Computer Architecture
Multithreading and Hardware Security

Ting-Jung Chang

NYCU CS

Agenda

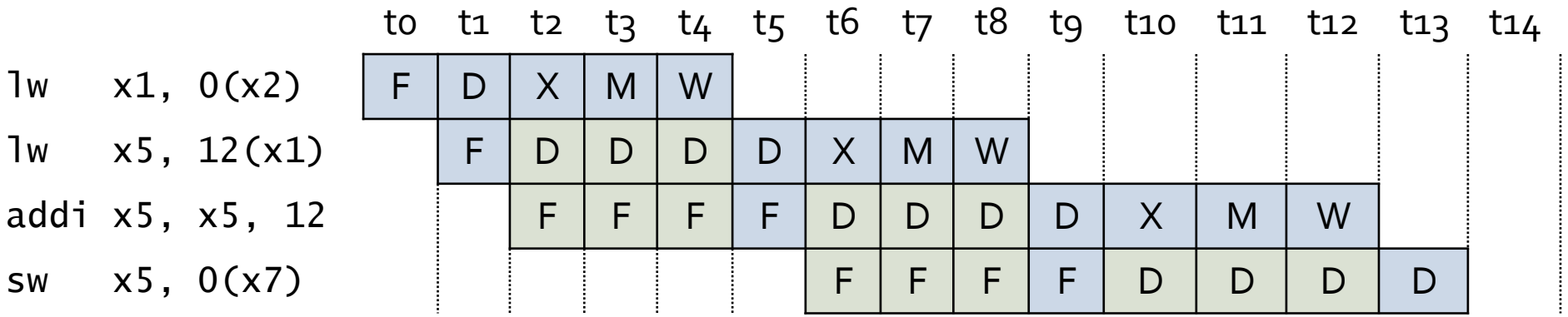
- Multithreading Motivation
- Coarse Grain Multithreading
- Simultaneous Multithreading
- Hardware Security

Multithreading

Thread-Level Parallelism

- Difficult to continue to extract instruction-level parallelism (ILP) or data level parallelism (DLP) 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

Pipeline Hazards

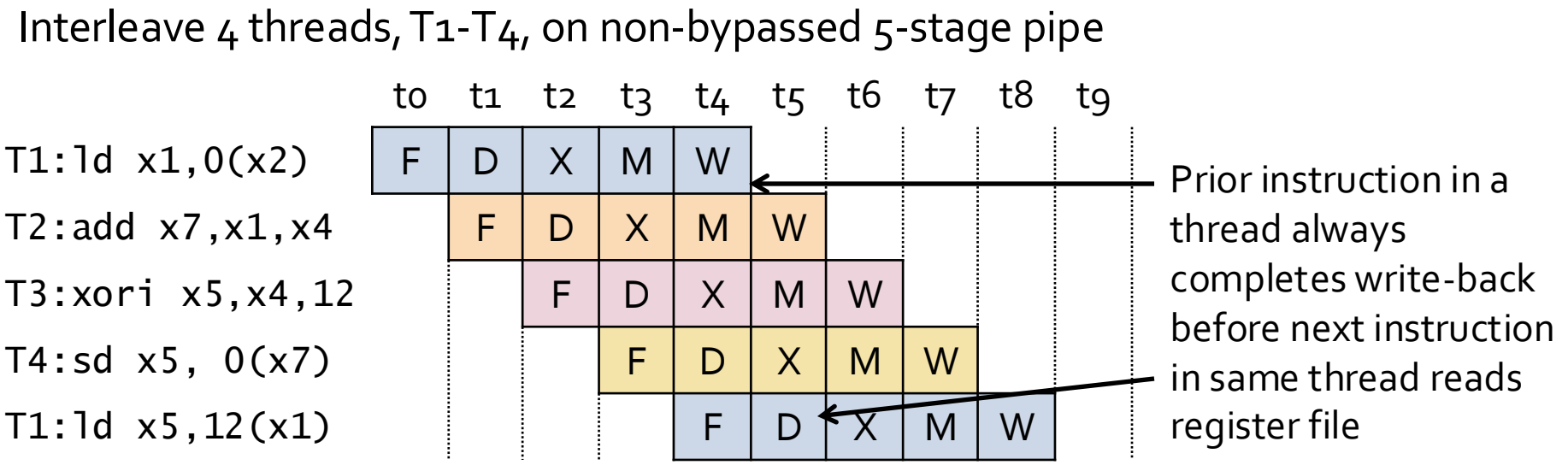


- Each instruction may depend on the next
- What is usually done to cope with this?
  - interlocks (slow)
  - or bypassing (needs hardware, doesn't help all hazards)

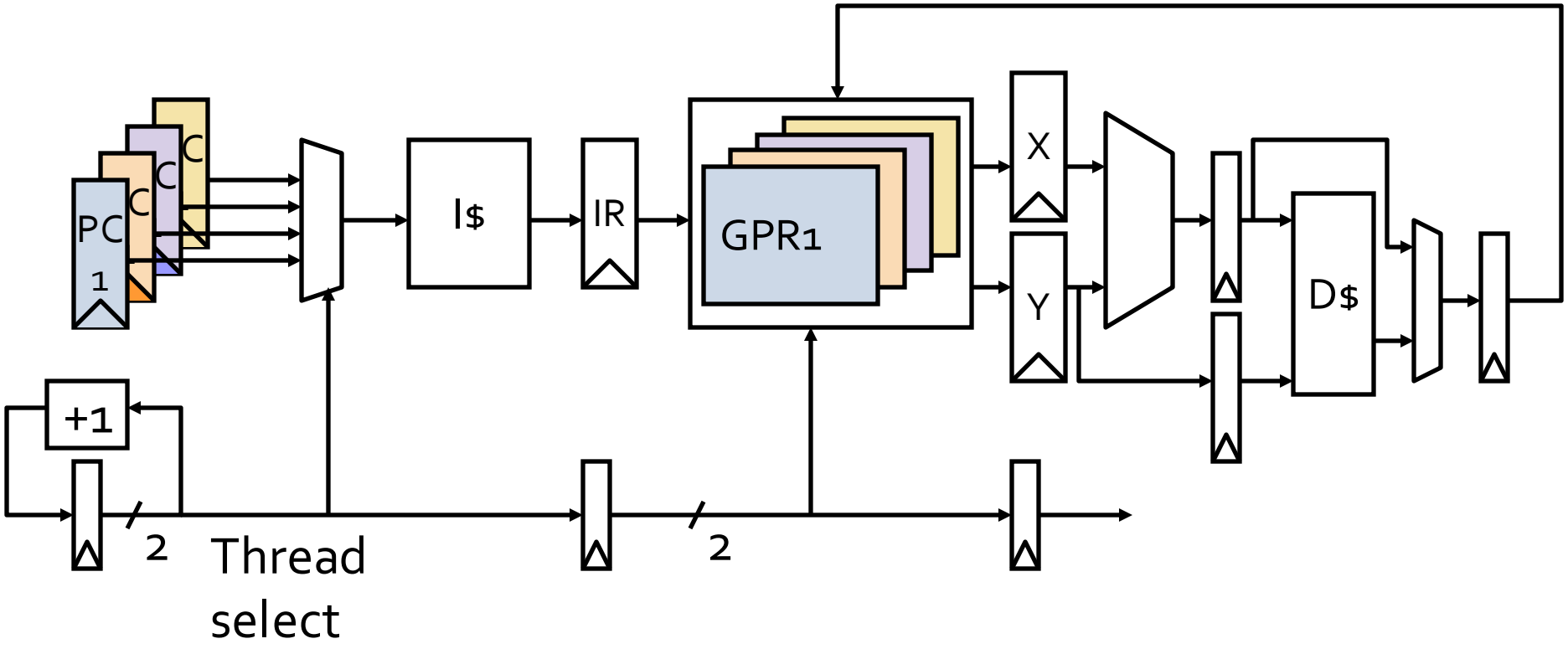
Multithreading

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



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 system state
- 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?)

Thread Scheduling Policies

- Fixed interleave (CDC 6600 PPUs, 1964)
  - Each of N threads executes one instruction every N cycles
  - If thread not ready to go in its slot, insert pipeline bubble
  - Can potentially remove bypassing and interlocking logic

- Software-controlled interleave (TI ASC PPUs, 1971)
  - OS allocates S pipeline slots amongst N threads
  - Hardware performs fixed interleave over S slots, executing whichever thread is in that slot

- Hardware-controlled thread scheduling (HEP, 1982)
  - Hardware keeps track of which threads are ready to go
  - Picks next thread to execute based on hardware priority scheme

Agenda

- Multithreading Motivation
- Coarse Grain Multithreading
- Simultaneous Multithreading
- Hardware Security

Coarse-Grain Hardware Multithreading

- Some architectures do not have many low-latency bubbles
- Add support for a few threads to hide occasional cache miss latency
- Swap threads in hardware on cache miss

Denelcor HEP

(Burton Smith, 1982)

BRL HEP Machine

Image Credit:
Denelcor
http://ftp.arl.army.mil/ftp/historic-computers/png/hep2.png

First commercial machine to use hardware threading in main CPU

- 120 threads per processor
- 10 MHz clock rate
- Up to 8 processors
- precursor to Tera MTA / Cray XMT / YARC Data / Cray URIKA

Tera (Cray) MTA (1990)

- Up to 256 processors
- Up to 128 active threads per processor
- Flat, shared main memory
  - No data cache
  - Sustains one main memory access per cycle per processor
- Tera MTA designed for supercomputing applications with large data sets and low locality
  - Many parallel threads needed to hide large memory latency

Image Credit:
Tera Computer Company

Oracle/Sun Niagara processors

- Target is datacenters running web servers and databases, with many concurrent requests
- Provide multiple simple cores each with multiple hardware threads, reduced energy/operation though much lower single thread performance

- Niagara-1 [2004], 8 cores, 4 threads/core
- Niagara-2 [2007], 8 cores, 8 threads/core
- Niagara-3 [2009], 16 cores, 8 threads/core
- SPARC M7 [2015], 32 core, 8 threads/core
- SPARC M8 [2017], 32 core, 8 threads/core

Oracle ended SPARC programs after M8 in 2017

Oracle/Sun Niagara-3, "Rainbow Falls" 2009

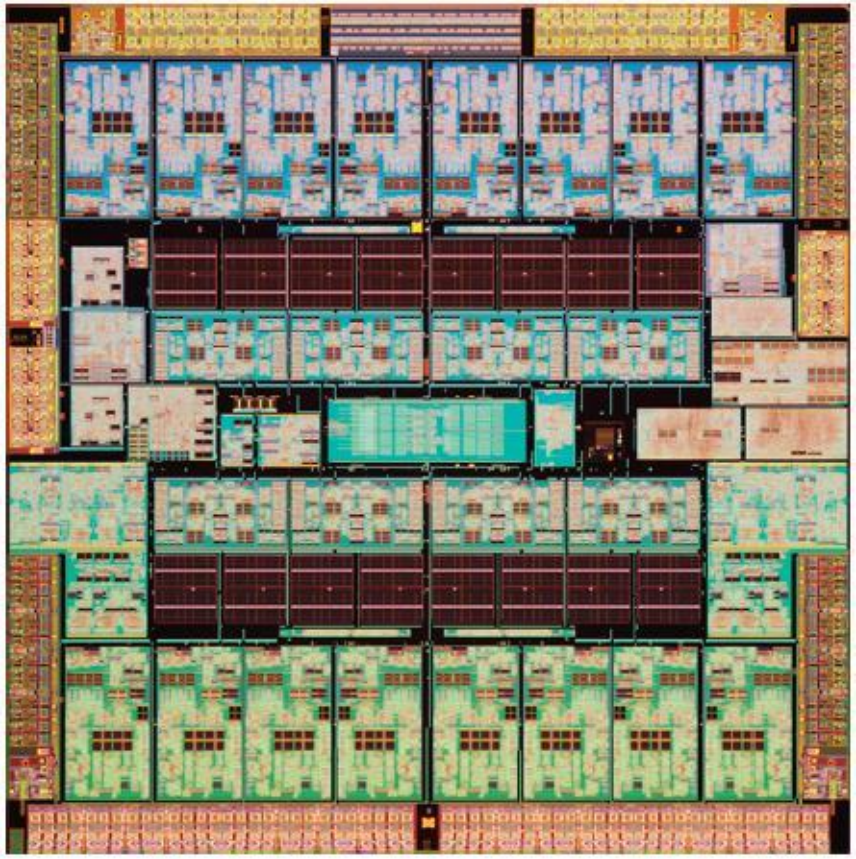


Image Credit: Oracle/Sun

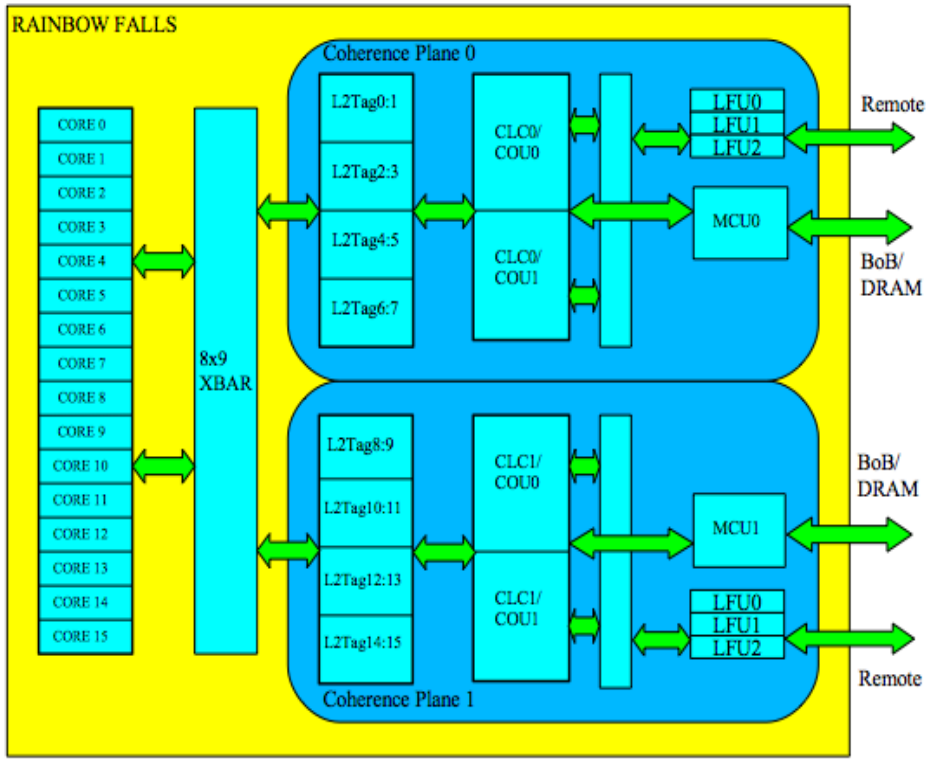


Image Credit: Oracle/Sun
From Hot Chips 2009 Presentation by Sanjay Patel

Multithreading Design Choices

- Tera MTA designed for supercomputing applications with large data sets and low locality
  - No data cache
  - Many parallel threads needed to hide large memory latency
- Other applications are more cache friendly
  - Few pipeline bubbles when cache getting hits
  - Just add a few threads to hide occasional cache miss latencies
  - Swap threads on cache misses -> Coarse-grained Multithreading

Multithreading Design Choices

- Fine-grained multithreading
  - Context switch among threads every cycle
- Coarse-grained multithreading
  - Context switch among threads every few cycles, e.g., on:
    - Function unit data hazard,
    - L1 miss,
    - L2 miss…
- Why choose one style over another?
- Choice depends on
  - Context-switch overhead
  - Number of threads supported (due to per-thread state)
  - Expected application-level parallelism…

Agenda

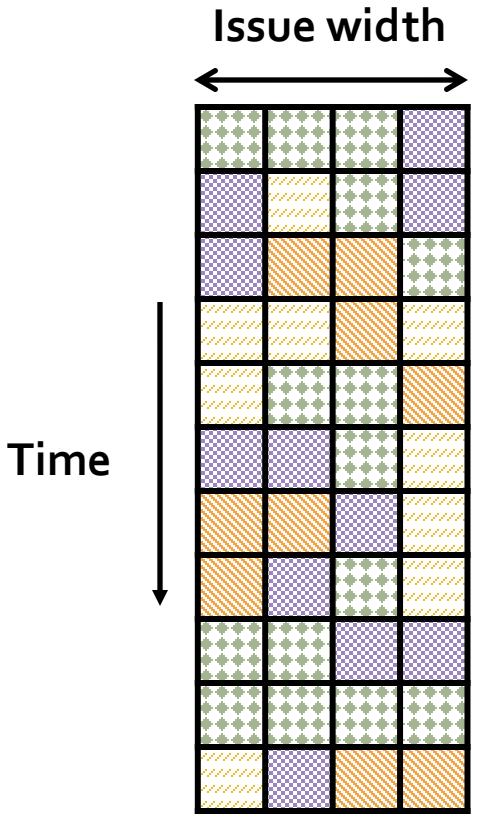
- Multithreading Motivation
- Coarse Grain Multithreading
- Simultaneous Multithreading
- Hardware Security

Simultaneous Multithreading (SMT) for OoO Superscalars

- Techniques presented so far have all been "vertical" multithreading where each pipeline stage works on one thread at a time
- SMT uses fine-grain control already present inside an OoO superscalar to allow instructions from multiple threads to enter execution on same clock cycle. Gives better utilization of machine resources

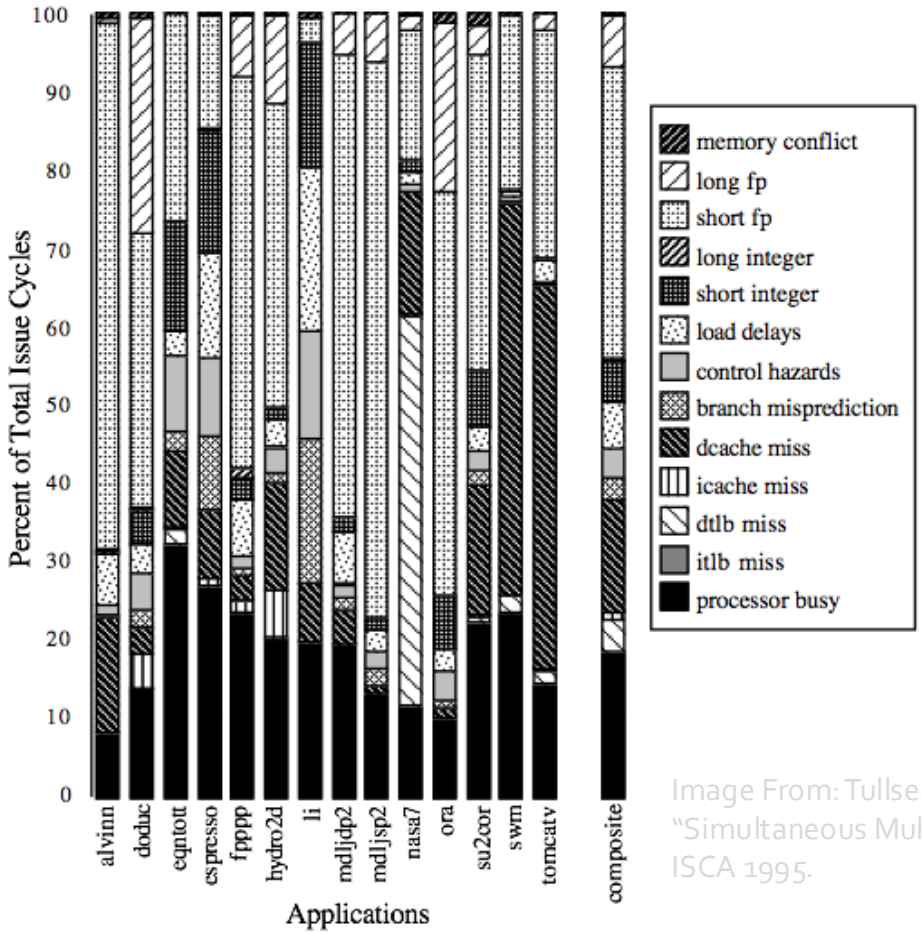
Ideal Superscalar Multithreading

[Tullsen, Eggers, Levy, UW, 1995]



- Interleave multiple threads to multiple issue slots with no restrictions

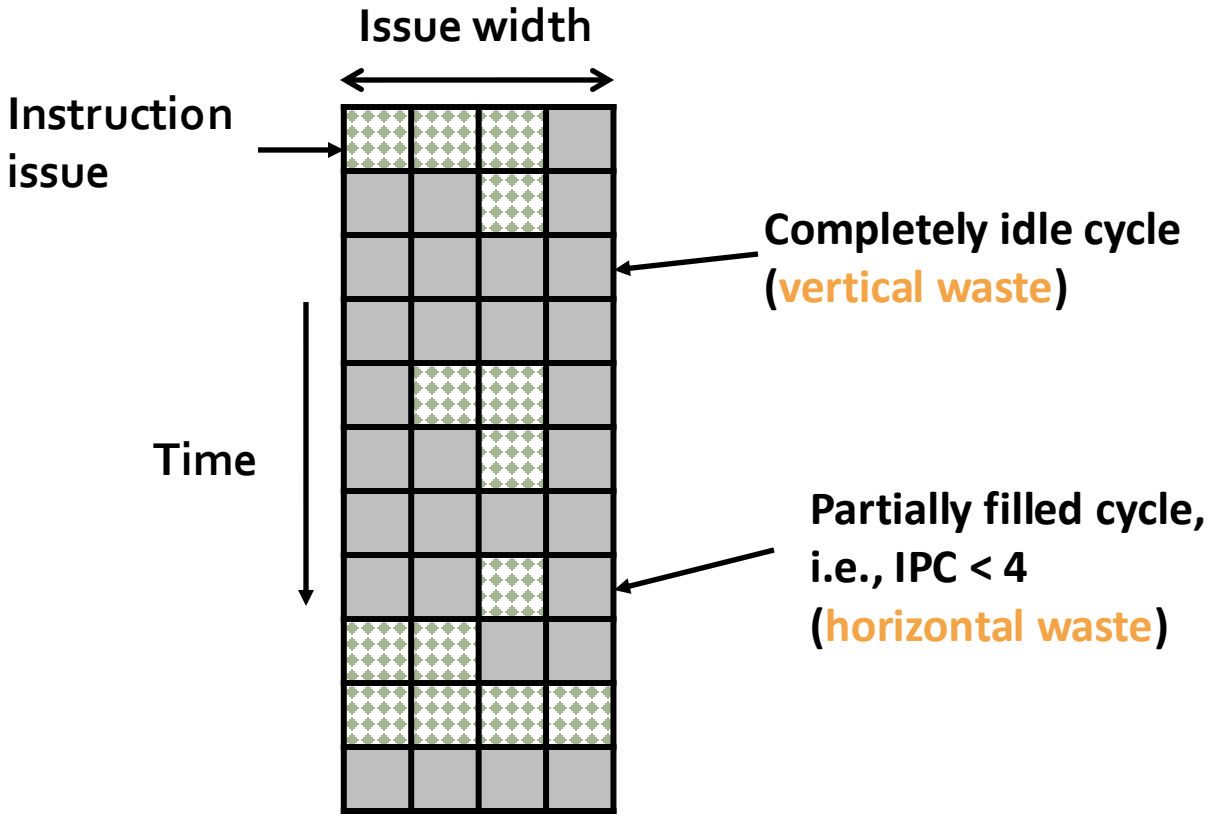
For most apps, most execution units lie idle in an OoO superscalar



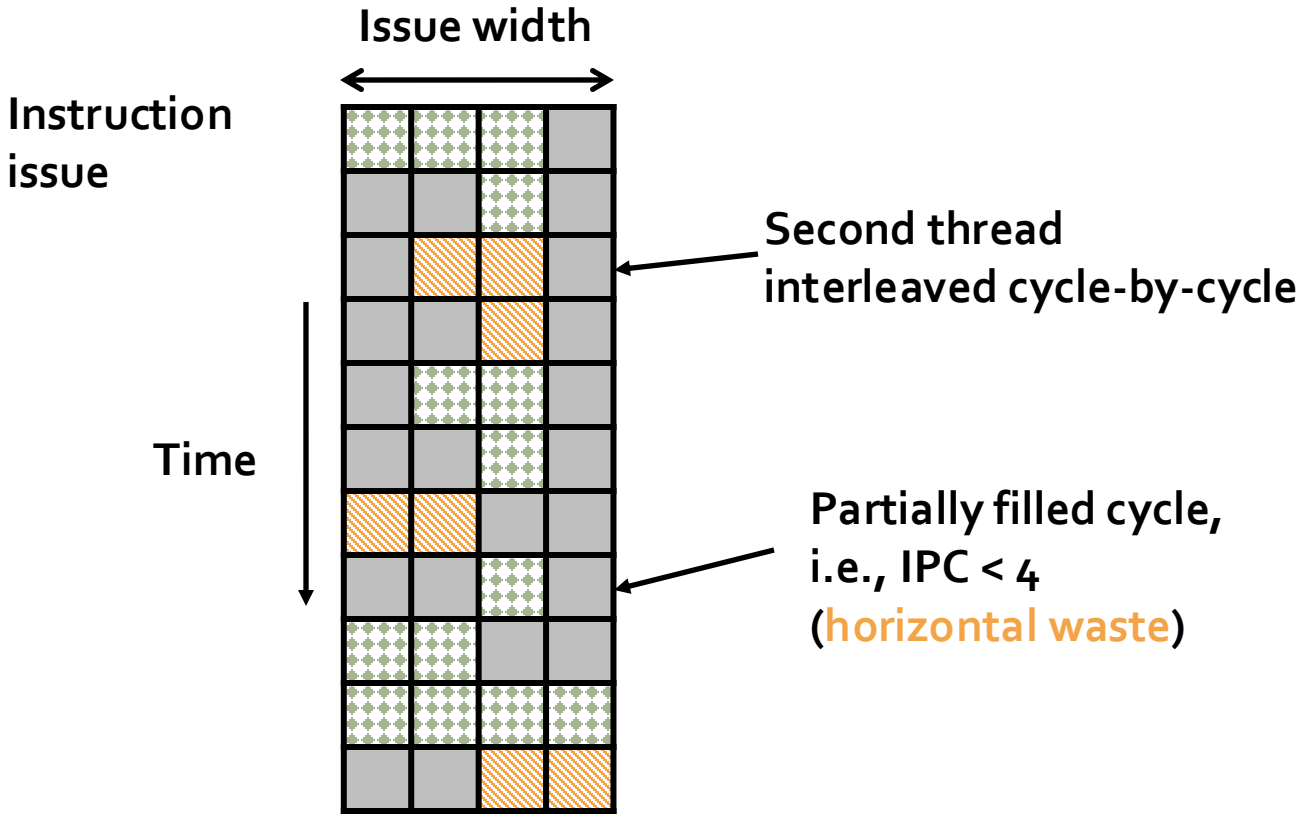
For an 8-way superscalar.

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

Superscalar Machine Efficiency



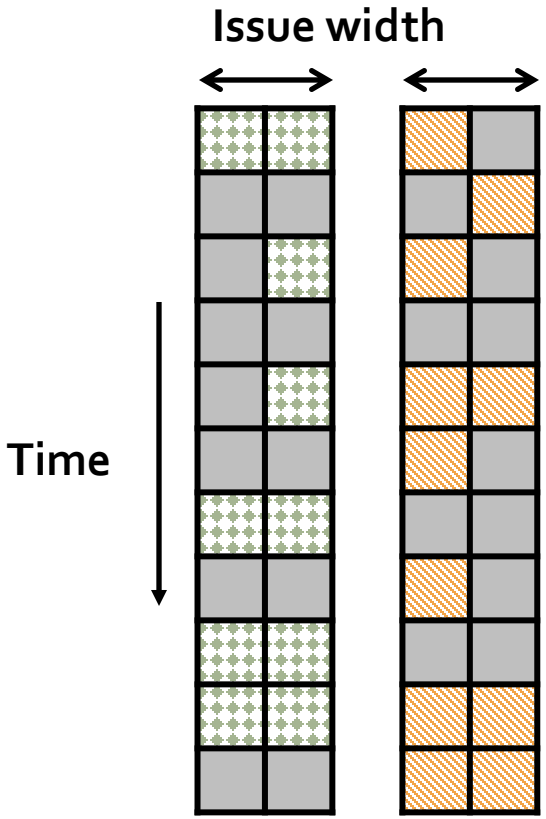
Vertical Multithreading



What is the effect of cycle-by-cycle interleaving?

removes vertical waste, but leaves some horizontal waste

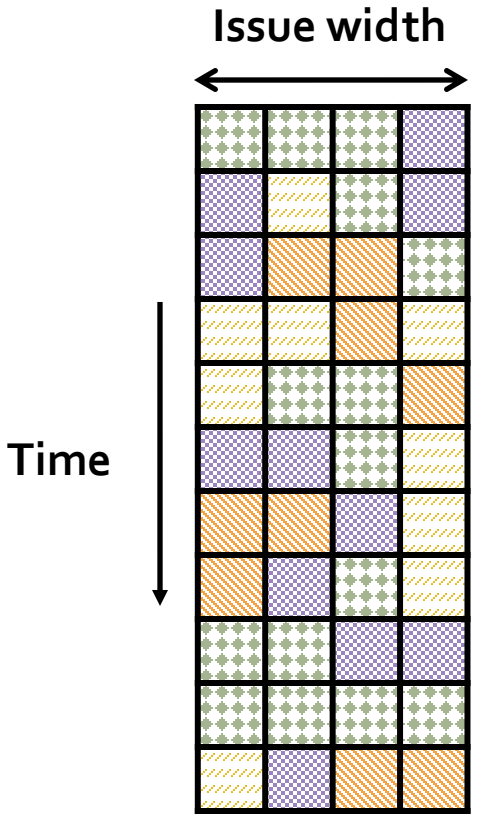
Chip Multiprocessing (CMP)



- What is the effect of splitting into multiple processors?
  - reduces horizontal waste
  - leaves some vertical waste
  - puts upper limit on peak throughput of each thread

Ideal Superscalar Multithreading

[Tullsen, Eggers, Levy, UW, 1995]



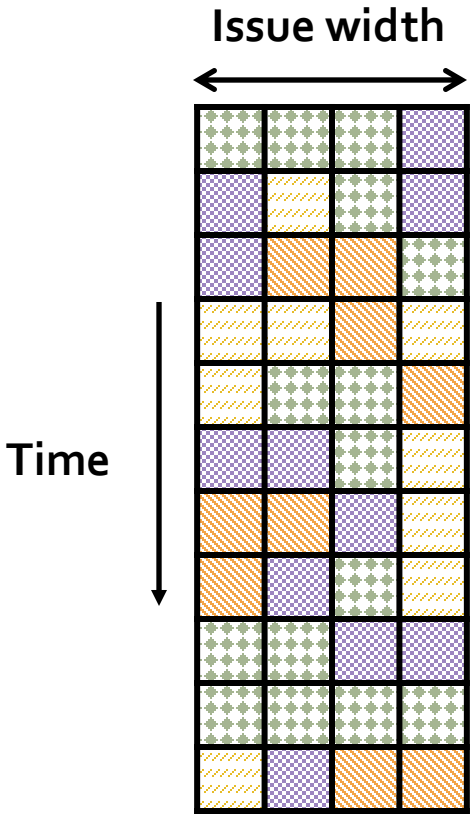
- Interleave multiple threads to multiple issue slots with no restrictions

OoO Simultaneous Multithreading

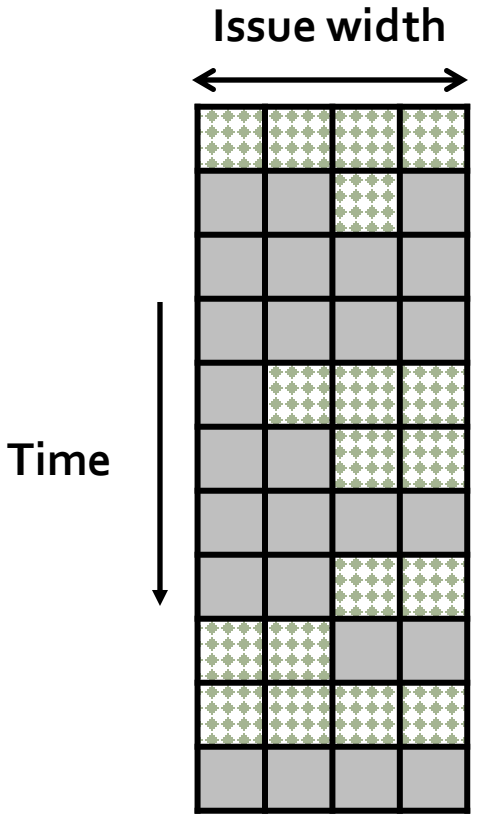
[Tullsen, Eggers, Emer, Levy, Stamm, Lo, DEC/UW, 1996]

- 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

SMT Adaptation to Parallelism Type

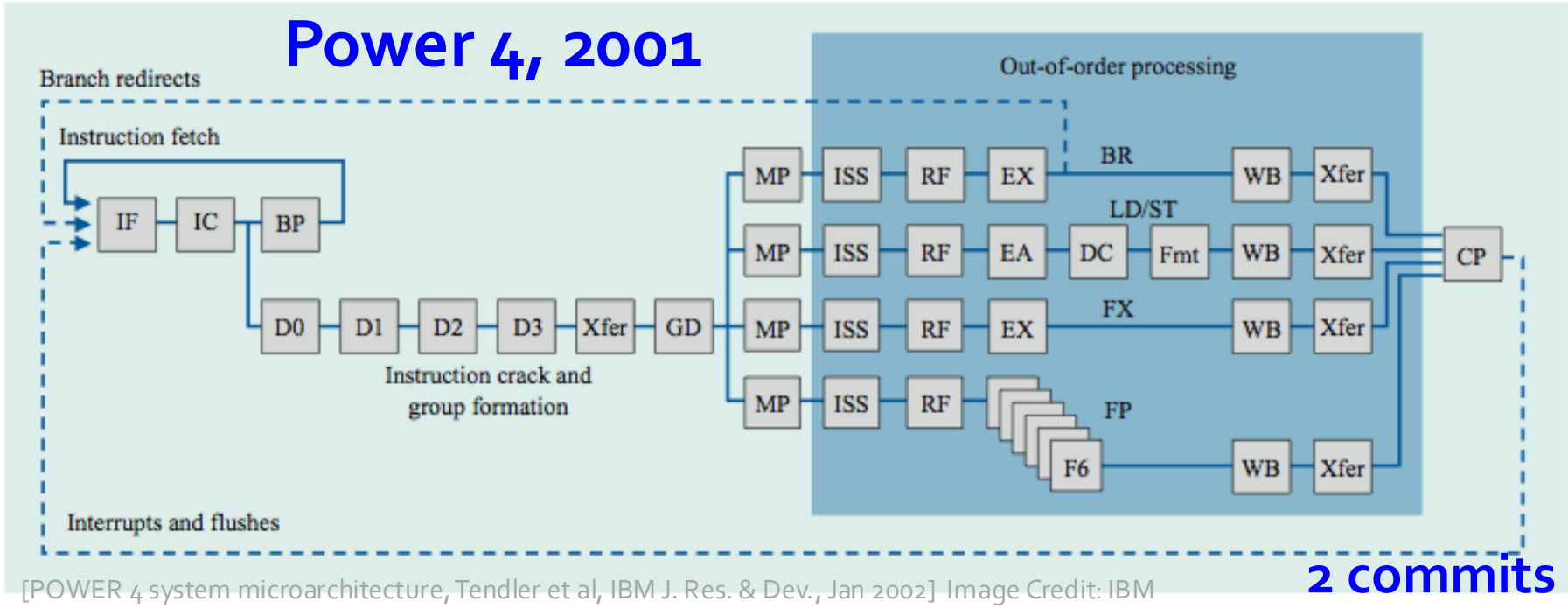


For regions with high thread level parallelism (TLP) entire machine width is shared by all threads



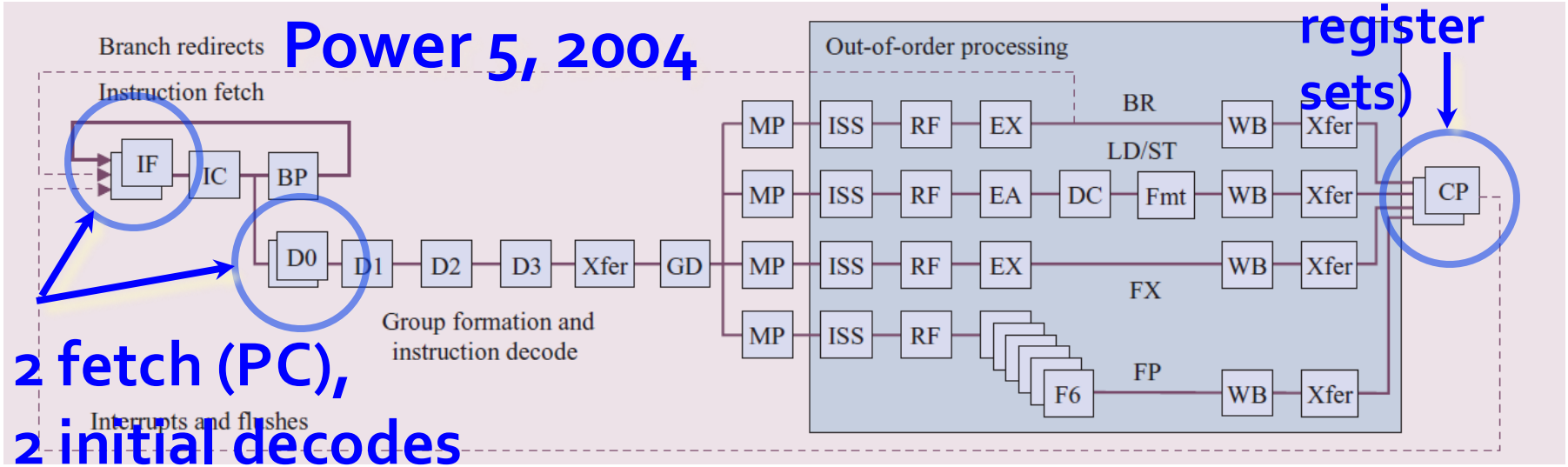
For regions with low thread level parallelism (TLP) entire machine width is available for instruction level parallelism (ILP)

Power 4, 2001



[POWER 4 system microarchitecture, Tendler et al, IBM J. Res. & Dev., Jan 2002] Image Credit: IBM
Courtesy of International Business Machines, © International Business Machines.

Power 5, 2004



2 fetch (PC), 2 initial decodes

2 commits (architected register sets)

[POWER 5 system microarchitecture, Sinharoy et al, IBM J. Res. & Dev., Jul/Sept 2005] Image Credit: IBM
Courtesy of International Business Machines, © International Business Machines.

Power 5 data flow …

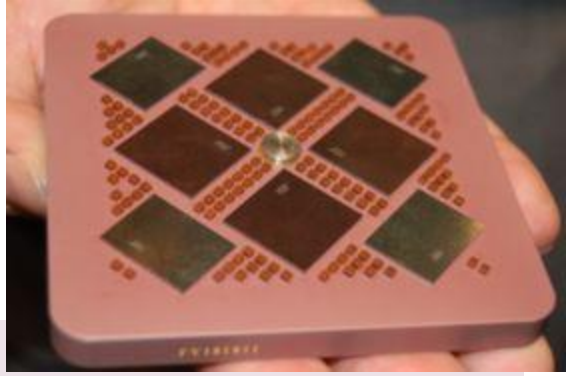
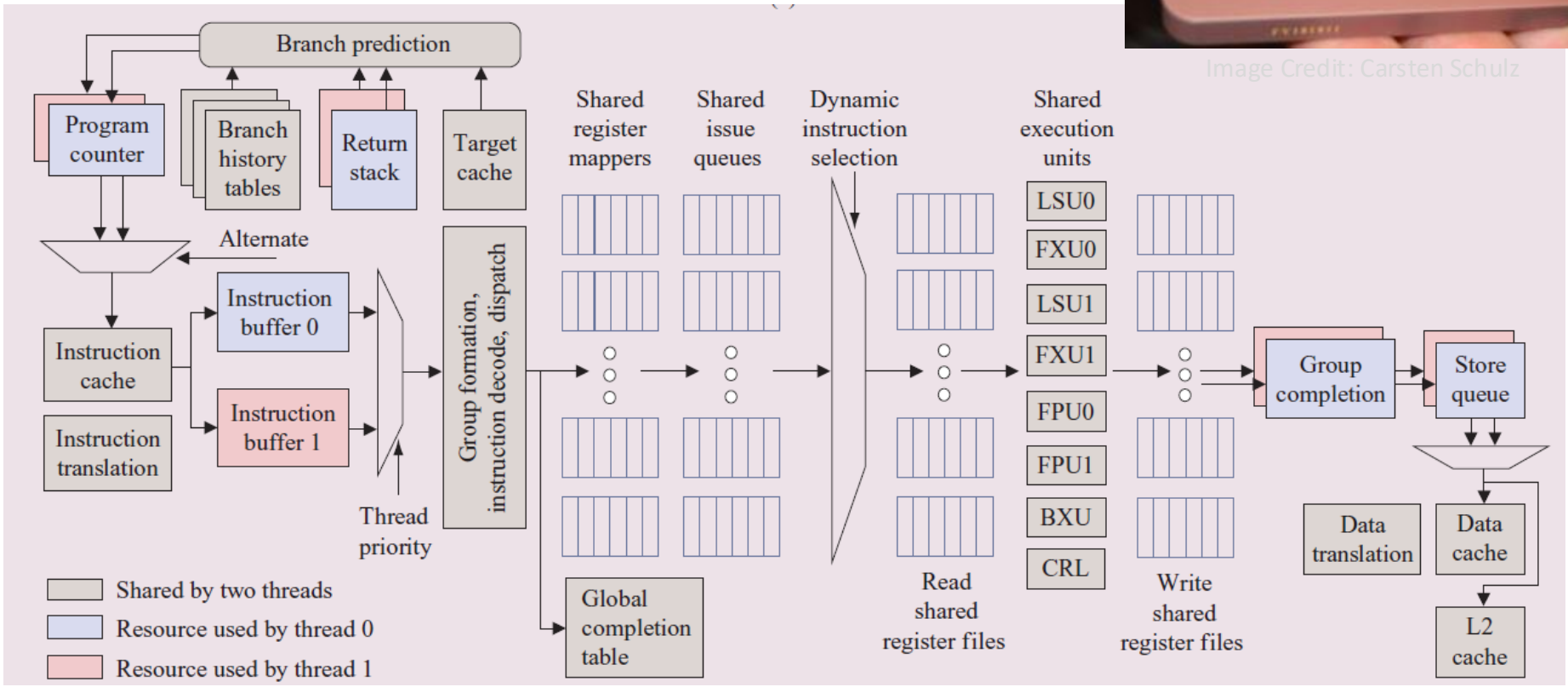


Image Credit: Carsten Schulz



[POWER 5 system microarchitecture, Sinharoy et al, IBM J. Res. & Dev., Jul/Sept 2005]
Image Credit: IBM Courtesy of International Business Machines, © International Business Machines.

- 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 and L3 caches
- Added separate instruction prefetch and buffering per thread
- Increased the number of physical 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

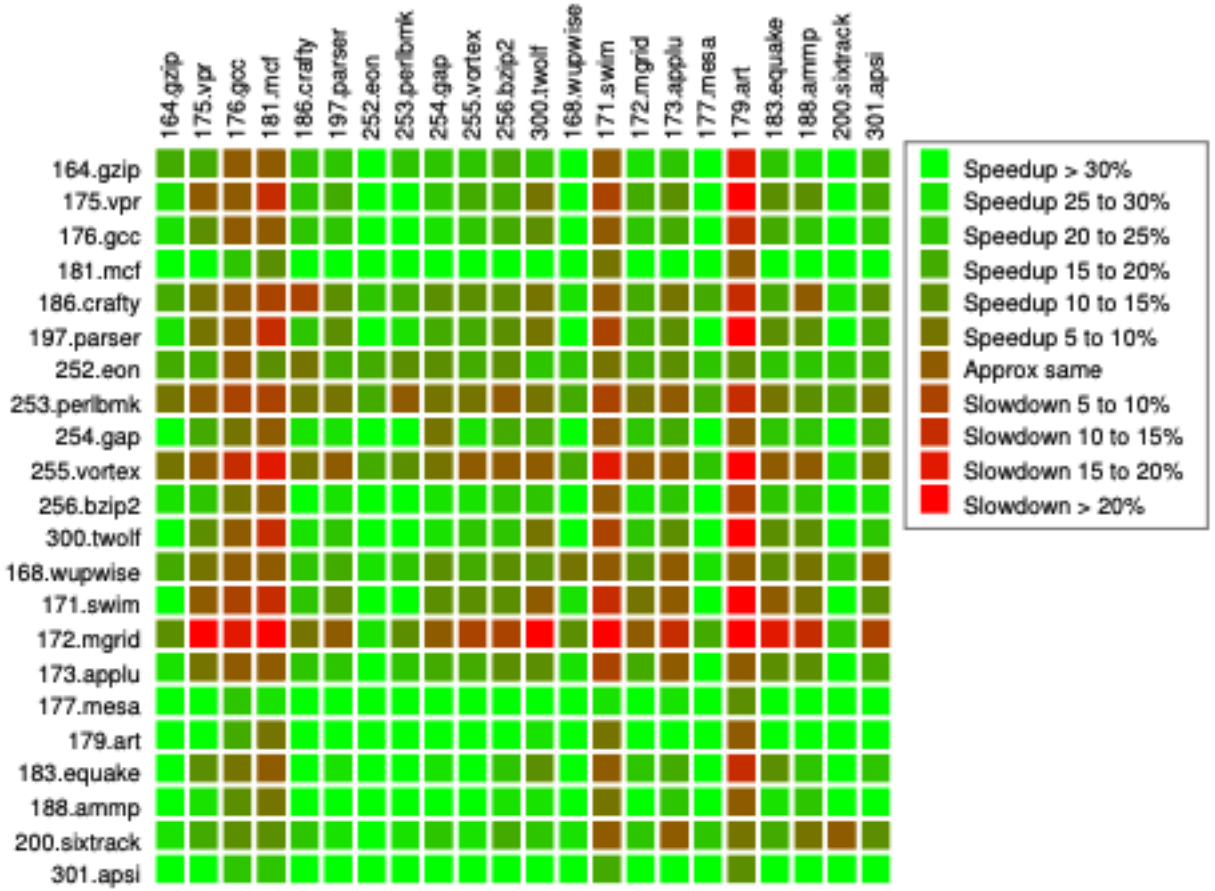
Pentium-4 Hyperthreading (2002)

- First commercial SMT design (2-way SMT)
- 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
- Claims 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.
- First Intel Atom (in-order x86 core) has two-way vertical multithreading
  - Hyperthreading == (SMT for Intel OoO & Vertical for Intel InO)

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
  - Floating Point apps had most cache conflicts and least gain

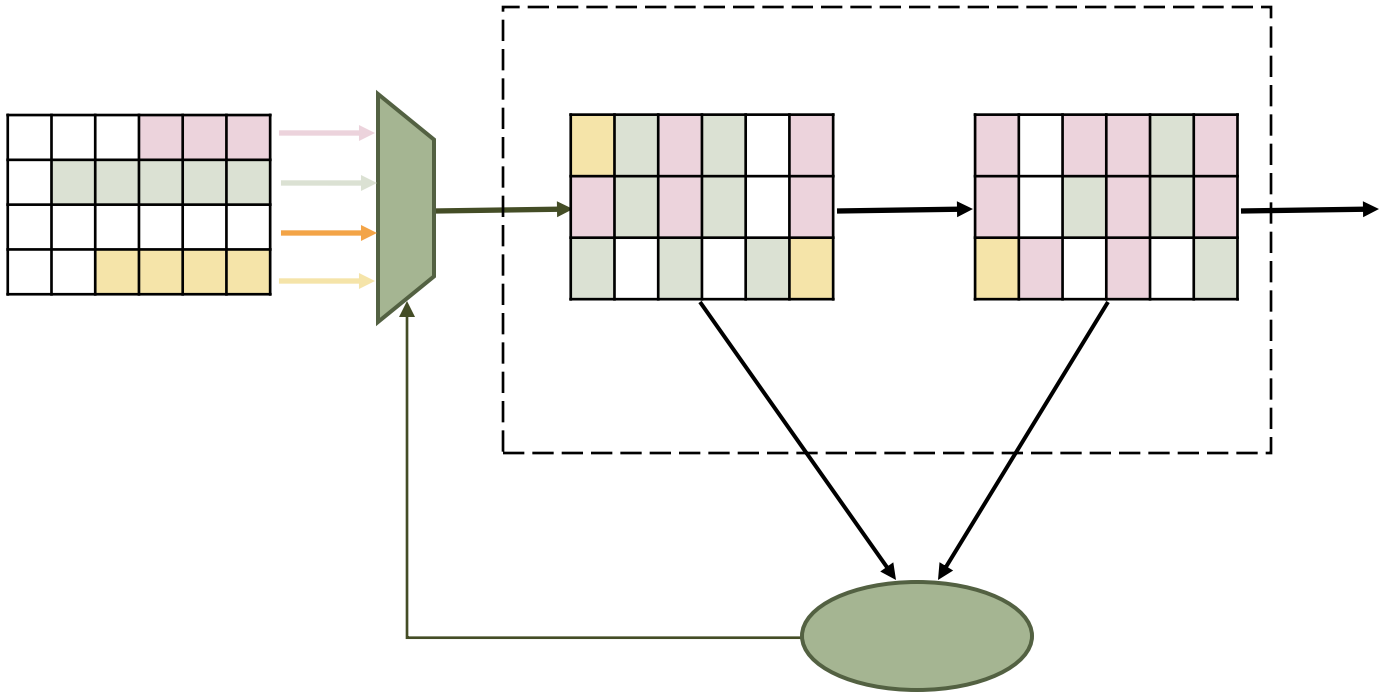
SMT Performance: Application Interaction



Bulpin et al, "Multiprogramming Performance of Pentium 4 with Hyper-Threading"

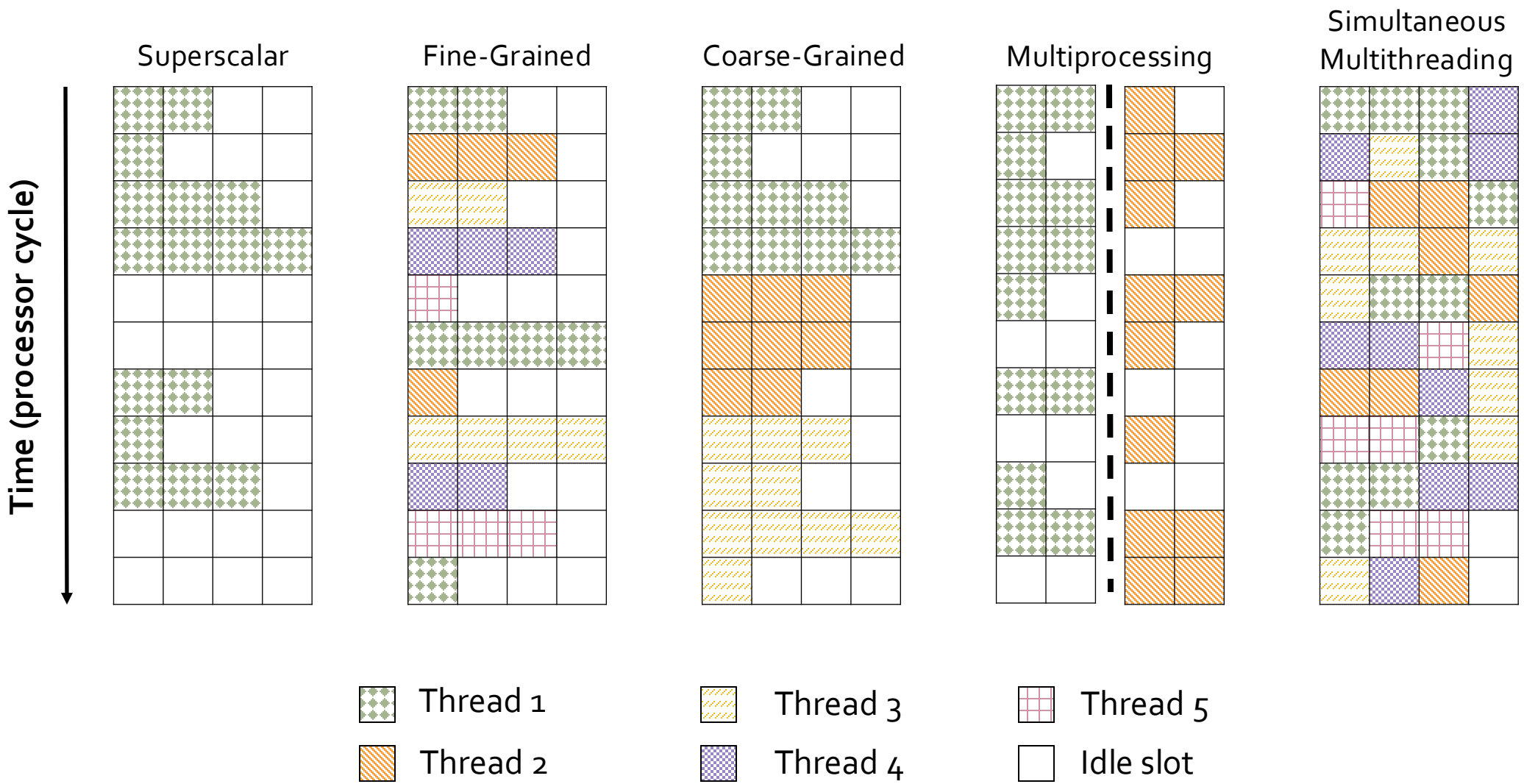
Icount Choosing Policy

- Fetch from thread with the least instructions in flight

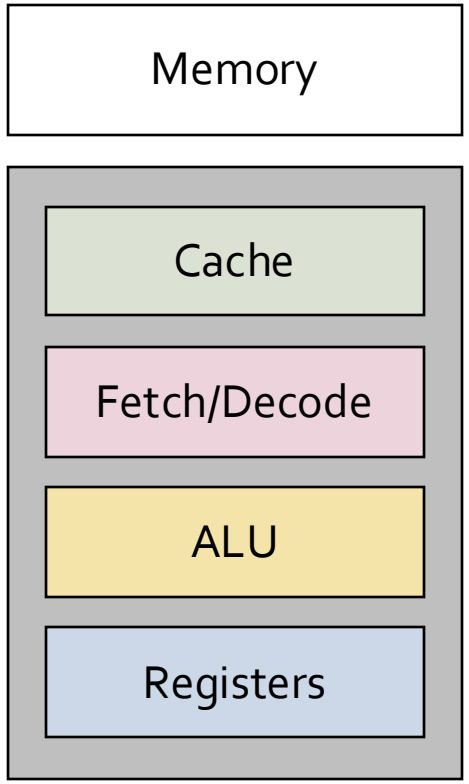


Why does this enhance throughput?

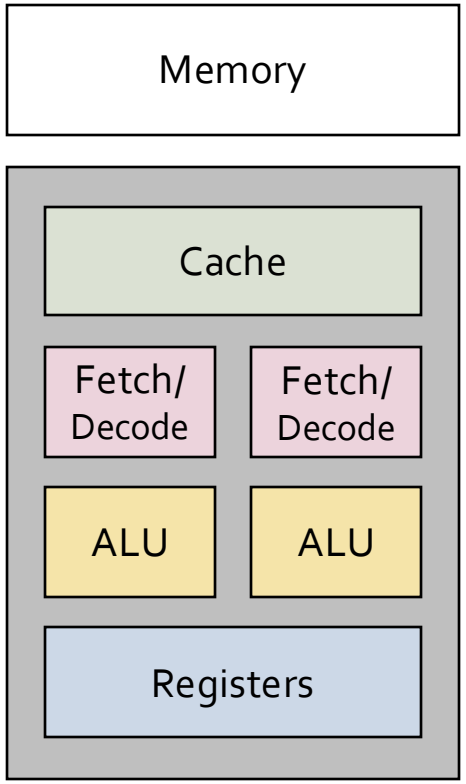
Summary: Multithreaded Categories



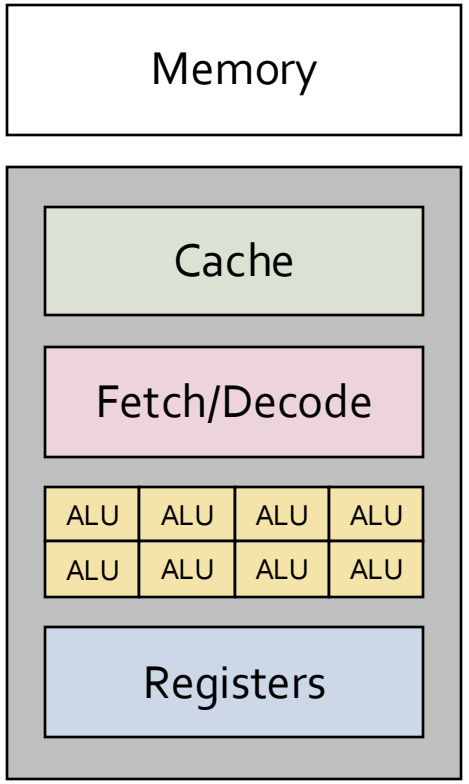
Simplified Processor Evolution



Single-core,
single-thread,
scalar processor

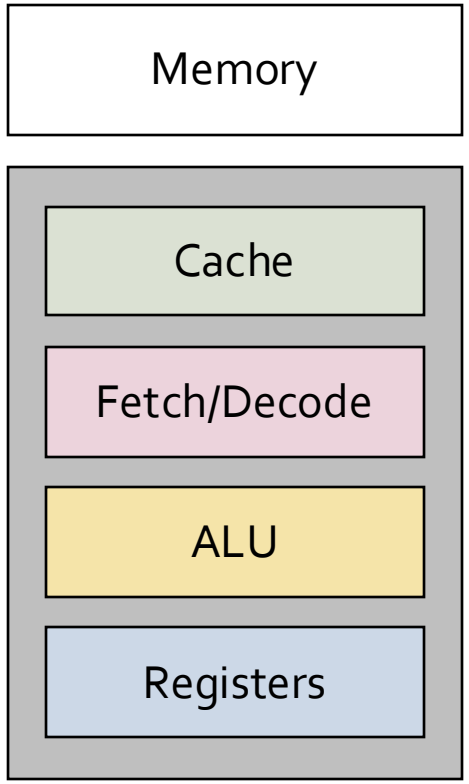


Single-core,
single-thread,
superscalar processor

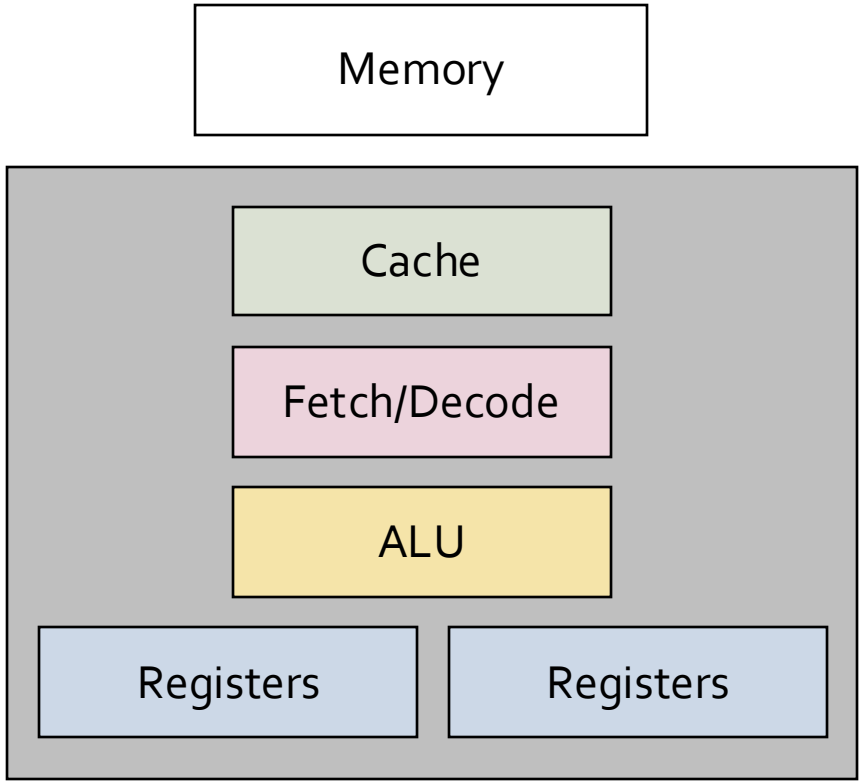


Single-core,
single-thread,
SIMD/vector processor

Simplified Processor Evolution

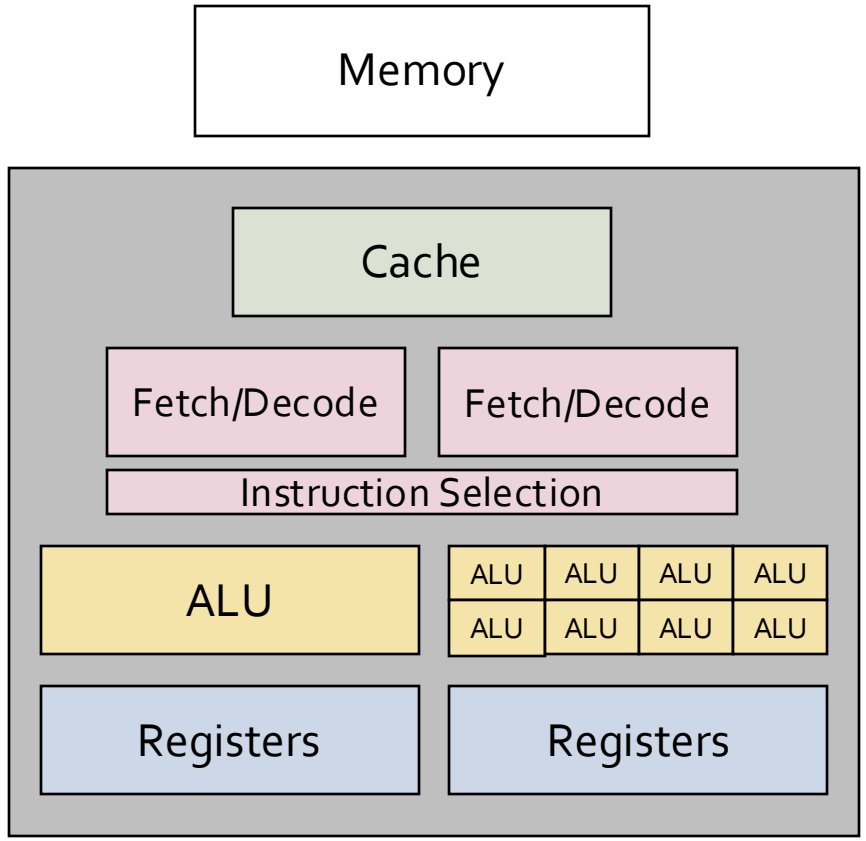


Single-core,
single-thread,
scalar processor

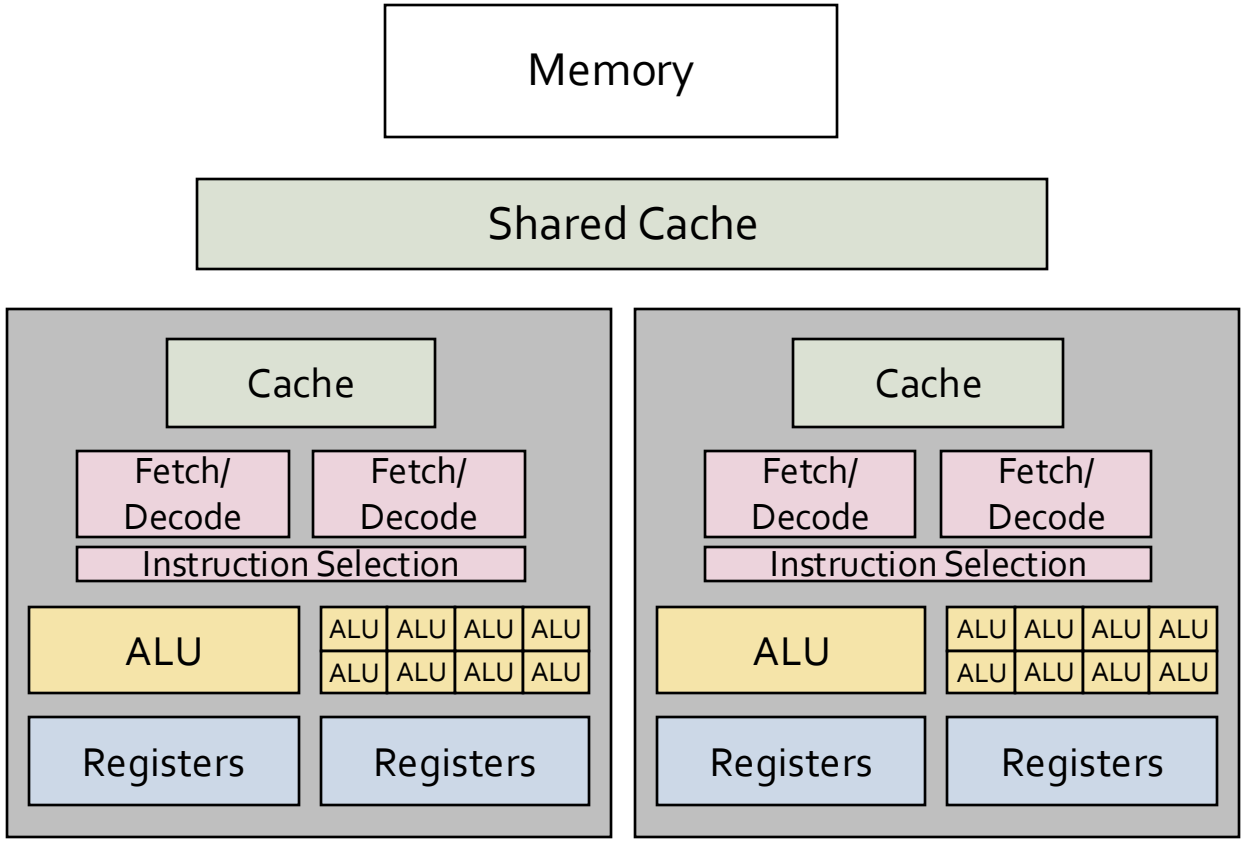


Single-core,
multi-thread,
scalar processor

Simplified Processor Evolution



Single-core,
multi-thread, superscalar,
OoO, SIMD
scalar processor



multi-core,
multi-thread, superscalar,
OoO, SIMD
scalar processor

Agenda

- Multithreading Motivation
- Coarse Grain Multithreading
- Simultaneous Multithreading
- Hardware Security

SMT & Security Risks

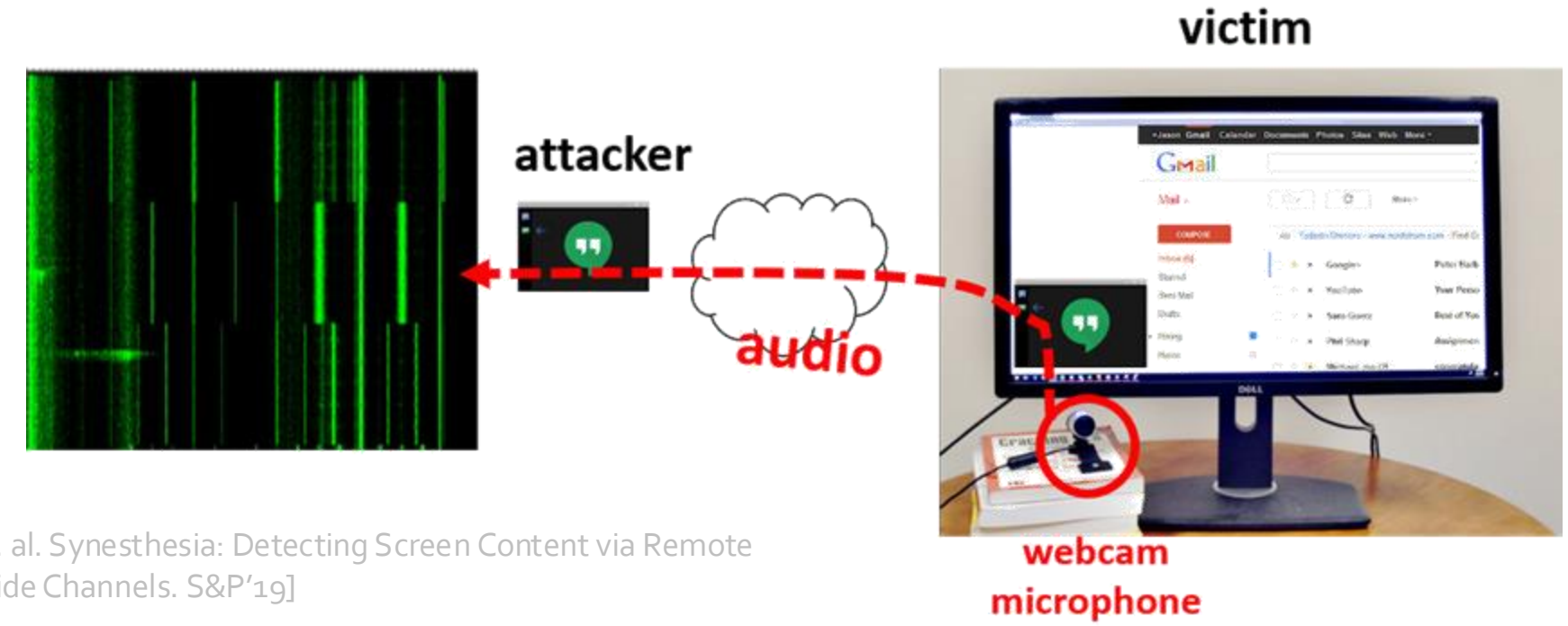
- Most hardware attacks rely on shared hardware resources to establish a side-channel
  - e.g., Shared outer caches, DRAM row buffers
- SMT gives attackers high-BW access to previously private hardware resources that are shared by co-resident threads:
  - TLBs: TLBleed (2018)
  - L1 caches: CacheBleed (2016)
  - Functional unit ports: PortSmash (2018)
- OpenBSD 6.4 → Disabled HT and SMT in BIOS for enhanced security against side-channel attacks.

Side/Covert Channel

- A side channel is a mechanism that leads to inadvertent information transfer
  - e.g., going through your neighbor's recycling
- A covert channel is a mechanism used for illicit communication between two cooperating parties
  - e.g., a double-agent spy communicating with their handler
- Both channels result in undesired information transfer
- Computer programs can have such channels
  - Even programs that utilize secrets, like encryption keys

Side Channels Are Almost Everywhere

- Monitor keystroke
  - You only need: a cheap microphone + an ML model
- "Hear" the screen



[Genkin et. al. Synesthesia: Detecting Screen Content via Remote Acoustic Side Channels. S&P'19]

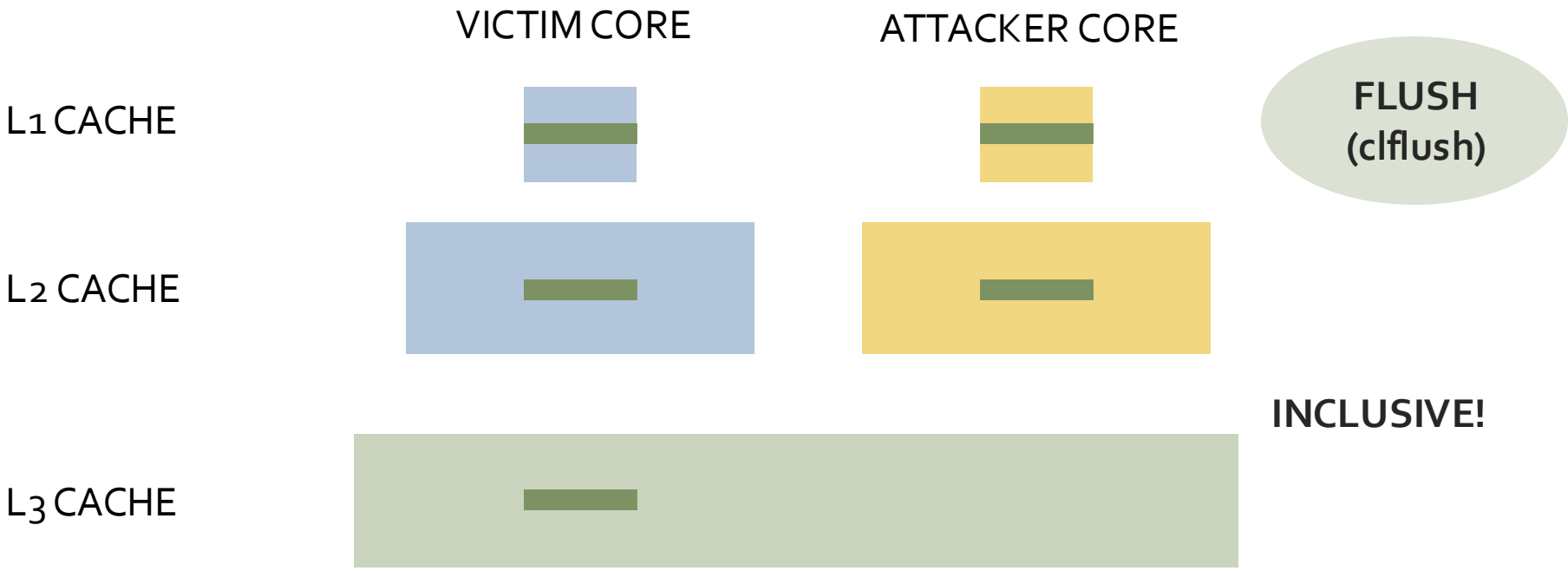
Cache Side Channel. Why Cache?

- Large attack surface. Shared across cores/sockets.
- Fast. Can be used to build high-bandwidth channels
- Many states. Can encode secrets spatially to further improve bandwidth and precision.
- There exist many cache-like structures. The same attack concepts and tricks will apply

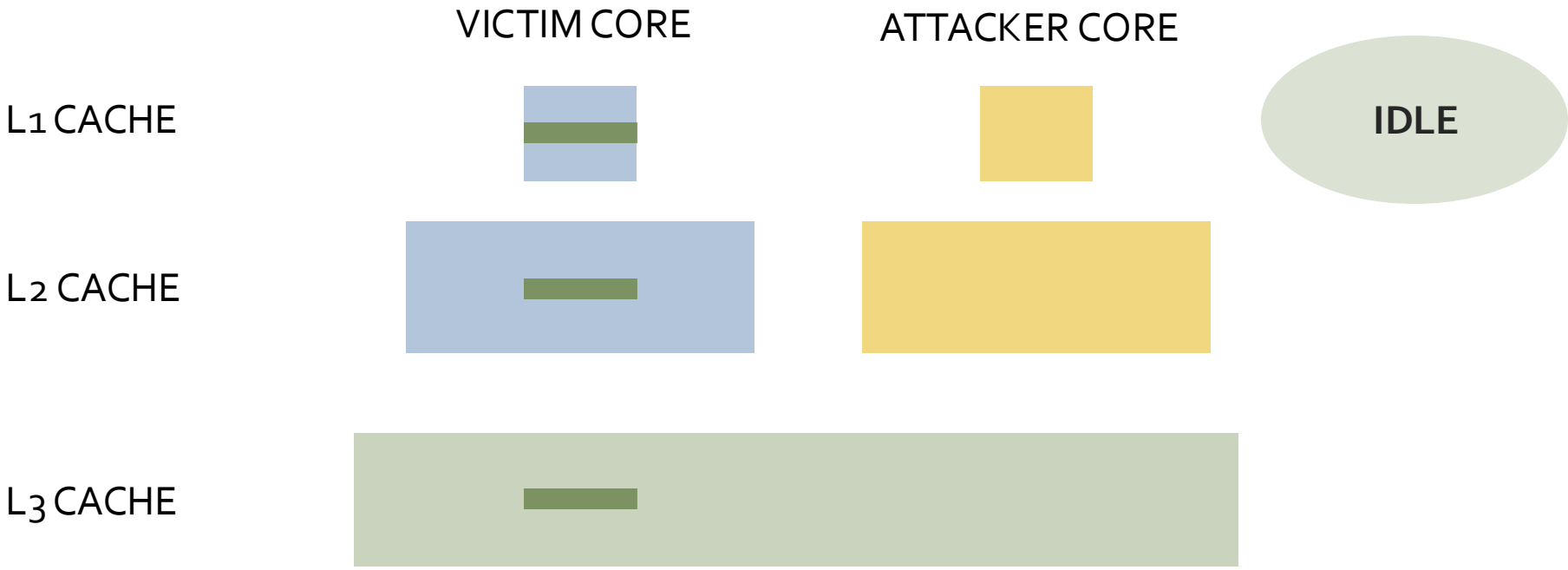
Attack Strategy: Flush+Reload

- The flush instructions allow explicit control of cache states
  - In X86, clflush vaddr
  - In ARM, DC CIVAC vaddr
- What are these flush instructions used for except for attacks?
  - For coherence, in the case when the data in the cache is inconsistent with the data in the DRAM.
  - 1) old time, incoherent DMA
  - 2) nowadays, Non-volatile memory for crash recovery

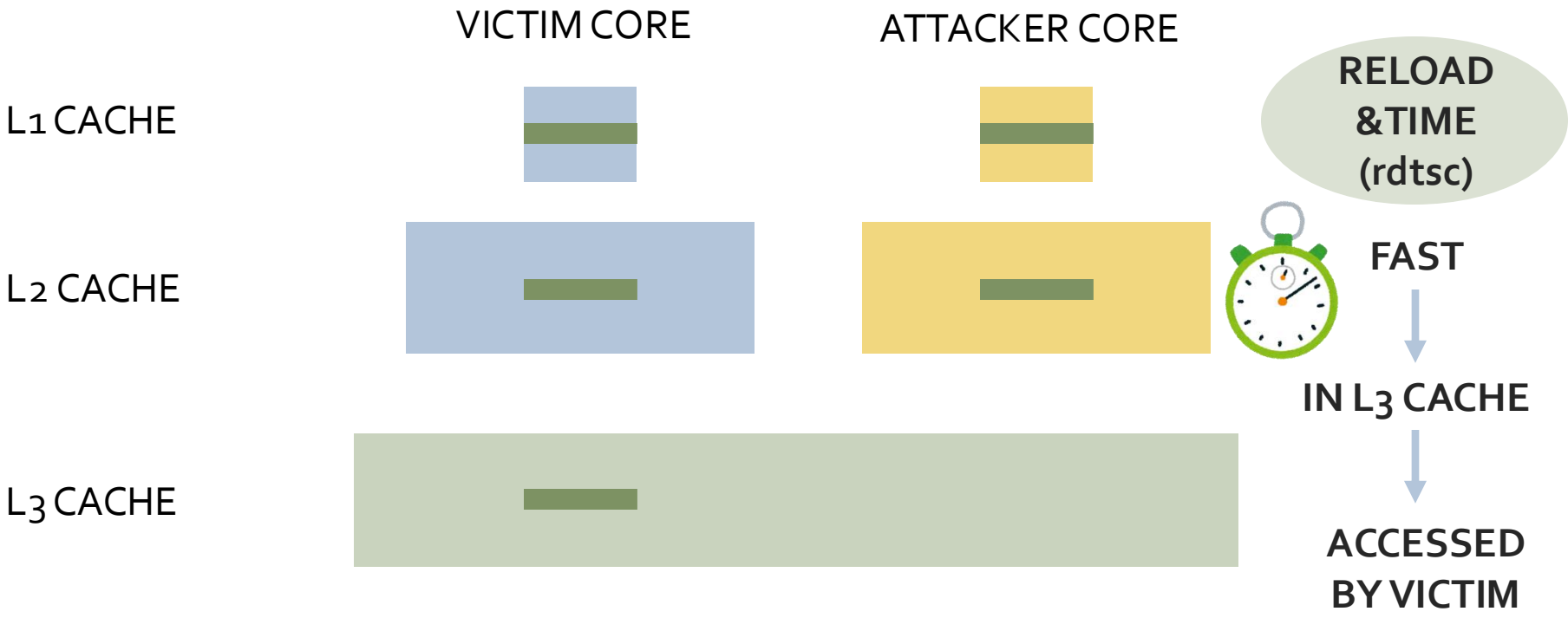
Flush-Reload Side-Channel Attack on x86



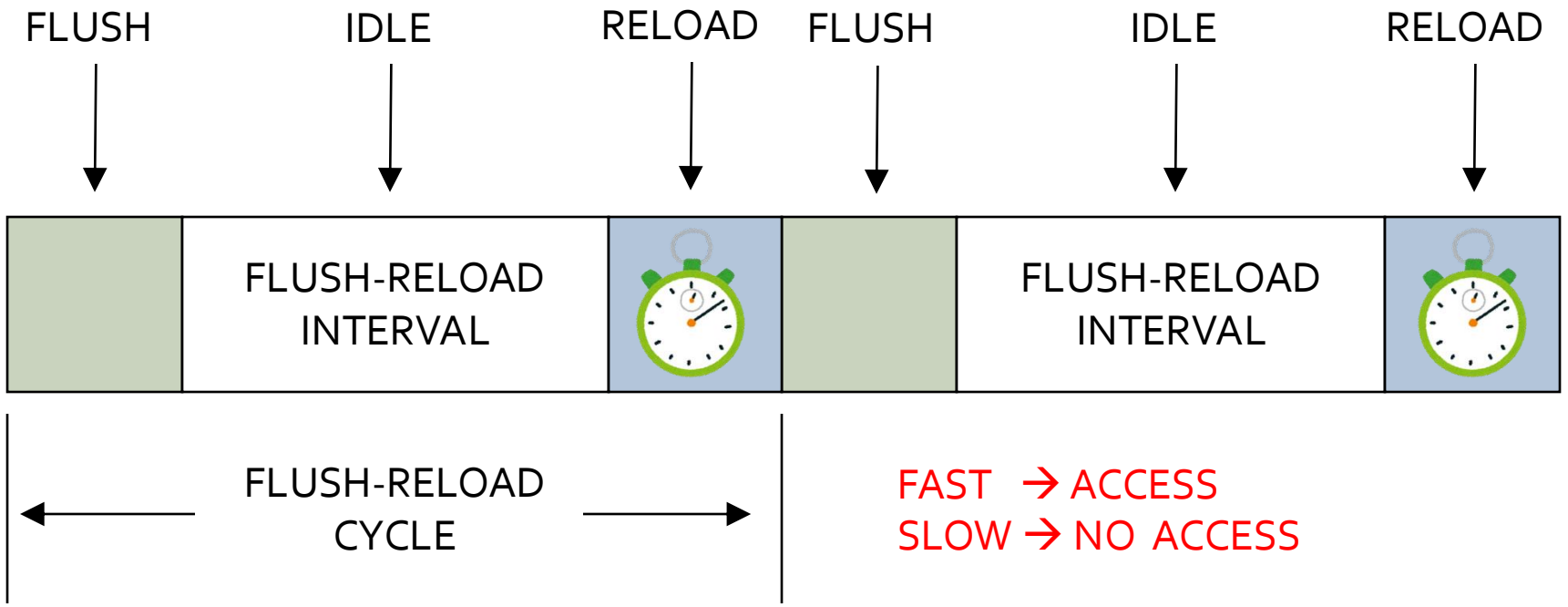
Flush-Reload Side-Channel Attack on x86



Flush-Reload Side-Channel Attack on x86



Flush-Reload Side-Channel Attack on x86



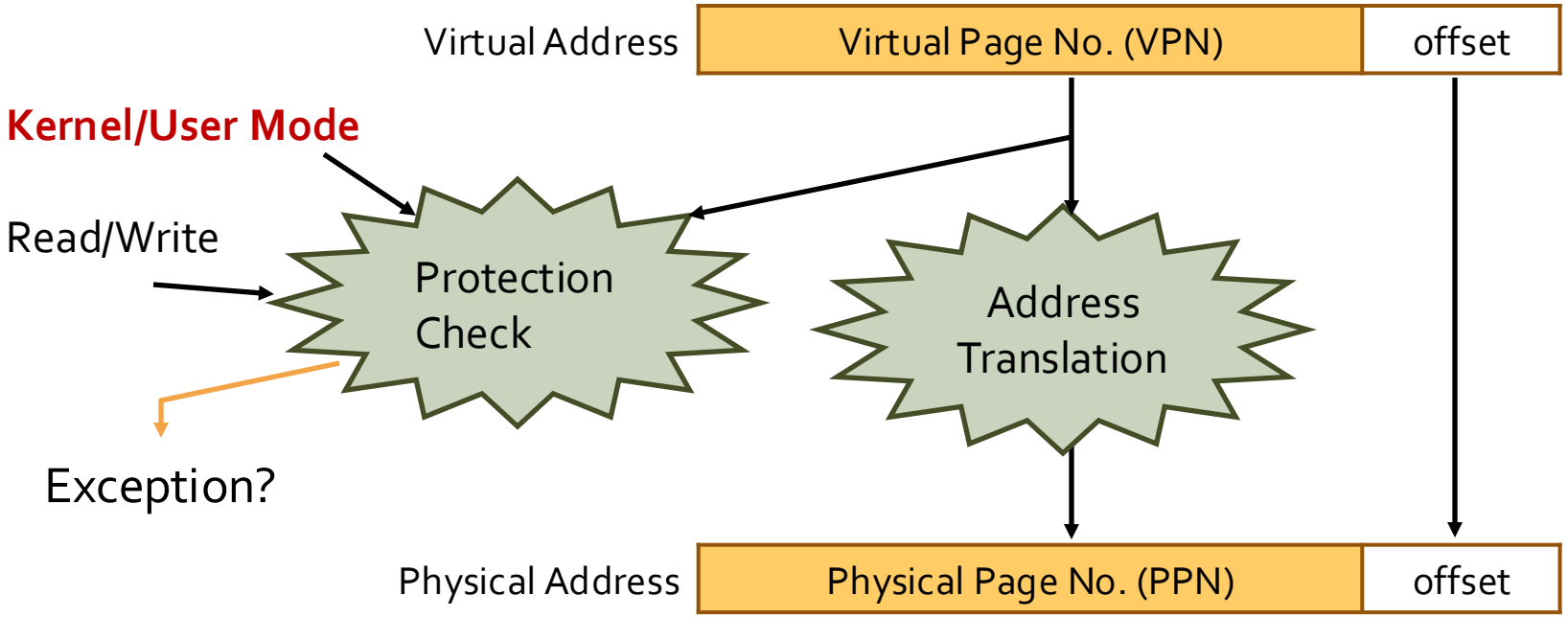
Recap

- Out-of-Order exceution
- Speculative execution
- Branch Prediction
- Hardware Caching

Meltdown

K	V	R	W	D	tag	PPN	PS	G	ASID

- Meltdown explores the combined effects of two optimizations
  - Hardware optimization: out-of-order execution
  - Software optimization: mapping kernel addresses into user space



Meltdown

Let's read kernel memory from user space 😎

r0, r1: temps

r2: kernel address we wish to read

r3: start of probe_array

```
ldr r0 <- [r2]
sll r0 <- r0, #6
add r1 <- r3, r0
ldr r1 <- [r1]
```

```
……
Ld1: uint8_t secret = *kernel_address;
Ld2: unit8_t dummy = probe_array[secret*64];
```

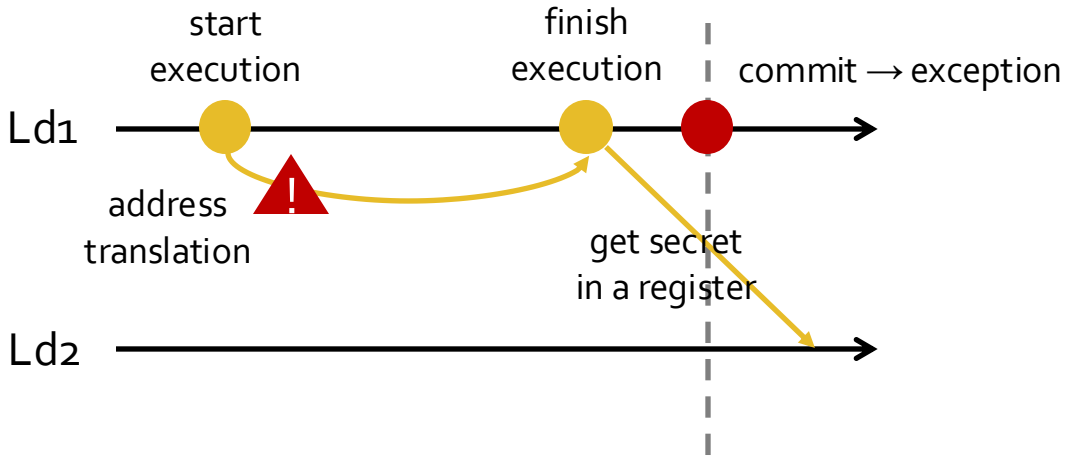
<= this raises an exception…but not until Commit!

After the execution, we clear the ROB and other OoO structures **but not the caches**, leaving open a side channel!

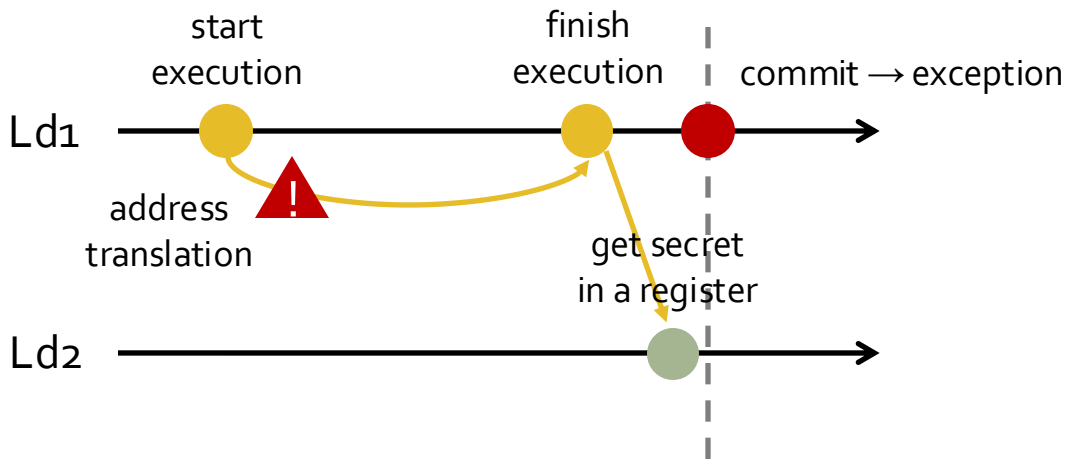
Meltdown Timing

```
……
Ld1: uint8_t secret = *kernel_address;
Ld2: unit8_t dummy = probe_array[secret*64];
```

- Case 1: Fail. Ld2 is squashed before the corresponding memory access is issued.



- Case 2: Attack works. Ld2's request is sent out before the instruction is squashed.



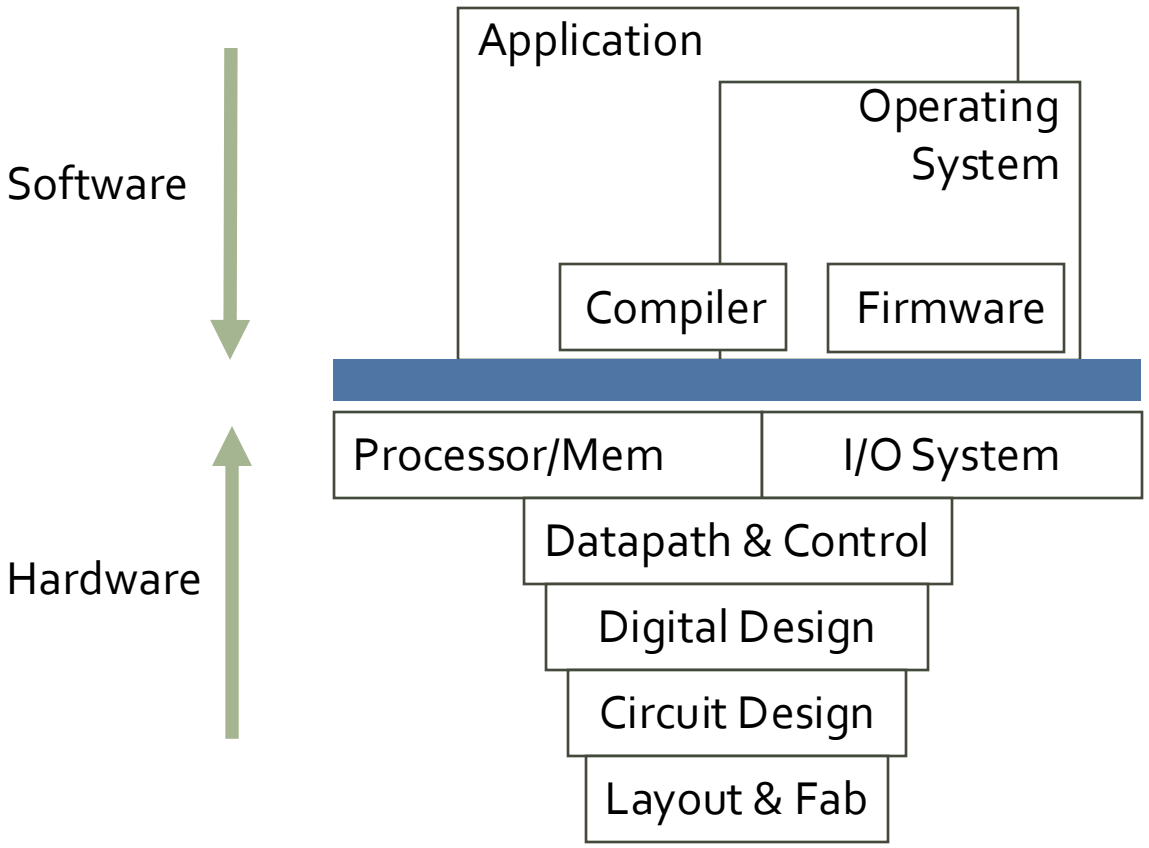
Meltdown w/ Flush+Reload

1. Attacker allocates probe_array, with 256 cache lines. Flushes all its cache lines
2. Attacker executes

```
……
Ld1: uint8_t secret = *kernel_address;
Ld2: unit8_t dummy = probe_array[secret*64];
```

3. After handling protection fault, attacker performs cache side channel attack to figure out which line of probe_array is accessed → recovers byte

Why It Took So Long for Meltdown to Be Discovered?



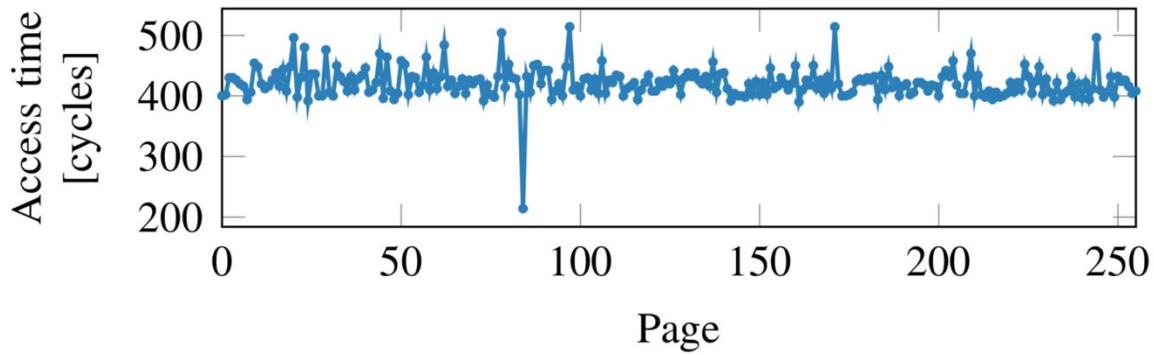
SW optimization: map kernel address in user space

Contract: Memory access goes through page permission check, and permission violation raises exceptions

HW optimization: speculation to delay exception handling

Speculative Execution Attacks

- An OoO processor speculates aggressively
  - on a mis-speculation, the incorrect instructions are wiped away and have no ISA-level visible impact
    - register writes, memory stores are all cleaned up
    - non-ISA level changes like cache contents **are not cleaned up**
- Real Meltdown uses pages instead of cache lines, exploiting a TLB side channel



Meltdown Mitigations

- Meltdown affects Intel OoO and some ARM OoO processors, AMD is immune
  - Intel did page permission checks at Commit, not Execute
  - Intel cores have hardware fixes as of ~2019
- Meltdown can be patched in software
  - all major OSes released patches in 2018
  - performance impact for system calls
  - KPTI/KAISER Linux patch maps minimal kernel code/data
- We generally consider Meltdown as a design bug

Spectre

- Goal: trick another process into leaking information
- Let's say my secure program contains this code:

```
if (x < array1_size) {
 y = array2[array1[x] * 256];
}
```

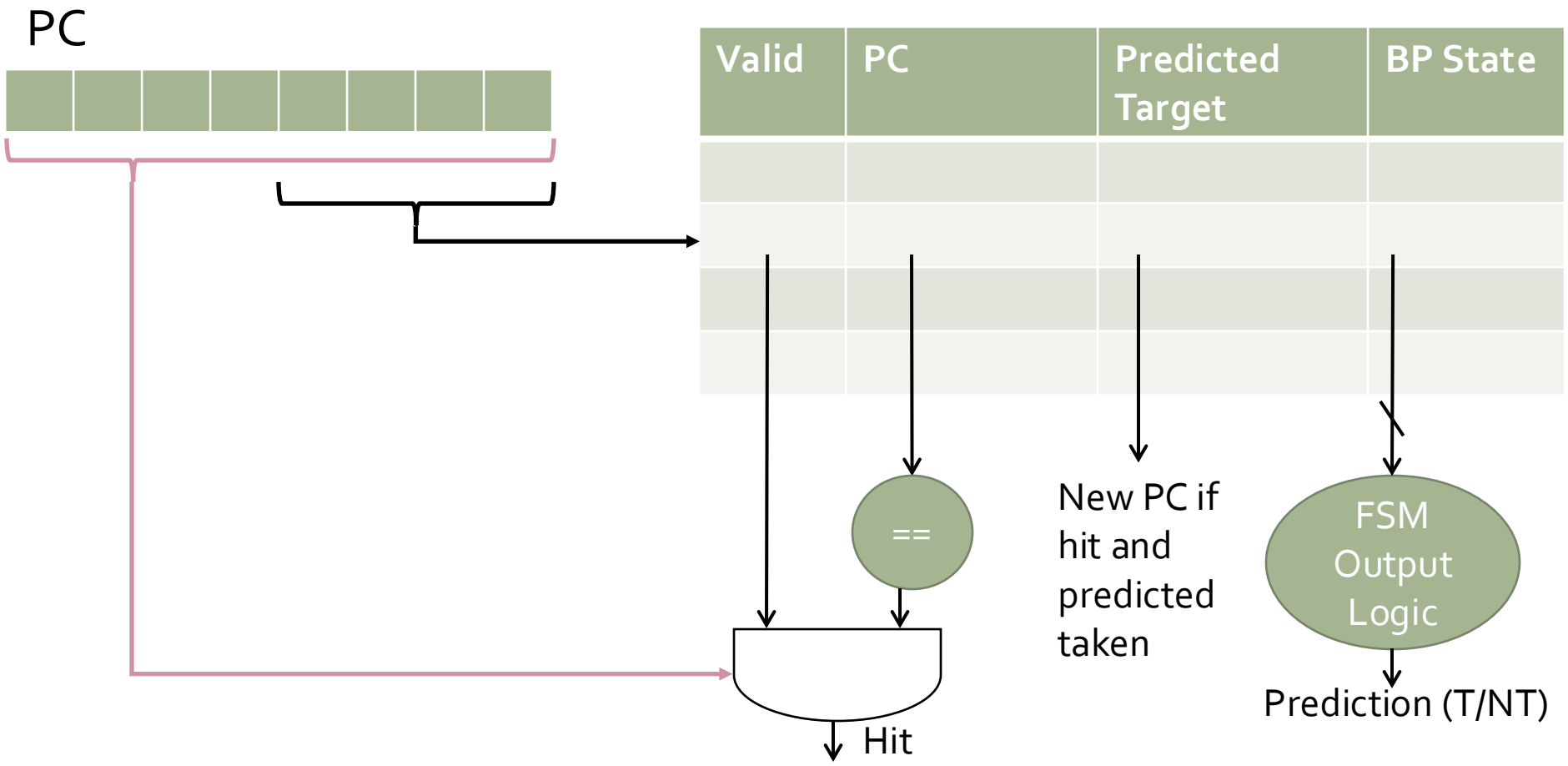
Always malicious?
No. It may be a benign misprediction.
We do not consider Spectre as a bug

- If attacker controls x, value of array1[x] can be leaked
  - problem: how to get around the bounds check?
  1. Train branch predictor
  2. Trigger branch misprediction; &array1[x] maps to some desired kernel address
  3. Attacker probes cache to infer which line of array2 was fetched

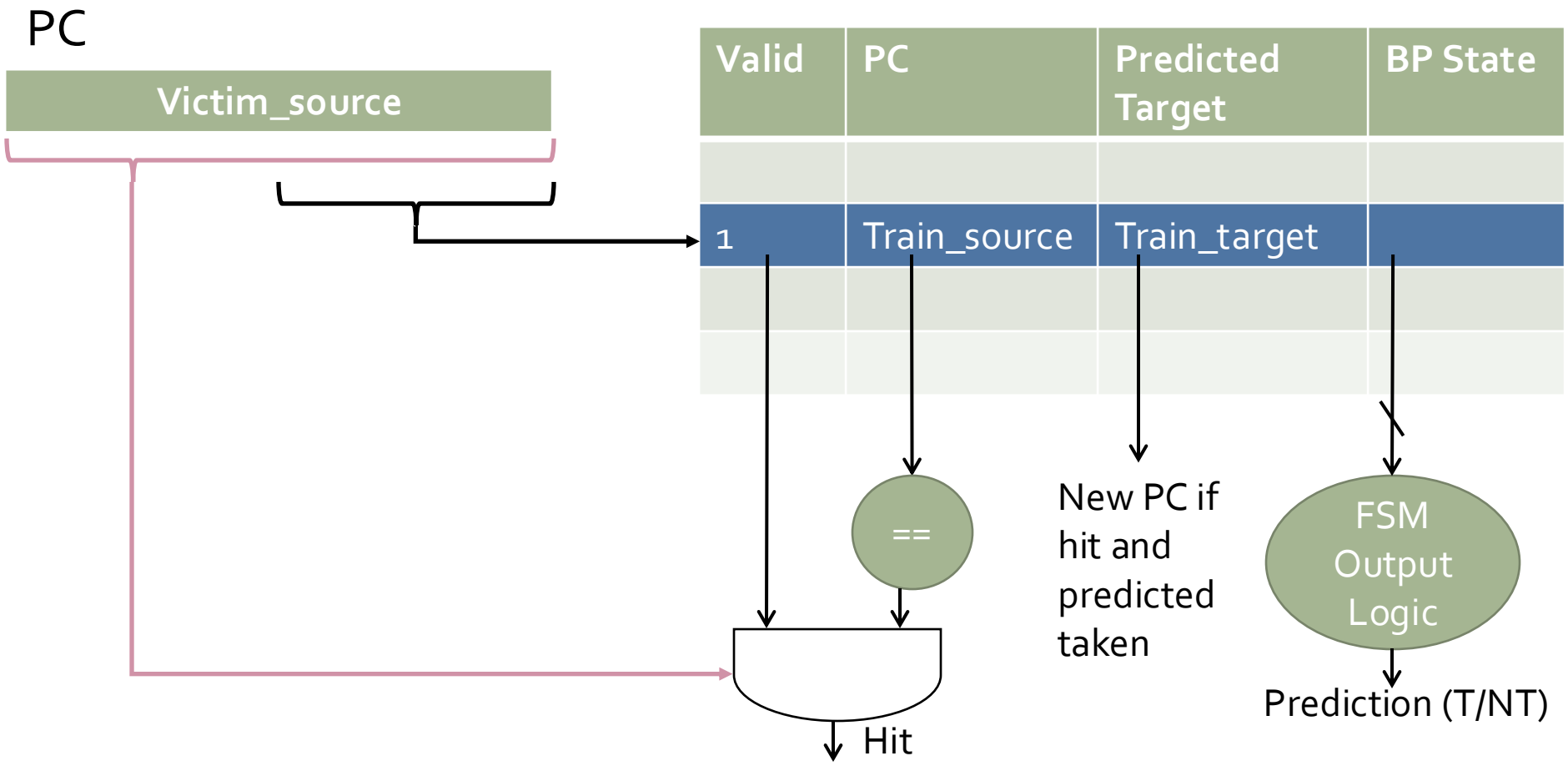
Poison the Branch Predictor

- How do I poison the BHT/BTB?
  - If I know the BHT/BTB entries that will be used for the bounds check branch, I can train them in advance
    - e.g., branch with the same PC in the attacker process
  - Not hard when the attacker provides their own code
    - JavaScript, VM on EC2
- How do I get the victim to run the vulnerable code?
  - highly victim-specific
  - easiest in JIT environments

Branch Target Buffer (BTB)



Branch Target Buffer (BTB)



Notes about Spectre

- What if accessed address is protected?
  - a[bad_illegal_x] might trigger exception!
  - Nope, because bad load never commits
  - It does trigger cache fill, however, which is the problem
- Variant is indirect branch exploitation
  - Train predictor to make speculative jump to bad code
  - Bad code alters microarchitecture state (e.g., cache)
  - Leak information via side channel

Spectre Mitigations

- Spectre affects OoO chips from everyone
  - confirmed on Intel, ARM, AMD
- Software patches are incomplete, slow
  - Reduce use of indirect branches
  - Disable speculation through special instructions at “critical” code points
- Ultimately, Spectre attacks aren’t that easy to pull off

These Attacks Break SW-HW Contract

- Software developer's problem
  - Software developers need to write software for devices with unknown design details.
  - How can I know whether the program is secure running on different devices?
- Hardware designer's problem
  - Hardware designers need to design processors for arbitrary programs.
  - How to describe what kind of programs can run securely on my device?

Need Architecture 2.0?

- Augment Architecture 1.0 with Architecture 2.0 specification of
  - (Abstration of) time visible micro-architecture
  - Bandwidth of known timing channels
  - Enforced limit on user software behavior
- Change micro-architecture to mitigate timing channel bandwidth
  - Suppress some speculation
  - Undo most changes on mis-peculation
- More generally, can we reduce dependence on speculation?
  - Accelerators!

Summary

- Multithreading Motivation
- Coarse Grain Multithreading
- Simultaneous Multithreading
- Hardware Security

Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)
  - Christopher Batten (Cornell)
  - David Wentzlaff (Princeton)
- MIT material derived from course 6.823
- UCB material derived from course CS252
- Cornell material derived from course ECE 4750
- Princeton material derived from course ECE 475