Lecture 19: Instruction Level Parallelism
-- SMT: Exploiting Thread-Level Parallelism to Improve Uniprocessor Throughput

CSCE 513 Computer Architecture

Department of Computer Science and Engineering

Yonghong Yan

yanyh@cse.sc.edu

https://passlab.github.io/CSCE513

Topics for Instruction Level Parallelism

- 5-stage Pipeline Extension, ILP Introduction, Compiler Techniques, and Branch Prediction
  - C.5, C.6
  - 3.1, 3.2
  - ~~Branch Prediction, C.2, 3.3~~
- Dynamic Scheduling (OOO)
  - 3.4, 3.5
- Hardware Speculation and Static Superscalar/VLIW
  - 3.6, 3.7
- Dynamic Superscalar, Advanced Techniques, ARM Cortex-A53, and Intel Core i7
  - 3.8, 3.9, 3.12
- **SMT: Exploiting Thread-Level Parallelism to Improve Uniprocessor Throughput**
  - **3.11**

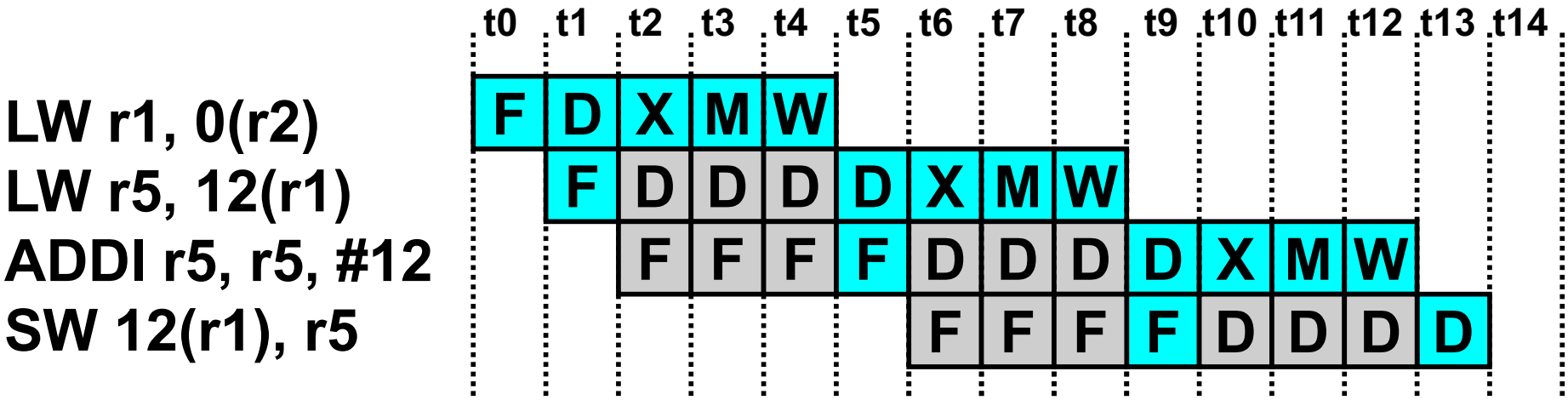
Acknowledge and Copyright

- **Slides adapted from**
  - **UC Berkeley course "Computer Science 252: Graduate Computer Architecture" of David E. Culler Copyright(C) 2005 UCB**
  - **UC Berkeley course Computer Science 252, Graduate Computer Architecture Spring 2012 of John Kubiatowicz Copyright(C) 2012 UCB**
  - **Computer Science 152: Computer Architecture and Engineering, Spring 2016 by Dr. George Michelogiannakis from UC Berkeley**
  - **Arvind (MIT), Krste Asanovic (MIT/UCB), Joel Emer (Intel/MIT), James Hoe (CMU), John Kubiatowicz (UCB), and David Patterson (UCB)**

- **https://passlab.github.io/CSCE513/copyrightack.html**

Pipeline Hazards in ILP

- Each instruction may depend on the next



What is usually done to cope with this?

- ***interlocks (slow)***
- ***or bypassing (needs hardware, doesn't help all hazards)***

Multithreading

- Difficult to continue to extract instruction-level parallelism (ILP) from a single sequential thread of control
- Many workloads can make use of thread-level parallelism (TLP)
  - TLP from multiprogramming (run independent sequential jobs)
  - TLP from multithreaded applications (run one job faster using parallel threads)
- Multithreading uses TLP to improve utilization of a single processor

Multithread Program in OpenMP

```
$ gcc -fopenmp hello.c


$ export OMP_NUM_THREADS=2
$ ./a.out
Hello World
Hello World

$ export OMP_NUM_THREADS=4
$ ./a.out
Hello World
Hello World
Hello World
Hello World
$
```

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

   #pragma omp parallel
   {
          printf("Hello World\n");

   } // End of parallel region

   return(0);
}
```

Typical OpenMP Parallel Program

Sequential code

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

OpenMP *parallel* region

```
#pragma omp parallel shared (a, b)
{
        int id, i, Nthrds, istart, iend;
        id = omp_get_thread_num();
        Nthrds = omp_get_num_threads();
        istart = id * N / Nthrds;
        iend = (id+1) * N / Nthrds;
        for(i=istart;i<iend;i++)   { a[i] = a[i] + b[i]; }
}
```

OpenMP *parallel* region and a worksharing *for* construct

```
#pragma omp parallel shared (a, b) private (i)
#pragma omp for schedule(static)
        for(i=0;i<N;i++)   { a[i] = a[i] + b[i]; }
```


7

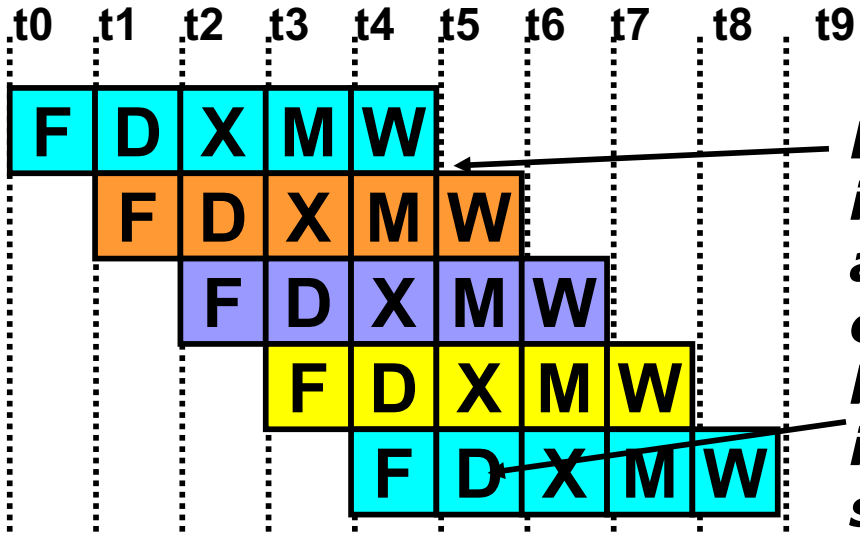
Multithreading

How can we guarantee no dependencies between instructions in a pipeline?

-- One way is to interleave execution of instructions from different program threads on same pipeline

Interleave 4 threads, T1-T4, on non-bypassed 5-stage pipe

T1: LW r1, 0(r2)
T2: ADD r7, r6, r3
T3: XORI r5, r4, #12
T4: SW 0(r8), r9
T1: LW r5, 12(r1)



Prior instruction in a thread always completes write-back before next instruction in same thread reads register file

CDC 6600 Peripheral Processors
(Cray, 1964)

- First multithreaded hardware
- 10 “virtual” I/O processors
- Fixed interleave on simple pipeline
- Pipeline has 100ns cycle time
- Each virtual processor executes one instruction every 1000ns
- Accumulator-based instruction set to reduce processor state

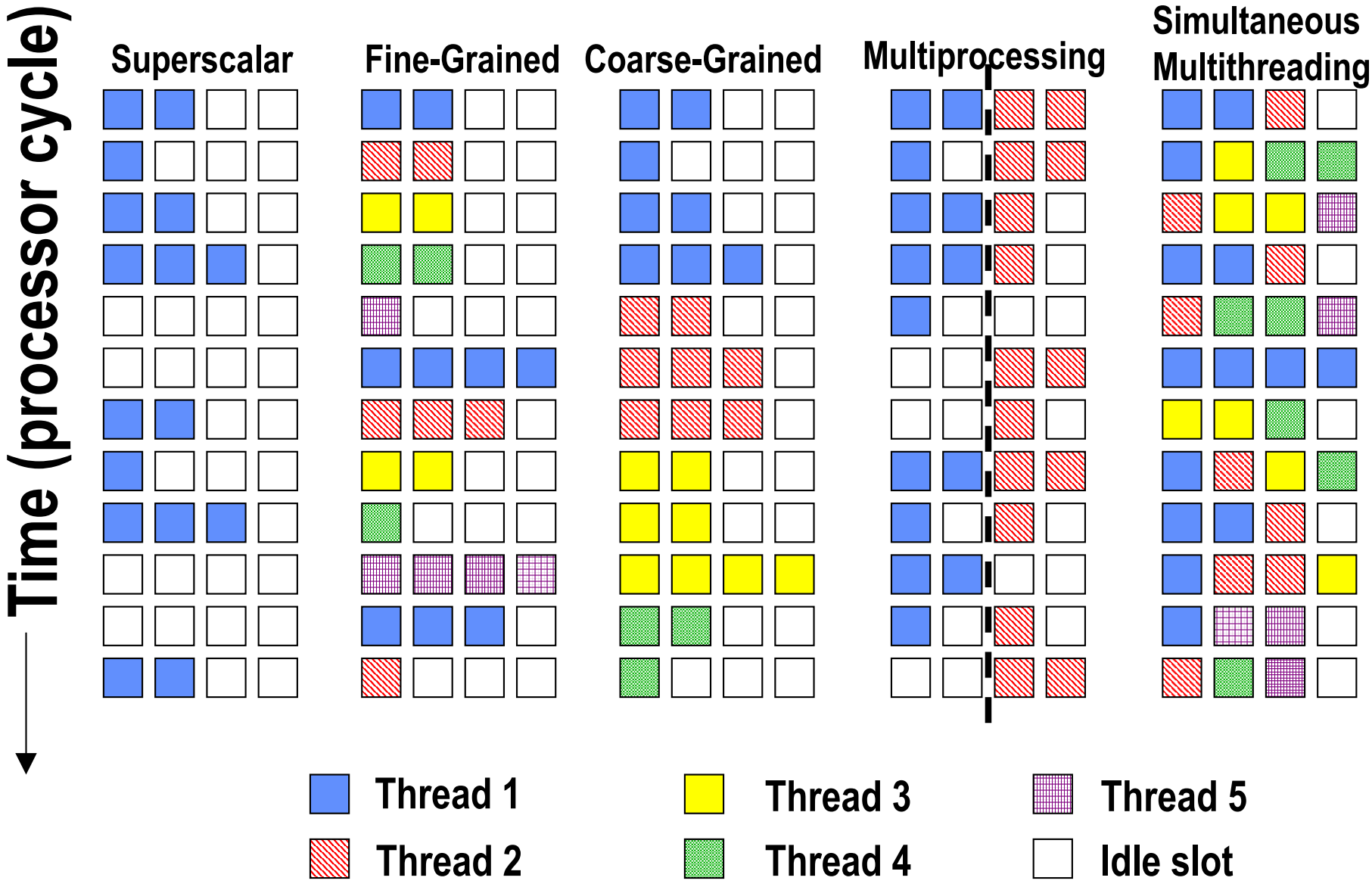
Performance beyond single thread ILP

- There can be much higher natural parallelism in some applications
  - e.g., Database or Scientific codes
  - Explicit Thread Level Parallelism or Data Level Parallelism
- Thread: instruction stream with own PC and data
  - thread may be a process part of a parallel program of multiple processes, or it may be an independent program
  - Each thread has all the state (instructions, data, PC, register state, and so on) necessary to allow it to execute
- Thread Level Parallelism (TLP):
  - Exploit the parallelism inherent between threads to improve performance
- Data Level Parallelism (DLP):
  - Perform identical operations on data, and lots of data

One approach to exploiting threads: Multithreading (TLP within processor)

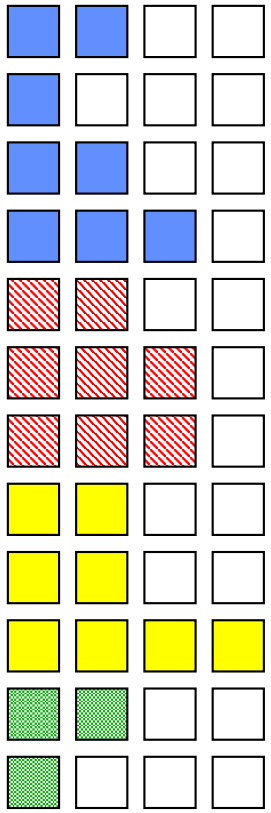
- **Multithreading: multiple threads to share the functional units of 1 processor via overlapping**
  - processor must duplicate independent state of each thread e.g., a separate copy of register file, a separate PC, and for running independent programs, a separate page table
  - memory shared through the virtual memory mechanisms, which already support multiple processes
  - HW for fast thread switch; much faster than full process switch ≈ 100s to 1000s of clocks
- **When switch?**
  - Alternate instruction per thread (fine grain)
  - When a thread is stalled, perhaps for a cache miss, another thread can be executed (coarse grain)

Multithreaded Categories



Course-Grained Multithreading

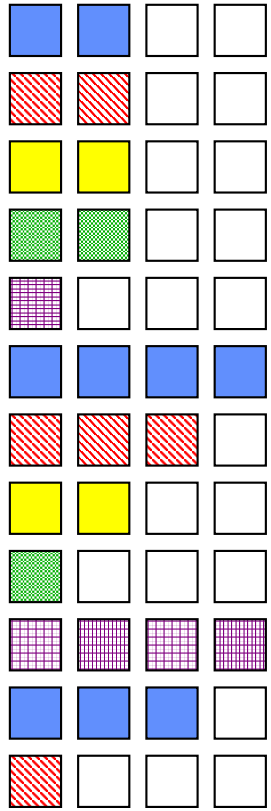
- Switches threads only on costly stalls, such as L2 cache misses
- Advantages
  - Relieves need to have very fast thread-switching
  - Doesn't slow down thread, since instructions from other threads issued only when the thread encounters a costly stall
- Disadvantage is hard to overcome throughput losses from shorter stalls, due to pipeline start-up costs
  - Since CPU issues instructions from 1 thread, when a stall occurs, the pipeline must be emptied or frozen
  - New thread must fill pipeline before instructions can complete



- Because of this start-up overhead, coarse-grained multithreading is better for reducing penalty of high cost stalls, where pipeline refill << stall time
- Used in IBM AS/400, Sparcle (for Alewife)

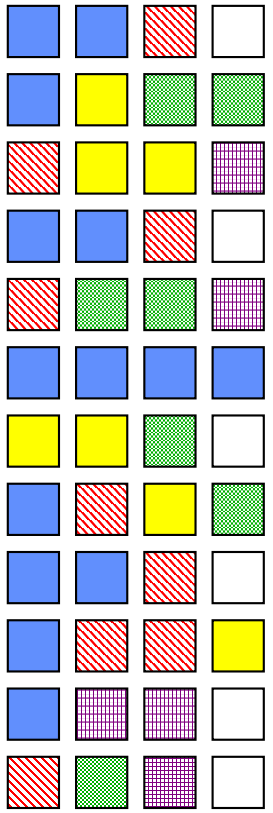
Fine-Grained Multithreading

- Switches between threads on each instruction, causing the execution of multiples threads to be interleaved
  - Usually done in a round-robin fashion, skipping any stalled threads
  - CPU must be able to switch threads every clock
- Advantage:
  - can hide both short and long stalls, since instructions from other threads executed when one thread stalls
- Disadvantage:
  - slows down execution of individual threads, since a thread ready to execute without stalls will be delayed by instructions from other threads
- Used on Oracle SPARC processor (Niagra from Sun), several research multiprocessors, Tera



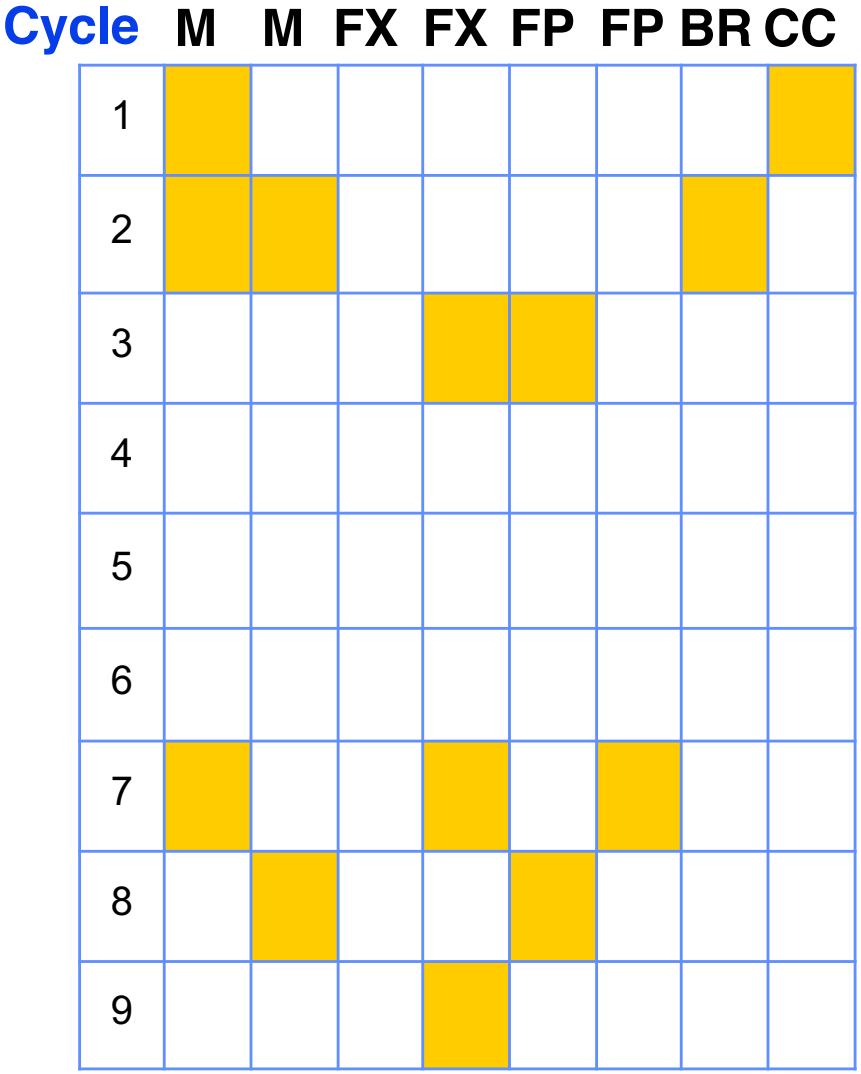
Simultaneous Multithreading (SMT): Do both ILP and TLP

- TLP and ILP exploit two different kinds of parallel structure in a program
- Could a processor oriented at ILP to exploit TLP?
  - functional units are often idle in data path designed for ILP because of either stalls or dependences in the code
- Could the TLP be used as a source of independent instructions that might keep the processor busy during stalls?
- Could TLP be used to employ the functional units that would otherwise lie idle when insufficient ILP exists?

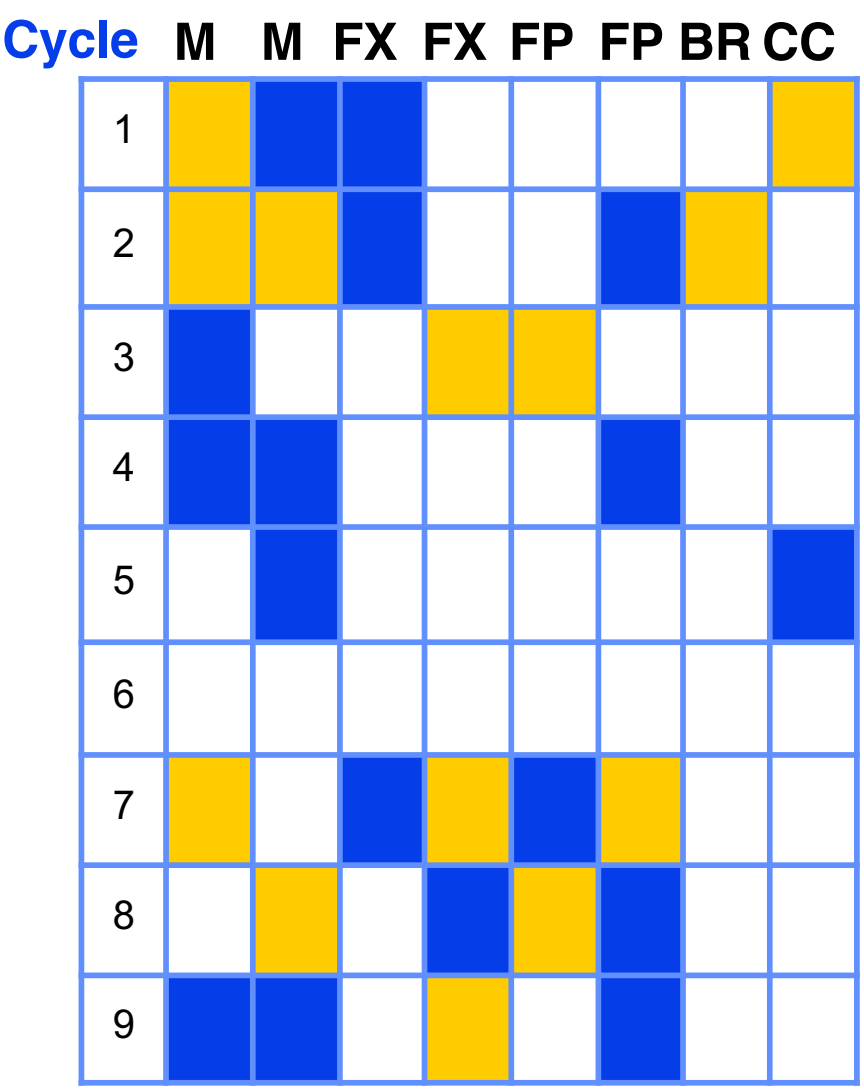


Simultaneous Multi-threading ...

One thread, 8 units



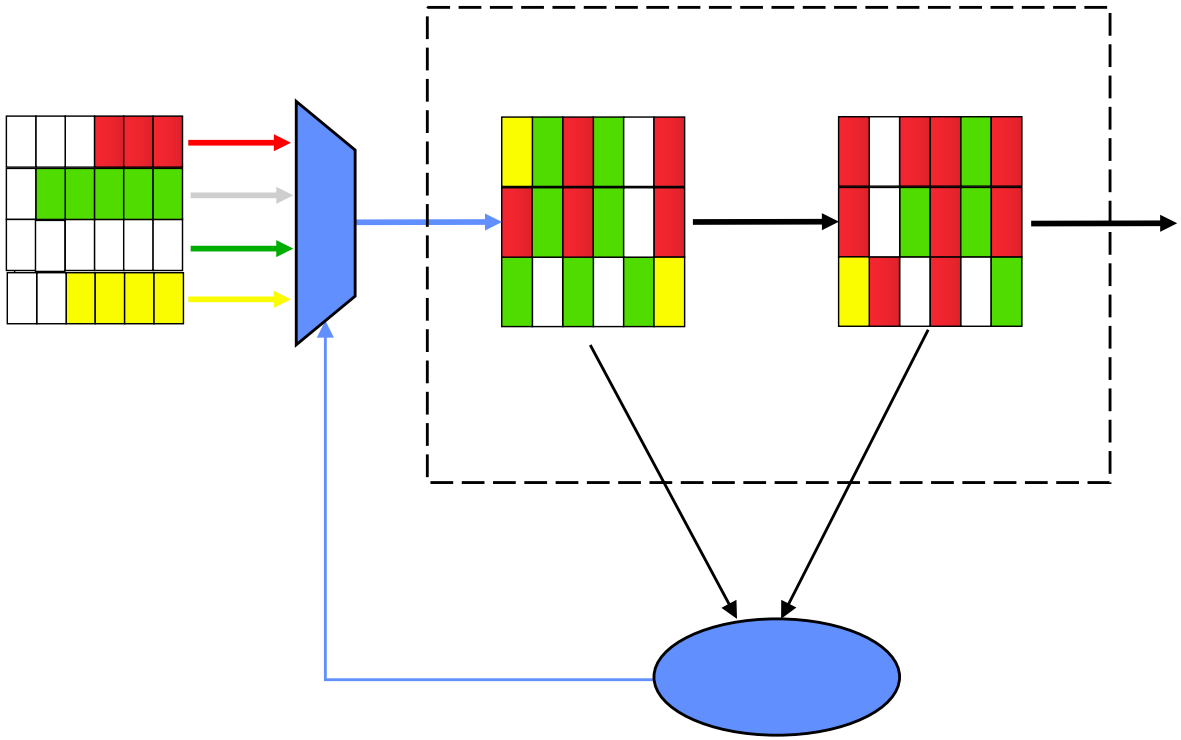
Two threads, 8 units



M = Load/Store, FX = Fixed Point, FP = Floating Point, BR = Branch, CC = Condition Codes

Choosing Policy

- **Among four threads, from which do we fetch?**
  - **Fetch from thread with the least instructions in flight.**



Simultaneous Multithreading Details

- **Simultaneous multithreading (SMT): insight that dynamically scheduled processor already has many HW mechanisms to support multithreading**
  - **Large set of virtual registers that can be used to hold the register sets of independent threads**
  - **Register renaming provides unique register identifiers, so instructions from multiple threads can be mixed in datapath without confusing sources and destinations across threads**
  - **Out-of-order completion allows the threads to execute out of order, and get better utilization of the HW**
- **Just adding a per thread renaming table and keeping separate PCs**
  - **Independent commitment can be supported by logically keeping a separate reorder buffer for each thread**

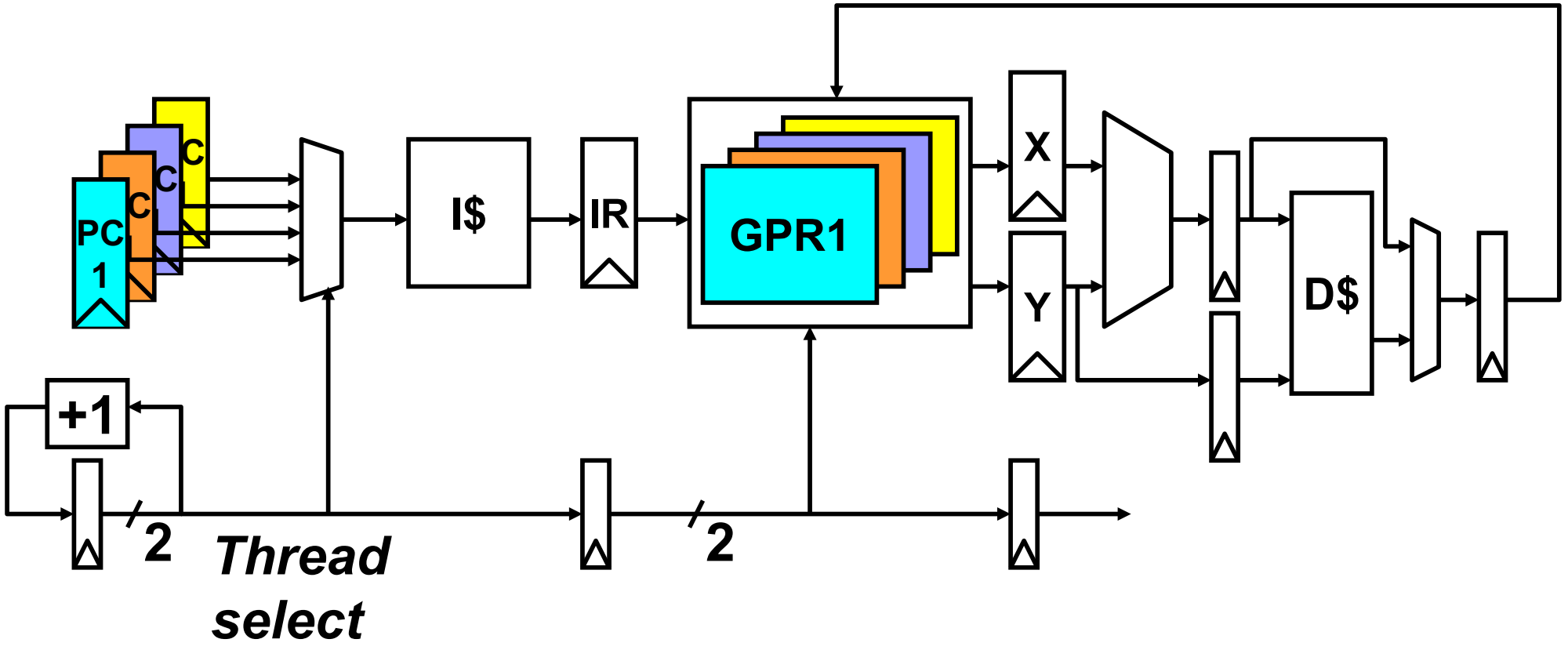
Source: Micrprocessor Report, December 6, 1999
"Compaq Chooses SMT for Alpha"

Design Challenges in SMT

- Since SMT makes sense only with fine-grained implementation, impact of fine-grained scheduling on single thread performance?
  - A preferred thread approach sacrifices neither throughput nor single-thread performance?
  - Unfortunately, with a preferred thread, the processor is likely to sacrifice some throughput, when preferred thread stalls
- Larger register file needed to hold multiple contexts
- Clock cycle time, especially in:
  - Instruction issue - more candidate instructions need to be considered
  - Instruction completion - choosing which instructions to commit may be challenging
- Ensuring that cache and TLB conflicts generated by SMT do not degrade performance

Simple Multithreaded Pipeline

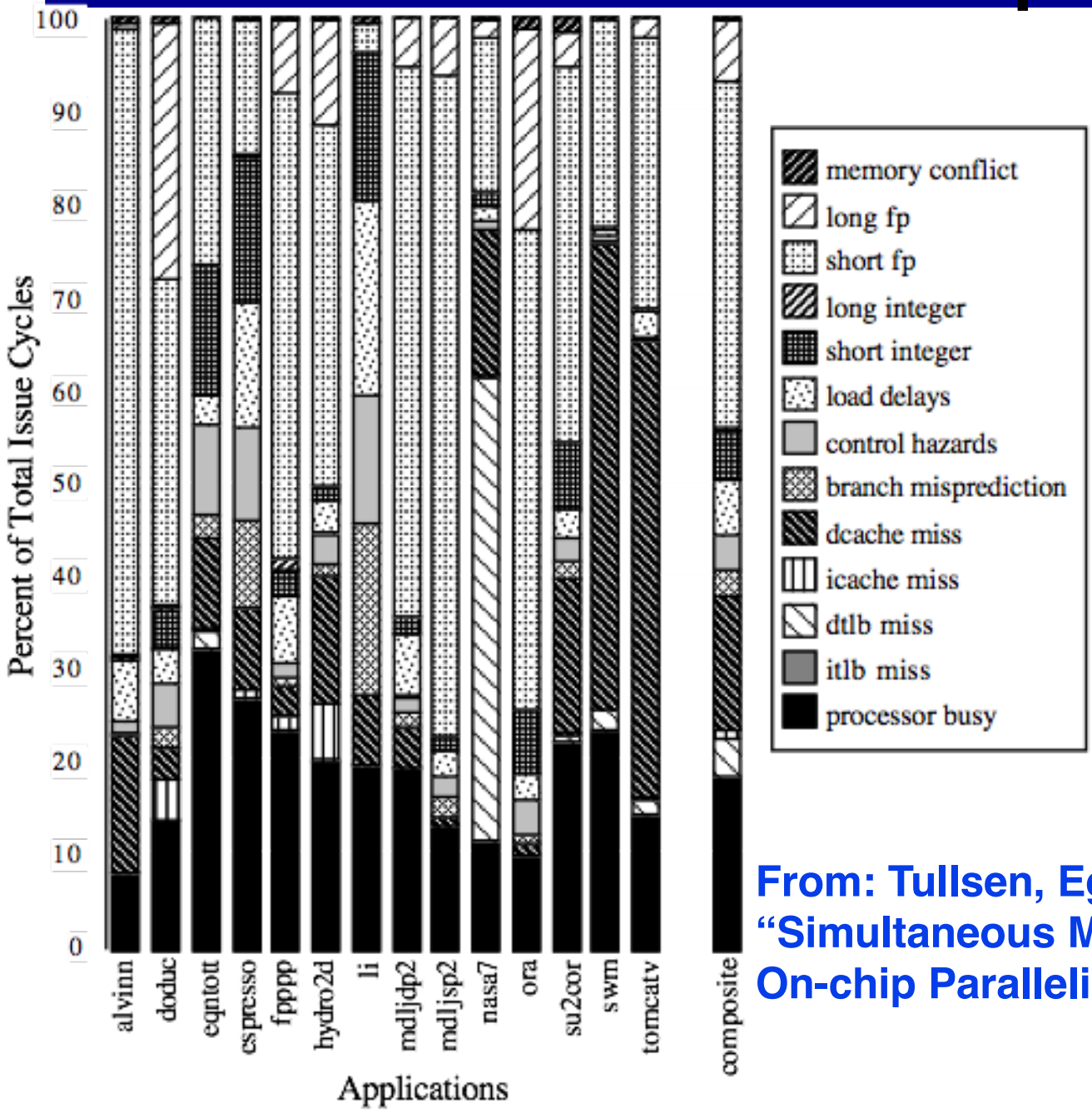
- Have to carry thread select down pipeline to ensure correct state bits read/written at each pipe stage
- Appears to software (including OS) as multiple, albeit slower, CPUs



Multithreading Costs

- Each thread requires its own user state
  - PC
  - GPRs
- Also, needs its own system state
  - Virtual-memory page-table-base register
  - Exception-handling registers
- *Other overheads:*
  - Additional cache/TLB conflicts from competing threads
  - (or add larger cache/TLB capacity)
  - More OS overhead to schedule more threads (where do all these threads come from?)

For most apps, most execution units lie idle in an OoO superscalar



For an 8-way superscalar.

"Processor busy"

are

the actual

used issue slots

From: Tullsen, Eggers, and Levy, "Simultaneous Multithreading: Maximizing On-chip Parallelism", ISCA 1995.

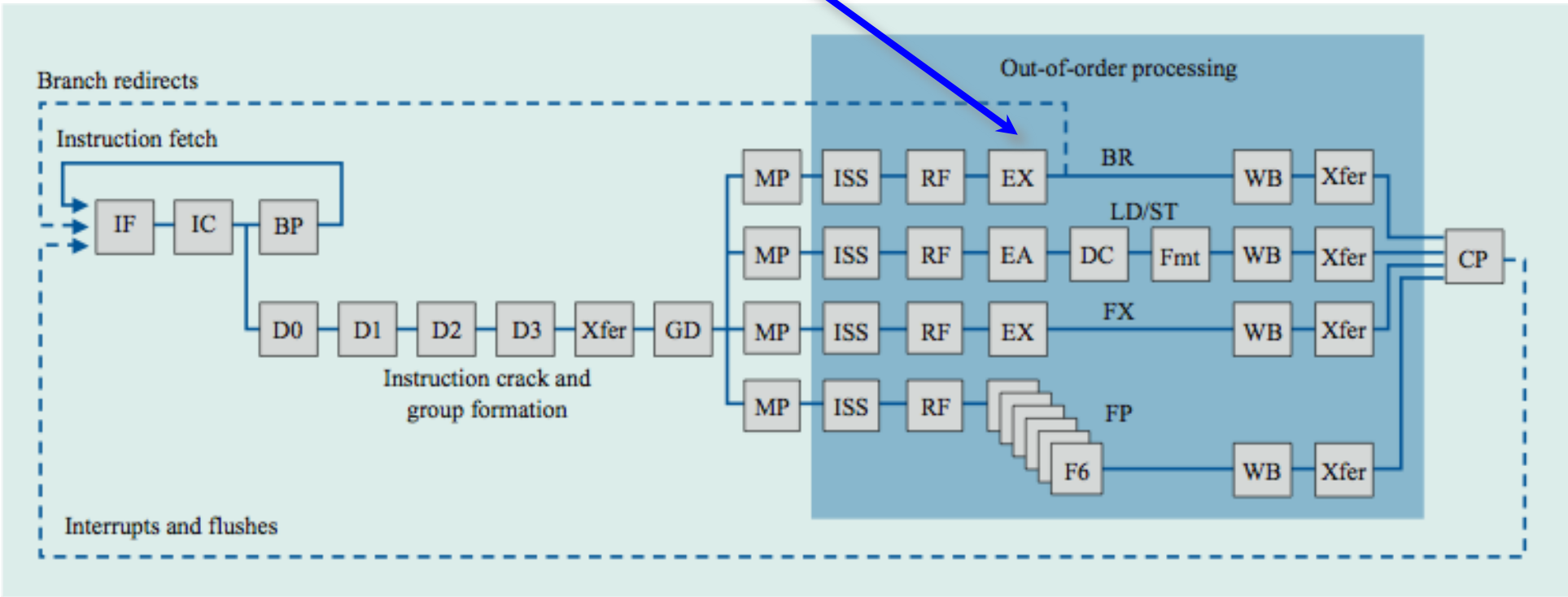
O-o-O Simultaneous Multithreading

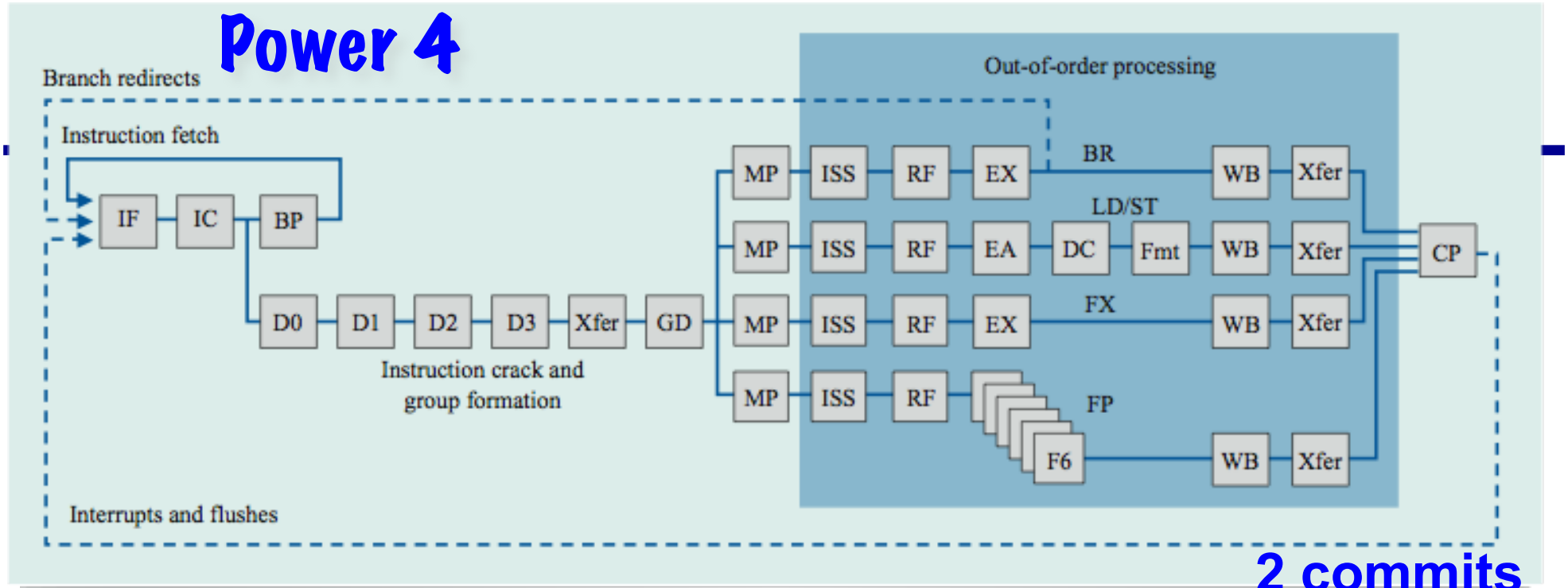
[Tullsen, Eggers, Emer, Levy, Stamm, Lo, DEC/UW, 1996]

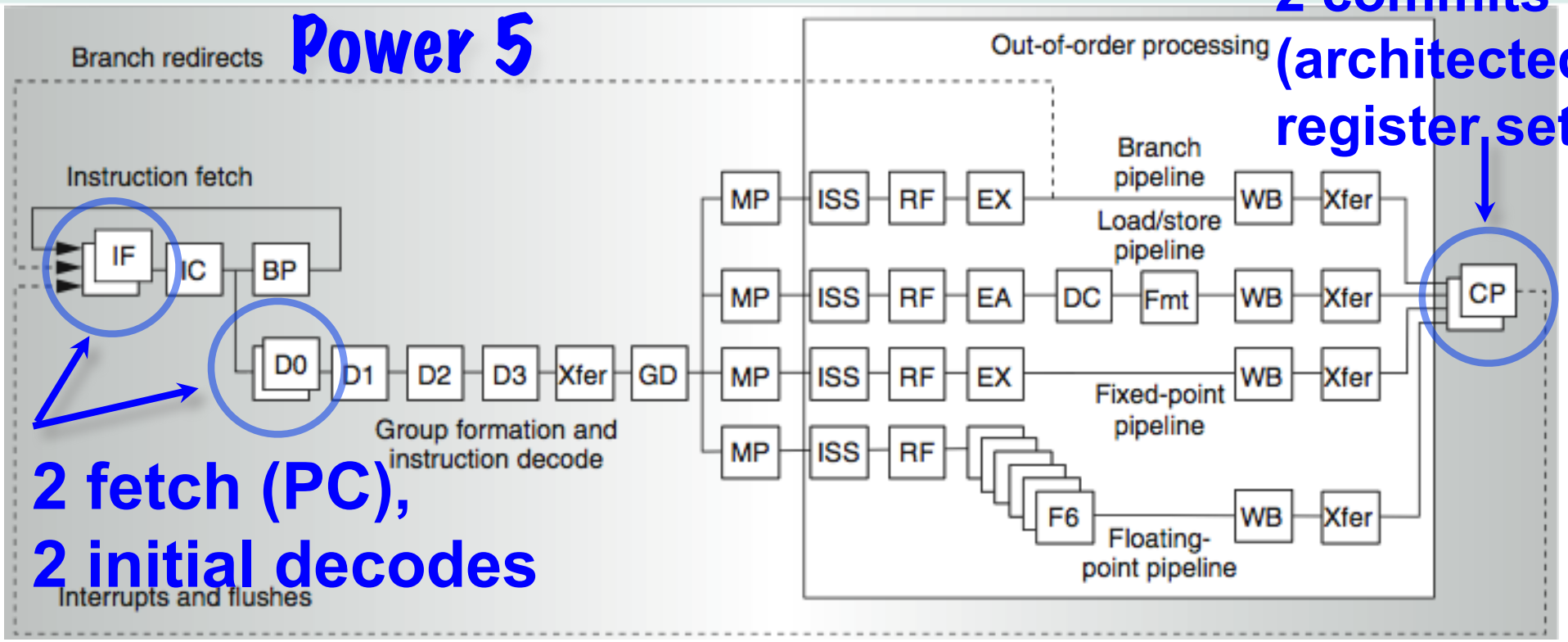
- Add multiple contexts and fetch engines and allow instructions fetched from different threads to issue simultaneously
- Utilize wide out-of-order superscalar processor issue queue to find instructions to issue from multiple threads
- OOO instruction window already has most of the circuitry required to schedule from multiple threads
- Any single thread can utilize whole machine

Power 4

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



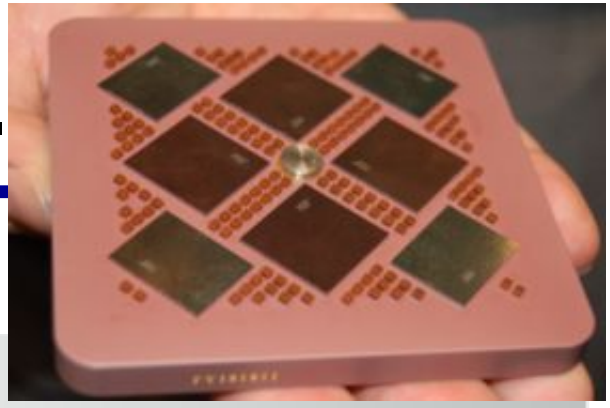


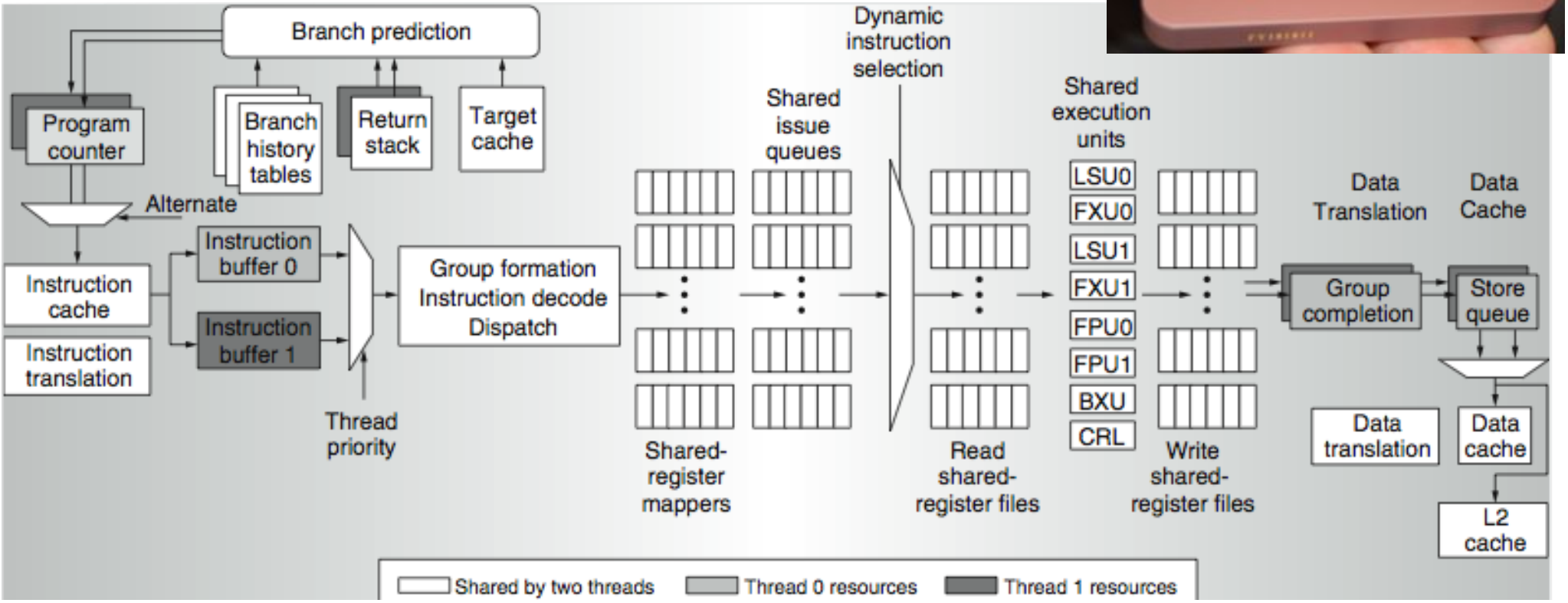


2 commits (architected register sets)

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

Changes in Power 5 to support SMT

- Increased associativity of L1 instruction cache and the instruction address translation buffers
- Added per thread load and store queues
- Increased size of the L2 (1.92 vs. 1.44 MB) and L3 caches
- Added separate instruction prefetch and buffering per thread
- Increased the number of virtual registers from 152 to 240
- Increased the size of several issue queues
- The Power5 core is about 24% larger than the Power4 core because of the addition of SMT support

Pentium-4 Hyperthreading (2002)

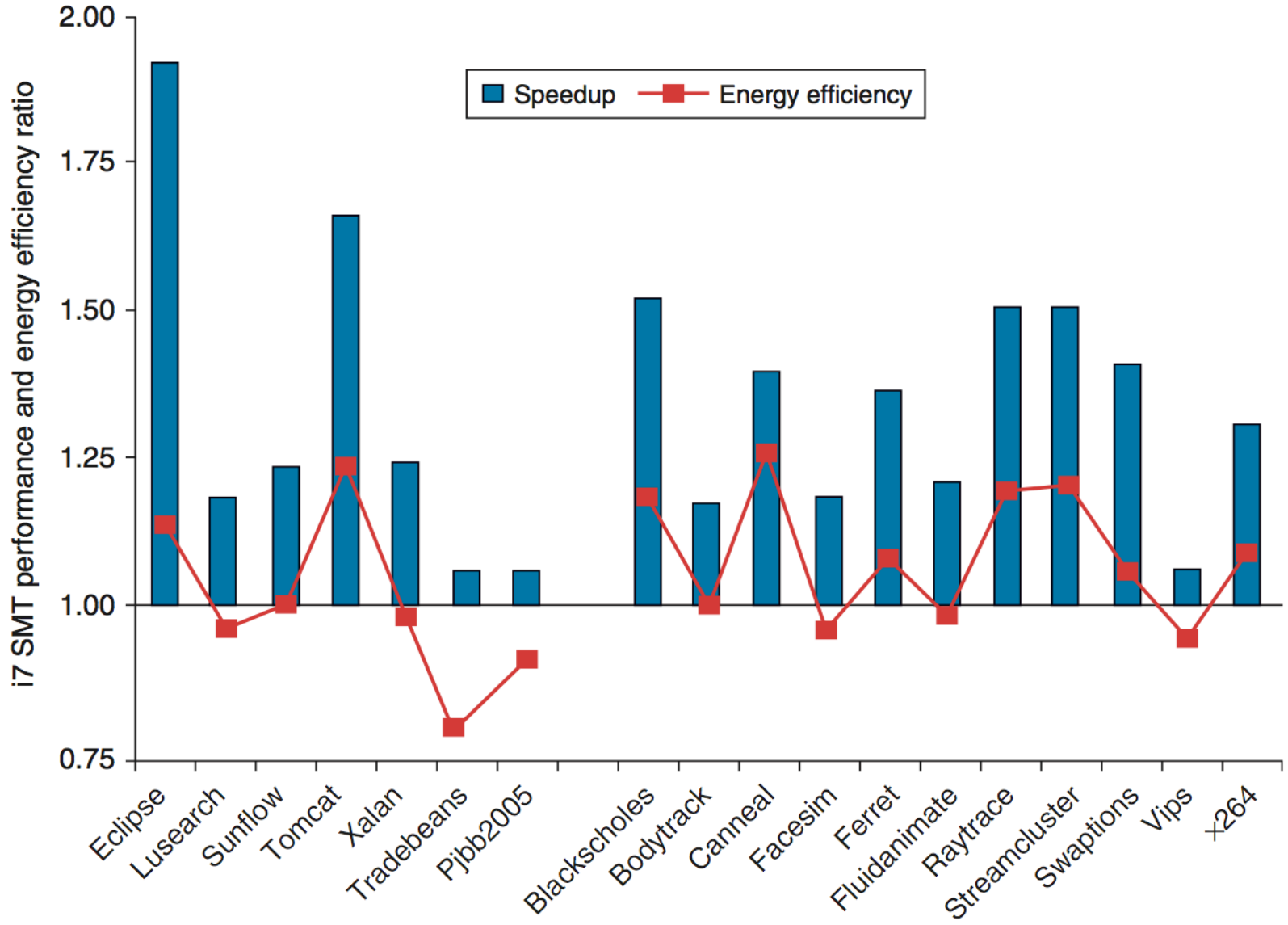
- First commercial SMT design (2-way SMT)
  - Hyperthreading == SMT
- Logical processors share nearly all resources of the physical processor
  - Caches, execution units, branch predictors
- Die area overhead of hyperthreading ~ 5%
- When one logical processor is stalled, the other can make progress
  - No logical processor can use all entries in queues when two threads are active
- Processor running only one active software thread runs at approximately same speed with or without hyperthreading
- Hyperthreading dropped on OoO P6 based follow-ons to Pentium-4 (Pentium-M, Core Duo, Core 2 Duo), until revived with Nehalem generation machines in 2008.
- Intel Atom (in-order x86 core) has two-way vertical multithreading

Initial Performance of SMT

- Pentium 4 Extreme SMT yields 1.01 speedup for SPECint_rate benchmark and 1.07 for SPECfp_rate
  - Pentium 4 is dual threaded SMT
  - SPECRate requires that each SPEC benchmark be run against a vendor-selected number of copies of the same benchmark
- Running on Pentium 4 each of 26 SPEC benchmarks paired with every other (26^2 runs) speed-ups from 0.90 to 1.58; average was 1.20
- Power 5, 8-processor server 1.23 faster for SPECint_rate with SMT, 1.16 faster for SPECfp_rate
- Power 5 running 2 copies of each app speedup between 0.89 and 1.41
  - Most gained some
  - Fl.Pt. apps had most cache conflicts and least gains

Intel i7 Performance

- 2-thread SMT



End of Chapter 3