Computer Architecture: Multithreading (II)

Prof. Onur Mutlu
Carnegie Mellon University

A Note on This Lecture

- These slides are partly from 18-742 Fall 2012, Parallel Computer Architecture, Lecture 10: Multithreading II

- Video of that lecture:
  - http://www.youtube.com/watch?v=e8lfl6MbILg&list=PL5PHm2jkkXmh4cDkC3s1VBB7-njlgiG5d&index=10

More Multithreading

Readings: Multithreading

- Required
  - Spracklen and Abraham, "Chip Multithreading: Opportunities and Challenges," HPCA Industrial Session, 2005.
  - Kalla et al., "IBM Power5 Chip: A Dual-Core Multithreaded Processor," IEEE Micro 2004.
  - Tullsen et al., "Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor," ISCA 1996.
  - Eyerman and Eeckhout, "A Memory-Level Parallelism Aware Fetch Policy for SMT Processors," HPCA 2007.

- Recommended
  - Hirata et al., "An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads," ISCA 1992
  - Smith, "A pipelined, shared resource MIMD computer," ICPP 1978.
  - Gabor et al., "Fairness and Throughput in Switch on Event Multithreading," MICRO 2006.
  - Agarwal et al., "APRIL: A Processor Architecture for Multiprocessing," ISCA 1990.

Review: Fine-grained vs. Coarse-grained MT

- Fine-grained advantages
  + Simpler to implement, can eliminate dependency checking, branch prediction logic completely
  + Switching need not have any performance overhead (i.e. dead cycles)
    + Coarse-grained requires a pipeline flush or a lot of hardware to save pipeline state
      → Higher performance overhead with deep pipelines and large windows

- Disadvantages
  - Low single thread performance: each thread gets 1/Nth of the bandwidth of the pipeline

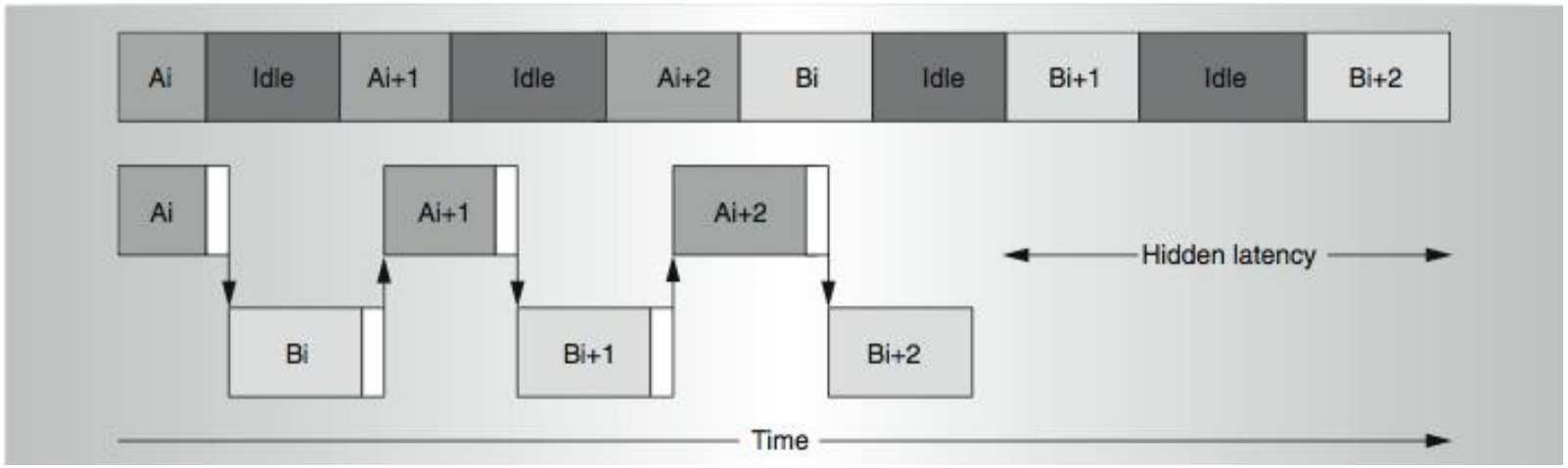
IBM RS64-IV

- 4-way superscalar, in-order, 5-stage pipeline
- Two hardware contexts
- On an L2 cache miss
  - Flush pipeline
  - Switch to the other thread

- Considerations
  - Memory latency vs. thread switch overhead
  - Short pipeline, in-order execution (small instruction window) reduces the overhead of switching

Intel Montecito

- McNairy and Bhatia, "Montecito: A Dual-Core, Dual-Thread Itanium Processor," IEEE Micro 2005.



- Thread switch on
  - L3 cache miss/data return
  - Timeout – for fairness
  - Switch hint instruction
  - ALAT invalidation – synchronization fault
  - Transition to low power mode
- <2% area overhead due to CGMT

Fairness in Coarse-grained Multithreading

- Resource sharing in space and time always causes fairness considerations
  - Fairness: how much progress each thread makes

- In CGMT, the time allocated to each thread affects both fairness and system throughput
  - When do we switch?
  - For how long do we switch?
    - When do we switch back?
  - How does the hardware scheduler interact with the software scheduler for fairness?
  - What is the switching overhead vs. benefit?
    - Where do we store the contexts?

Fairness in Coarse-grained Multithreading

- Gabor et al., "Fairness and Throughput in Switch on Event Multithreading," MICRO 2006.
- How can you solve the below problem?

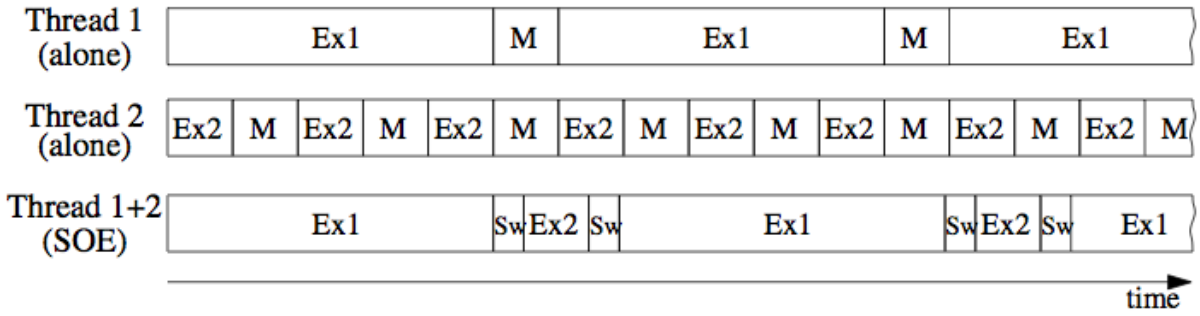


Figure 1. Intuitive example of unfair execution in SOE. *Ex1* marks execution of instructions from thread 1, *Ex2* from thread 2, *M* marks last level cache misses and *Sw* denotes thread switch overheads. When both threads run together using SOE (bottom), the 2nd thread runs extremely slowly while the 1st thread's performance is hardly affected by the multithreading.

Fairness vs. Throughput

- Switch not only on miss, but also on data return

- Problem: Switching has performance overhead
  - Pipeline and window flush
  - Reduced locality and increased resource contention (frequent switches increase resource contention and reduce locality)

- One possible solution
  - Estimate the slowdown of each thread compared to when run alone
  - Enforce switching when slowdowns become significantly unbalanced
  - Gabor et al., "Fairness and Throughput in Switch on Event Multithreading," MICRO 2006.

Thread Switching Urgency in Montecito

- Thread urgency levels
  - 0-7
- Nominal level 5: active progress
- After timeout: set to 7
- After ext. interrupt: set to 6
- Reduce urgency level for each blocking operation
  - L3 miss
- Switch if urgency of foreground lower than that of background

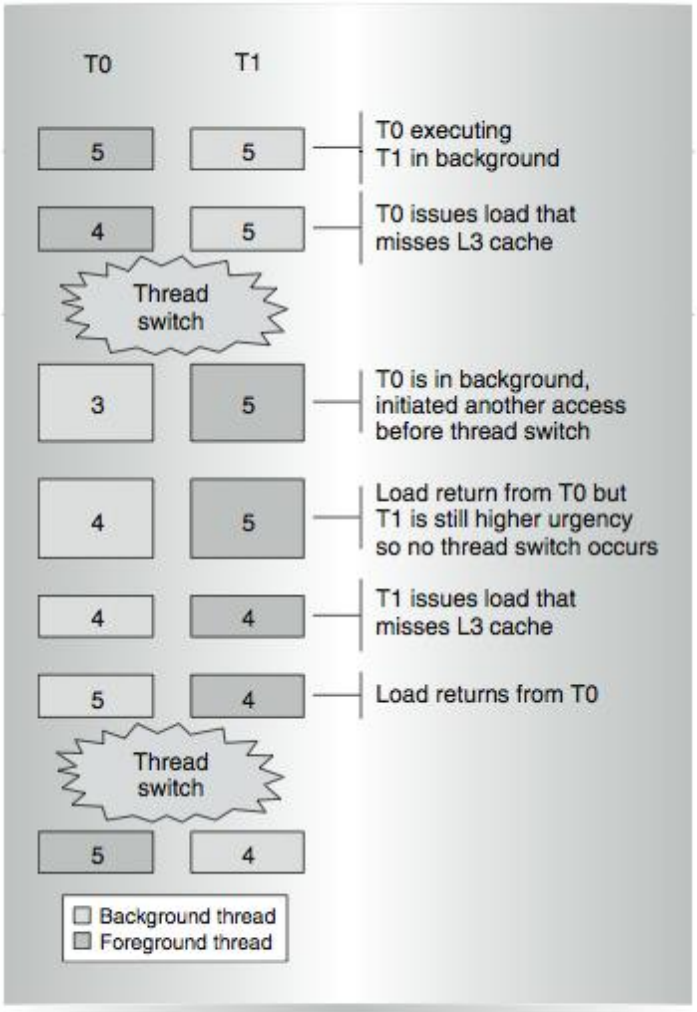
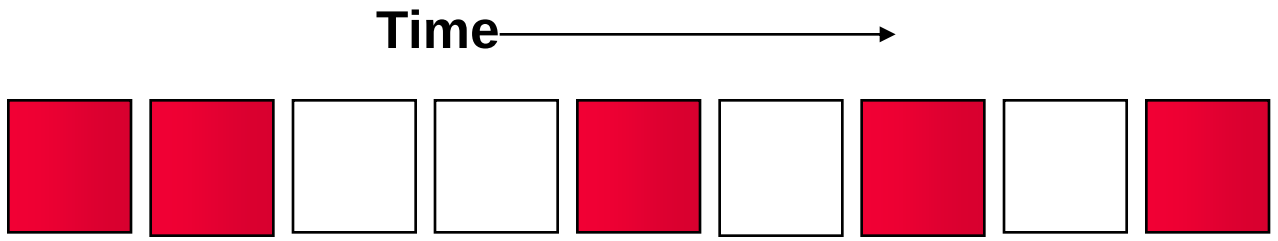


Figure 4. Urgency and thread switches on the Montecito processor.

Simultaneous Multithreading

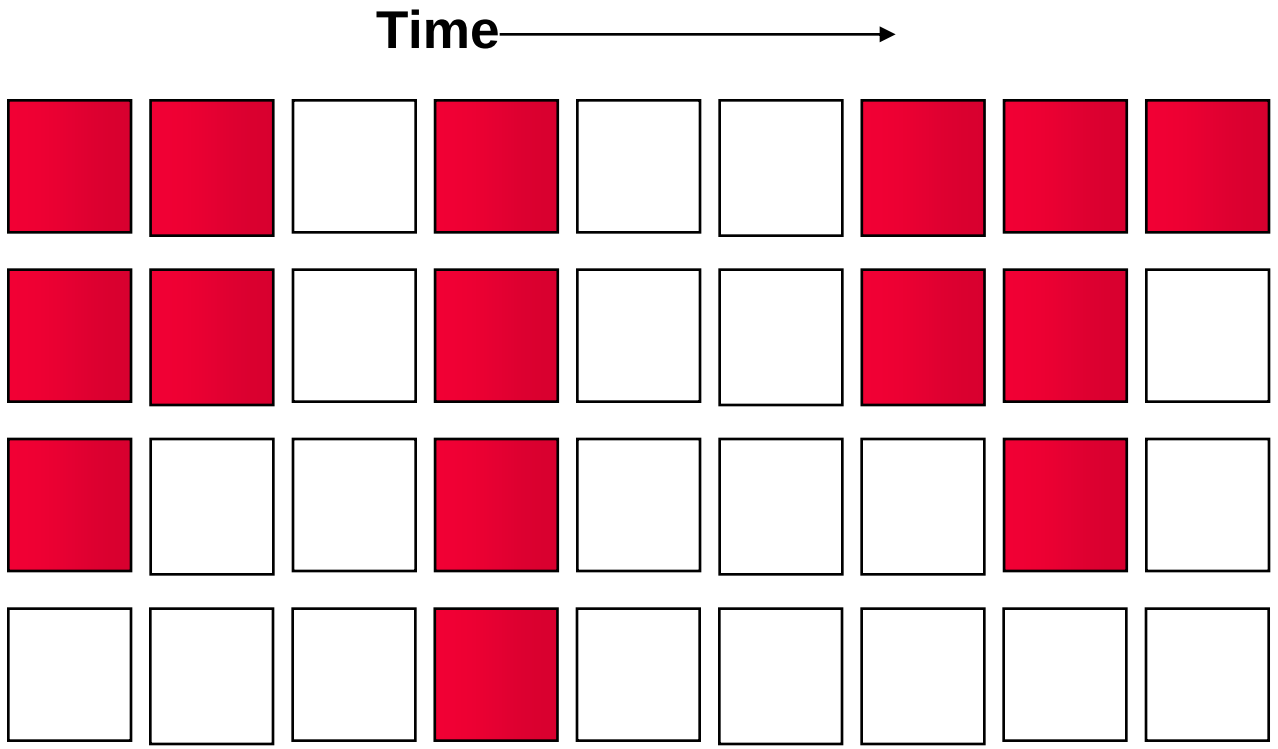
- Fine-grained and coarse-grained multithreading can start execution of instructions from *only* a single thread at a given cycle
- Execution unit (or pipeline stage) utilization can be low if there are not enough instructions from a thread to "dispatch" in one cycle
  - In a machine with multiple execution units (i.e., superscalar)
- Idea: Dispatch instructions from multiple threads in the same cycle (to keep multiple execution units utilized)
  - Hirata et al., "An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads," ISCA 1992.
  - Yamamoto et al., "Performance Estimation of Multistreamed, Superscalar Processors," HICSS 1994.
  - Tullsen et al., "Simultaneous Multithreading: Maximizing On-Chip Parallelism," ISCA 1995.

Functional Unit Utilization



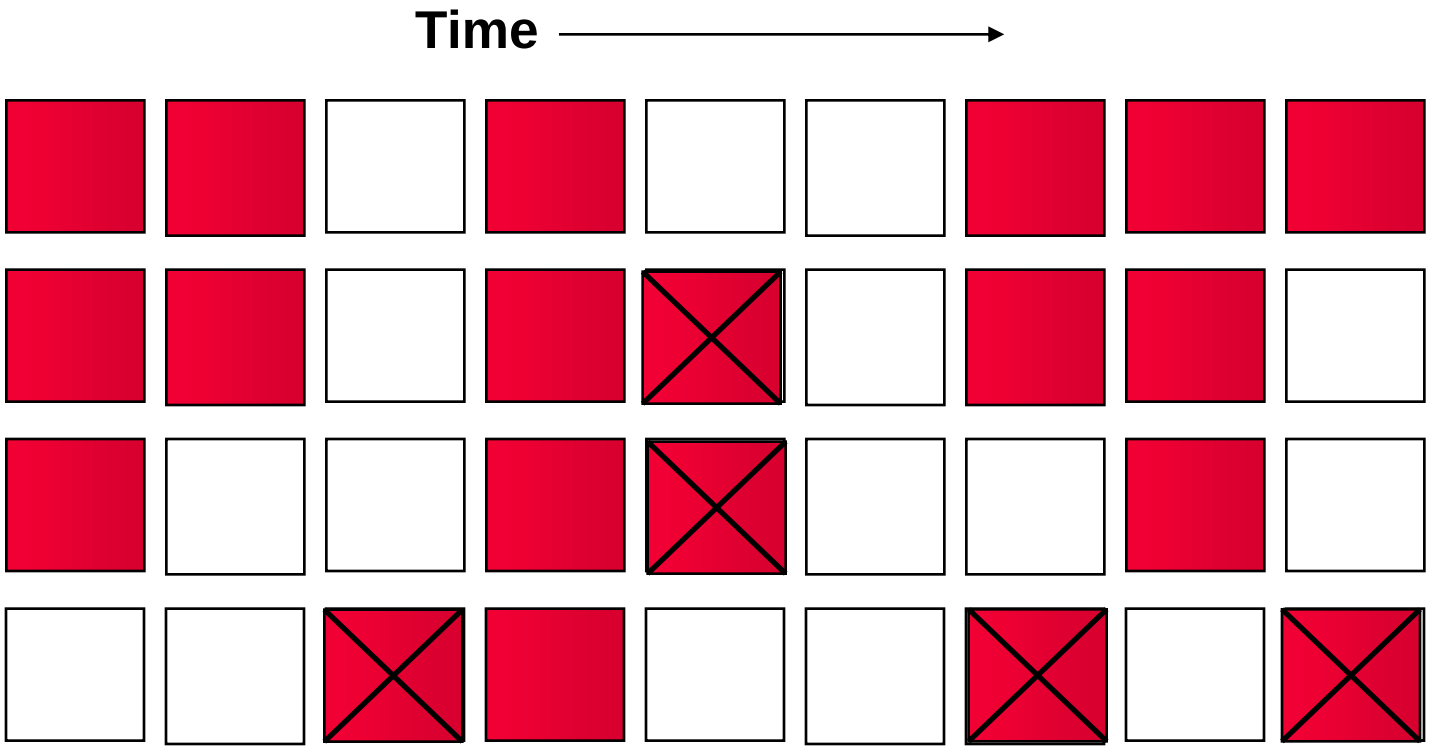
- Data dependencies reduce functional unit utilization in pipelined processors

Functional Unit Utilization in Superscalar



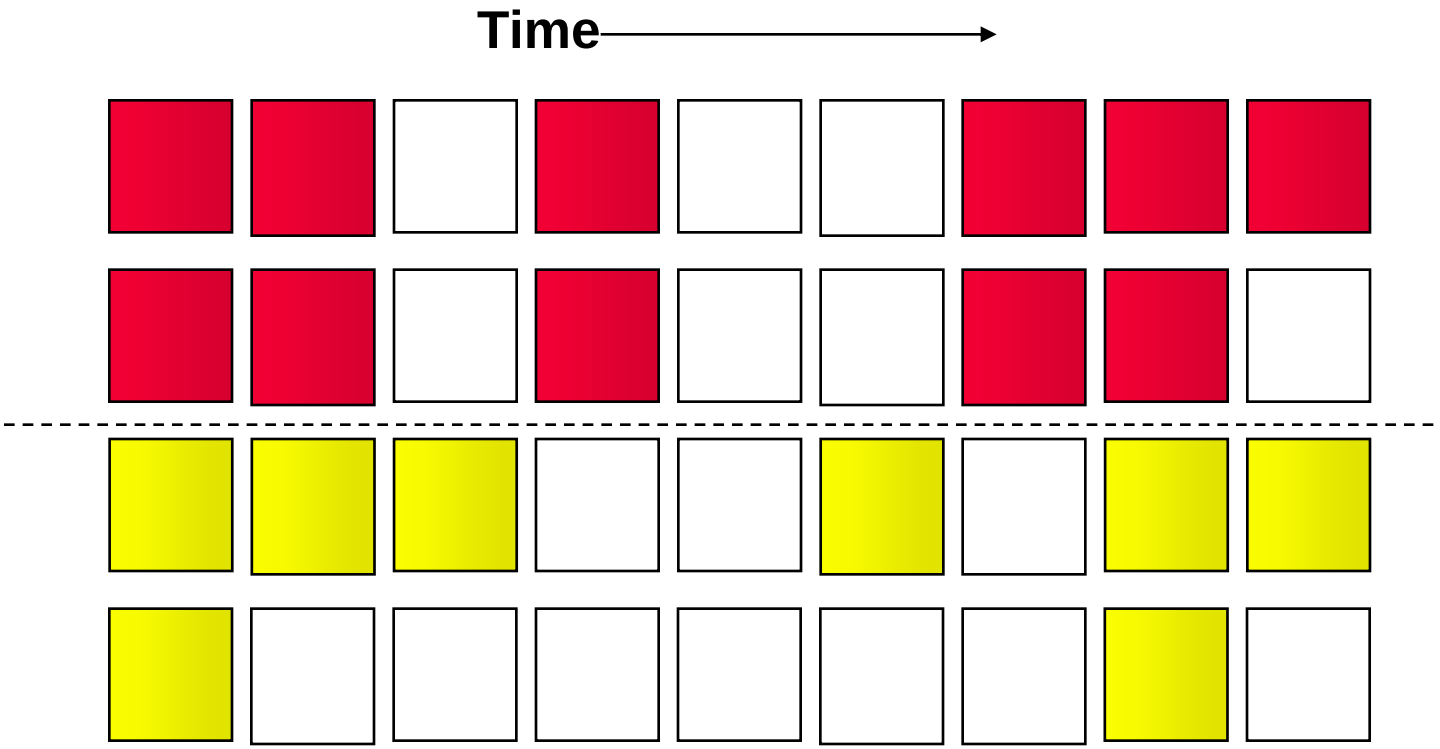
- Functional unit utilization becomes lower in superscalar, OoO machines. Finding 4 instructions in parallel is not always possible

Predicated Execution



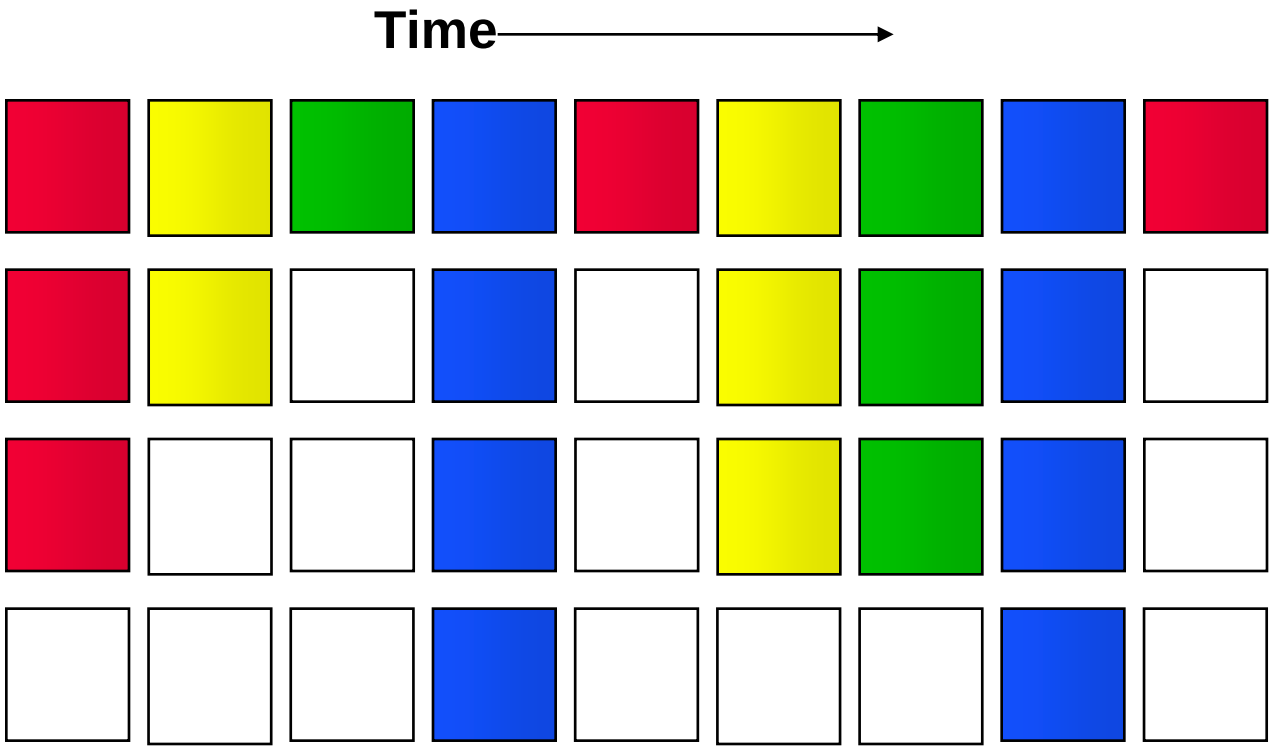
- Idea: Convert control dependencies into data dependencies
- Improves FU utilization, but some results are thrown away

Chip Multiprocessor



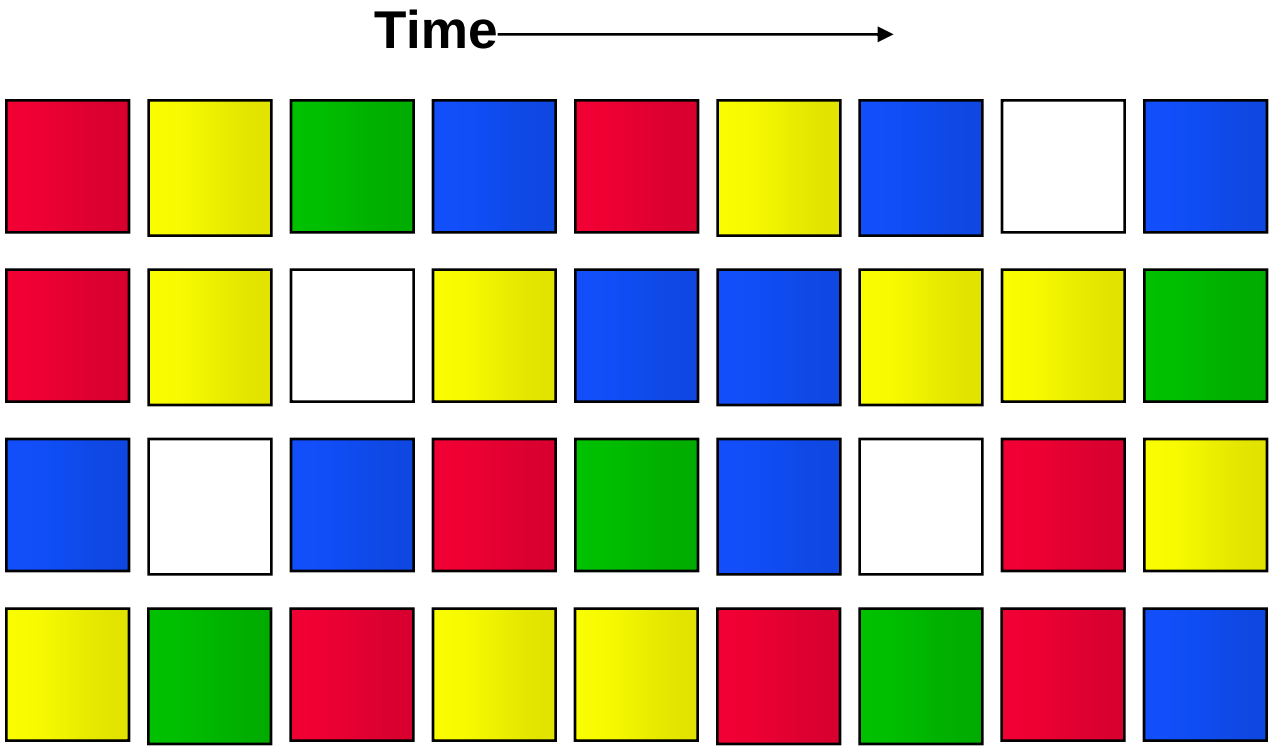
- Idea: Partition functional units across cores
- Still limited FU utilization within a single thread; limited single-thread performance

Fine-grained Multithreading



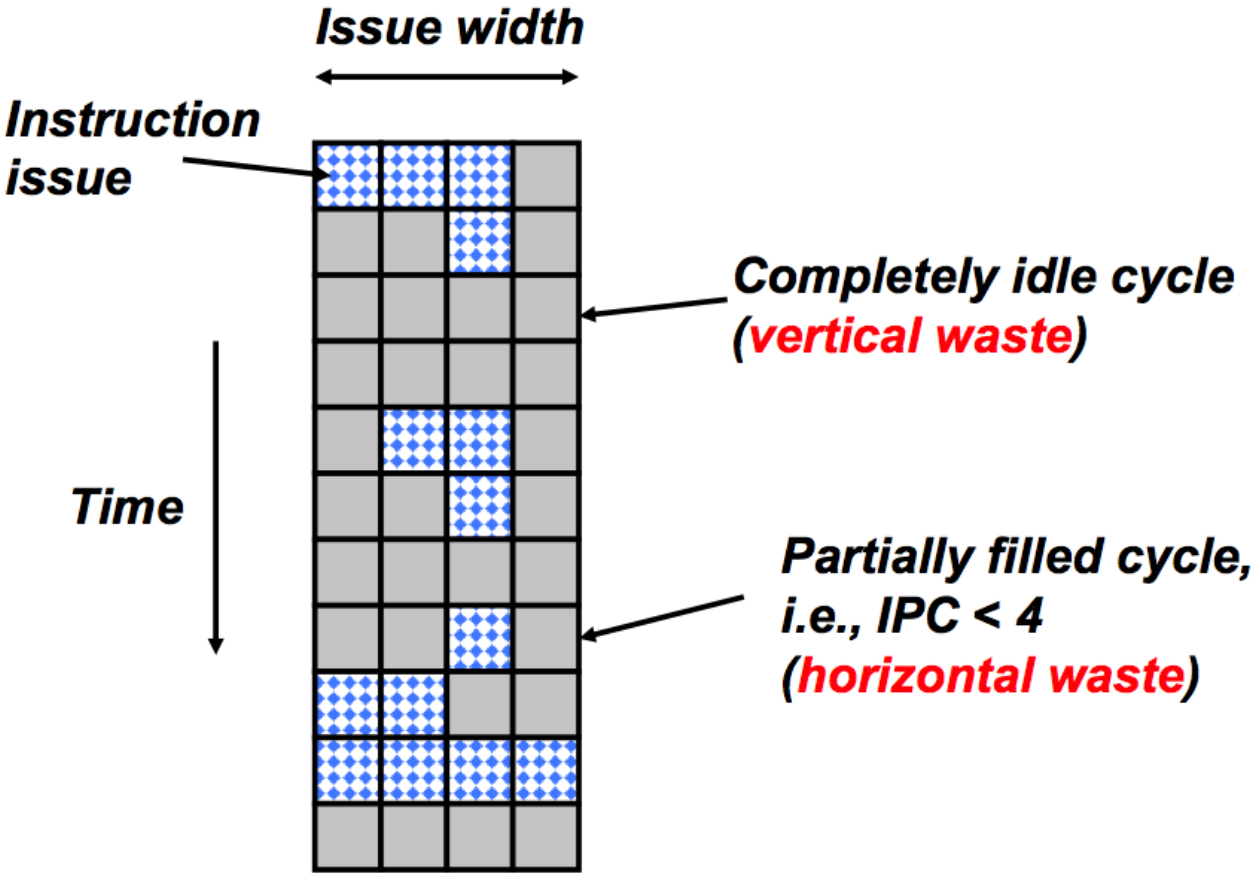
- Still low utilization due to intra-thread dependencies
- Single thread performance suffers

Simultaneous Multithreading



- Idea: Utilize functional units with independent operations from the same or different threads

Horizontal vs. Vertical Waste



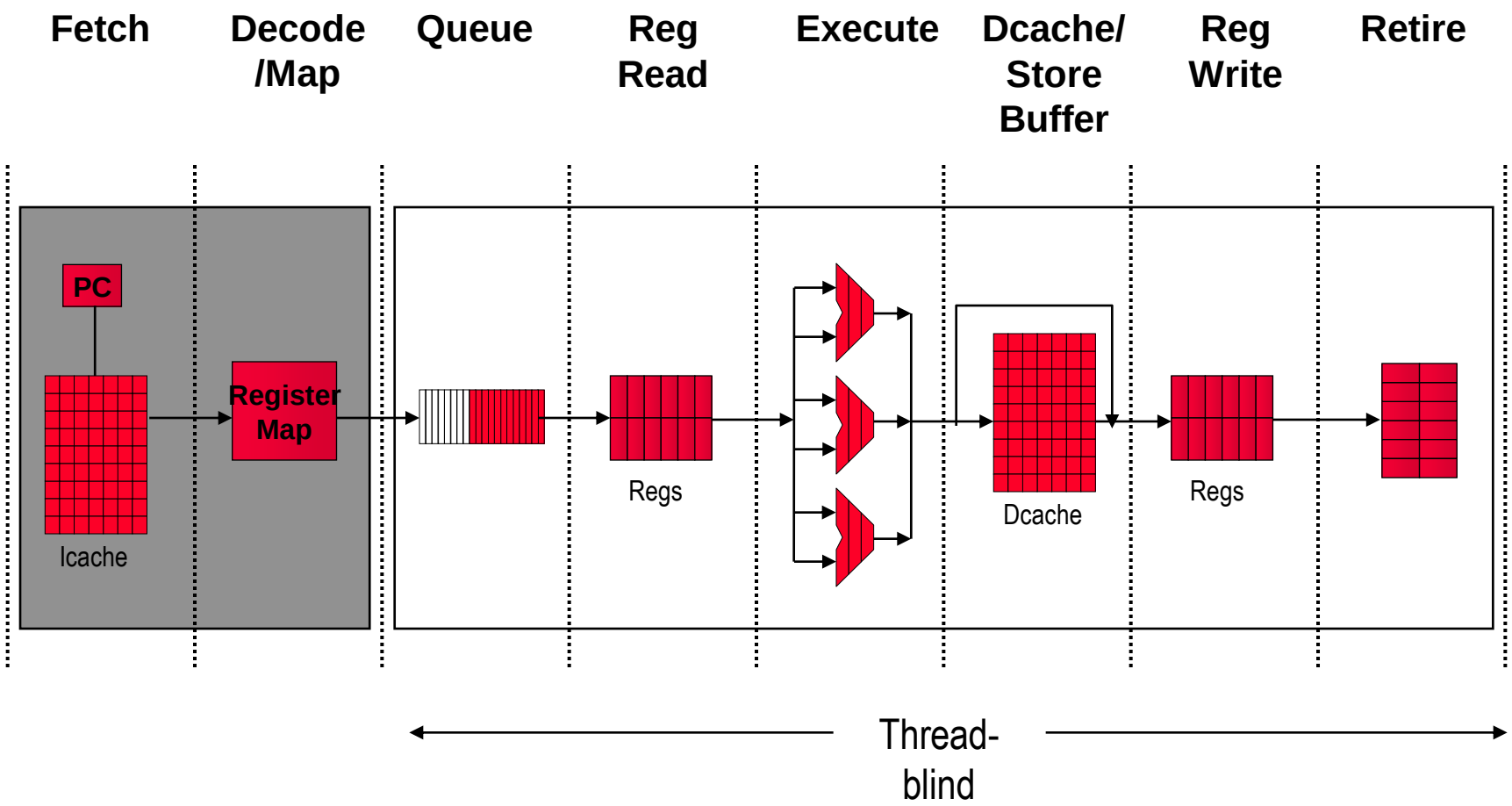
- Why is there horizontal and vertical waste?
- How do you reduce each?

Simultaneous Multithreading

- Reduces both horizontal and vertical waste
- Required hardware
  - The ability to dispatch instructions from multiple threads simultaneously into different functional units

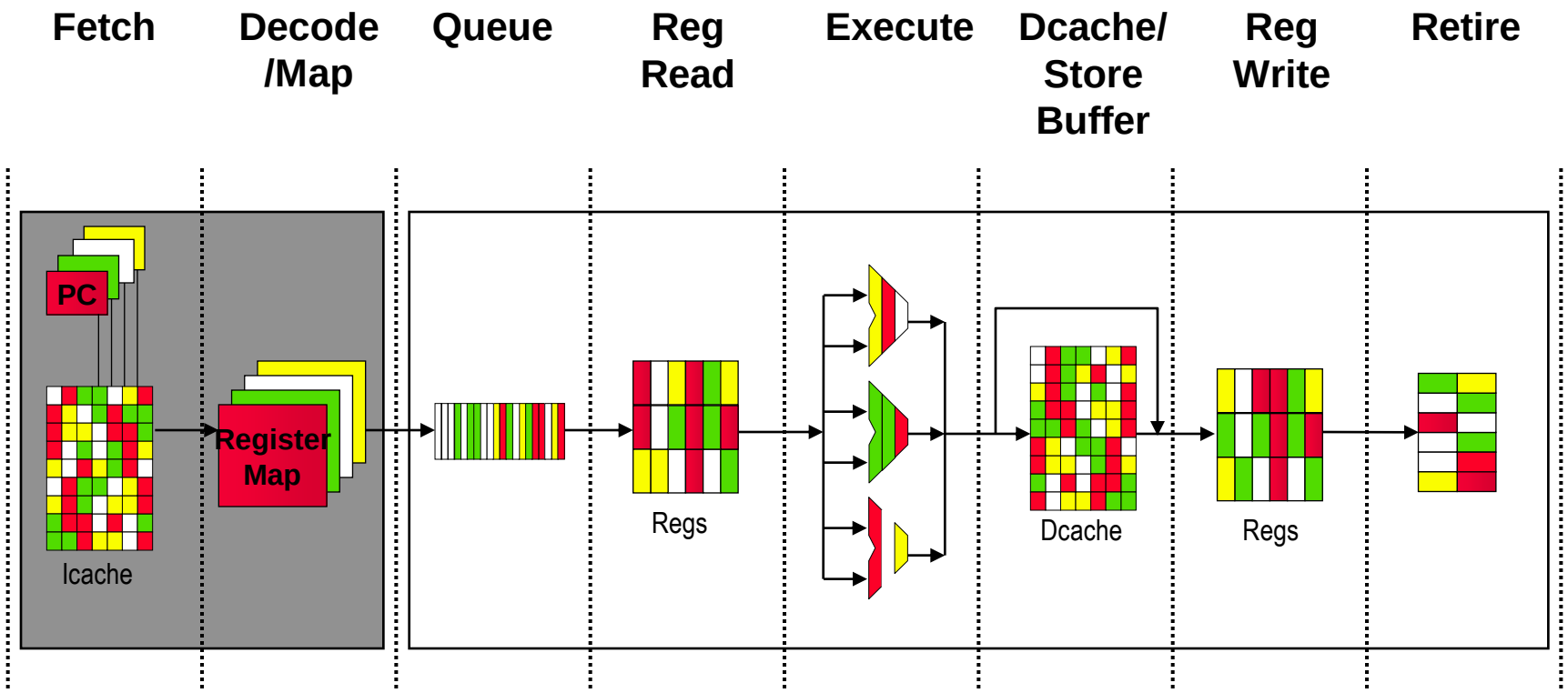
- Superscalar, OoO processors already have this machinery
  - Dynamic instruction scheduler searches the scheduling window to wake up and select ready instructions
  - As long as dependencies are correctly tracked (via renaming and memory disambiguation), scheduler can be thread-agnostic

Basic Superscalar OoO Pipeline



SMT Pipeline

- Physical register file needs to become larger. Why?



Changes to Pipeline for SMT

- Replicated resources
  - Program counter
  - Register map
  - Return address stack
  - Global history register
- Shared resources
  - Register file (size increased)
  - Instruction queue (scheduler)
  - First and second level caches
  - Translation lookaside buffers
  - Branch predictor

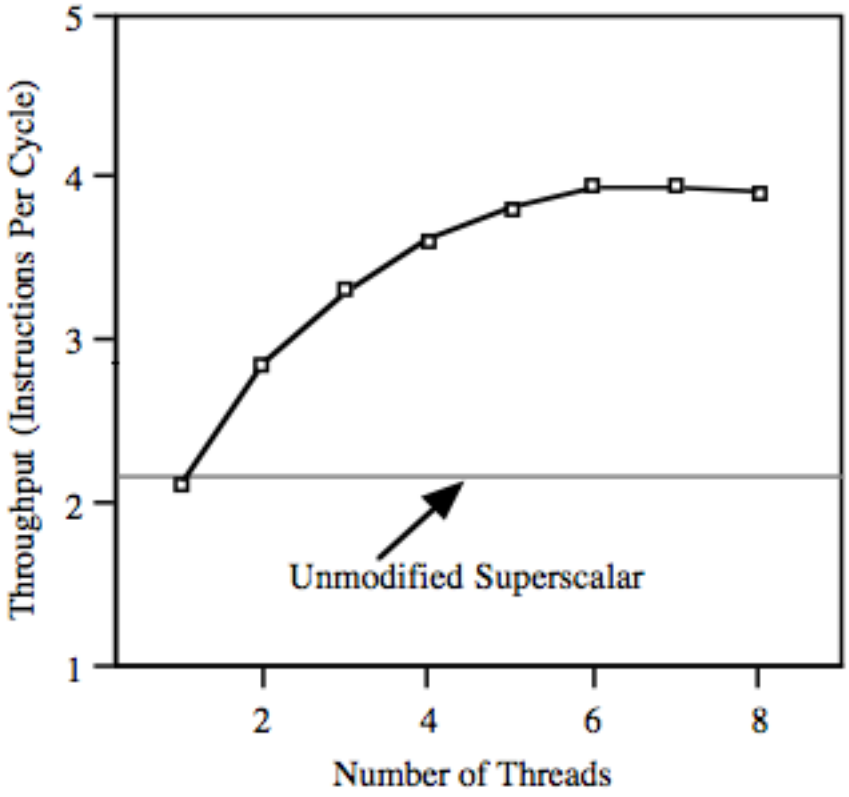
Changes to OoO+SS Pipeline for SMT

- multiple program counters and some mechanism by which the fetch unit selects one each cycle,
- a separate return stack for each thread for predicting subroutine return destinations,
- per-thread instruction retirement, instruction queue flush, and trap mechanisms,
- a thread id with each branch target buffer entry to avoid predicting phantom branches, and
- a larger register file, to support logical registers for all threads plus additional registers for register renaming. The size of the register file affects the pipeline (we add two extra stages) and the scheduling of load-dependent instructions, which we discuss later in this section.

Tullsen et al., "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," ISCA 1996.

SMT Scalability

- Diminishing returns from more threads. Why?



SMT Design Considerations

- Fetch and prioritization policies
  - Which thread to fetch from?
- Shared resource allocation policies
  - How to prevent starvation?
  - How to maximize throughput?
  - How to provide fairness/QoS?
  - Free-for-all vs. partitioned
- How to measure performance
  - Is total IPC across all threads the right metric?
- How to select threads to co-schedule
  - Snavely and Tullsen, "Symbiotic Jobscheduling for a Simultaneous Multithreading Processor," ASPLOS 2000.

Which Thread to Fetch From?

- (Somewhat) Static policies
  - Round-robin
  - 8 instructions from one thread
  - 4 instructions from two threads
  - 2 instructions from four threads
  - …
- Dynamic policies
  - Favor threads with minimal in-flight branches
  - Favor threads with minimal outstanding misses
  - Favor threads with minimal in-flight instructions
  - …

Which Instruction to Select/Dispatch?

- Can be thread agnostic.
- Why?

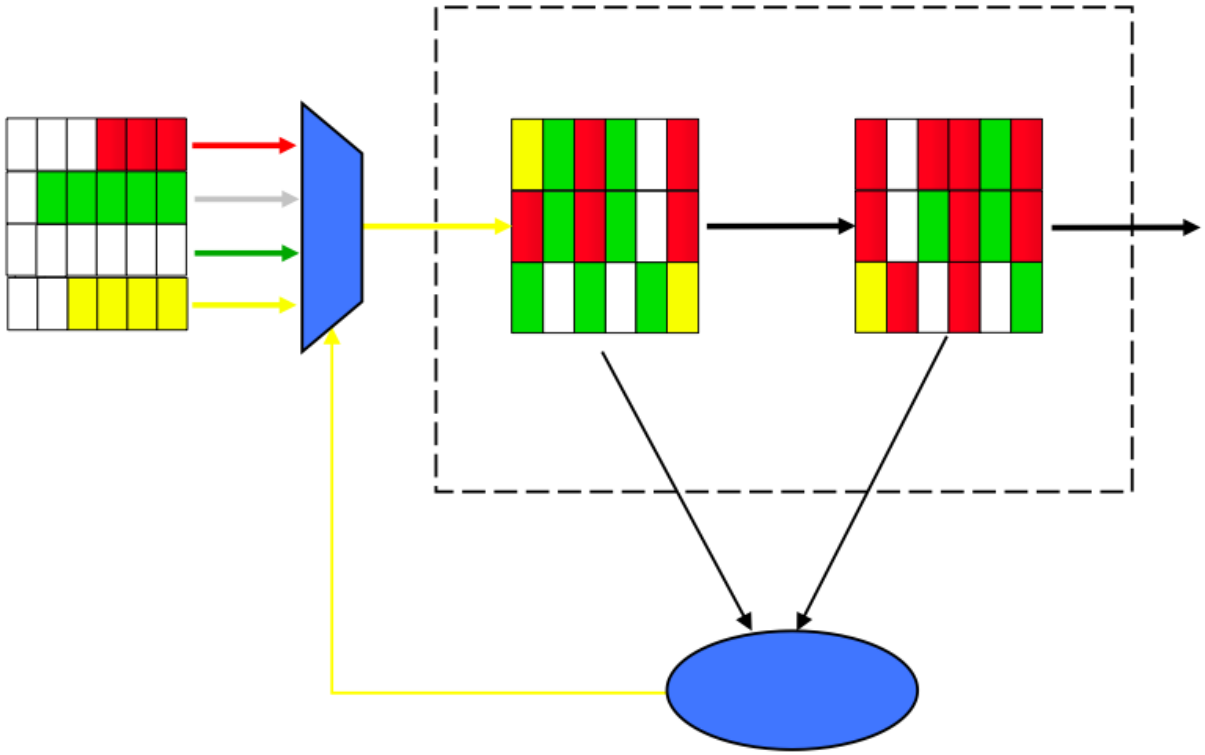
SMT Fetch Policies (I)

- Round robin: Fetch from a different thread each cycle
- Does not work well in practice. Why?

- Instructions from slow threads hog the pipeline and block the instruction window
  - E.g., a thread with long-latency cache miss (L2 miss) fills up the window with its instructions
  - Once window is full, no other thread can issue and execute instructions and the entire core stalls

SMT Fetch Policies (II)

- ICOUNT: Fetch from thread with the least instructions in the earlier pipeline stages (before execution)



- Why does this improve throughput?

SMT ICOUNT Fetch Policy

- Favors faster threads that have few instructions waiting

- Advantages over round robin

  + Allows faster threads to make more progress (before threads with long-latency instructions block the window fast)

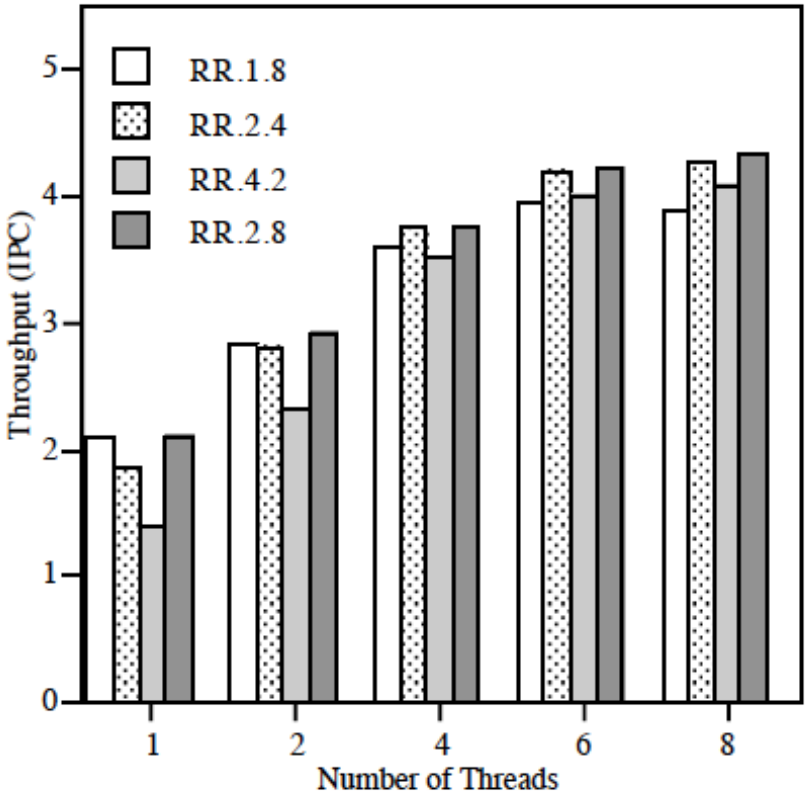
    + Higher IPC throughput

- Disadvantages over round robin

  - Is this fair?

  - Prone to short-term starvation: Need additional methods to ensure starvation freedom

Some Results on Fetch Policy



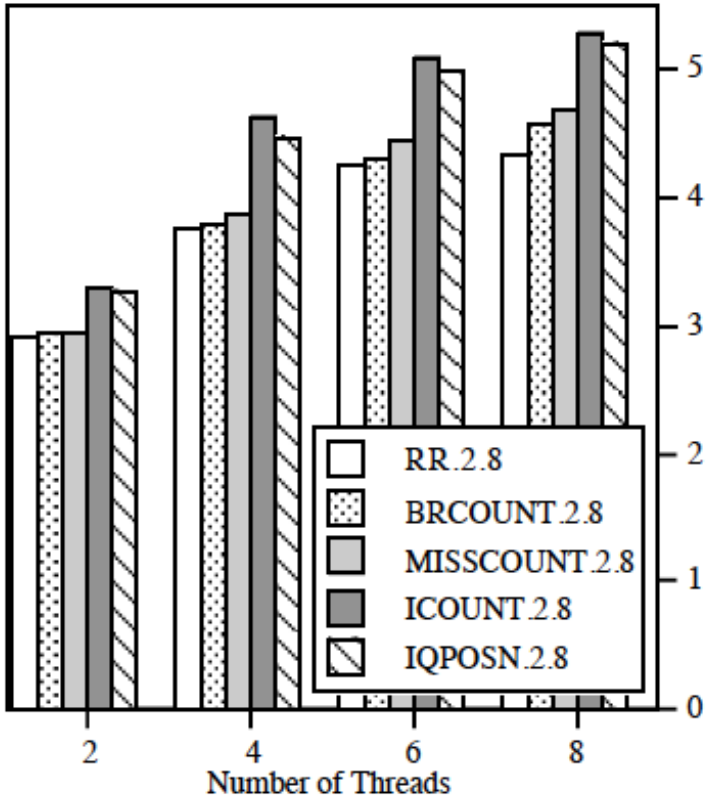
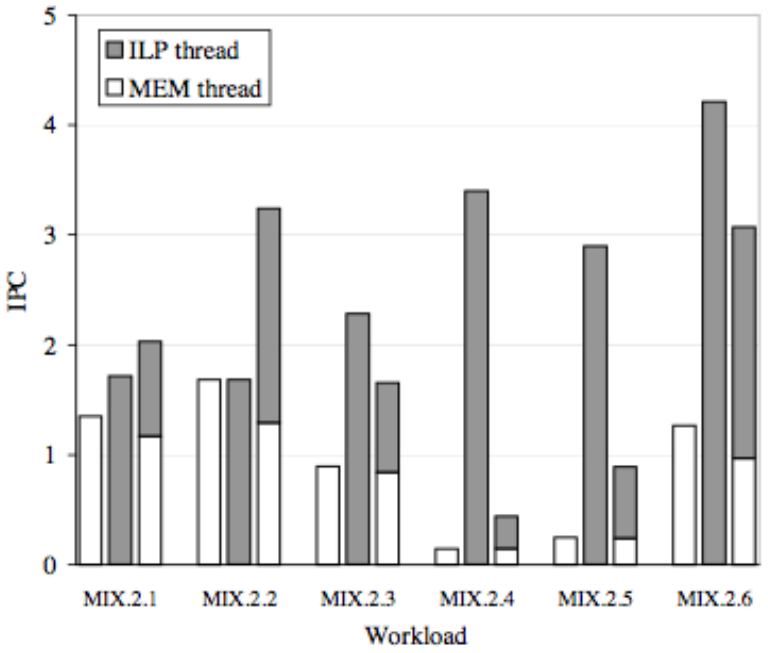


Figure 4: **Instruction throughput for the different instruction cache interfaces with round-robin instruction scheduling.**

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.



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.

MLP-Aware Fetch Policies

- The *stall fetch* approach proposed by Tullsen and Brown [24], *i.e.*, a thread that experiences a long-latency load is fetch stalled until the data returns from memory.
- The *predictive stall fetch* approach, following [2], extends the above stall fetch policy by predicting long-latency loads in the front-end pipeline. Predicted long-latency loads trigger fetch stalling a thread.
- The *MLP-aware stall fetch* approach predicts long-latency loads, predicts the amount of MLP for predicted long-latency loads and fetch stalls threads when the number of instructions has been fetched as predicted by the MLP predictor.
- The *flush* approach proposed by Tullsen and Brown [24] flushes on long-latency loads. Our implementation flushes when a long-latency load is detected (this is the 'TM' or trigger on long-latency miss in [24]) and flushes starting from the instruction following the long-latency load (this is the 'next' approach in [24]).
- The *MLP-aware flush* approach predicts the amount of MLP for a long-latency load, and fetch stalls or flushes the thread after m instructions since the long-latency load, with m the MLP distance predicted by the MLP predictor.

Eyerman and Eeckhout, "A Memory-Level Parallelism Aware Fetch Policy for SMT Processors," HPCA 2007.

More Results …

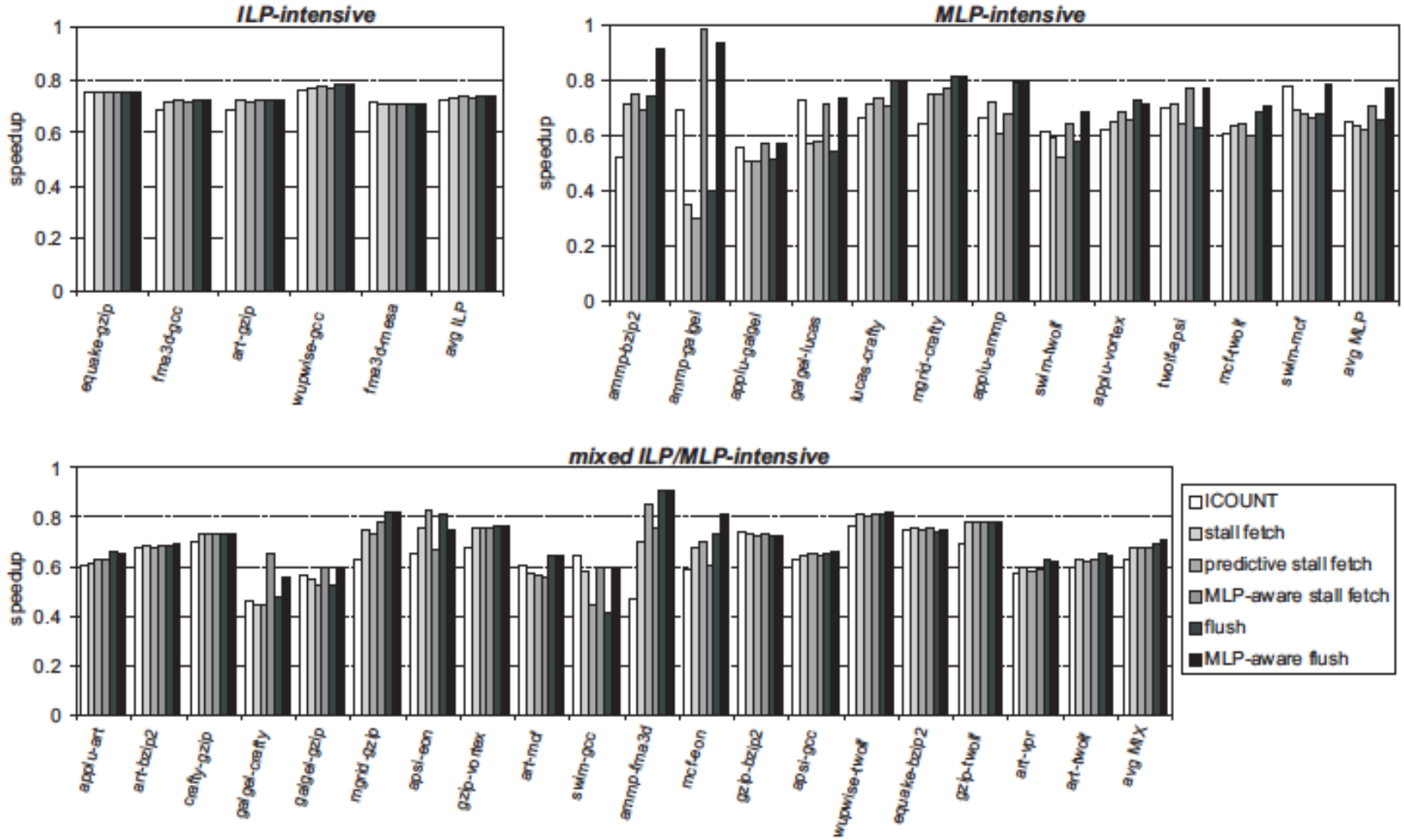


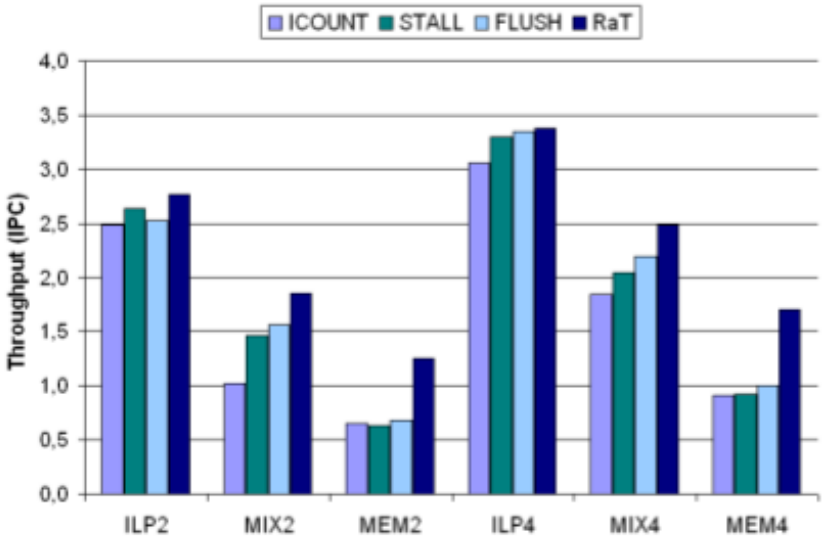
Figure 5. The speedup for the various SMT fetch policies compared to single-threaded execution for the 2-thread workloads.

Runahead Threads

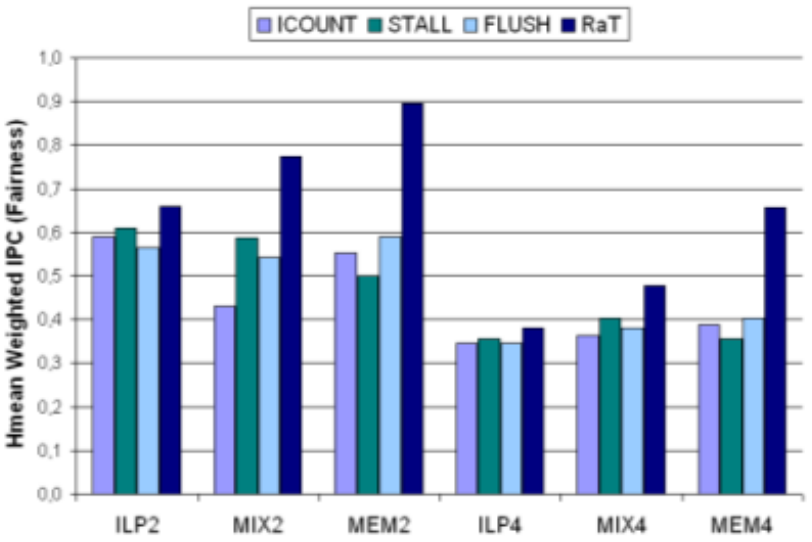
- Idea: Use runahead execution on a long-latency load

+ Improves both single thread and multi-thread performance

- Ramirez et al., "Runahead Threads to Improve SMT Performance," HPCA 2008.



(a) Throughput (IPC)



(b) Fairness

Doing Even Better

- Predict whether runahead will do well
- If so, runahead on thread until runahead becomes useless
- Else, exploit and flush thread

- Ramirez et al., "Efficient Runahead Threads," PACT 2010.
- Van Craeynest et al., "MLP-Aware Runahead Threads in a Simultaneous Multithreading Processor," HiPEAC 2009.

Commercial SMT Implementations

- Intel Pentium 4 (Hyperthreading)
- IBM POWER5
- Intel Nehalem
- …

SMT in IBM POWER5

- Kalla et al., "IBM Power5 Chip: A Dual-Core Multithreaded Processor," IEEE Micro 2004.

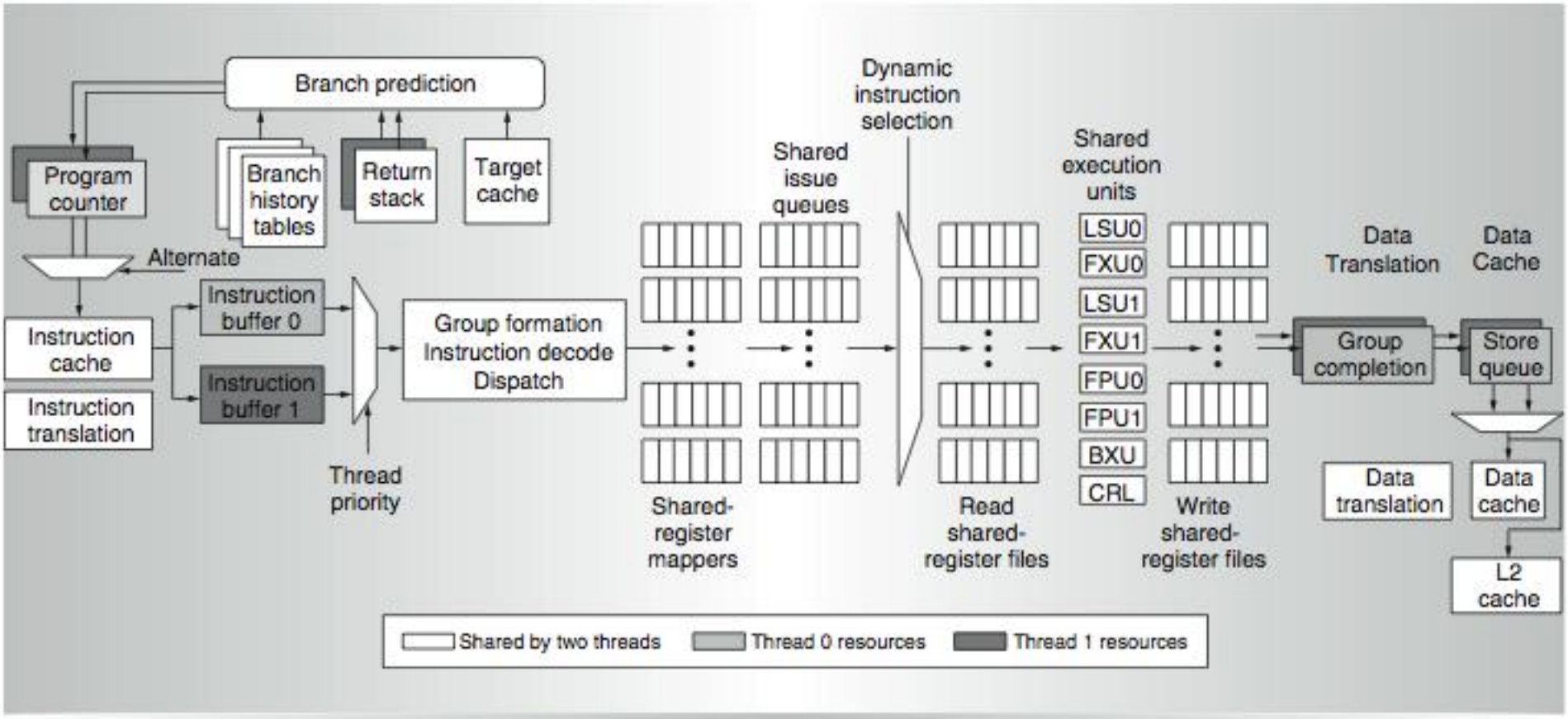


Figure 4. Power5 instruction data flow (BXU = branch execution unit and CRL = condition register logical execution unit).

IBM POWER5 HW Thread Priority Support

- Adjust decode cycles dedicated to thread based on priority level

- Why?
- A thread is in a spin loop waiting for a lock
- A thread has no immediate work to do and is waiting in an idle loop
- One application is more important than another

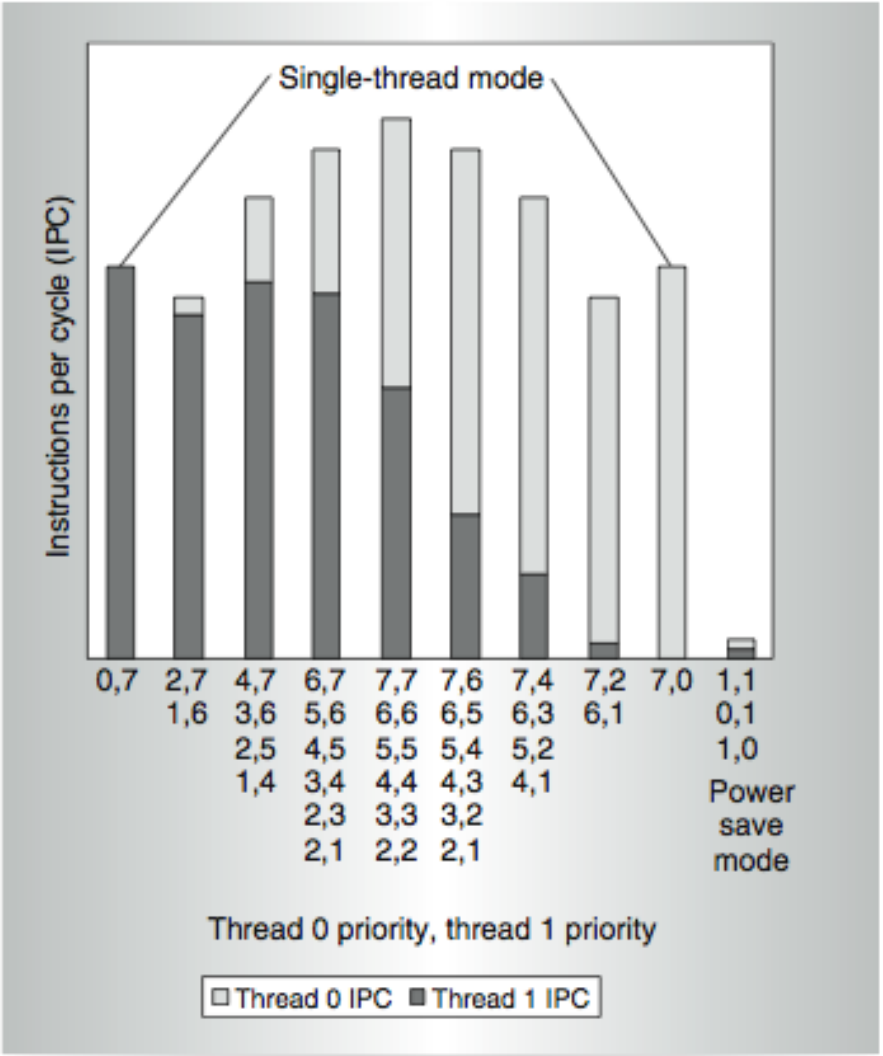


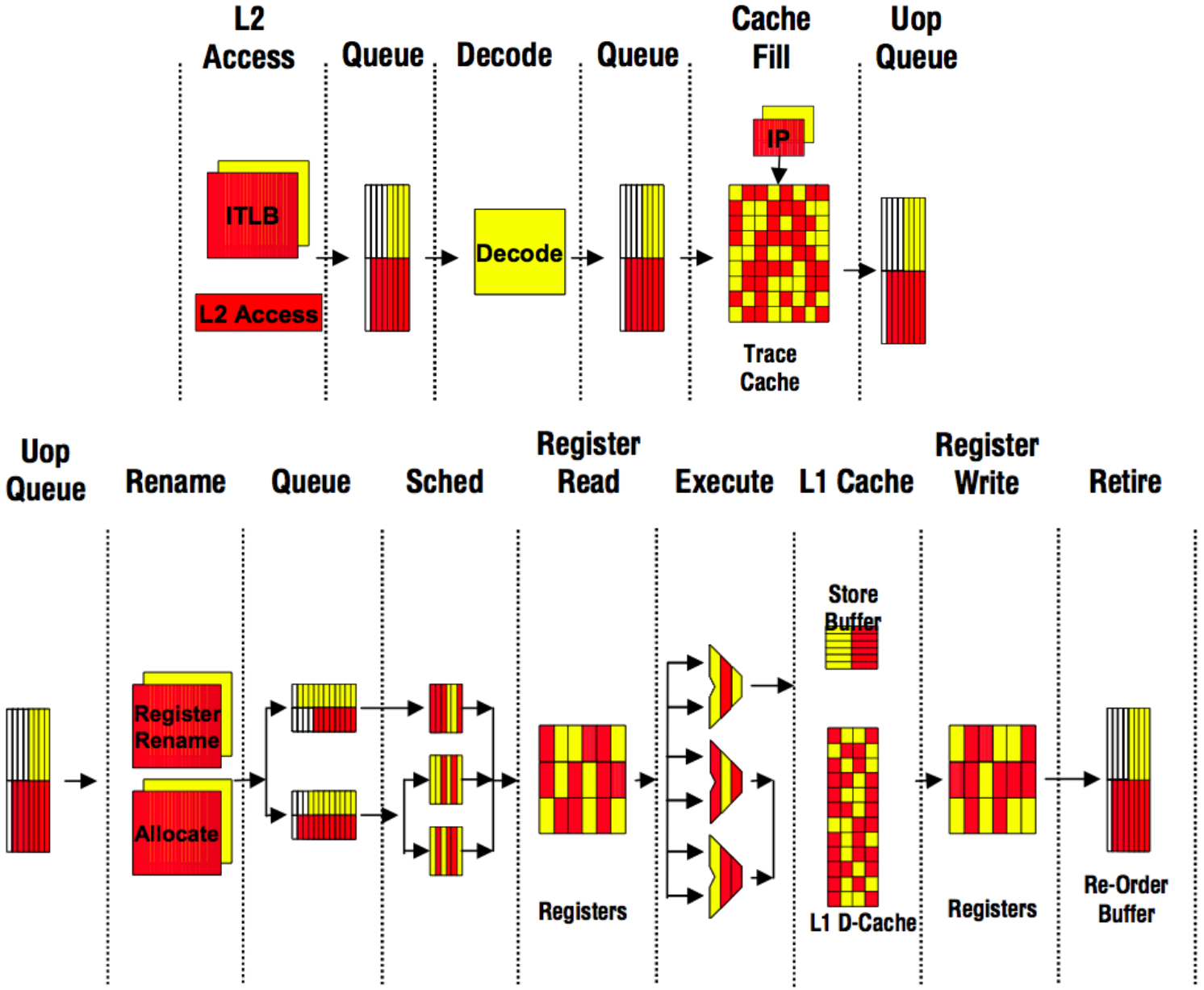
Figure 5. Effects of thread priority on performance.

IBM POWER5 Thread Throttling

- Throttle under two conditions:
  - Resource-balancing logic detects the point at which a thread reaches a threshold of load misses in the L2 cache and translation misses in the TLB.
  - Resource-balancing logic detects that one thread is beginning to use too many GCT (i.e., reorder buffer) entries.

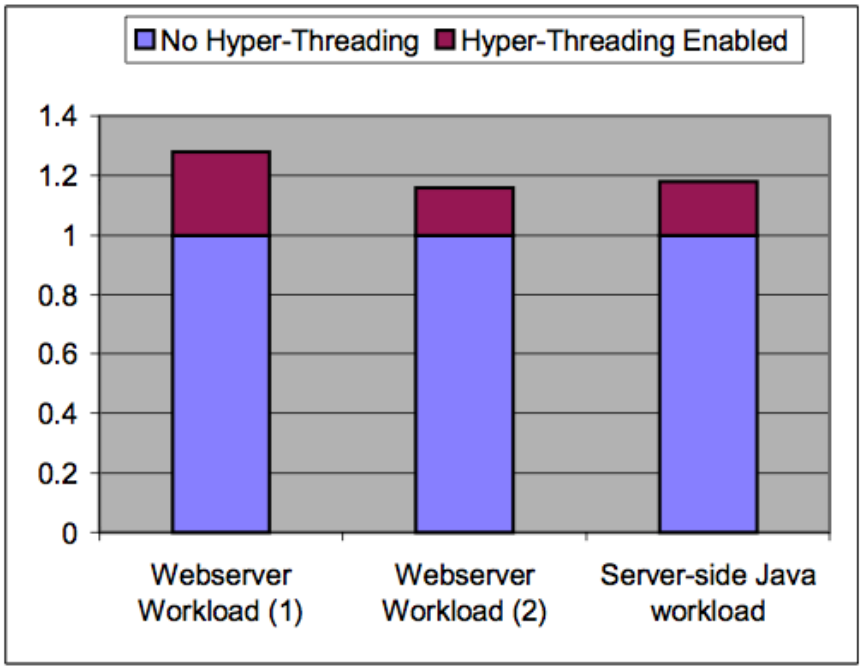
- Throttling mechanisms:
  - Reduce the priority of the thread
  - Inhibit the instruction decoding of the thread until the congestion clears
  - Flush all of the thread's instructions that are waiting for dispatch and stop the thread from decoding additional instructions until the congestion clears

Intel Pentium 4 Hyperthreading



Intel Pentium 4 Hyperthreading

- Long latency load handling
  - Multi-level scheduling window
- More partitioned structures
  - I-TLB
  - Instruction Queues
  - Store buffer
  - Reorder buffer
- 5% area overhead due to SMT



- Marr et al., "Hyper-Threading Technology Architecture and Microarchitecture," Intel Technology Journal 2002.

Other Uses of Multithreading

Now that We Have MT Hardware …

- … what else can we use it for?
- Redundant execution to tolerate soft (and hard?) errors
- Implicit parallelization: thread level speculation
  - Slipstream processors
  - Leader-follower architectures
- Helper threading
  - Prefetching
  - Branch prediction
- Exception handling