CS/ECE 752: Advanced Computer Architecture I

Source: XKCD

Prof. Matthew D. Sinclair

Multithreading

Slide History/Attribution Diagram:

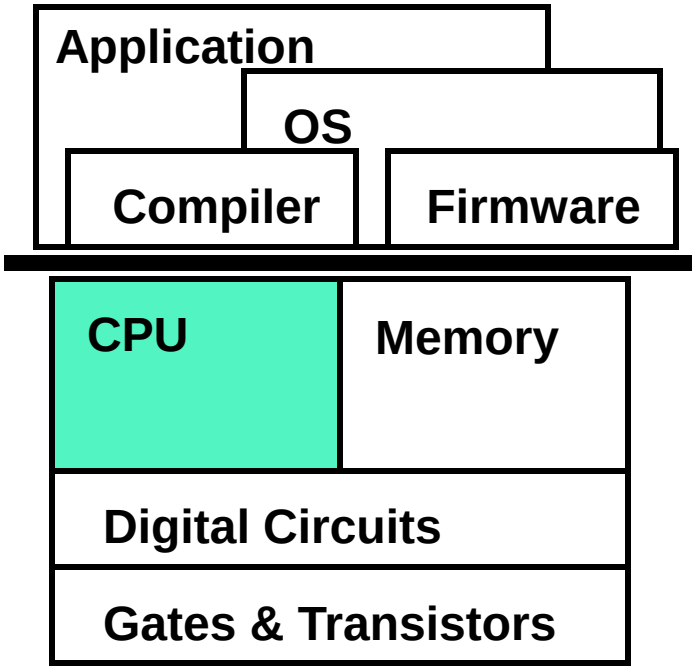
Announcements 10/30/24

- Weird (scam/phishing) emails from "me"
  - I will never ask you for iTunes gift cards, Zelle info, etc.
  - I will always email you from my UW account
- HW6 Due Friday
- HW5 grading ongoing
- Project Proposal grading inching closer to completion…
- **Potential schedule change – please vote in Piazza poll by Wednesday**
  - Enables all lectures to be in person but would push exam to 11/20
- **Potential HW change – please vote in Piazza poll by Thursday**
  - Would convert HW7 to ungraded practice problems

Forms of Parallelism

- Instruction-level Parallelism (ILP): Instructions which are proximate within program order executing together.
- Memory-level Parallelism (MLP): Memory requests which are proximate within program order overlapped.
- Thread-level Parallelism (TLP): Independent threads (only explicit ordering) running simultaneously.
- Task-level Parallelism: Collection of asynchronous tasks, not started/stopped together, data is shared loosely, dynamically.
- Data-level Parallelism (DLP): All tasks are similar – basically doing the same thing to multiple data items.

This Unit: Multithreading (MT)



- Why multithreading (MT)?
  - Utilization vs. performance
- Three implementations
  - Coarse-grained MT
  - Fine-grained MT
  - Simultaneous MT (SMT)
- MT for reliability
  - Redundant multithreading
- Multithreading for performance
  - Speculative multithreading

In-Class Assignment

- With a partner, answer the following questions:
  - What was the #1 factor in SMT becoming so successful commercially?
  - Why do most machines only use 2 threads for SMT?
  - What is/are the downside(s) to SMT?

- In 4 minutes we'll discuss as a class

better perf w/o cost of addit. cores
need bigger RF
thrashing b/w threads (e.g., $'s) + branch pred
contention for shared rs.
switching overhead → especially w/ Fetch
throughput vs latency

In-Class Assignment

- With a partner, answer the following questions:
  - What was the #1 factor in SMT becoming so successful commercially?
  - Why do most machines only use 2 threads for SMT?
  - What is/are the downside(s) to SMT?

- In 4 minutes we'll discuss as a class

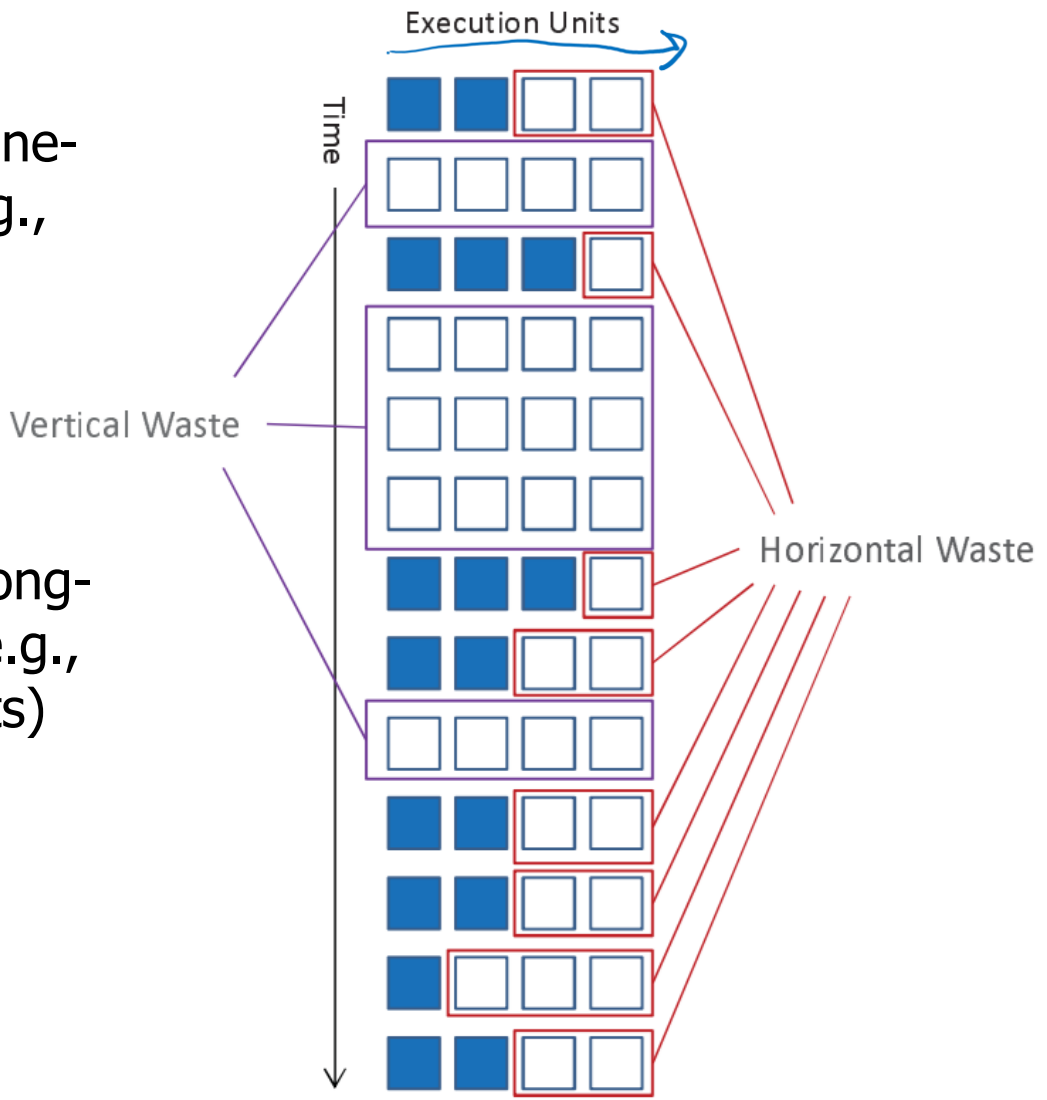
bigger RF
shared res. contention/thrashing
reg pressure → deeper pipeline
power
security of shared resources?

Performance And Utilization

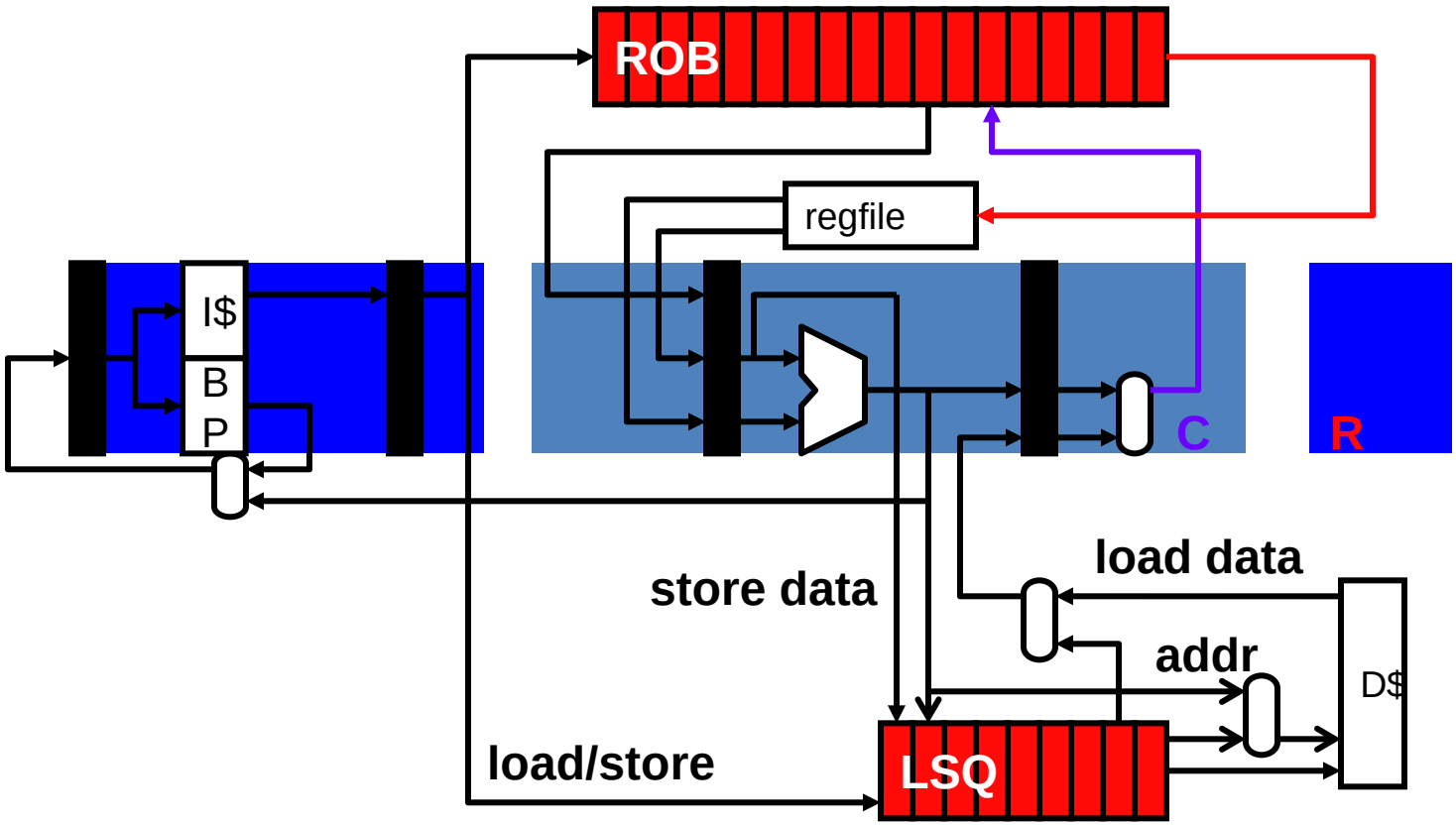
- Performance (IPC) important
- Utilization (actual IPC / peak IPC) important too, why?
  - Hardware costs
  - Scalability to many cores
- Even moderate superscalars (e.g., 4-way) not fully utilized
  - Average sustained IPC: 1.5–2 $\rightarrow$ <50% utilization
    - Mis-predicted branches
    - Cache misses, especially L2
    - Data dependences

Insight 1: Processors have waste…

- Horizontal Waste:
  - Low Utilization due to fine-grain dependences. (e.g., dependences between arithmetic instructions)
- Vertical Waste:
  - Low Utilization due to long-latency dependences (e.g., cache or memory events)



Insight 2: programs have unique bottlenecks

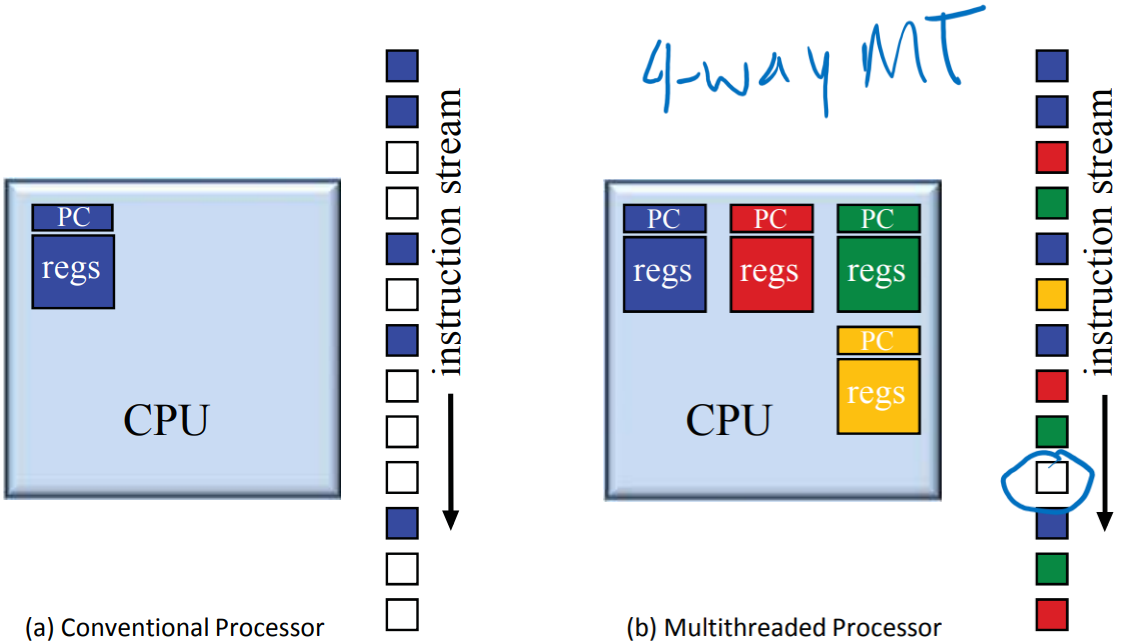


- Possible bottlenecks: Memory Latency, Fetch, FP unit bound, branch mispredictions, too many program dependences…

Multi-threading

- Single-threaded machine (e.g., page fault)
  - Only one thread at a time per CPU, context switch between them
- **Multi-threading (MT)**
  - Improve utilization by multiplexing multiple threads on single CPU
  - One thread cannot fully utilize CPU? Maybe 2, 4 (or 100) can

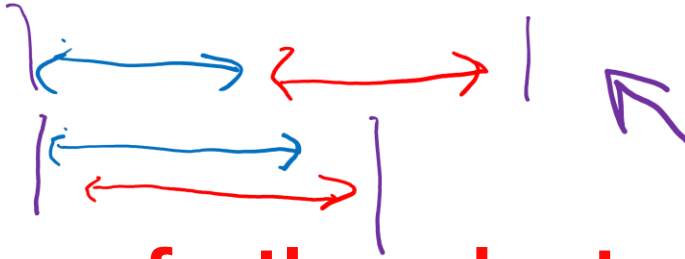
Question: Which state absolutely must be replicated for MT to work?



(a) Conventional Processor

(b) Multithreaded Processor

Latency vs Throughput



- **MT trades (single-thread) latency for throughput**
  - – Sharing processor degrades latency of individual threads
  - \+ But improves aggregate latency of both threads
  - \+ Improves utilization
- Example
  - Thread A: individual latency=10s, latency with thread B=15s
  - Thread B: individual latency=20s, latency with thread A=25s
  - Sequential latency (first A then B or vice versa): 30s
  - Parallel latency (A and B simultaneously): 25s
  - – MT slows each thread by 5s
  - \+ But improves total latency by 5s
- **Different workloads have different parallelism**
  - SpecFP has lots of ILP (can use an 8-wide machine)
  - Server workloads have TLP (can use multiple threads)

10 + 20 = 30 s

max(15, 25) = 25s

MT Implementations: Similarities

- How do multiple threads share a single processor?
  - Different sharing mechanisms for different kinds of structures
  - Depend on what kind of state structure stores

- **Persistent hard state (aka "context")**: PC, registers
  - Replicated
- **No state**: ALUs
  - Dynamically shared
- **Persistent soft state**: caches, bpred , LSQs, TLBs
  - Dynamically partitioned
    - TLBs need ASIDs, caches/bpred tables don't (and BTB?)
  - Exception: **ordered "soft" state** (BHR, RAS) is replicated
- **Transient state**: pipeline latches, ROB, RS

— starting to change (Spectre + friends)

MT Implementations: Differences

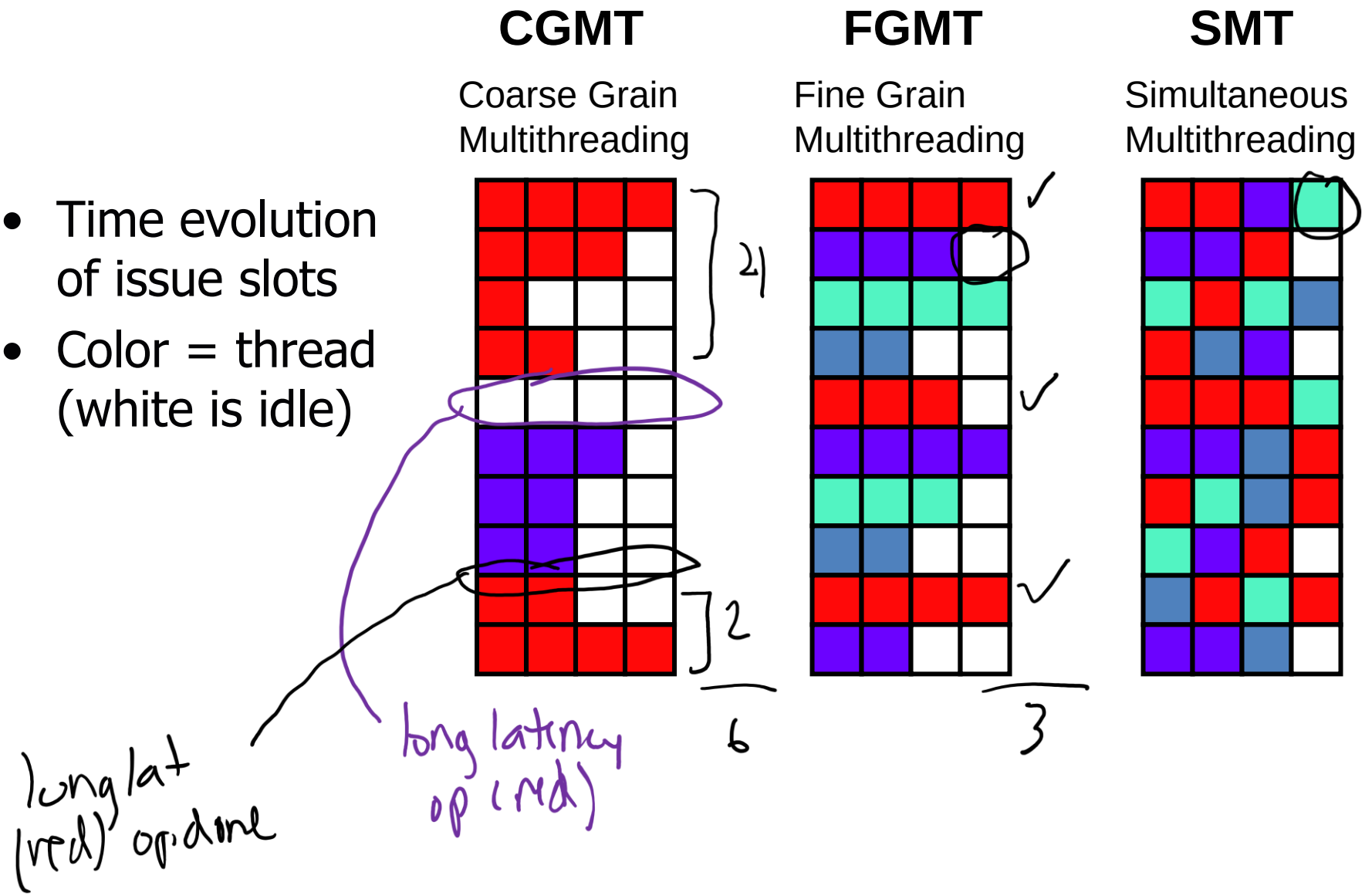
- Main question: **thread scheduling policy**
  - When to switch from one thread to another?
- Related question: **pipeline partitioning**
  - How exactly do threads share the pipeline itself?

- Choice depends on
  - What kind of latencies (specifically, length) you want to tolerate
  - How much single thread performance you are willing to sacrifice

- Three designs
  - Coarse-grain multithreading (CGMT)
  - Fine-grain multithreading (FGMT)
  - Simultaneous multithreading (SMT)

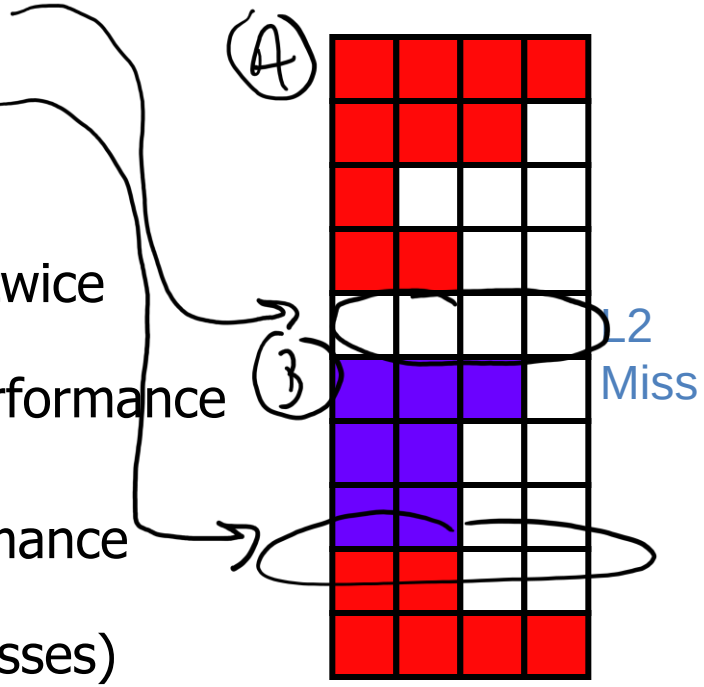
The Standard Multithreading Picture

- Time evolution of issue slots
- Color = thread (white is idle)



Coarse-Grain Multithreading (CGMT)

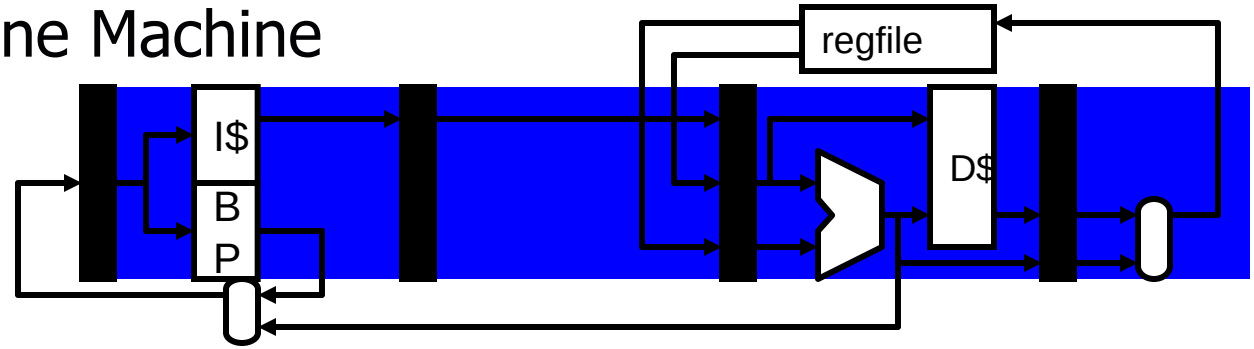
- Thread scheduling policy:
  - Designate a "preferred" thread (e.g., thread A)
  - Switch to thread B on thread A L2 miss
  - Switch back to A when A L2 miss returns
- Pipeline partitioning
  - None, flush on switch
    - Can't tolerate latencies shorter than twice pipeline depth
  - Need short in-order pipeline for good performance
- Tradeoffs:
  - \+ Sacrifices very little single thread performance (does it though?)
  - – Tolerates only long latencies (e.g., L2 misses)



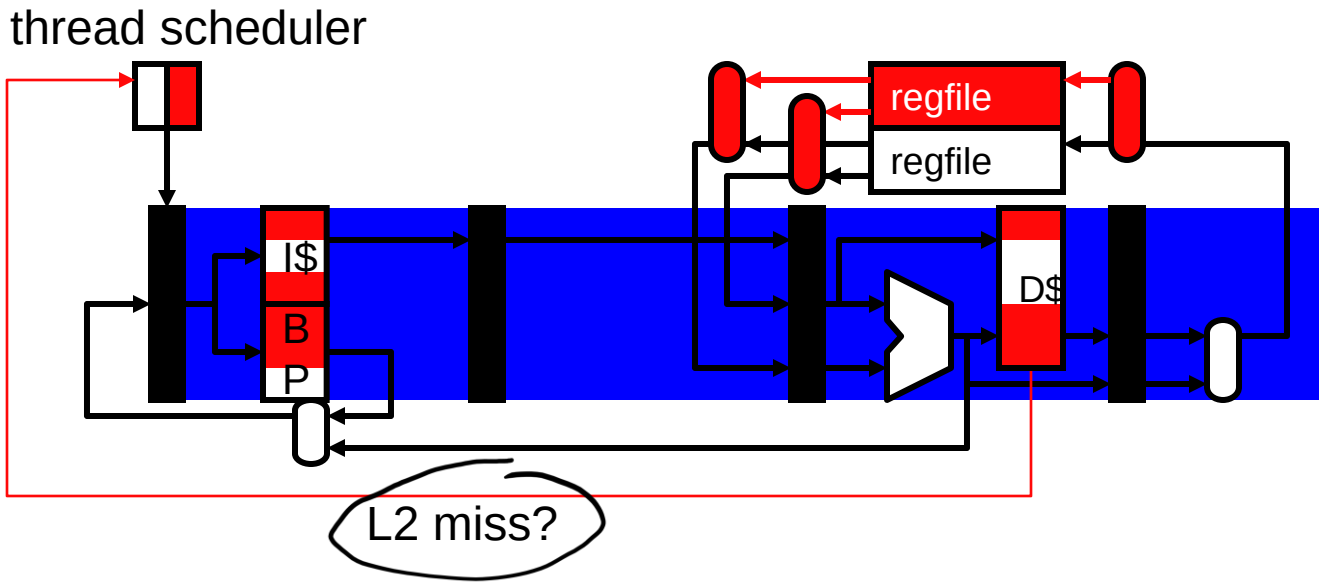
  - Example: IBM Northstar/Pulsar (1998)
    - Switches on L1 cache miss
  - Very uncommon now – why?

CGMT

- Baseline Machine



- Extensions for CGMT (red: thread B)

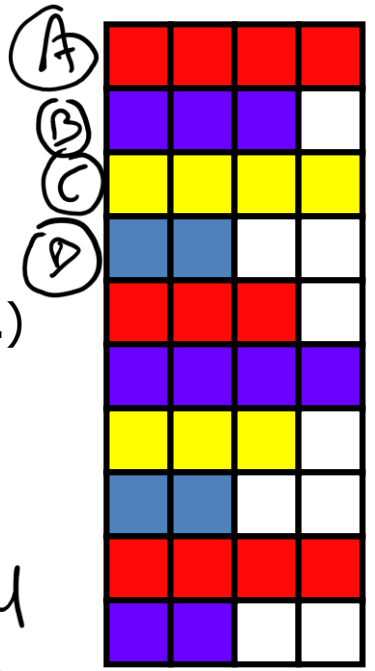


Fine-Grain Multithreading (FGMT)

- Thread scheduling policy
  - Switch threads every cycle (round-robin), L2 miss or no
- Pipeline partitioning
  - Dynamic, no flushing
  - Length of pipeline doesn't matter
- Tradeoffs:
  - – Sacrifices significant single thread performance
  - + Tolerates all latencies (e.g., L2 misses, mispred. branches..)
  - – Need a lot of threads (reg files size, #ports same though)
- Extreme example: Denelcor HEP (1981-1985)
  - So many threads (100+), it didn't even need caches
  - Failed commercially (slightly ahead of its time, cost/performance)
- Semi-success: Sun Niagara (aka Ultrasparc T1)
  - Four threads x Register windows → lots of registers

FGMT

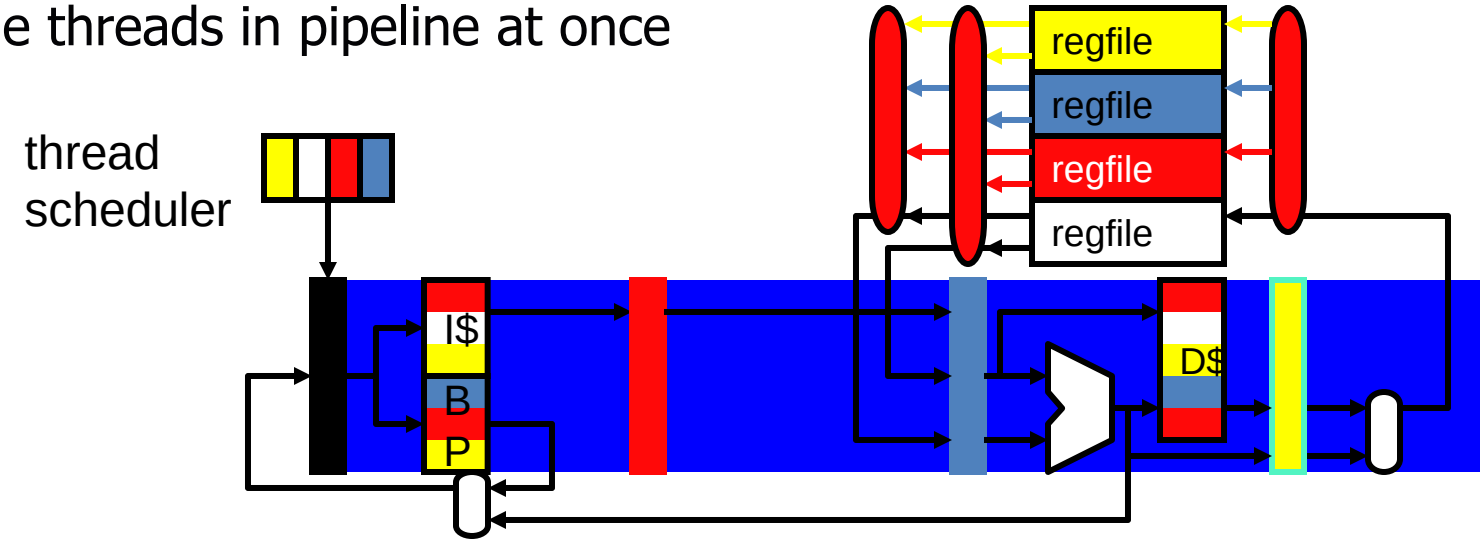
Fine Grain Multithreading



GPU now

Fine-Grain Multithreading

- FGMT
  - (Many) more threads
  - Multiple threads in pipeline at once

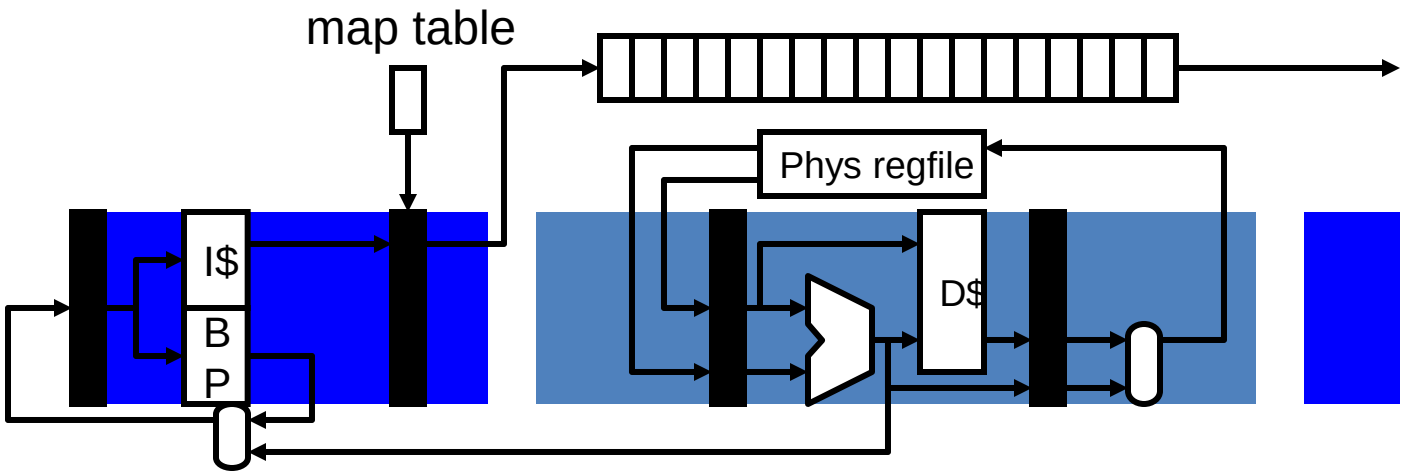


- Do we assume that we always have multiple threads?
  - If yes: Get rid of bypass (get rid of branch prediction?) –
    - Use this to increase frequency or more cores?
  - If no: Must keep bypass/bpred etc.

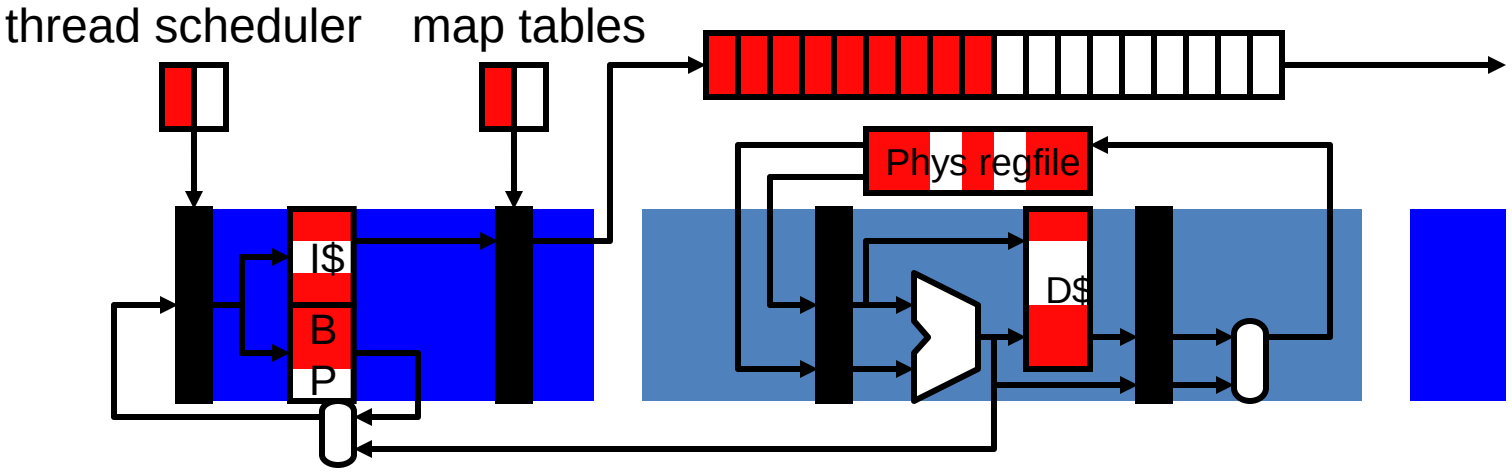
Simultaneous Multithreading (SMT)

- Motivation: Multithread an out-of-order machine?
  - Don't want to give up performance benefits
  - Don't want to give up natural tolerance of D$ (L1) miss latency
- **Simultaneous multithreading (SMT)**
  - Thread scheduling policy
    - Round-robin (just like FGMT)
  - Pipeline partitioning
    - Dynamic, hmmm…
- Tradeoffs:
  - \+ Tolerates all latencies (e.g., L2 misses, mispredicted branches)
  - ± Sacrifices some single thread performance
- Example: Pentium4 (hyper-threading): 5-way issue, 2 threads (and every design afterwards)
- Another example: Alpha 21464: 8-way issue, 4 threads

Simultaneous Multithreading (SMT)



- SMT
  - Replicate map table, share physical register file. ROB?, LSQ?



Implementation Issues for SMT

- Good: OOO is a great fit for SMT…
  - Issue logic doesn't change (surprising?)
  - Reason: Once you rename registers, no reason to further distinguish threads in issue…
- Bad:
  - Large map table and physical register file
    - #map-table-entries = (**#threads** * #arch-regs)
    - #phys-regs = (**#threads** * #arch-regs) + #in-flight insns
  - Per-thread pipeline-flush
- Upshot: Probably less % increase to implement SMT on OOO (compared to FGMT on in-order)

SMT Resource Partitioning

- How are ROB/LSQ, RS partitioned in SMT?
  - Depends on what you want to achieve
- **Static partitioning**
  - Divide ROB/LSQ, RS into T static equal-sized partitions
  + Ensures that low-IPC threads don't starve high-IPC ones
    - Low-IPC threads stall and occupy ROB/LSQ, RS slots
  – Low utilization
- **Dynamic partitioning**
  - Divide ROB/LSQ, RS into dynamically resizing partitions
  - Let threads fight amongst themselves
  + High utilization
  – Possible starvation

- flush logic more complex

Control Speculation Contention

- Bad:
  - Must share total state between multiple threads
  - Fetch from multiple threads at the same time -> multiple contexts for branch prediction in the same cycle.
- Good:
  - Less need for control speculation?
  - Speculate less far in each thread
  - Get ILP from threads rather than large instruction window

- (contrast with FGMT+inorder – might not need it at all)

Fetch Multiple Lines? [Tullsen 1996]

- Which threads to fetch from
  - **RR.1.8:** One thread fetches up to 8 instructions at a time
  - **RR.2.4 (RR.4.2):** Two (four) threads each statically getting four (two) instructions at a time
  - **RR.2.8:** Fetch for two threads fetches up to 8 instructions

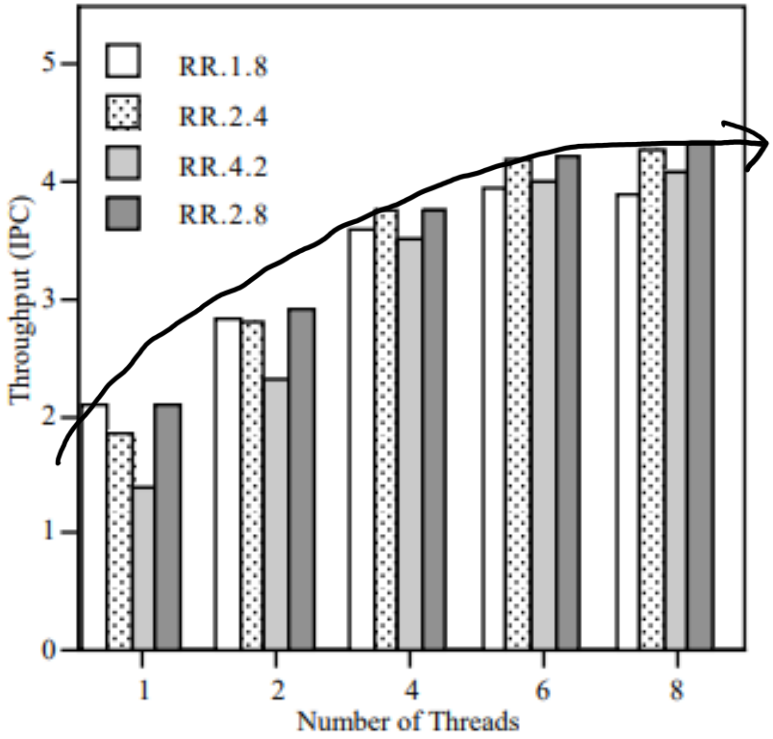


Figure 4: **Instruction throughput for the different instruction cache interfaces with round-robin instruction scheduling.**

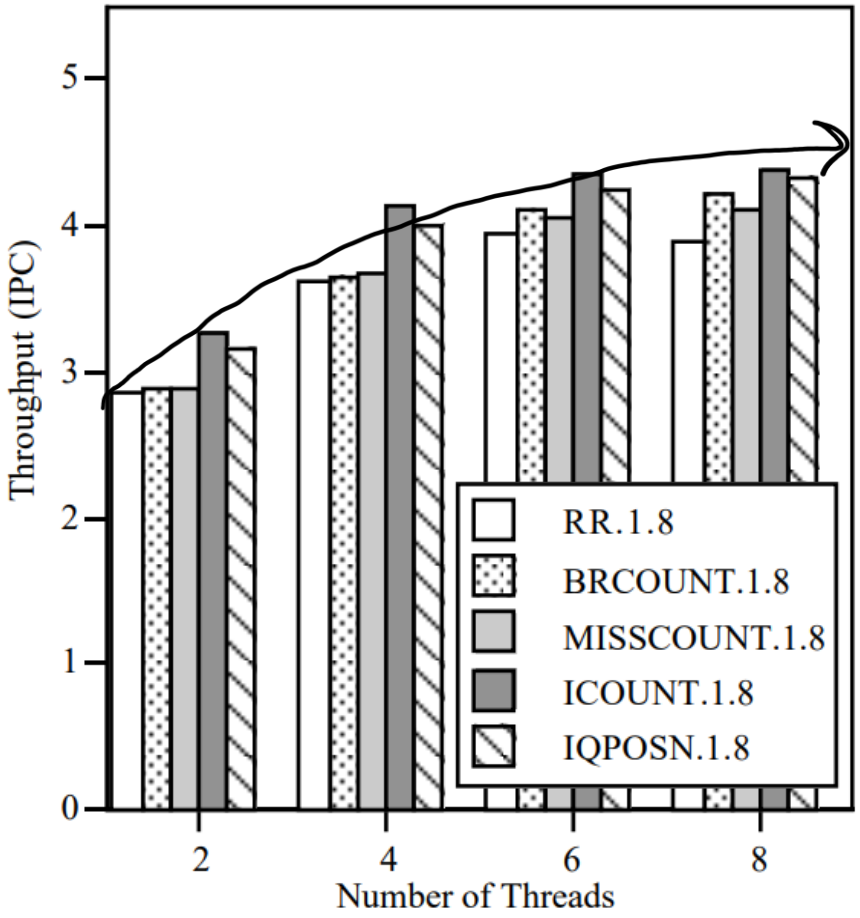
Spec 92 Benchmarks! :)

How would you decide which thread?

- Assume:
  - "1.8" scheme
  - Dynamic resource partitioning
- Considerations:
  - How speculative is the thread? (avoid over-fetching unlikely thread)
  - How much does it cost to fetch from a thread? (avoid fetching for a thread that is blocked for too long)

Thread Selection [Tullsen 1996]

- Which thread to give priority?
  - **BRCOUNT:** Least likely to be on a wrong path, for least waste (counting branch instructions in flight), favoring those with the fewest branches.
  - **MISSCOUNT:** priority to those threads that have the fewest outstanding D cache misses (don't want clogger-threads)
  - **ICOUNT:** Thread with fewest instructions in decode, rename, and the instruction queues. (prevents clogging, favors high ILP threads)
  - **IQPOSN:** Priority to threads with youngest instruction in IQ (poor man's ICOUNT – no counter per thread).



Handling Long Latency Loads

- Long-latency (L2/L3 miss) loads are a problem in a single-threaded processor
  - Block instruction/scheduling windows and cause the processor to stall
- In SMT, a long-latency load instruction can block the window for ALL threads
  - i.e. reduce the memory latency tolerance benefits of SMT

Brown and Tullsen, "Handling Long-latency Loads in a Simultaneous Multithreading Processor," MICRO 2001.

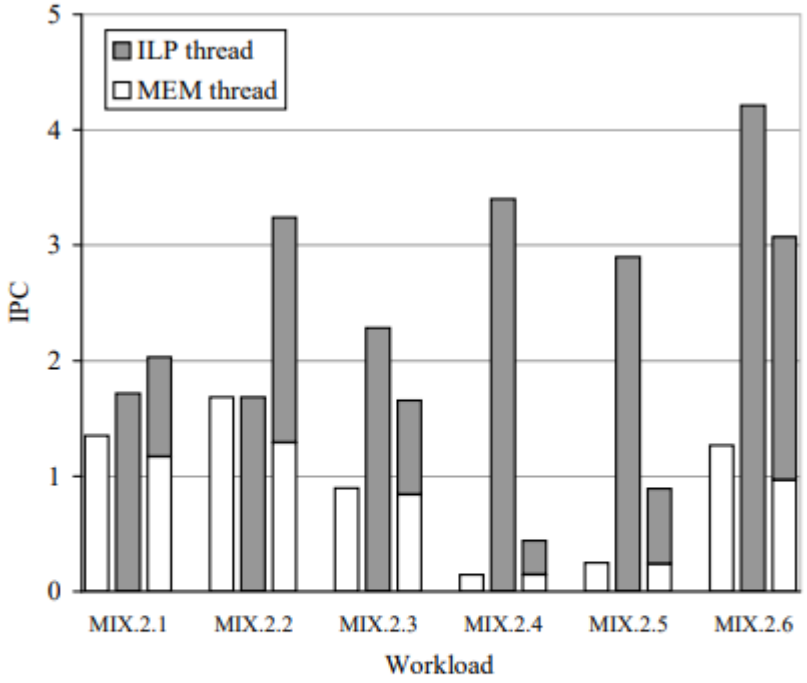


Figure 1. The performance of several two-thread mixes of memory-bound and ILP-bound applications. The stacked bars represent two-thread runs, the single bars represent the single-thread runs for the same two benchmarks.

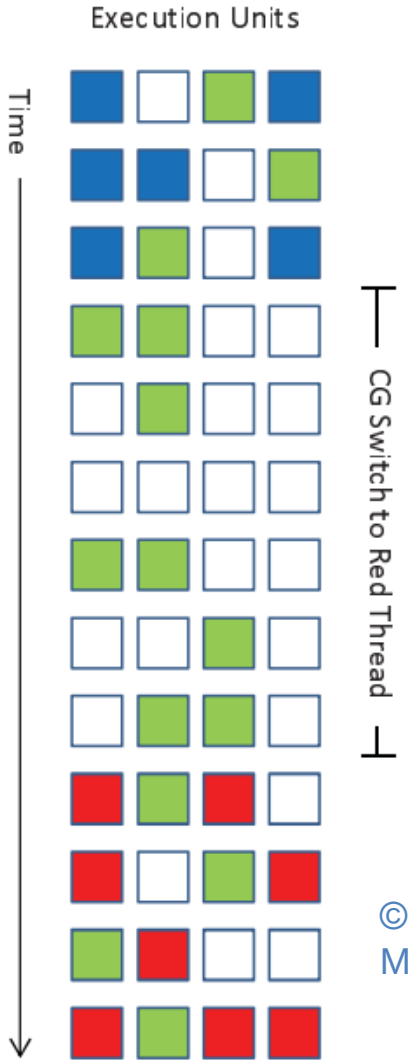
Following two slides courtesy of Multithreading Lecture by Onur Mutlu

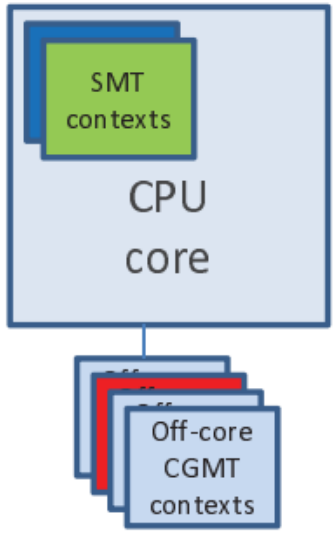
Proposed Solutions to Long Latency Loads

- Idea: Flush the thread that incurs an L2 cache miss
  - Brown and Tullsen, "Handling Long-latency Loads in a Simultaneous Multithreading Processor," MICRO 2001.
- Idea: Predict load miss on fetch and do not insert following instructions from that thread into the scheduler
  - El-Moursy and Albonesi, "Front-End Policies for Improved Issue Efficiency in SMT Processors," HPCA 2003.
- Idea: Partition the shared resources among threads so that a thread's long latency load does not affect another
  - Raasch and Reinhardt, "The Impact of Resource Partitioning on SMT Processors," PACT 2003.
- Idea: Predict if (and how much) a thread has MLP when it incurs a cache miss; flush the thread after its MLP is exploited
  - Eyerman and Eeckhout, "A Memory-Level Parallelism Aware Fetch Policy for SMT Processors," HPCA 2007.

Hybrid Models

- Something in between: Balanced MT [2004]?
- Some number of simultaneous threads + some number of coarse grain threads.
  - Simultaneous threads hide fine-grain latencies
  - Coarse grain threads get swapped in to hide long latencies.
- Drawbacks: OS sees *lots* of threads...





© "Multithreading Architecture" Mario Nemirovsky, Dean M. Tullsen

Why not MT: Cache interference? (thrashing)

- Irony: Reason for doing MT was to increase memory level parallelism to hide accesses to memory, but…
  - Drawback of having multiple threads is that the working set size is sum over all threads -> more contention -> more misses
- Best case for SMT: Working set does not fit in caches
  - MT increases memory-level parallelism (MLP)
    - Helps most for big "server" workloads
- Working set of at least one thread fits in caches
  - Where do threads come from?
    - Single-program multiple threads (threads work together)] SIMD
      - Maybe same insns & data?! (less contention)
    - Multi-programmed (random unrelated applications)
      - Different instructions & data! (bad for threads with locality)

Energy Implications of MT

- Is MT (of any kind) energy efficient?
  - Static energy?
    - Didn't add too much hardware, better than adding more cores
    - Higher utilization, so can "turn off" machine quicker
    - Seems to be yes…
  - Dynamic energy?
    - Again, not too many additional structures, only small overhead
    - But additional cache pressure… so some debate here
  - Overall probably a win for energy

Announcements 11/1/24

- HW6 Due Saturday
  - See Piazza for bug after Wednesday lecture with se.py command
  - I also posted several (hopefully) helpful tips about using HTCondor
- HW5 grading done and will be posted sometime today
- Project Proposal grading stalled …
  - I will have to try and complete these during my trip
  - But unlikely to be done before Monday or Tuesday
- **Schedule change enacted**
  - Enables all lectures to be in person, midterm 2 now 11/20
  - All assignments and dates updated in Canvas+course website
  - **No class next week Monday or Wednesday**
- **HW7 will remain graded per vote on Piazza**
  - Posted, due Saturday 11/9/24

MT for Reliability?

- Can multithreading help with reliability?
  - Design bugs/manufacturing defects? No
  - Gradual defects, e.g., thermal wear? No
  - Transient errors? Yes
    - Caused by cosmic rays (e.g., neutrons)
    - Leads to transient changes in wires and state (e.g., 0/1)
- **Background: lock-step execution (DMR, TMR…)**
  - Two processors run same program and same time
  - Compare cycle-by-cycle; flush both and restart on mismatch
- **Staggered redundant multithreading (SRT)**
  - Run two copies of program at a slight stagger
  - Compare results, difference? Flush both copies and restart
  - Significant performance overhead
  - Other ways of doing this (e.g., DIVA – inorder checker at commit)

MT for Prefetching? ("runahead")

- Idea: Pre-execute a piece of the (pruned) program solely for prefetching data
  - Only need to distill pieces that lead to cache misses
- Speculative thread: Pre-executed program piece can be considered a "thread"
- Speculative thread can be executed
  - On a separate processor/core
  - On a separate hardware thread context (SMT)
  - On the same thread context in idle cycles (during cache misses)

Helper Threading for Prefetching

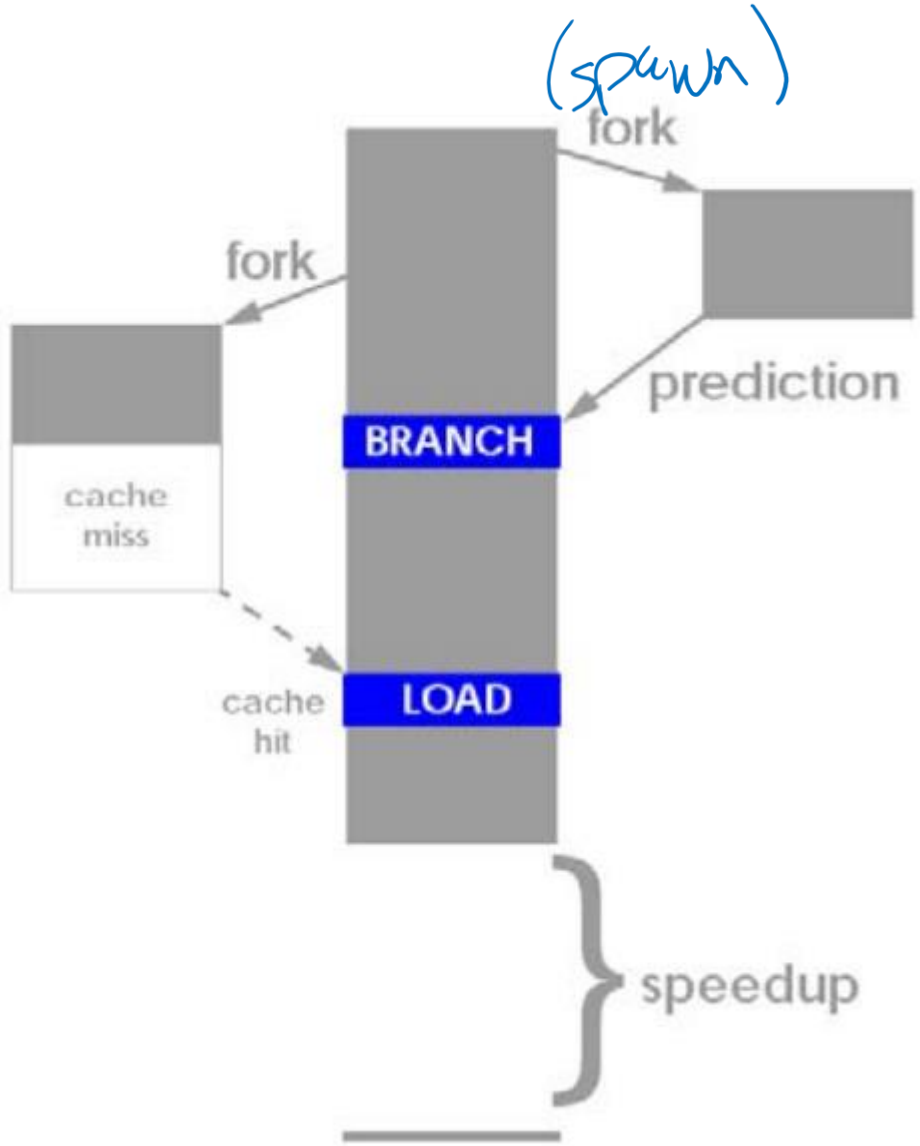
- How to construct the speculative thread:
  - Software based pruning and "spawn" instructions
  - Hardware based pruning and "spawn" instructions
  - Use the original program (no construction), but
    - Execute it faster without stalling and correctness constraints

- Speculative thread
  - Needs to discover misses before the main program (profiling)
    - Avoid waiting/stalling and/or compute less
    - Maybe with some combination of: Branch prediction, value prediction, only address generation computation

Generalized Thread-Based Pre-Execution

- Also works for branch prediction as well
  - Slice the program so that only instructions critical for a hard-to-predict branch are executed on a separate thread.

- E.g., "Execution-based Prediction Using Speculative Slices", Zilles and Sohi, ISCA 2001



SMT vs. CMP

- If you wanted to run multiple threads would you build a…
  - Chip multiprocessor (CMP): multiple separate pipelines?
  - A multithreaded processor (SMT): a single larger pipeline?
- **Both will get you throughput on multiple threads**
  - CMP will be simpler, possibly faster clock (from core's perspective)
  - SMT will get you better performance (IPC) on a single thread
    - SMT is basically an ILP engine that converts TLP to ILP
    - CMP is mainly a TLP engine
- **Again, do both**
  - Sun's Niagara (UltraSPARC T1)
  - 8 processors, each with 4-threads (fine-grained threading)
  - 1Ghz clock, in-order, short pipeline (6 stages)
  - Designed for power-efficient "throughput computing"

revisit w/ GPUs

Is FGMT popular today in server context?

- Intuition: Massive parallelism in server context coming from many independent requests (think webserver)

- But Out-of-order cores still king… why?
  - Single core performance matters, even in context of server machines
  - Request latency is hugely important!

Ignores hyperscalars having ∞ GPUs

Multithreading Summary

- Latency vs. throughput
- Partitioning different processor resources
- Three multithreading variants
  - Coarse-grain: no single-thread degradation, but long latencies only
  - Fine-grain: other end of the trade-off
  - Simultaneous: fine-grain with out-of-order
- Multithreading vs. chip multiprocessing

Research: Speculative Multithreading

- **Speculative multithreading**
  - Use multiple threads/processors for ILP
  - Speculatively parallelize sequential loops
    - CMP processing elements (called PE) arranged in logical ring
    - Compiler or hardware assigns iterations to consecutive PEs
    - Hardware tracks logical order to detect mis-parallelization
  - Techniques for doing this on non-loop code too
  - Effectively chains ROBs of different processors into one big ROB
    - Global commit "head" travels from one PE to the next
    - Mis-speculation flushes entire PEs
  - Also known as split-window or "Multiscalar"
- Not commercially available yet…
  (Farewell, Sun Rock, we hardly knew ye)