Exploiting modern microarchitectures: Meltdown, Spectre, and other attacks

Jon Masters, Computer Architect, Red Hat, Inc.

jcm@redhat.com | @jonmasters

MELTDOWN

SPECTRE

Overview

Today's lecture will cover the following:

- Introduction to microarchitecture as implementation of architecture
- In order vs. Out-of-Order execution in microarchitectures
- Caches, virtual memory, and side channel analysis
- Branch prediction and speculative execution
- Spectre and Meltdown vulnerabilities
- Mitigation approaches and solutions
- Related research into hardware

Architecture

Architecture

- An Instruction Set Architecture (ISA) describes the contract between hardware and software
  - Defines the instructions that all machines implementing the architecture must support
    - Load/Store from memory, architectural registers, stack, branches/control flow
    - Arithmetic, floating point, vector operations, and various possible extensions
  - Defines user (unprivileged, problem state) and supervisor (privileged) execution states
    - Exception levels used for software exceptions and hardware interrupts
    - Privileged registers used by the Operating System for system management
    - Mechanisms for application task context management and switching
  - Defines the memory model used by machines compliant with the ISA
- The lowest level targeted by an application programmer or (more often) compiler

Common concepts in modern architectures

- Application programs make use of a standard (non-privileged) set of ISA instructions
  - Programs (known as “processes” or “tasks” when running) execute in a lower privilege state
  - These are often referred to as “rings”, “exception levels”, etc.
- Application programs execute using a virtual memory environment
  - Virtual memory is divided into 4K (or larger) “pages”, the smallest unit at which it is managed
  - The processor Memory Management Unit (MMU) translates all memory accesses using page tables
  - The Operating System provides the illusion of a flat large address space by managing page tables
- Application programs request runtime services from the Operating System using system calls
  - The Operating System provided system calls run in the same virtual memory environment
    - e.g. Linux maps all of physical memory beginning at the high end of every process
  - Page table protections (normally) prevent applications from seeing this OS memory

Common concepts in modern architectures

- Operating System software makes use of additional privileged set of ISA instructions
    - These include instructions to manage application context (registers, MMU state, etc.)
    - e.g. on x86 this includes being able to set the CR3 (page table base) control register that hardware uses to automatically translate virtual addresses into physical memory addresses
- Operating System software is responsible for switching between applications
    - Save the process state (including registers), update the control registers
- Operating System software maintains application page tables
    - The hardware triggers a “page fault” whenever a virtual address is inaccessible
    - This could be because an application has been partially “swapped” (paged) out to disk, is being demand loaded, or because the application does not have permission to access that address

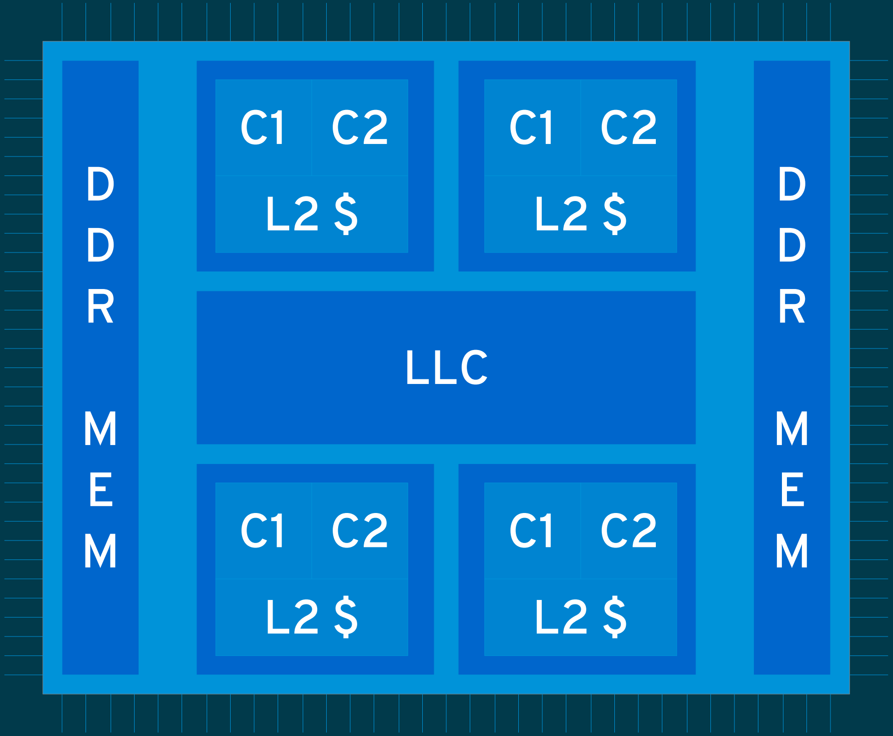
Examples of computer architectures

- Intel “x86” (Intel x64/AMD64)
  - CISC (Complex Instruction Set Computer)
  - Variable width instructions (up to 15 bytes)
  - 16 GPRs (General Purpose Registers)
  - Can operate directly on memory
  - 64-bit flat virtual address space
    - “Canonical” 48/56-bit addressing
    - Upper half kernel, Lower half user
    - Removal of older segmentation registers (except FS/GS)

- ARM ARMv8 (AArch64)
  - RISC (Reduced Instruction Set Computer)
  - Fixed width instructions (4 bytes fixed)
    - Clean uniform decode table
  - 32 GPRs (General Purpose Registers)
  - Classical RISC load/store using registers for all operations (first load from memory)
  - 64-bit flat virtual address space
    - Split into lower and upper halves

Microarchitecture

Elements of a modern System-on-Chip (SoC)



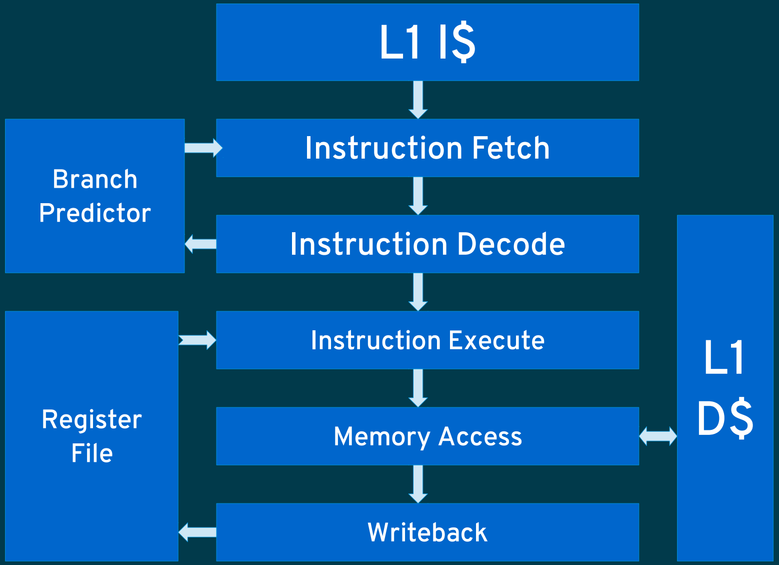
Elements of a modern System-on-Chip (SoC)

- Programmers often think in terms of “processors” by which they usually mean “cores”
  - Some cores are “multi-threaded” (SMT) sharing execution resources between two threads
  - Minimal context separation is maintained through some (lightweight) duplication
- Many cores are integrated into today's processor packages (SoCs)
  - These are connected using interconnect(ion) networks and cache coherency protocols
  - Provides a hardware managed coherent view of system memory shared between cores
- Memory controllers handle load/store of program instructions and data to/from RAM
  - Manage scheduling of DDR (or other memory) and sometimes leverage hardware access hints
- Cache hierarchy sits between external (slow) RAM and (much faster) processor cores
  - Progressively tighter coupling from LLC (L3) through to L1 running at core speed

Microarchitecture

- The term microarchitecture (“uarch”) refers to a specific implementation of an architecture
  - Compatible with the architecture defined ISA at a programmer visible level
  - Implies various design choices about the SoC platform upon which the core uarch relies
- Cores may be simpler “in-order” (similar to the classical 5-stage RISC pipeline)
  - Common in embedded microprocessors and those targeting low power points
  - Many Open Source processor designs leverage this design philosophy
- Cores may be “out-of-order” similar to a dataflow machine inside
  - Programmer sees (implicitly assumed) sequential program order
  - Core uses an dataflow model with dynamic data dependency tracking
  - Results complete (retire) in-order to preserve sequential model

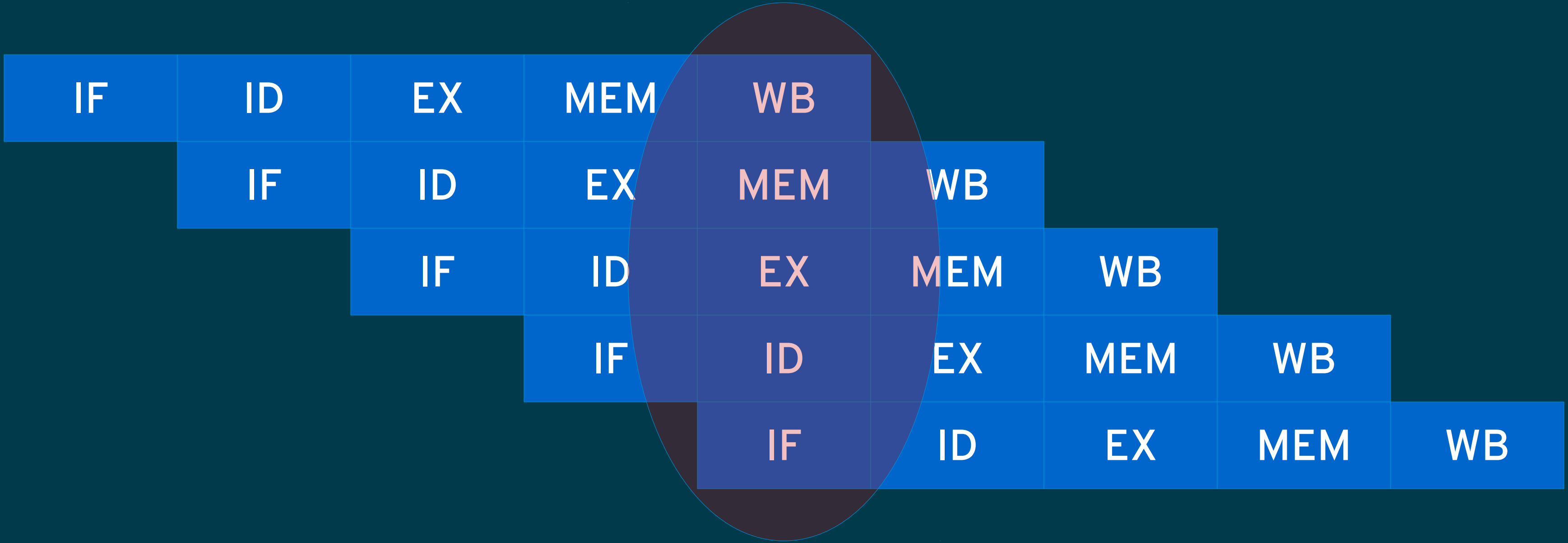
Elements of a modern in-order core



In order microarchitectures

- This is the classical “RISC” pipeline often taught first in computer architecture courses
  - Pipelining means instruction processing is split into multiple clock cycles
  - Multiple instructions may be at different “stages” in the pipeline simultaneously
- 1. Instructions are fetched from a dedicated L1 Instruction Cache (I$)
  - L1 cache automatically fills cache lines from “unified” L2/LLC on demand
- 2. Instructions are then decoded according to the ISA defined set of “encodings”
  - e.g. “add r3, r1, r2”
- 3. Instructions are executed by the execution units
- 4. Memory access is performed to/from the dedicated L1 Data Cache (D$)
- 5. The architectural register file is updated
  - e.g. r3 becomes the result of r1 + r2

An in-order pipeline visualized

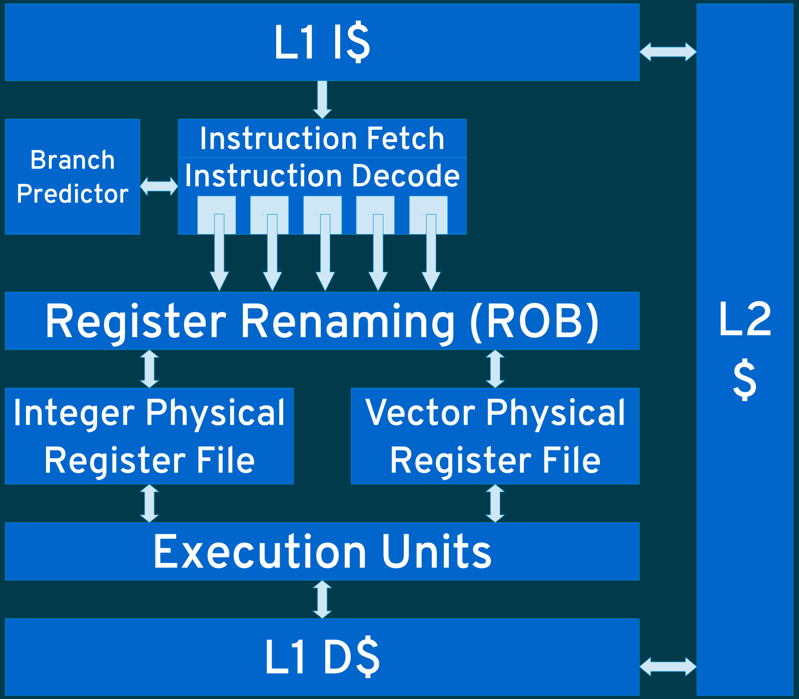


IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB

In order microarchitectures (continued)

- An in-order machine can suffer from pipeline stalls when stages are not ready
- The memory access stage may be able to load from the L1 D$ in a single cycle
- But if it is not in the L1 D$ then we insert a pipeline “bubble” while we wait for the data
  - This may take many additional cycles while the data is fetched from further away
- Limited capability to hide latency of instructions
  - Future instructions may not be dependent upon stalling earlier instructions

Elements of a modern out-of-order core



Out-of-Order (OoO) microarchitectures

- This type of design is common in aggressive high performance microprocessors
  - Also known as “dynamic execution” because it can change at runtime
  - Invented by Robert Tomasulo (used in System/360 Model 91 Floating Point Unit)
- Instructions are fetched and decoded by an in-order “front end” similar to before
- Instructions are dispatched to an out-of-order “backend”
  - Allocated an entry in a ROB (Re-Order Buffer), Reservation Stations
  - May use a Re-Order Buffer and separate Retirement (Architectural) Register File or single physical register file and a Register Alias Table (RAT)
- Re-Order Buffer defines an execution window of out-of-order processing
  - These can be quite large – over 200 entries in contemporary designs

Out-of-Order (OoO) microarchitectures (cont.)

- Instructions wait only until their dependencies are available
    - Later instructions may execute prior to earlier instructions
    - Re-Order Buffer allows for more physical registers than defined by the ISA
    - Removes some so-called data “hazards”
        - WAR (Write-After-Read) and WAW (Write-After-Write)
- Instructions complete (“retire”) in-order
    - When an instruction is the oldest in the machine, it is “retired”
    - State becomes architecturally visible (updates the architectural register file)

Microarchitecture (continued)

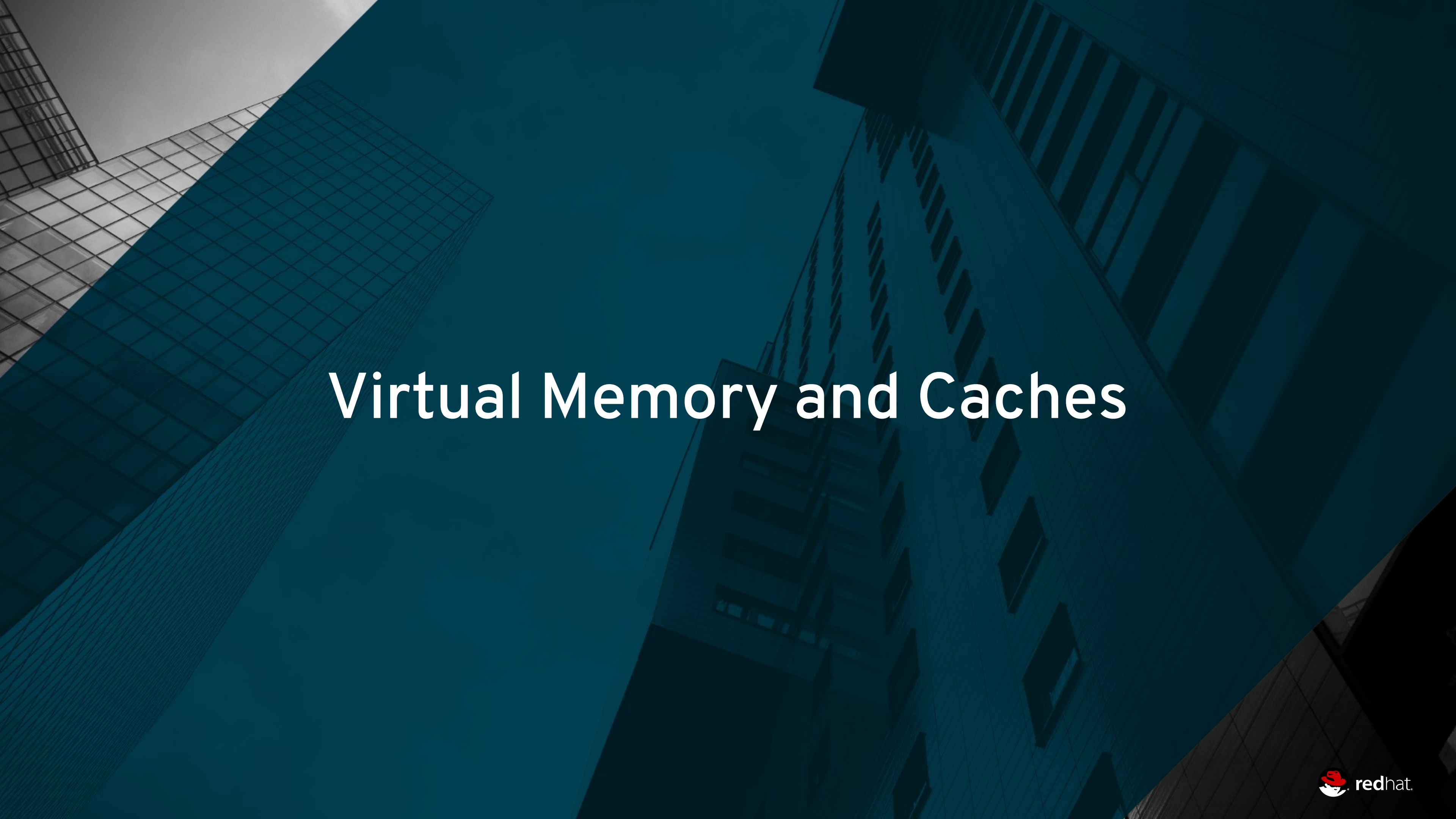
- The term microarchitecture (“uarch”) refers to a specific implementation of an architecture
  - Implies various design choices about the SoC platform upon which the core uarch relies
- Questions we can ask about a given implementation include the following:
  - What's the design point for an implementation – Power vs Performance vs Area (cost)
    - Low power simple in-order design vs Fully Out-of-Order high performance design
  - How are individual instructions implemented? How many cycles do they take?
    - How many pipelines are there? Which instructions can issue to a given pipe?
  - How many microarchitectural registers are implemented? How many ports in the register file?
    - How big is the Re-Order Buffer (ROB) and the execution window?

Examples of computer microarchitectures

- Intel Core i7-6560U (“Skylake” uarch)
  - 2 SMT threads per core (configurable)
  - 32KB L1I$, 32KB L1D$, 256KB L2$
  - 4-8* uops instruction issue per cycle
  - 8 execution ports (14-19 stage pipeline)
  - 224 entry ROB (Re-Order Buffer)
  - 14nm FinFET with 13 metal layers

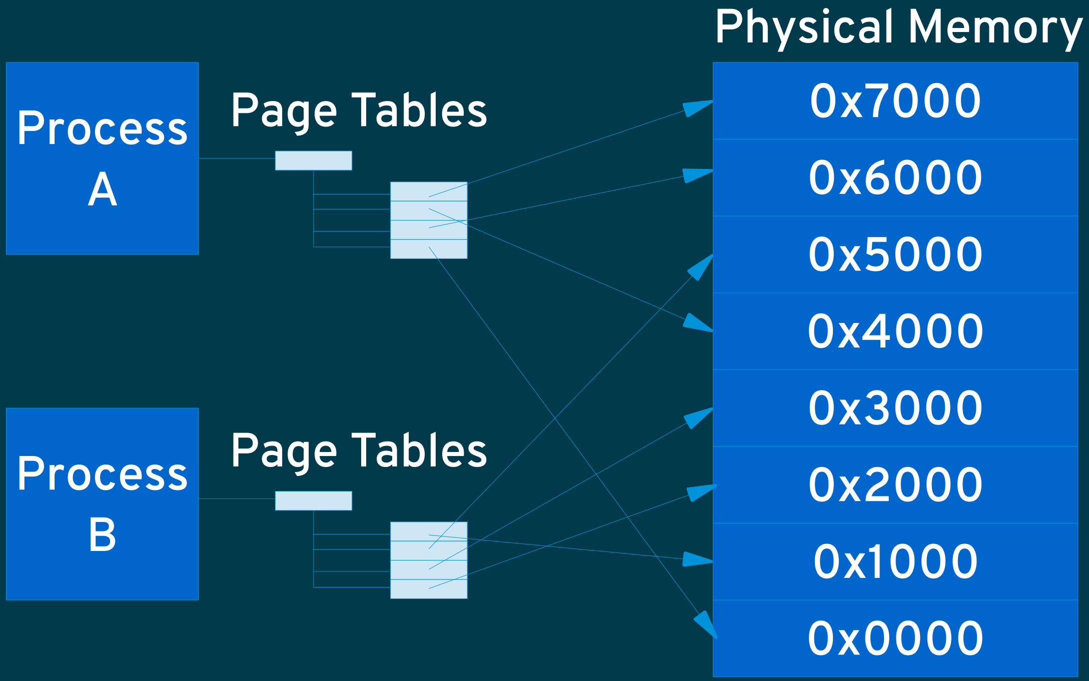
* Typical is 4uops with rate exception

- IBM POWER8E (POWER8 uarch)
  - Up to 8 SMT threads per core (configurable)
  - 32KB L1I$, 64KB L1D$, 512KB L2$
  - 8-10 wide instruction issue per cycle
  - 16 execution piplines (15-23 stage pipeline)
  - 224 entry Global Completion Table (GCT)
  - 22nm SOI with 15 metal layers



Virtual Memory and Caches

Virtual memory



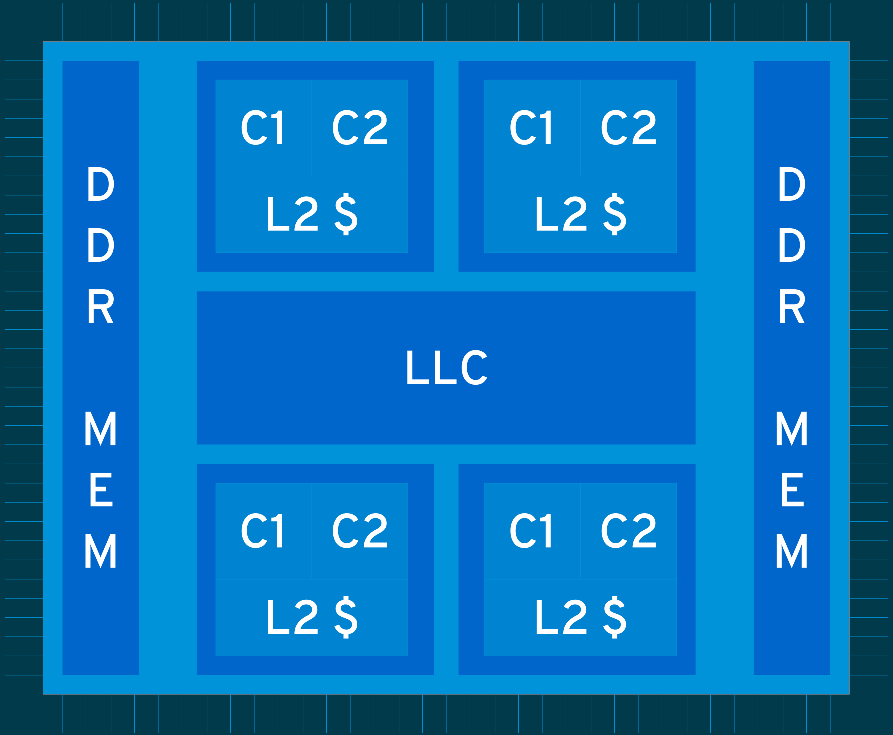
Virtual memory

- Memory accesses are translated (possibly multiple times) before reaching memory
  - Applications use virtual addresses (VAs) that are managed at page-sized granularity
  - A VA may be mapped to an intermediate address if a Hypervisor is in use
  - Either the Hypervisor or Operating System kernel manages physical translations
- Translations use hardware-assisted page table walkers that traverse page tables
  - The Operating System creates and manages the page tables for each application
  - Hardware manages TLBs (Translation Lookaside Buffers) filled with recent translations
- The collection of currently valid addresses is known as a (virtual) address space
  - On “context switch” from one process to another, page table base pointers are swapped, and existing TLB entries are invalidated. Cache flushing may be required depending upon the use of address space IDs (ASIDs, PCIDs, etc.) in the architecture and the Operating System

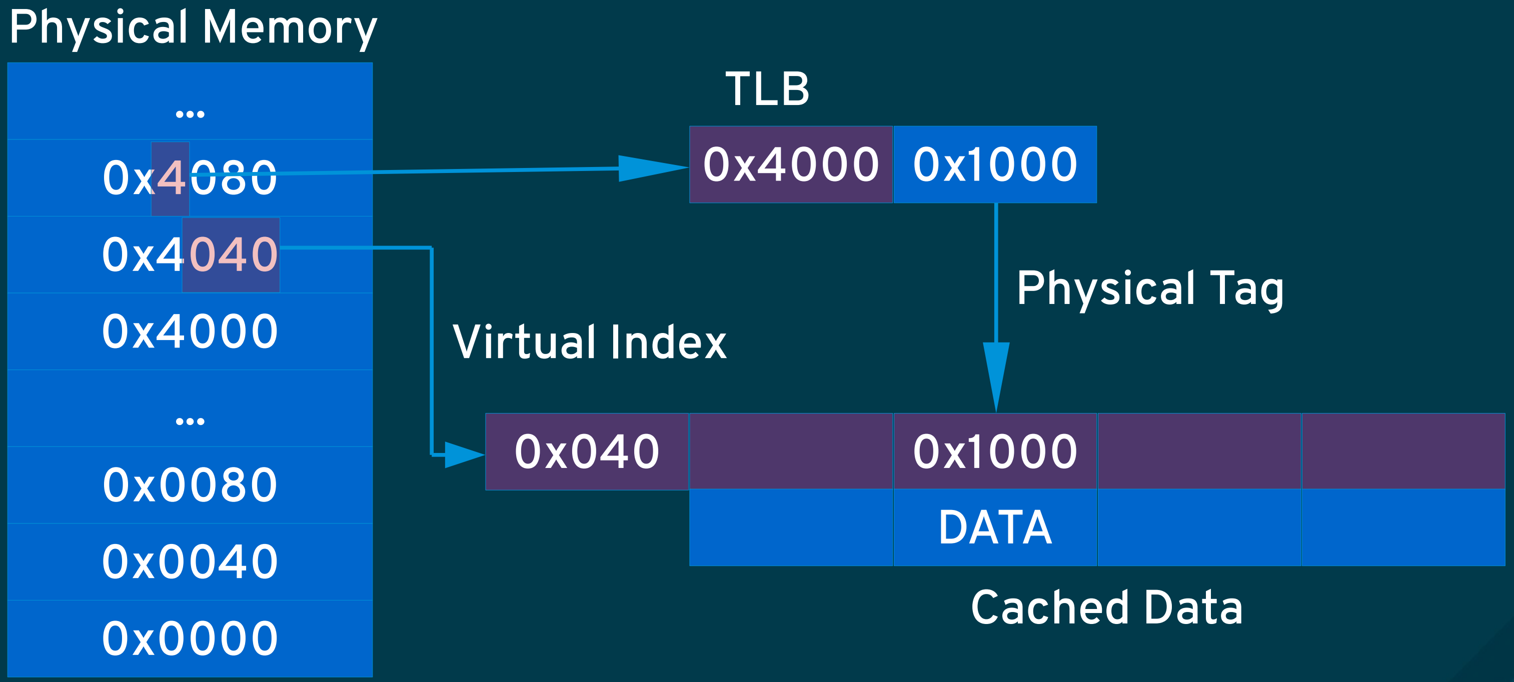
Virtual memory

- Applications have a large flat Virtual Address space mostly to themselves
  - Text (code) and data are dynamically linked into the virtual address space at application load automatically using metadata from the ELF (Executable Linkng Format) application binary
  - Dynamic libraries are mapped into the Virtual Address space and may be shared by applications
- Operating Systems may map some OS kernel data into application virtual address space
  - Limited examples intended for deliberate use by applications (e.g. Linux VDSO) for performance
    - Data can be directly read from the Virtual Dynamic Shared Object without a system call
  - The rest is explicitly protected by marking it as inaccessible in the application page tables
- Linux (used to) maps all of physical memory into every running application process
  - Allows for system calls into the OS without performing a full context switch on entry
  - The kernel is linked with high virtual addresses and mapped into every process

Caches



Caches


Physical Memory
...
0x4080
0x4040
0x4000
...
0x0080
0x0040
0x0000
TLB
0x4000
0x1000
Physical Tag
Virtual Index
0x040
0x1000
DATA
Cached Data

Caches

- Caches exist because the principal of locality says recently used data is likely to be used again
- Unfortunately we have a choice between “small and fast” and “large and slow”
    - Levels of cache provide the best of both, replacement policies handle cache eviction
- Caches are organized into sets where each set can contain multiple cache lines
    - A typical cache line is 64 or 128 bytes and represents a block of memory
    - A typical memory block will map to single cache set, but can be in any “way” of a set
- Caches may be direct mapped or (fully) associative depending upon complexity
    - Direct mapped allows one memory location to exist only in a specific cache location
    - Associative caches allow one memory location to map to one of N cache locations

Caches

- Cache entries are located using a combination of indexes and tags
  - Index and tag pages are formed from a given address address
  - The index locates the set that may contain blocks for an address
  - Each entry of the set is checked using the tag for an address match
- Caches may use virtual or physical memory addresses, or a combination
  - Fully virtual caches can result in homonyms for identical physical addresses
  - Fully physical caches can be much slower as they must use translated addresses
- A common optimization is to use VIPT (Virtually Indexed, Physically Tagged)
  - VIPT caches search index using the low order (page offset, e.g. 12) bits of a VA
  - Meanwhile the core finds the PA from the MMU/TLB and supplies to tag compare

Caches as side channels

- Caches exist fundamentally because they provide faster access to frequently used data
  - The closer data is to the compute cores, the less time is required to load it when needed
- This difference in access time for a given address can be measured by software
  - Data closer to the cores will take fewer cycles to access
  - Data further away from the cores will take more cycles to access
- Consequently it is possible to determine whether a specific address is in the cache
  - Calibrate by measuring access time for known cached/not cached data
  - Time access to a memory location and compare with calibration

Caches as side channels

- Consequently it is possible to determine whether a specific address is in the cache
    - Calibrate by measuring access time for known cached/not cached data
    - Time access to a memory location and compare with calibration

```
time = rdtsc();
maccess(&data[0x300]);
delta3 = rdtsc() - time;

time = rdtsc();
maccess(&data[0x200]);
delta2 = rdtsc() - time;
```

Caches as side channels (continued)

- Many instruction sets provide convenient high resolution cycle-accurate timers
  - e.g. x86 provides RDTSC (Read Time Stamp Counter) and RDTSCP instructions
- But there are other ways to measure cycles for architectures without an unprivileged TSC
- Some instruction sets (e.g. x86) also provide convenient unprivileged cache flush instructions
  - CLFLUSH guarantees that a given (virtual) address is not present in any level of cache
- But possible to also flush using a “displacement” approach on other arches
  - Create data structure the size of cache and access entry mapping to desired cache line
- On x86 the time for a flush is proportionate to whether the data was in the cache
  - flush+flush attack determines whether an entry was cached without doing a load
  - Harder to detect using CPU performance counter hardware (measuring cache misses)

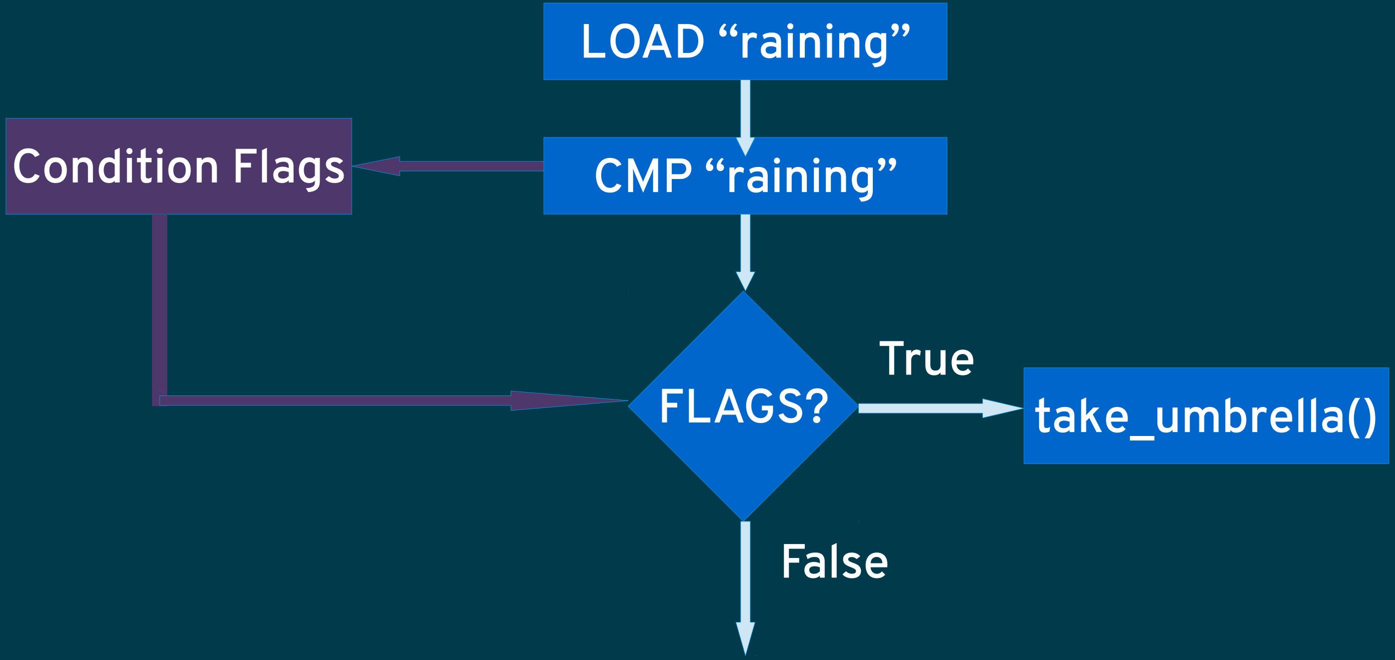
Caches as side channels (continued)

- Some processors provide a means to prefetch data that will be needed soon
    - Usually encoded as “hint” or “nop space” instructions that may have no effect
    - x86 processors provide several variants of PREFETCH with a temporal hint
    - This may result in a prefetched address being allocated into a cache
- Processors will perform page table walks and populate TLBs on prefetch
    - This may happen even if the address is not actually fetched into the cache

```
asm volatile ("prefetcht0 (%0)" : : "r" (p));
asm volatile ("prefetcht1 (%0)" : : "r" (p));
asm volatile ("prefetcht2 (%0)" : : "r" (p));
asm volatile ("prefetchnta (%0)" : : "r" (p));
```

Branch Prediction and Speculation

Branch prediction



Branch prediction

- Applications frequently use program control flow instructions (branches)
  - Conditionals such as “if then” are implemented as conditional direct branches
  - e.g. “if (raining) pack_umbrella();” depends upon the value of “raining”
- Branch condition evaluation is known as “resolving” the branch condition
  - This might require (slow) loads from memory (e.g. not immediately in the L1 D$)
- Rather than wait for branch resolution, predict outcome of the branch
  - This keeps the pipeline(s) filled with (hopefully) useful instructions
- Some ISAs allow compile-time “hints” to be provided for branches
  - These are encoded into the branch instruction, but may not be used
  - “if (likely(condition))” sequences in Operating System kernels

Speculative Execution

- Speculative Execution is implemented as a variation of Out-of-Order Execution
  - Uses the same underlying structures already present such as ROB, etc.
- Instructions that are speculative are specially tagged in the Re-Order Buffer
  - They must not have an architecturally visible effect on the machine state
  - Do not update the architectural register file until speculation is committed
  - Stores to memory are tagged in the ROB and will not hit the store buffers
- Exceptions caused by instructions will not be raised until instruction retirement
  - Tag the ROB to indicate an exception (e.g. privilege check violation on load)
  - If the instruction never retires, then no exception handling is invoked

Branch prediction and speculation

- Once the branch condition is successfully resolved:
  1) If the predicted branch was correct, speculated instructions can be retired
     - Once instructions are the oldest in the machine, they can retire normally
     - They become architecturally visible and stores ultimately reach memory
     - Exceptions are handled for instructions failing an access privilege check
     - Significant performance benefit from executing the speculated path
  2) If the predicted branch was incorrect, speculated instructions can be discarded
     - They exist only in the ROB, remove/fix, and discard store buffer entries
     - They do not become architecturally visible
     - Performance hit incurred from flushing the pipeline/undoing speculation

Conditional branches

- A conditional branch will be performed based upon the state of the condition flags
  - Condition flags are commonly implemented in modern ISAs and set by certain instructions
  - Some ISAs are optimized to set the condition flags only in specific instruction variants
- Most loops are implemented as a conditional backward jump following a test:

```
       movq $0, %rax
loop:
       incq %rax
       cmpq $10, %rax
       jle loop
```

Conditional branch prediction

- Branch behavior is rarely random and can usually be predicted with high accuracy
  - Branch predictor is first “trained” using historical direction to predict future
  - Over 99% accuracy is possible depending upon the branch predictor sophistication
- Branches are identified based upon the (virtual) address of the branch instruction
  - Index into branch prediction structure containing pattern history e.g. T,T,N,N,T,T,N,N
  - These may be tagged during instruction fetch/decode using extra bits in the I$
- Most contemporary high performance branch predictors combine local/global history
  - Recognizing that branches are rarely independent and usually have some correlation
  - A Global History Register is combined with saturating counters for each history entry
  - May also hash GHR with address of the branch instruction (e.g. “Gshare “ predictor)

Conditional branch prediction

- Modern designs combine various different branch predictors
    - Simple loop predictors include BTFN (Backward Taken Forward Not)
    - Contemporary processors would identify the previous example as early as decode
    - May directly issue repeated loop instructions from pre-decoded instruction cache
- Optimize size of predictor internal structures by hashing/indexing on address
    - Common not to use the full address of a branch instruction in the history table
    - This causes some level of (known) interference between unrelated branches

Indirect branch prediction

- More complex branch types include “indirect branches”
  - Target is stored within a register or a memory location (e.g. function pointer, virtual method)
  - The destination of the branch is not known at compile time
- Indirect predictor attempts to guess the location of an indirect branch
  - Recognizes the branch based upon the (virtual) address of the instruction
  - Uses historical data from previous branches to guess the next time
- Speculation occurs beyond indirect branch into predicted target address
  - If the predicted target address is incorrect, discard speculative instructions

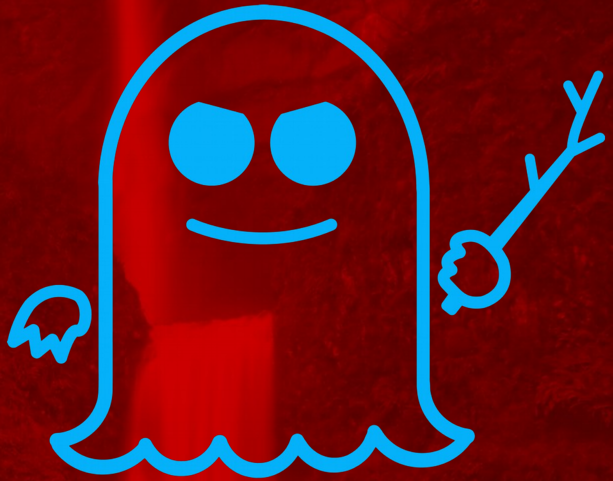
Branch predictor optimization

- Branch prediction is vital to modern microprocessor performance
  - Significant research has gone into optimization of prediction algorithms
  - Many different predictors may be in use simultaneously with voting arbitration
  - Accuracy rates of over 99% are possible depending upon the workload
- Predictors are in the critical path for instruction fetch/decode
  - Must operate quickly to prevent adding delays to instruction dispatch
  - Common industry optimizations aimed at reducing predictor storage
  - Optimizations include indexing on low order address bits of branches

Speculation in modern processors

- Modern microprocessors heavily leverage speculative execution of instructions
  - This achieves significant performance benefit at the cost of complexity and power
  - Required in order to maintain the level of single thread performance gains anticipated
- Speculation may cross contexts and privilege domains (even hypervisor entry/exit)
- Conventional wisdom holds that speculation is invisible to programmer and applications
  - Speculatively executed instructions are discarded and their results flushed from store buffers
  - However speculation may result in cache loads (allocation) for values being processed
- It is now realized that certain side effects of speculation may be observable
  - This can be used in various exploits against popular implementations

MELTDOWN



SPECTRE

Meltdown and Spectre microarchitecture vulnerabilities

- Meltdown (CVE-2017-5754) and Spectre (CVE-2017-2753, CVE-2017-5715) are branded vulnerabilities discovered in common industry-wide microprocessor optimizations
    - Discovered independently by multiple parties including TU Graz and Google Project Zero
    - They came with a website and logos as well as scary videos to motivate public reaction
    - These are serious exploits that require mitigation especially in shared environments
- They exploit speculative execution to bypass normal system security boundaries
    - e.g. page table protections against reading Operating System memory
- We do not need to panic and throw away all of our performance toys
    - Speculation is not entirely broken forevermore, some implementations are vulnerable

Meltdown and Spectre microarchitecture vulnerabilities

- Operating System vendors provide tools to determine vulnerability and mitigation
  - The specific mitigations vary from one architecture and Operating System to another
- Windows includes new PowerShell scripts, various Linux tools have been created
- Very recent (upstream) Linux kernels include the following new “sysfs” entries:

```
$ grep . /sys/devices/system/cpu/vulnerabilities/*
/sys/devices/system/cpu/vulnerabilities/meltdown:Mitigation: PTI
/sys/devices/system/cpu/vulnerabilities/spectre_v1:Vulnerable
/sys/devices/system/cpu/vulnerabilities/spectre_v2:Vulnerable: Minimal
generic ASM retpoline
```

Spectre-v1: Bounds Check Bypass (CVE-2017-2573)

- Modern microprocessors may speculate beyond a bounds check condition
- What's wrong with the following code?

```
If (untrusted_offset < limit) {
    trusted_value = trusted_data[untrusted_offset];
    tmp = other_data[(trusted_value)&mask];
    ...
}
```

Spectre-v1: Bounds Check Bypass (cont)

- The code following the bounds check is known as a “gadget” (see ROP attacks)
  - Existing code contained within a different victim context (e.g. Operating System/Hypervisor)
- Code following the untrusted_offset bounds check may be executed speculatively
  - Resulting in the speculative loading of trusted data into a local variable
  - This trusted data is used to calculate an offset into another structure
- Relative offset of other_data accessed can be used to infer trusted_value
  - L1D$ cache load will occur for other_data at an offset correlated with trusted_value
  - Measure which cache location was loaded speculatively to infer the secret value

Mitigating Spectre-v1: Bounds Check Bypass

- Existing hardware lacks the capability to limit speculation in this instance
- Mitigation: modify software programs in order to prevent the speculative load
    - On most architectures this requires the insertion of a serializing instruction (e.g. “lfence”)
    - Some architectures can use a conditional masking of the untrusted_offset
        - Prevent it from ever (even speculatively) having an out-of-bounds value
    - Linux adds new “nospec” accessor macros to prevent speculative loads
- Tooling exists to scan source and binary files for offending sequences
    - Much more work is required to make this a less painful experience

Mitigating Spectre-v1: Bounds Check Bypass (cont)

- Example of mitigated code sequence:

```
If (untrusted_offset < limit) {
    serializing_instruction();
    trusted_value = trusted_data[untrusted_offset];
    tmp = other_data[(trusted_value)&mask];
    ...
}
```

Spectre-v2: Branch Predictor Poisoning (CVE-2017-5715)

- Modern microprocessors may be susceptible to “poisoning” of the branch predictors
- Rogue application “trains” the indirect predictor to predict branch to “gadget” code
  - Processor incorrectly speculates down an indirect branch into existing code but the offset of the branch is under malicious user control – repurpose existing privileged code as a “gadget”
- Relies upon the branch prediction hardware not fully disambiguating branch addresses
  - Virtual address of branch in malicious user code constructed to use same predictor entry as a branch in another application or the Operating System kernel running at higher privilege
- Privileged data is extracted using a similar cache access pattern to Spectre-v1

Mitigating Spectre-v2: Big hammer approach

- Existing branch prediction hardware lacks capability to disambiguate different contexts
  - Relatively easy to add this in future cores (e.g. using ASID/PCID tagging in branches)
- Initial mitigation is to disable the indirect branch predictor hardware (sometimes)
  - Completely disabling indirect prediction would seriously harm core performance
  - Instead disable indirect branch prediction when it is most vulnerable to exploit
  - e.g. on entry to kernel or Hypervisor from less privileged application context
- Flush the predictor state on context switch to a new application (process)
  - Prevents application-to-application attacks across a new context
- A fine grained solution may not be possible on existing processors

Tangent: Microcode, Millicode, and Chicken Bits

- Modern microprocessors are extremely complex machines requiring huge capital investment
  - A high performance core might require a 300+ person team, and 4 years of engineering effort
- Consequently the ability to handle potential issues in the field is extremely compelling
- Modern cores provide thousands of hidden tunable knobs (chicken bits) that allow a design team to “chicken out” and disable certain features that aren't working in whole or in part
  - A high performance core might have as many as 10,000 different chicken bits available
- A chicken bit might be programmed in firmware prior to system boot
  - e.g. “disable all indirect branch prediction when in privileged state” (if this is possible)
- Or it might be exposed to the Operating System to poke it as needed

Microcode, Millicode, and Chicken Bits (cont)

- Some processors contain a mix of hardwired logic and microcoded instructions
  - Typically used for complex instructions in CISC architectures such as x86, POWER, Z/Arch…
  - Not used in the fast path for critical core logic (such as caches, page table walks, etc.)
- Microcode defines control signals within the core and state transitions between them
  - A microcode sequencer (simple state machine) within the core sets control signals following a “program” (really a simple set of state transitions) contained within fast on-chip ROM
  - Example is repeated instructions in x86 which can be implemented in microcode sequences
- A small (a few KB) microcode patch RAM can be used to patch some instruction behavior
  - Microcode is an encrypted, signed blob from the CPU manufacturer, format is (mostly) secret
  - Operating Systems or firmware can load microcode/millicode at system boot time or later

Mitigating Spectre-v2: Big hammer (cont)

- Microcode can be used on some microprocessors to alter instruction behavior
- It can also be used add new “instructions” or system registers that exhibit side effects
- On Spectre-v2 impacted x86 microprocessors, microcode adds new SPEC_CTRL MSRs
  - Model Specific Registers are special memory addresses that control core behavior
  - Identified using the x86 “CPUID” instruction which enumerates available capabilities
  - IBRS (Indirect Branch Restrict Speculation)
    - Used on entry to more privileged context to restrict branch speculation
  - IBPB (Indirect Branch Predictor Barrier)
    - Used on context switch into a new process to flush predictor entries
- What are the problems with using microcode interfaces?

Mitigating Spectre-v2 with Retpolines

- Microcoded mitigations are effective but expensive due to their implementation
  - Many cores do not have convenient logic to disable predictors so “IBRS” must also disable independent logic within the core. It may take many thousands of cycles on kernel entry
- Google decided to try an alternative solution using a pure software approach
  - If indirect branches are the problem, then the solution is to avoid using them
  - “Retpolines” stand for “Return Trampolines” which replace indirect branches
  - Setup a fake function call stack and “return” in place of the indirect call

Mitigating Spectre with Retpolines (cont)

- Example retpoline call sequence on x86 (source: https://support.google.com/faqs/answer/7625886)

```
  call set_up_target;
capture_spec:
  pause;
  jmp capture_spec;
set_up_target:
  mov %r11, (%rsp);
  ret;
```

Mitigating Spectre-v2 with Retpolines (cont)

- Retpolines are a novel solution to an industry-wide problem with indirect branches
    - Credit to Google for releasing these freely without patent claims and encouraging adoption
- However they present a number of challenges for Operating Systems and users
    - Requires a recompilation of software, and possibly dynamic patching to disable on future cores
        - Mitigation should be temporary in nature, automatically disabled on future silicon
    - Cores will speculate the return path from functions using an RSB (Return Stack Buffer)
        - Need to explicitly manage (stuff) the RSB to avoid malicious interference
    - Certain cores will use alternative predictors when RSB underflow occurs

Meltdown

- Implementations of Out-of-Order execution that strictly follow the original Tomasulo algorthim handle exceptions arising from speculatively executed instructions at instruction retirement
- Speculated instructions do not trigger (synchronous) exceptions in response to execution
  - Loads that are not permitted will not be reported until they are no longer speculative
  - At that time, the application will likely receive a “segmentation fault” or other error
- Some implementations may perform load permission checks in parallel with the load
  - This improves performance and the rationale is that the load is only speculative
- A race condition may thus exist allowing access to privileged data

Meltdown (continued)

- A malicious attacker arranges for exploit code similar to the following to speculatively execute:

```
if (spec_cond) {
            unsigned char value = *(unsigned char *)ptr;
            unsigned long index2 = (((value>>bit)&1)*0x100)+0x200;
            maccess(&data[index2]);
}
```

- “data” is a user controller array to which the attacker has access, “ptr” contains privileged data

Meltdown (continued)

- When the right conditions exist, this branch of code will run speculatively
  - The privilege check for “value” will fail, but only result in an entry tag in the ROB
  - The access will occur although “value” will be discarded when speculation is undone
- The offset accessed in the “data” user array is dependent upon the value of privileged data
  - We can use this as a 1 bit counter between several possible entries of the user data array
- Cache side channel timing analysis is used to measure which “data” location was accessed
  - Time access to “data” locations 0x200 and 0x300 to infer value of desired bit
  - Access is done in reverse in my code to account for cache line prefetcher

Mitigating Meltdown

- The “Meltdown” vulnerability requires several conditions:
  - Privileged data must reside in memory for which active translations exist
  - On some processor designs the data must also be in the L1 data cache
- Primary Mitigation: separate application and Operating System page tables
  - Each application continues to have its own page tables as before
  - The kernel has separate page tables not shared with applications
  - Limited shared pages exist only for entry/exit trampolines and exceptions

Mitigating Meltdown

- Linux calls this page table separation “PTI”: Page Table Isolation
    - Requires an expensive write to core control registers on every entry/exit from the OS kernel
    - e.g. TTBR write on impacted ARMv8, CR3 on impacted x86 processors
- Only enabled by default on known-vulnerable microprocessors
    - An enumeration is defined to discover future non-impacted silicon
- Address Space IDentifiers (ASIDs) can significantly improve performance
    - ASIDs on ARMv8, PCIDs (Process Context IDs) on x86 processors
    - TLB entries are tagged with address space so a full invalidation isn't required
    - Significant performance delta between older (pre-2010 x86) cores and newer ones

Variations on a theme: variant 3a (Sysreg read)

- Variations of these microarchitecture attacks are likely to be found for many years
- An example is known as “variant 3a”. Some microprocessors will allow speculative read of privileged system registers to which an application should not normally have access
  - Can be used to determine the address of key structures such as page table base registers

Related Research

- Meltdown and Spectre are only recent examples of microarchitecture attack
- A memorable attack known as “Rowhammer” was discovered previously
  - Exploit the implementation of (especially non-ECC) DDR memory
  - Possible to perturb bits in adjacent memory lines with frequent access
  - Can use this approach to flip bits in sensitive memory and bypass access restrictions
  - For example change page access permissions in the system page tables
- Another recent attack known as “MAGIC” exploits NBTI in silicon
  - Negative-bias temperature instability impacts reliability of MOSFETs (“transistors”)
  - Can be exploited to artificially age silicon devices and decrease longevity
  - Proof of concept demonstrated with code running on OpenSPARC core

Summary

Summary

Today's lecture covered the following topics:

- Introduction to microarchitecture as implementation of architecture
- In order vs. Out-of-Order execution in microarchitectures
- Caches, virtual memory, and side channel analysis
- Branch prediction and speculative execution
- Spectre and Meltdown vulnerabilities
- Mitigation approaches and solutions
- Related research into hardware

redhat.

THANK YOU

g+ plus.google.com/+RedHat

f facebook.com/redhatinc

in linkedin.com/company/red-hat

twitter.com/RedHatNews

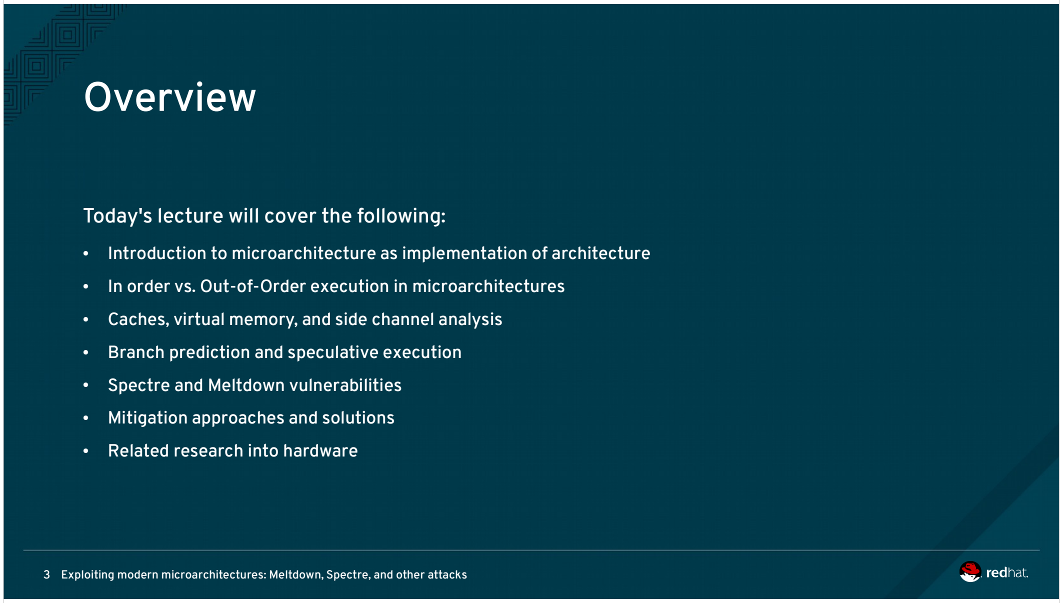
YouTube youtube.com/user/RedHatVideos

redhat®

Exploiting modern microarchitectures: Meltdown, Spectre, and other attacks

Jon Masters, Computer Architect, Red Hat, Inc.
jcm@redhat.com | @jonmasters

MELTDOWN
SPECTRE

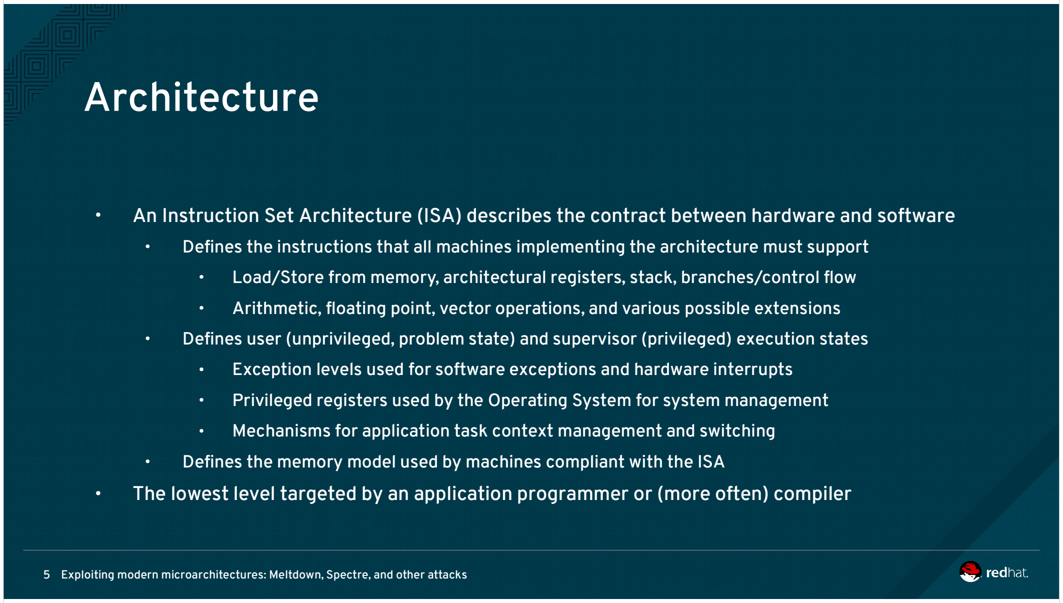


Overview

Today's lecture will cover the following:

- Introduction to microarchitecture as implementation of architecture
- In order vs. Out-of-Order execution in microarchitectures
- Caches, virtual memory, and side channel analysis
- Branch prediction and speculative execution
- Spectre and Meltdown vulnerabilities
- Mitigation approaches and solutions
- Related research into hardware

Architecture

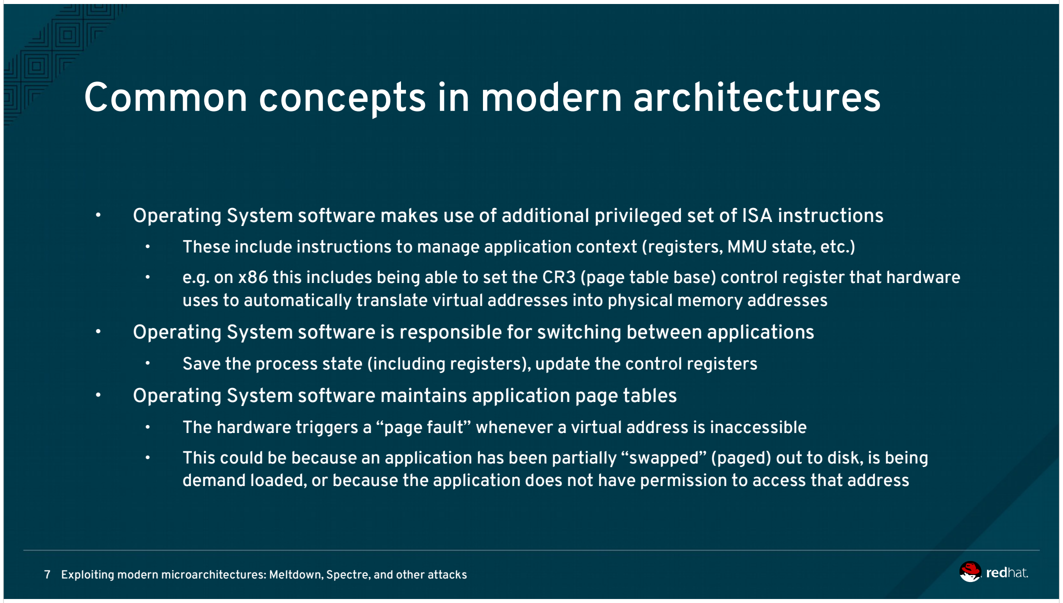


Architecture

- An Instruction Set Architecture (ISA) describes the contract between hardware and software
  - Defines the instructions that all machines implementing the architecture must support
    - Load/Store from memory, architectural registers, stack, branches/control flow
    - Arithmetic, floating point, vector operations, and various possible extensions
  - Defines user (unprivileged, problem state) and supervisor (privileged) execution states
    - Exception levels used for software exceptions and hardware interrupts
    - Privileged registers used by the Operating System for system management
    - Mechanisms for application task context management and switching
  - Defines the memory model used by machines compliant with the ISA
- The lowest level targeted by an application programmer or (more often) compiler

Common concepts in modern architectures

- Application programs make use of a standard (non-privileged) set of ISA instructions
  - Programs (known as “processes” or “tasks” when running) execute in a lower privilege state
  - These are often referred to as “rings”, “exception levels”, etc.
- Application programs execute using a virtual memory environment
  - Virtual memory is divided into 4K (or larger) “pages”, the smallest unit at which it is managed
  - The processor Memory Management Unit (MMU) translates all memory accesses using page tables
  - The Operating System provides the illusion of a flat large address space by managing page tables
- Application programs request runtime services from the Operating System using system calls
  - The Operating System provided system calls run in the same virtual memory environment
    - e.g. Linux maps all of physical memory beginning at the high end of every process
  - Page table protections (normally) prevent applications from seeing this OS memory



Common concepts in modern architectures

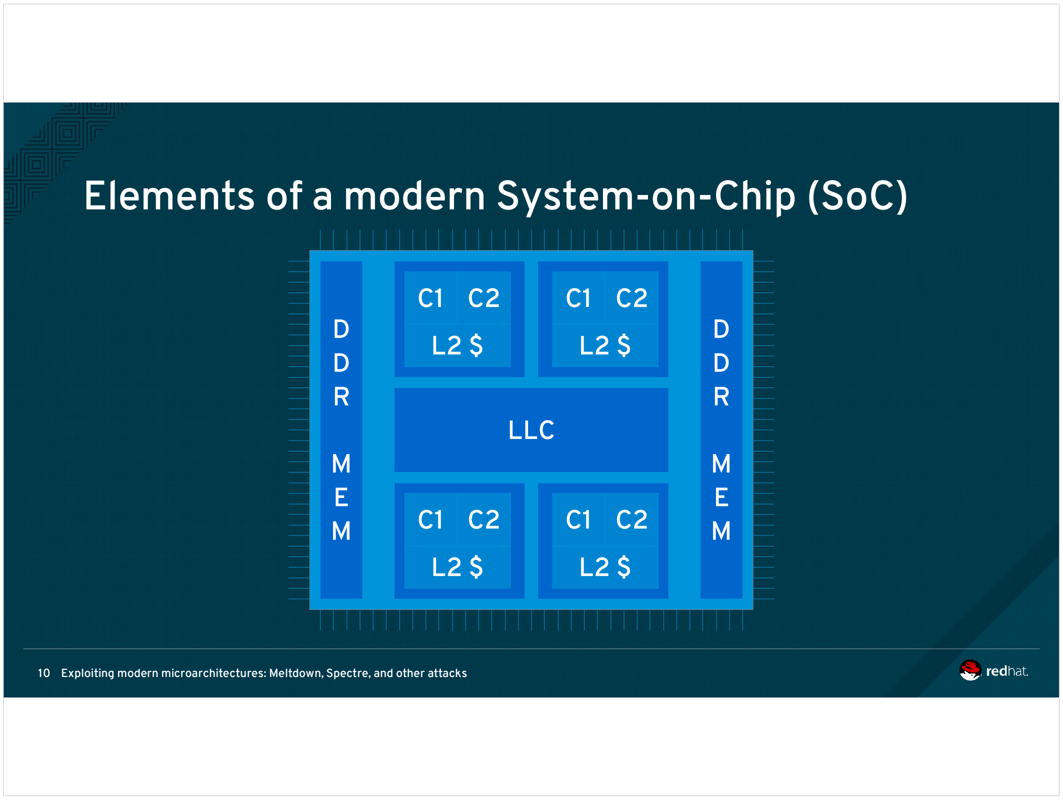
- Operating System software makes use of additional privileged set of ISA instructions
  - These include instructions to manage application context (registers, MMU state, etc.)
  - e.g. on x86 this includes being able to set the CR3 (page table base) control register that hardware uses to automatically translate virtual addresses into physical memory addresses
- Operating System software is responsible for switching between applications
  - Save the process state (including registers), update the control registers
- Operating System software maintains application page tables
  - The hardware triggers a “page fault” whenever a virtual address is inaccessible
  - This could be because an application has been partially “swapped” (paged) out to disk, is being demand loaded, or because the application does not have permission to access that address

Examples of computer architectures

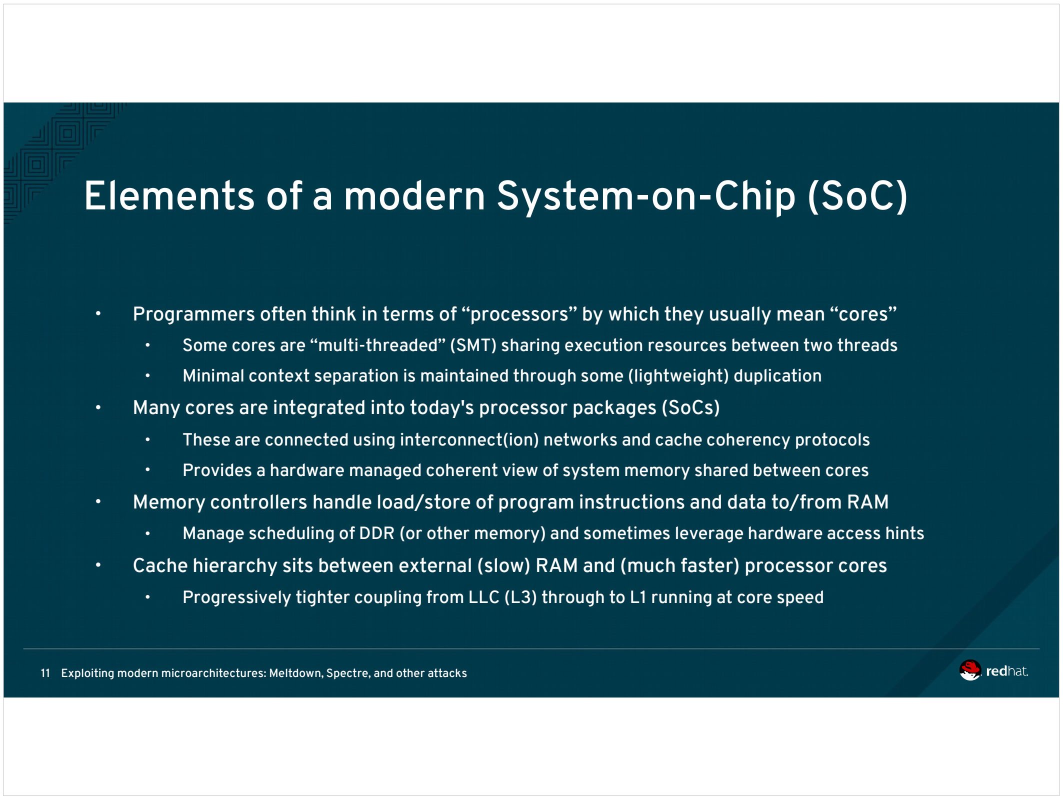
- Intel “x86” (Intel x64/AMD64)
  - CISC (Complex Instruction Set Computer)
  - Variable width instructions (up to 15 bytes)
  - 16 GPRs (General Purpose Registers)
  - Can operate directly on memory
  - 64-bit flat virtual address space
    - “Canonical” 48/56-bit addressing
    - Upper half kernel, Lower half user
    - Removal of older segmentation registers (except FS/GS)

- ARM ARMv8 (AArch64)
  - RISC (Reduced Instruction Set Computer)
  - Fixed width instructions (4 bytes fixed)
    - Clean uniform decode table
  - 32 GPRs (General Purpose Registers)
  - Classical RISC load/store using registers for all operations (first load from memory)
  - 64-bit flat virtual address space
    - Split into lower and upper halves

Microarchitecture

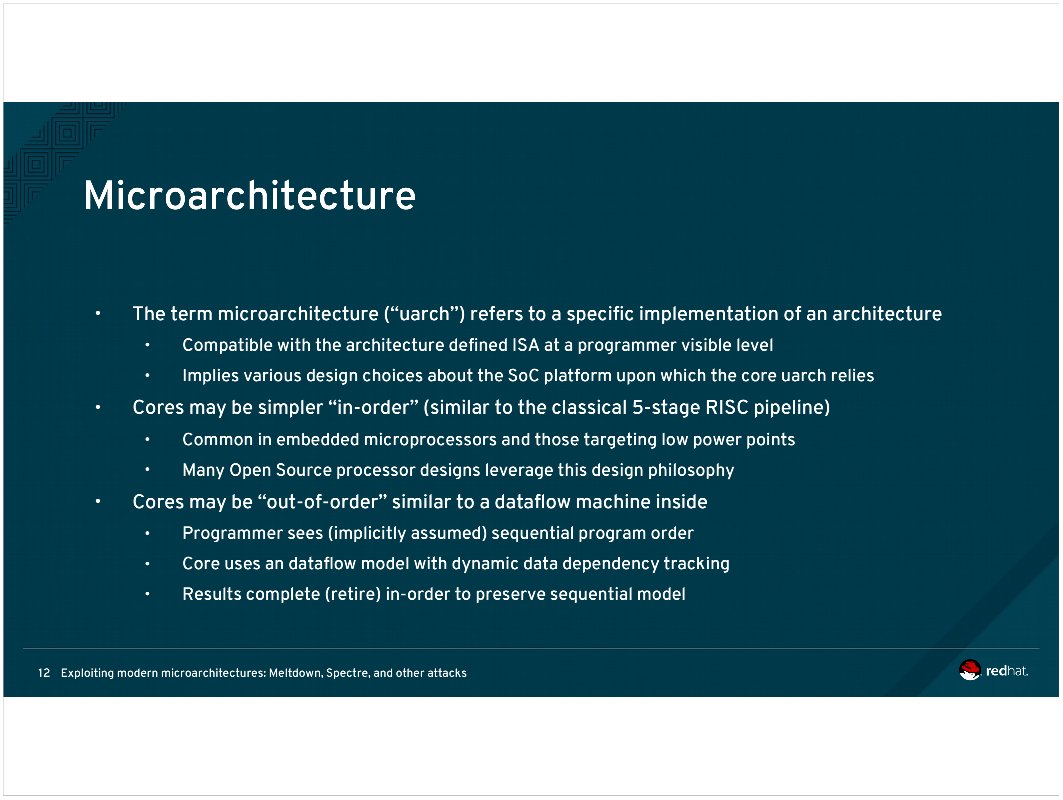


Elements of a modern System-on-Chip (SoC)



Elements of a modern System-on-Chip (SoC)

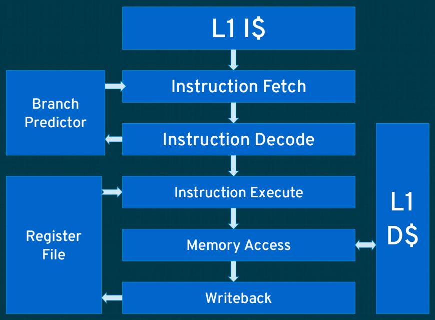
- Programmers often think in terms of “processors” by which they usually mean “cores”
  - Some cores are “multi-threaded” (SMT) sharing execution resources between two threads
  - Minimal context separation is maintained through some (lightweight) duplication
- Many cores are integrated into today's processor packages (SoCs)
  - These are connected using interconnect(ion) networks and cache coherency protocols
  - Provides a hardware managed coherent view of system memory shared between cores
- Memory controllers handle load/store of program instructions and data to/from RAM
  - Manage scheduling of DDR (or other memory) and sometimes leverage hardware access hints
- Cache hierarchy sits between external (slow) RAM and (much faster) processor cores
  - Progressively tighter coupling from LLC (L3) through to L1 running at core speed

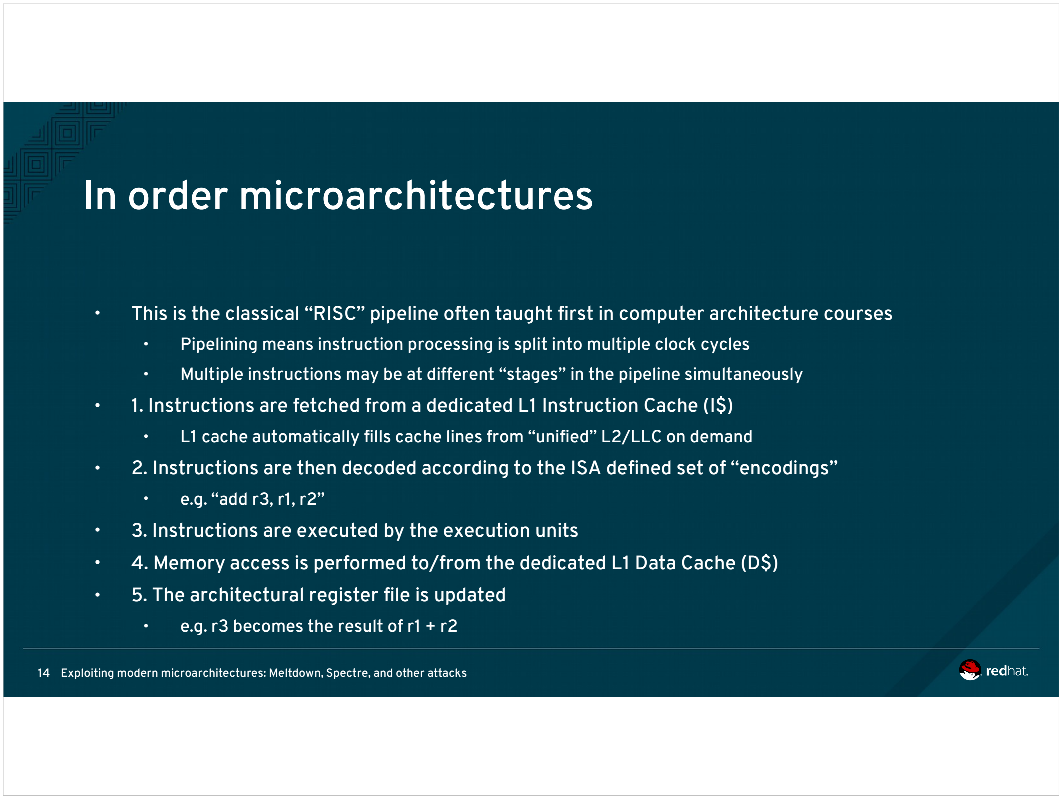


Microarchitecture

- The term microarchitecture (“uarch”) refers to a specific implementation of an architecture
  - Compatible with the architecture defined ISA at a programmer visible level
  - Implies various design choices about the SoC platform upon which the core uarch relies
- Cores may be simpler “in-order” (similar to the classical 5-stage RISC pipeline)
  - Common in embedded microprocessors and those targeting low power points
  - Many Open Source processor designs leverage this design philosophy
- Cores may be “out-of-order” similar to a dataflow machine inside
  - Programmer sees (implicitly assumed) sequential program order
  - Core uses an dataflow model with dynamic data dependency tracking
  - Results complete (retire) in-order to preserve sequential model

Elements of a modern in-order core

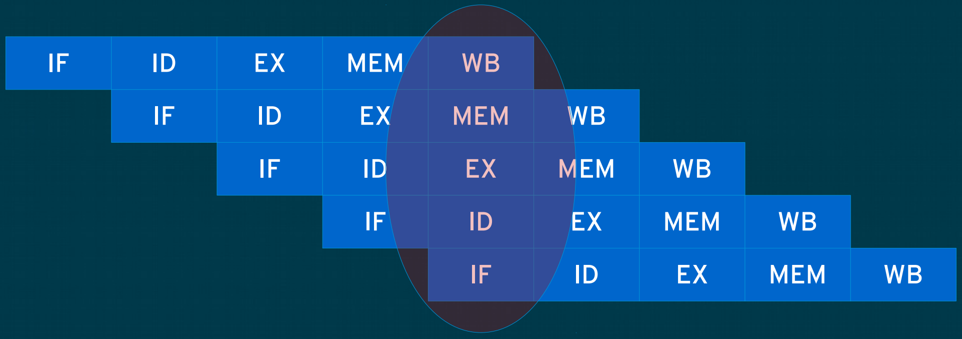




In order microarchitectures

- This is the classical “RISC” pipeline often taught first in computer architecture courses
  - Pipelining means instruction processing is split into multiple clock cycles
  - Multiple instructions may be at different “stages” in the pipeline simultaneously
- 1. Instructions are fetched from a dedicated L1 Instruction Cache (I$)
  - L1 cache automatically fills cache lines from “unified” L2/LLC on demand
- 2. Instructions are then decoded according to the ISA defined set of “encodings”
  - e.g. “add r3, r1, r2”
- 3. Instructions are executed by the execution units
- 4. Memory access is performed to/from the dedicated L1 Data Cache (D$)
- 5. The architectural register file is updated
  - e.g. r3 becomes the result of r1 + r2

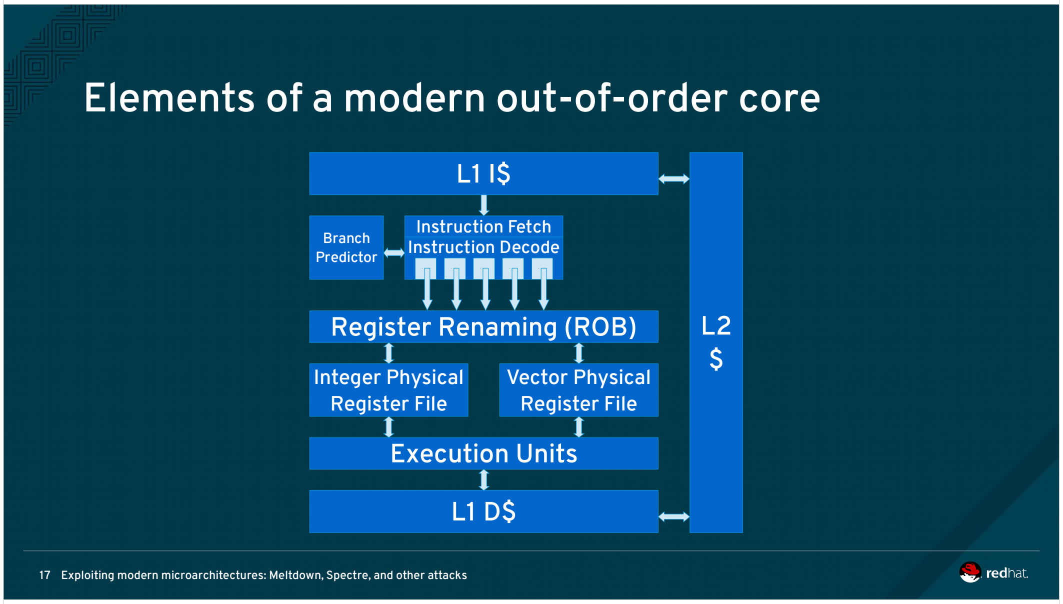
An in-order pipeline visualized



IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB

In order microarchitectures (continued)

- An in-order machine can suffer from pipeline stalls when stages are not ready
- The memory access stage may be able to load from the L1 D$ in a single cycle
- But if it is not in the L1 D$ then we insert a pipeline “bubble” while we wait for the data
  - This may take many additional cycles while the data is fetched from further away
- Limited capability to hide latency of instructions
  - Future instructions may not be dependent upon stalling earlier instructions



Elements of a modern out-of-order core

L1 I$

Branch Predictor

Instruction Fetch
Instruction Decode

Register Renaming (ROB)

Integer Physical Register File

Vector Physical Register File

Execution Units

L1 D$

L2 $

Out-of-Order (OoO) microarchitectures

- This type of design is common in aggressive high performance microprocessors
  - Also known as “dynamic execution” because it can change at runtime
  - Invented by Robert Tomasulo (used in System/360 Model 91 Floating Point Unit)
- Instructions are fetched and decoded by an in-order “front end” similar to before
- Instructions are dispatched to an out-of-order “backend”
  - Allocated an entry in a ROB (Re-Order Buffer), Reservation Stations
  - May use a Re-Order Buffer and separate Retirement (Architectural) Register File or single physical register file and a Register Alias Table (RAT)
- Re-Order Buffer defines an execution window of out-of-order processing
  - These can be quite large – over 200 entries in contemporary designs

Out-of-Order (OoO) microarchitectures (cont.)

- Instructions wait only until their dependencies are available
  - Later instructions may execute prior to earlier instructions
  - Re-Order Buffer allows for more physical registers than defined by the ISA
  - Removes some so-called data “hazards”
    - WAR (Write-After-Read) and WAW (Write-After-Write)
- Instructions complete (“retire”) in-order
  - When an instruction is the oldest in the machine, it is “retired”
  - State becomes architecturally visible (updates the architectural register file)

Microarchitecture (continued)

- The term microarchitecture ("uarch") refers to a specific implementation of an architecture
  - Implies various design choices about the SoC platform upon which the core uarch relies
- Questions we can ask about a given implementation include the following:
  - What's the design point for an implementation – Power vs Performance vs Area (cost)
    - Low power simple in-order design vs Fully Out-of-Order high performance design
  - How are individual instructions implemented? How many cycles do they take?
    - How many pipelines are there? Which instructions can issue to a given pipe?
  - How many microarchitectural registers are implemented? How many ports in the register file?
    - How big is the Re-Order Buffer (ROB) and the execution window?

Examples of computer microarchitectures

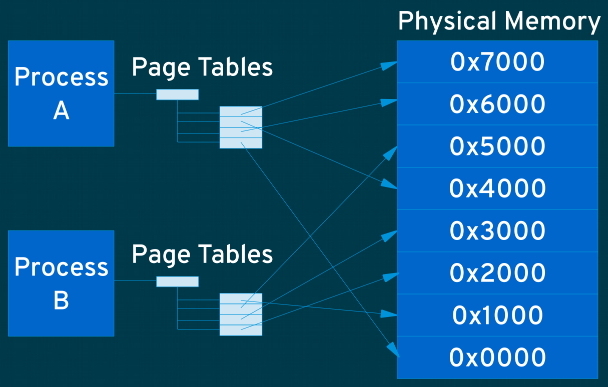
- Intel Core i7-6560U (“Skylake” uarch)
  - 2 SMT threads per core (configurable)
  - 32KB L1I$, 32KB L1D$, 256KB L2$
  - 4-8* uops instruction issue per cycle
  - 8 execution ports (14-19 stage pipeline)
  - 224 entry ROB (Re-Order Buffer)
  - 14nm FinFET with 13 metal layers

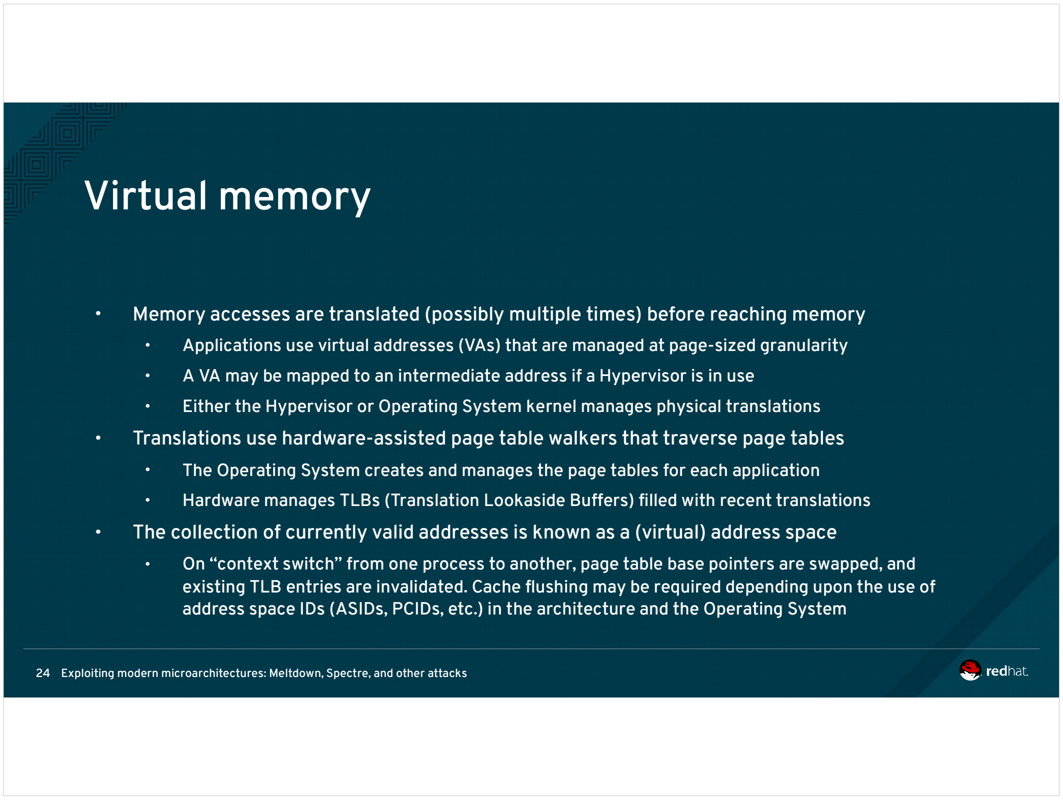
- IBM POWER8E (POWER8 uarch)
  - Up to 8 SMT threads per core (configurable)
  - 32KB L1I$, 64KB L1D$, 512KB L2$
  - 8-10 wide instruction issue per cycle
  - 16 execution piplines (15-23 stage pipeline)
  - 224 entry Global Completion Table (GCT)
  - 22nm SOI with 15 metal layers

* Typical is 4uops with rate exception

Virtual Memory and Caches

Virtual memory



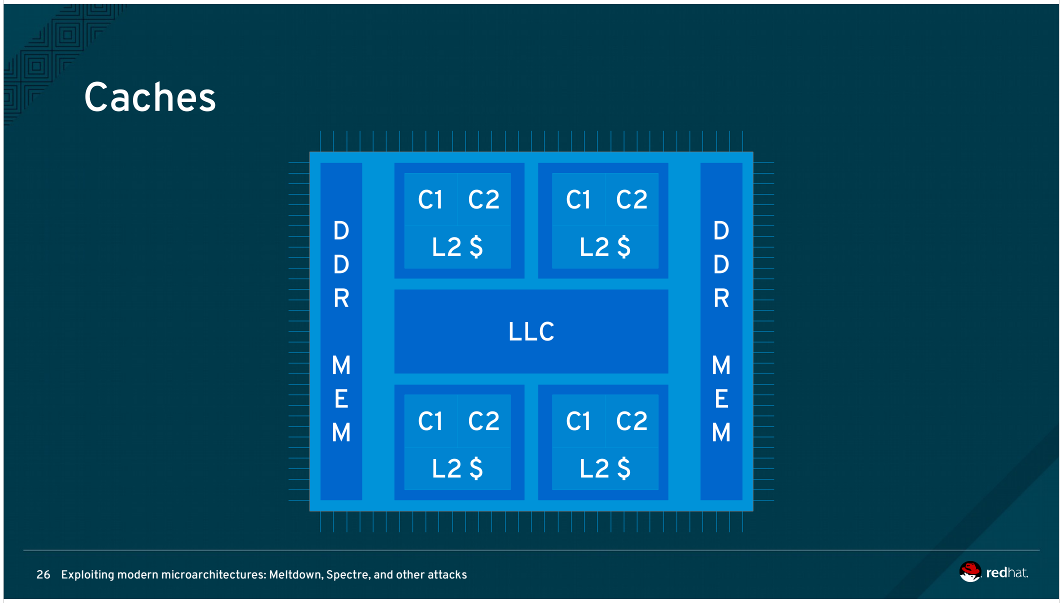


Virtual memory

- Memory accesses are translated (possibly multiple times) before reaching memory
    - Applications use virtual addresses (VAs) that are managed at page-sized granularity
    - A VA may be mapped to an intermediate address if a Hypervisor is in use
    - Either the Hypervisor or Operating System kernel manages physical translations
- Translations use hardware-assisted page table walkers that traverse page tables
    - The Operating System creates and manages the page tables for each application
    - Hardware manages TLBs (Translation Lookaside Buffers) filled with recent translations
- The collection of currently valid addresses is known as a (virtual) address space
    - On “context switch” from one process to another, page table base pointers are swapped, and existing TLB entries are invalidated. Cache flushing may be required depending upon the use of address space IDs (ASIDs, PCIDs, etc.) in the architecture and the Operating System

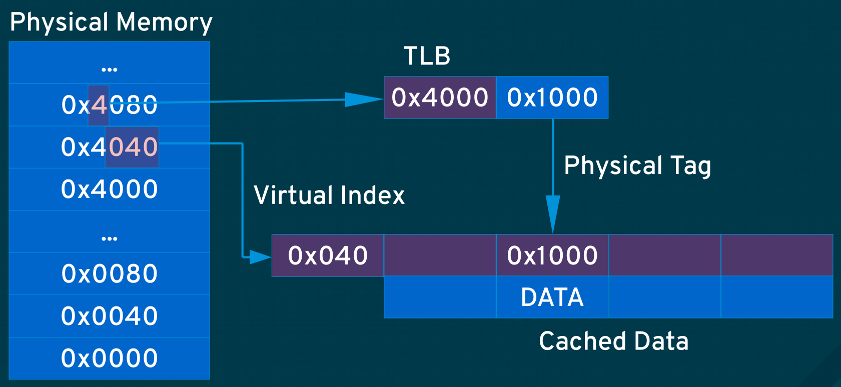
Virtual memory

- Applications have a large flat Virtual Address space mostly to themselves
  - Text (code) and data are dynamically linked into the virtual address space at application load automatically using metadata from the ELF (Executable Linkng Format) application binary
  - Dynamic libraries are mapped into the Virtual Address space and may be shared by applications
- Operating Systems may map some OS kernel data into application virtual address space
  - Limited examples intended for deliberate use by applications (e.g. Linux VDSO) for performance
    - Data can be directly read from the Virtual Dynamic Shared Object without a system call
  - The rest is explicitly protected by marking it as inaccessible in the application page tables
- Linux (used to) maps all of physical memory into every running application process
  - Allows for system calls into the OS without performing a full context switch on entry
  - The kernel is linked with high virtual addresses and mapped into every process



Caches

Caches



Physical Memory

...

0x4080

0x4040

0x4000

...

0x0080

0x0040

0x0000

TLB

0x4000 0x1000

Physical Tag

Virtual Index

0x040

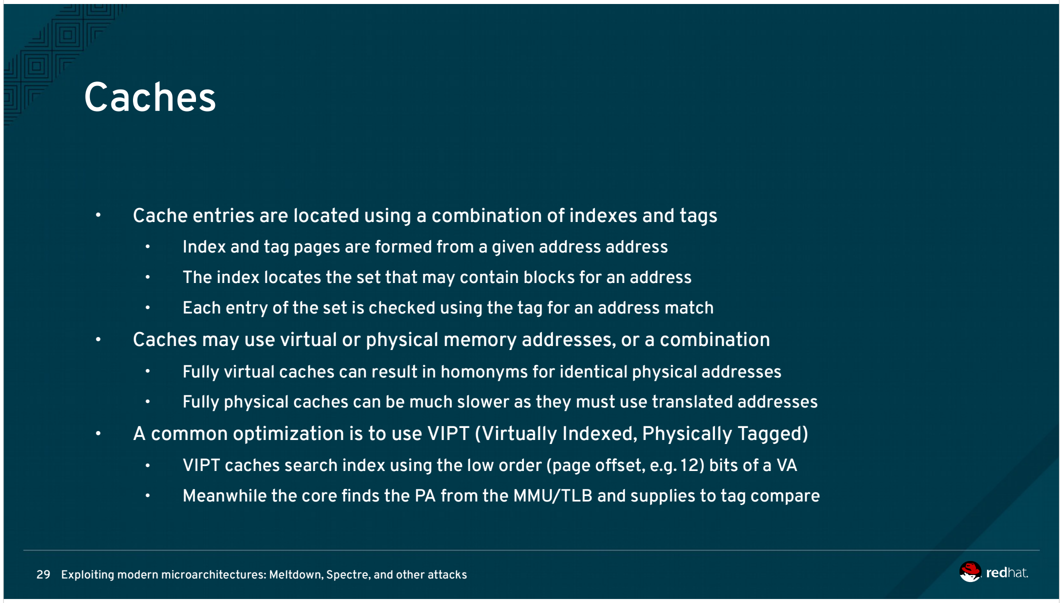
0x1000

DATA

Cached Data

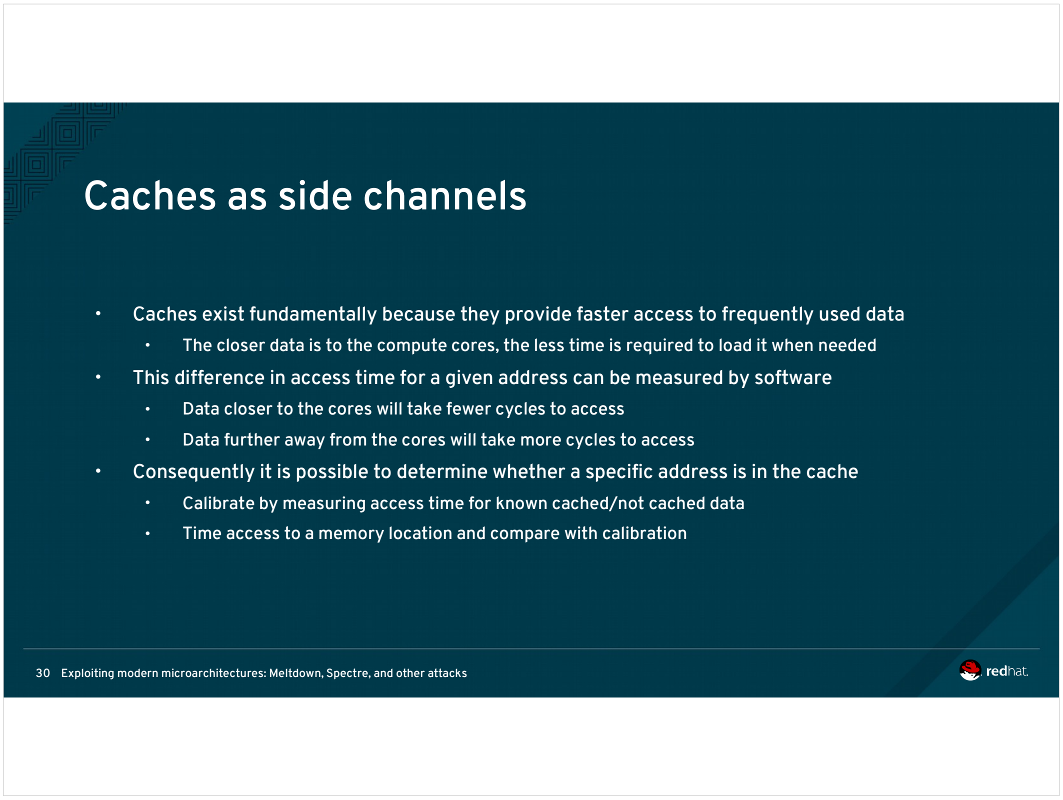
Caches

- Caches exist because the principal of locality says recently used data is likely to be used again
- Unfortunately we have a choice between “small and fast” and “large and slow”
  - Levels of cache provide the best of both, replacement policies handle cache eviction
- Caches are organized into sets where each set can contain multiple cache lines
  - A typical cache line is 64 or 128 bytes and represents a block of memory
  - A typical memory block will map to single cache set, but can be in any “way” of a set
- Caches may be direct mapped or (fully) associative depending upon complexity
  - Direct mapped allows one memory location to exist only in a specific cache location
  - Associative caches allow one memory location to map to one of N cache locations



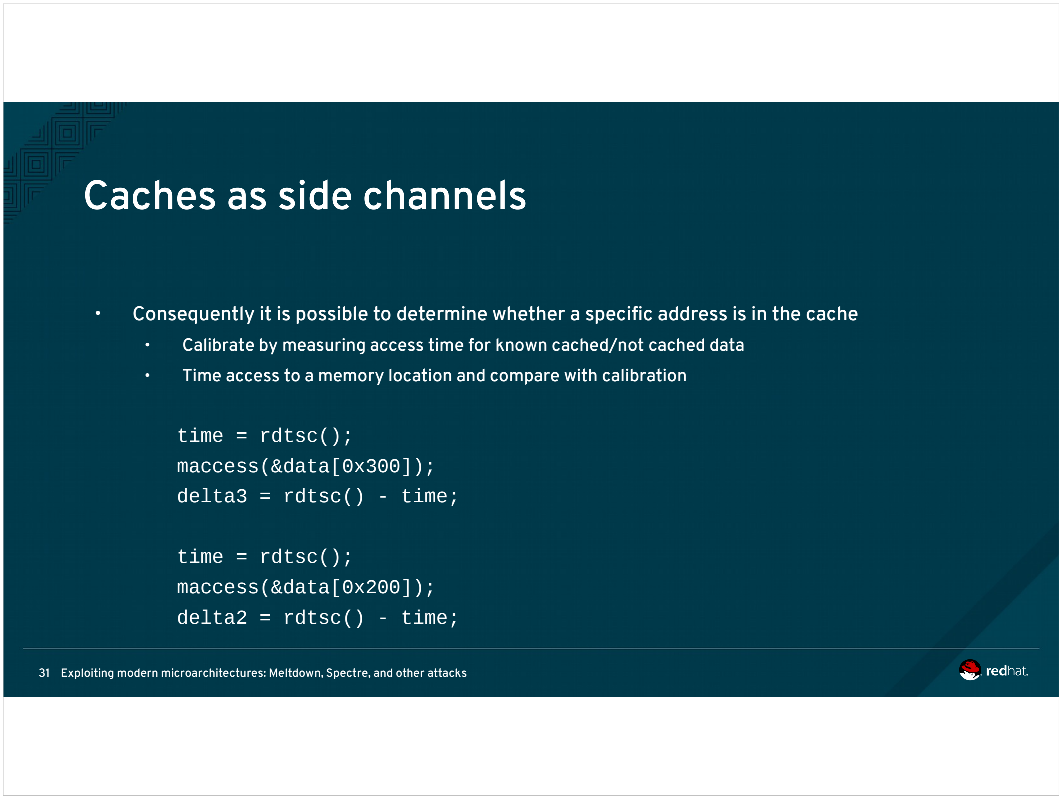
Caches

- Cache entries are located using a combination of indexes and tags
  - Index and tag pages are formed from a given address address
  - The index locates the set that may contain blocks for an address
  - Each entry of the set is checked using the tag for an address match
- Caches may use virtual or physical memory addresses, or a combination
  - Fully virtual caches can result in homonyms for identical physical addresses
  - Fully physical caches can be much slower as they must use translated addresses
- A common optimization is to use VIPT (Virtually Indexed, Physically Tagged)
  - VIPT caches search index using the low order (page offset, e.g. 12) bits of a VA
  - Meanwhile the core finds the PA from the MMU/TLB and supplies to tag compare



Caches as side channels

- Caches exist fundamentally because they provide faster access to frequently used data
    - The closer data is to the compute cores, the less time is required to load it when needed
- This difference in access time for a given address can be measured by software
    - Data closer to the cores will take fewer cycles to access
    - Data further away from the cores will take more cycles to access
- Consequently it is possible to determine whether a specific address is in the cache
    - Calibrate by measuring access time for known cached/not cached data
    - Time access to a memory location and compare with calibration



Caches as side channels

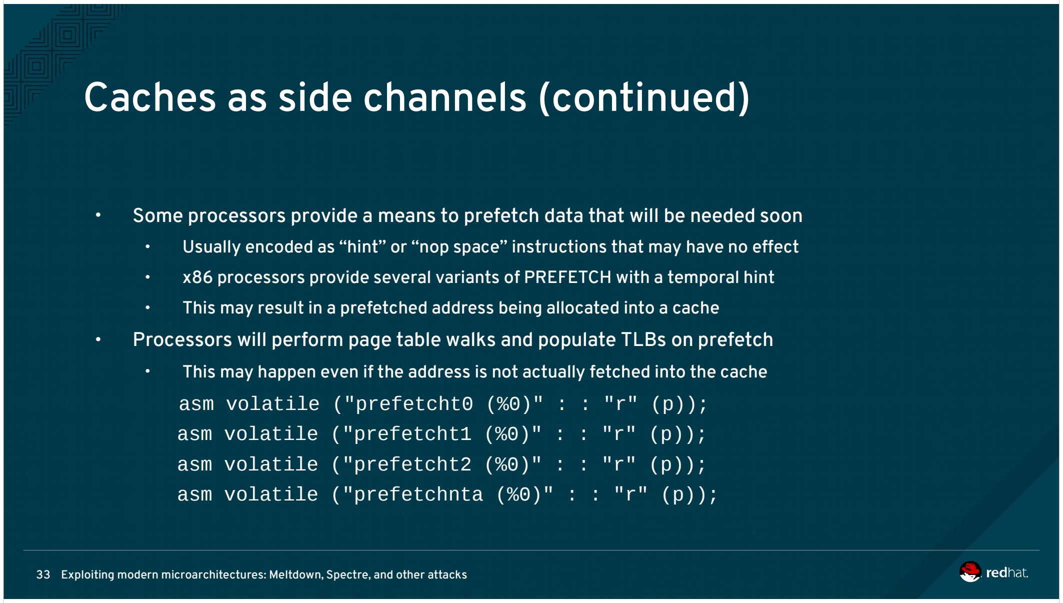
- Consequently it is possible to determine whether a specific address is in the cache
  - Calibrate by measuring access time for known cached/not cached data
  - Time access to a memory location and compare with calibration

```
time = rdtsc();
maccess(&data[0x300]);
delta3 = rdtsc() - time;

time = rdtsc();
maccess(&data[0x200]);
delta2 = rdtsc() - time;
```

Caches as side channels (continued)

- Many instruction sets provide convenient high resolution cycle-accurate timers
  - e.g. x86 provides RDTSC (Read Time Stamp Counter) and RDTSCP instructions
- But there are other ways to measure cycles for architectures without an unprivileged TSC
- Some instruction sets (e.g. x86) also provide convenient unprivileged cache flush instructions
  - CLFLUSH guarantees that a given (virtual) address is not present in any level of cache
- But possible to also flush using a “displacement” approach on other arches
  - Create data structure the size of cache and access entry mapping to desired cache line
- On x86 the time for a flush is proportionate to whether the data was in the cache
  - flush+flush attack determines whether an entry was cached without doing a load
  - Harder to detect using CPU performance counter hardware (measuring cache misses)



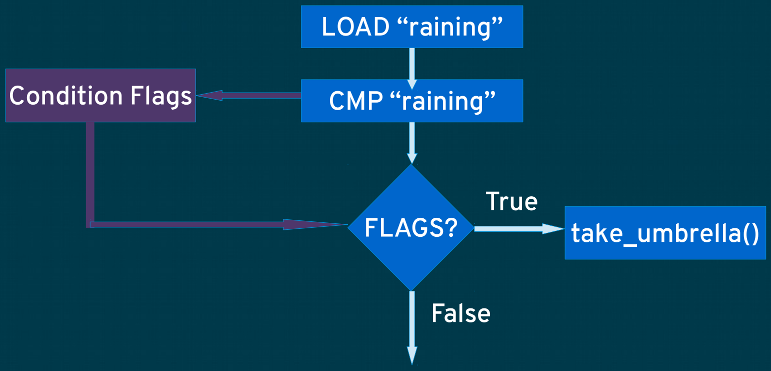
Caches as side channels (continued)

- Some processors provide a means to prefetch data that will be needed soon
  - Usually encoded as “hint” or “nop space” instructions that may have no effect
  - x86 processors provide several variants of PREFETCH with a temporal hint
  - This may result in a prefetched address being allocated into a cache
- Processors will perform page table walks and populate TLBs on prefetch
  - This may happen even if the address is not actually fetched into the cache

```
asm volatile ("prefetcht0 (%0)" : : "r" (p));
asm volatile ("prefetcht1 (%0)" : : "r" (p));
asm volatile ("prefetcht2 (%0)" : : "r" (p));
asm volatile ("prefetchnta (%0)" : : "r" (p));
```

Branch Prediction and Speculation

Branch prediction



LOAD “raining”

CMP “raining”

Condition Flags

FLAGS?

True

take_umbrella()

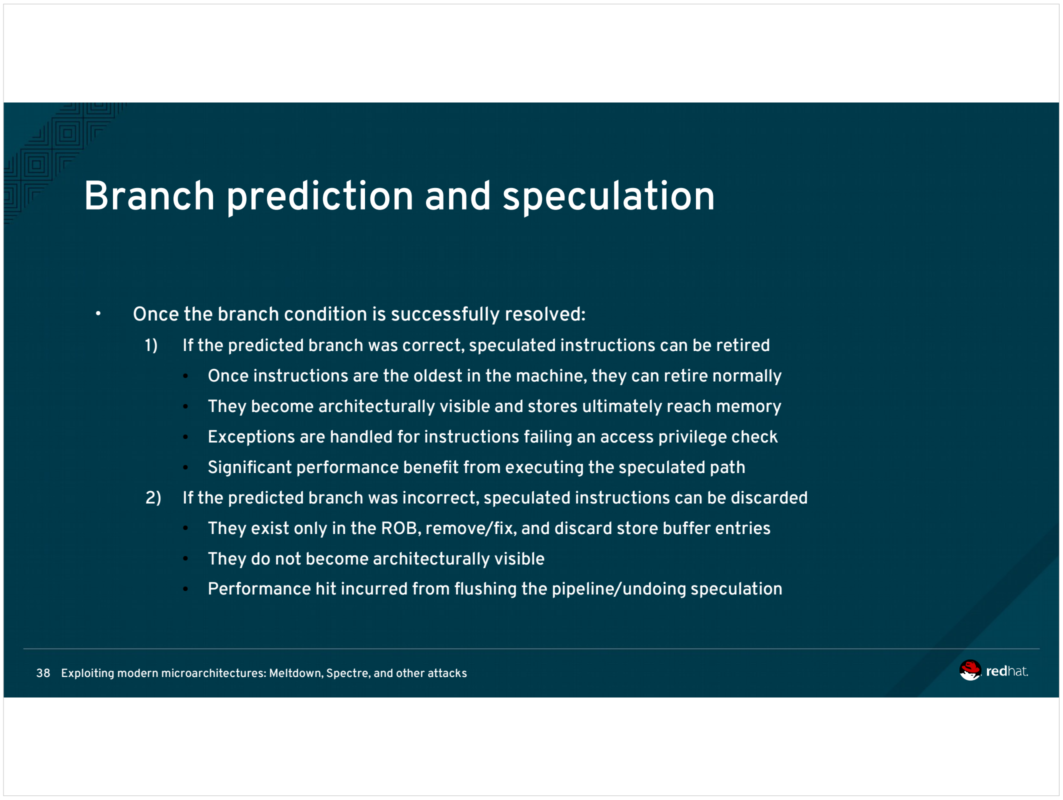
False

Branch prediction

- Applications frequently use program control flow instructions (branches)
  - Conditionals such as “if then” are implemented as conditional direct branches
  - e.g. “if (raining) pack_umbrella();” depends upon the value of “raining”
- Branch condition evaluation is known as “resolving” the branch condition
  - This might require (slow) loads from memory (e.g. not immediately in the L1 D$)
- Rather than wait for branch resolution, predict outcome of the branch
  - This keeps the pipeline(s) filled with (hopefully) useful instructions
- Some ISAs allow compile-time “hints” to be provided for branches
  - These are encoded into the branch instruction, but may not be used
  - “if (likely(condition))” sequences in Operating System kernels

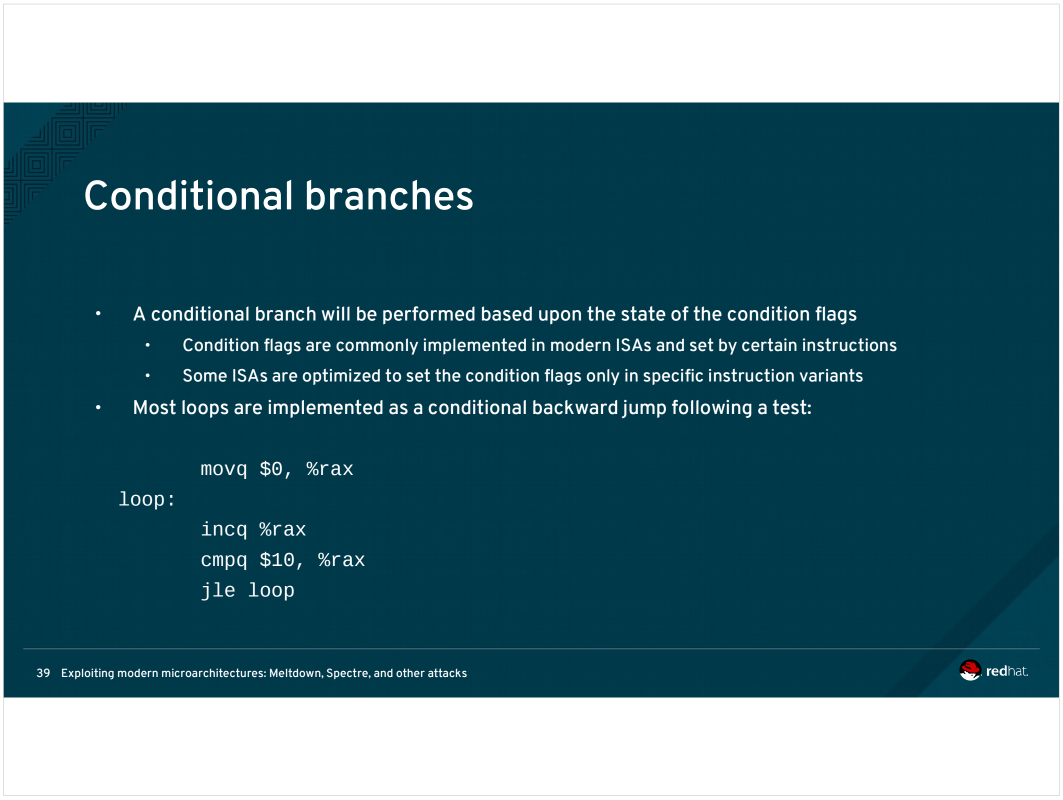
Speculative Execution

- Speculative Execution is implemented as a variation of Out-of-Order Execution
  - Uses the same underlying structures already present such as ROB, etc.
- Instructions that are speculative are specially tagged in the Re-Order Buffer
  - They must not have an architecturally visible effect on the machine state
  - Do not update the architectural register file until speculation is committed
  - Stores to memory are tagged in the ROB and will not hit the store buffers
- Exceptions caused by instructions will not be raised until instruction retirement
  - Tag the ROB to indicate an exception (e.g. privilege check violation on load)
  - If the instruction never retires, then no exception handling is invoked



Branch prediction and speculation

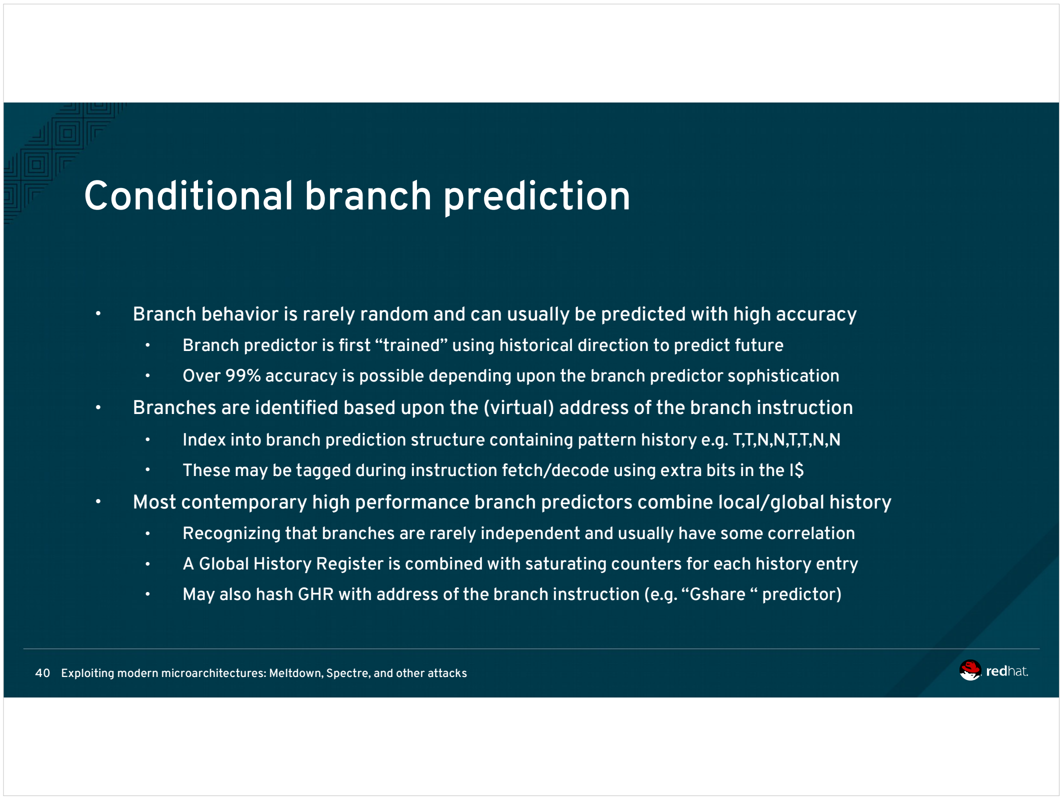
- Once the branch condition is successfully resolved:
  1) If the predicted branch was correct, speculated instructions can be retired
     - Once instructions are the oldest in the machine, they can retire normally
     - They become architecturally visible and stores ultimately reach memory
     - Exceptions are handled for instructions failing an access privilege check
     - Significant performance benefit from executing the speculated path
  2) If the predicted branch was incorrect, speculated instructions can be discarded
     - They exist only in the ROB, remove/fix, and discard store buffer entries
     - They do not become architecturally visible
     - Performance hit incurred from flushing the pipeline/undoing speculation



Conditional branches

- A conditional branch will be performed based upon the state of the condition flags
  - Condition flags are commonly implemented in modern ISAs and set by certain instructions
  - Some ISAs are optimized to set the condition flags only in specific instruction variants
- Most loops are implemented as a conditional backward jump following a test:

```
       movq $0, %rax
loop:
       incq %rax
       cmpq $10, %rax
       jle loop
```

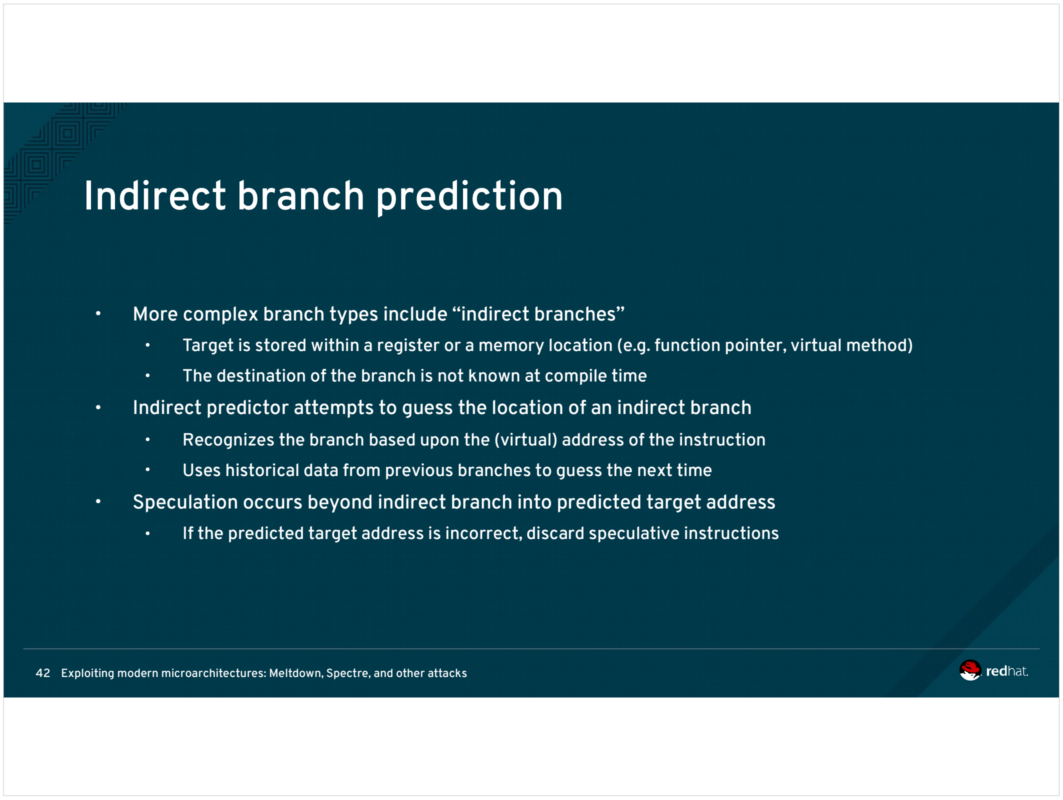


Conditional branch prediction

- Branch behavior is rarely random and can usually be predicted with high accuracy
  - Branch predictor is first “trained” using historical direction to predict future
  - Over 99% accuracy is possible depending upon the branch predictor sophistication
- Branches are identified based upon the (virtual) address of the branch instruction
  - Index into branch prediction structure containing pattern history e.g. T,T,N,N,T,T,N,N
  - These may be tagged during instruction fetch/decode using extra bits in the I$
- Most contemporary high performance branch predictors combine local/global history
  - Recognizing that branches are rarely independent and usually have some correlation
  - A Global History Register is combined with saturating counters for each history entry
  - May also hash GHR with address of the branch instruction (e.g. “Gshare “ predictor)

Conditional branch prediction

- Modern designs combine various different branch predictors
  - Simple loop predictors include BTFN (Backward Taken Forward Not)
  - Contemporary processors would identify the previous example as early as decode
  - May directly issue repeated loop instructions from pre-decoded instruction cache
- Optimize size of predictor internal structures by hashing/indexing on address
  - Common not to use the full address of a branch instruction in the history table
  - This causes some level of (known) interference between unrelated branches

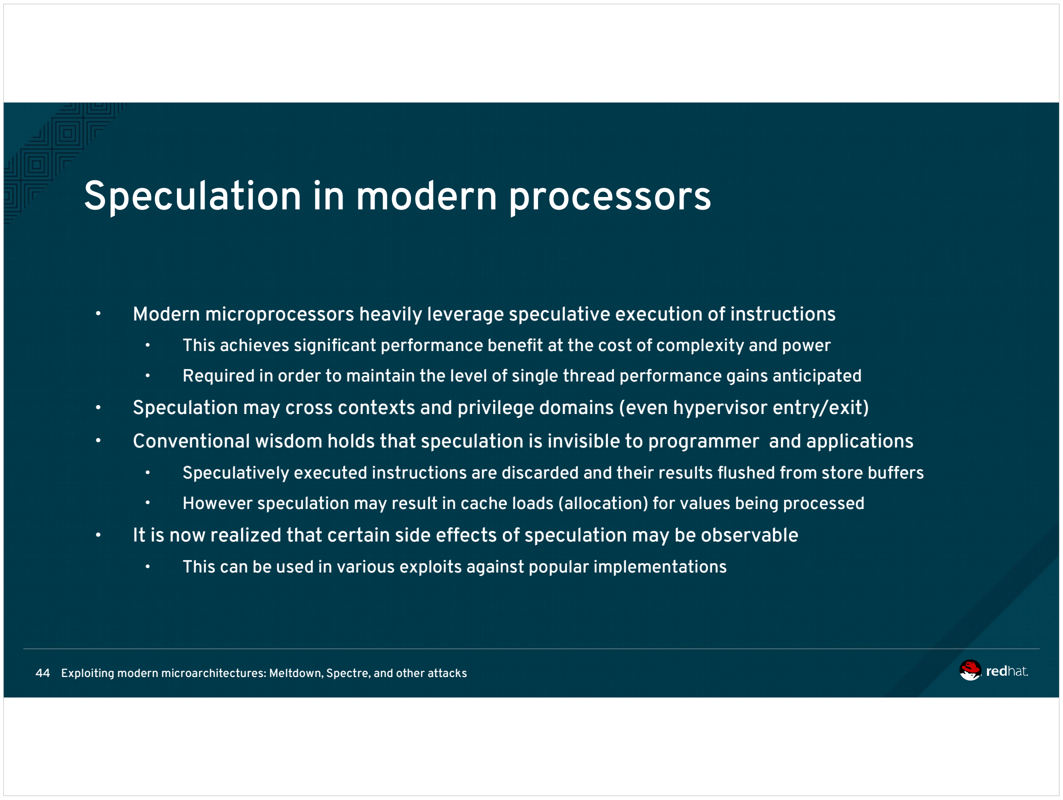


Indirect branch prediction

- More complex branch types include “indirect branches”
  - Target is stored within a register or a memory location (e.g. function pointer, virtual method)
  - The destination of the branch is not known at compile time
- Indirect predictor attempts to guess the location of an indirect branch
  - Recognizes the branch based upon the (virtual) address of the instruction
  - Uses historical data from previous branches to guess the next time
- Speculation occurs beyond indirect branch into predicted target address
  - If the predicted target address is incorrect, discard speculative instructions

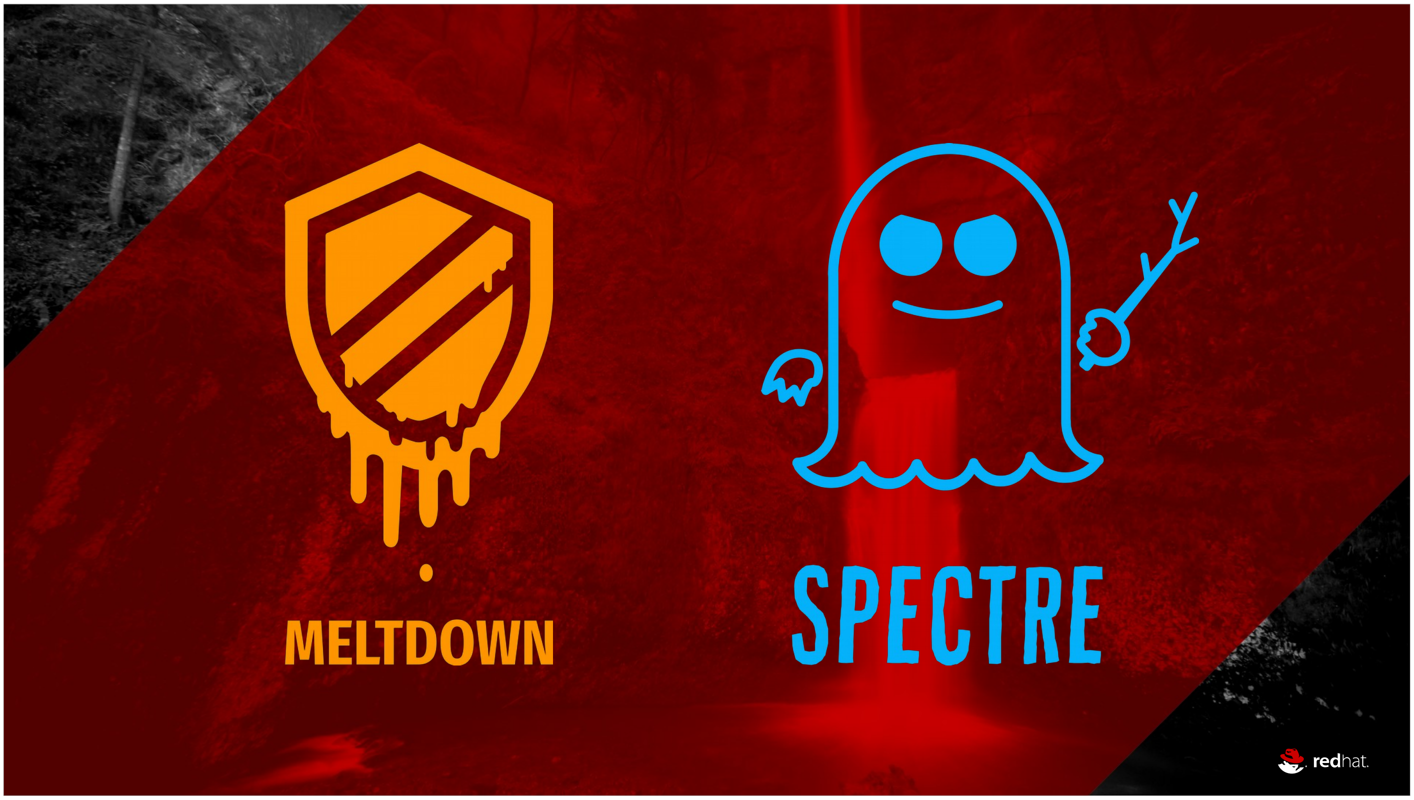
Branch predictor optimization

- Branch prediction is vital to modern microprocessor performance
    - Significant research has gone into optimization of prediction algorithms
    - Many different predictors may be in use simultaneously with voting arbitration
    - Accuracy rates of over 99% are possible depending upon the workload
- Predictors are in the critical path for instruction fetch/decode
    - Must operate quickly to prevent adding delays to instruction dispatch
    - Common industry optimizations aimed at reducing predictor storage
    - Optimizations include indexing on low order address bits of branches



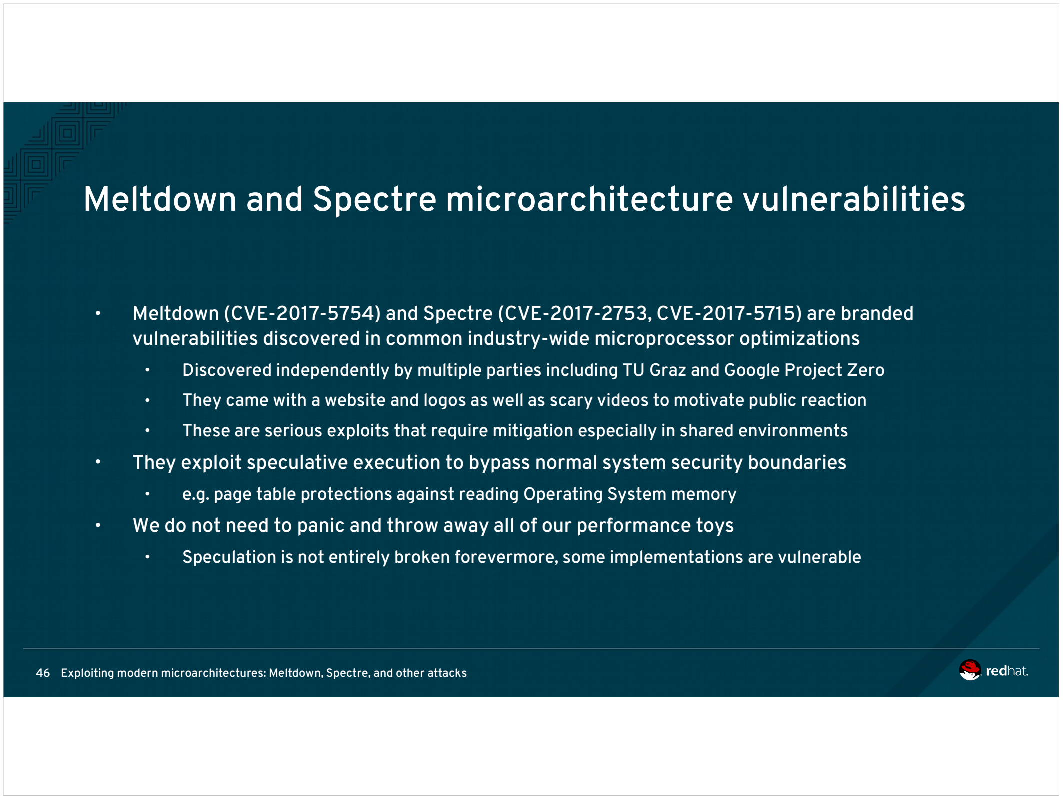
Speculation in modern processors

- Modern microprocessors heavily leverage speculative execution of instructions
  - This achieves significant performance benefit at the cost of complexity and power
  - Required in order to maintain the level of single thread performance gains anticipated
- Speculation may cross contexts and privilege domains (even hypervisor entry/exit)
- Conventional wisdom holds that speculation is invisible to programmer and applications
  - Speculatively executed instructions are discarded and their results flushed from store buffers
  - However speculation may result in cache loads (allocation) for values being processed
- It is now realized that certain side effects of speculation may be observable
  - This can be used in various exploits against popular implementations



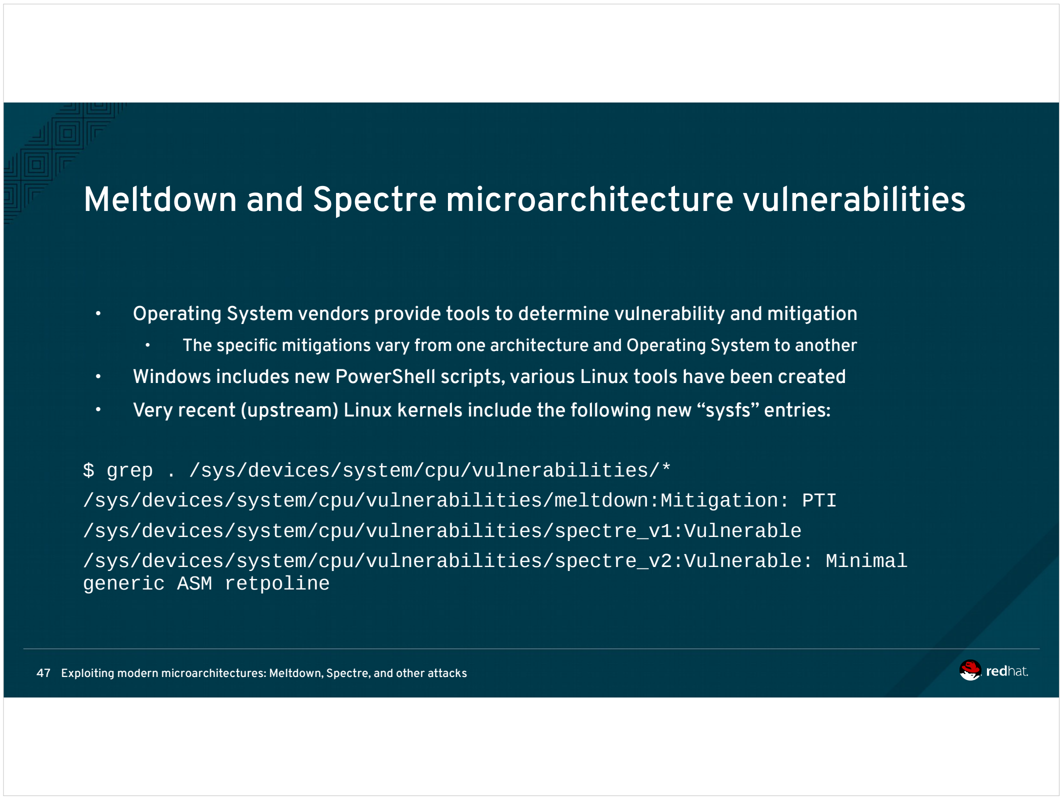
MELTDOWN

SPECTRE



Meltdown and Spectre microarchitecture vulnerabilities

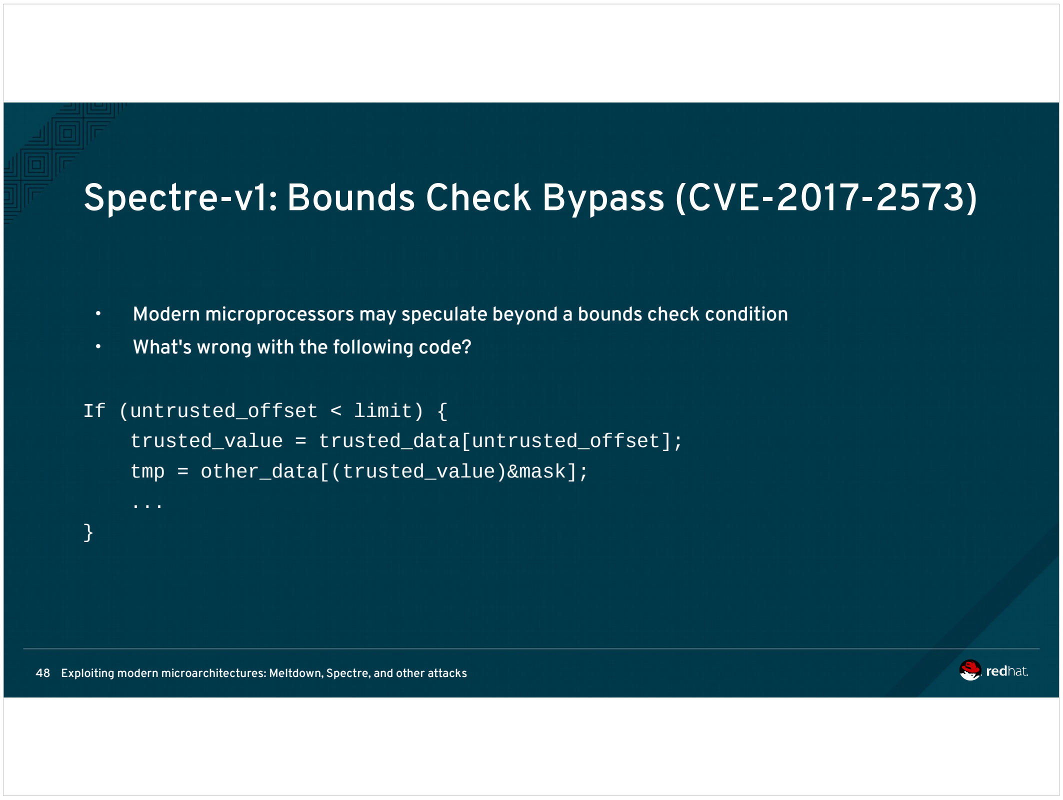
- Meltdown (CVE-2017-5754) and Spectre (CVE-2017-2753, CVE-2017-5715) are branded vulnerabilities discovered in common industry-wide microprocessor optimizations
  - Discovered independently by multiple parties including TU Graz and Google Project Zero
  - They came with a website and logos as well as scary videos to motivate public reaction
  - These are serious exploits that require mitigation especially in shared environments
- They exploit speculative execution to bypass normal system security boundaries
  - e.g. page table protections against reading Operating System memory
- We do not need to panic and throw away all of our performance toys
  - Speculation is not entirely broken forevermore, some implementations are vulnerable



Meltdown and Spectre microarchitecture vulnerabilities

- Operating System vendors provide tools to determine vulnerability and mitigation
  - The specific mitigations vary from one architecture and Operating System to another
- Windows includes new PowerShell scripts, various Linux tools have been created
- Very recent (upstream) Linux kernels include the following new “sysfs” entries:

```
$ grep . /sys/devices/system/cpu/vulnerabilities/*
/sys/devices/system/cpu/vulnerabilities/meltdown:Mitigation: PTI
/sys/devices/system/cpu/vulnerabilities/spectre_v1:Vulnerable
/sys/devices/system/cpu/vulnerabilities/spectre_v2:Vulnerable: Minimal
generic ASM retpoline
```



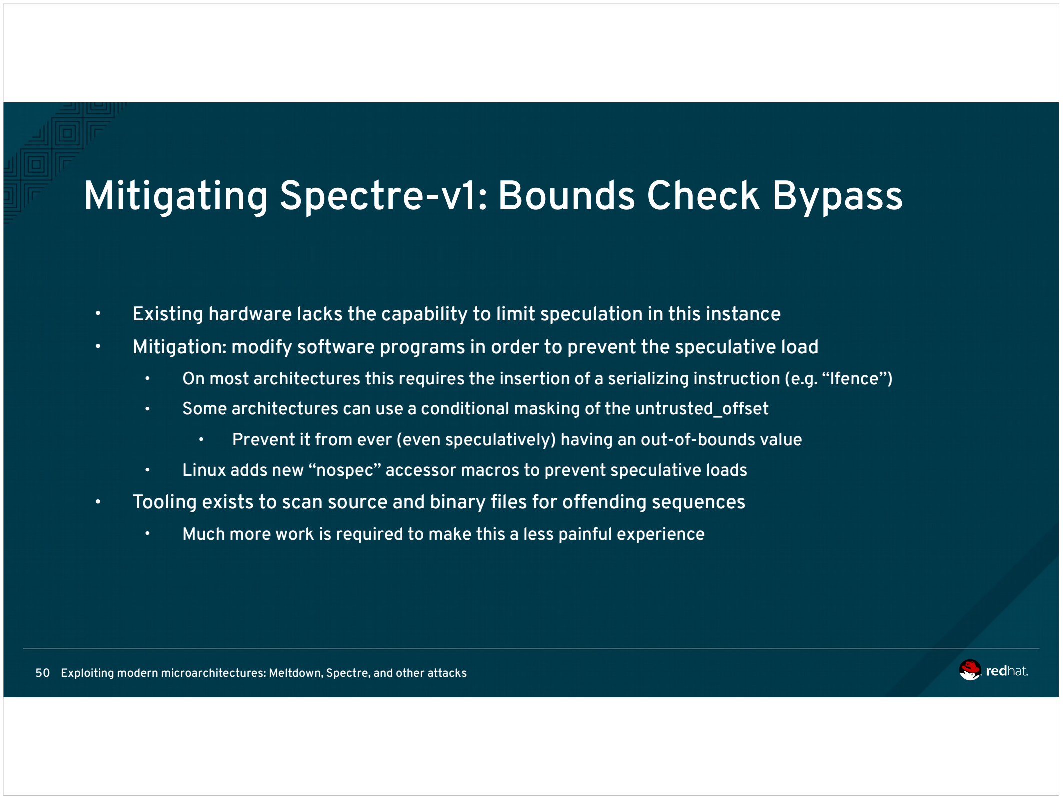
Spectre-v1: Bounds Check Bypass (CVE-2017-2573)

- Modern microprocessors may speculate beyond a bounds check condition
- What's wrong with the following code?

```
If (untrusted_offset < limit) {
    trusted_value = trusted_data[untrusted_offset];
    tmp = other_data[(trusted_value)&mask];
    ...
}
```

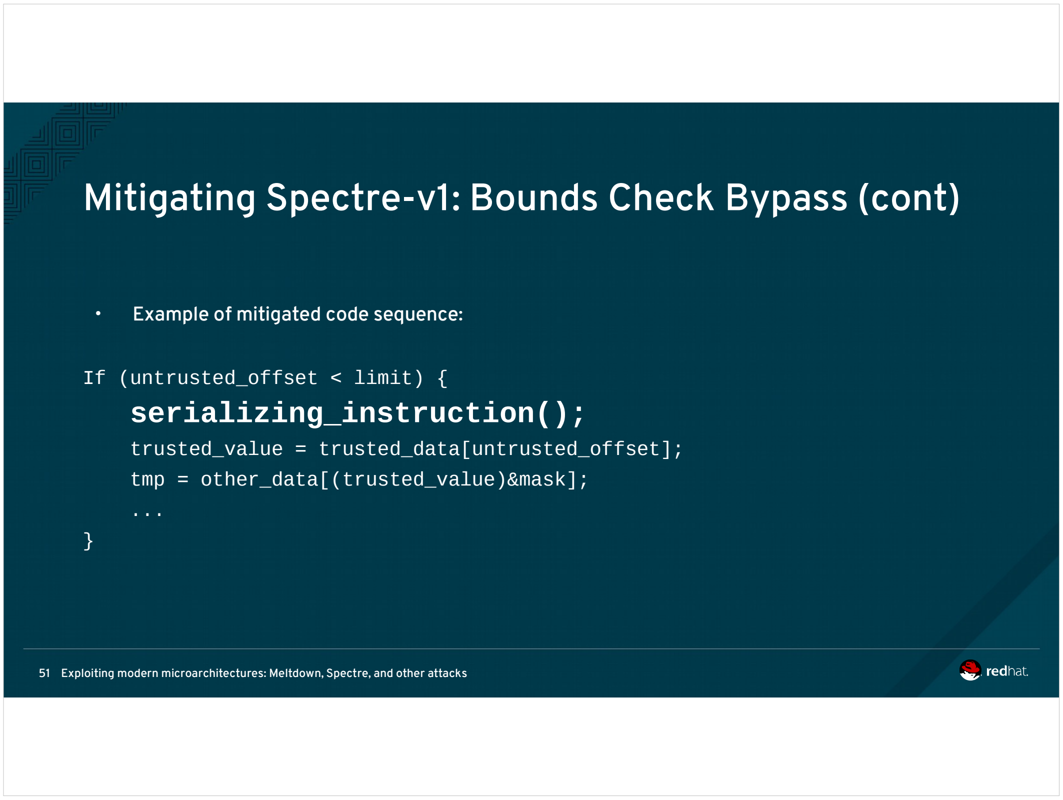
Spectre-v1: Bounds Check Bypass (cont)

- The code following the bounds check is known as a “gadget” (see ROP attacks)
    - Existing code contained within a different victim context (e.g. Operating System/Hypervisor)
- Code following the untrusted_offset bounds check may be executed speculatively
    - Resulting in the speculative loading of trusted data into a local variable
    - This trusted data is used to calculate an offset into another structure
- Relative offset of other_data accessed can be used to infer trusted_value
    - L1D$ cache load will occur for other_data at an offset correlated with trusted_value
    - Measure which cache location was loaded speculatively to infer the secret value



Mitigating Spectre-v1: Bounds Check Bypass

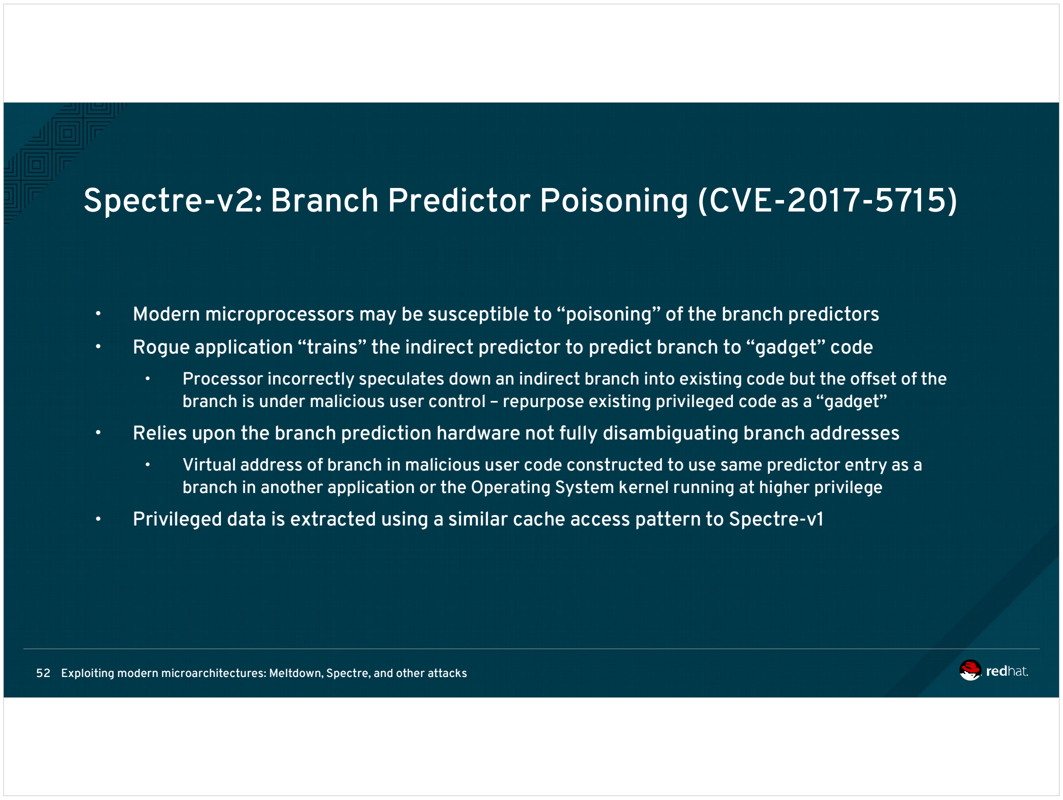
- Existing hardware lacks the capability to limit speculation in this instance
- Mitigation: modify software programs in order to prevent the speculative load
  - On most architectures this requires the insertion of a serializing instruction (e.g. “lfence”)
  - Some architectures can use a conditional masking of the untrusted_offset
    - Prevent it from ever (even speculatively) having an out-of-bounds value
  - Linux adds new “nospec” accessor macros to prevent speculative loads
- Tooling exists to scan source and binary files for offending sequences
  - Much more work is required to make this a less painful experience



Mitigating Spectre-v1: Bounds Check Bypass (cont)

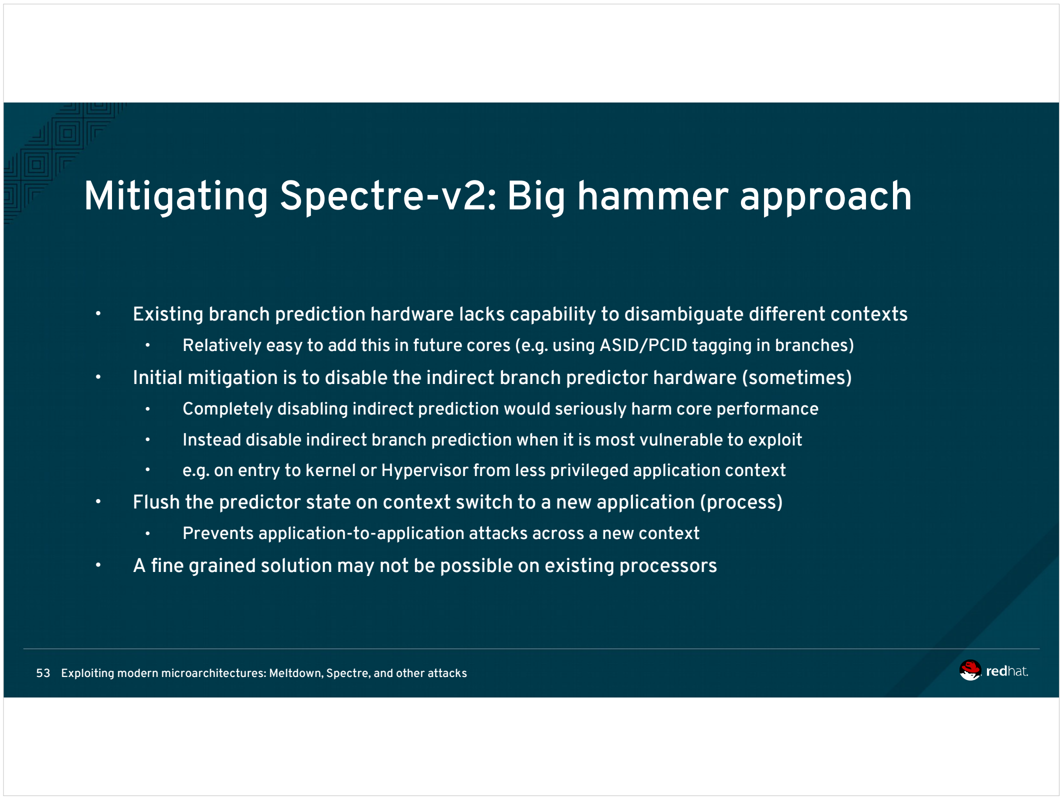
- Example of mitigated code sequence:

```
If (untrusted_offset < limit) {
    serializing_instruction();
    trusted_value = trusted_data[untrusted_offset];
    tmp = other_data[(trusted_value)&mask];
    ...
}
```



Spectre-v2: Branch Predictor Poisoning (CVE-2017-5715)

- Modern microprocessors may be susceptible to “poisoning” of the branch predictors
- Rogue application “trains” the indirect predictor to predict branch to “gadget” code
  - Processor incorrectly speculates down an indirect branch into existing code but the offset of the branch is under malicious user control – repurpose existing privileged code as a “gadget”
- Relies upon the branch prediction hardware not fully disambiguating branch addresses
  - Virtual address of branch in malicious user code constructed to use same predictor entry as a branch in another application or the Operating System kernel running at higher privilege
- Privileged data is extracted using a similar cache access pattern to Spectre-v1



Mitigating Spectre-v2: Big hammer approach

- Existing branch prediction hardware lacks capability to disambiguate different contexts
  - Relatively easy to add this in future cores (e.g. using ASID/PCID tagging in branches)
- Initial mitigation is to disable the indirect branch predictor hardware (sometimes)
  - Completely disabling indirect prediction would seriously harm core performance
  - Instead disable indirect branch prediction when it is most vulnerable to exploit
  - e.g. on entry to kernel or Hypervisor from less privileged application context
- Flush the predictor state on context switch to a new application (process)
  - Prevents application-to-application attacks across a new context
- A fine grained solution may not be possible on existing processors

Tangent: Microcode, Millicode, and Chicken Bits

- Modern microprocessors are extremely complex machines requiring huge capital investment
  - A high performance core might require a 300+ person team, and 4 years of engineering effort
- Consequently the ability to handle potential issues in the field is extremely compelling
- Modern cores provide thousands of hidden tunable knobs (chicken bits) that allow a design team to “chicken out” and disable certain features that aren't working in whole or in part
  - A high performance core might have as many as 10,000 different chicken bits available
- A chicken bit might be programmed in firmware prior to system boot
  - e.g. “disable all indirect branch prediction when in privileged state” (if this is possible)
- Or it might be exposed to the Operating System to poke it as needed

Microcode, Millicode, and Chicken Bits (cont)

- Some processors contain a mix of hardwired logic and microcoded instructions
  - Typically used for complex instructions in CISC architectures such as x86, POWER, Z/Arch…
  - Not used in the fast path for critical core logic (such as caches, page table walks, etc.)
- Microcode defines control signals within the core and state transitions between them
  - A microcode sequencer (simple state machine) within the core sets control signals following a “program” (really a simple set of state transitions) contained within fast on-chip ROM
  - Example is repeated instructions in x86 which can be implemented in microcode sequences
- A small (a few KB) microcode patch RAM can be used to patch some instruction behavior
  - Microcode is an encrypted, signed blob from the CPU manufacturer, format is (mostly) secret
  - Operating Systems or firmware can load microcode/millicode at system boot time or later

Mitigating Spectre-v2: Big hammer (cont)

- Microcode can be used on some microprocessors to alter instruction behavior
- It can also be used add new “instructions” or system registers that exhibit side effects
- On Spectre-v2 impacted x86 microprocessors, microcode adds new SPEC_CTRL MSRs
  - Model Specific Registers are special memory addresses that control core behavior
  - Identified using the x86 “CPUID” instruction which enumerates available capabilities
  - IBRS (Indirect Branch Restrict Speculation)
    - Used on entry to more privileged context to restrict branch speculation
  - IBPB (Indirect Branch Predictor Barrier)
    - Used on context switch into a new process to flush predictor entries
- What are the problems with using microcode interfaces?

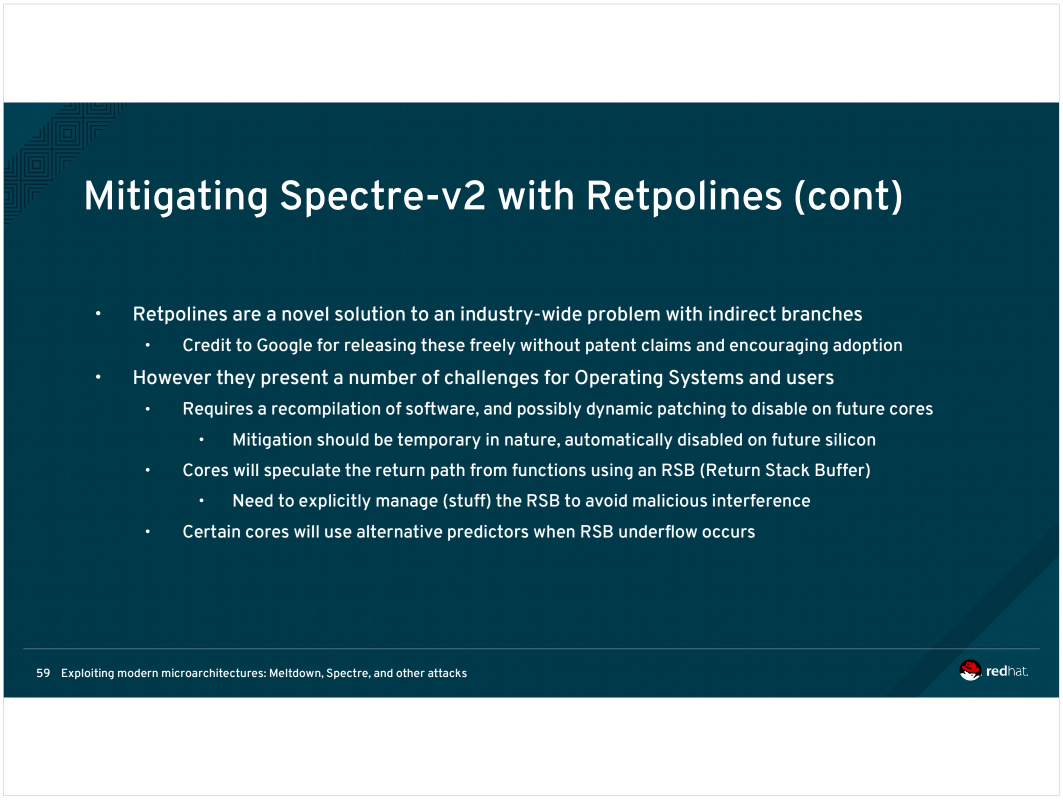
Mitigating Spectre-v2 with Retpolines

- Microcoded mitigations are effective but expensive due to their implementation
  - Many cores do not have convenient logic to disable predictors so “IBRS” must also disable independent logic within the core. It may take many thousands of cycles on kernel entry
- Google decided to try an alternative solution using a pure software approach
  - If indirect branches are the problem, then the solution is to avoid using them
  - “Retpolines” stand for “Return Trampolines” which replace indirect branches
  - Setup a fake function call stack and “return” in place of the indirect call

Mitigating Spectre with Retpolines (cont)

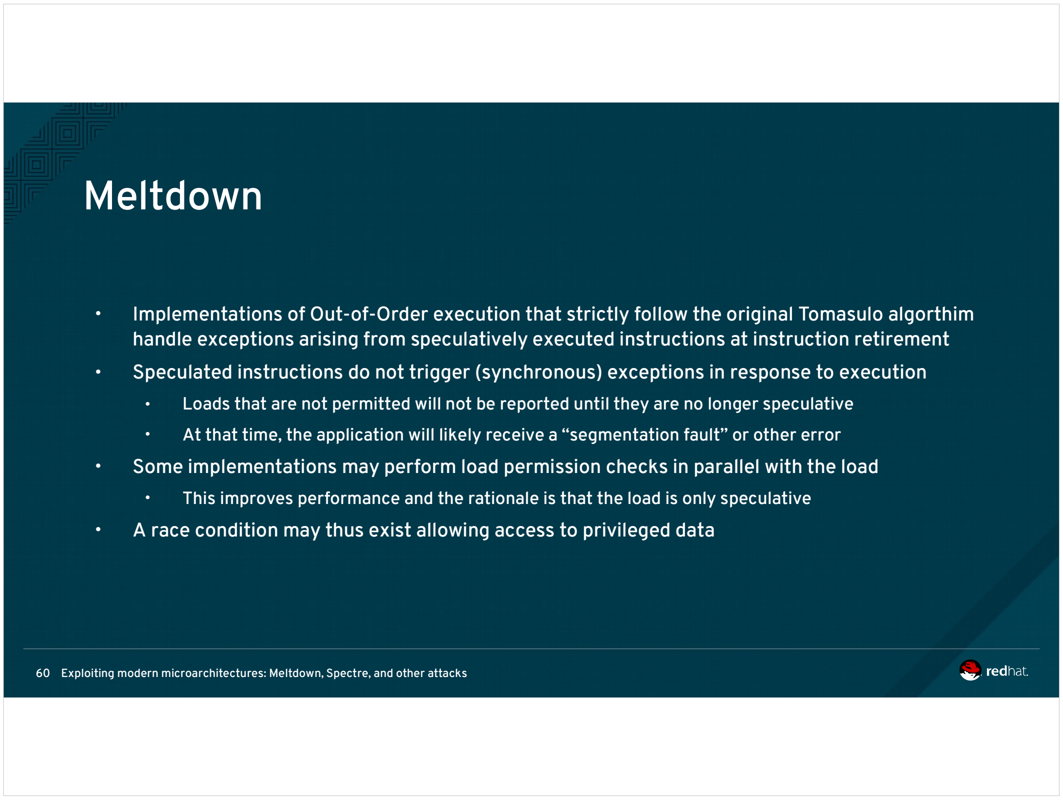
- Example retpoline call sequence on x86 (source: https://support.google.com/faqs/answer/7625886)

```
  call set_up_target;
capture_spec:
  pause;
  jmp capture_spec;
set_up_target:
  mov %r11, (%rsp);
  ret;
```



Mitigating Spectre-v2 with Retpolines (cont)

- Retpolines are a novel solution to an industry-wide problem with indirect branches
  - Credit to Google for releasing these freely without patent claims and encouraging adoption
- However they present a number of challenges for Operating Systems and users
  - Requires a recompilation of software, and possibly dynamic patching to disable on future cores
    - Mitigation should be temporary in nature, automatically disabled on future silicon
  - Cores will speculate the return path from functions using an RSB (Return Stack Buffer)
    - Need to explicitly manage (stuff) the RSB to avoid malicious interference
  - Certain cores will use alternative predictors when RSB underflow occurs



Meltdown

- Implementations of Out-of-Order execution that strictly follow the original Tomasulo algorthim handle exceptions arising from speculatively executed instructions at instruction retirement
- Speculated instructions do not trigger (synchronous) exceptions in response to execution
  - Loads that are not permitted will not be reported until they are no longer speculative
  - At that time, the application will likely receive a “segmentation fault” or other error
- Some implementations may perform load permission checks in parallel with the load
  - This improves performance and the rationale is that the load is only speculative
- A race condition may thus exist allowing access to privileged data

Meltdown (continued)

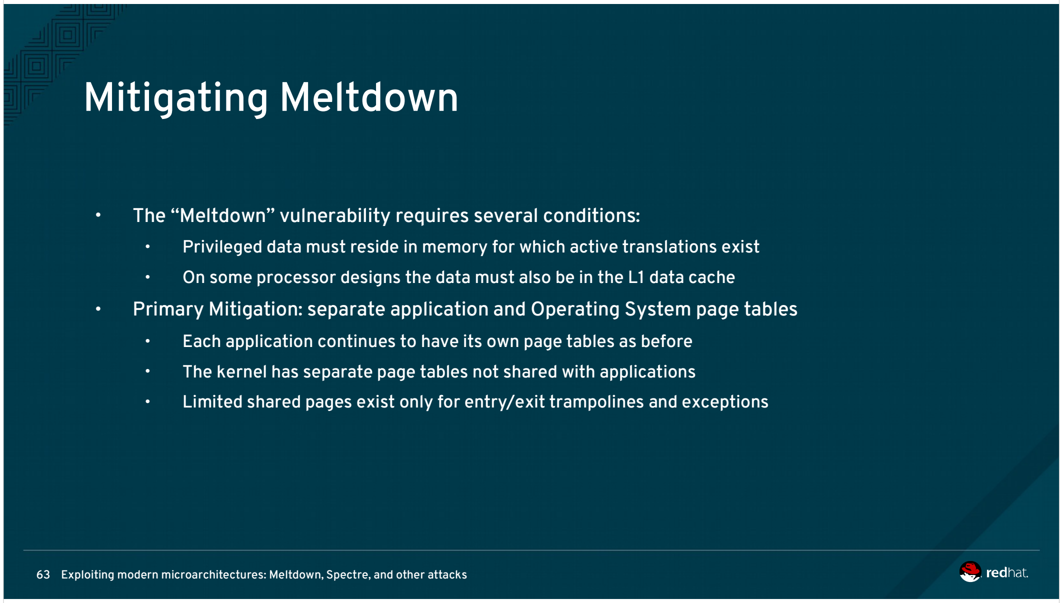
- A malicious attacker arranges for exploit code similar to the following to speculatively execute:

```
if (spec_cond) {
            unsigned char value = *(unsigned char *)ptr;
            unsigned long index2 = (((value>>bit)&1)*0x100)+0x200;
            maccess(&data[index2]);
 }
```

- “data” is a user controller array to which the attacker has access, “ptr” contains privileged data

Meltdown (continued)

- When the right conditions exist, this branch of code will run speculatively
  - The privilege check for “value” will fail, but only result in an entry tag in the ROB
  - The access will occur although “value” will be discarded when speculation is undone
- The offset accessed in the “data” user array is dependent upon the value of privileged data
  - We can use this as a 1 bit counter between several possible entries of the user data array
- Cache side channel timing analysis is used to measure which “data” location was accessed
  - Time access to “data” locations 0x200 and 0x300 to infer value of desired bit
  - Access is done in reverse in my code to account for cache line prefetcher



Mitigating Meltdown

- The “Meltdown” vulnerability requires several conditions:
  - Privileged data must reside in memory for which active translations exist
  - On some processor designs the data must also be in the L1 data cache
- Primary Mitigation: separate application and Operating System page tables
  - Each application continues to have its own page tables as before
  - The kernel has separate page tables not shared with applications
  - Limited shared pages exist only for entry/exit trampolines and exceptions

Mitigating Meltdown

- Linux calls this page table separation “PTI”: Page Table Isolation
  - Requires an expensive write to core control registers on every entry/exit from the OS kernel
  - e.g. TTBR write on impacted ARMv8, CR3 on impacted x86 processors
- Only enabled by default on known-vulnerable microprocessors
  - An enumeration is defined to discover future non-impacted silicon
- Address Space IDentifiers (ASIDs) can significantly improve performance
  - ASIDs on ARMv8, PCIDs (Process Context IDs) on x86 processors
  - TLB entries are tagged with address space so a full invalidation isn't required
  - Significant performance delta between older (pre-2010 x86) cores and newer ones

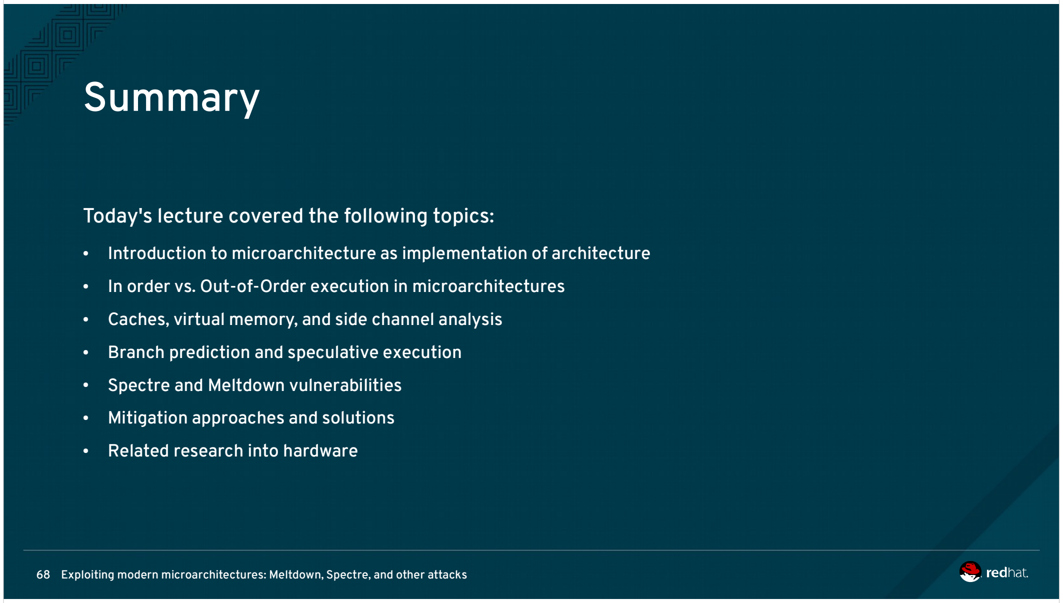
Variations on a theme: variant 3a (Sysreg read)

- Variations of these microarchitecture attacks are likely to be found for many years
- An example is known as “variant 3a”. Some microprocessors will allow speculative read of privileged system registers to which an application should not normally have access
  - Can be used to determine the address of key structures such as page table base registers

Related Research

- Meltdown and Spectre are only recent examples of microarchitecture attack
- A memorable attack known as “Rowhammer” was discovered previously
  - Exploit the implementation of (especially non-ECC) DDR memory
  - Possible to perturb bits in adjacent memory lines with frequent access
  - Can use this approach to flip bits in sensitive memory and bypass access restrictions
  - For example change page access permissions in the system page tables
- Another recent attack known as “MAGIC” exploits NBTI in silicon
  - Negative-bias temperature instability impacts reliability of MOSFETs (“transistors”)
  - Can be exploited to artificially age silicon devices and decrease longevity
  - Proof of concept demonstrated with code running on OpenSPARC core

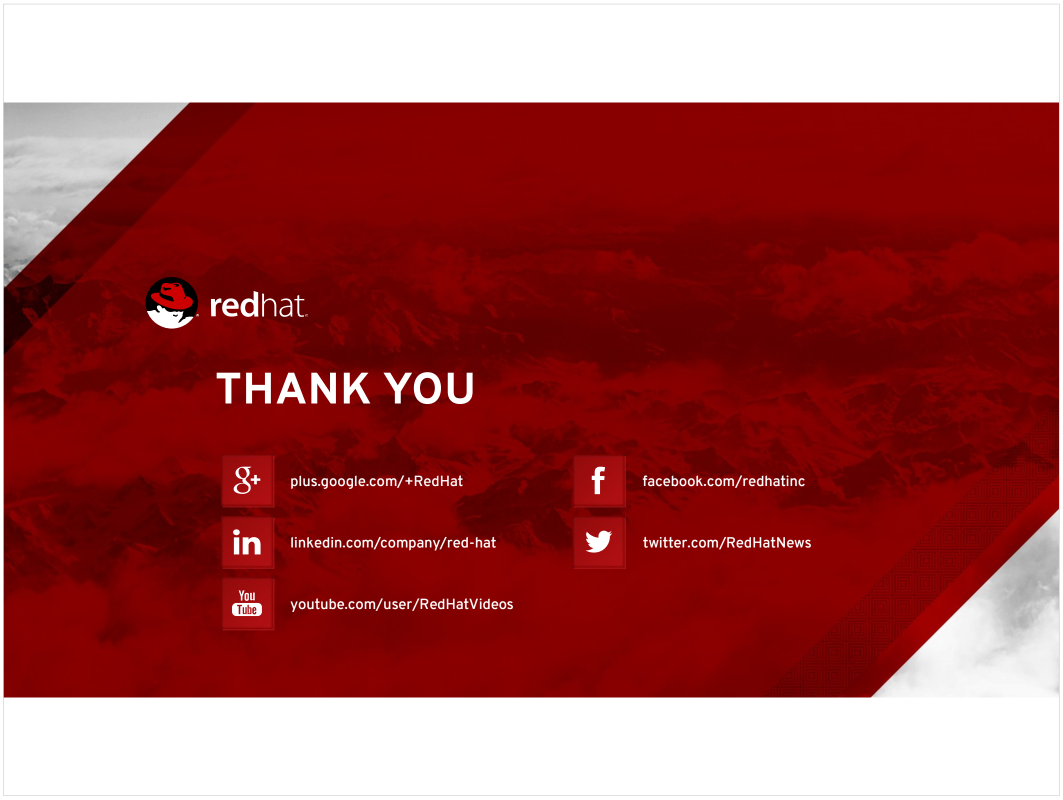
Summary



Summary

Today's lecture covered the following topics:

- Introduction to microarchitecture as implementation of architecture
- In order vs. Out-of-Order execution in microarchitectures
- Caches, virtual memory, and side channel analysis
- Branch prediction and speculative execution
- Spectre and Meltdown vulnerabilities
- Mitigation approaches and solutions
- Related research into hardware



redhat.

THANK YOU

plus.google.com/+RedHat

facebook.com/redhatinc

linkedin.com/company/red-hat

twitter.com/RedHatNews

youtube.com/user/RedHatVideos

redhat®