow-down or even stop power climb

  *** this only works well when we have sufficient "parallel" workloads to keep all cores busy

  *** "uncore" portion of processors become very important

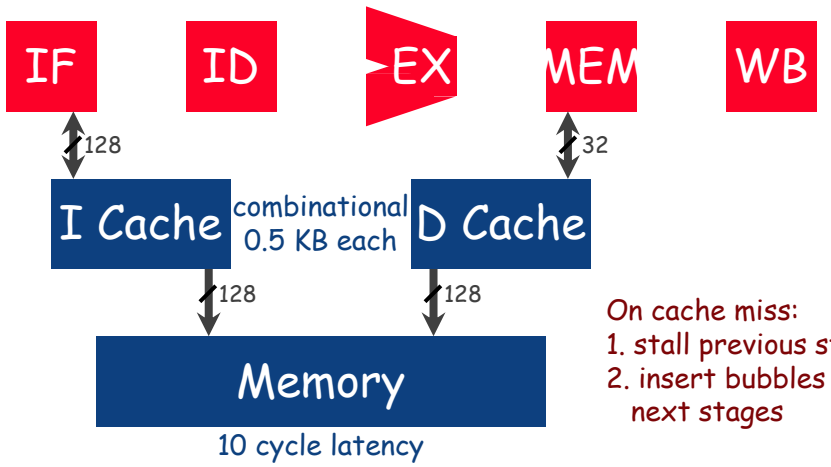
Chip-Multiprocessor



- Current CMPs adopt the familiar SMP paradigm
- future design focus on the "uncore"
  - how to support interprocessor communication
  - how to support programmability

Proj 4: Multithreading Multicore

Checkpoint 1: Caches



On cache miss:
1. stall previous stages
2. insert bubbles for next stages

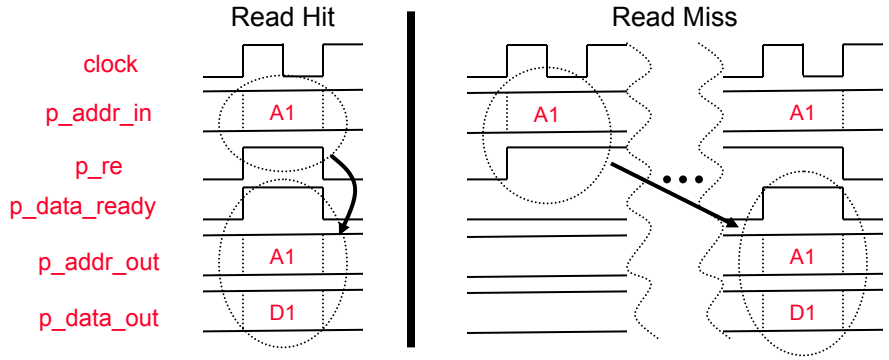
Real memory are even slower, 150 cycles for 3.0 GHz CPU w/DDR2

Selected cache details

- Both caches are
  - direct-mapped
  - 128-bit cache lines
  - write-back
- Cache hits indicated by p_data_ready in same cycle as read/write request
- Misses take 10 or more cycles
- Instruction cache read-only, returns entire cache line
- Modify the caches to output hit/miss statistics
- A new memory module
  - Wider data bus to accommodate caches

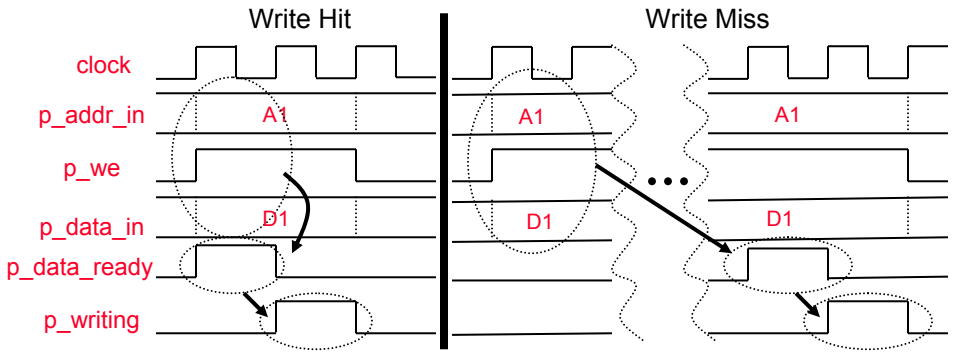
Using the Cache

- Performing a cache read
  - Provide address in p_addr_in, and assert p_re
  - Maintain inputs until p_data_ready is asserted
    - Cache hit: within the same cycle (combinational)
    - Cache miss: at least 10 cycles to read from memory
  - Then, check p_addr_out, and get data from p_data_out



Using the Cache

- Performing a cache write
  - Provide p_addr_in, p_data_in, and assert p_we
  - Maintain inputs until p_data_ready and ***p_writing*** asserted
    - Cache hit: within the same cycle (combinational)
    - Cache miss: at least 10 cycles to bring block from mem
    - Note: p_writing asserted ***1 cycle*** after p_data_ready



Structural Hazard on D-cache Writes

- On cache hit, need extra cycle to perform the write
  - D-cache unavailable for 2 cycles
  - If a ST is followed by a memory inst → structural hazard!
- How to deal with this?
  - On Decode stage, detect hazard
    - i.e., if inst in E is ST, and inst in D is mem inst
  - On hazard, inject one bubble
    - Allow the older ST to complete before executing the subsequent mem inst

Checkpoint 1: Pipeline Changes

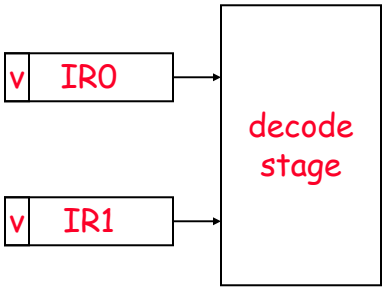
- I-cache
  - Addressed at a cache block granularity
  - Stall fetch on miss, propagate bubbles
- D-cache
  - Addressed at a word granularity
  - Stall on load/store misses
  - Stall on (2 cycle) store hits
- Be deliberate when enabling reads and writes
- As before, interlock for ld dependencies (no delay slot)
  - Stall on load dependency

Checkpoint 2: Multithreading

- We will make one pipeline run two programs/ threads "simultaneously"
  - add a second set of architectural state registers (PC and register file) for thread 0 and thread 1
  - instructions in the pipeline are tagged by thread-ID, 0 or 1
  - use instruction's thread-ID to choose which set of architectural states to access
  - what about memory states? (more later)
- Keep pipeline full *without forwarding*
  - the decode stage tries to issue instruction from thread 0 unless it encounters a dependency or structural hazard
  - while thread 0 is stalled, decode issues from thread 1; if both thread 0 and thread 1 stall, decode stage stalls
  - as soon as thread 0 is able to advance, return to issuing from thread 0

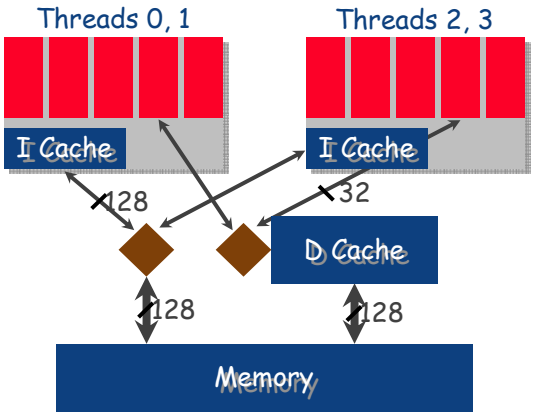
How do we fetch?

- Separate IRs for thread 0 and 1
  - ideally both are full on each cycle so the decode stage can choose thread according to hazards
  - if one IR is empty, decode should try to issue from the other thread
- The "fetch" stage needs to be synchronized with decode to replenish the IR is issuing, except
  - if IR0 is empty, always try to fetch IR0 next (even if thread 1 is issuing)
  - if IR1 is empty, it is only fetched if IR0 is not empty and thread 0 is not issuing



Checkpoint 3: Multicores

- Two cores have private I-caches (read-only); the two I-caches share a common instruction memory port
- Two cores share a common D-cache interface
- Require a 2-port fair arbiter
  - if both cores need to access memory in the same cycle, one side is told to stall
  - the core that is stalled is guaranteed to go on the next "round."



Deliverables

- All project checkpoints due by 4/30
  - no late credit, partial credit by complete checkpoints
  - checkpoint 1 90 points
  - checkpoint 2 240 points
  - checkpoint 3 120 points
- Extra Credit (50 max)
  - Checkpoint 1 checked-off by 4/9 10 points
  - Checkpoint 1&2 checked-off by 4/21 20 points
  - Checkpoint 1&2&3 checked-off by 4/28 20 points