SFU

CMPT 450/750: Computer Architecture
Fall 2024

Multithreading

Alaa Alameldeen & Arrvindh Shriraman

Multithreading: Basics

- Thread
  - Instruction stream with state (registers and memory)
  - Register state is 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
- 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 (initial motivation)
    - Latency of memory operations, dependent instructions, branch resolution
    - By utilizing processing resources more efficiently
  - To improve system throughput
    - By exploiting thread-level parallelism
    - By improving superscalar/OoO processor utilization
  - To reduce context switch penalty

Hardware Multithreading

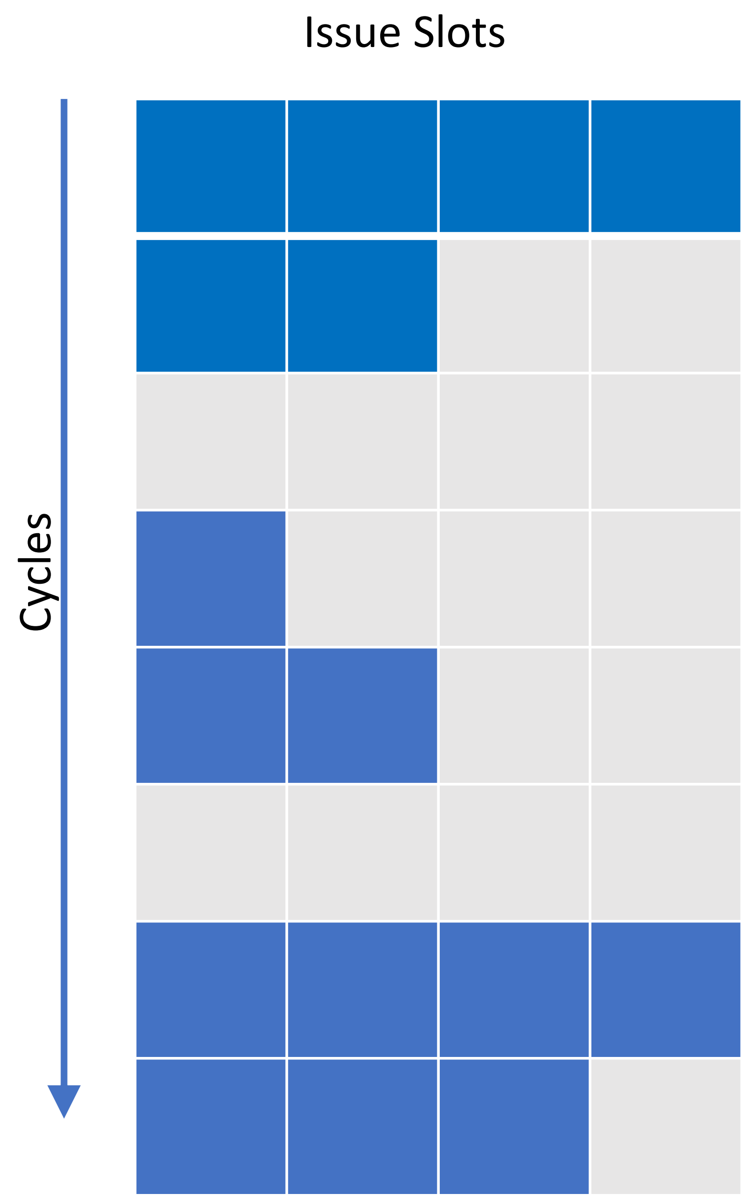
- **Benefit**
  - + Latency tolerance
  - + Better hardware utilization (when?)
  - + Reduced context switch penalty

- **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)

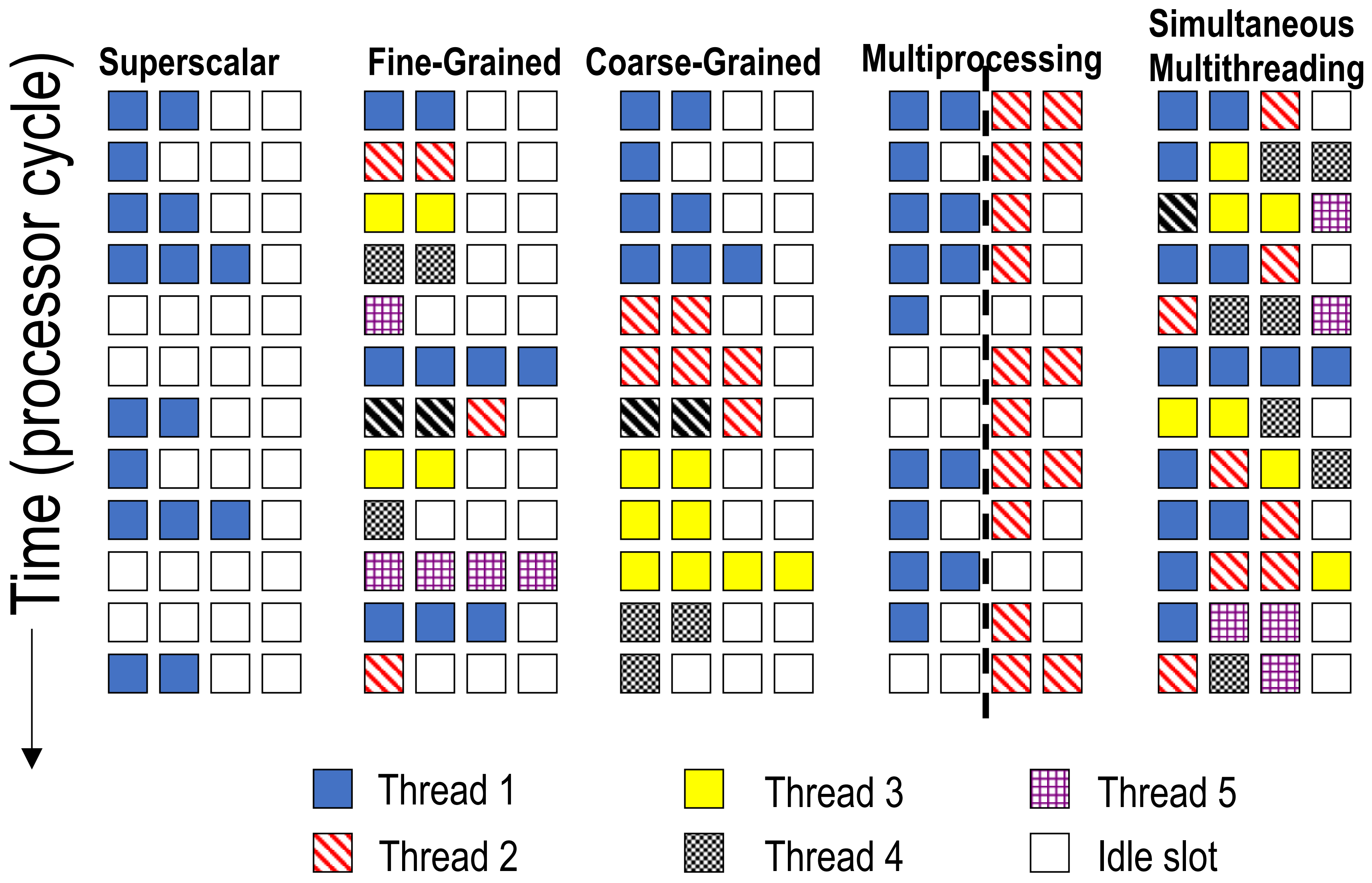
Simultaneous Multithreading

Why Multithreading?

- **ILP limitations of superscalar processors**
  - ➢Many control, data and functional dependences
- **Wide superscalar pipelines cannot use all issue slots**
  - ➢Vertical Waste: All issue slots in a cycle are not used
  - ➢Horizontal waste: Some issue slots in a cycle are not used
- **To increase throughput, we need to use thread-level parallelism (TLP)**



Multithreaded Categories



Simultaneous Multi-threading

One thread, 8 units

Cycle	M	M	FX	FX	FP	FP	BR	CC
1	🟨							🟨
2	🟨	🟨					🟨	
3				🟨	🟨			
4								
5								
6								
7	🟨			🟨		🟨		
8		🟨			🟨			
9				🟨				

Two threads, 8 units

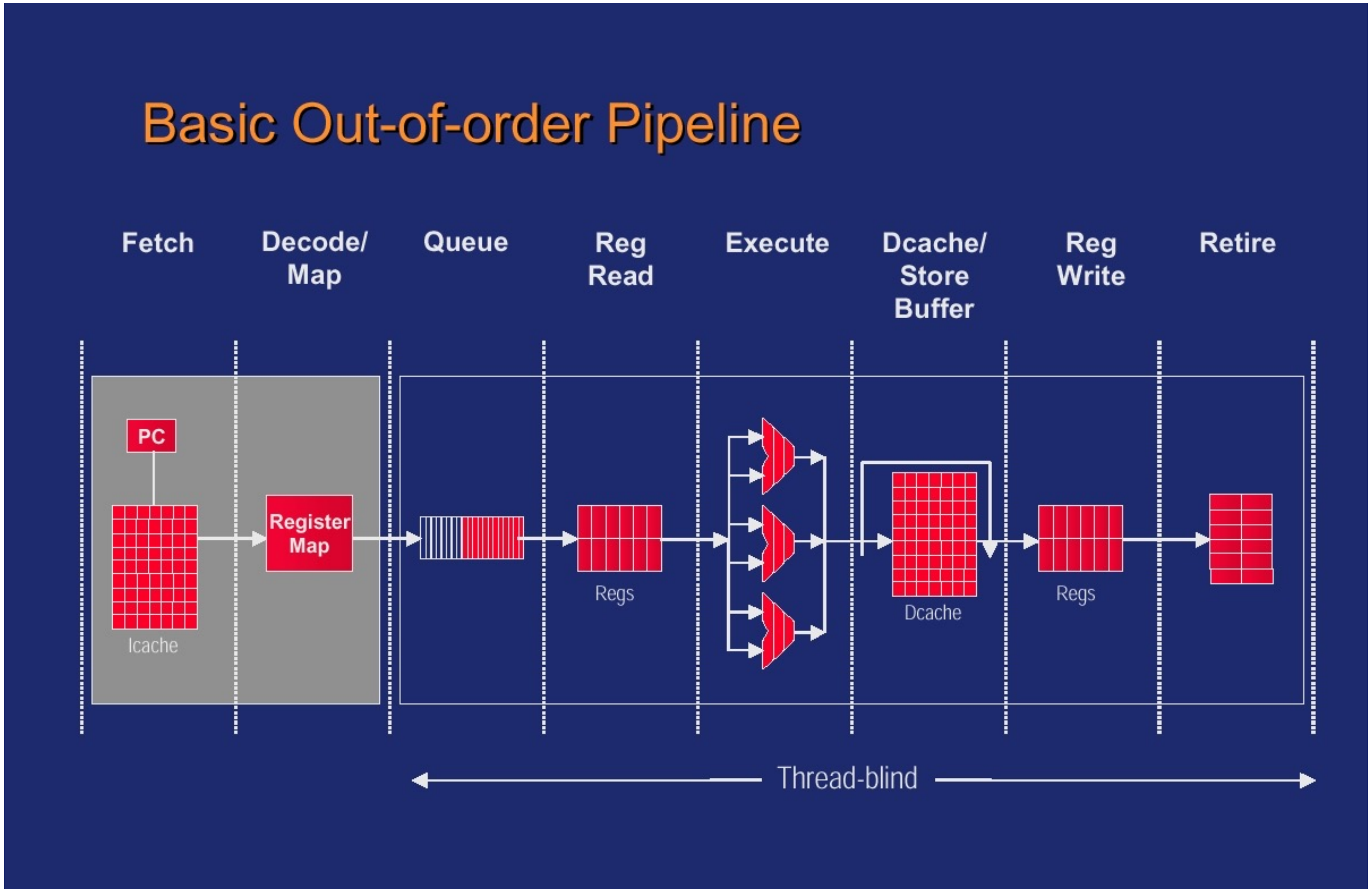
Cycle	M	M	FX	FX	FP	FP	BR	CC
1	🟨	🟦	🟦					🟨
2	🟨	🟨	🟦			🟦	🟨	
3	🟦			🟨	🟨			
4	🟦	🟦				🟦		
5		🟦						🟦
6								
7	🟨		🟦	🟨	🟦	🟨		
8		🟨		🟦	🟨	🟦		
9	🟦	🟦		🟨		🟦		

Multithreaded/Multicore Processors

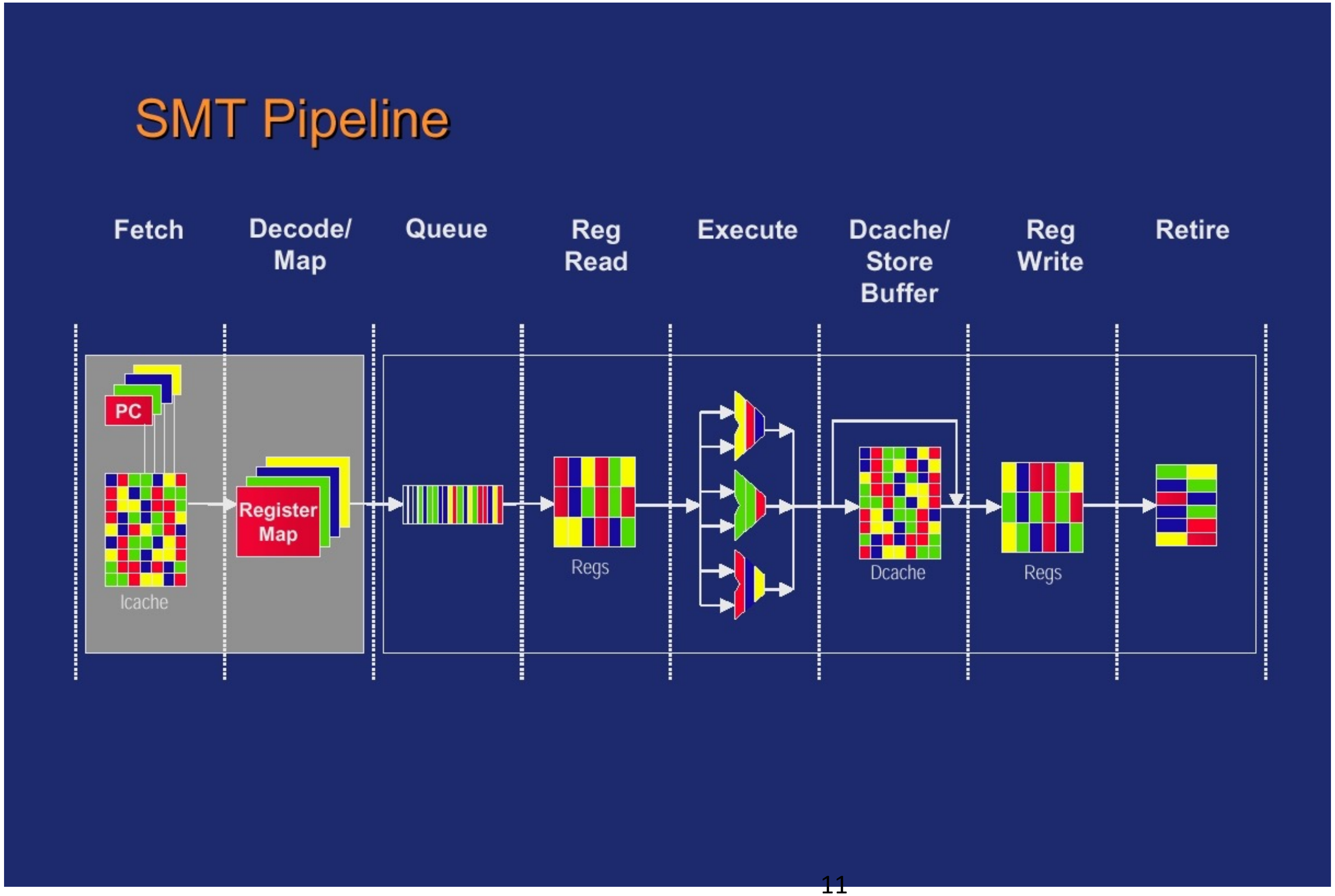
MT Approach	Resources shared between threads	Context Switch Mechanism
None	Everything	Explicit operating system context switch
Fine-grained	Everything but register file and control logic/state	Switch every cycle
Coarse-grained	Everything but I-fetch buffers, register file and con trol logic/state	Switch on pipeline stall
SMT	Everything but instruction fetch buffers, return address stack, architected register file, control logic/state, reorder buffer, store queue, etc.	All contexts concurrently active; no switching
CMT	Various core components (e.g. FPU), secondary cache, system interconnect	All contexts concurrently active; no switching
CMP	Secondary cache, system interconnect	All contexts concurrently active; no switching

- **Many approaches for executing multiple threads on a single die**
  - Mix-and-match: IBM Power8 CMP+SMT

SMT Microarchitecture [Emer,'01]



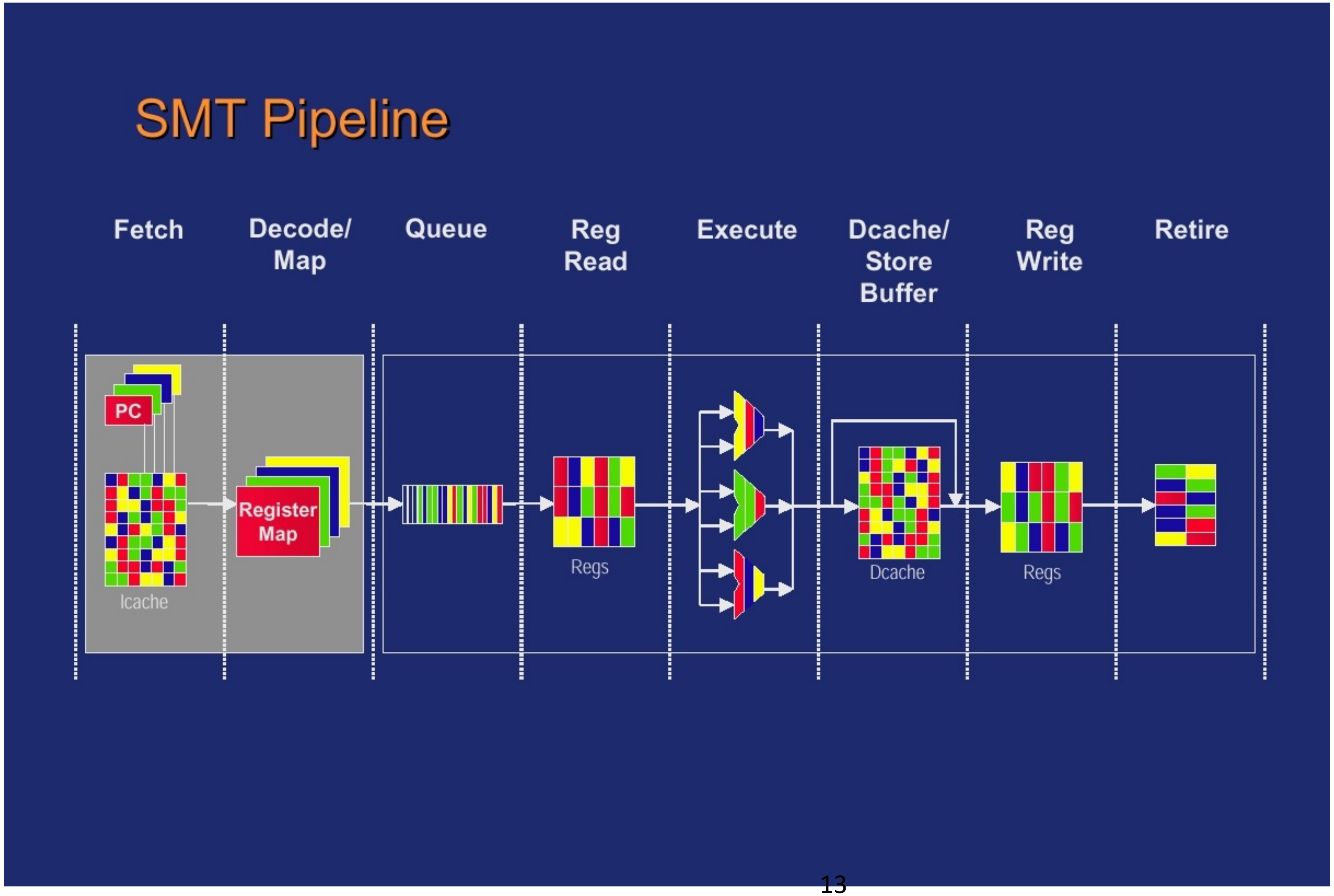
SMT Microarchitecture [Emer,'01]



Multithreaded Programs

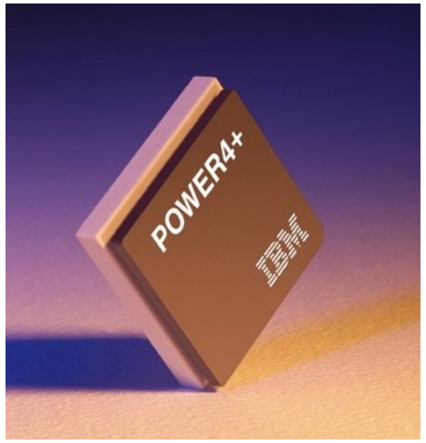
- **Thread vs. process**
  - ➢Threads in a process share virtual address space
  - ➢Processes have different virtual address spaces
- **Design Issues:**
  - ➢Each thread needs its own set of registers (register address space is not shared)
  - ➢Threads cause interference in instruction and data caches
  - ➢Programs need to be parallelizable into multiple threads
  - ➢Synchronization is necessary, may cause some threads to be idle (OS idle loop)

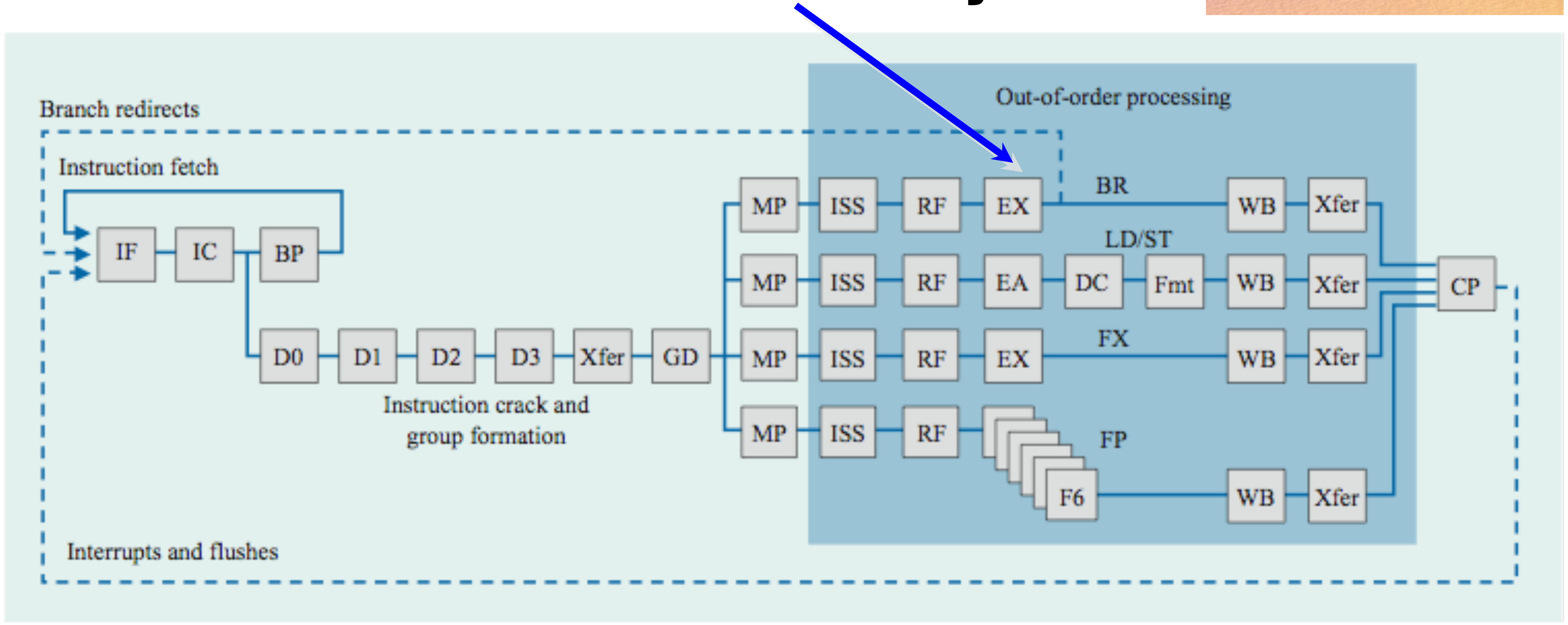
SMT Microarchitecture [Emer,'01]

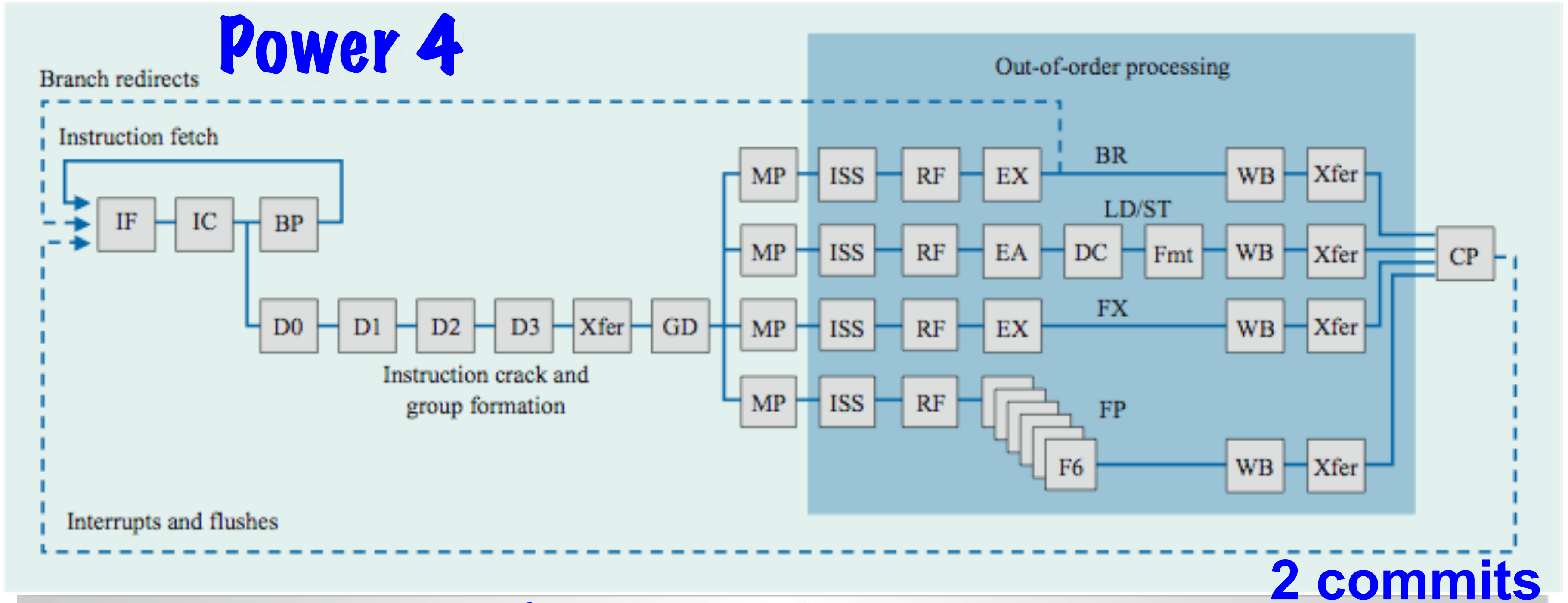


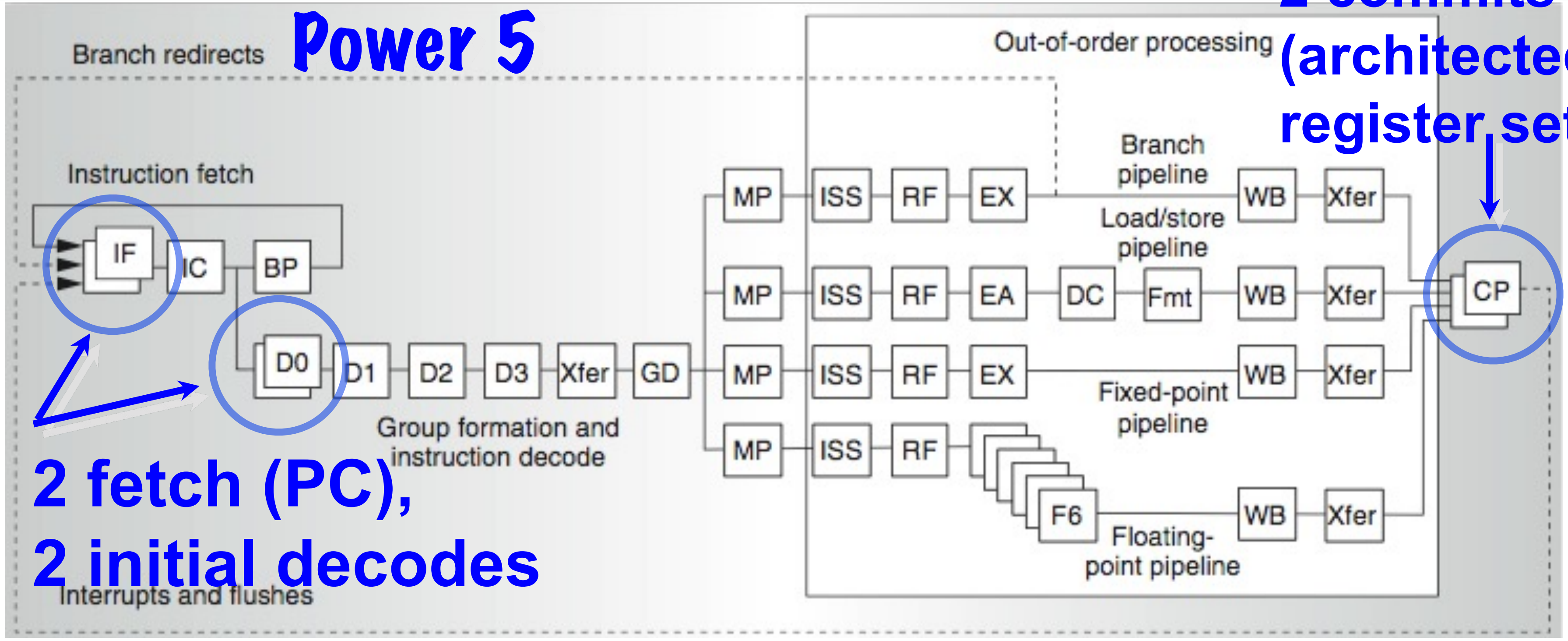
Power 4

Single-threaded predecessor to Power 5. 8 execution units in out-of-order engine, each may issue an instruction each cycle.





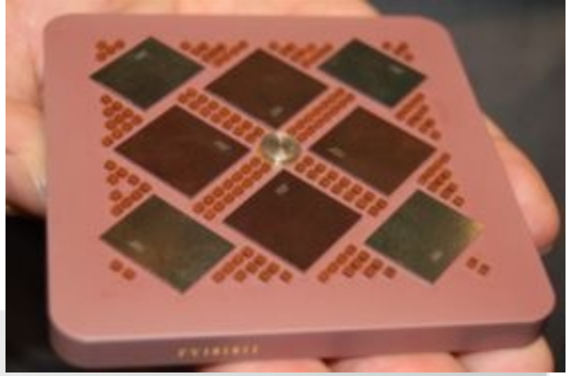


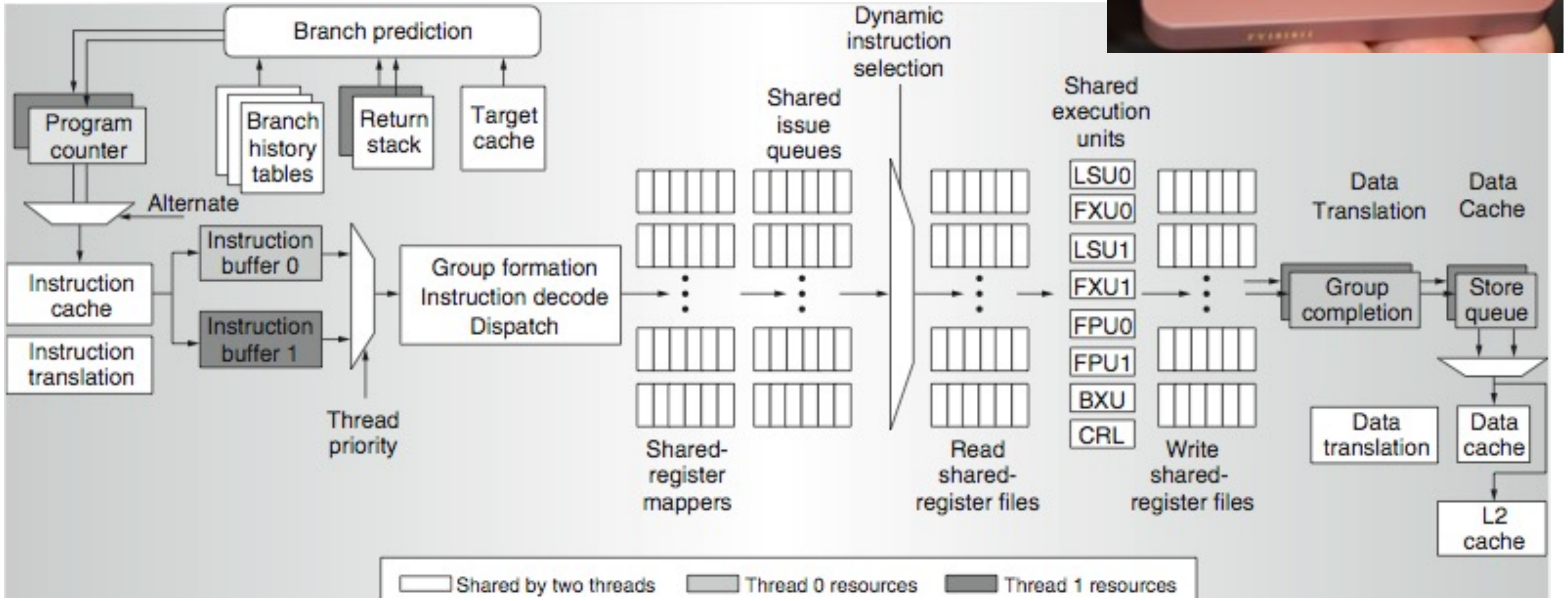


2 commits (architected register sets)

2 fetch (PC),
2 initial decodes

Power 5 data flow …



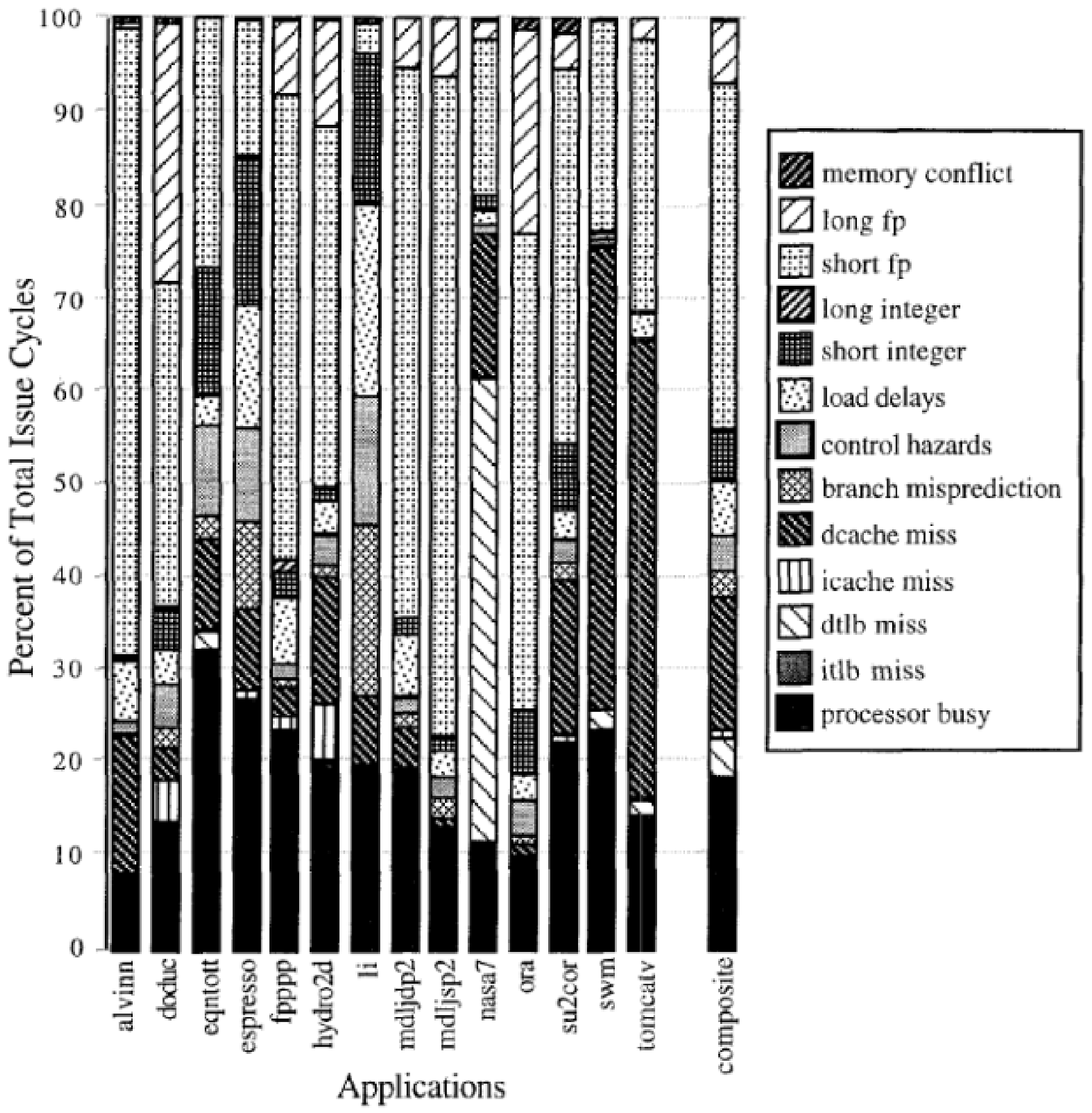


Why only 2 threads? With 4, one of the shared resources (physical registers, cache, memory bandwidth) would be prone to bottleneck

Superscalar Processors: Where Have Cycles Gone?

Tullsen et al., Figure 2

- Issue slots are utilized only 19% of the time
- Many causes for issue stall cycles (Figure)
- Need aggressive latency-hiding techniques

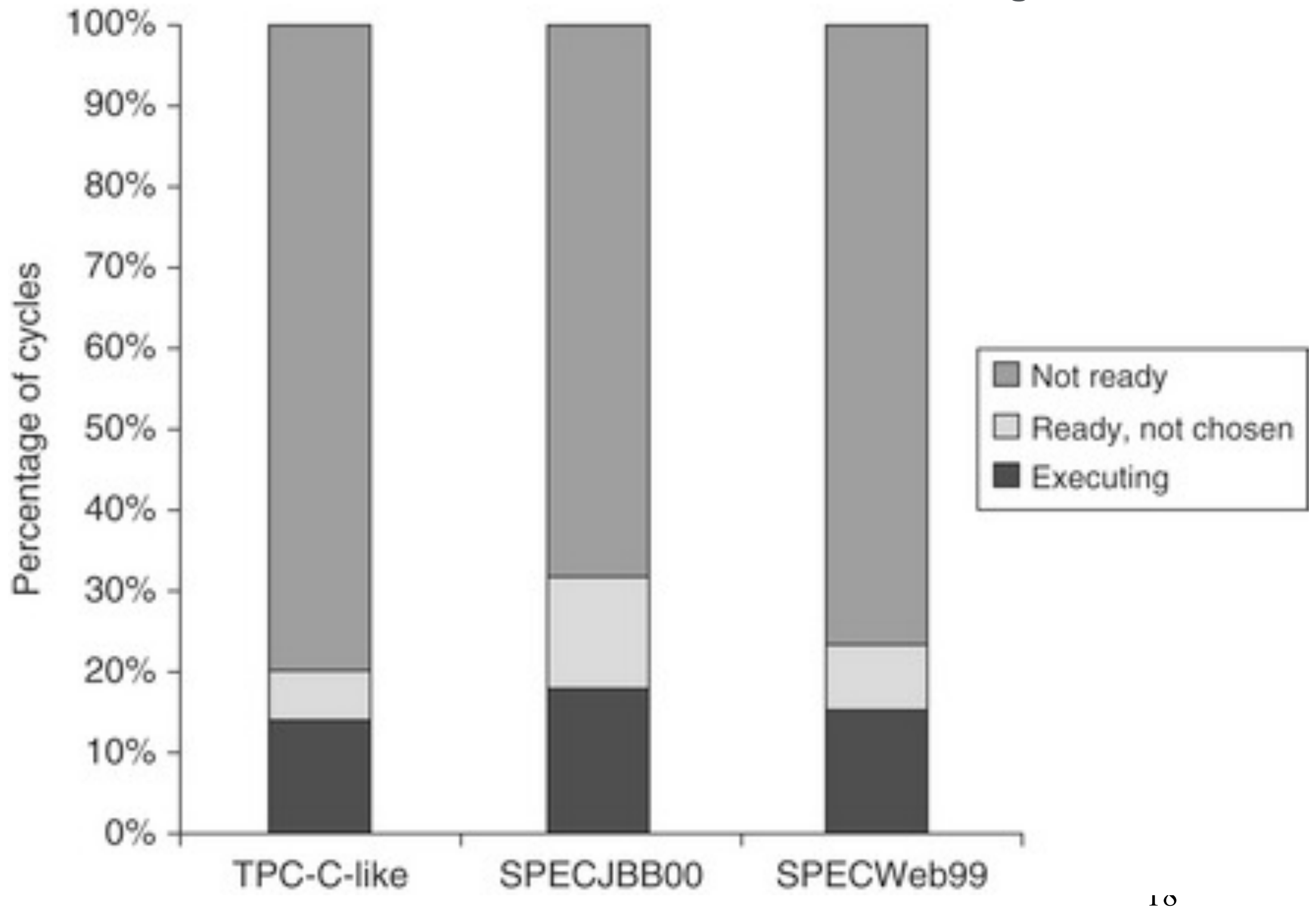


Commercial Multithreaded Workloads: Lost Cycles

- **"Ready, not chosen":** Threads could issue but other threads use all issue resources

- **"Not Ready":** Cannot Issue

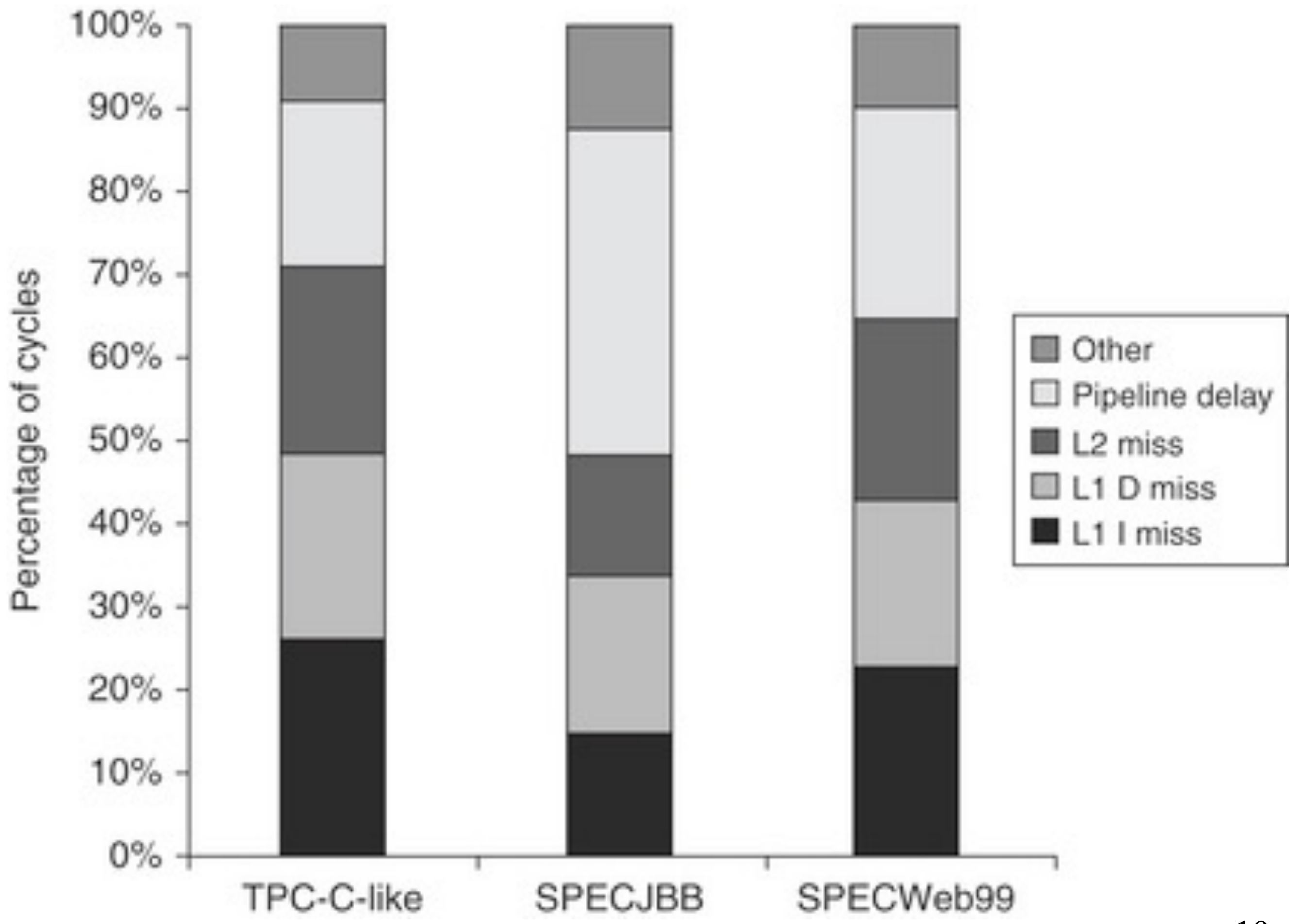
ARCH Figure 3.31



Commercial Workloads: Where Have Cycles Gone?

- Figure shows percentage of cycles lost for different reasons
- "Other" is mainly "store buffer full" in TPC-C, "atomic instructions" in jbb, both in SPECweb

ARCH Figure 3.32

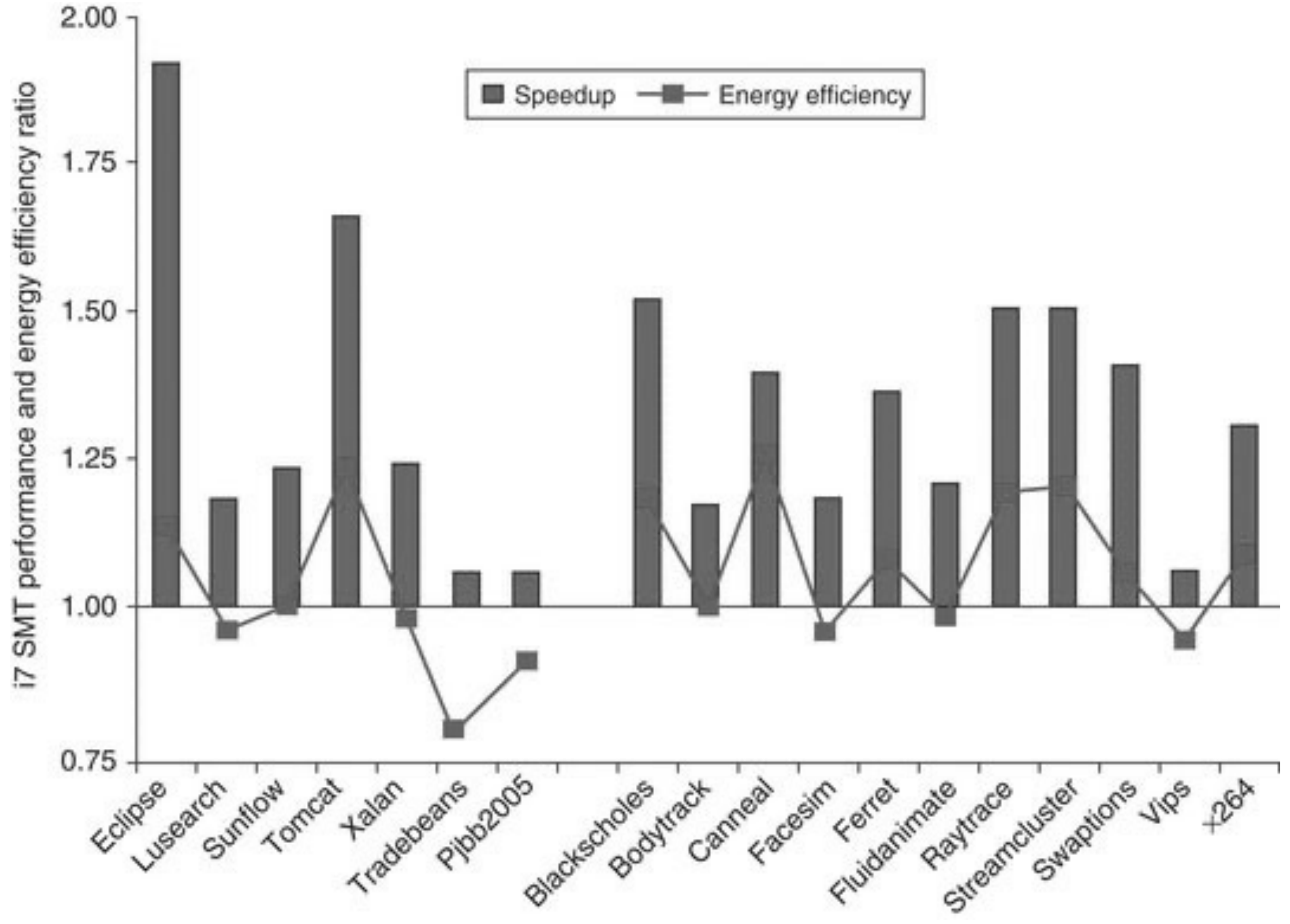


Simultaneous Multithreading Models

- **SM: Full Simultaneous Issue**
  - ➢Completely flexible model: All threads compete for each of the issue slots every cycle
  - ➢Disadvantage: Hardware complexity
- **SM: Single Issue**
  - ➢Each thread can issue at most one instruction every cycle
- **SM: Dual Issue and SM: Four Issue**
  - ➢Each thread can issue at most two (Dual Issue) or four (Four issue) instructions every cycle
- **SM: Limited Connection**
  - ➢Each thread is connected to exactly one of each type of functional unit
  - ➢Limits scheduling choices for functional units to reduce hardware complexity

SMT Performance: Java and PARSEC benchmarks

- **Figure shows speedup and energy efficiency (high is better) for Intel Core i7**
  - ➢Uses hyperthreading: similar to 2-way SMT
- **Speedup averages 1.28x for Java and 1.31x for PARSEC**
- **Energy Efficiency average 0.99 for Java and 1.07 for PARSEC**



ARCH Figure 3.35

SMT vs. Multiprocessors Discussion

- **SMT outperforms multiprocessing for all scenarios considered. Why?**
- **Advantages of SMT vs. Multicore**
  - ➢Area efficiency
  - ➢Reducing number of threads (i.e., threads becoming idle) allows other threads to progress faster in SMT processors, no change in MP
  - ➢Granularity and flexibility of design: Unit of design is a whole processor for MP, more flexible in SMT
- **Disadvantages?**

SMT Performance Side Effects

- **Lowest priority thread runs much slower than high priority thread**
- **Highest priority thread sees degraded performance as more threads are added**
  - ➢Sharing of resources (e.g., caches, TLB, BP tables)
- **Caches are more strained by an MT workload vs. ST workload due to a decrease in locality**

SMT Design Issues

- **Hardware complexity**
  - ➢Scheduling hardware requirements increase with threads
  - ➢Register file size increase
  - ➢May need more ports
- **Pipeline depth**
  - ➢Bigger structures (e.g., register file) require longer access time
  - ➢Leads to increasing the number of pipeline stages
- **Issue policy**
  - ➢Fixed thread priority
  - ➢Round-Robin priority
  - ➢ICOUNT
  - ➢Others?

Helper Threading

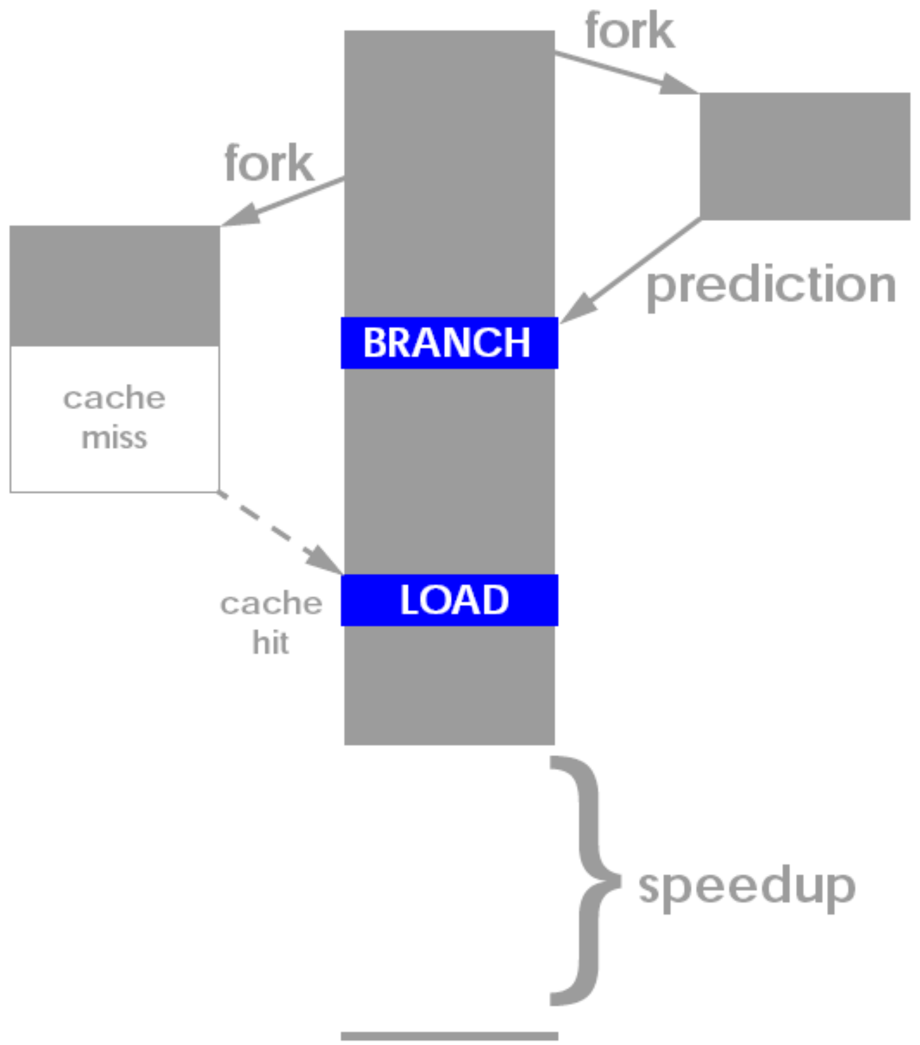
Helper Threading for Prefetching

- 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"

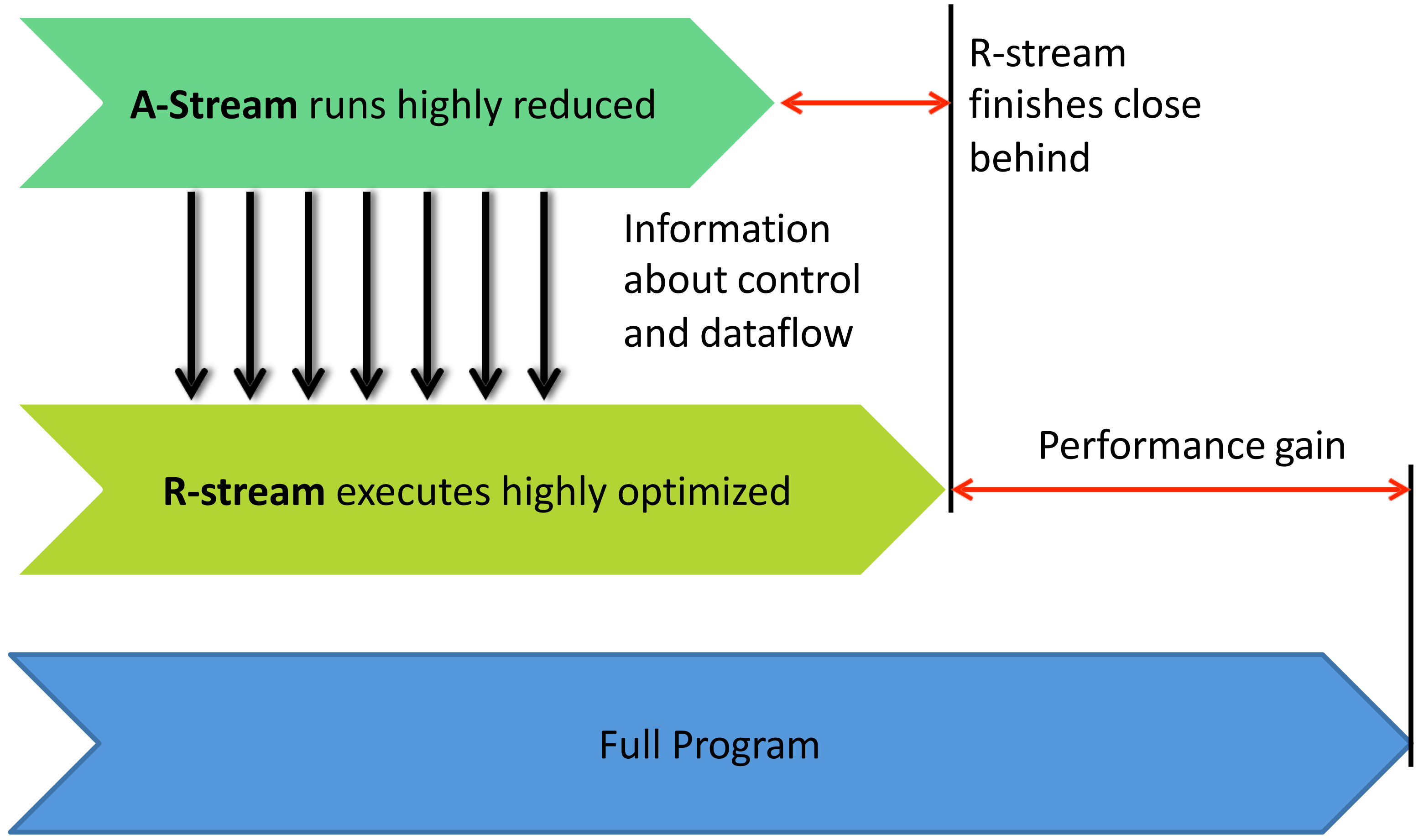
  - Speculative thread can be executed
  - On a separate processor/core
  - On a separate hardware thread context
  - On the same thread context in idle cycles (during cache misses)

Generalized Thread-Based Pre-Execution



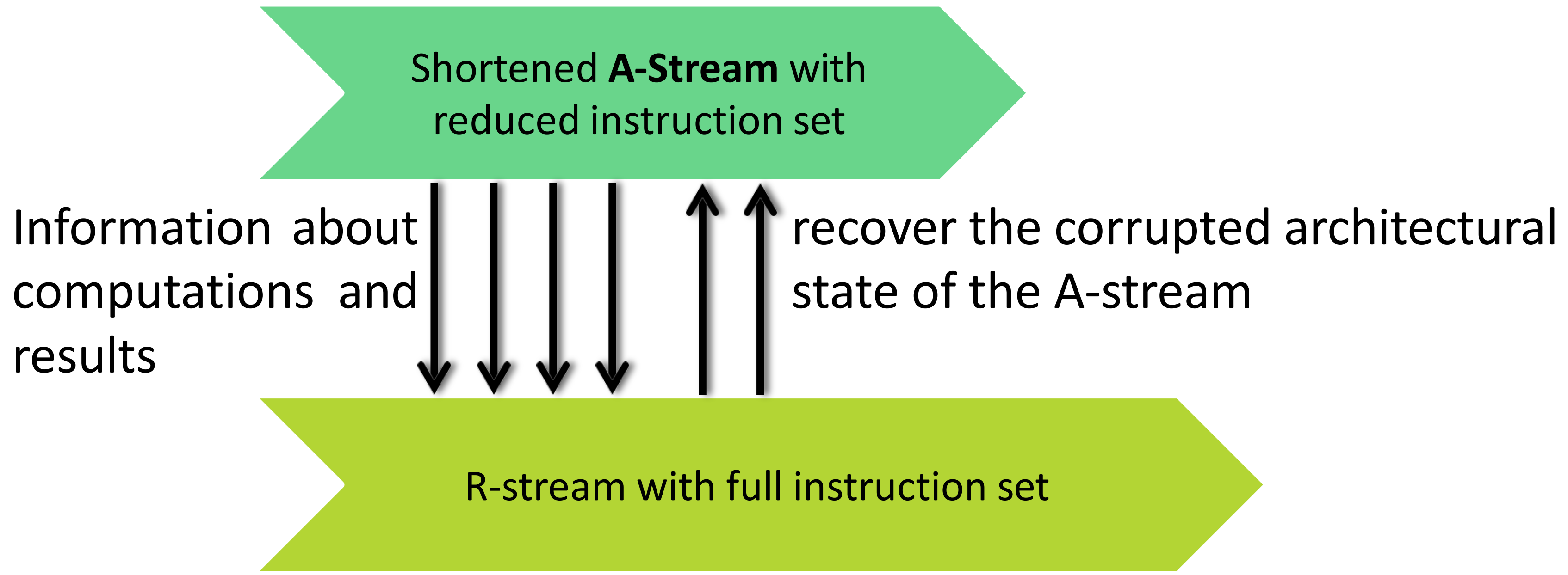
- Dubois and Song, "Assisted Execution," USC Tech Report 1998.
- Chappell et al., "Simultaneous Subordinate Microthreading (SSMT)," ISCA 1999.
- Zilles and Sohi, "Execution-based Prediction Using Speculative Slices", ISCA 2001.

Improve Performance


A-Stream runs highly reduced
R-stream finishes close behind
Information about control and dataflow
R-stream executes highly optimized
Performance gain
Full Program

SFU

Improve Fault Tolerance


Shortened **A-Stream** with reduced instruction set
Information about computations and results
recover the corrupted architectural state of the A-stream
R-stream with full instruction set

If R-Stream detects a mismatch between results we can recover

Thread-Based Pre-Execution Issues

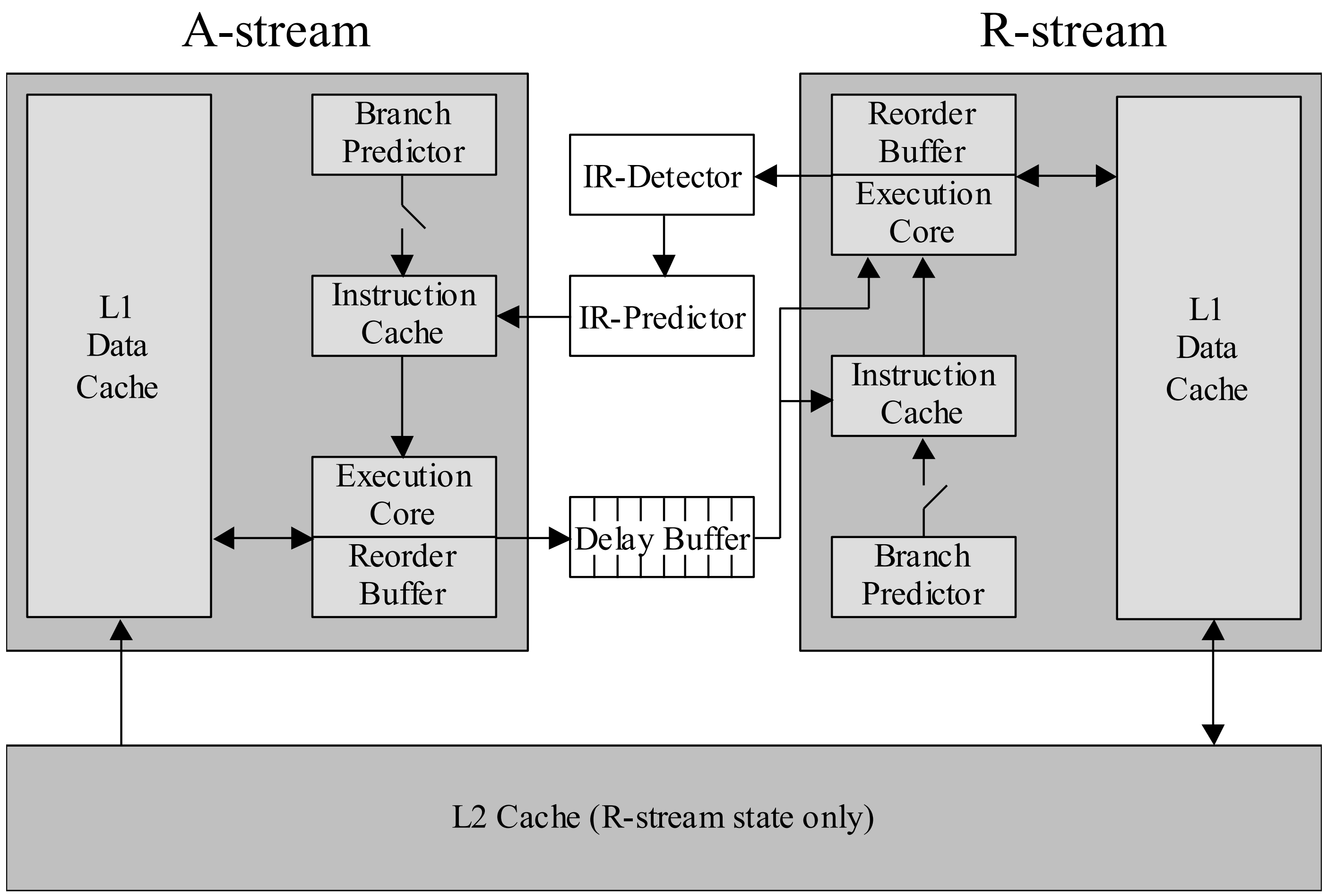
- **Where to execute the precomputation thread?**
  1. Core (least contention with main thread)
  2. SMT thread on the same core (more contention)
  3. Same core, same context, when the main thread is stalled
- **When to spawn the precomputation thread?**
  1. Insert spawn instructions well before the “problem” load
     - How far ahead?
       - Too early: prefetch might not be needed
       - Too late: prefetch might not be timely
  2. When the main thread is stalled
- **When to terminate the precomputation thread?**
  1. With pre-inserted CANCEL instructions
  2. Based on effectiveness/contention feedback

SFU

Slipstream Processors

- Goal: use multiple hardware contexts to speed up single thread execution (implicitly parallelize the program)
- Idea: Divide program execution into two threads:
  - Advanced thread executes a reduced instruction stream, speculatively
  - Redundant thread uses results, prefetches, predictions generated by advanced thread and ensures correctness
- Benefit: Execution time of the overall program reduces
- Core idea is similar to many thread-level speculation approaches, except with a reduced instruction stream
- Sundaramoorthy et al., "Slipstream Processors: Improving both Performance and Fault Tolerance," ASPLOS 2000.

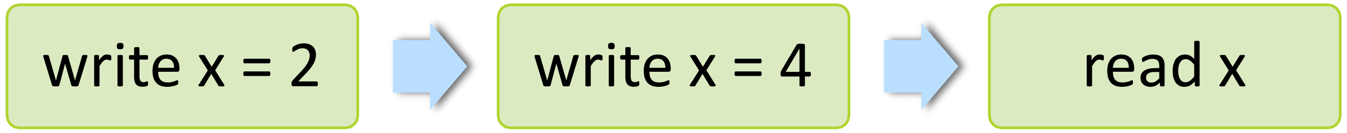
Slipstream Idea and Possible Hardware



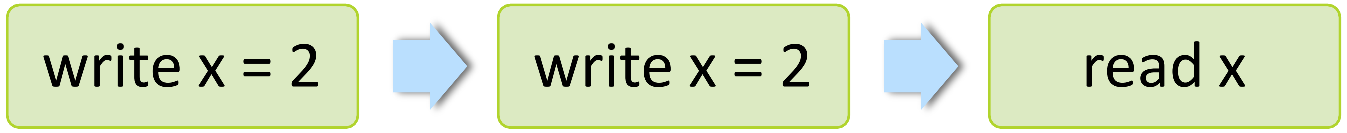
Removable Instructions

Distinguish three categories of ineffectual computation

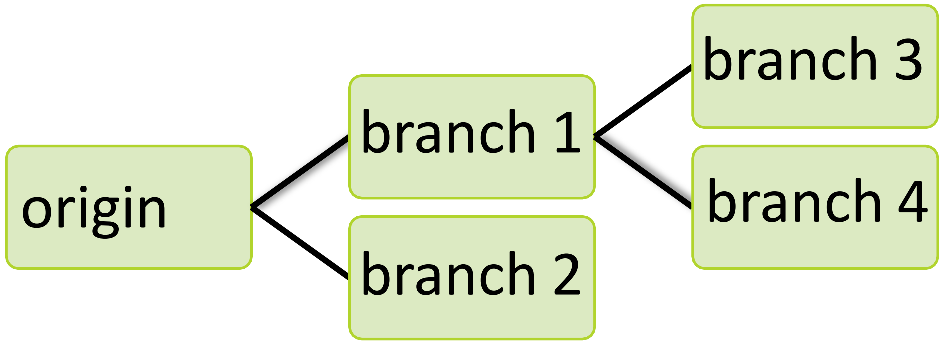
1. Unreferenced writes are values overwritten before use



2. Writes that do not modify the state of location



3. Dynamic branches whose outcomes are consistently predicted correctly.



Speculative Multithreading

Speculative Parallelization Concepts

- Idea: Execute threads unsafely in parallel
  - Threads can be from a sequential or parallel application

- Hardware or software monitors for data dependence violations

- If data dependence ordering is violated
  - Offending thread is squashed and restarted
- If data dependences are not violated
  - Thread commits
  - If threads are from a sequential order, the sequential order needs to be preserved → threads commit one by one and in order

Example Tasks

```
for (indx – 0; indx < BUFSIZE; indx++) {
    /* get the symbol for which to search */
    symbol = SYMVAL(buffer[indx]);

    /* do a linear search fo rthe symbol in the list */
    for (list – listhd; list; list – LNEXT(list) {
        /* if symbol already present, process entry */
        if (symbol == LELE(list)) {
            process(list);
            break;
        }
    }

    /* if symbol not found, add it to the tail */
    if (! list) {
        addlist(symbol);
    }


}
```

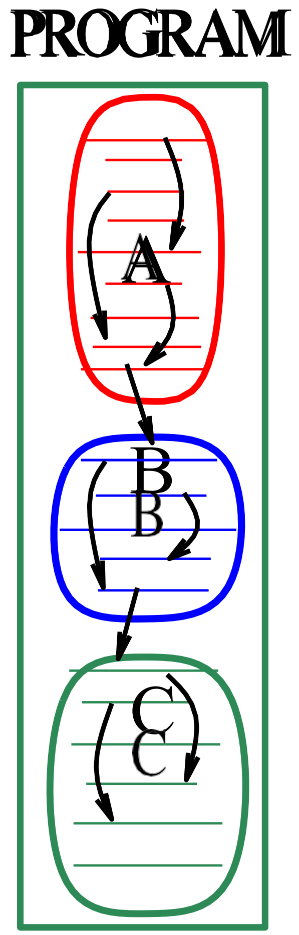
Parallel Programs vs. Thread Speculation

Attributes	Multicore	Multiscalar
Speculative task initiation	No/Difficult	Yes
Multiple flows of control	Yes	Yes
Task determination	Static	Static (possibly dynamic)
Software guarantee of inter-task control independence	Required	Not required
Software knowledge of inter-task data dependences	Required	Not required
Inter-task sync.	Explicit	Implicit/Explicit
Inter-task communication	Through memory Through messages	Through registers and memory
Register space	Distinct for PEs	Common for PEs
Memory space	Common Distinct	Common for PEs

Big Idea

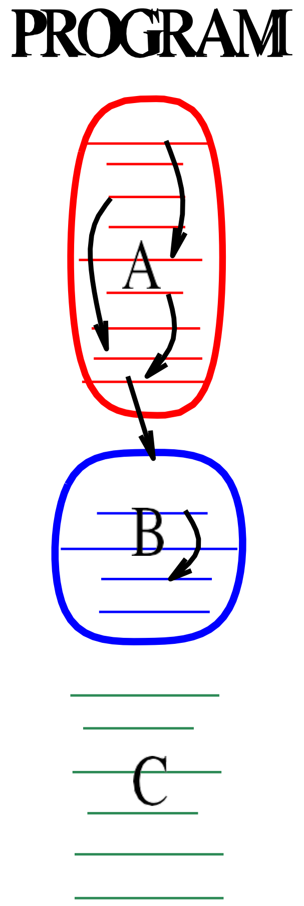
- Start with a static representation of a program
- Sequence through the program to generate the dynamic stream of operations
  - Use single PC to walk through static representation
- Execute operations in dynamic stream as quickly as is possible

Speed up this entire process

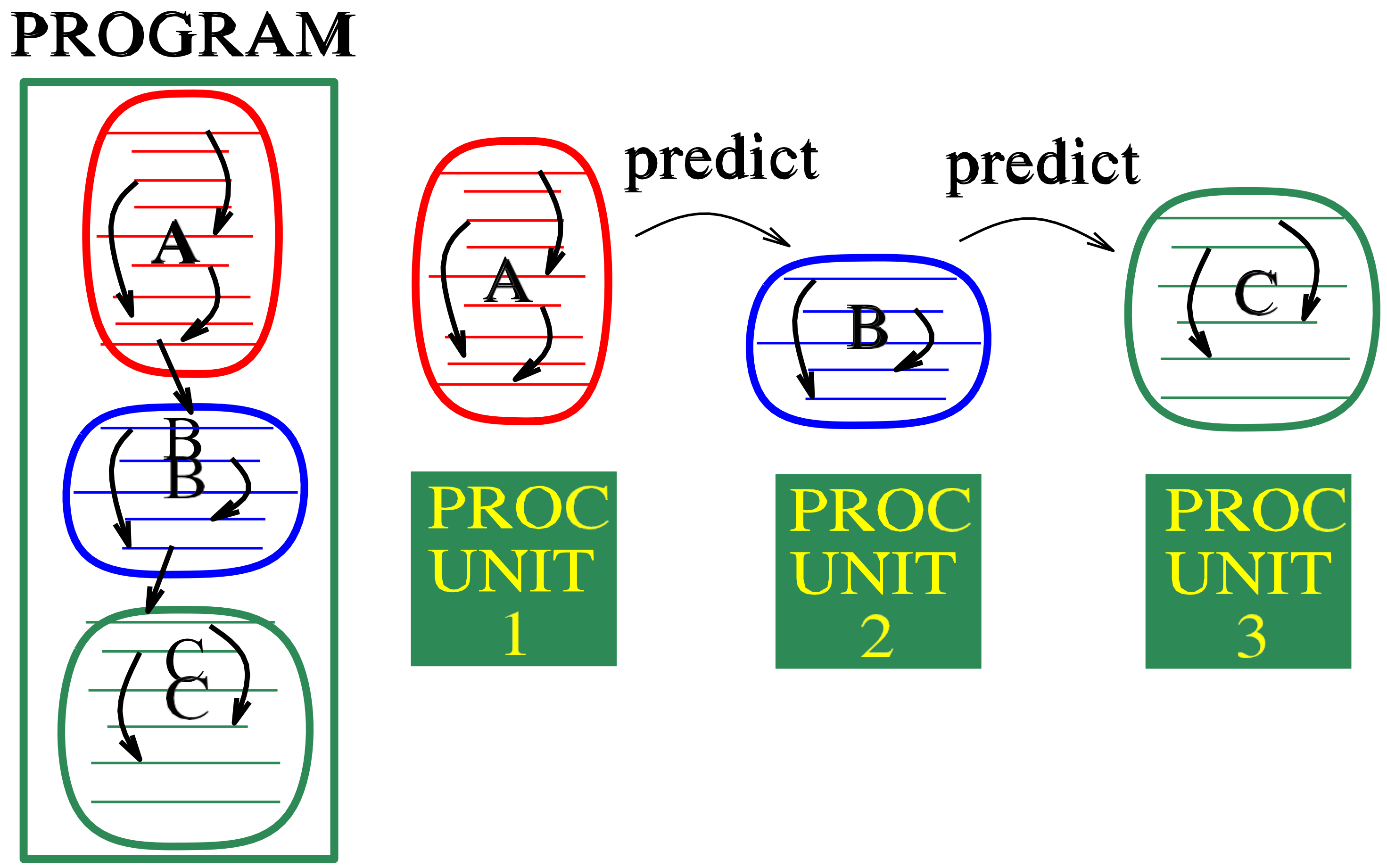


What is a task?

- A portion of the static representation resulting in a contiguous portion of the dynamic instruction stream
  - part of a basic block
  - basic block
  - multiple basic blocks
  - loop iteration
  - entire loop
  - procedure call, etc

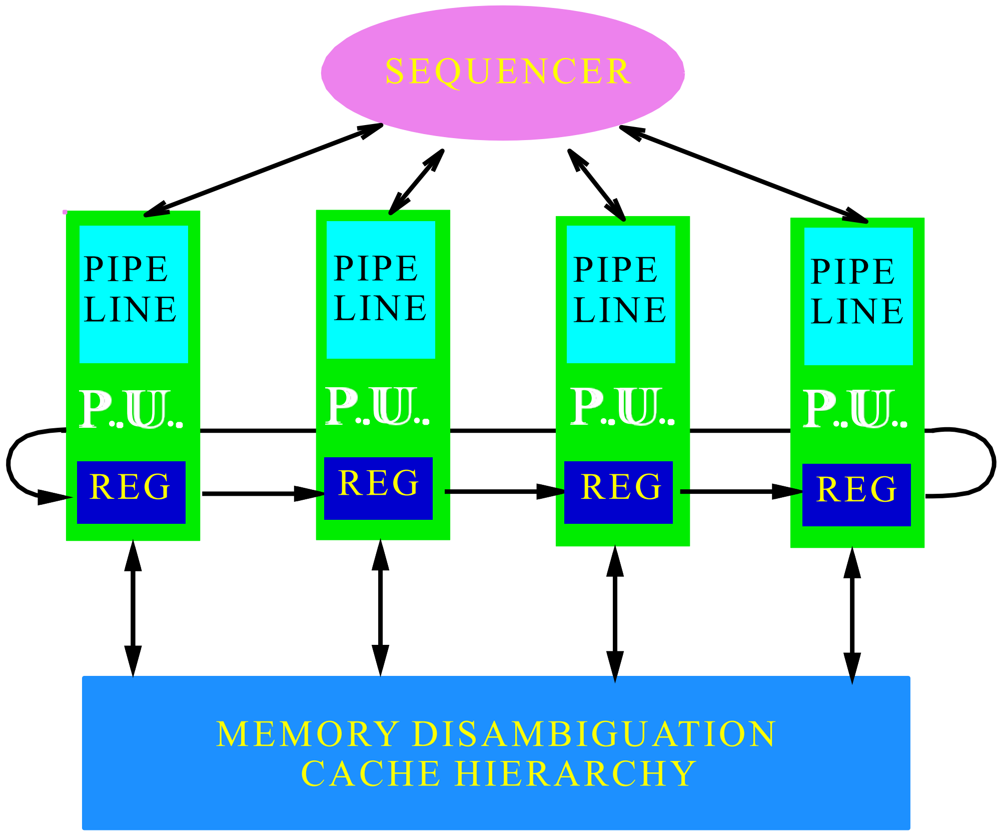


Big Idea


PROGRAM
A
B
C
A
predict
B
predict
C
PROC UNIT 1
PROC UNIT 2
PROC UNIT 3

Big Idea

- Processing Units (PUs) execute tasks
- Each PU has a processing element, instruction cache and a register file
- Sequencer assigns tasks to PUs
- After task is assigned, PU fetches instructions and executes task until completion
- May need to use multi-version caches to store multiple versions of same value simultaneously
- At a high level, Multiscalar could also use multiple threads in an SMT processor



- Processor consists of several processing cores (or units)
  - each core executes a task
  - each core is equivalent to a typical datapath
- Execution cores are connected in a logical order (queue)
  - hardware pointers to head and tail
  - share logical register and memory address spaces
- ***Active*** cores (ones between head and tail)
  - contain tasks in logical (sequential) order
  - together constitute a large dynamic window

Handling Inter-Task Dependences

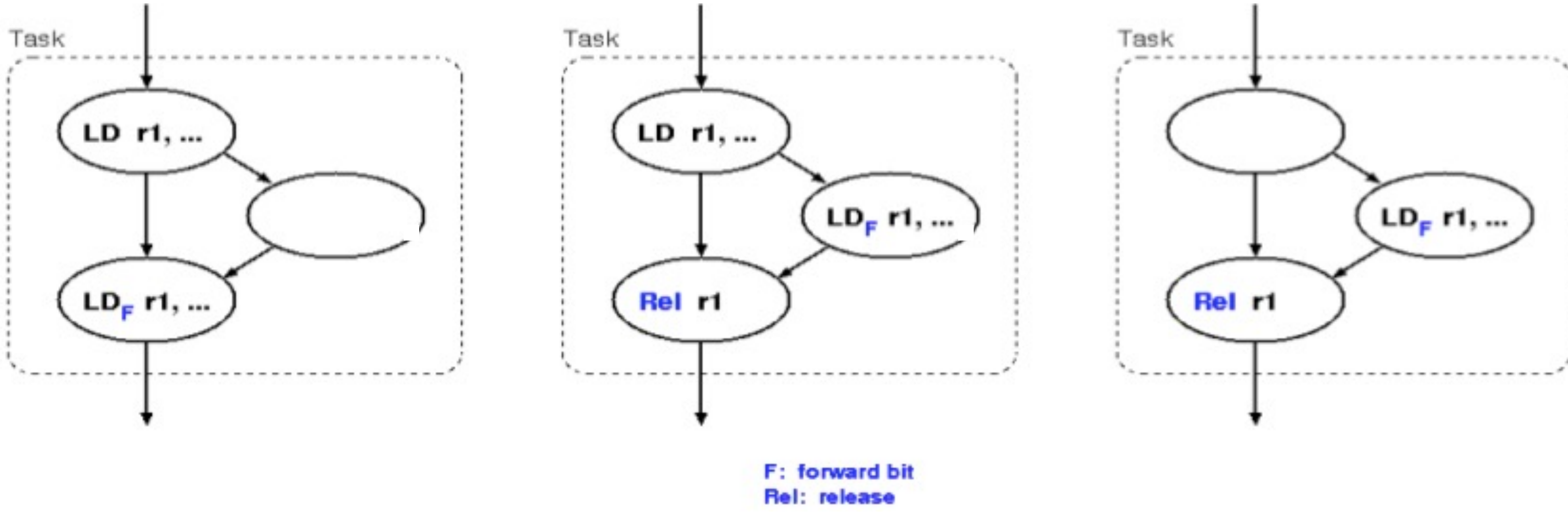
- **Control dependences**
  - Predict
  - Squash subsequent tasks on inter-task misprediction
    - Intra-task mispredictions do not need to cause flushing of later tasks
- **Data dependences**
  - Register file: mask bits and forwarding (stall until available)
  - Memory: address resolution buffer (speculative load, squash on violation)

Multiscalar Programs

- **Each task has to specify which registers it creates, how to forward register values, when the task ends, and which tasks follow it**
  - ➢ Information stored in *Task Descriptor*
- **Create Mask**
  - ➢ Register values that a task might produce
  - ➢ Conservative definition including all registers that could be produced (even if they are not produced in a particular instance of the task)
- **Forward Bits**
  - ➢ One bit associated with every instruction in task
  - ➢ Indicates whether destination register value is the last write by current task to that register, should be forwarded to subsequent tasks
- **Stop Bits**
  - ➢ Needs to check if conditions for stopping current task are satisfied at current instruction then task is exited
- **Release Instructions**
  - ➢ Indicates no further updates to register, can be forwarded to subsequent tasks

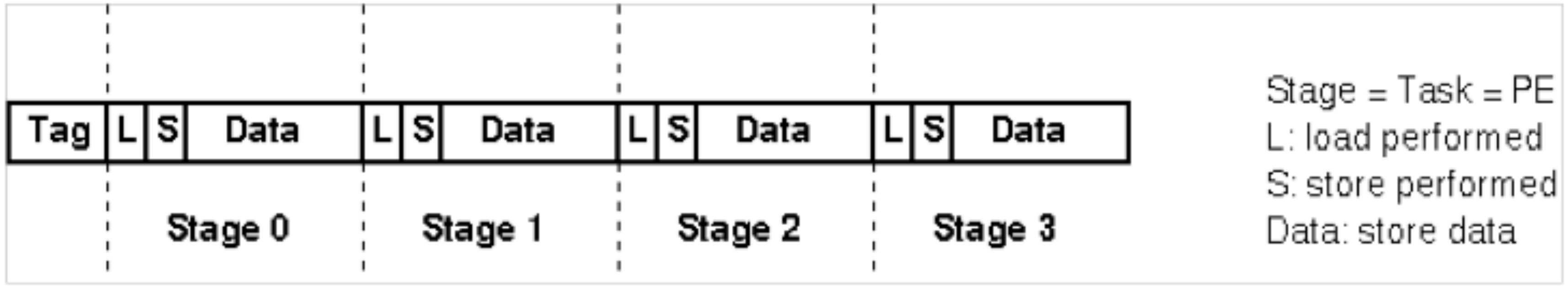
Forwarding Registers Between Tasks

- **Compiler must identify the last instance of write to a register within a task**
  - Opcodes that write a register have additional forward bit, indicating the instance should be forwarded
  - Stop bits - indicate end of task
  - Release instruction
    - tells PE value not neede



Address Resolution Buffer

- Multiscalar issues loads to ARB/D-cache as soon as address is computed
- ARB is organized like a cache, maintaining state for all outstanding load/store addresses
- Franklin and Sohi, "ARB: A hardware mechanism for dynamic reordering of memory references," IEEE TC 1996.
- An ARB entry:



Address Resolution Buffer

- **Loads**
  - ARB miss: data comes from D-cache (no prior stores yet)
  - ARB hit: get most recent data to the load, which may be from D-cache, or nearest prior task with S=1

- **Stores**
  - ARB buffers speculative stores
  - If store from an older task finds a load from a younger task to the same address → misspeculation detected
  - When a task commits, *commit all of the task's stores into the D-cache*

SpMT/TLS Implementation Cost

- **When speculative tasks violate sequential order, they need to be squashed**
  - Consumes power without gaining performance
- **Need to support multi-version caches for store values written by speculative tasks**
  - Values can only be written to memory from non-speculative tasks
  - Adds complexity to cache design
- **Dependence checking across tasks may require complex hardware**
- **Requires compiler support: Program analysis, creating task descriptors, adding code for dependence checking**
- **Adds more instructions or prefix bits to existing instructions**
  - Increases program size
  - Code may not be portable across processor implementations
- **Some optimizations have been proposed to reduce energy overhead and hardware cost**

VLIW Architectures

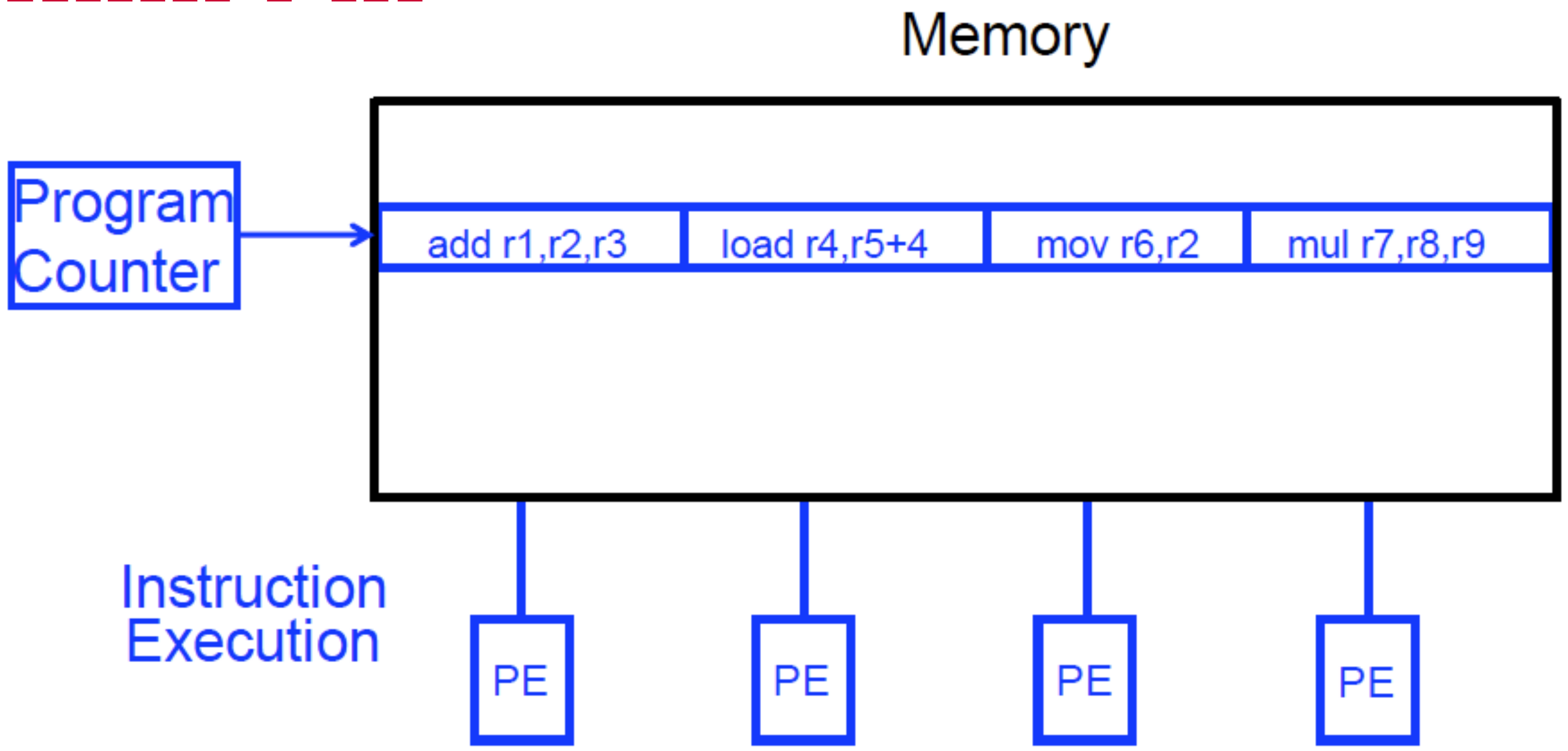
VLIW (Very Long Instruction Word)

- **A very long instruction word consists of multiple independent instructions packed together by the compiler**
  - Packed instructions can be logically unrelated (contrast with SIMD)

- **Idea: Compiler finds independent instructions and statically schedules (i.e. packs/bundles) them into a single VLIW instruction**

- **Traditional Characteristics**
  - Multiple functional units
  - Each instruction in a bundle executed in lock step
  - Instructions in a bundle statically aligned to be directly fed into the functional units

VLIW Concept



- Fisher, "Very Long Instruction Word architectures and the ELI-512," ISCA 1983.
  - ELI: Enormously longword instructions (512 bits)

VLIW Philosophy

- **Philosophy similar to RISC (simple instructions and hardware)**
  - ➢Except multiple instructions in parallel

- **RISC (John Cocke, 1970s, IBM 801 minicomputer)**
  - ➢Compiler does the hard work to translate high-level language code to simple instructions (John Cocke: control signals)
    - ❑And, to reorder simple instructions for high performance
  - ➢Hardware does little translation/decoding → very simple

- **VLIW (Fisher, ISCA 1983)**
  - ➢Compiler does the hard work to find instruction level parallelism
  - ➢Hardware stays as simple and streamlined as possible
    - ❑Executes each instruction in a bundle in lock step
    - ❑Simple → higher frequency, easier to design

Commercial VLIW Machines

- **Multiflow TRACE, Josh Fisher (7-wide, 28-wide)**
- **Cydrome Cydra 5, Bob Rau**
- **Transmeta Crusoe: x86 binary-translated into internal VLIW**
- **TI C6000, Trimedia, STMicro (DSP & embedded processors)**
  - ➢Most successful commercially

- **Intel IA-64**
  - ➢Not fully VLIW, but based on VLIW principles
  - ➢EPIC (Explicitly Parallel Instruction Computing)
  - ➢Instruction bundles can have dependent instructions
  - ➢A few bits in the instruction format specify explicitly which instructions in the bundle are dependent on which other ones

VLIW Tradeoffs

- **Advantages**
  - + No need for dynamic scheduling hardware → simple hardware
  - + No need for dependency checking within a VLIW instruction → simple hardware for multiple instruction issue + no renaming
  - + No need for instruction alignment/distribution after fetch to different functional units → simple hardware

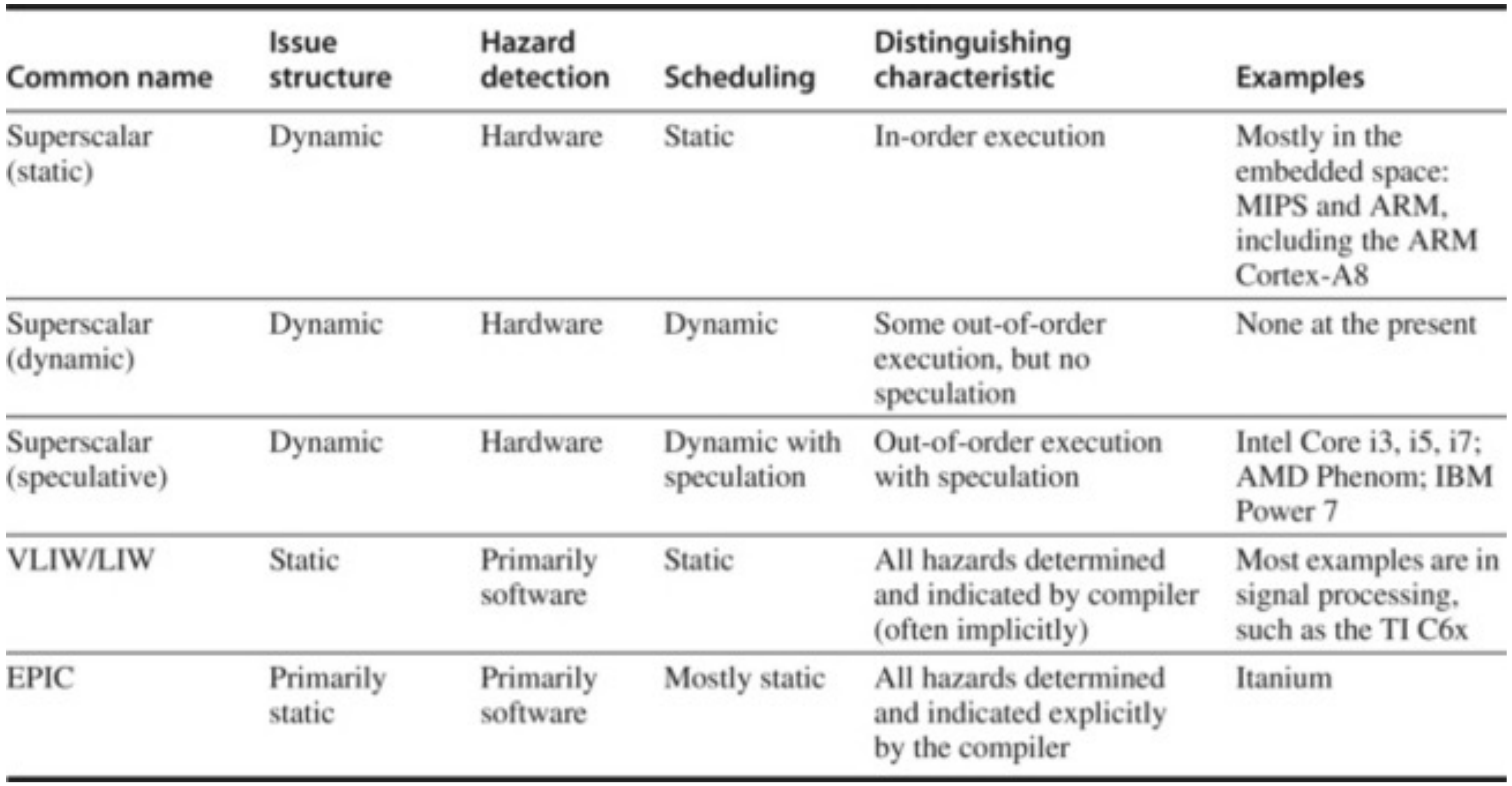
- **Disadvantages**
  - -- Compiler needs to find N independent operations
    - -- If it cannot, inserts NOPs in a VLIW instruction
    - -- Parallelism loss AND code size increase
  - -- Recompilation required when execution width (N), instruction latencies, functional units change (Unlike superscalar processing)
  - -- Lockstep execution causes independent operations to stall
    - -- No instruction can progress until the longest-latency instruction completes

VLIW Issues: Dynamic Execution

- **Compiler cannot anticipate dynamic events or account for variable execution latencies**

1. **Cache misses**
   - ➢Static scheduling assumes cache hits
   - ➢VLIW requires blocking caches so a cache miss blocks issue for future instruction words, degrading performance
2. **Memory disambiguation**
   - ➢Pointer references are assumed to be dependent, couldn't belong to same instruction word
   - ➢Adds false dependences between loads and stores
3. **Branch outcomes**
   - ➢Branch mispredictions lead to executing compensation code

Comparing Multiple-Issue Processor Designs



Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the ARM Cortex-A8
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

ARCH Figure 3.15

Static vs Dynamic Scheduling

- **Arguments against dynamic scheduling:**
  - requires complex structures to identify independent instructions (scoreboards, issue queue)
    - high power consumption
    - low clock speed
    - the compiler can "easily" compute instruction latencies and dependences – complex software is always preferred to complex hardware (?)

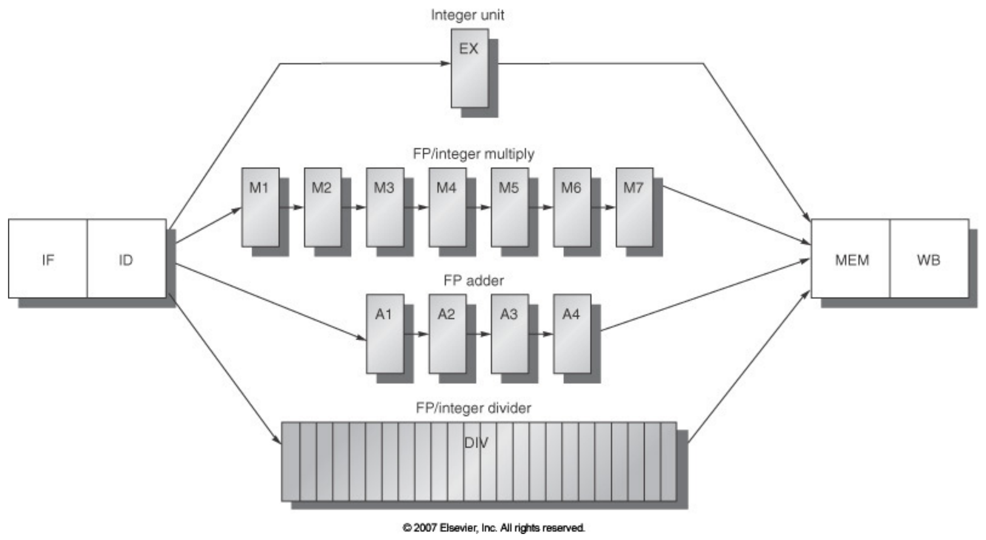
- **Instruction-level parallelism: overlap among instructions: pipelining or multiple instruction execution**
- **What determines the degree of ILP?**
  - dependences: property of the program
  - hazards: property of the pipeline

Loop Scheduling

- The compiler’s job is to minimize stalls
- Focus on loops: account for most cycles, relatively easy to analyze and optimize

Assumptions

- Load: 2-cycles (1 cycle stall for consumer)
- FP ALU: 4-cycles (3 cycle stall for consumer; 2 cycle stall if the consumer is a store)
- One branch delay slot
- Int ALU: 1-cycle (no stall for consumer, 1 cycle stall if the consumer is a branch)



LD -> any : 1 stall
FPALU -> any: 3 stalls
FPALU -> ST : 2 stalls
IntALU -> BR : 1 stall

SFU

Loop Example

```
for (i=1000; i>0; i--)
    x[i] = x[i] + s;
```

Source code

```
Loop:   L.D      F0, 0(R1)        ; F0 = array element
        ADD.D    F4, F0, F2       ; add scalar
        S.D      F4, 0(R1)        ; store result
        DADDUI   R1, R1,# -8      ; decrement address pointer
        BNE      R1, R2, Loop     ; branch if R1 != R2
        NOP
```

Assembly code

Loop Example

LD -> any : 1 stall
FPALU -> any: 3 stalls
FPALU -> ST : 2 stalls
IntALU -> BR : 1 stall

```
for (i=1000; i>0; i--)
    x[i] = x[i] + s;
```

Source code

```
Loop:   L.D      F0, 0(R1)       ; F0 = array element
        ADD.D    F4, F0, F2      ; add scalar
        S.D      F4, 0(R1)       ; store result
        DADDUI   R1, R1,# -8     ; decrement address pointer
        BNE      R1, R2, Loop    ; branch if R1 != R2
        NOP
```

Assembly code

```
Loop:   L.D      F0, 0(R1)       ; F0 = array element
        stall
        ADD.D    F4, F0, F2      ; add scalar
        stall
        stall
        S.D      F4, 0(R1)       ; store result
        DADDUI   R1, R1,# -8     ; decrement address pointer
        stall
        BNE      R1, R2, Loop    ; branch if R1 != R2
        stall
```

10-cycle schedule

Smart Schedule

LD -> any : 1 stall
FPALU -> any: 3 stalls
FPALU -> ST : 2 stalls
IntALU -> BR : 1 stall

```
Loop:   L.D       F0, 0(R1)
        stall
        ADD.D     F4, F0, F2
        stall
        stall
        S.D       F4, 0(R1)
        DADDUI    R1, R1,# -8
        stall
        BNE       R1, R2, Loop
        stall
```

```
Loop:   L.D       F0, 0(R1)
        DADDUI    R1, R1,# -8
        ADD.D     F4, F0, F2
        stall
        BNE       R1, R2, Loop
        S.D       F4, 8(R1)
```

- By re-ordering instructions, it takes 6 cycles per iteration instead of 10
- We were able to violate an anti-dependence easily because an immediate was involved
- Loop overhead (instrs that do book-keeping for the loop): 2
  Actual work (the ld, add.d, and s.d): 3 instrs
  Can we somehow get execution time to be 3 cycles per iteration?

Loop Unrolling

```
Loop:   L.D      F0, 0(R1)
        ADD.D    F4, F0, F2
        S.D      F4, 0(R1)
        L.D      F6, -8(R1)
        ADD.D    F8, F6, F2
        S.D      F8, -8(R1)
        L.D      F10,-16(R1)
        ADD.D    F12, F10, F2
        S.D      F12, -16(R1)
        L.D      F14, -24(R1)
        ADD.D    F16, F14, F2
        S.D      F16, -24(R1)
        DADDUI   R1, R1, #-32
        BNE      R1,R2, Loop
```

- Loop overhead: 2 instrs; Work: 12 instrs
- How long will the above schedule take to complete?

SFU

Scheduled and Unrolled Loop

```
Loop:   L.D      F0, 0(R1)
        L.D      F6, -8(R1)
        L.D      F10,-16(R1)
        L.D      F14, -24(R1)
        ADD.D    F4, F0, F2
        ADD.D    F8, F6, F2
        ADD.D    F12, F10, F2
        ADD.D    F16, F14, F2
        S.D      F4, 0(R1)
        S.D      F8, -8(R1)
        DADDUI   R1, R1, # -32
        S.D      F12, 16(R1)
        BNE      R1,R2, Loop
        S.D      F16, 8(R1)
```

LD -> any : 1 stall
FPALU -> any: 3 stalls
FPALU -> ST : 2 stalls
IntALU -> BR : 1 stall

- Execution time: 14 cycles or 3.5 cycles per original iteration

Automatic Loop Unrolling

- Determine the dependences across iterations: in the example, we knew that loads and stores in different iterations did not conflict and could be re-ordered

- Determine if unrolling will help – possible only if iterations are independent

- Determine address offsets for different loads/stores

- Dependency analysis to schedule code without introducing hazards; eliminate name dependences by using additional registers

The End?

What We Covered In this Course

- Superscalar Processors: OoO execution, dynamic scheduling, issue logic
- Speculative Execution: Branch prediction, memory dependence prediction
- Technology: Trends, impact on architecture, power and energy
- Domain Specific Accelerators, Dataflow, SIMD, Vector Processors
- Memory Hierarchy: Caches, memory-level parallelism, cache prefetching, replacement and insertion policies, DRAM basics, novel memory technologies
- Parallel Architectures: Multicore processors, shared-memory and distributed memory architecture
- Cache Coherence Protocols and Memory Consistency Models
- Multithreading: SMT, SpMT, VLIW architectures

Other Important Topics Not Covered In This Course

- Graphics Processors: GPUs, GPGPUs
- Dataflow architectures
- Security
- Reliability
- Virtual Memory implementations
- On-chip interconnection networks (Networks-on-Chip “NoC”)
- Synchronization primitives and lock/barrier implementations
- Architecting warehouse-scale computers
- Embedded Processors
- … and many other topics

Reading Assignments

- ARCH Chapter 3.2, 3.7, 3.12 (Read)
- D. Tullsen et al., “Simultaneous Multithreading: Maximizing On-Chip Parallelism,” ISCA 1995 (Read)
- G. Sohi et al., “Multiscalar Processors” (Read)
- J. Renau et al., “Thread-Level Speculation on a CMP Can be Energy Efficient,” ICS, 2005 (Skim)