Multithreading

Lokwon Kim

Multithreading: Basics

- **Thread**
  - Instruction stream with state (registers and memory)
  - Register states are also called “thread context”

- Threads could be part of the same process (program) or from different programs
  - Threads in the same program share the same address space (shared memory model)

- Traditionally, the processor keeps track of the context of a single thread
- Thread context switching for multitasking: When a new thread needs to be executed, old thread’s context in hardware written back to memory and new thread’s context loaded

Hardware Multithreading

- General idea: **Have multiple thread contexts in a single processor**
  - When the hardware executes from those hardware contexts determines the granularity of multithreading

- Why?
  - **To tolerate latency** (e.g. **reducing pipeline stalls**)
    - Latency of memory operations, dependent instructions, branch resolution
    - To utilize processing resources more efficiently
  - In other words, **to improve system throughput**
    - To exploit thread-level parallelism
    - To improve superscalar/OoO processor utilization
  - **To reduce context switch penalty**

Initial Motivations

- Tolerate latency
  - **When one thread encounters a long-latency operation, the processor can execute a useful operation from another thread**

Hardware Multithreading

- Benefit
  + Latency tolerance (by running other thread work durind the delay time)
  + Better hardware utilization (because of reduced pipeline stalls)
  + Reduced context switch penalty (having more register file resources for multiple threads)

- Cost
  - High HW cost! - Requires multiple thread contexts to be implemented in hardware (area, power, latency cost)
  - Usually reduced single-thread performance
    - Resource sharing, contention
    - Switching penalty (can be reduced with additional hardware)

Types of Multithreading

- Fine-grained MT
  - Cycle by cycle
- Coarse-grained MT
  - Switch on event (e.g., cache miss)
  - Switch on quantum/timeout
- Simultaneous MT
  - Instructions from multiple threads executed concurrently in the same cycle

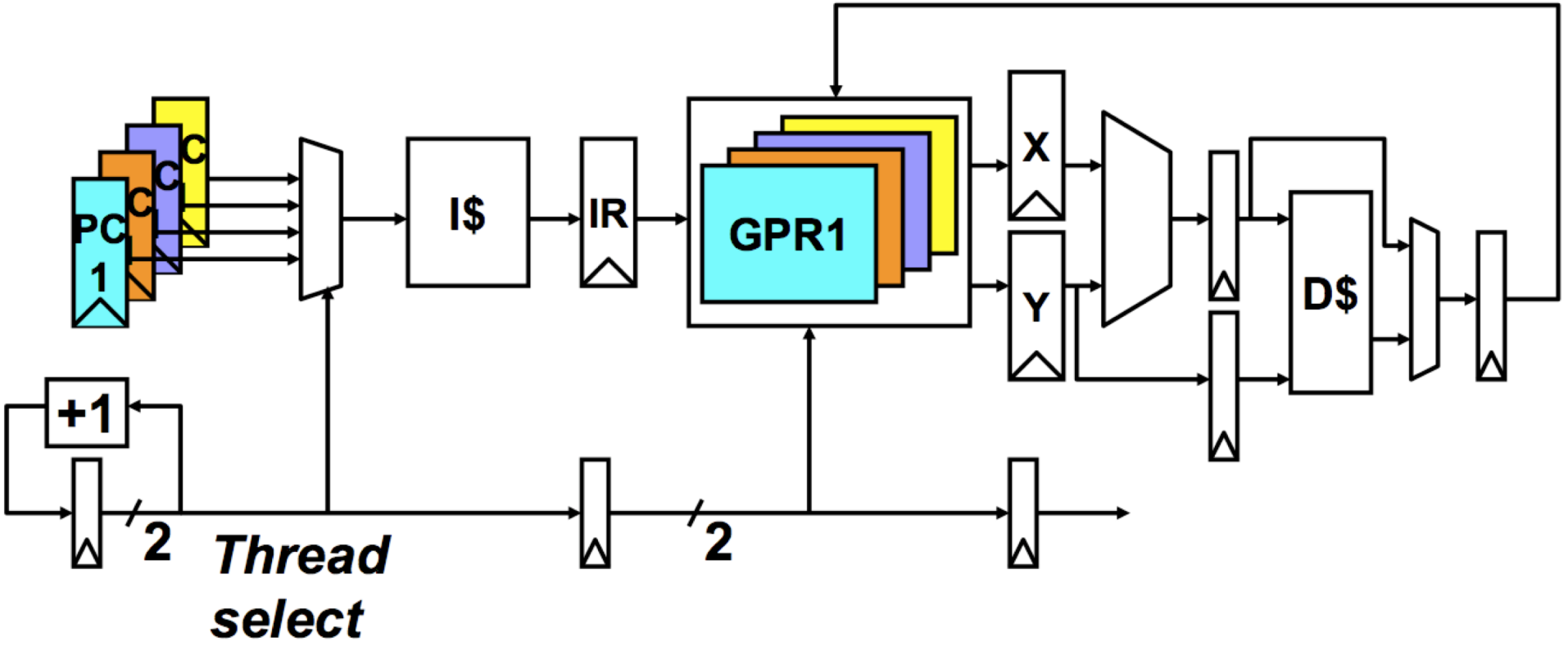
Fine-grained Multithreading

- Idea: Switch to another thread every clock cycle such that no instructions from the thread are in other pipeline stages concurrently

- **Improves pipeline utilization by taking advantage of multiple threads**
- Alternative way of looking at it: **Eliminate the control and data hazards among pipeline stages** by overlapping the latency with useful work from other threads

- Thornton, "Parallel Operation in the Control Data 6600," AFIPS 1964.
- Smith, "A pipelined, shared resource MIMD computer," ICPP 1978.

Multithreaded Pipeline Example



- Slide from Joel Emer

Fine-grained Multithreading

- Advantages
  + No need for dependency checking between instructions in different pipeline stages.

    (only one instruction in pipeline from a single thread)
  + No need for branch prediction logic for instructions in different pipeline stages.
  + Otherwise-bubble cycles used for executing useful instructions from different threads
  + Improved system throughput, latency tolerance, utilization
- Disadvantages
  - Extra hardware complexity: multiple hardware contexts, thread selection logic
  - Reduced single thread performance (one instruction fetched every N cycles)
  - Increased resource contention between threads in caches and memory
  - Dependency checking logic between threads remains (load/store)
  - **Dependency checking and branch prediction for instructions in a pipeline stage still remain (In case of SuperScalar)**

An example: Tera MTA Fine-grained Multithreading

- 256 processors, each with a 21-cycle pipeline
- 128 active threads
- A thread can issue instructions every 21 cycles
  - Then, why 128 threads?

- Memory latency: approximately 150 cycles
  - No data cache
  - Threads can be blocked waiting for memory
  - More threads → better ability to tolerate memory latency

- Thread state per processor
  - 128 x 32 general purpose registers
  - 128 x 1 thread status registers

Coarse-grained Multithreading

- Idea: When a thread is stalled due to some events (incurring long latency accordingly pipelin stalls), switch to a different hardware context
  - **Switch-on-event multithreading**

- Possible stall events
  - Cache misses
  - Synchronization events (e.g., load an empty location)
  - FP operations
  - Accessing slow I/O devices

Fine-grained vs. Coarse-grained MT

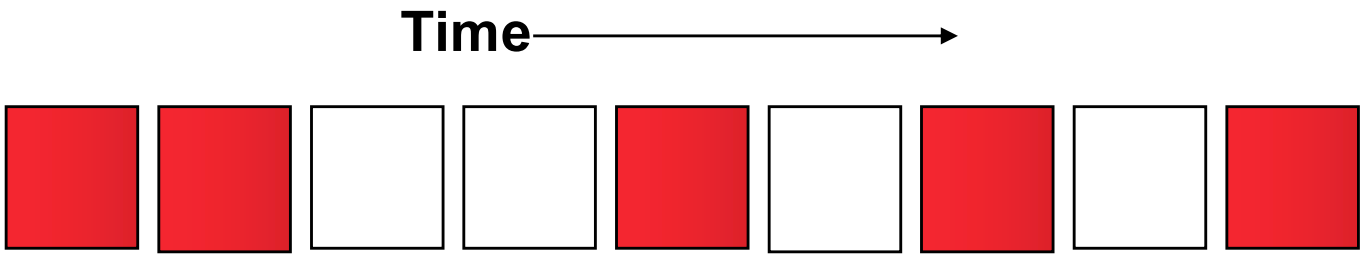
- Fine-grained advantages compared to Coarse-grained
  - + Simpler to implement, (can eliminate dependency checking, branch prediction logic completely in no superscalar processors)
  - + Switching need not have any performance overhead (i.e. dead cycles)
    - + Coarse-grained requires a pipeline flush or a lot of hardware to save pipeline state
      - → Higher performance degradation with deep pipelines and large windows

- Fine-grained Disadvantages
  - - Low single thread performance: each thread gets at most 1/Nth of the bandwidth of the pipeline (Nth is the number of pipeline stages)

Simultaneous Multithreading (SMT)

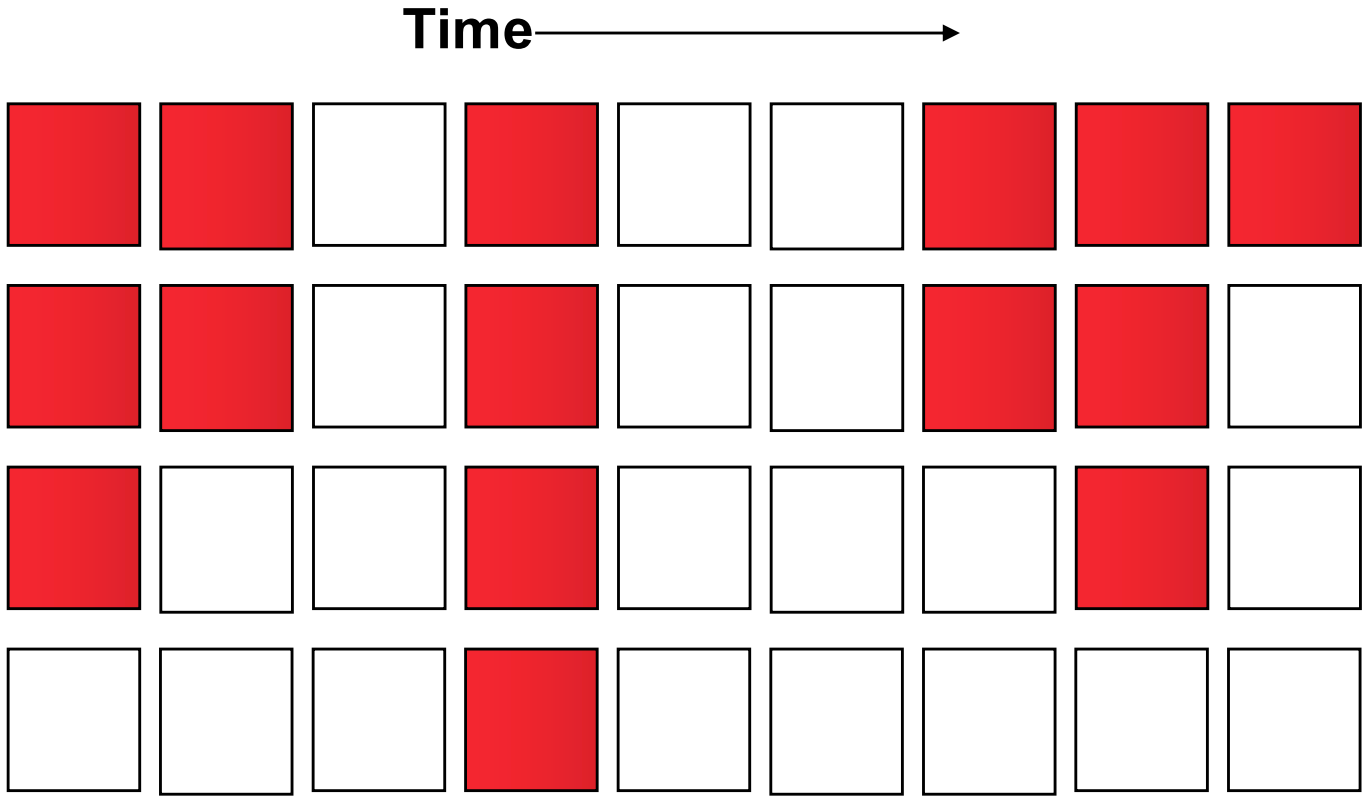
- Fine-grained and coarse-grained multithreading can start execution of instructions from *only* a single thread at a given cycle
- In FG and CG MT methods, execution unit (or pipeline stage) utilization can be low if there are not enough instructions from a thread to "dispatch" in one cycle
  - In a 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 in single channel pipeline



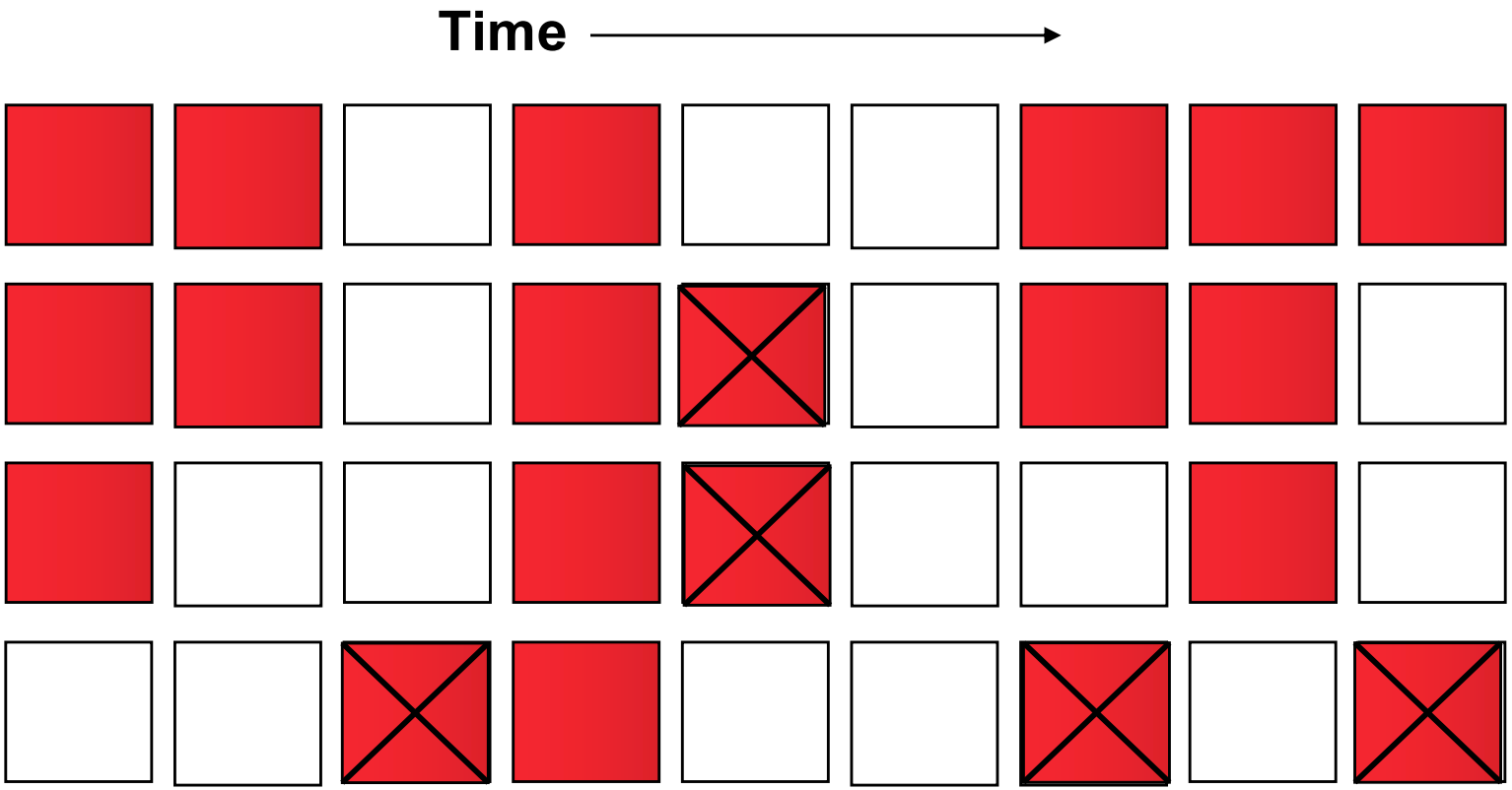
- Data dependencies reduce functional unit utilization in pipelined processors

Functional Unit Utilization in Superscalar



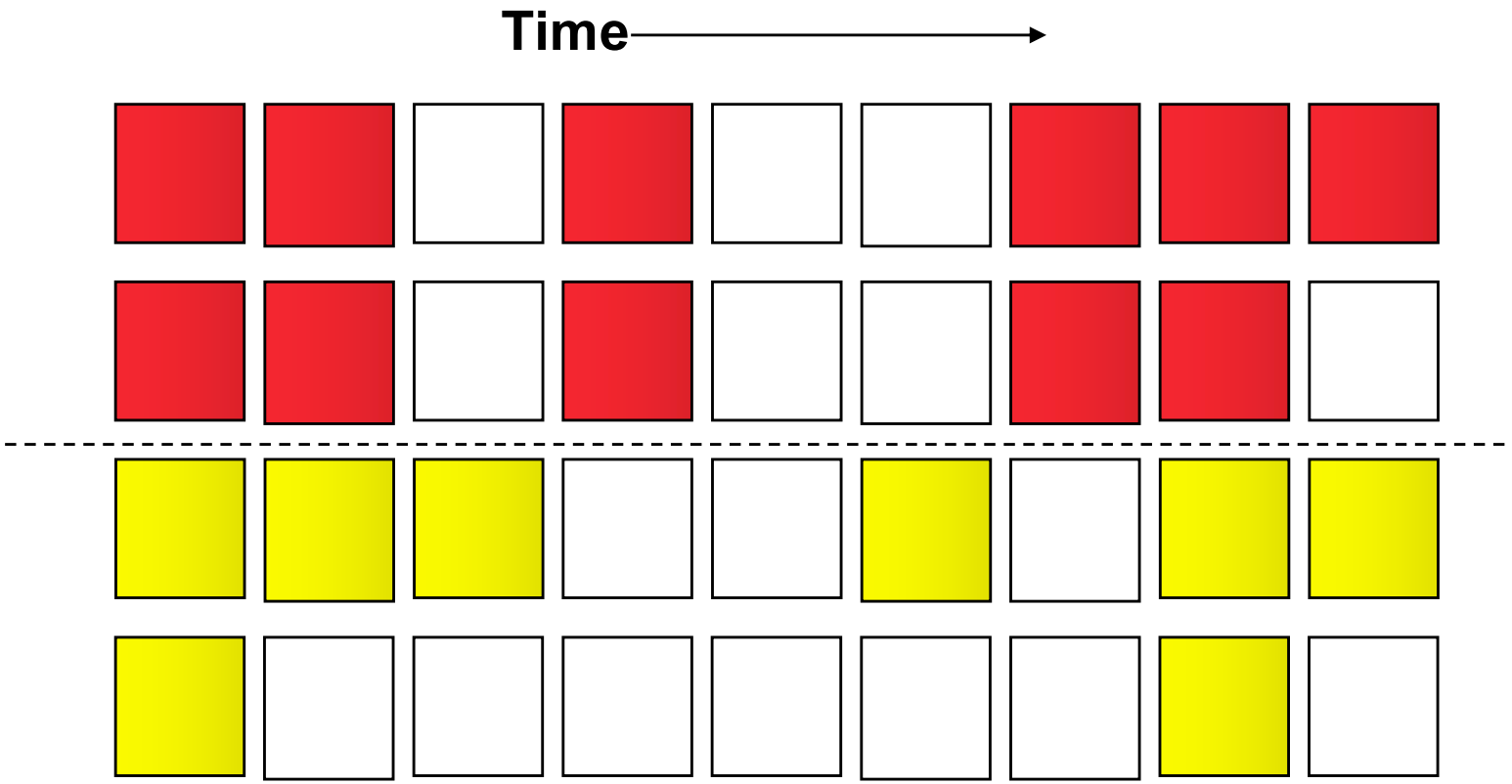
- **Functional unit utilization becomes lower in superscalar, OoO machines than the simple single channel pipeline.** *(*Finding 4 instructions in parallel is harder than finding one instruction)

Predicated Execution



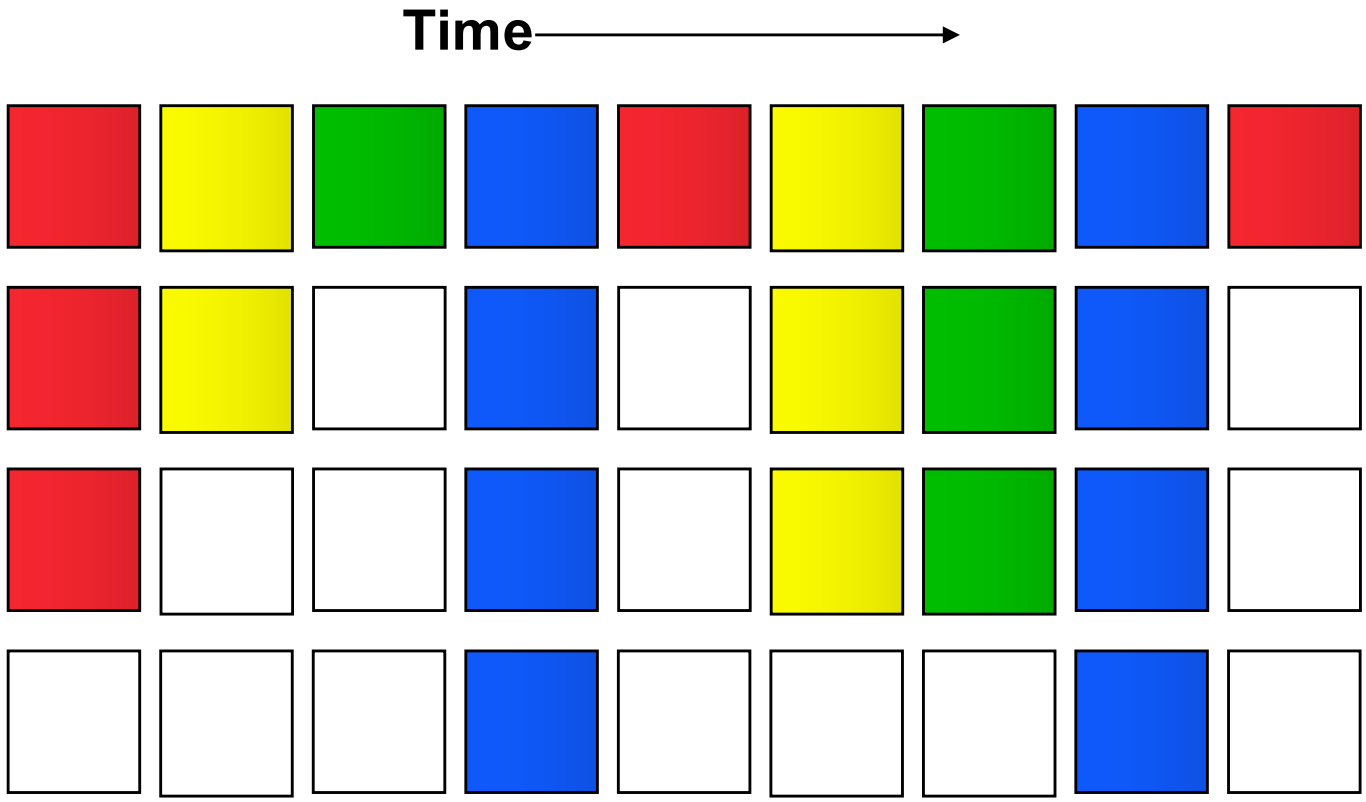
- Idea: Convert control dependencies into data dependencies
- It looks improved FU utilization, but some of the instructions are actually NOP

Chip Multiprocessor



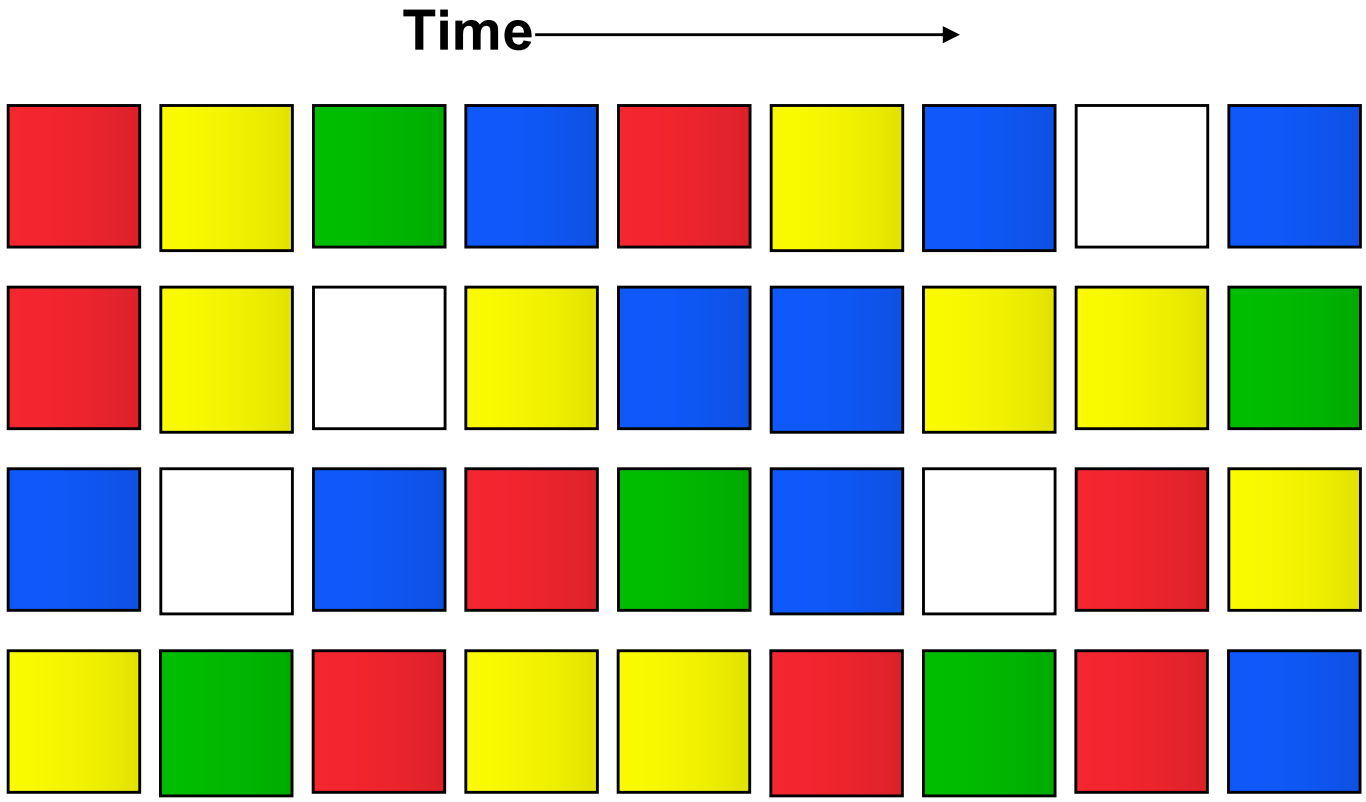
- Idea: Partition functional units across cores
- Still limited FU utilization within a single thread; limited single-thread performance

Fine-grained Multithreading



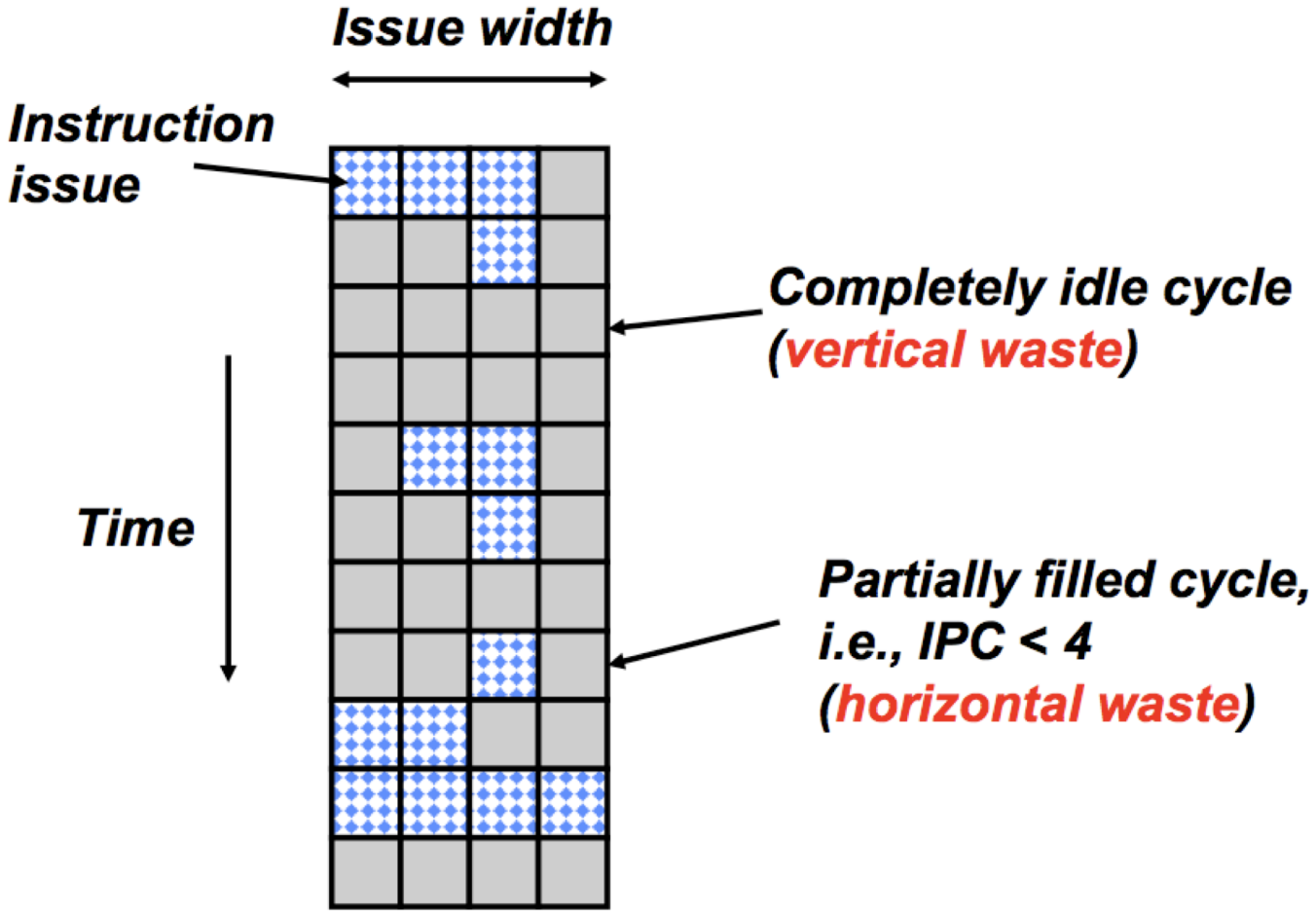
- **Far better than single tread one, but still low utilization due to intra-thread dependencies**
- Single thread performance suffers

Simultaneous Multithreading



- **Idea: Utilize functional units with independent operations from the same or different threads**
- **Best performance but the highest HW cost.**

Horizontal vs. Vertical Waste



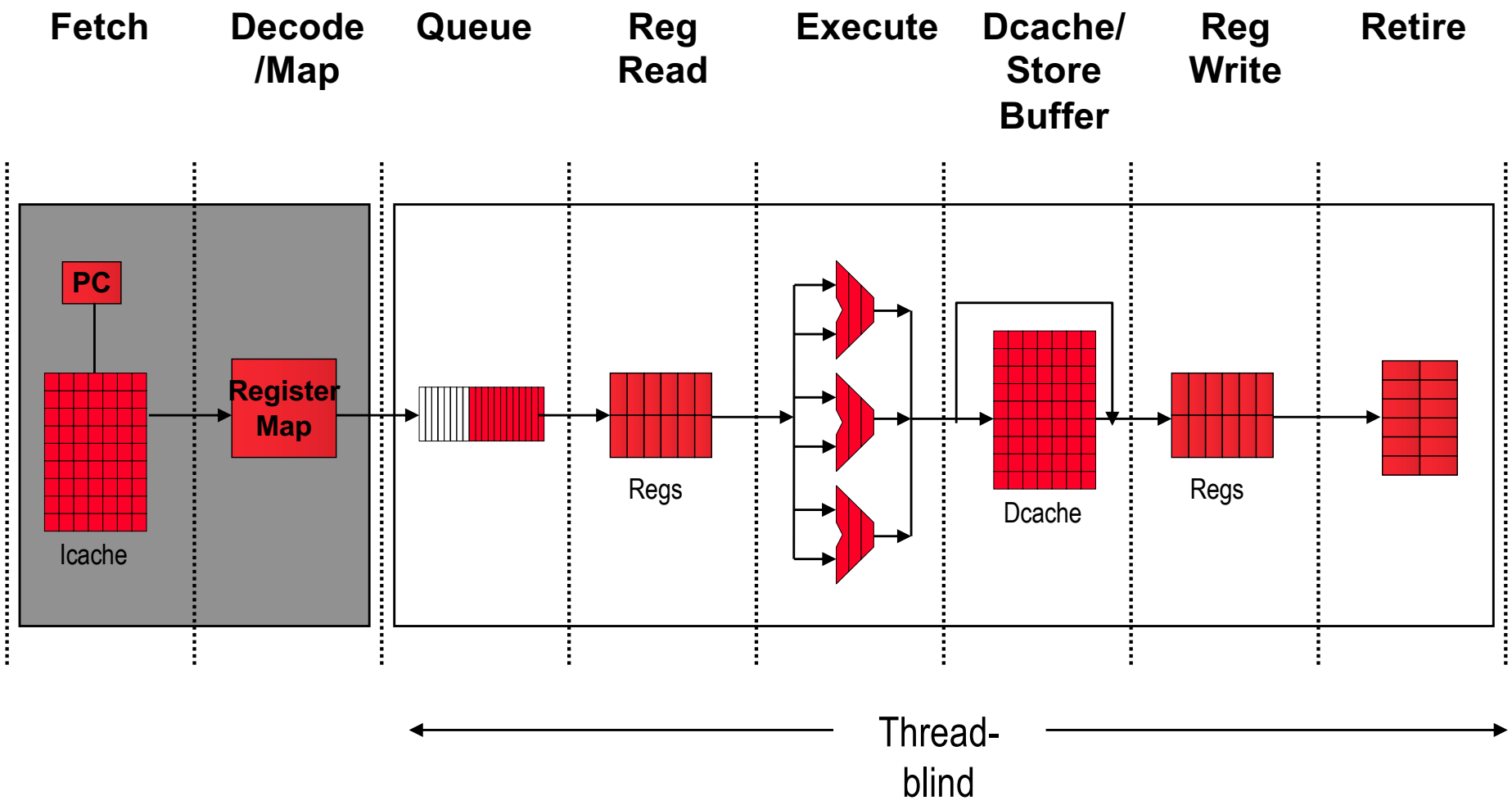
- Why is there horizontal and vertical waste?
- How do you reduce each?

Simultaneous Multithreading

- **Reduces both horizontal and vertical waste**
- Required hardware
  - The ability to dispatch instructions from multiple threads simultaneously into different functional units (Require very high HW cost for analysis)

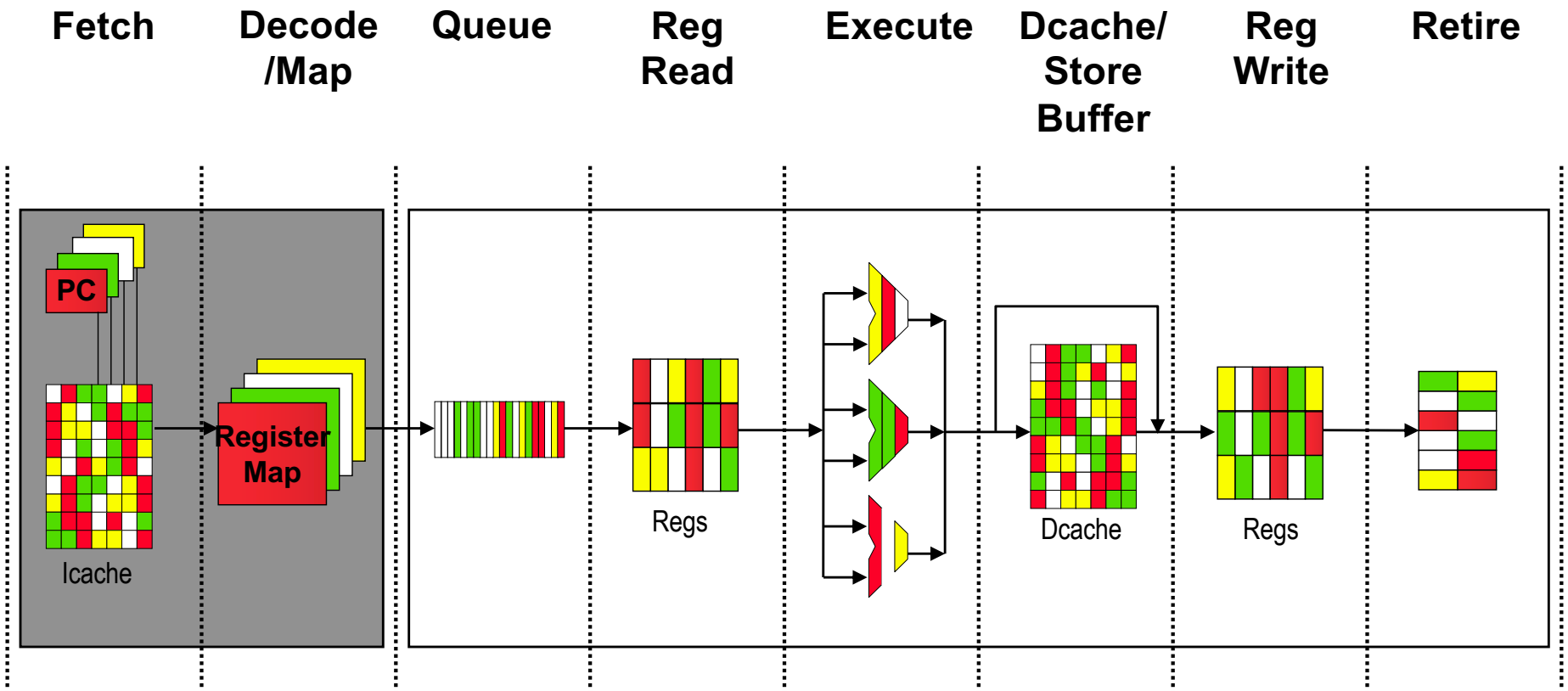
- Superscalar, OoO processors already have this machinery
  - Dynamic instruction scheduler searches the scheduling window to wake up and select ready instructions
  - As long as dependencies are correctly tracked (via renaming and memory disambiguation), scheduler can be thread-agnostic

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

  - Green colored ones can be shared or replicated for designers.

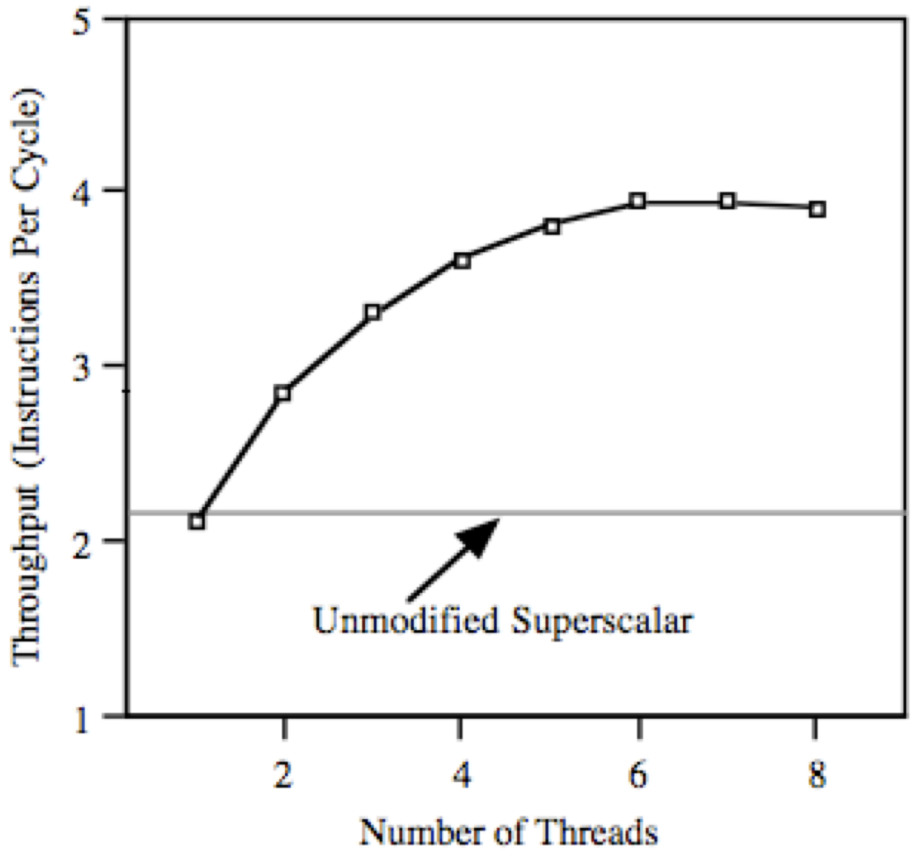
Changes to OoO+SS Pipeline for SMT

- 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? (The number of channels of SS processors are fixed)**



Case: # of channel of SS processor is 4

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
  - **Favor threads with higher real time requirements**
  - …

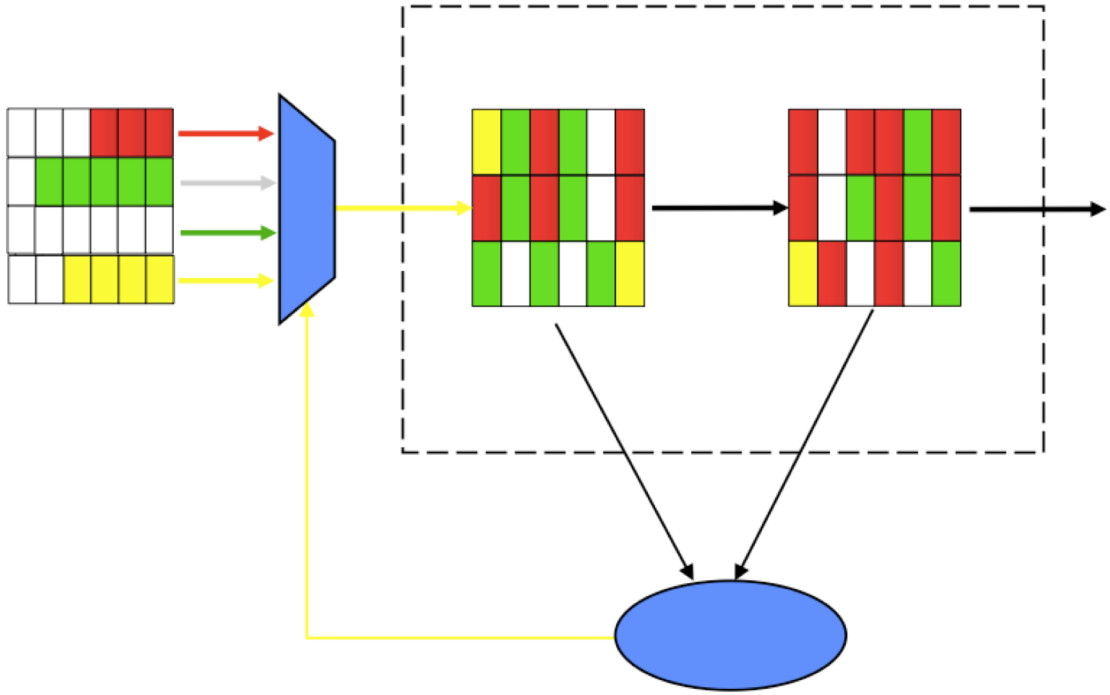
SMT Fetch Policies (I)

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

- Instructions from slow threads monopoly 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 instructions for a thread with the least instructions in the earlier pipeline stages (before execution)
- Refer to SMT_PAPER.



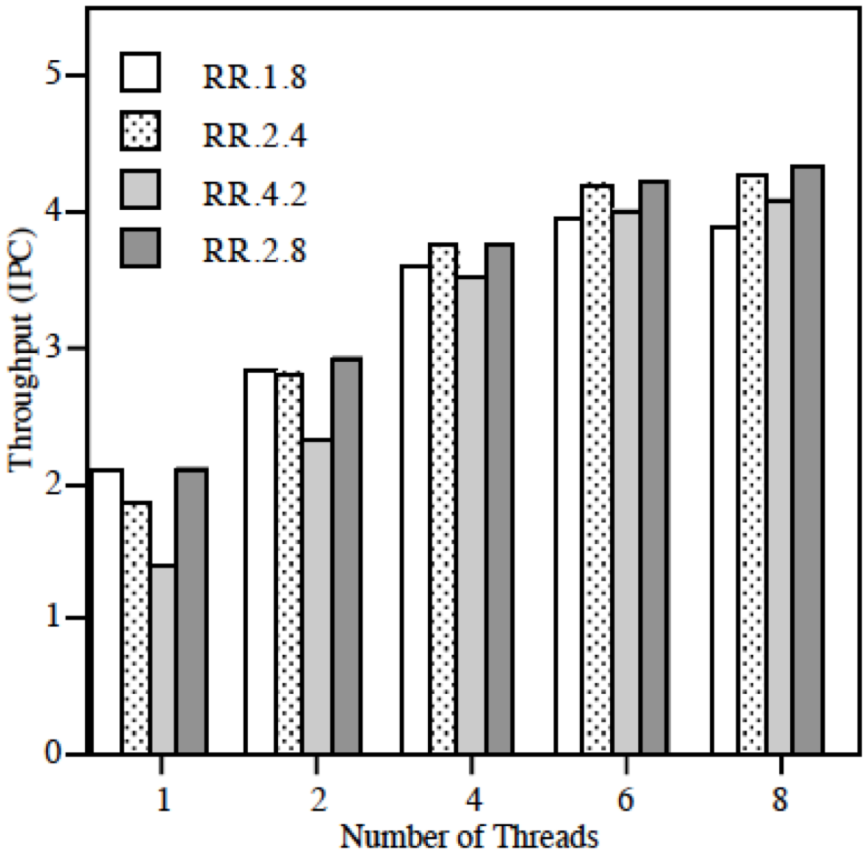
- **Why does this improve throughput?**

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

Some Results on Fetch Policy



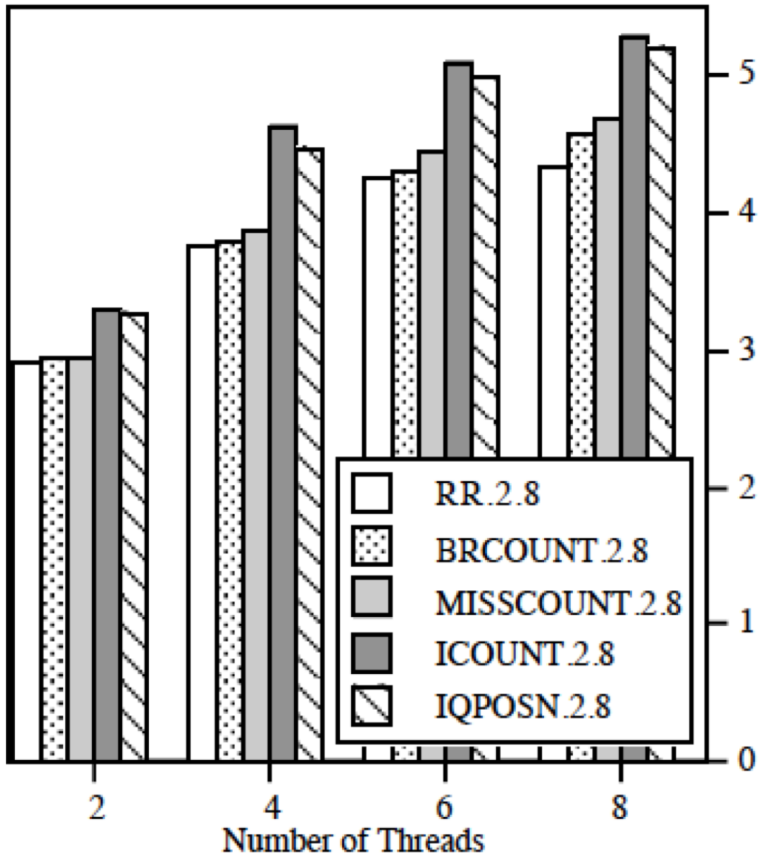


Figure 4: **Instruction throughput for the different instruction cache interfaces with round-robin instruction scheduling.**

- **Please compare the throughputs between RR (round robin) and ICOUNT methods.**

Example: Intel Pentium 4 Hyperthreading