CS302

Multi-threaded Processors

Spring 2025

Arkaprava Basu & Babak Falsafi

parsa.epfl.ch/course-info/cs302

Adapted from slides originally developed by Profs. Falsafi, Fatahalian, Mowry, Wenisch, Etsion and Weiser of CMU, Michigan and Technion

Where are We?

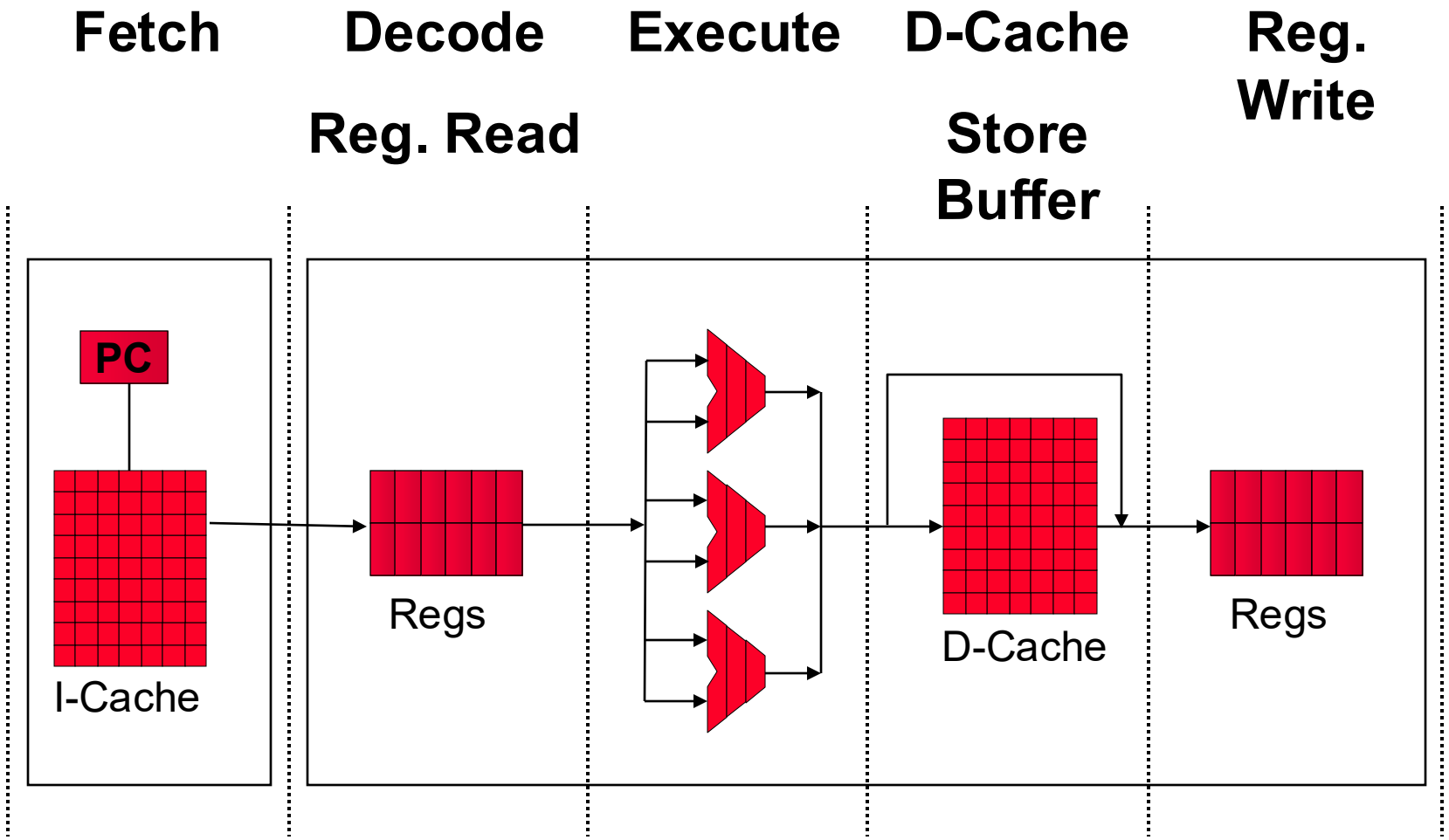
M	T	W	T	F
17-Feb	18-Feb	19-Feb	20-Feb	21-Feb
24-Feb	25-Feb	26-Feb	27-Feb	28-Feb
3-Mar	4-Mar	5-Mar	6-Mar	7-Mar
10-Mar	11-Mar	12-Mar	13-Mar	14-Mar
17-Mar	18-Mar	19-Mar	20-Mar	21-Mar
24-Mar	25-Mar	26-Mar	27-Mar	28-Mar
31-Mar	1-Apr	2-Apr	3-Apr	4-Apr
7-Apr	8-Apr	9-Apr	10-Apr	11-Apr
14-Apr	15-Apr	16-Apr	17-Apr	18-Apr
21-Apr	22-Apr	23-Apr	24-Apr	25-Apr
28-Apr	29-Apr		1-May	2-May
5-May	6-May	7-May	8-May	9-May
12-May	13-May	14-May	15-May	16-May
19-May	20-May	21-May	22-May	23-May
26-May	27-May	28-May	29-May	30-May

- HW Multithreading
  - Motivation
  - Various forms
- Next Tuesday:
  - Intro to GPUs

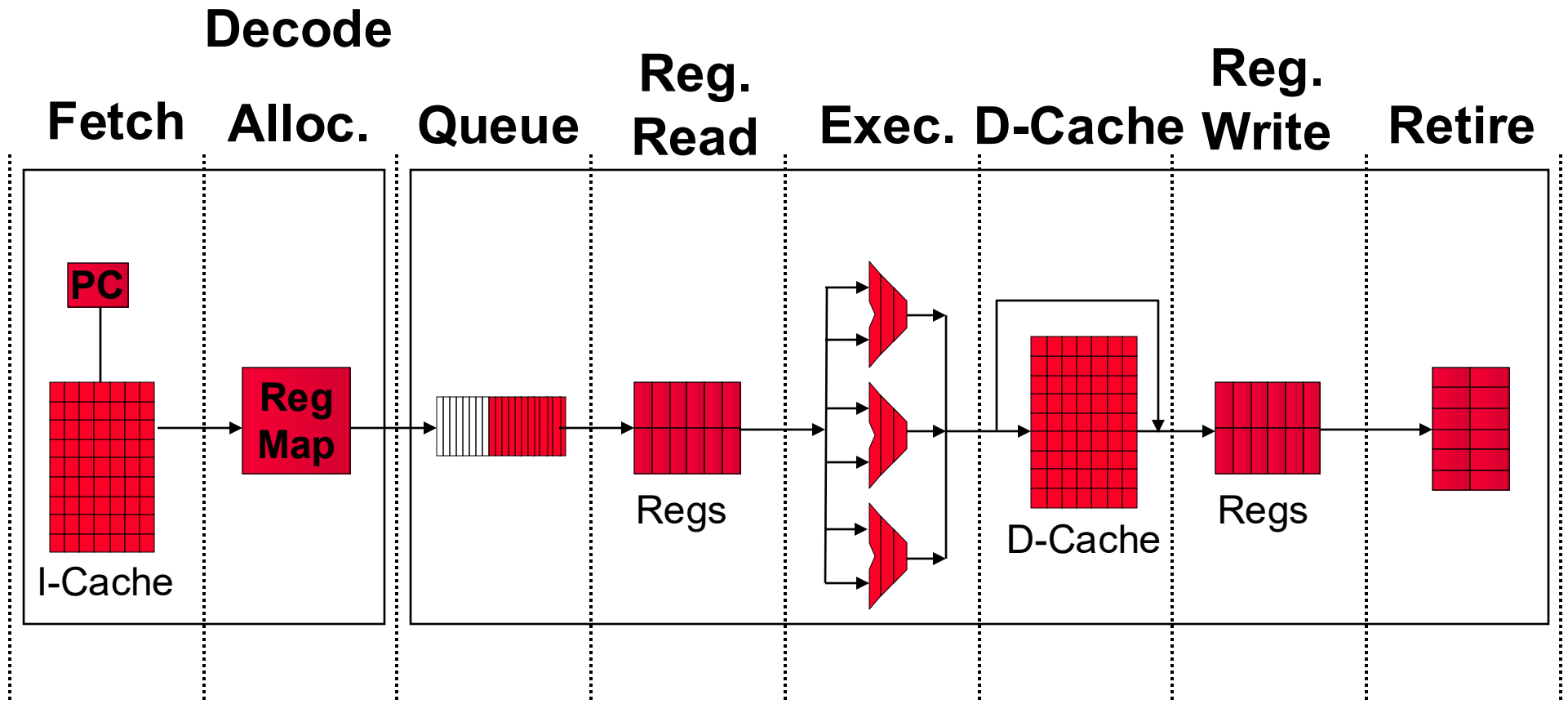
Hardware Multithreading

- Arithmetic units in CPUs are often not busy
  - Pipeline bubbles
  - Long-latency memory accesses
  - Exceptions due to I/O or syscalls
- We often switch threads a lot
  - Threaded programming is popular
  - Mobile apps heavily use threading to overlap compute with I/O
  - Google claims there are 10,000 threads per socket across workloads
- Can we keep thread context around in hardware?

Reminder: Basic Scalar Pipeline



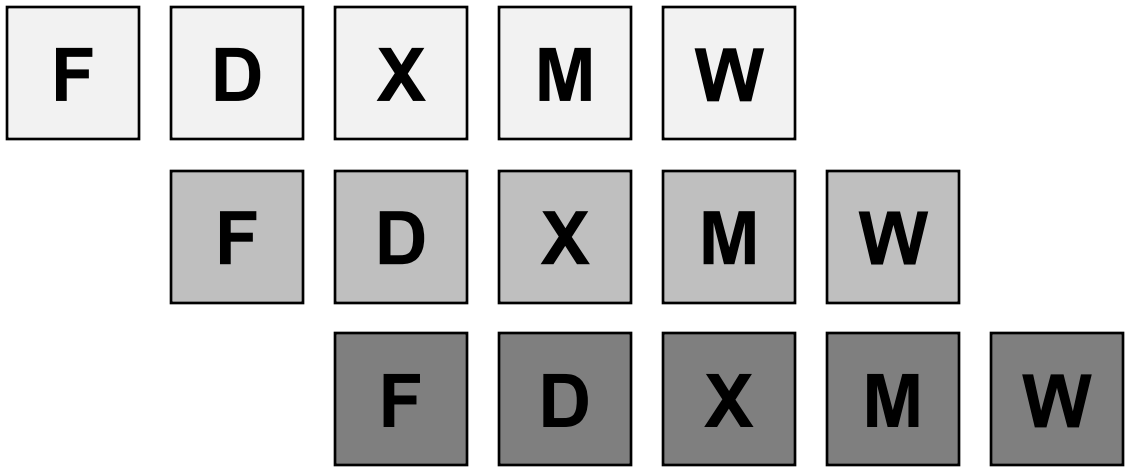
Reminder: Basic Superscalar Pipeline



Review: Why Pipeline?

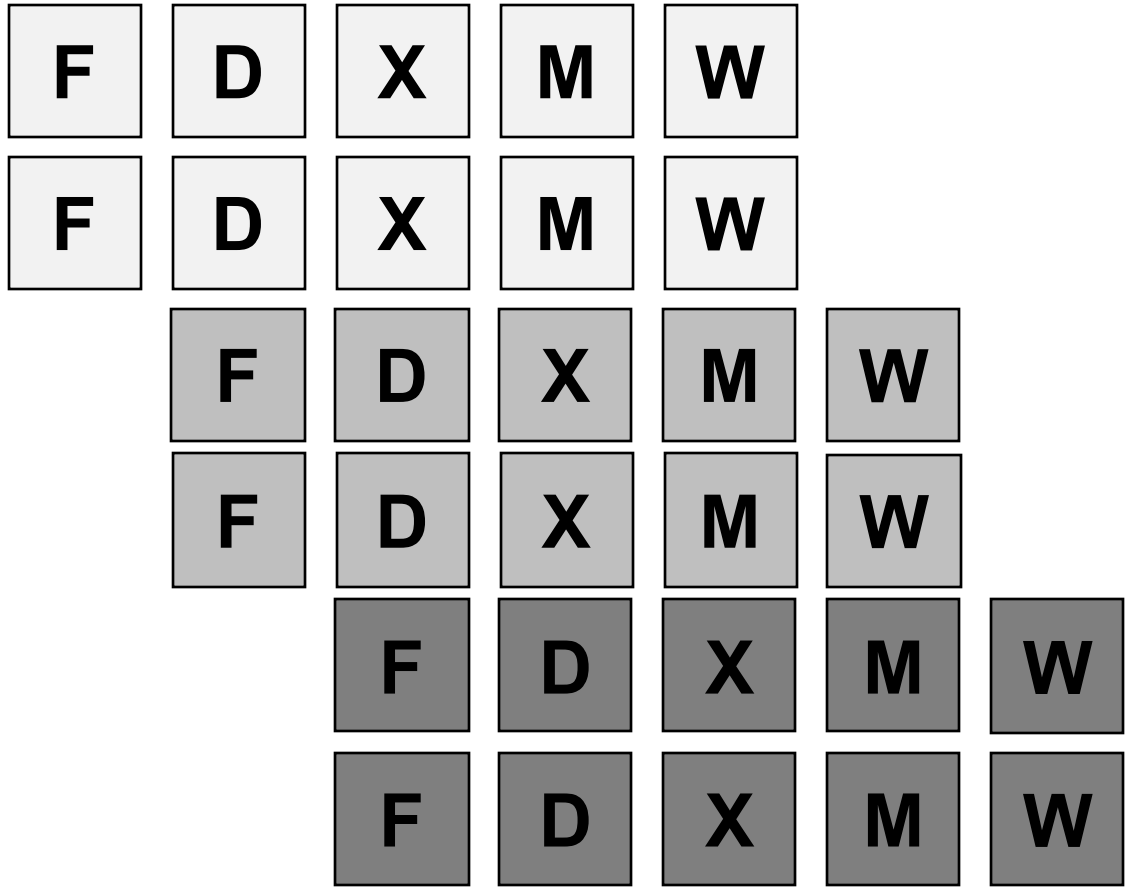
- First CPUs were “multi-cycle”
  - Executed 1 instruction at a time, varied # of cycles

- Pipeline allows multiple instructions to be in-flight
  - Perfect case reduces instructions per cycle (IPC) to 1
  - Assuming 5 stages (F, D, X, M, W):



How does Superscalar Affect IPC?

- Assuming a perfect pipeline, scalar IPC = 1
- Duplicating the pipeline halves it, IPC = 2



Pipeline Bubbles Decrease IPC

- Reason 1: structural hazards
  - Not enough resources are available
  - Instructions queue, waiting for resources to free up

- For example
  - 2-way in-order superscalar processor
  - One cache port (one load or store per cycle)
  - If two loads align in same cycle, second must wait

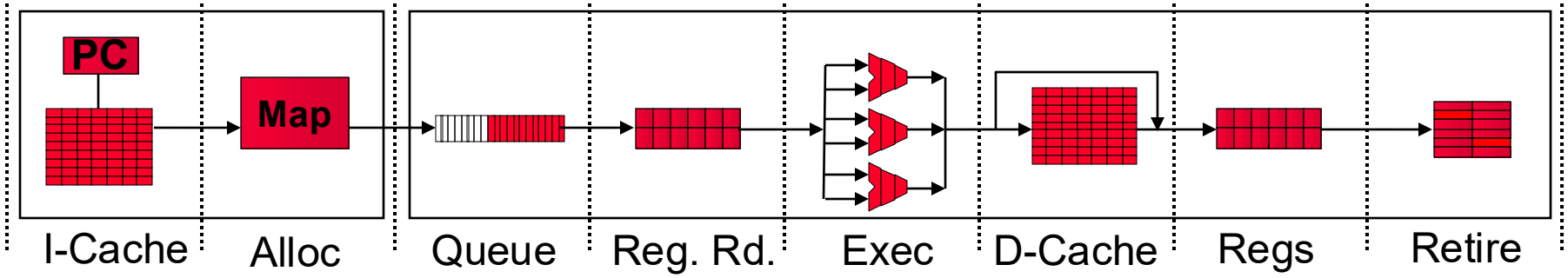
Structural Hazards Ex.

```
for (i =0; i < N; i++)
  a[i] += (b[i] + c[i]);
```

- The loop iterates N times
- Increments each element of `a[i]`, by `b[i]` and `c[i]`

```
; a[]➔r2 , b[]➔r3
; c[]➔r4 , &a[N]->r8
; r5, r6, r7, r8, r9 temp
loop:
  lw   r5 , 0(r2)
  lw   r6 , 0(r3)
  lw   r7 , 0(r4)
  add  r5 , r5 , r6
  add  r5 , r5 , r7
  sw   r5 , 0(r2)
  addi r2 , r2 , 4
  addi r3 , r3 , 4
  sub  r9, r8, r2 ; iter count
  addi r4 , r4 , 4
  bne  r9, loop
```

Structural Hazards Ex.



lw r5
lw r6

lw r5
lw r6

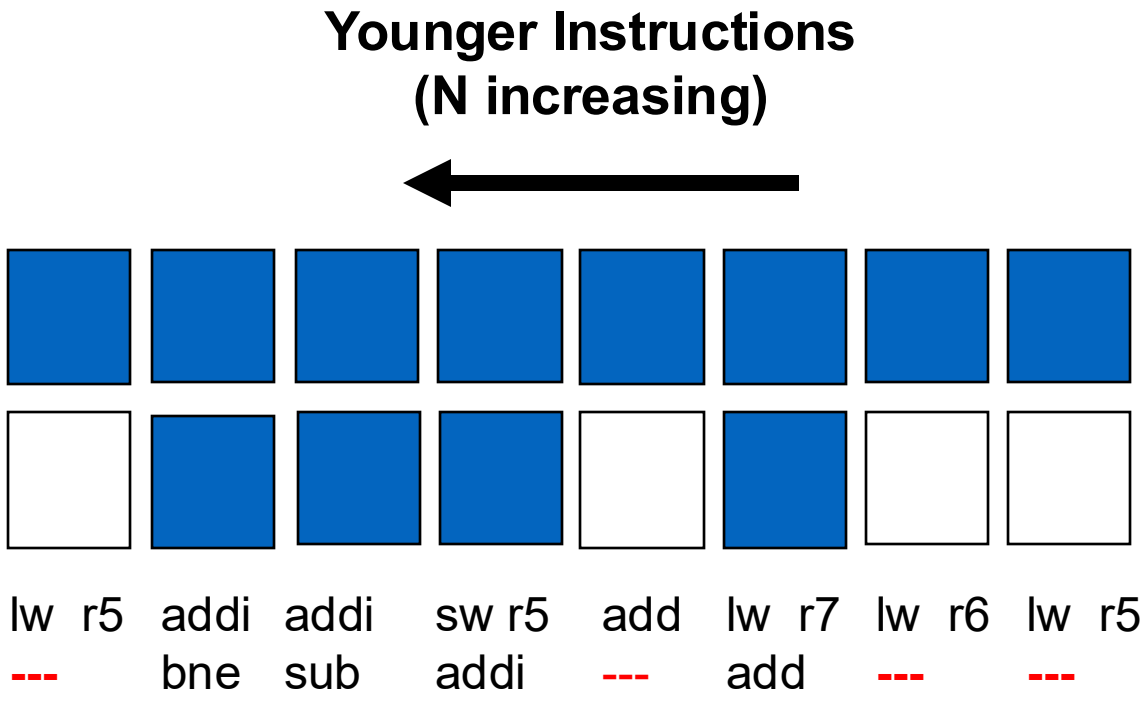
lw r5
lw r6

lw r5
lw r6

lw r5
lw r6

lw r5
bubble
lw r6
bubble

Pipeline View: Retire Stage




```
loop:
  lw   r5 , 0(r2)
  lw   r6 , 0(r3)
  lw   r7 , 0(r4)
  add  r5 , r5 , r6
  add  r5 , r5 , r7
  sw   r5 , 0(r2)
  addi r2 , r2 , 4
  addi r3 , r3 , 4
  sub  r9, r8, r2
  addi r4 , r4 , 4
  bne  r9, loop
```

- Bubbles caused by `lw r5`, `lw r6` and `add` propagate
  - `lw` bubbles are due to a single cache port
  - `sw` waits for add

Exercise: IPC Reduction w. Bubbles

- What is the average IPC when running this loop on our 2-way in-order superscalar?

```
loop:
  lw   r5 , 0(r2)
  lw   r6 , 0(r3)
  lw   r7 , 0(r4)
  add  r5 , r5 , r6
  add  r5 , r5 , r7
  sw   r5 , 0(r2)
  addi r2 , r2 , 4
  addi r3 , r3 , 4
  sub  r9, r8, r2
  addi r4 , r4 , 4
  bne  r9, loop
```

Exercise: IPC Reduction w. Bubbles

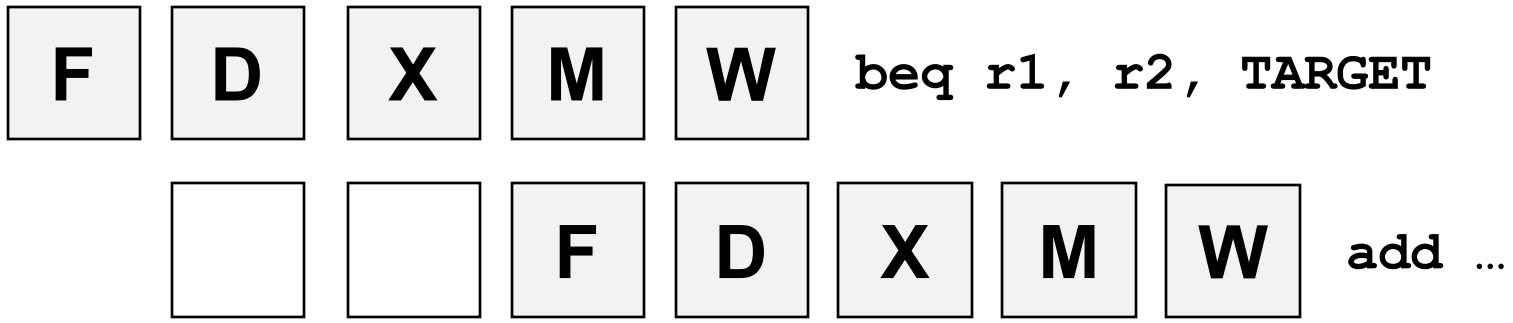
```
loop:
  lw   r5 , 0(r2)
  lw   r6 , 0(r3)
  lw   r7 , 0(r4)
  add  r5 , r5 , r6
  add  r5 , r5 , r7
  sw   r5 , 0(r2)
  addi r2 , r2 , 4
  addi r3 , r3 , 4
  sub  r9, r8, r2
  addi r4 , r4 , 4
  bne  r9, loop
```

- What is the average IPC when running this loop on our 2-way in-order superscalar?
  - Answer: 1.6
    - $\text{IPC} = \frac{\#\ instrs}{\#\ cycles} = \frac{11}{7} = 1.6$
    - Needs 7 cycles because: first two loads are serialized, second add is serialized

Pipeline Bubbles Continued

- Reason 2: Branch control flow
  - Predict branch direction to keep fetching instructions

- Waiting for the real direction introduces bubbles
  - 2 cycles at a minimum, with a shallow pipeline
  - Real pipelines are **much** deeper (14+ cycles)
    - Branch prediction accuracy paramount



Pipeline Bubbles Continued

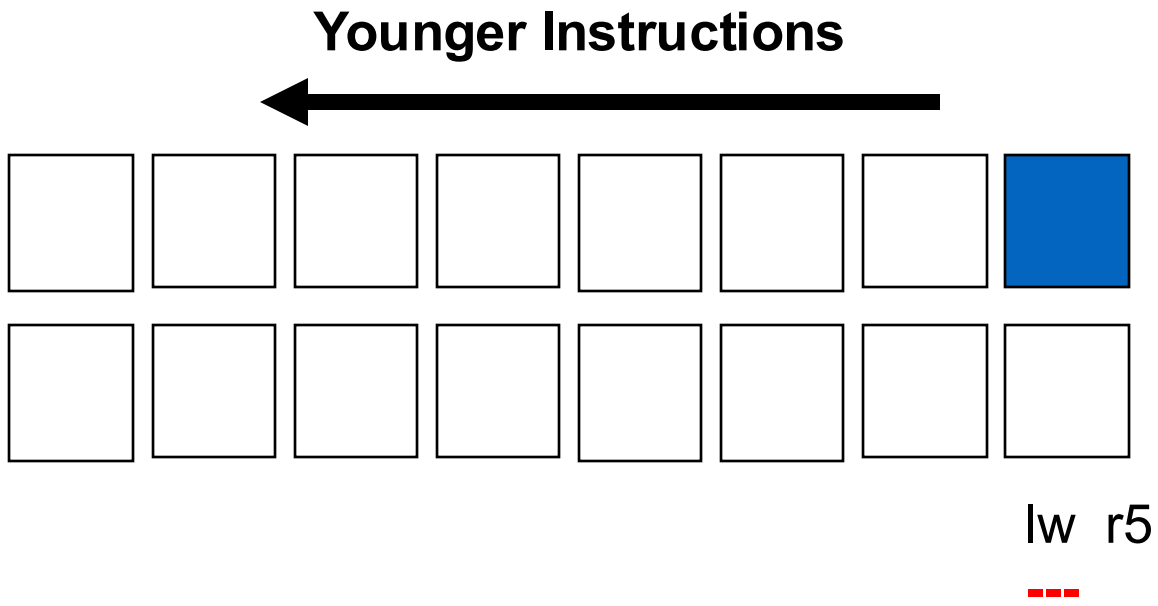
- Reason 2: Front-end hazards
  - Branch target address resolved after decode
    - Target addresses cached in Branch Target Buffer (BTB)
    - But BTB hit rate is not perfect (bubbles when BTB misses)
  - Branch direction (for conditionals) resolved in Execute
    - Branch directions are predicted
    - Latest predictors are quite advanced (perceptron, multi-length history, multi-level tables)
    - Branch prediction is not 100%
    - When mispredicted there are bubbles
  - Instruction cache misses freeze the pipeline
    - Next-line prefetchers capture only 50% of misses

Pipeline Bubbles Continued

- Reason 3: Data dependences
  - Data flows from one instruction to another
  - Dependent instructions can't execute in the same cycle
  - Data cache misses lengthen pipeline stall time
    - Misses can completely empty the pipeline
    - e.g. An LLC miss costs 70+ CPU cycles
  - Cache prefetching only useful for trivial patterns
    - Modern processors have strided prefetchers
    - Roughly 30% of accesses on average in integer code

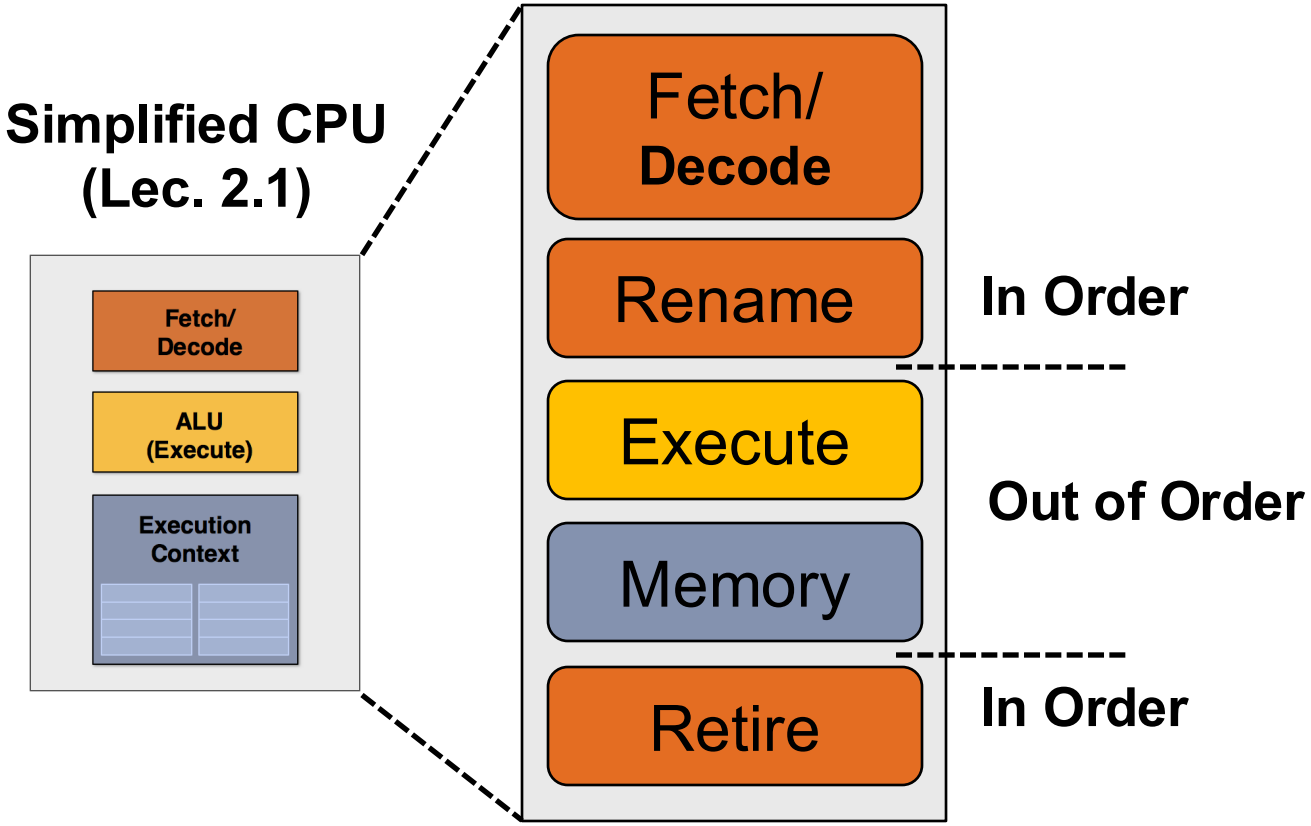
Pipeline Draining in In-order Core

- Load miss stalls all pipeline stages
  - Worst case for hardware usage



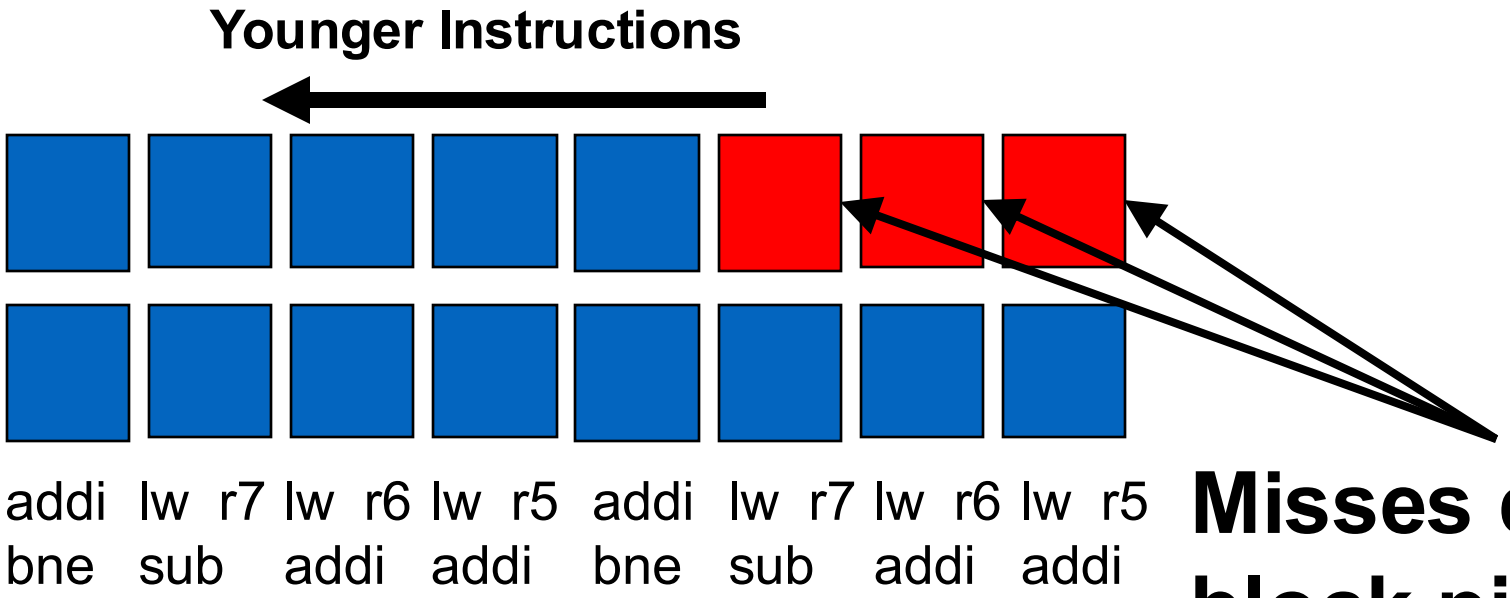
Reminder: OoO Execution (Lec. 2.1)

- Fetch instructions in-order, execute out of order, reconstruct order when retiring
  - Why? Expose parallelism for long-latency events



OoO Pipeline Handles Cache Misses

- Finds subsequent loads and executes them
  - Note: can't do **entire** loop trips
  - Because of `ld` → `add` → `st`

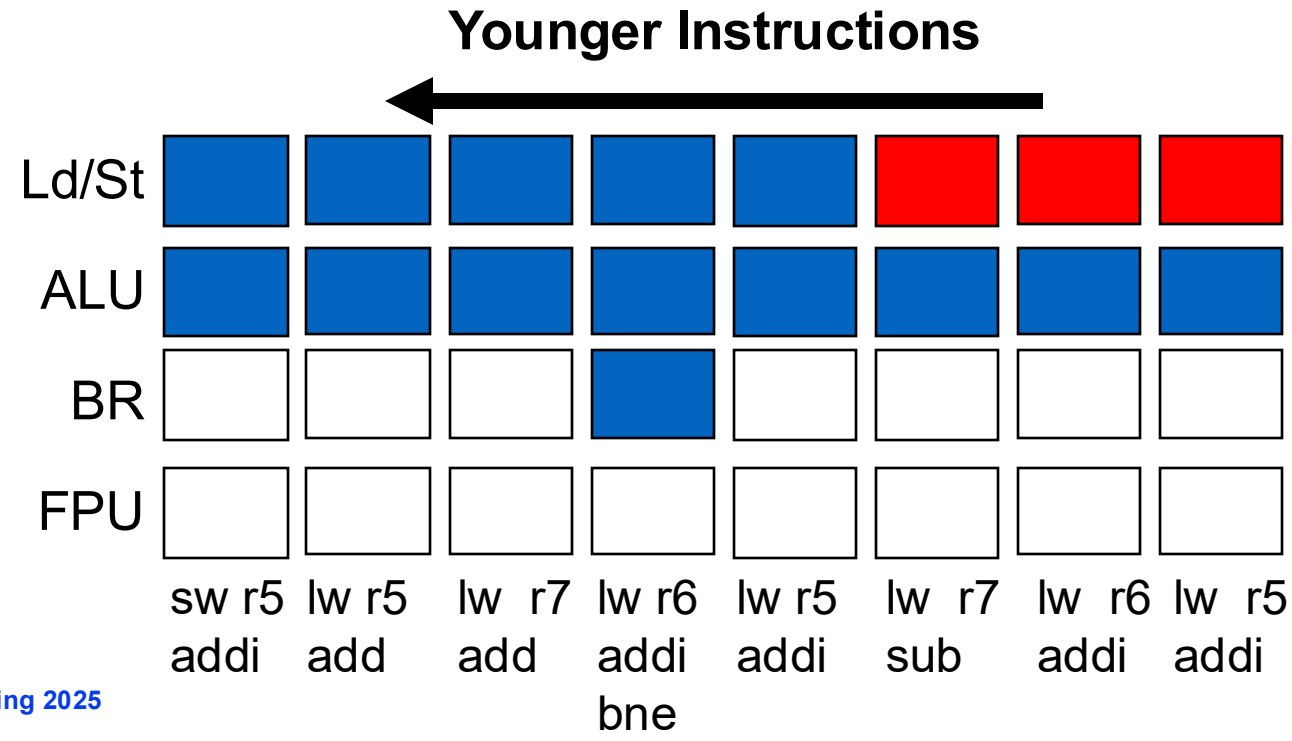



```
loop:
   lw   r5 , 0(r2)
   lw   r6 , 0(r3)
   lw   r7 , 0(r4)
   add  r5 , r5 , r6
   add  r5 , r5 , r7
   sw   r5 , 0(r2)
   addi r2 , r2 , 4
   addi r3 , r3 , 4
   sub  r9, r8, r2
   addi r4 , r4 , 4
   bne  r9, loop
```

Misses do not block pipeline!

OoO With Wider Pipelines

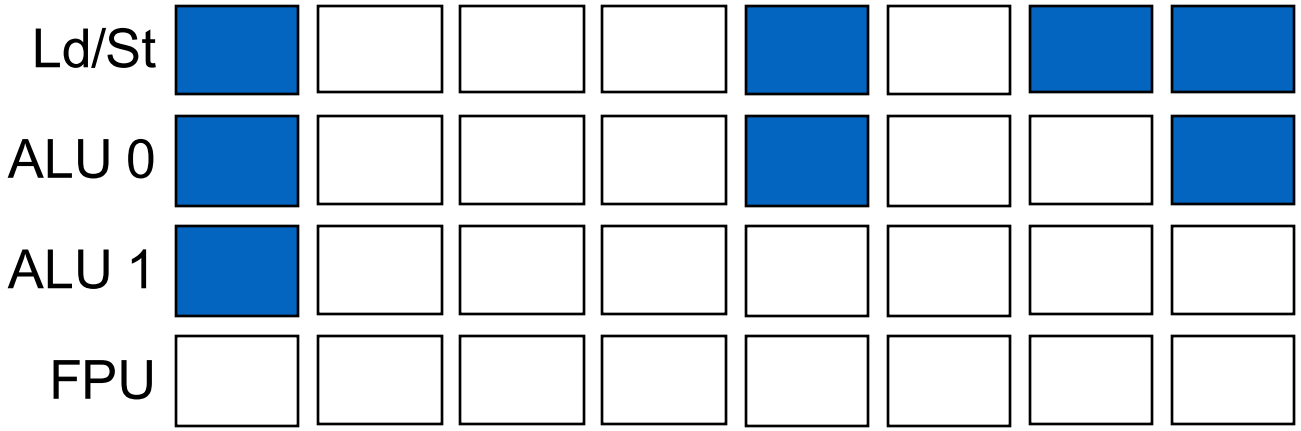
- Limited number of independent instructions
  - Making the pipeline 4-wide has limited benefit with this program




```
loop:
  lw   r5 , 0(r2)
  lw   r6 , 0(r3)
  lw   r7 , 0(r4)
  add  r5 , r5 , r6
  add  r5 , r5 , r7
  sw   r5 , 0(r2)
  addi r2 , r2 , 4
  addi r3 , r3 , 4
  sub  r9, r8, r2
  addi r4 , r4 , 4
  bne  r9, loop
```

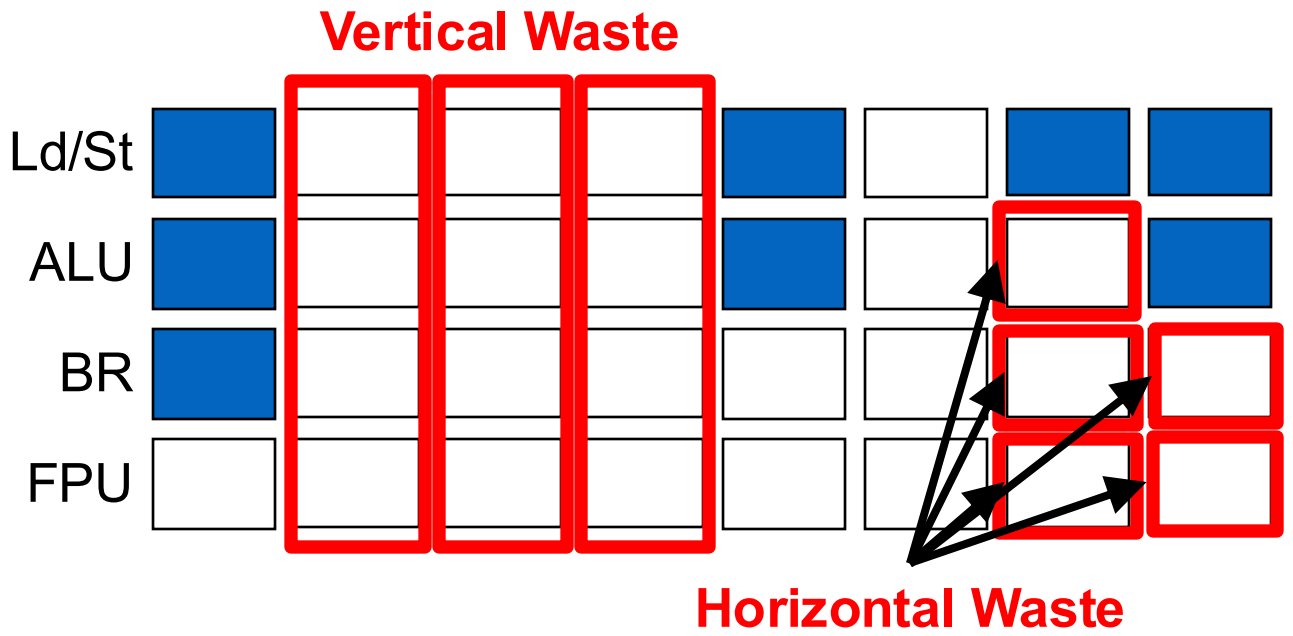
Realistic Pipeline Utilization

- In previous example, IPC ~ 2
  - Product of a very trivial program
  - OoO core easily traverses loops and issues loads
- Real programs are not so simple
  - Many cycles where nothing can be issued
  - More realistic IPC is between 0.5 and 1



Classifications of Hardware Waste

- Two categories: horizontal and vertical
- Vertical: whole cycle empty, nothing issued
  - Most common after long latency events
- Horizontal: unable to use full issue width
  - Software not exposing enough ILP for hardware



HW Multithreading

- Find ready instructions from many threads
  - Requires having thread context in **hardware**
  - Context entails: PC, SP, register file, interrupt descriptors
- Give the issue slot (each cycle) to another thread
  - Assuming multiple context this is easy
  - What type of waste does this target?
- Mix instructions from multiple threads per issue slot
  - Problems:
    - Scheduling policy, i.e., which thread to choose from?
    - Fairness?

Cost of Storing Multiple Contexts

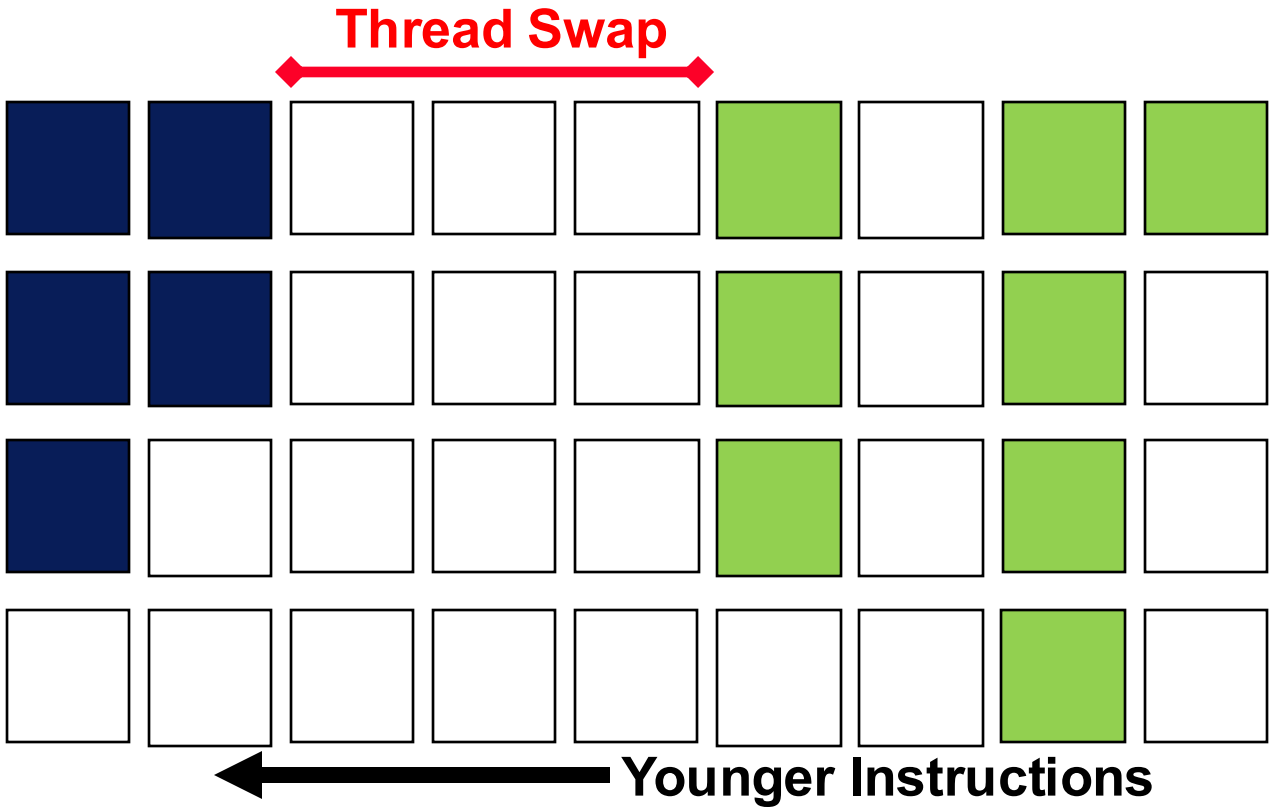
- Each thread needs **architectural** state
  - Registers, PC, stack, interrupt tables
  - Costs ~ KB of SRAM
  - Can impact area especially for small cores

- Threads share the memory hierarchy
  - Potential contention in data cache
  - Contention in instruction cache (problem in servers)
  - Will affect single thread performance (better throughput but worse latency)

Multithreading Trade-off

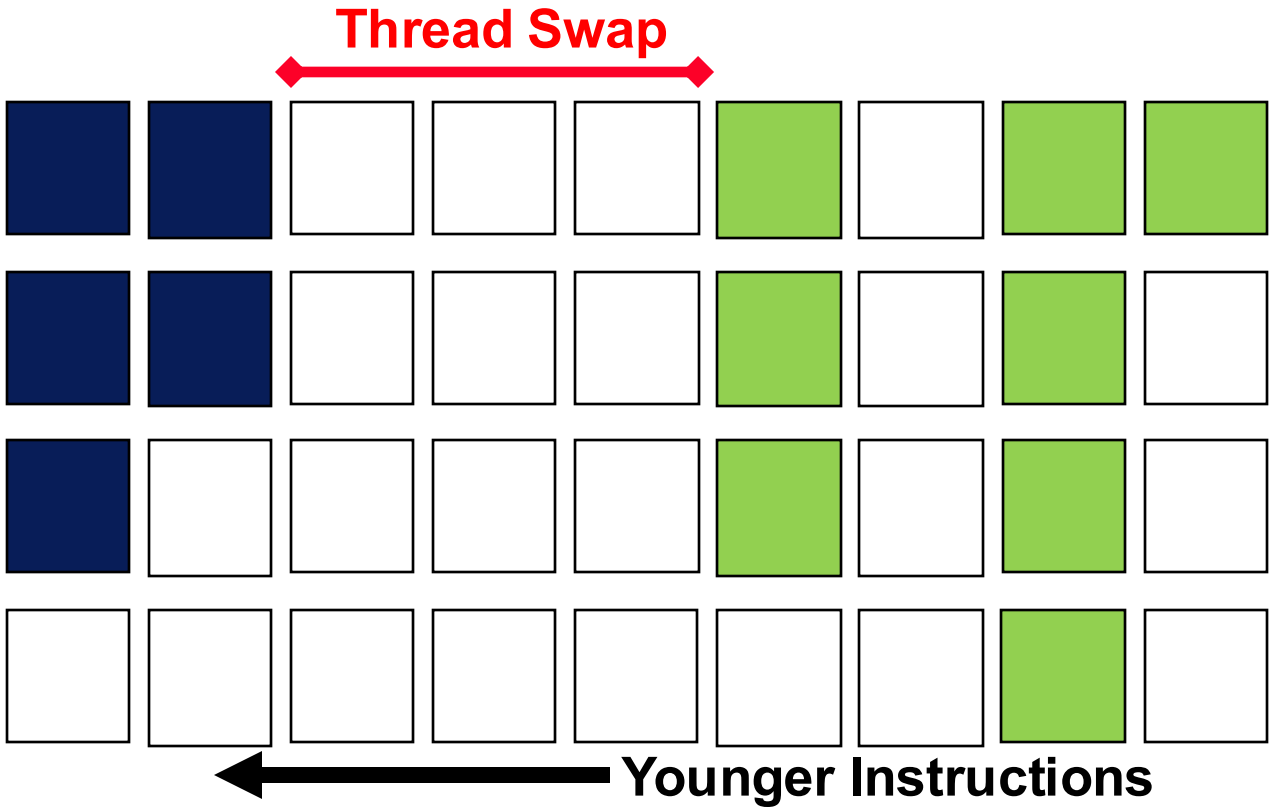
- Fundamental:
  - Multithreading **always** increases single-threaded latency. Why?
  - But increases pipeline utilization, higher throughput
- Likely an acceptable sacrifice
  - If the program can't use the hardware anyway…

Blocked (Coarse Grain) Multithreading



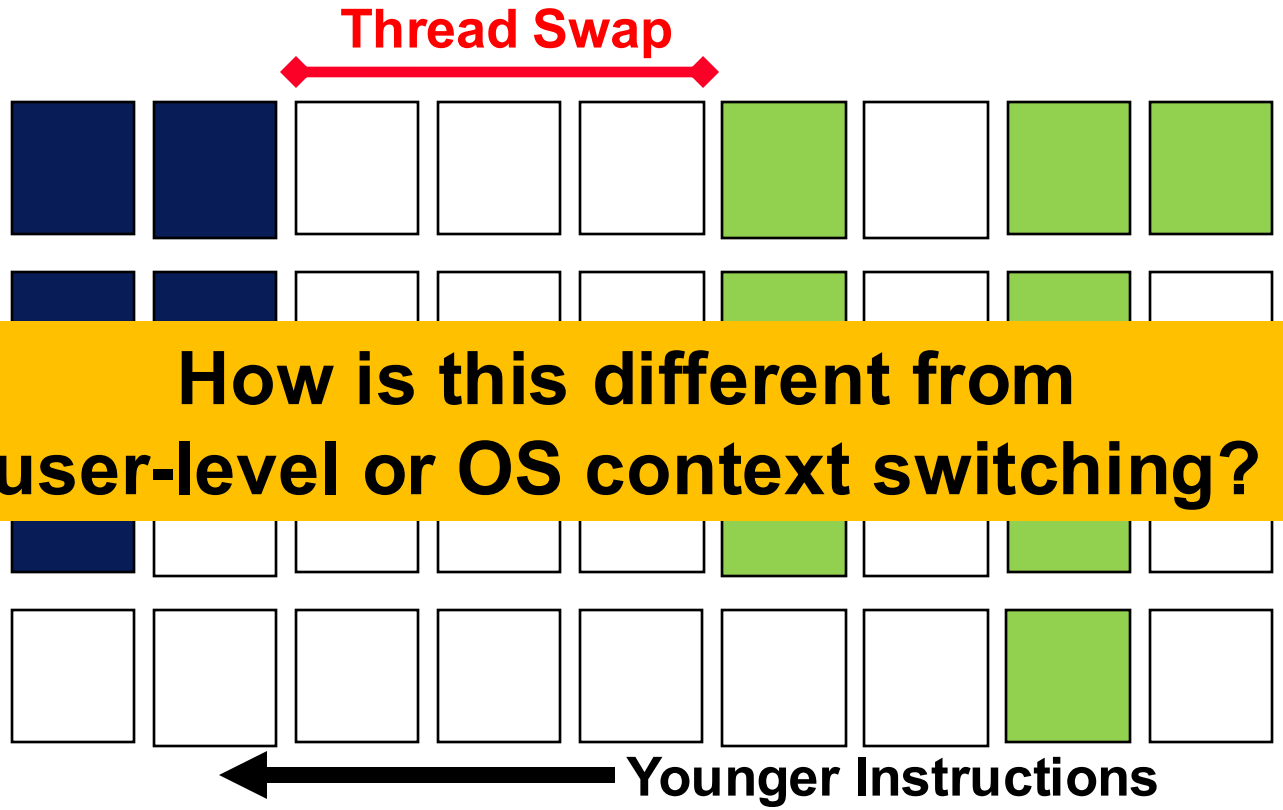
- Switch to a new thread on a long latency event
  - Pay a small cost to switch in a new context
  - Addresses purely vertical waste

Blocked (Coarse Grain) Multithreading



- May increase utilization
  - But, required to swap on long latency events

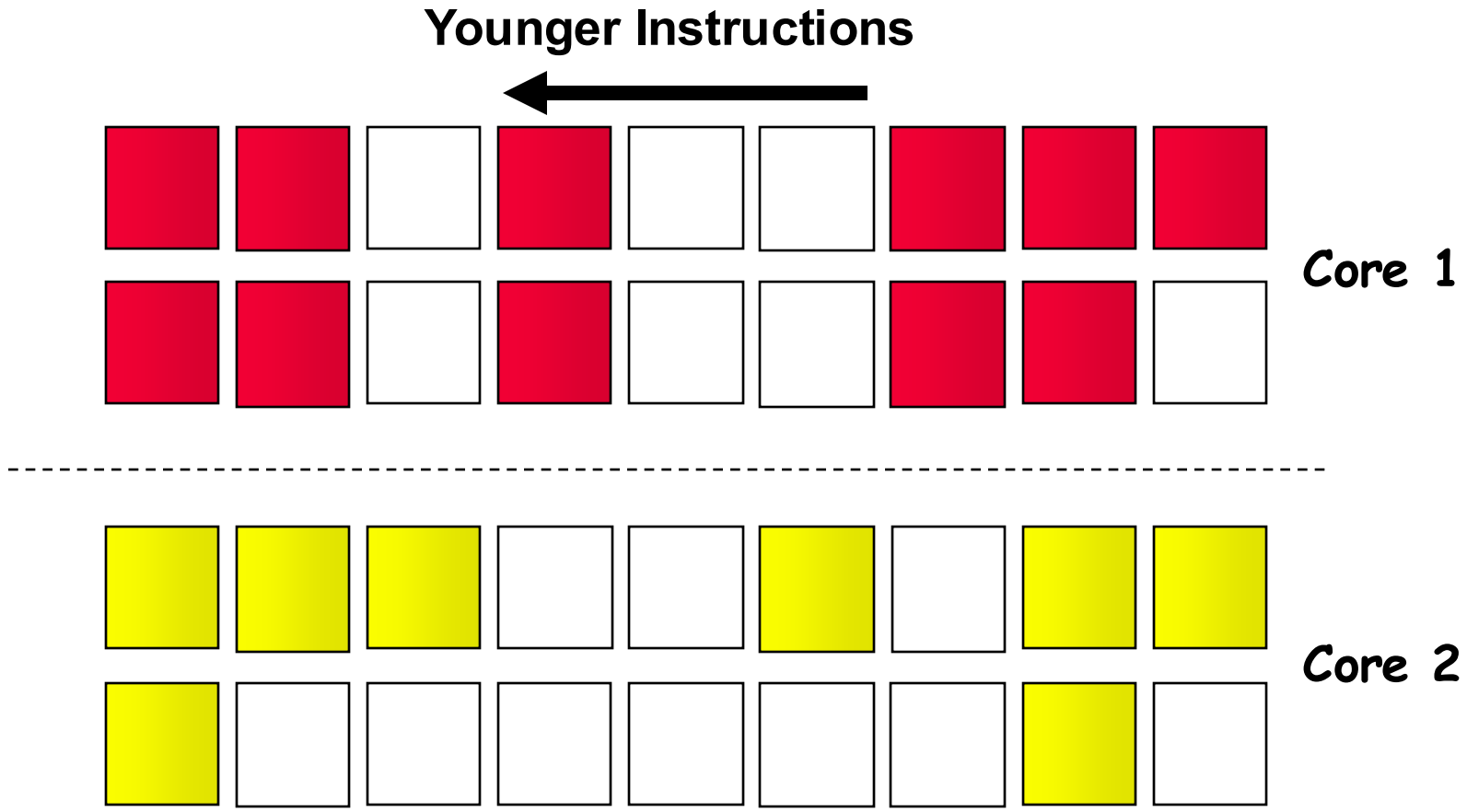
Blocked (Coarse Grain) Multithreading



How is this different from user-level or OS context switching?

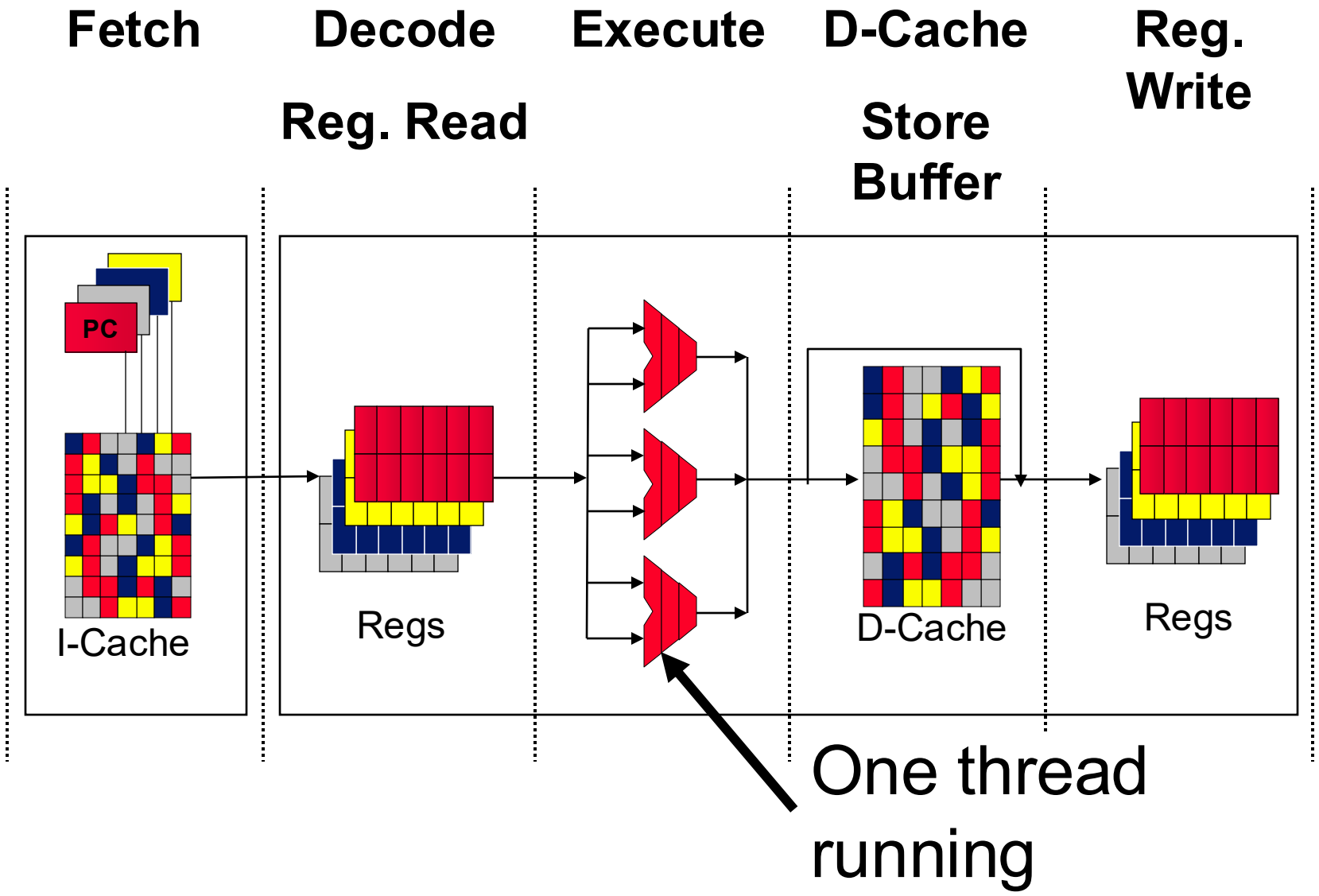
- May increase utilization
  - But, required to swap on long latency events

Comparison to Multicore (narrower pipelines)



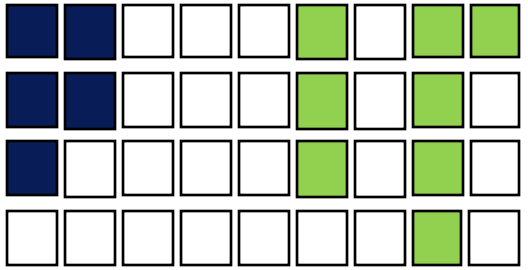
- Multiple narrow pipes have less horizontal waste
  - But would suffer from more vertical on long latency

CGMT Implementation



CGMT Performance

- Critical decision: when to switch threads
  - When current thread's utilization is about to drop (e.g. L2 cache miss)



- Requirements for improving throughput:
  - (Thread switch) + (pipe fill time) << blocking latency
    - Need useful work to be done before other thread comes back
  - Fast thread-switch: multiple register banks
  - Fast pipe fill: short pipe
- Advantage: small changes to existing hardware
- Drawback: single-thread performance suffers

Examples

- Macro-dataflow machine
- MIT Alewife/SPARCle
- IBM Northstar

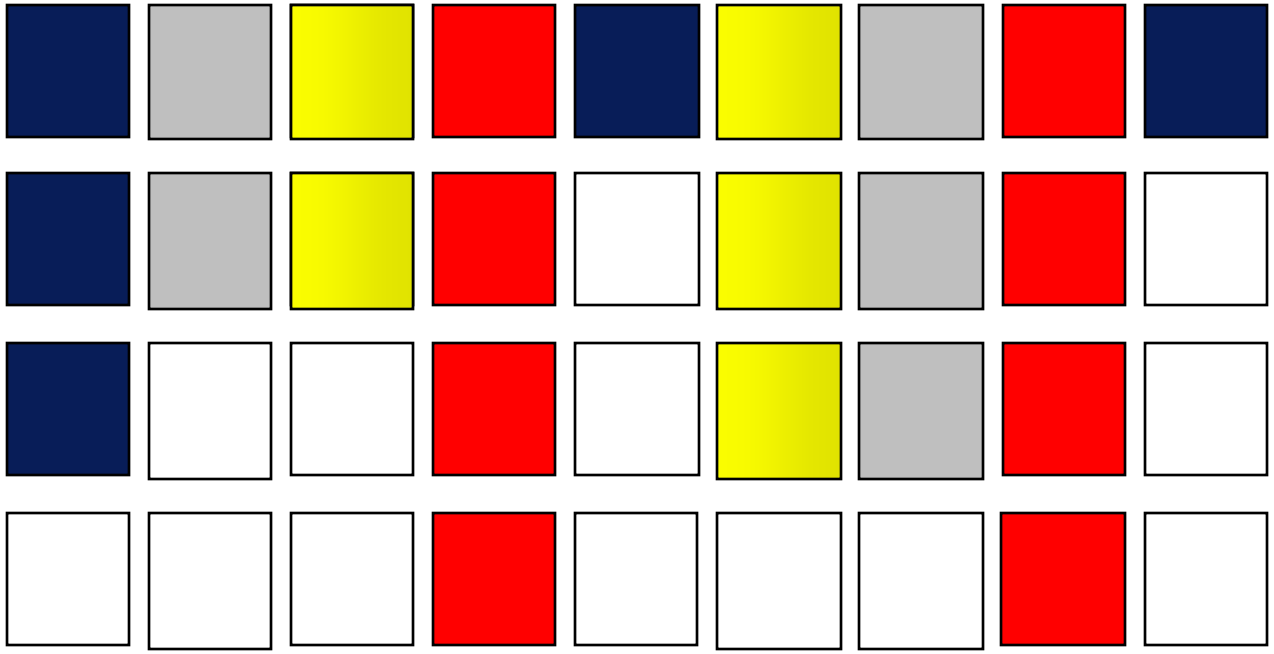
Exercise

- Assume IPC=1 per thread, switch time of zero
- Sharing results in single-thread performance dropping by 10%
- What is the average IPC with two threads?

Exercise

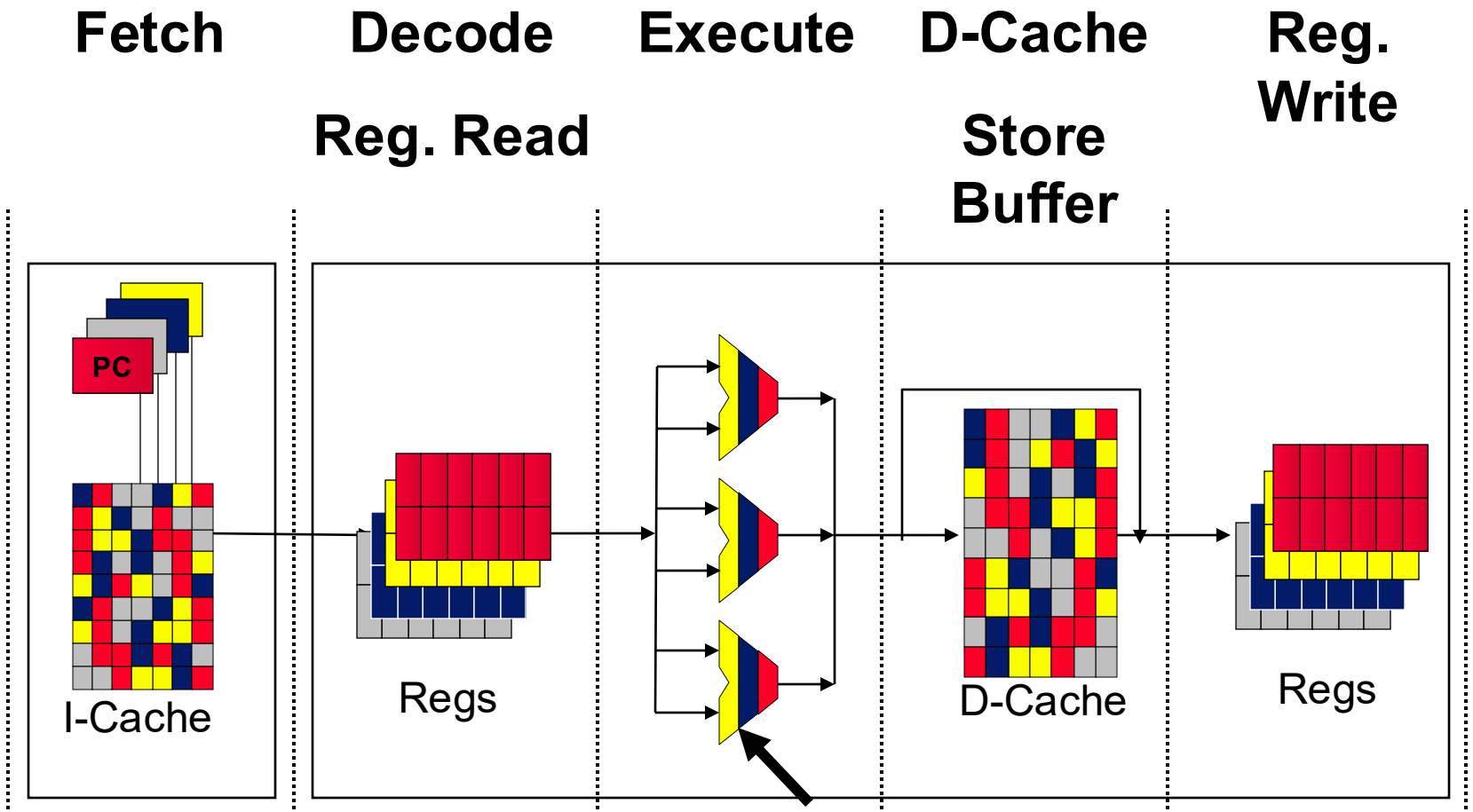
- Assume IPC=1 per thread, switch time of zero
- Sharing results in single-thread performance dropping by 10%
- What is the average IPC with two threads?
- **Answer: 0.9**
  - Each thread is 10% worse, and no overhead of switching between them
  - Note: do not multiply by two, since they are not running at the same time (they are multithreaded)

Fine Grained Multithreading (FGMT)



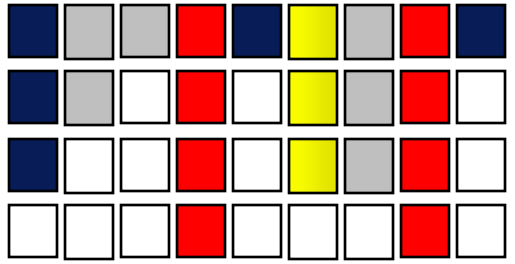
- Cycle between multiple threads periodically
  - Eliminate “switch time” by keeping all thread state hot

FGMT Implementation

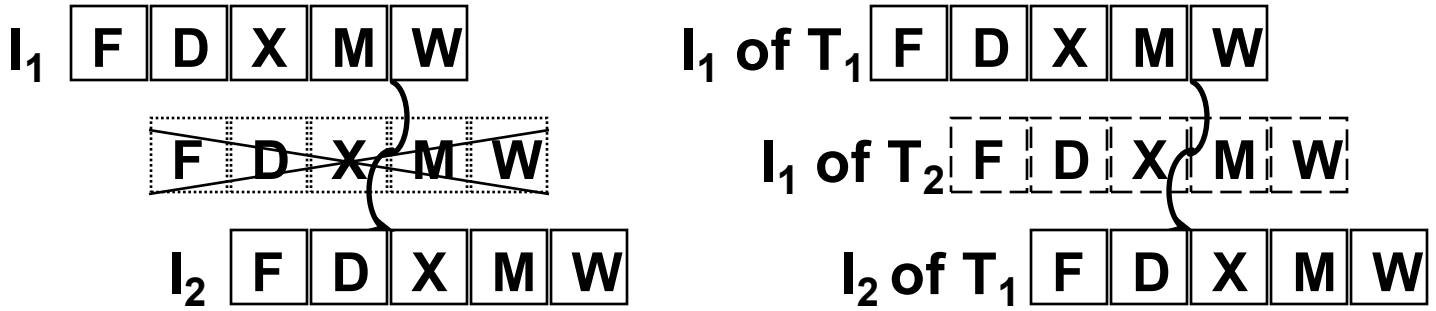


Many threads running

FGMT Design Points



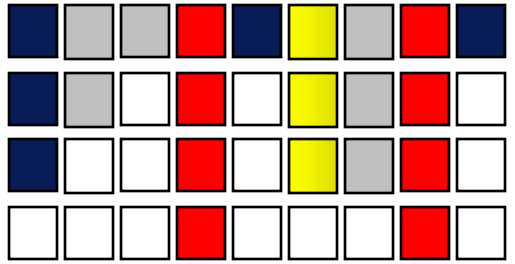
- Critical decision: none?
- Requirements for improving throughput:
  - Enough threads to eliminate vertical waste
  - e.g., Pipeline depth of 8, using 8 threads not enough
- Compensate for hardware cost in other areas
  - e.g., Bypass network may not be needed



Superscalar pipeline **Multithreaded pipeline**

FGMT vs. CGMT

- Thread switch policy:
  - Most historical designs of FGMT are round robin
  - CGMT swaps on long latency events



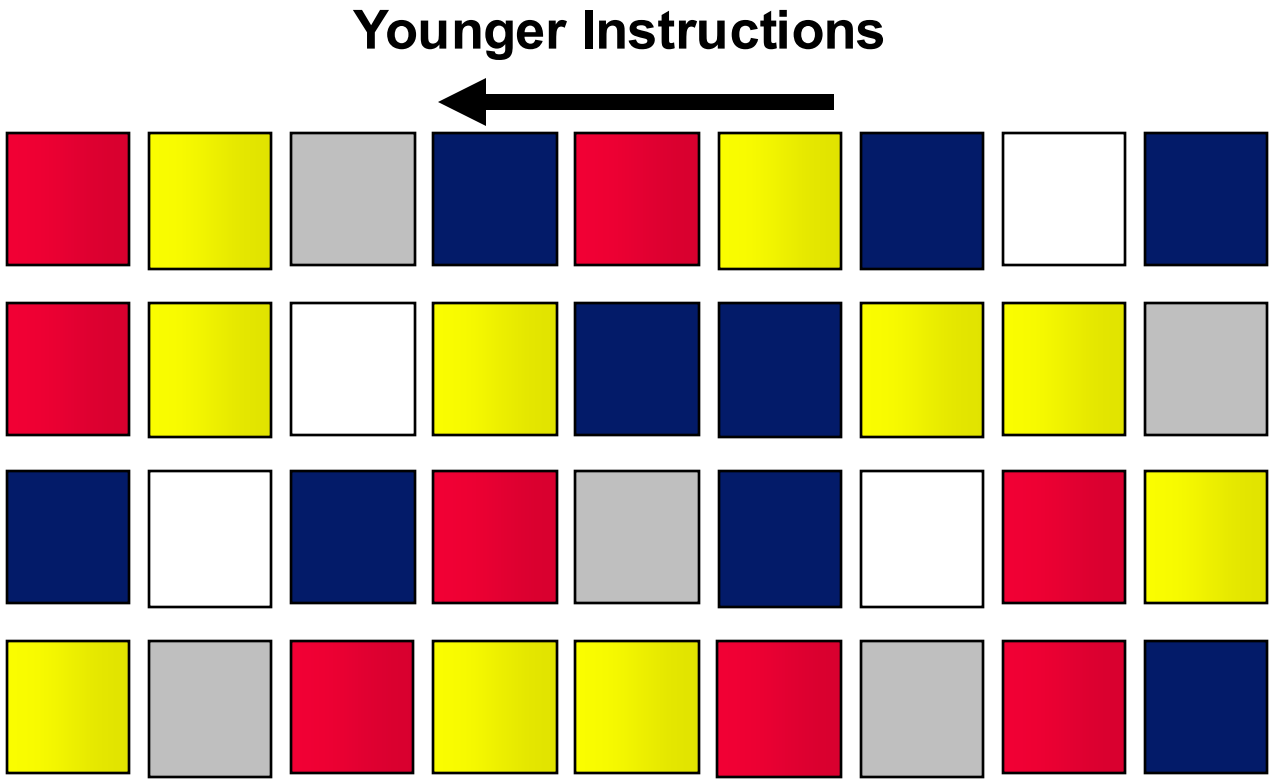
- FGMT addresses much shorter latencies
  - Hardware costs to keep all contexts immediately ready

- For threads with abundant parallelism, FGMT could suffer
  - Can introduce “flexible interleaving” to solve
  - One thread could remain scheduled for many cycles

How well does FGMT do?

- Advantages:
  - With flexible interleave:
    - Reasonable single thread performance
    - High processor utilization (esp. in case of many thread)
  - Without:
    - Suits “regular” applications with repetitive latencies
- Drawbacks:
  - More complicated hardware to track dependences
  - Many more contexts means we need many registers
  - Without flex. interleaving, limited single thread perf.
- **Still cannot address horizontal waste!**

Simultaneous Multithreading (SMT)



- Instructions from multiple threads in **same cycle**
  - First design that can reduce horizontal waste
  - Foundational papers: (fun reads if you are enjoying this topic!) SMT [ISCA'95], "Exploiting Choice" [ISCA'96]

SMT Shares Pipeline Structures

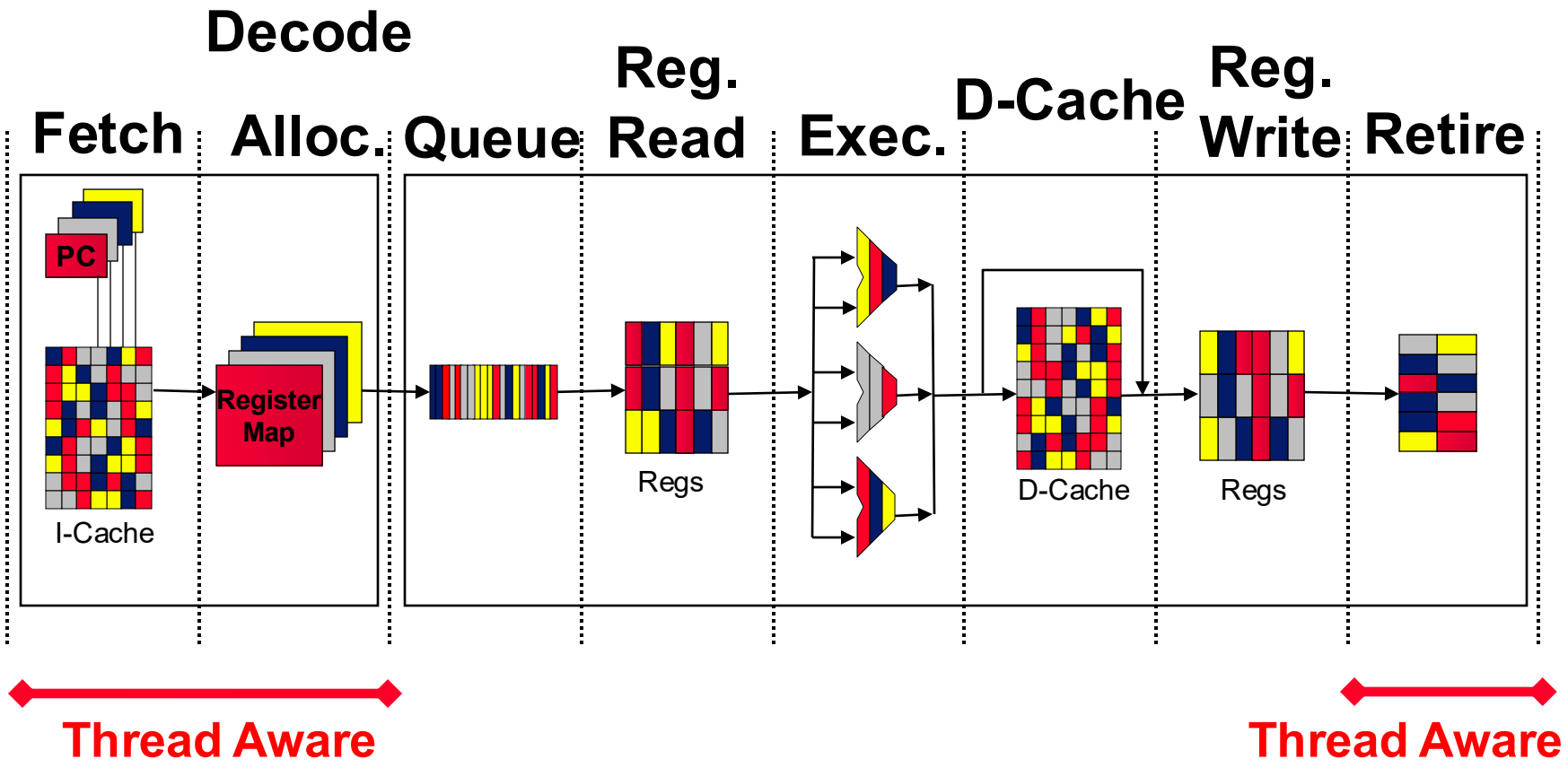
- Previously, only caches and predictors shared
  - With SMT, instructions from diff. threads are interleaved cycle by cycle
  - Do we need to resolve all potential dependences?
    - e.g. Both T_1 and T_2 execute `add r1, r2, r3`
  - No! Register allocation takes care of that

- Recall: once registers become physically mapped, the instructions can be issued freely
  - T_1: `add r1, r2, r3` becomes: `add p4, p2, p3`
    T_2: `add r1, r2, r3` `add p9, p8, p5`

SMT Implementation

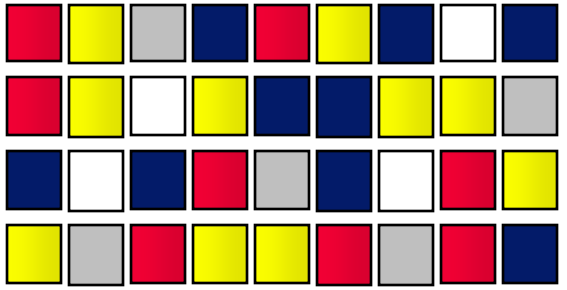
- After allocation, scheduler is thread agnostic
- Stages that duplicate their state:
  - Fetch, Decode, Allocate (one map table per thread)
  - Retire stage needs to know which registers to free
- Policies to share the pipeline structures:
  - Static partitioning i.e., each thread gets ½ the structures
    - First generation Intel Pentium 4
  - Dynamic sharing i.e., every entry is tagged
    - Current Intel and AMD processors (Kaby Lake, Ryzen)

SMT Implementation



SMT Design Points

- Critical decision: fetch-interleaving policy
- Requirements for throughput:
  - Enough threads to utilize resources
    - Fewer than needed to stretch dependences
- Examples:
  - Compaq Alpha EV8 (cancelled)
  - Intel Hyper-Threading/AMD SMT Technology (x86)
  - Marvell/Cavium Thunder X2, X3 (ARM)
  - IBM POWER9 (POWER ISA)



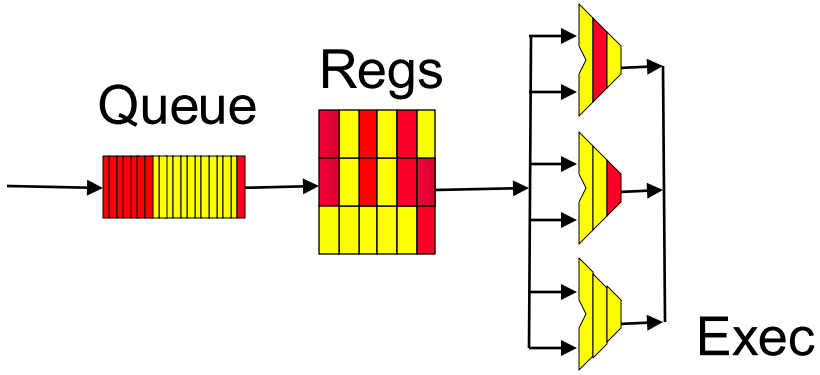
Fetch Interleaving Policies

- How to decide which threads fetch instructions?
  - Some may bring in useful instructions
  - Others may fetch instructions that are stalled
- Fundamental goals:
  - Maximize active use of issue width
  - Guarantee forward progress
    - Can’t stop a blocked thread forever, e.g., on synchronization

Fetch Interleaving Policies

- Common Policy: ICOUNT
  - Thread with fewest instructions in pipe has priority
  - Adapts to all sources of stalls (cache, FP units, etc…)

- Increased mem. latency, ICOUNT does worse
  - Thread's rare fetch opportunities can still fill pipeline
  - Holding resources adversely affects other thread(s)
    e.g., if red thread is blocked, but it holds ½ resources



Conditions for Good SMT Performance

- When threads do not thrash each other's state
  - Important structures: branch predictor, ROB, caches
- SMT performs better with ample HW resources
  - Without horizontal slots to fill, extra context is wasted
  - Natural fit for wide-issue OoO cores

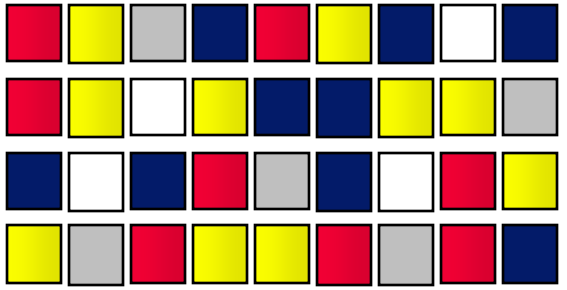
Example SPARC T5

- 8 threads per core, 16 cores per processor
  - Out of order, superscalar pipeline
  - Private L1 and L2
  - 3.6 GHz Frequency
- 2 of out 8 can execute simultaneously
- “Modified least recently used” thread selection algorithm
  - Also known as “round robin”

Example x86, ARM and IBM CPUs

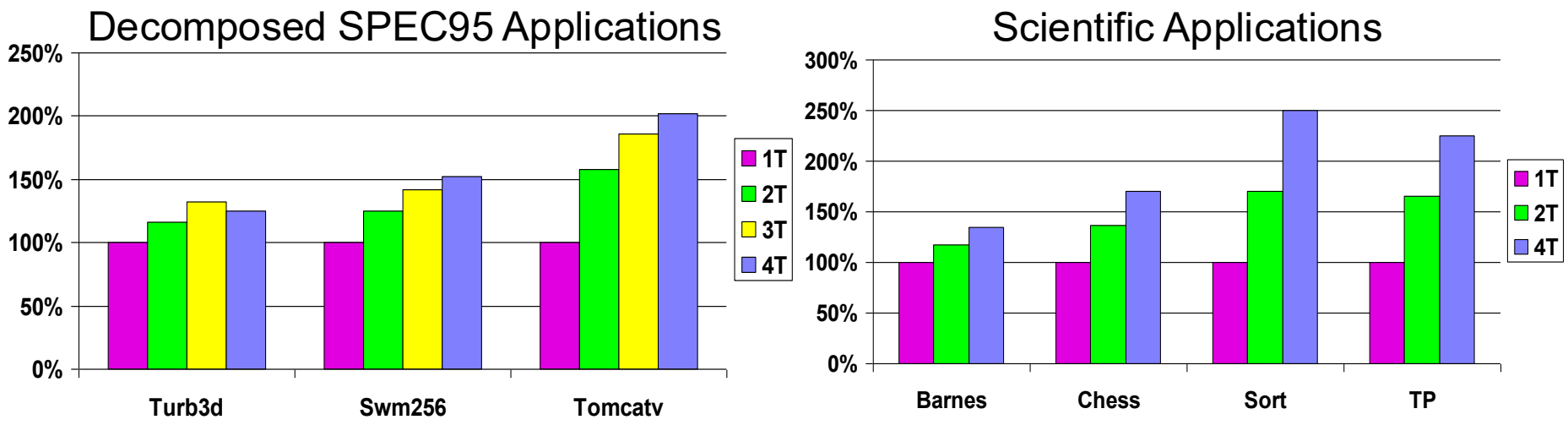
- x86 cores have 2 threads/core
  - Intel calls it “Hyperthreading”
  - AMD calls it Simultaneous Multithreading
- There are also ARM cores with SMT
  - Cortex A65-E, Neoverse E1, Cavium Thunder X2, 2 threads/core
  - Cavium Thunder X3, 4 threads/core
- IBM’s POWER CPUs are highly threaded
  - POWER 7 had 4 threads/core
  - POWER 8 and POWER 10 have 8 threads/core

SMT Case Study: Alpha EV8



- 8-issue OOO processor (wide machine)
- SMT Support
  - Multiple sequencers (PC): 4
  - Alternate fetch from multiple threads
    - Separate register renaming for each thread
  - More (4x) physical registers
  - Thread tags on all shared resources
    - ROB, LSQ, BTB, etc.
    - Allow per thread flush/trap/retirement
  - Process tags on address resources: caches, TLB's, …
  - Notice: none of these between allocate and retire

Scalability of SMT

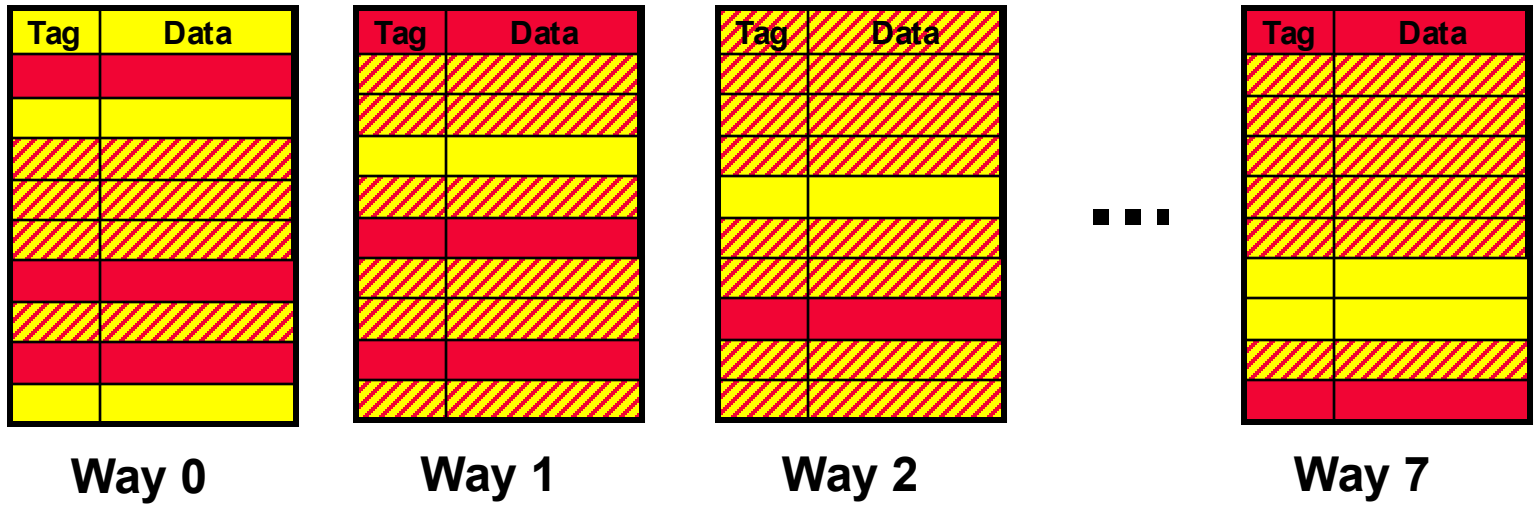


- SPEC benefits less from MT than scientific apps
  - 4T gives 200% boost in sort/TP, only 125% in turb3d

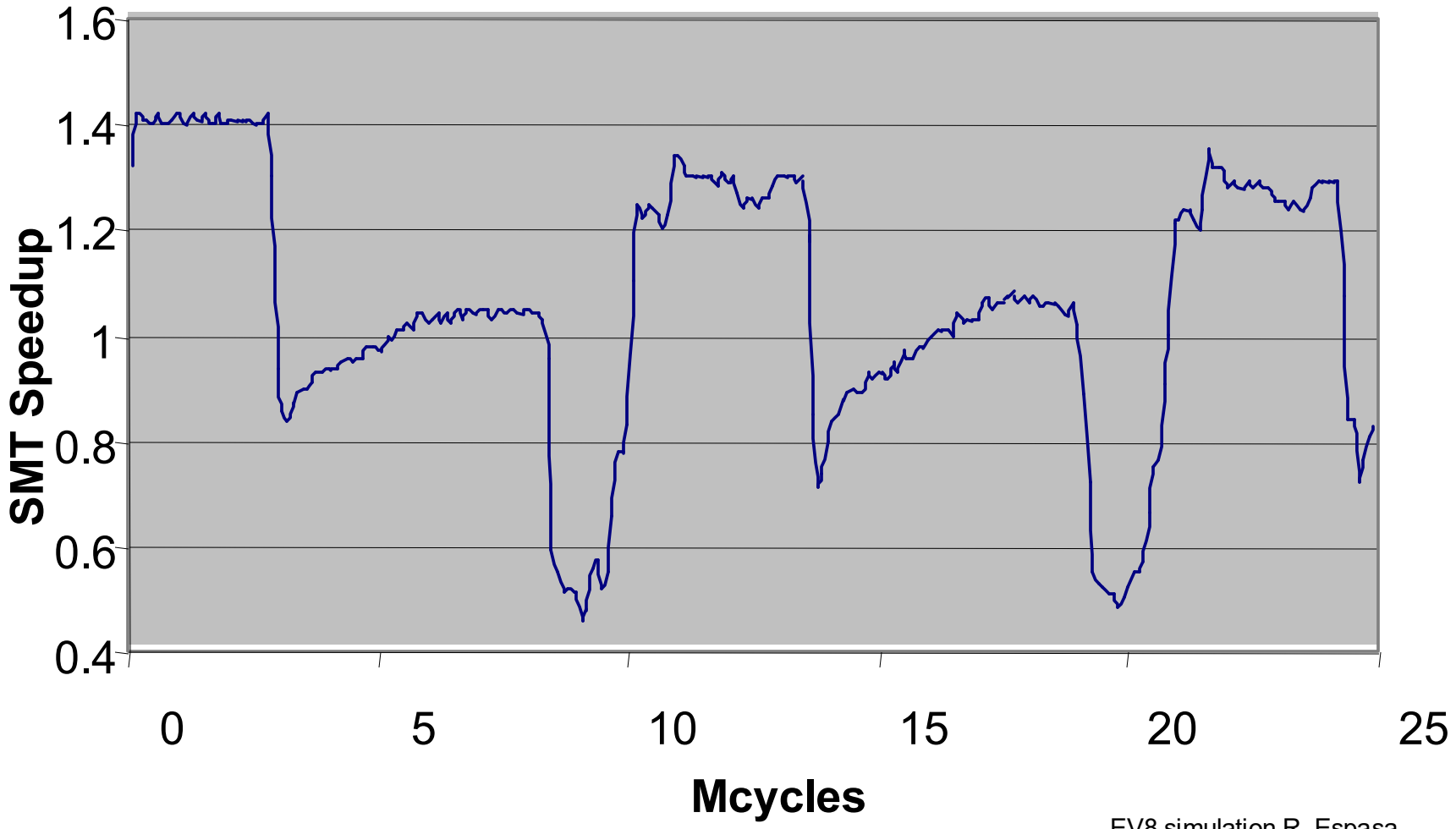
- Although most of the pipeline is SMT-agnostic, cycle time limits scalability for map table, registers
  - Beyond 2-4 threads, program dependences limit benefit

Cache Sharing Amongst Threads

- Private caches
  - Easier to implement – use existing MP like protocol
- Shared Caches
  - Faster to communicate among threads
  - No coherence overhead
  - Flexibility in allocating resources



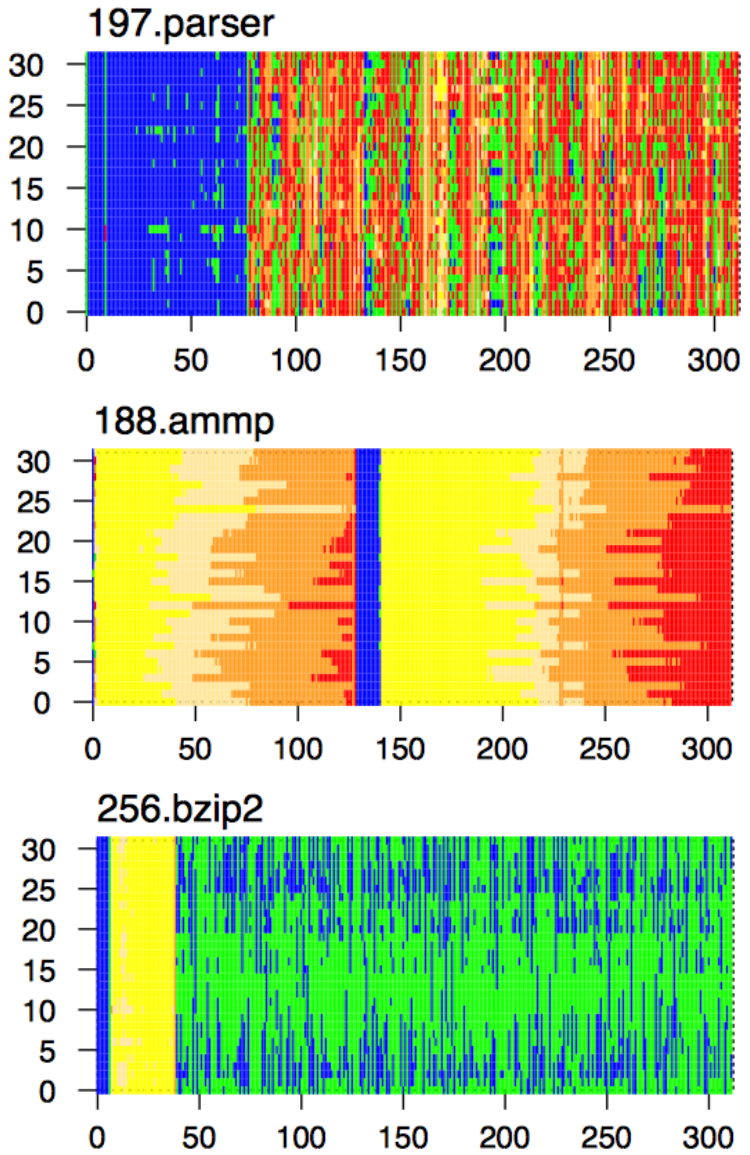
Performance Variability in EV8 SMT



EV8 simulation R. Espasa

Cache and Memory Contention

- Apps. may use different sets in a shared cache
  - e.g., 3 from SPEC2k6
- Blue = few references
  red/yellow = many refs.
- Running *parser* and *bzip2* on co-located SMT threads will have much less contention than *parser* and *ammp*



Source: "Contention on L2 Caches May Limit the Effectiveness of SMT", Hily & Seznec, JILP'05

Adjusting SMT Scheduling

- Add a cache-usage counter (Hily & Seznec, 2005), to each L2 cache set
  - Ticks on requests to same set from each SMT thread
  - Serves as a proxy for conflicts
- OS can read this counter and adjust scheduling
- Reduces overall cache miss rate by ~10%

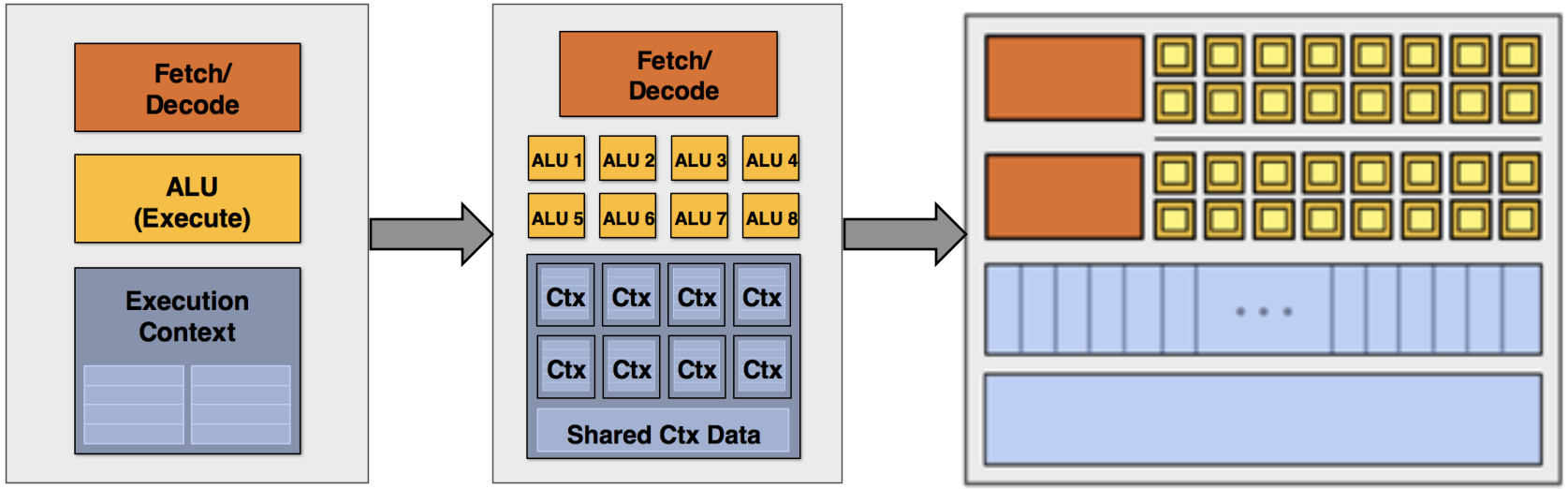
Constructive Sharing?

- Instruction cache behavior can be the opposite
  - If SMT threads are executing similar code
  - Very common in database applications or servers
  - Database workloads are intrinsically multithreaded
    - Natural fit for using FGMT or SMT

- Data from Oracle database apps. shows…
  - Apx. 35% less instruction cache misses using SMT (Lo, ISCA'98)
  - Using smart OS policy, can achieve same miss rates in **data cache** as a single thread, with 8-way SMT

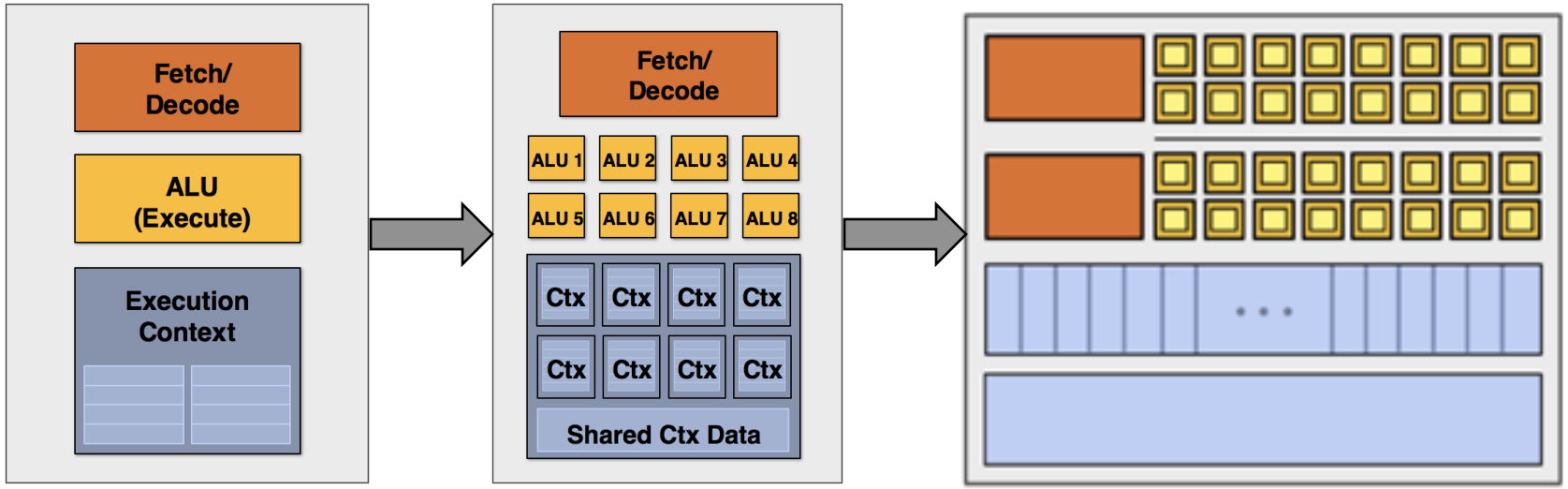
Reminder: Taking Parallelism Further (Lec. 2)

- Start with a single instruction CPU…
- Add ALUs to make it SIMD, then go to extreme
  - Making one instruction control hundreds of ALUs



Reminder: Taking Parallelism Further (Lec. 2.1)

- Works great for embarrassingly parallel programs
- The problem? Memory and long latency ops.
  - Solution: Use huge degree of multithreading
    - Every clock cycle should have a thread ready to execute



GPUs

- Make cores simple
  - In order pipelines
  - No branch prediction
- Put many cores in the die
  - Simple cores are smaller
- Take throughput-latency trade-off to extreme
  - Trillions of integer operations per second
  - … but, huge single-thread latency

GPUs: Two Levels of Multithreading

- Vector lane threading
  - Assign threads to a vector lane (as in SIMD/Vector)
  - Each lane has one PC
  - Group threads called “Warp” to run together
  - Multithreading across pipeline width
- Multiple warps can share the pipeline
  - Multithreading across pipeline depth

Summary

- Need to keep pipeline utilization high
- Solution: Multithreading
  - Blocked multithreading
  - Fine-grained multithreading
  - Simultaneous multithreading
- Taking multithreading to the limit: GPUs
  - Next week, will discuss in detail!