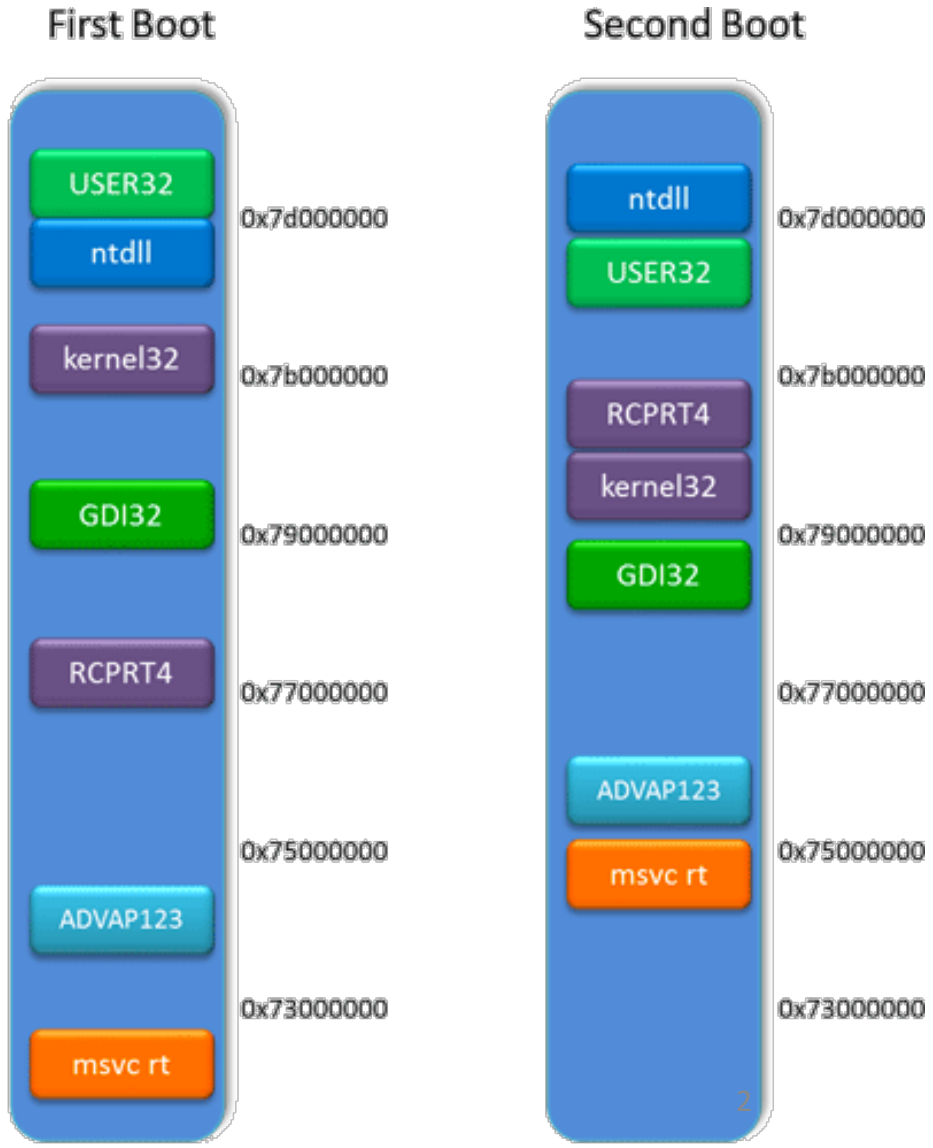
# Breaking Kernel Address Space Layout Randomization (KASLR) with Intel TSX

**Yeongjin Jang**, Sangho Lee, and Taesoo Kim

Georgia Institute of Technology

# Kernel Address Space Layout Randomization (KASLR)

- A statistical mitigation for memory corruption exploits
- Randomize address layout per each boot
  - Efficient (<5% overhead)
- Attacker should guess where code/data are located for exploit.
  - In Windows, a successful guess rate is 1/8192.



# Example: Linux

- To escalate privilege to root through a kernel exploit, attackers want to call commit_creds(prepare_kernel_creds(0)).

```
// full-nelson.c
static int __attribute__((regparm(3)))
getroot(void * file, void * vma)
{
    commit_creds(prepare_kernel_cred(0));
    return -1;
}
// https://blog.plenz.com/2013-02/privilege-escalation-kernel-exploit.html
int privesc(struct sk_buff *skb, struct nlmsghdr *nlh)
{
    commit_creds(prepare_kernel_cred(0));
    return 0;
}
```

# Example: Linux

- KASLR changes kernel symbol addresses every boot.

# Example: Linux

- KASLR changes kernel symbol addresses every boot.

```
[blue9057@pt ~$] sudo cat /proc/kallsyms | grep ' commit_creds\| prepare_kernel'
ffffffffaa0a3bd0 T commit_creds
ffffffffaa0a3fc0 T prepare_kernel_cred
```

1st Boot

# Example: Linux

- KASLR changes kernel symbol addresses every boot.

```
[blue9057@pt ~$] sudo cat /proc/kallsyms | grep ' commit_creds\| prepare_kernel'
ffffffffaa0a3bd0 T commit_creds
ffffffffaa0a3fc0 T prepare_kernel_cred
```

1st Boot

```
[blue9057@pt ~$] sudo cat /proc/kallsyms | grep ' commit_creds\| prepare_kernel'
ffffffffbd0a3bd0 T commit_creds
ffffffffbd0a3fc0 T prepare_kernel_cred
```

2nd Boot

# KASLR Makes Attacks Harder

- KASLR introduces an additional bar to exploits
  - Finding an information leak vulnerability

$$\Pr[\ \exists\ \text{Memory Corruption Vuln}\ ]$$

- Both attackers and defenders aim to detect info leak vulnerabilities.

# KASLR Makes Attacks Harder

- KASLR introduces an additional bar to exploits
  - Finding an information leak vulnerability

$$\Pr[\ \exists\ \text{Memory Corruption Vuln}\ ]$$

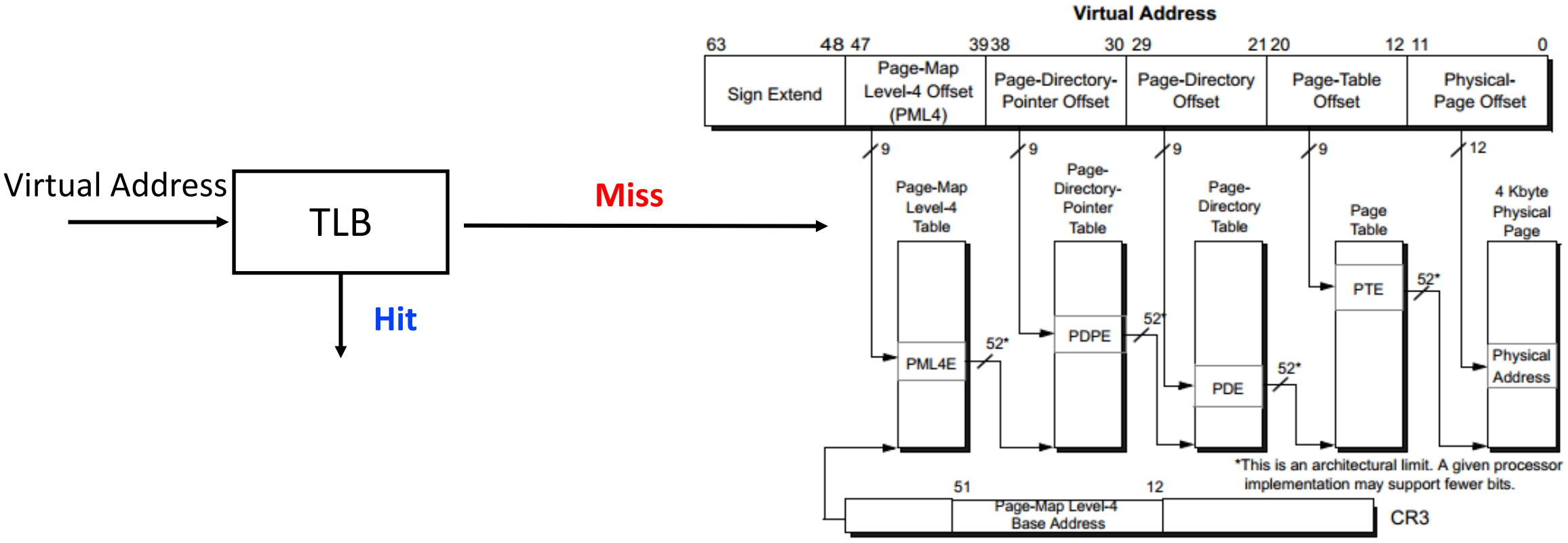
$$\Pr[\ \exists\ \text{information\_leak}\ ] \times \Pr[\ \exists\ \text{Memory Corruption Vuln}]$$

- Both attackers and defenders aim to detect info leak vulnerabilities.

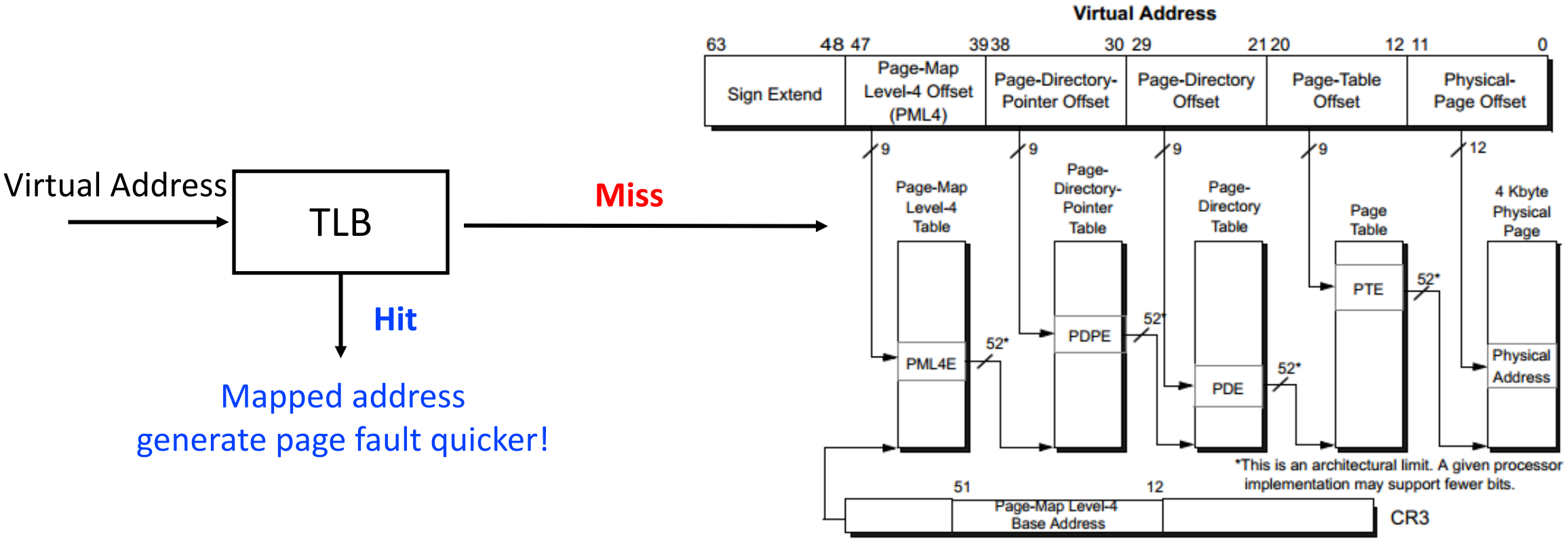
# Is there any other way than info leak?

- Practical Timing Side Channel Attacks Against Kernel Space ASLR (Hund et al., Oakland 2013)
  - A **hardware-level** side channel attack against KASLR
  - **No** information leak vulnerability in OS is required

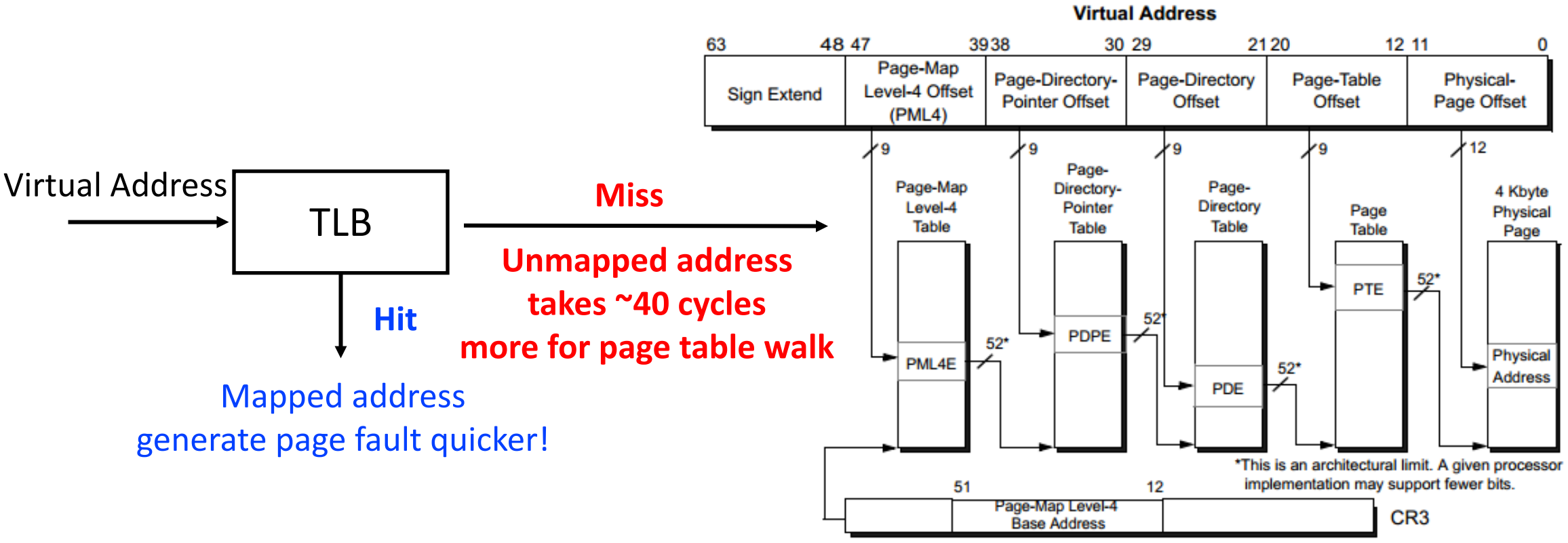
# TLB Timing Side Channel



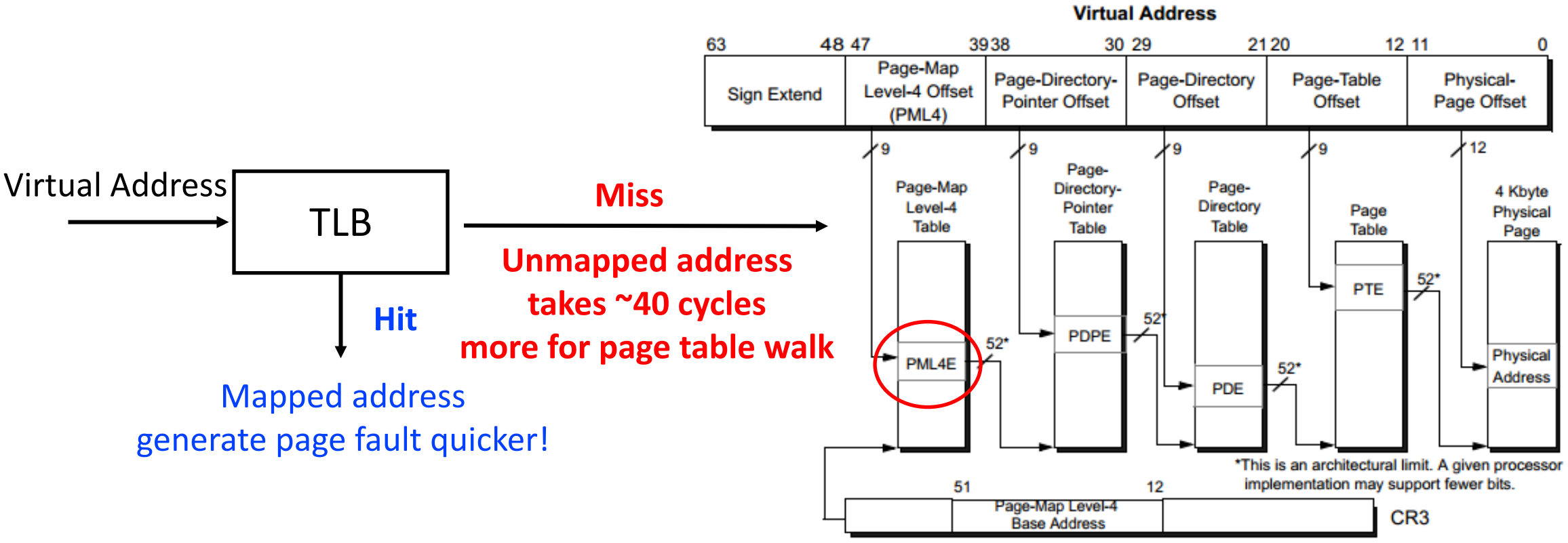
# TLB Timing Side Channel



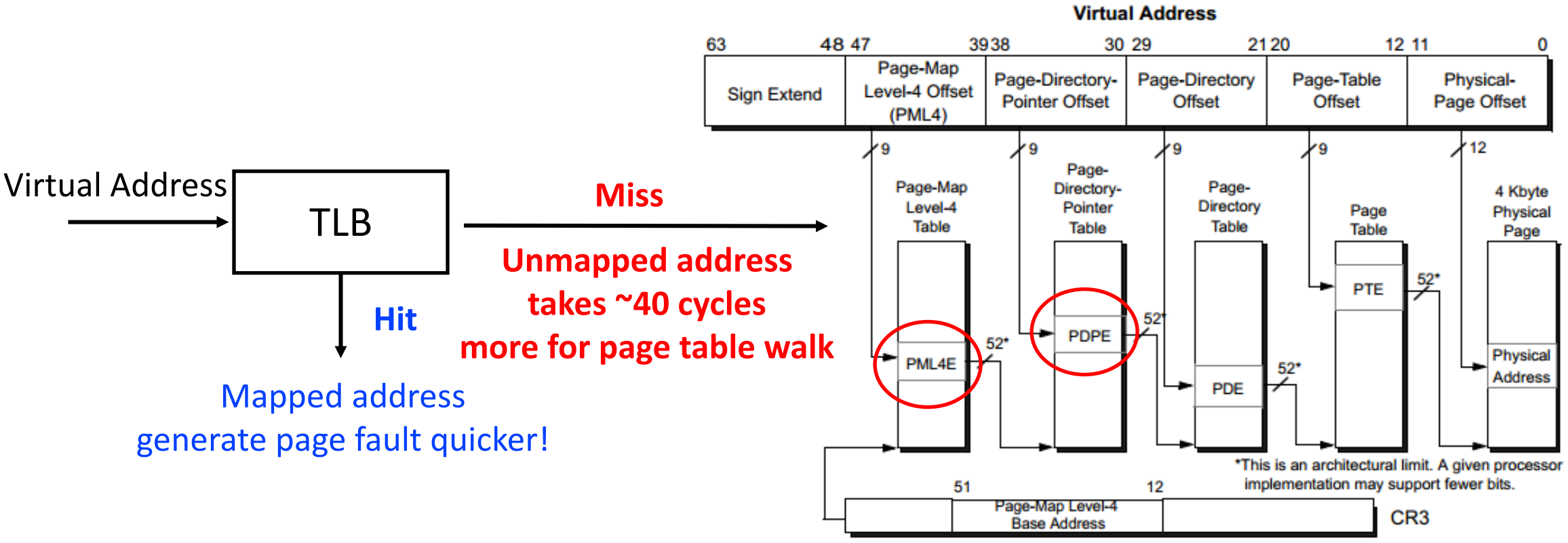
# TLB Timing Side Channel



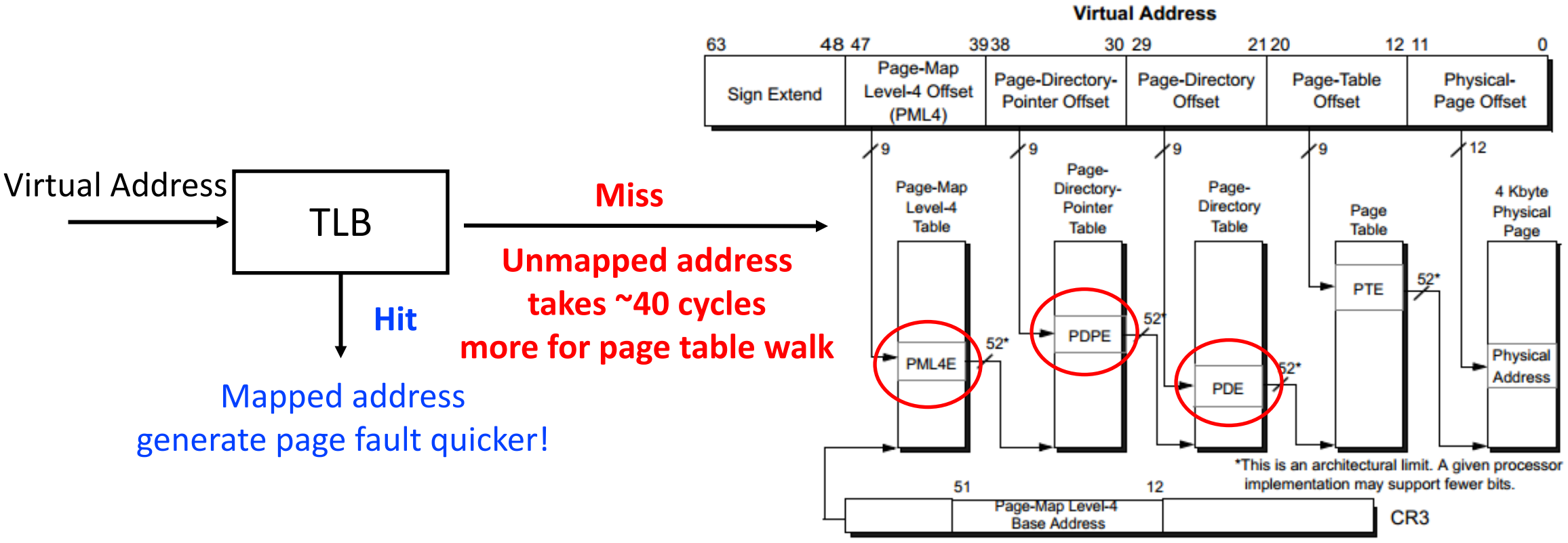
# TLB Timing Side Channel



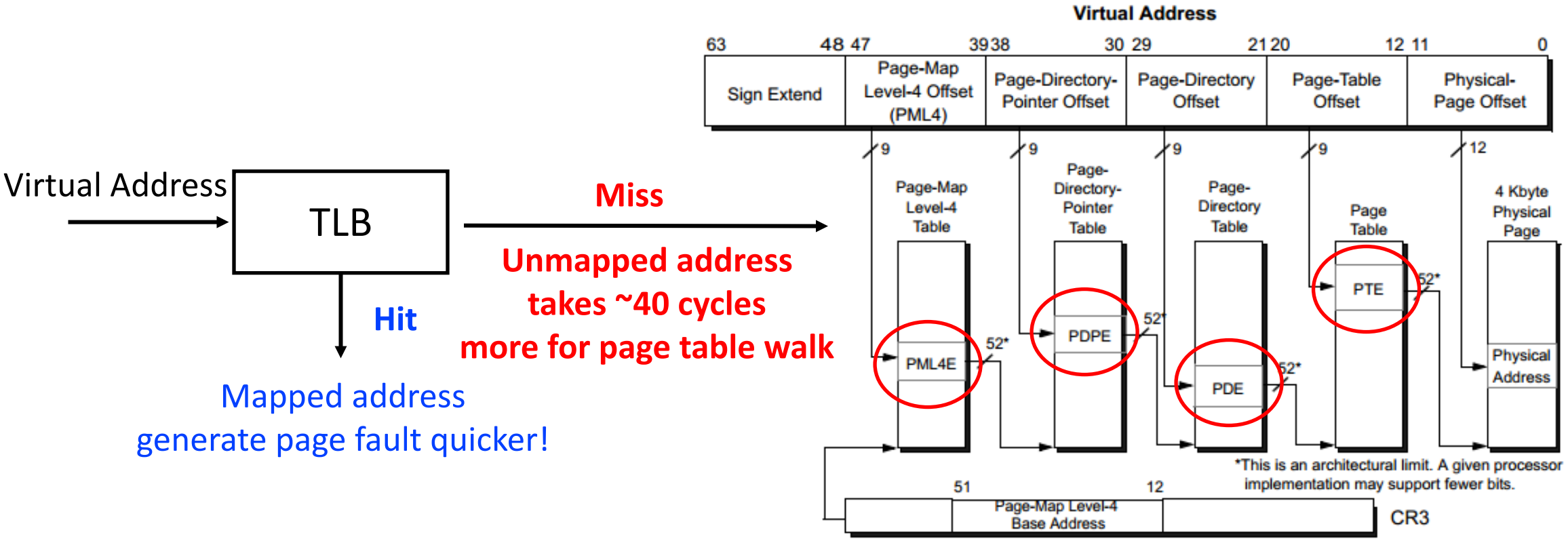
# TLB Timing Side Channel



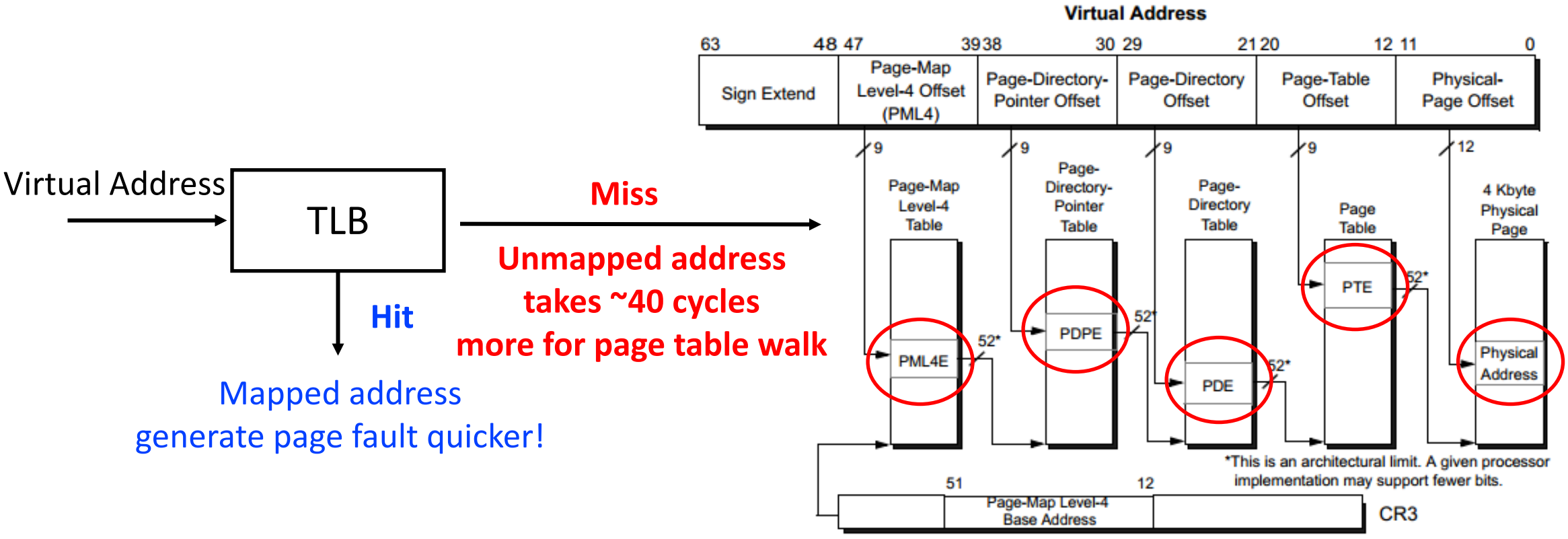
# TLB Timing Side Channel



# TLB Timing Side Channel

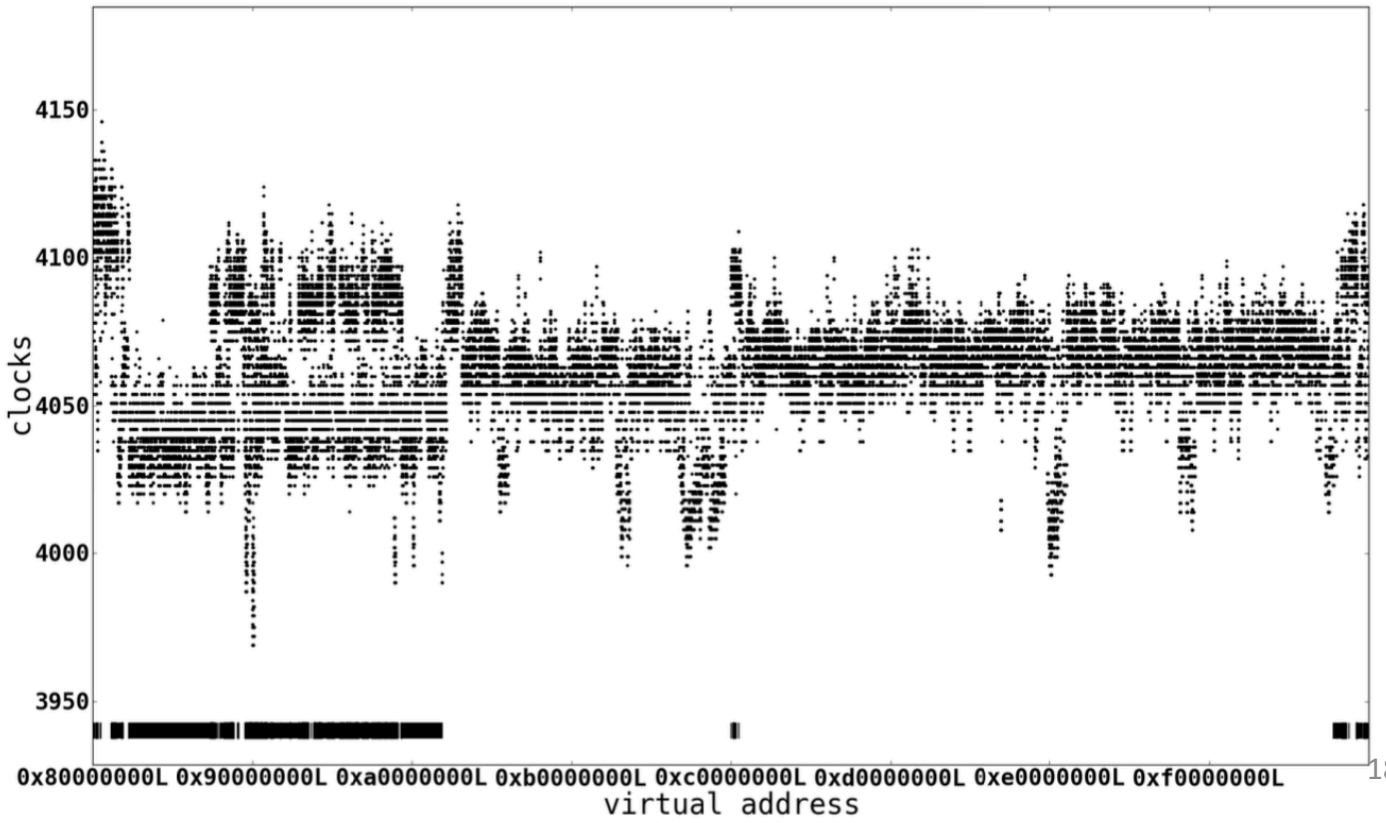


# TLB Timing Side Channel



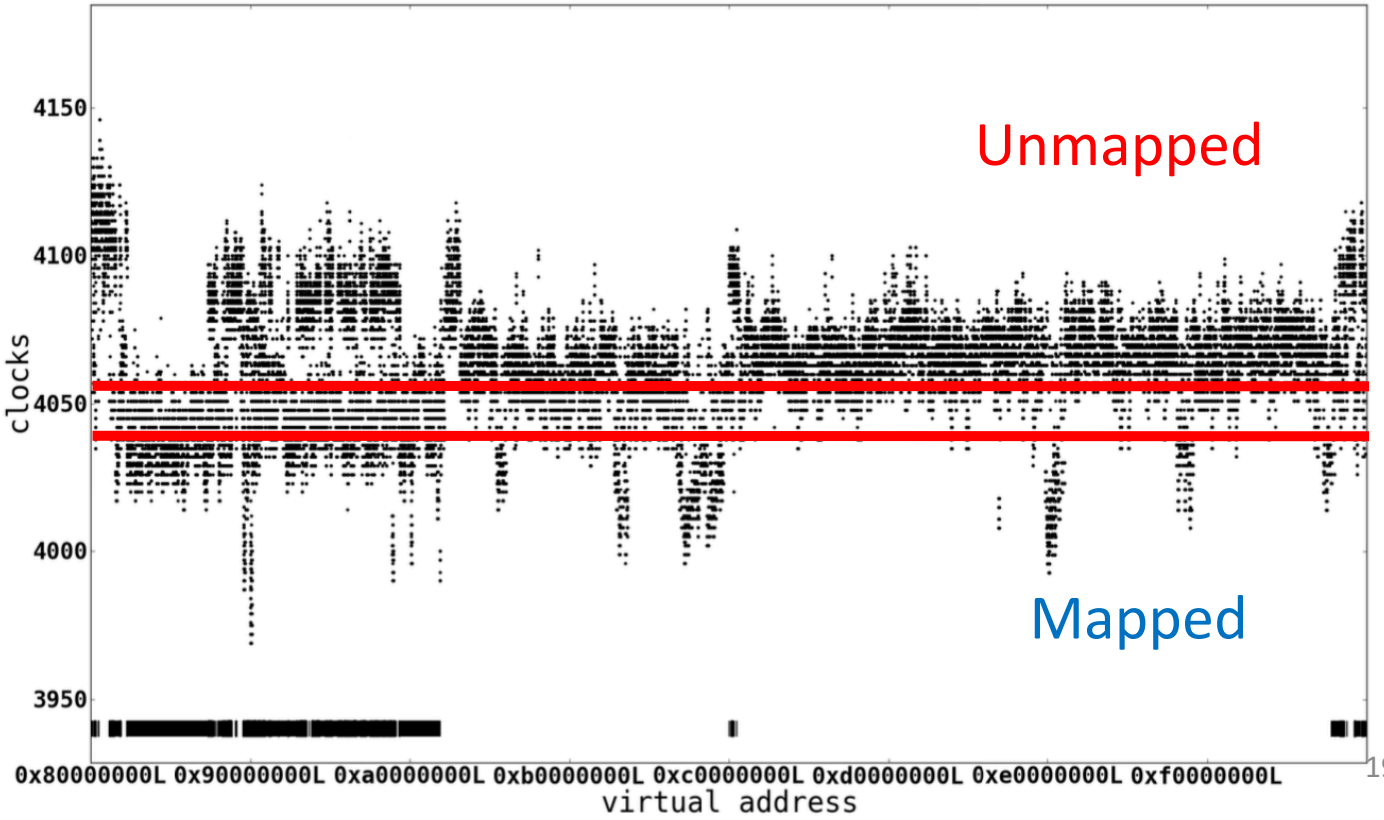
# TLB Timing Side Channel

- Result: Fault with TLB hit took less than 4050 cycles
  - While TLB miss took more than that…
- Limitation: Too noisy
  - Why????

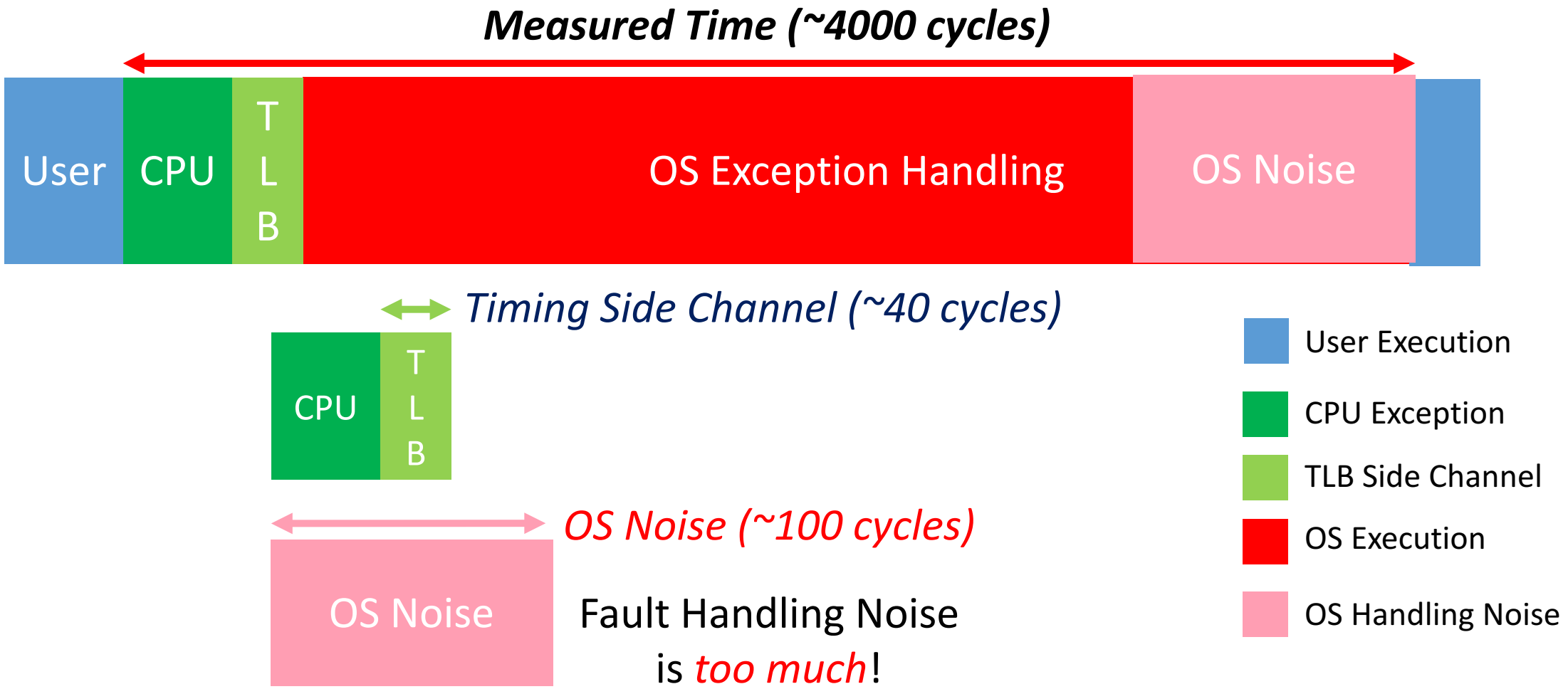


# TLB Timing Side Channel

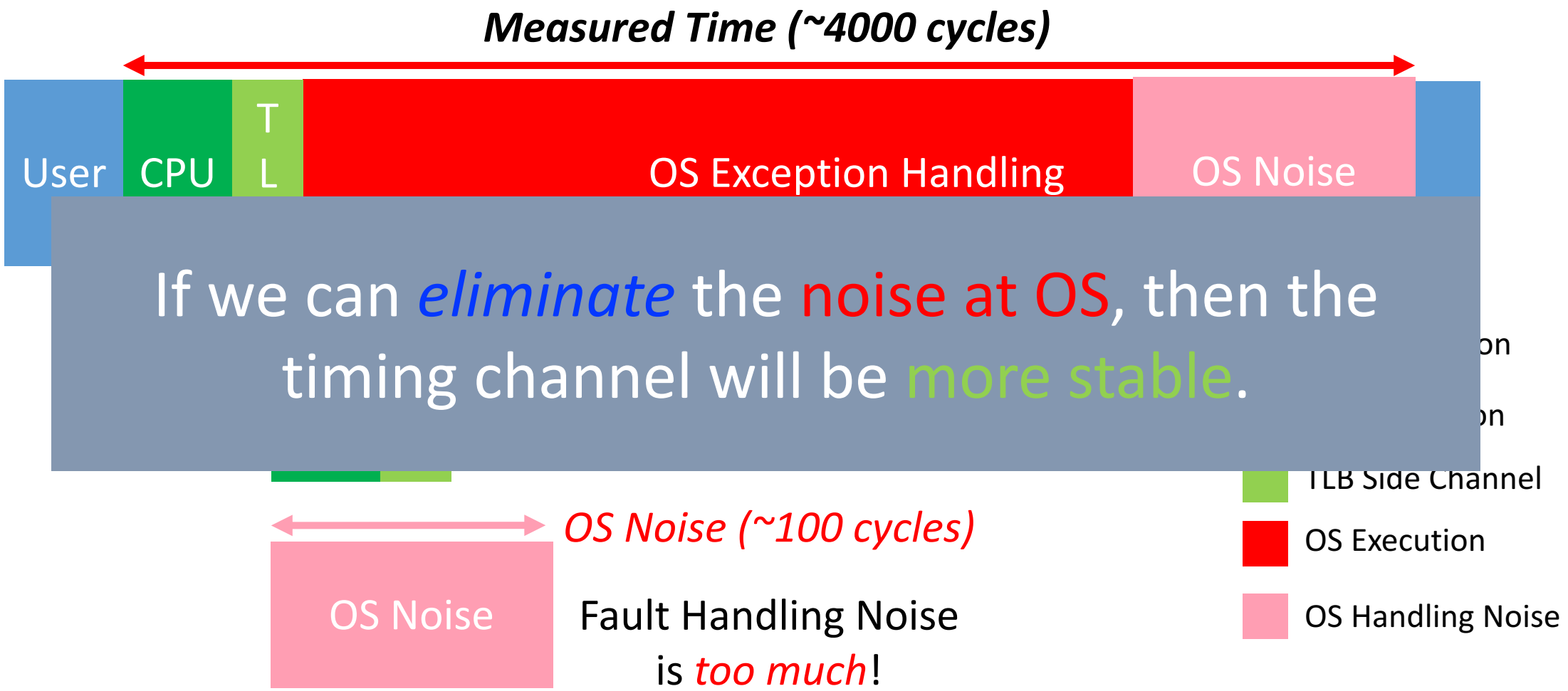
- Result: Fault with TLB hit took less than 4050 cycles
  - While TLB miss took more than that…
- Limitation: Too noisy
  - Why????



# TLB Timing Side Channel



# TLB Timing Side Channel



# A More Practical Side Channel Attack on KASLR

- The DrK Attack: We present a practical side channel attack on KASLR
  - De-randomizing Kernel ASLR (this is where DrK comes from)

- Exploit Intel TSX for eliminate the noise from OS
  - Distinguish mapped and unmapped pages
  - Distinguish executable and non-executable pages

# Transactional Synchronization Extension (Intel TSX)

- TSX: relaxed but faster way of handling synchronization

```
int status = 0;
if( (status = _xbegin()) == _XBEGIN_STARTED) {

    // atomic region
    try_atomic_operation();

    _xend();
    // atomic region end
}
else {

    // if failed,
    handle_abort();

}
```

# Transactional Synchronization Extension (Intel TSX)

- TSX: relaxed but faster way of handling synchronization

```
int status = 0;
if( (status = _xbegin()) == _XBEGIN_STARTED) {
    // atomic region
    try_atomic_operation();

    _xend();
    // atomic region end
}
else {

    // if failed,
    handle_abort();

}
```

**1. Do not block, do not use lock**

# Transactional Synchronization Extension (Intel TSX)

- TSX: relaxed but faster way of handling synchronization

```
int status = 0;
if( (status = _xbegin()) == _XBEGIN_STARTED) {
    // atomic region
    try_atomic_operation();

    _xend();
    // atomic region end
}
else {

    // if failed,
    handle_abort();

}
```

**1. Do not block, do not use lock**

**2. Try atomic operation (can fail)**

# Transactional Synchronization Extension (Intel TSX)

- TSX: relaxed but faster way of handling synchronization

```
int status = 0;
if( (status = _xbegin()) == _XBEGIN_STARTED) {
    // atomic region
    try_atomic_operation();

    _xend();
    // atomic region end
}
else {

    // if failed,
    handle_abort();

}
```

**1. Do not block, do not use lock**

**2. Try atomic operation (can fail)**

**3. If failed, handle failure with abort handler (retry, get back to traditional lock, etc.)**

# Transaction Aborts If Exist any of a Conflict

```
int status = 0;
if( (status = _xbegin()) == _XBEGIN_STARTED) {

    // atomic region
    try_atomic_operation();

    _xend();
    // atomic region end
}
else {

    // if failed,
    handle_abort();

}
```

Run If Transaction Aborts

- Condition of Conflict
  - Thread races
  - Cache eviction (L1 write/L3 read)
  - Interrupt
    - Context Switch (timer)
    - Syscalls
  - Exceptions
    - **Page Fault**
    - General Protection
    - Debugging
    - …

# Transaction Aborts If Exist any of a Conflict

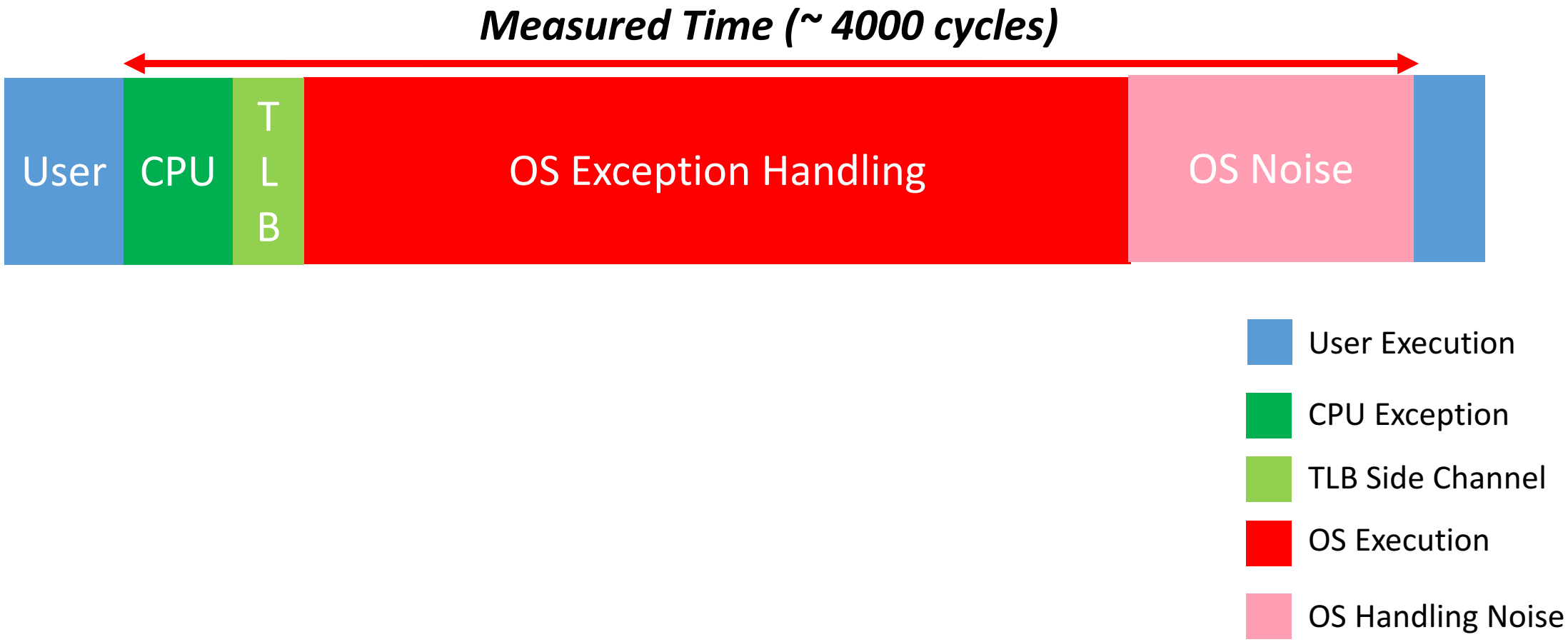
```
int status = 0;
if( (status = _xbegin()) == _XBEGIN_STARTED) {

    // atomic region
    try_atomic_operation();

    _xend();
    // atomic region end
}
else {

    // if failed,
    handle_abort();

}
```

Run If Transaction Aborts

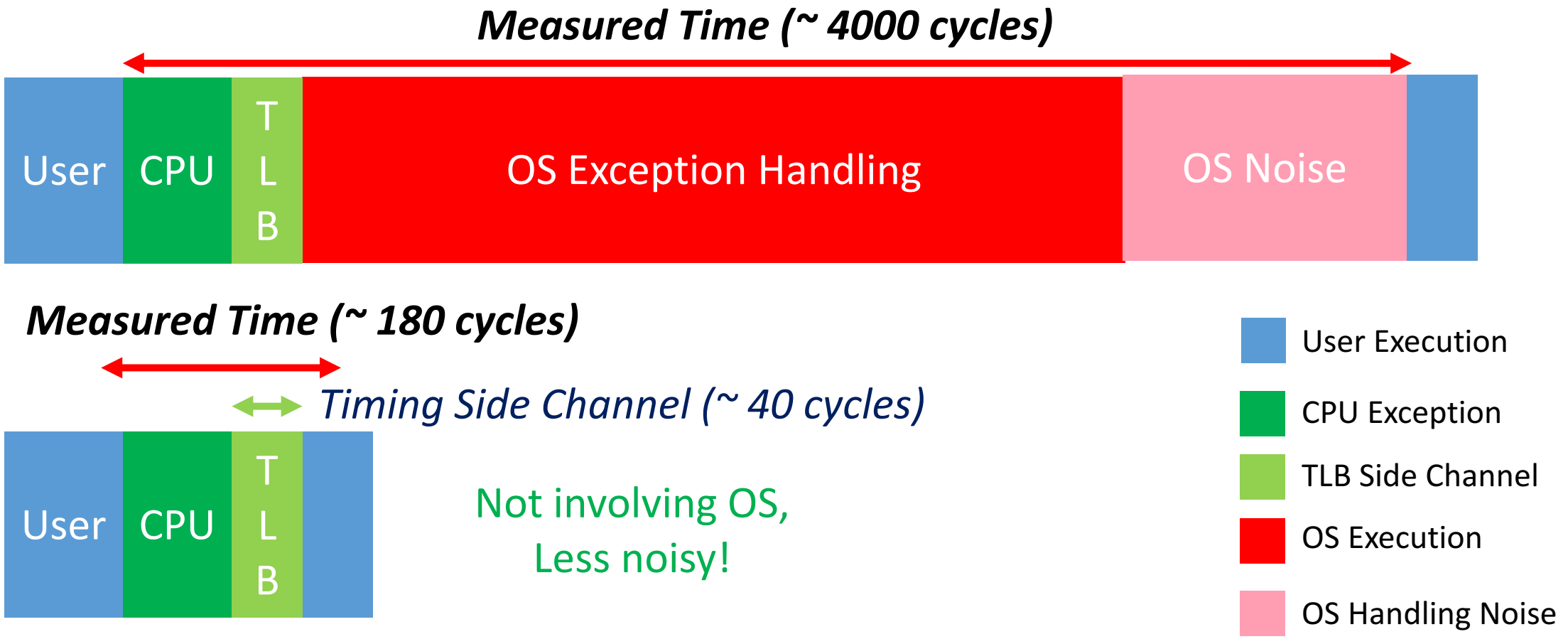
- Abort Handler of TSX
  - Suppress all sync. exceptions
    - E.g., page fault
  - **Do not notify OS**
    - Just jump into abort_handler()

No Exception delivery to the OS!
(returns quicker, so less noisy than OS exception handler)

# Reducing Noise with Intel TSX



# Reducing Noise with Intel TSX



# Exploiting TSX as an Exception Handler

- How to use TSX as an exception handler?

```
uint64_t time_begin, time_diff;
int status = 0;
int *p = (int*)0xffffffff80000000; // kernel addresss
time_begin = __rdtscp();
if((status = _xbegin()) == _XBEGIN_STARTED) {
    // TSX transaction
    *p; // read access
    // or,
    ((int(*)())p)(); // exec access
}
else {
    // abort handler
    time_diff = __rdtscp() - time_begin;
}
```

# Exploiting TSX as an Exception Handler

- How to use TSX as an exception handler?

```
uint64_t time_begin, time_diff;
int status = 0;
int *p = (int*)0xffffffff80000000; // kernel addresss
time_begin = __rdtscp();
if((status = _xbegin()) == _XBEGIN_STARTED) {
    // TSX transaction
    *p; // read access
    // or,
    ((int(*)())p)(); // exec access
}
else {
    // abort handler
    time_diff = __rdtscp() - time_begin;
}
```

**1. Timestamp at the beginning**

# Exploiting TSX as an Exception Handler

- How to use TSX as an exception handler?

```
uint64_t time_begin, time_diff;
int status = 0;
int *p = (int*)0xffffffff80000000; // kernel addresss
time_begin = __rdtscp();
if((status = _xbegin()) == _XBEGIN_STARTED) {
    // TSX transaction
    *p; // read access
    // or,
    ((int(*)())p)(); // exec access
}
else {
    // abort handler
    time_diff = __rdtscp() - time_begin;
}
```

**1. Timestamp at the beginning**

**2. Access kernel memory within the TSX region (always aborts)**

# Exploiting TSX as an Exception Handler

- How to use TSX as an exception handler?

```
uint64_t time_begin, time_diff;
int status = 0;
int *p = (int*)0xffffffff80000000; // kernel addresss
time_begin = __rdtscp();
if((status = _xbegin()) == _XBEGIN_STARTED) {
    // TSX transaction
    *p; // read access
    // or,
    ((int(*)())p)(); // exec access
}
else {
    // abort handler
    time_diff = __rdtscp() - time_begin;
}
```

**1. Timestamp at the beginning**

**2. Access kernel memory within the TSX region (always aborts)**

**3. Measure timing at abort handler**

# Exploiting TSX as an Exception Handler

- How to use TSX as an exception handler?

```
uint64_t time_begin, time_diff;
int status = 0;
int *p = (int*)0xffffffff80000000; // kernel addresss
time_begin = __rdtscp();
if((status = _xbegin()) == _XBEGIN_STARTED) {
    // TSX transaction
    *p; // read access
    // or,
    ((int(*)())p)(); // exec access
}
else {
    // abort handler
    time_diff = __rdtscp() - time_begin;
}
```

**1. Timestamp at the beginning**

**2. Access kernel memory within the TSX region (always aborts)**

Processor directly calls the handler
OS handling path is *not* involved

**3. Measure timing at abort handler**

# Measuring Timing Side Channel

- Mapped / Unmapped kernel addresses (across 4 CPUs)
  - Ran 1000 iterations for the probing, minimum clock on 10 runs

| Processor | Mapped Page | Unmapped Page |
|---|---|---|
| i7-6700K (4.0Ghz) | 209 | 240 (+31) |
| i5-6300HQ (2.3Ghz) | 164 | 188 (+24) |
| i7-5600U (2.6Ghz) | 149 | 173 (+24) |
| E3-1271v3 (3.6Ghz) | 177 | 195 (+18) |

  - Mapped page always faults faster

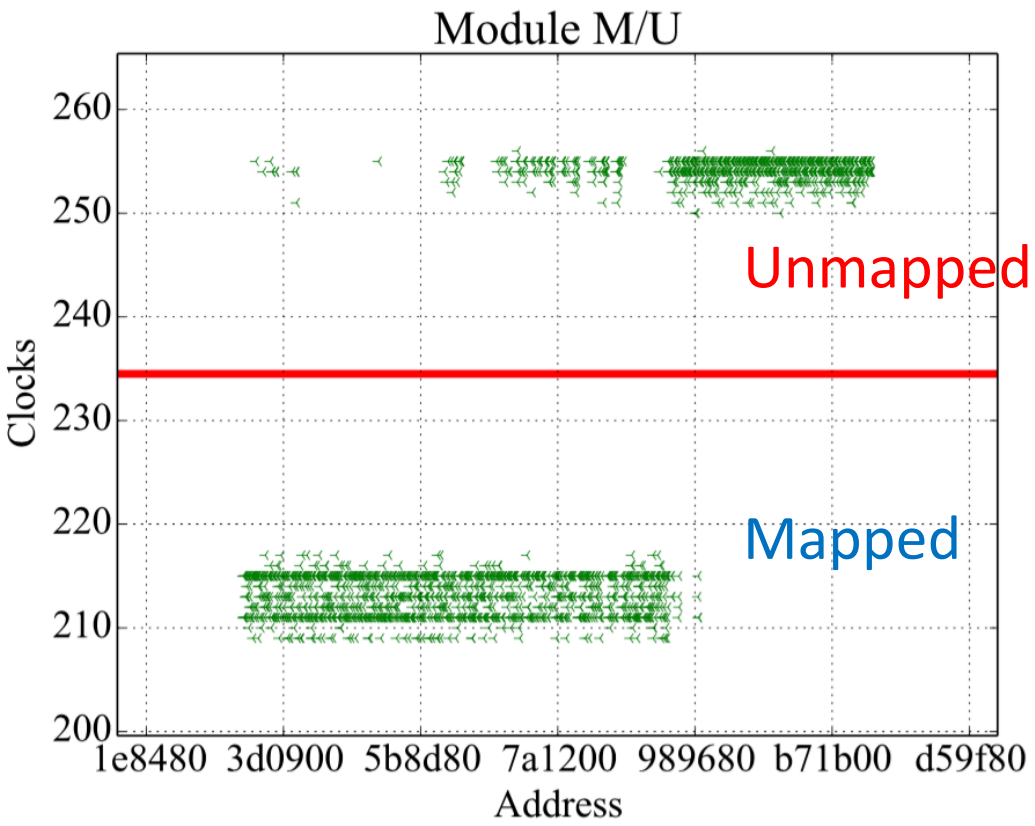
# Measuring Timing Side Channel

- Executable / Non-executable kernel addresses
  - Ran 1000 iterations for the probing, minimum clock on 10 runs

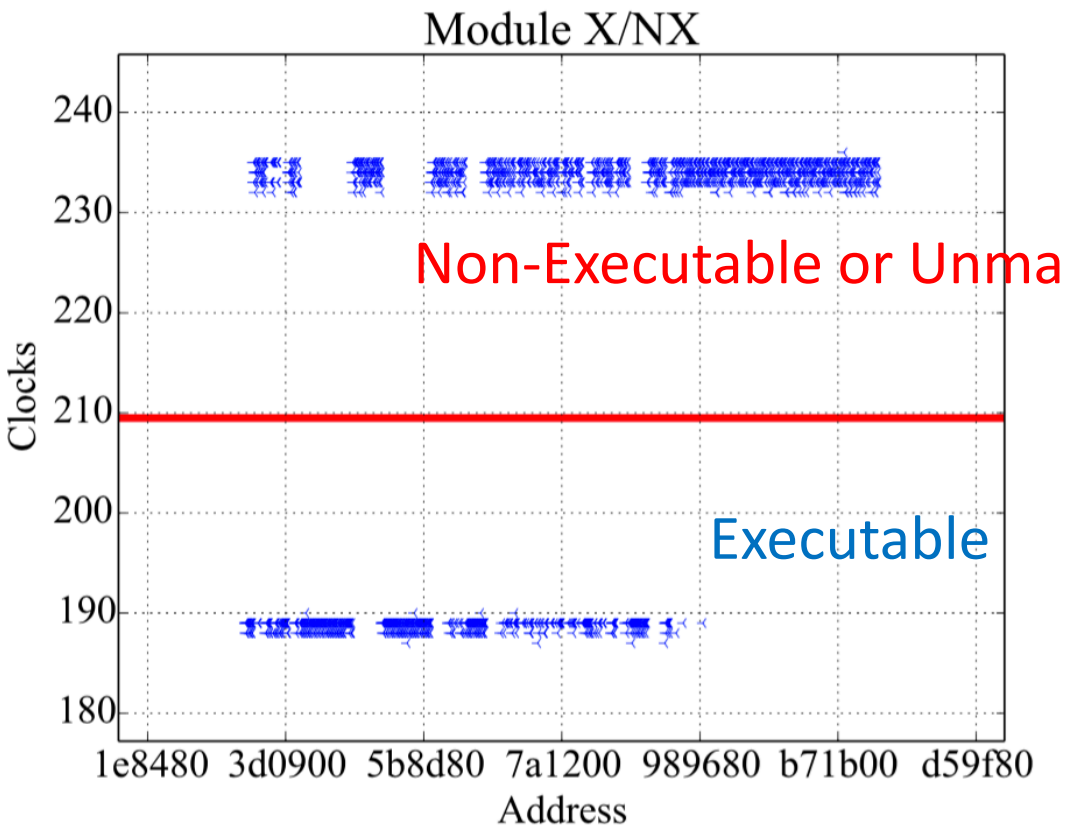
| Processor | Executable Page | Non-exec Page |
|---|---|---|
| i7-6700K (4.0Ghz) | 181 | 226 (+45) |
| i5-6300HQ (2.3Ghz) | 142 | 178 (+36) |
| i7-5600U (2.6Ghz) | 134 | 164 (+30) |
| E3-1271v3 (3.6Ghz) | 159 | 189 (+30) |

  - Executable page always faults faster

# Clear Timing Channel




(a) Mapped vs. Unmapped




(b) Executable vs. Non-executable

Clear separation between different mapping status!

# Attack on Various OSes

- Attack Targets
  - DrK is hardware side-channel attack
    - The mechanism is independent to OS
  - We target popular OSes: Linux, Windows, and macOS

- Attack Types
  - Type 1: Revealing mapping status of each page (X / NX / U)
  - Type 2: Finer-grained module detection

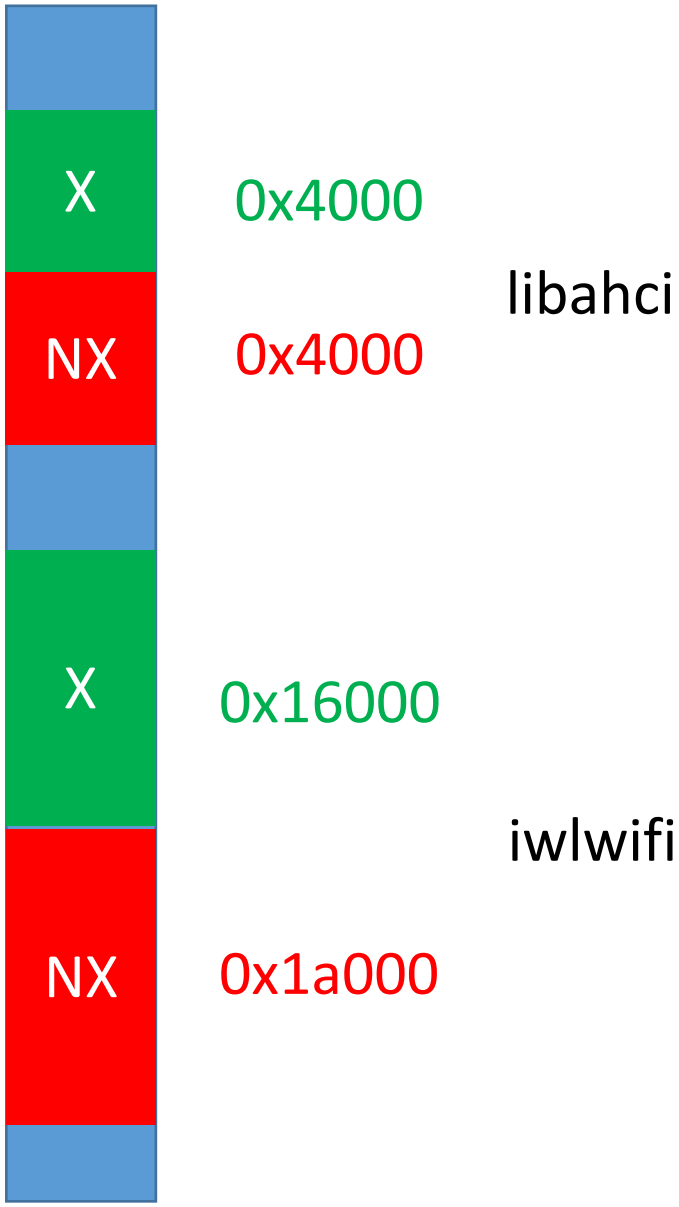
# Attack on Various OSes

- Type 1: Revealing mapping status of each page
  - Try to reveal the mapping status per each page in the area
    - X (executable) / NX (Non-executable) / U (unmapped)

```
0xffffffffc0278000-0xffffffffc027d000 U
0xffffffffc027d000-0xffffffffc0281000 X
0xffffffffc0281000-0xffffffffc0285000 NX
0xffffffffc0285000-0xffffffffc0289000 U
0xffffffffc0289000-0xffffffffc028b000 X
0xffffffffc028b000-0xffffffffc028e000 NX
0xffffffffc028e000-0xffffffffc0293000 U
0xffffffffc0293000-0xffffffffc02b7000 X
0xffffffffc02b7000-0xffffffffc02e9000 NX
0xffffffffc02e9000-0xffffffffc02ea000 U
0xffffffffc02ea000-0xffffffffc02f0000 X
```

# Attack on Various OSes

- Type 2: Finer-grained module detection
  - Section-size Signature
    - Modules are allocated in fixed size of X/NX sections if the attacker knows the binary file
  - Example
    - If the size of executable map is 0x4000, and the size of non-executable section is 0x4000, then it is libahci!



```
[blue9057@sgx-Inspiron-7559 (master) ~/drk/linux$]
```

# Result Summary

- Linux: 100% of accuracy around 0.1 second
- Windows: 100% for M/U in 5 sec, 99.28% for X/NX for 45 sec
- OS X: 100% for detecting ASLR slide, in 31ms
- Linux on Amazon EC2: 100% of accuracy in 3 seconds

# Timing Side Channel (M/U)

- For Mapped / Unmapped addresses
  - Measured performance counters (on 1,000,000 probing)

| Perf. Counter | Mapped Page | Unmapped Page | Description |
|---|---|---|---|
| dTLB-loads | 3,021,847 | 3,020,243 | |
| dTLB-load-misses | 84 | 2,000,086 | TLB-miss on U |
| Observed Timing | 209 (fast) | 240 (slow) | |

  - dTLB hit on mapped pages, but not for unmapped pages.
    - Timing channel is generated by dTLB hit/miss

# Timing Side Channel (M/U)

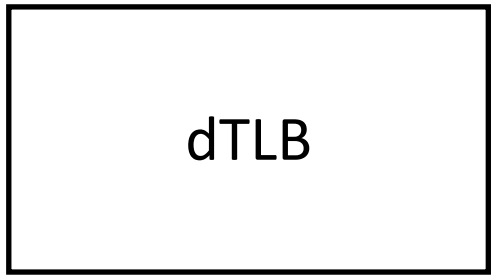
- For Mapped / Unmapped addresses
  - Measured performance counters (on 1,000,000 probing)

| Perf. Counter | Mapped Page | Unmapped Page | Description |
|---|---|---|---|
| dTLB-loads | 3,021,847 | 3,020,243 | |
| dTLB-load-misses | 84 | 2,000,086 | TLB-miss on U |
| Observed Timing | 209 (fast) | 240 (slow) | |

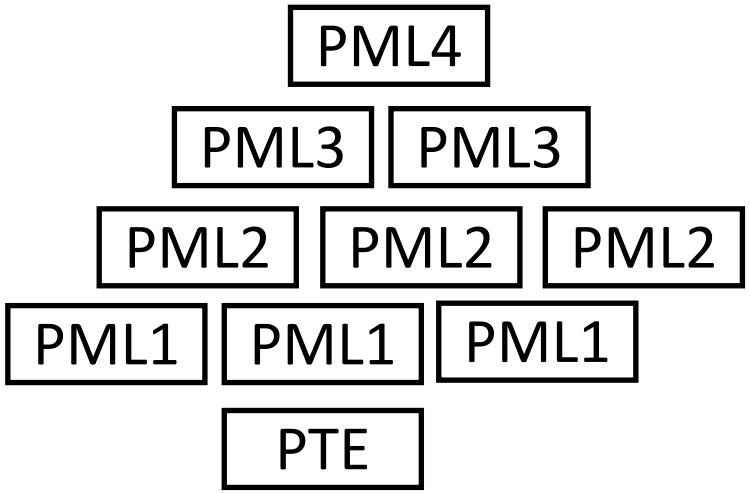
  - dTLB hit on mapped pages, but not for unmapped pages.
    - Timing channel is generated by dTLB hit/miss

# Path for an Unmapped Page

Probing an unmapped page took 240 cycles

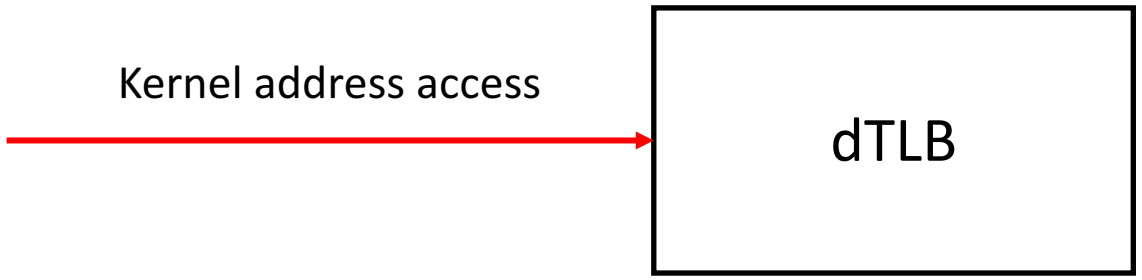


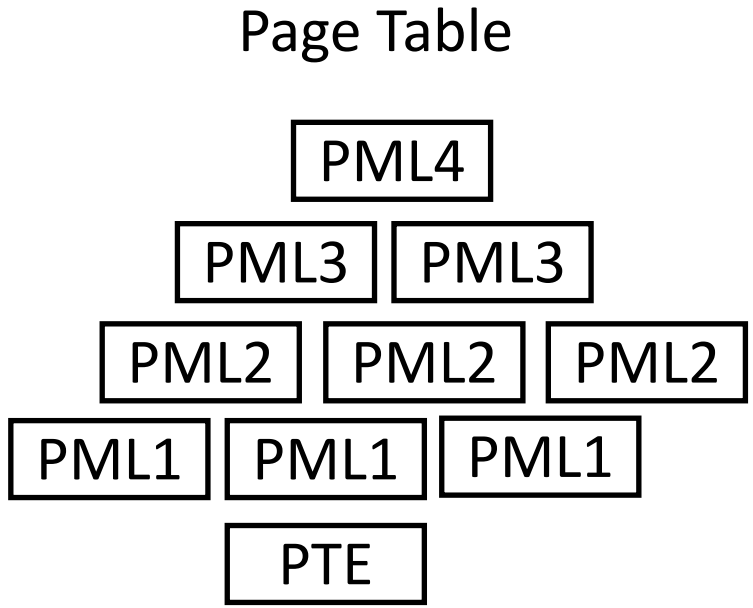

Page Table



# Path for an Unmapped Page

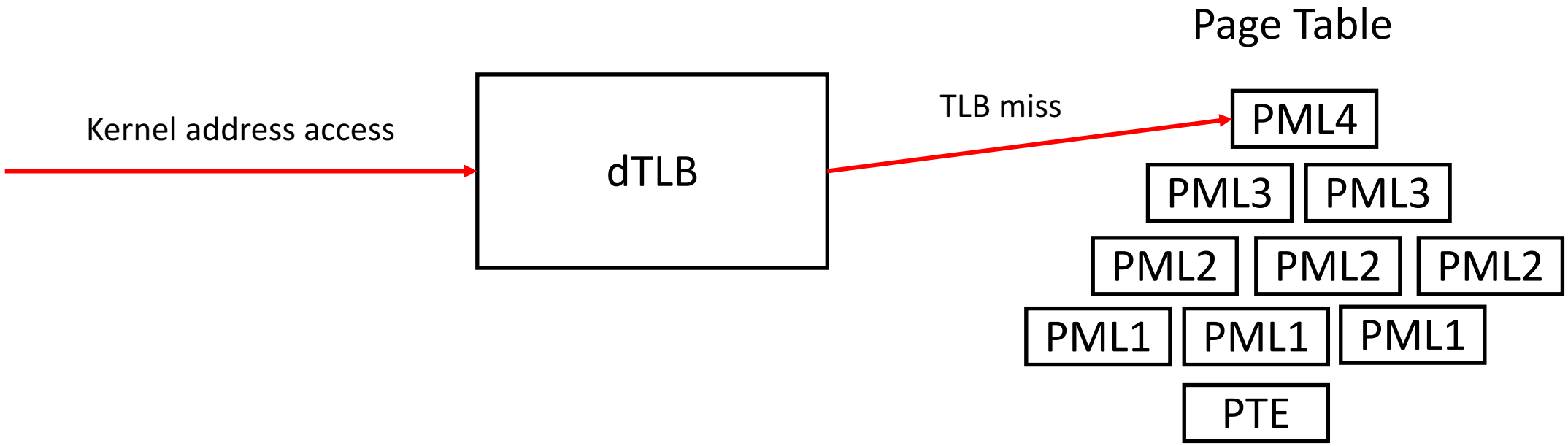
Probing an unmapped page took 240 cycles



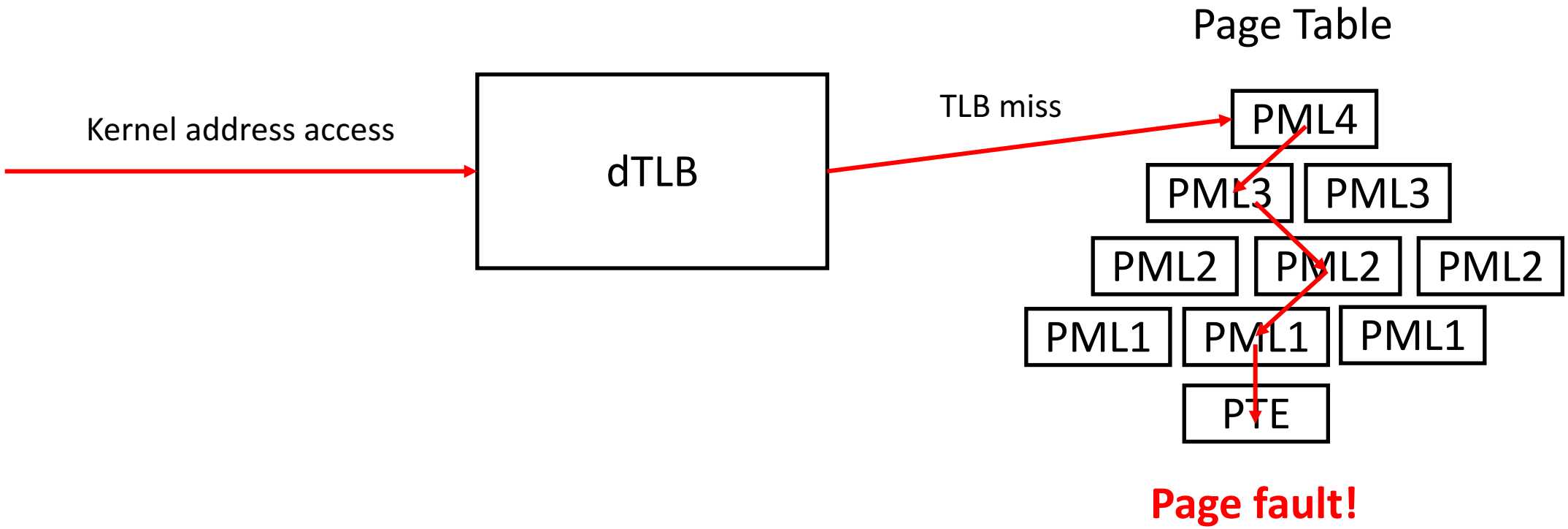
# Path for an Unmapped Page

Probing an unmapped page took 240 cycles



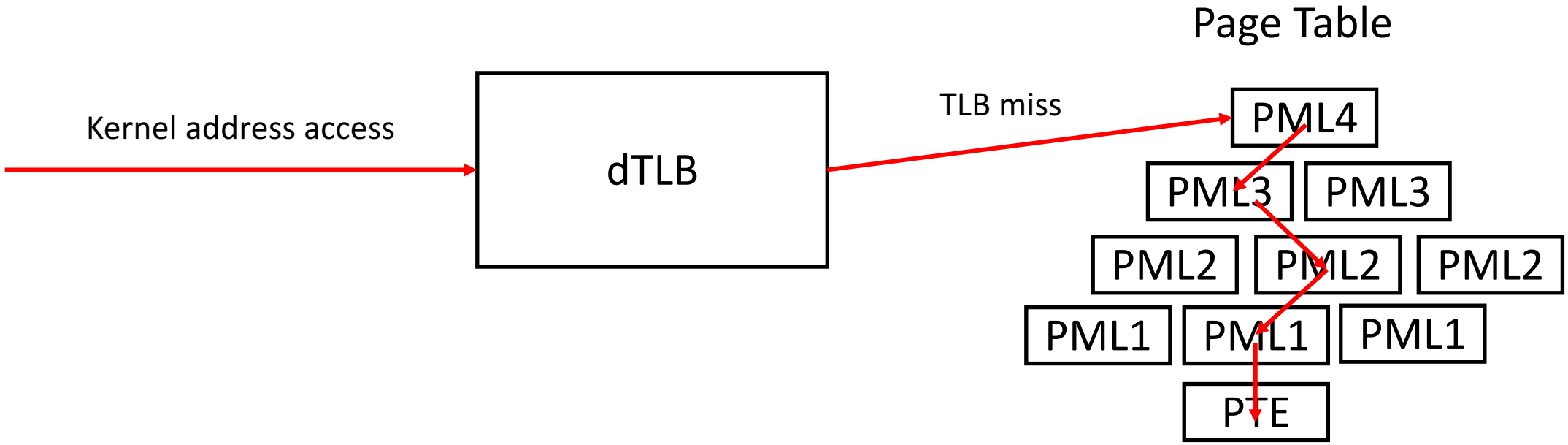
# Path for an Unmapped Page

Probing an unmapped page took 240 cycles



# Path for an Unmapped Page

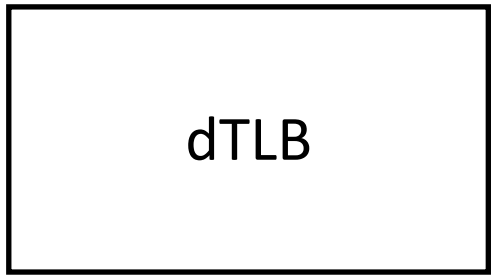
Probing an unmapped page took 240 cycles

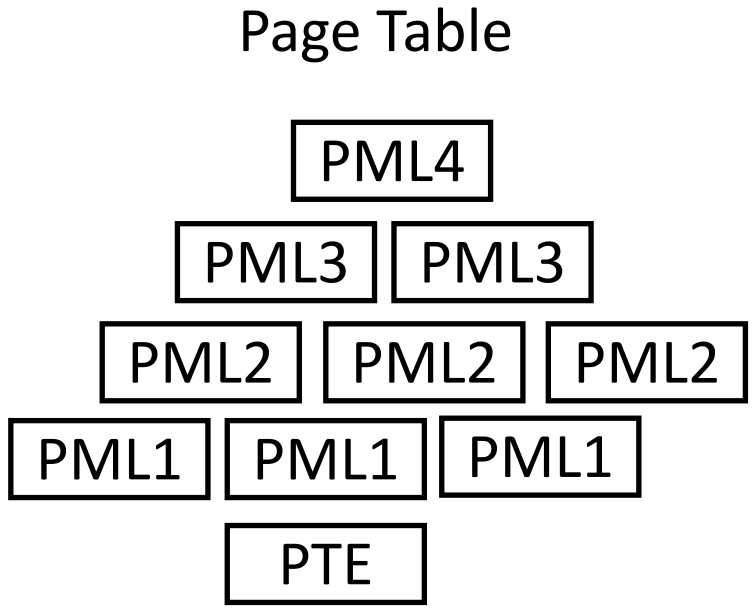



Always do page table walk (slow)

# Path for a mapped Page

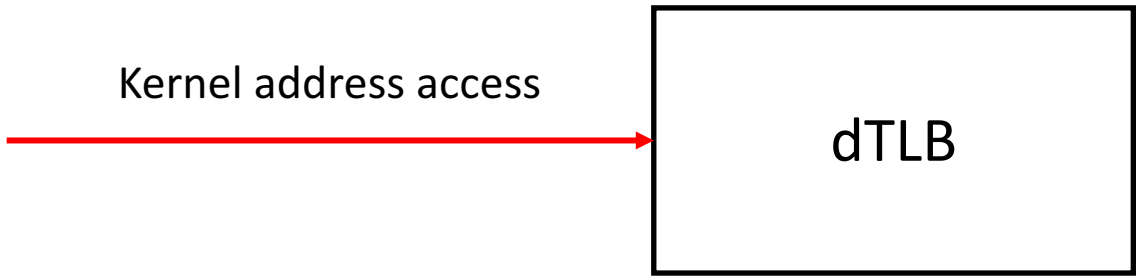
On the first access, 240 cycles

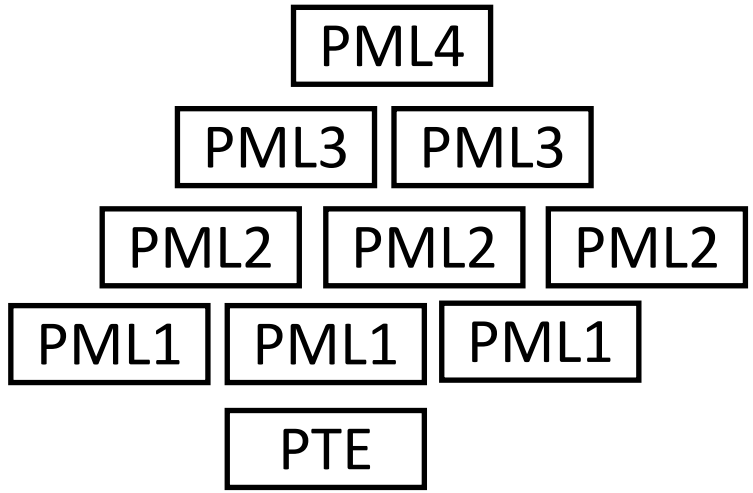


# Path for a mapped Page

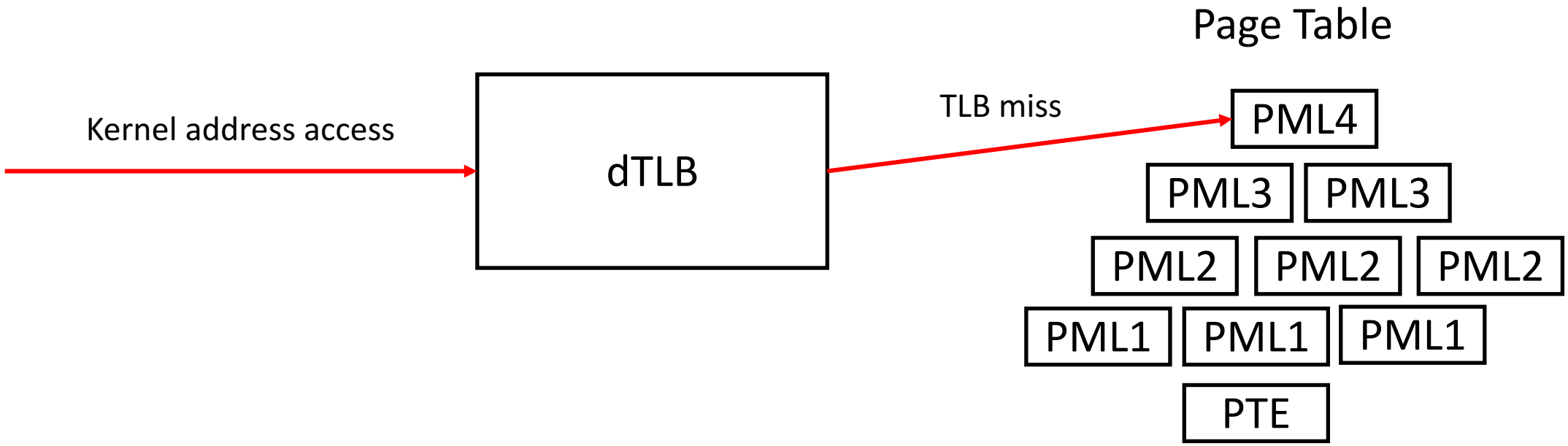
On the first access, 240 cycles



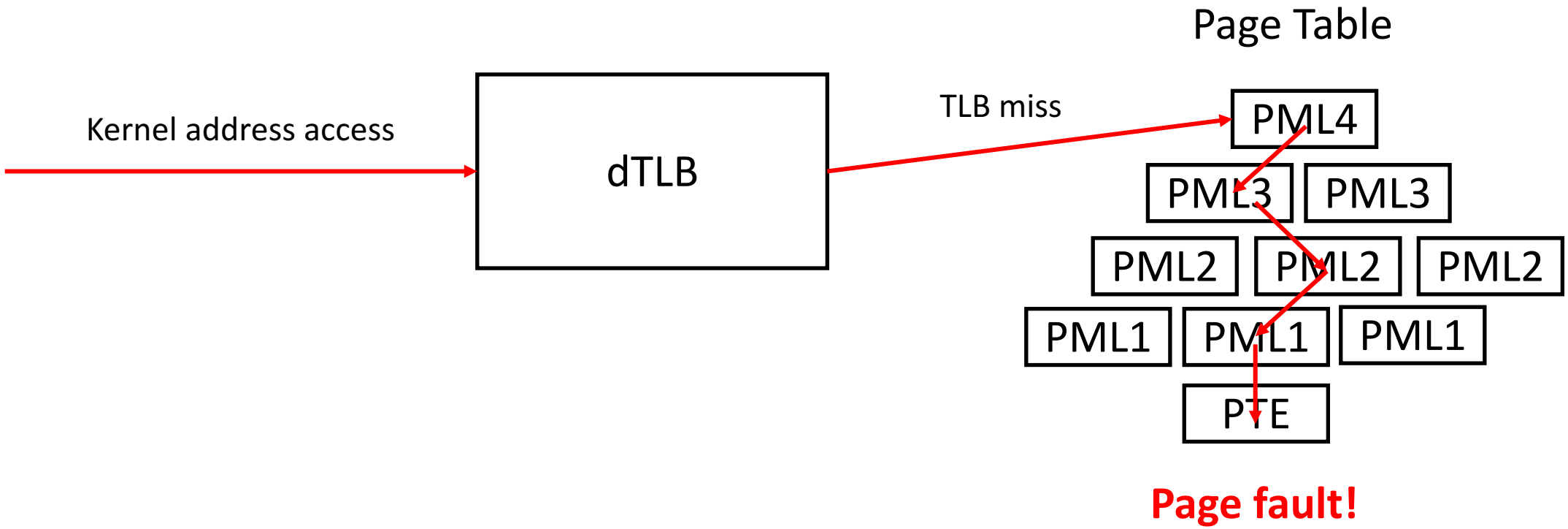
# Path for a mapped Page

On the first access, 240 cycles



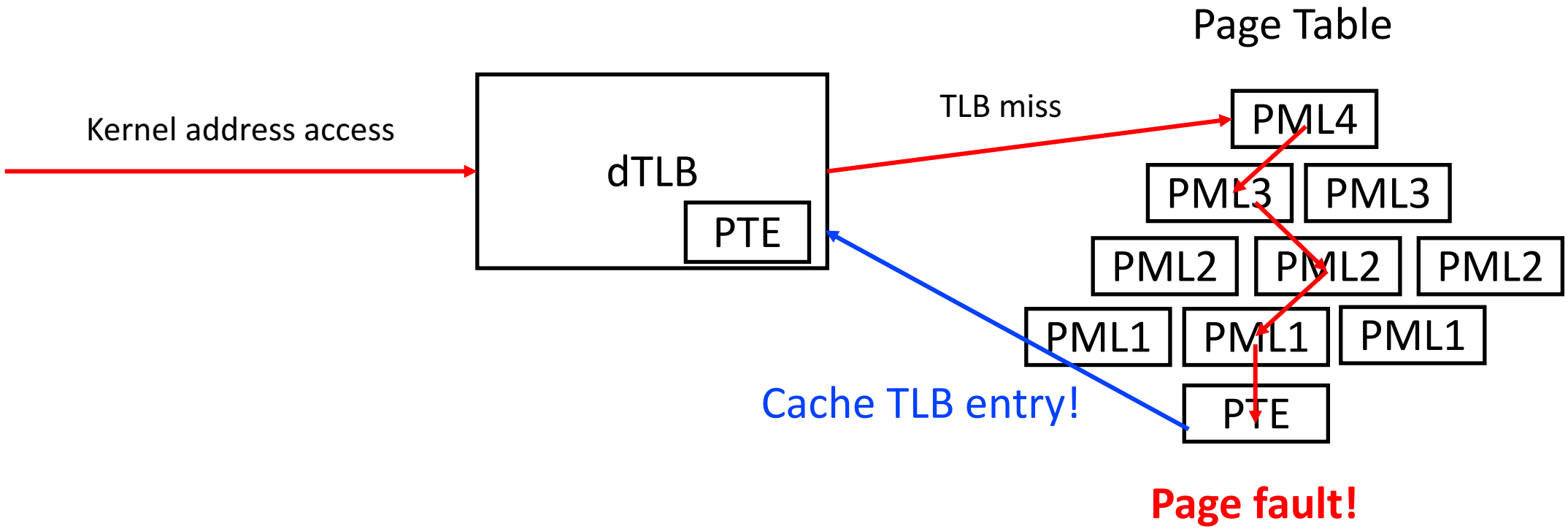
# Path for a mapped Page

On the first access, 240 cycles



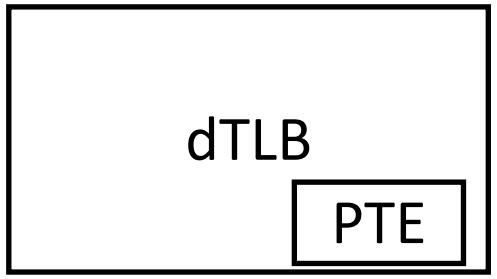
# Path for a mapped Page

On the first access, 240 cycles

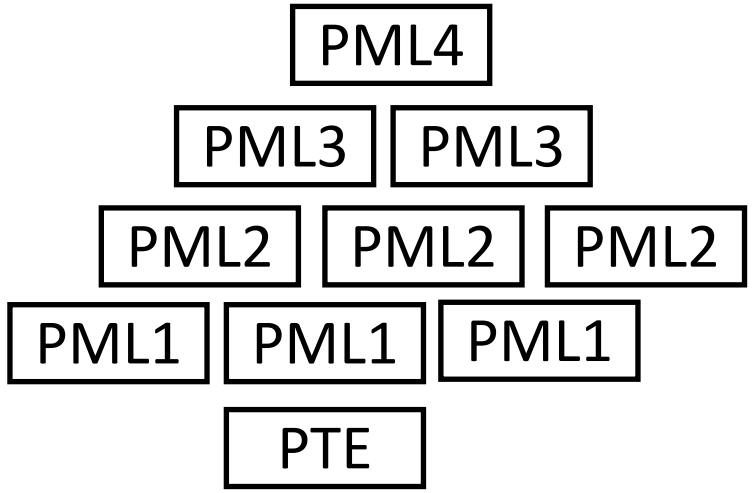


# Path for a mapped Page

On the second access, 209 cycles

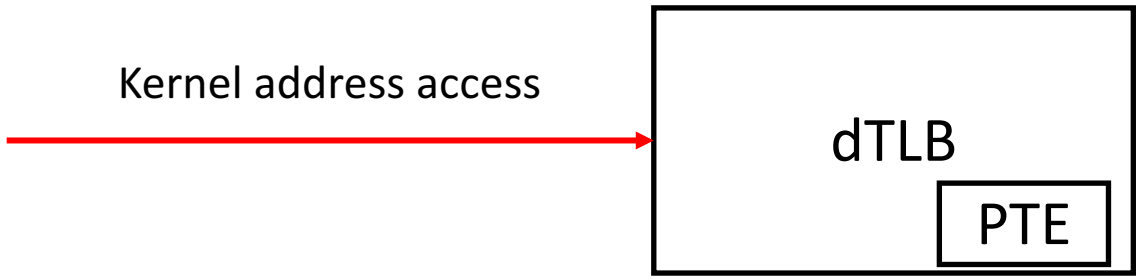



Page Table

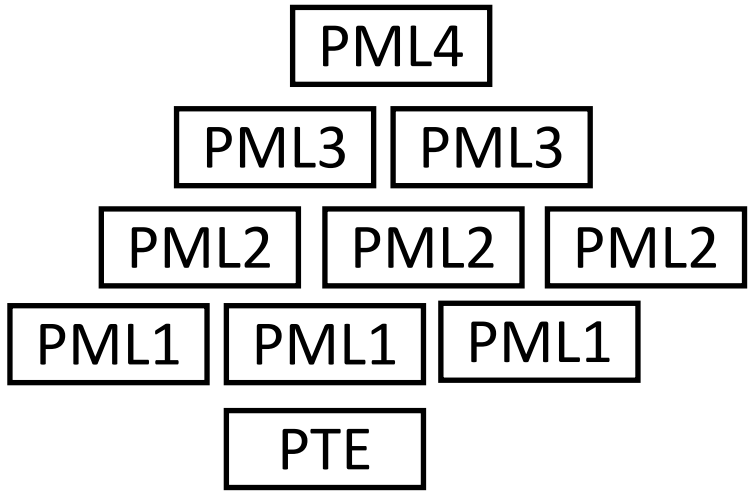


# Path for a mapped Page

On the second access, 209 cycles

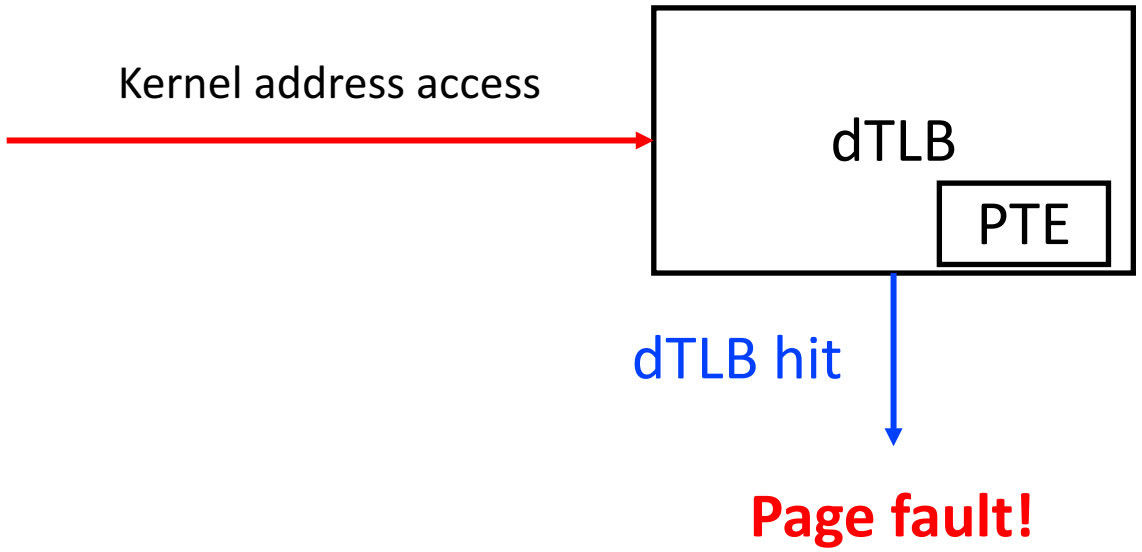


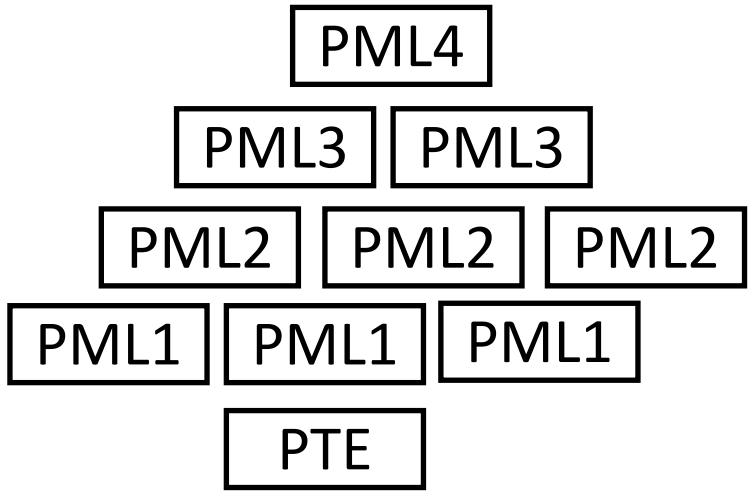

Page Table



# Path for a mapped Page

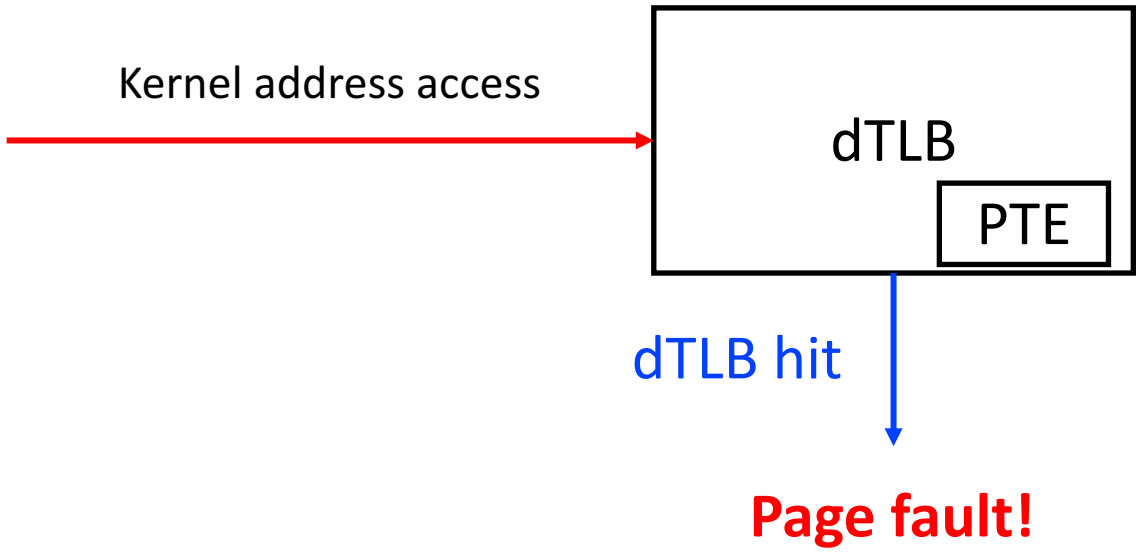
On the second access, 209 cycles

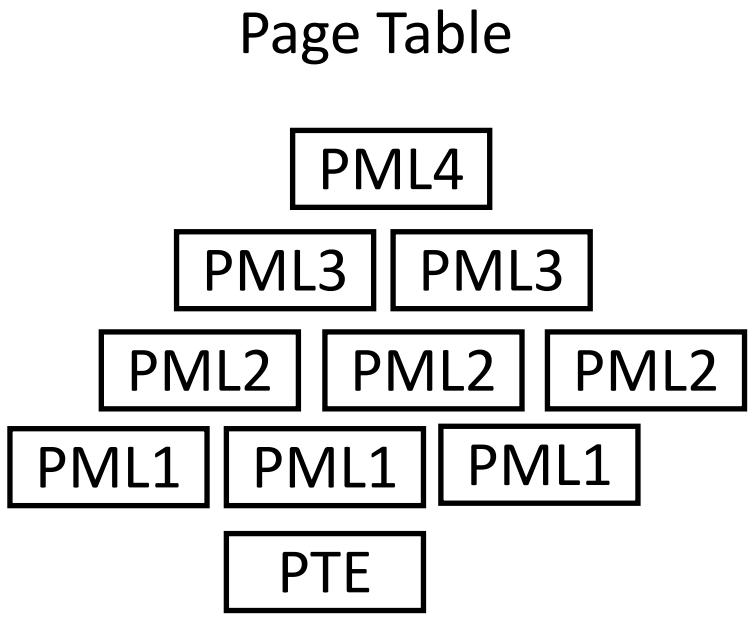


# Path for a mapped Page

On the second access, 209 cycles




No page table walk on the second access (fast)

# Timing Side Channel (X/NX)

- For Executable / Non-executable addresses
  - Measured performance counters (on 1,000,000 probing)

| Perf. Counter | Exec Page | Non-exec Page | Unmapped Page |
|---|---|---|---|
| iTLB-loads (hit) | 590 | 1,000,247 | 272 |
| iTLB-load-misses | 31 | 12 | 1,000,175 |
| Observed Timing | 181 (fast) | 226 (slow) | 226 (slow) |

  - Point #1: iTLB hit on Non-exec, but it is slow (226) why?
  - iTLB is not the origin of the side channel

# Timing Side Channel (X/NX)

- For Executable / Non-executable addresses
  - Measured performance counters (on 1,000,000 probing)

| Perf. Counter | Exec Page | Non-exec Page | Unmapped Page |
|---|---|---|---|
| iTLB-loads (hit) | 590 | 1,000,247 | 272 |
| iTLB-load-misses | 31 | 12 | 1,000,175 |
| Observed Timing | 181 (fast) | 226 (slow) | 226 (slow) |

  - Point #1: iTLB hit on Non-exec, but it is slow (226) why?
  - iTLB is not the origin of the side channel

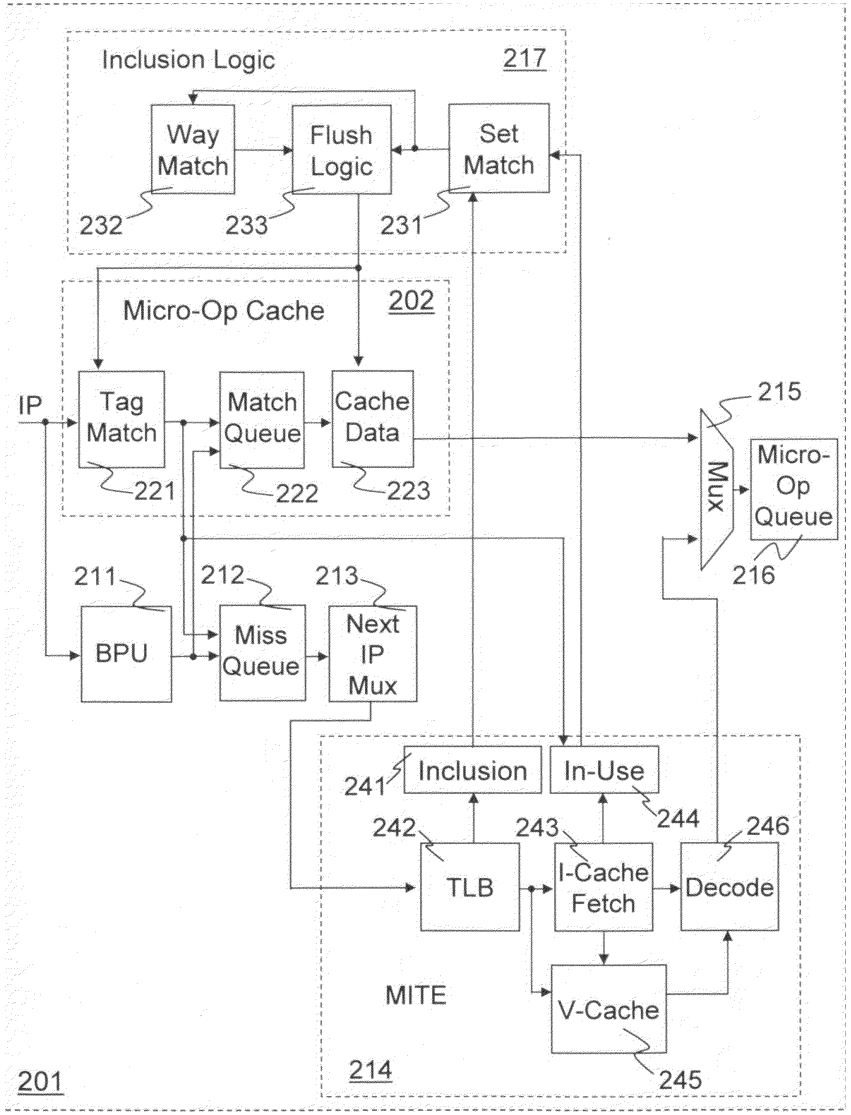
# Timing Side Channel (X/NX)

- For Executable / Non-executable addresses
  - Measured performance counters (on 1,000,000 probing)

| Perf. Counter | Exec Page | Non-exec Page | Unmapped Page |
|---|---|---|---|
| iTLB-loads (hit) | 590 | 1,000,247 | 272 |
| iTLB-load-misses | 31 | 12 | 1,000,175 |
| Observed Timing | 181 (fast) | 226 (slow) | 226 (slow) |

  - Point #2: iTLB does not even hit on Exec page, while NX page hits iTLB
  - iTLB did not involve in the fast path
    - Is there any cache that does not require address translation?

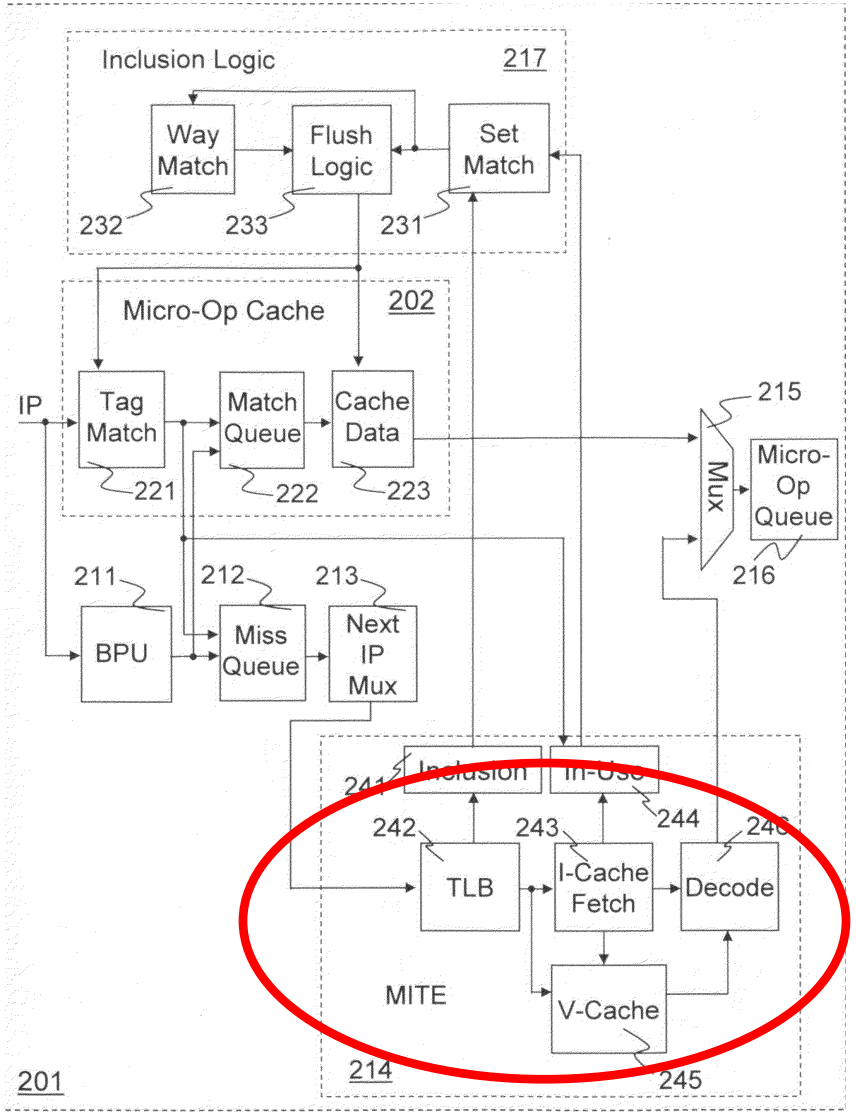
# Intel Cache Architecture




From the patent **US 20100138608 A1**, registered by Intel Corporation

# Intel Cache Architecture

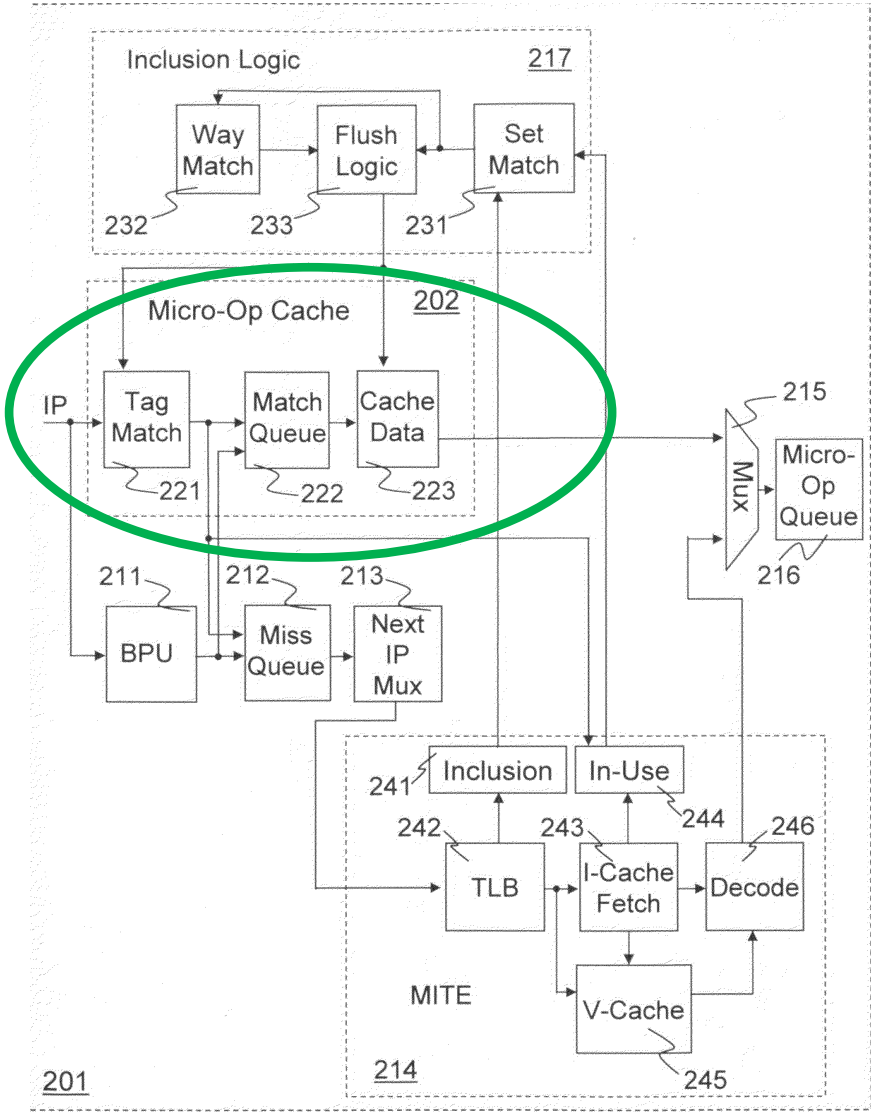
- L1 instruction cache
  - Virtually-indexed, Physically-tagged cache (requires TLB access)
  - Caches actual x86/x64 opcode




From the patent **US 20100138608 A1**, registered by Intel Corporation

# Intel Cache Architecture

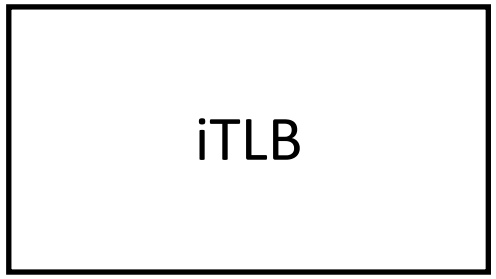
- Decoded i-cache
  - An instruction will be decoded as micro-ops (RISC-like instruction)
  - Decoded i-cache stores micro-ops
  - Virtually-indexed, Virtually-tagged cache (no TLB access)

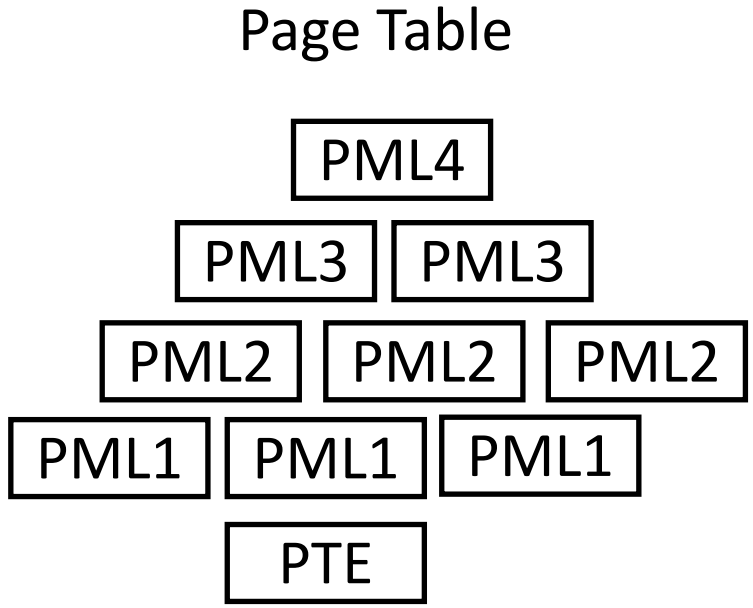



From the patent **US 20100138608 A1**, registered by Intel Corporation

# Path for an Unmapped Page

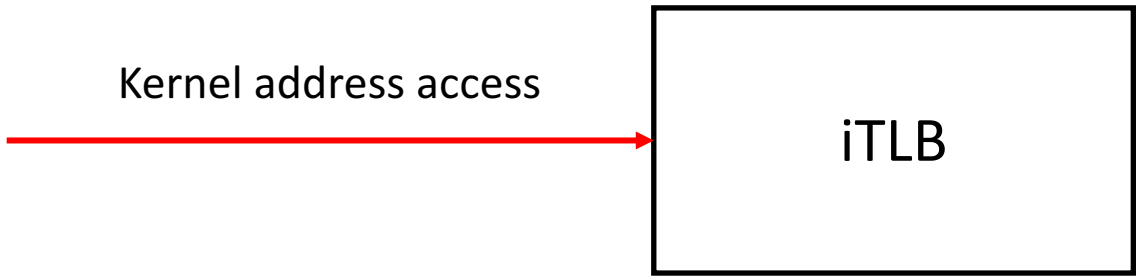
On the second access, 226 cycles



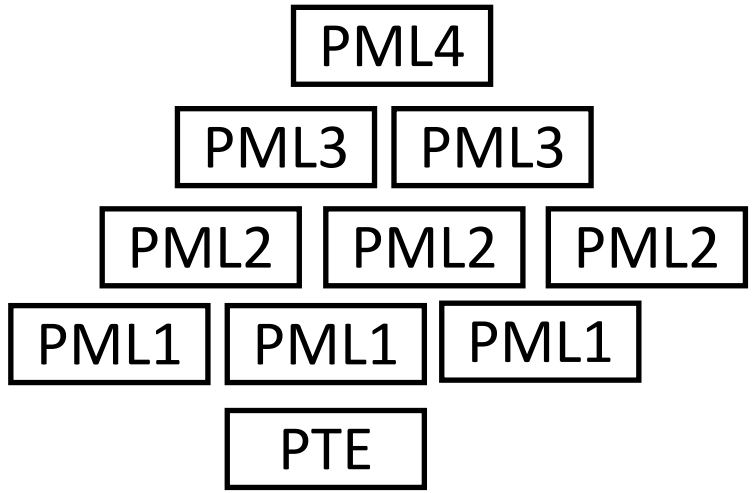



# Path for an Unmapped Page

On the second access, 226 cycles

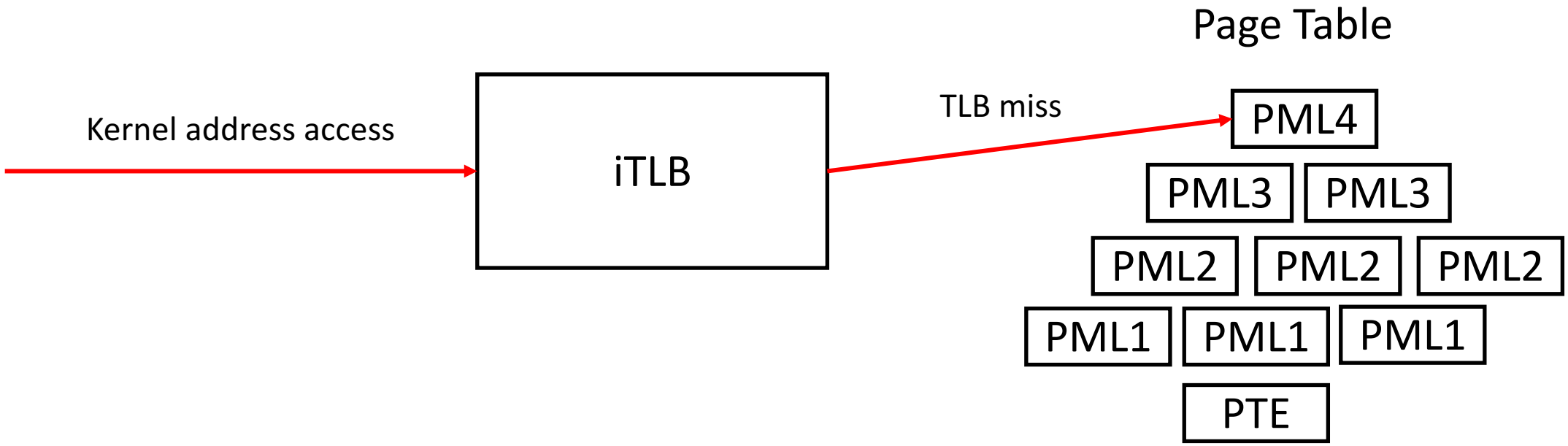



Page Table



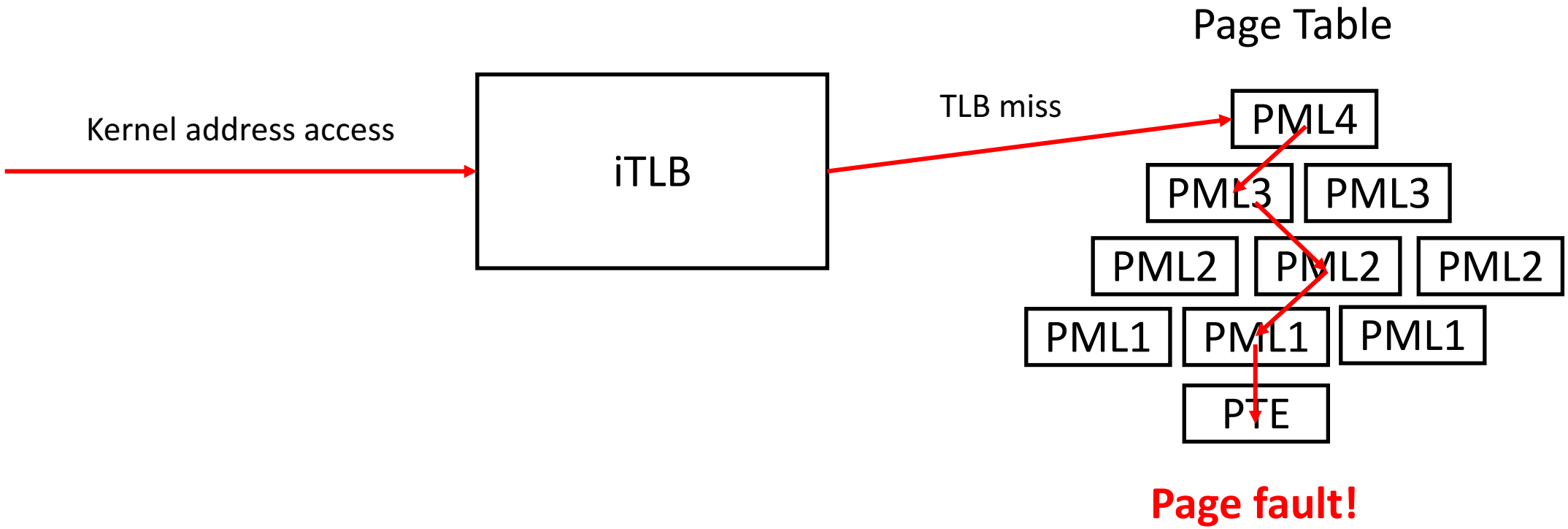
# Path for an Unmapped Page

On the second access, 226 cycles



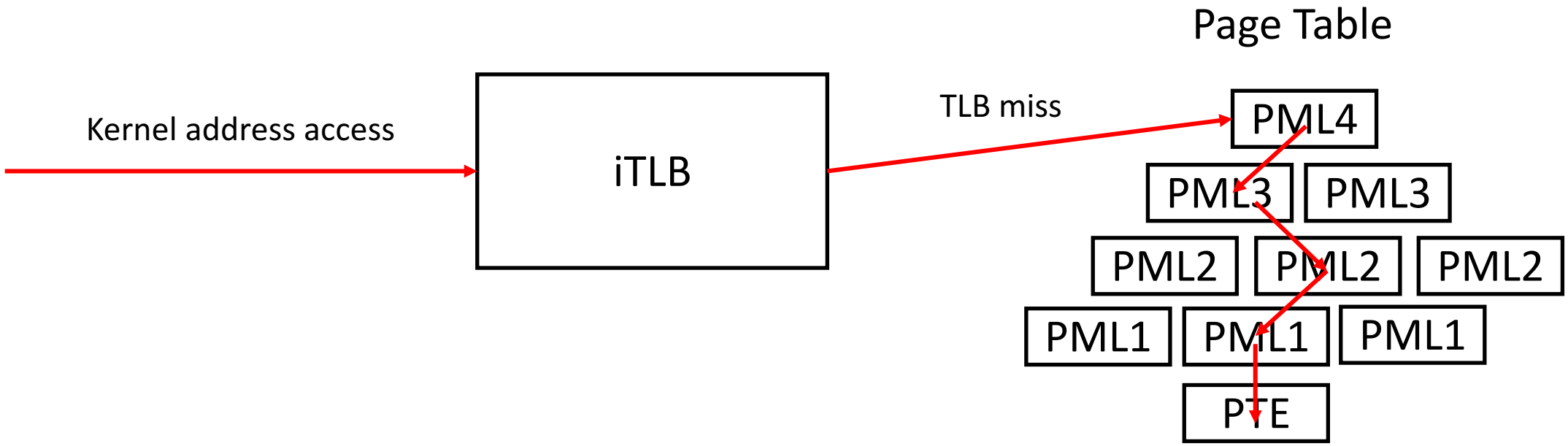
# Path for an Unmapped Page

On the second access, 226 cycles



# Path for an Unmapped Page

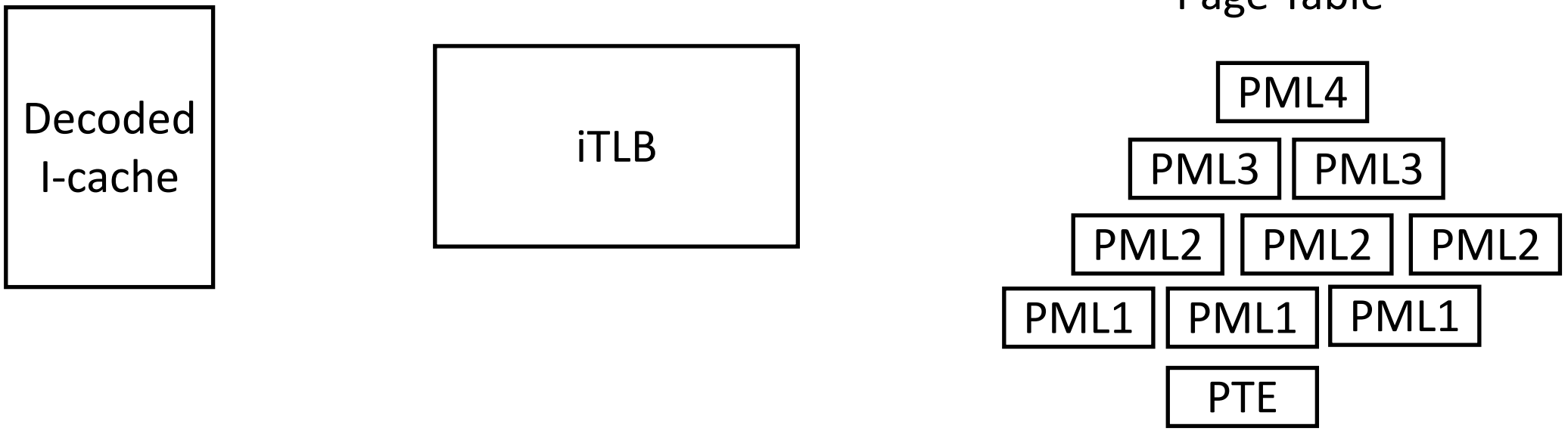
On the second access, 226 cycles




Always do page table walk (slow)

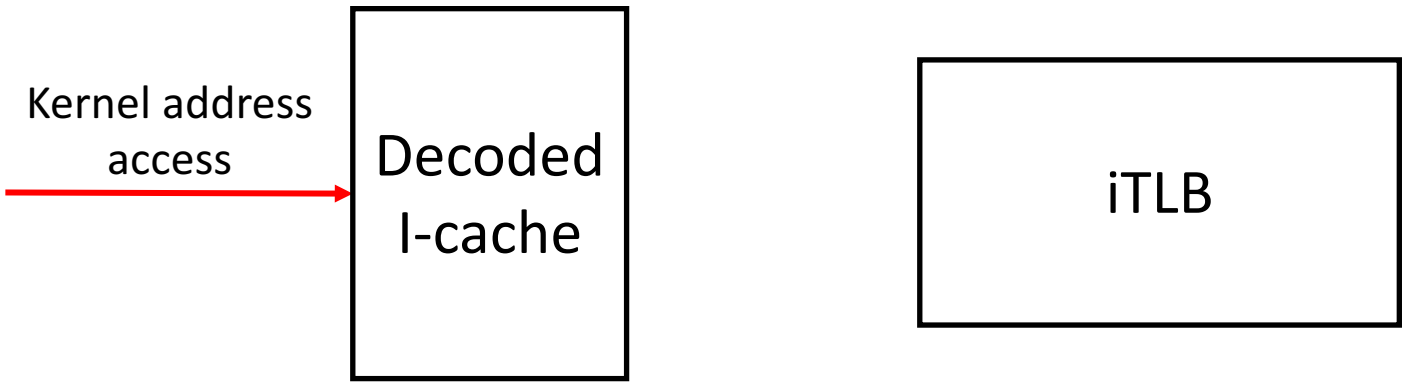
# Path for an Executable Page

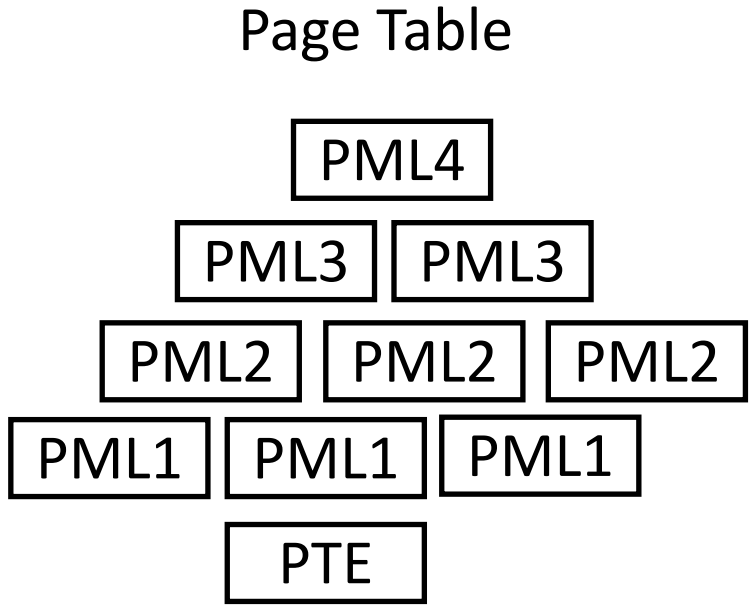
On the first access



# Path for an Executable Page

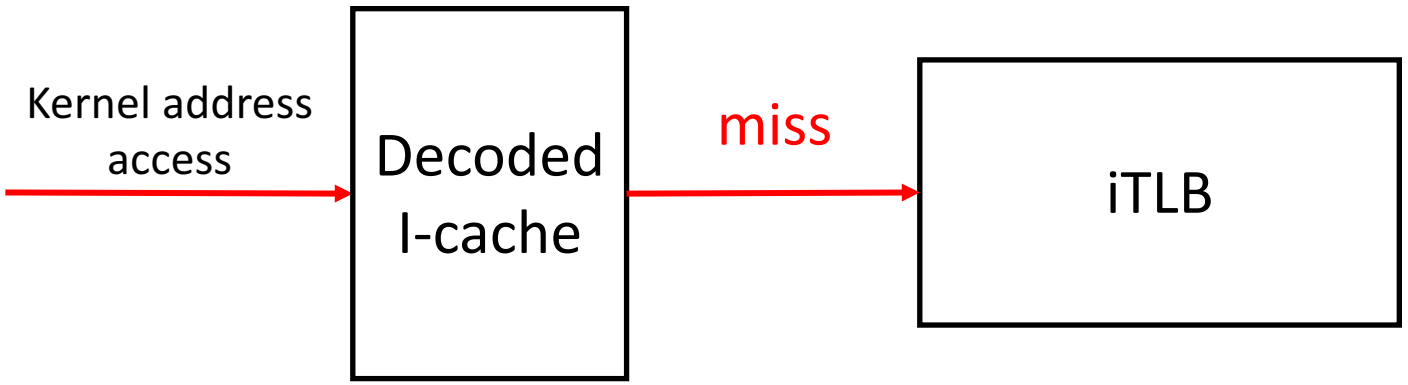
On the first access



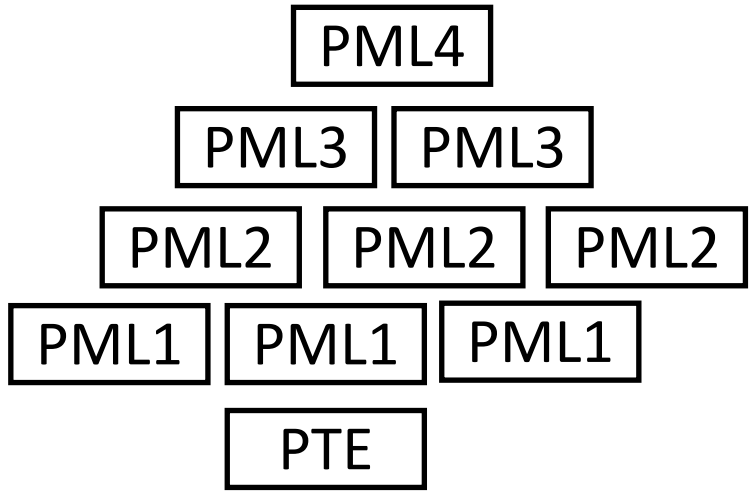



# Path for an Executable Page

On the first access

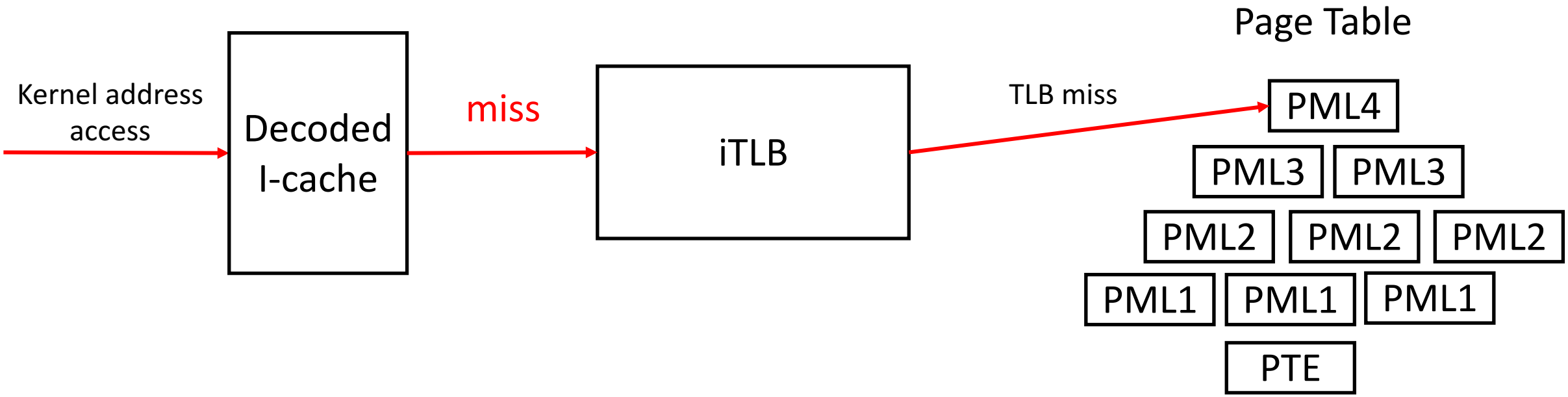



Page Table



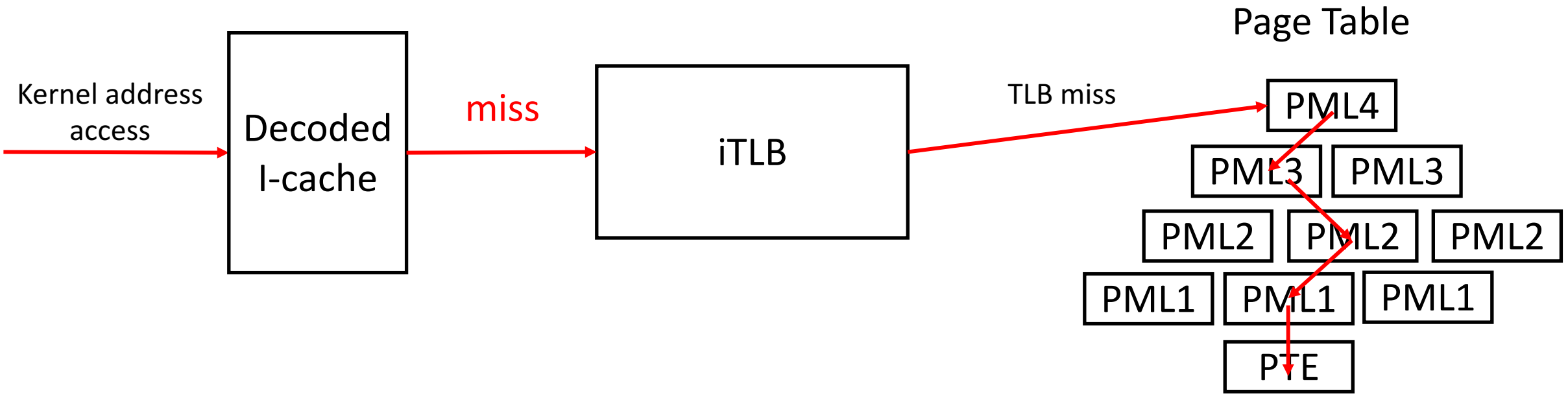
# Path for an Executable Page

On the first access



# Path for an Executable Page

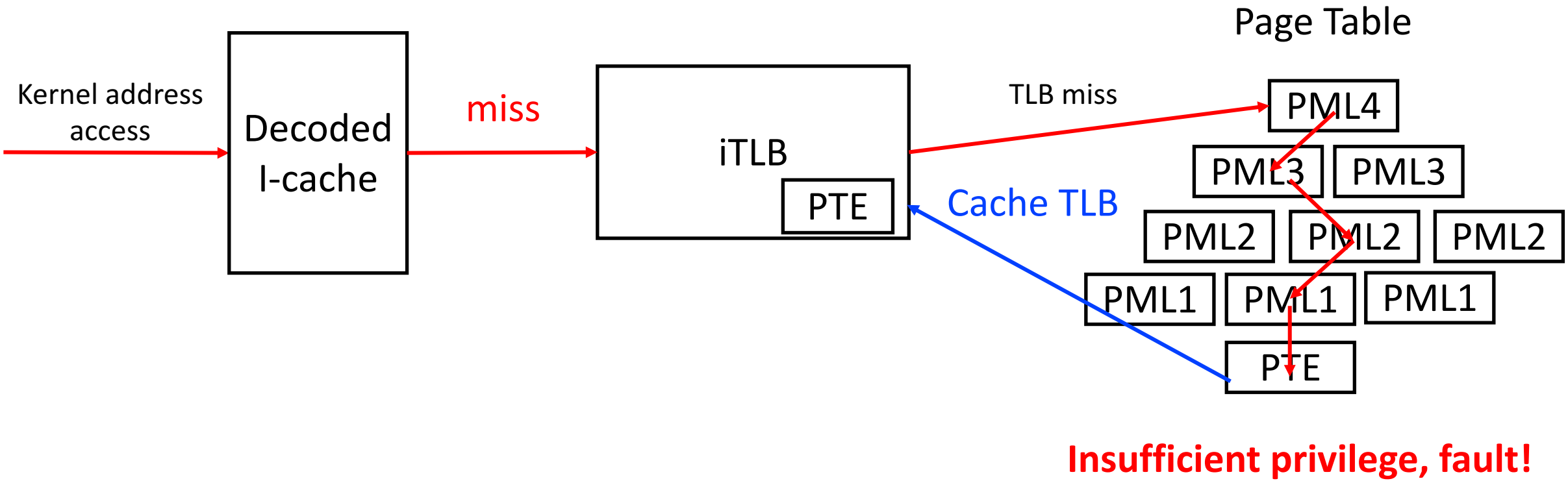
On the first access




**Insufficient privilege, fault!**

# Path for an Executable Page

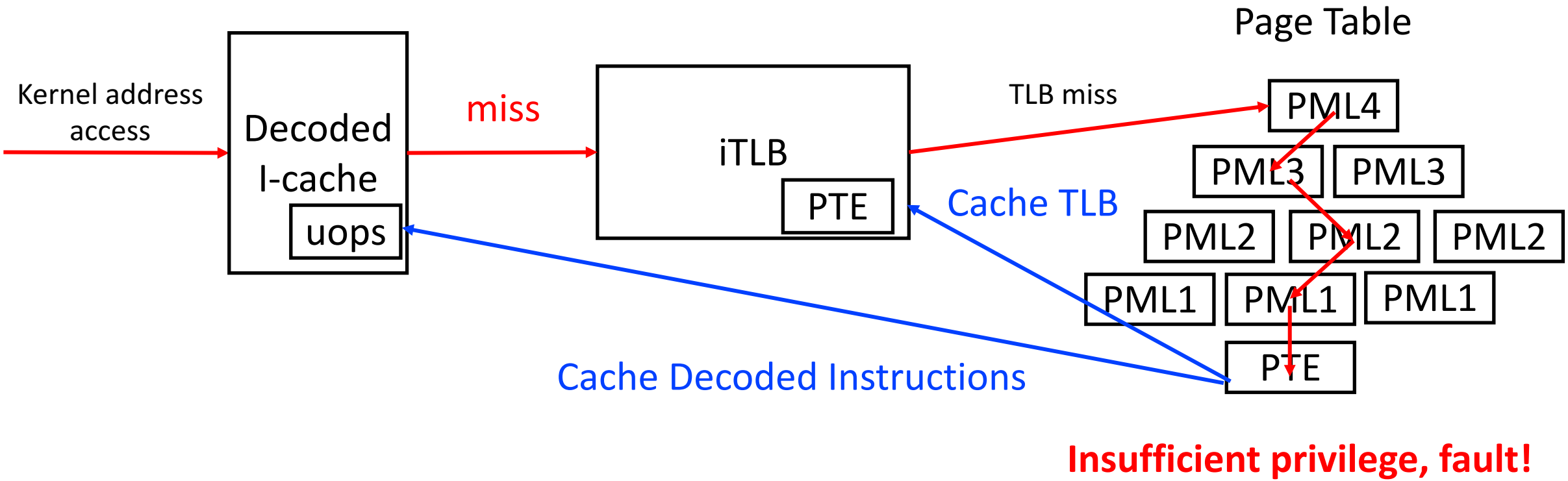
On the first access




**Insufficient privilege, fault!**

# Path for an Executable Page

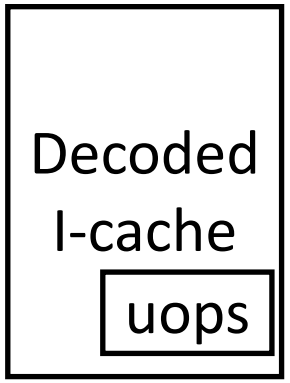
On the first access

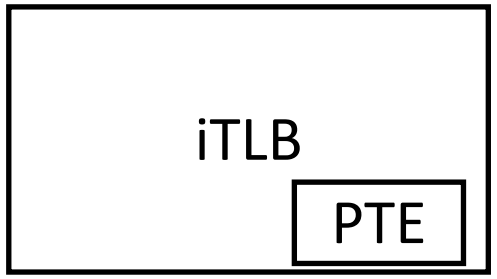


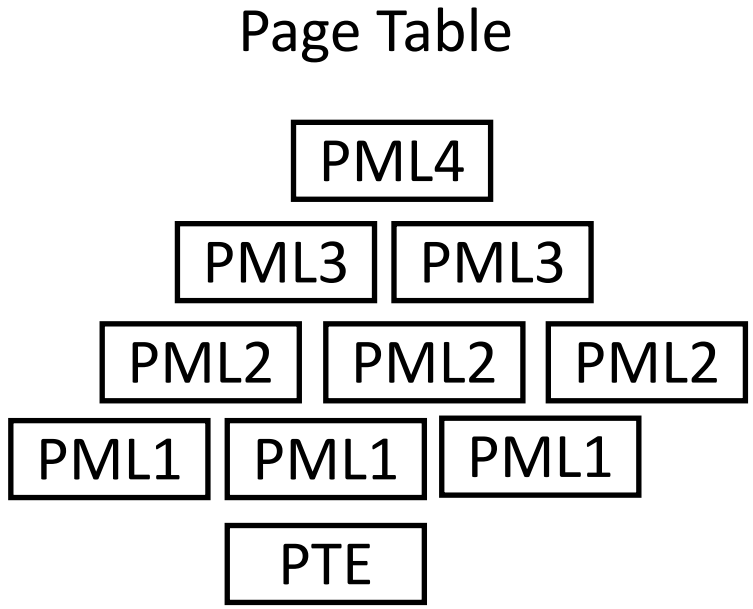

**Insufficient privilege, fault!**

# Path for an Executable Page

On the second access, 181 cycles

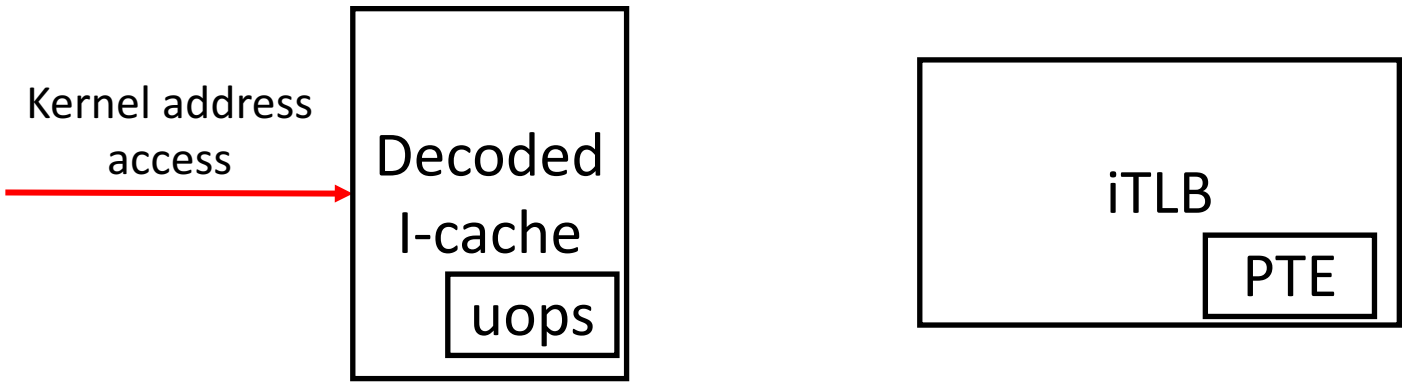


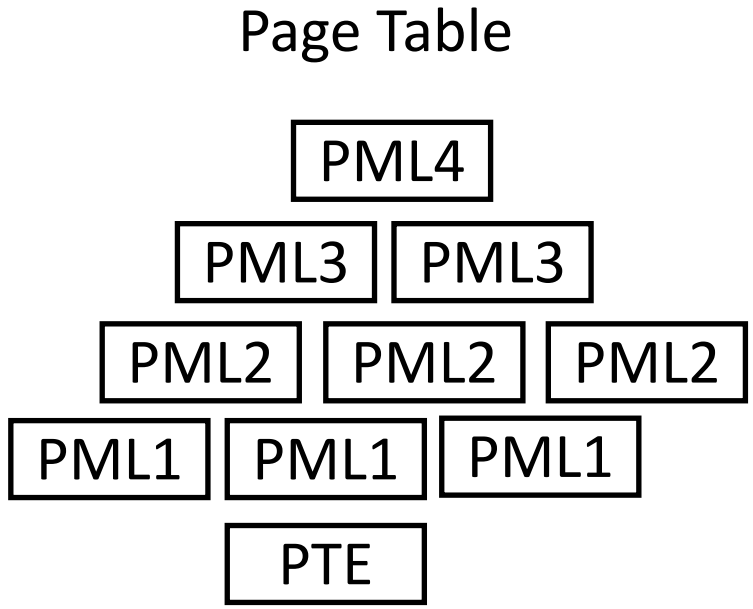



# Path for an Executable Page

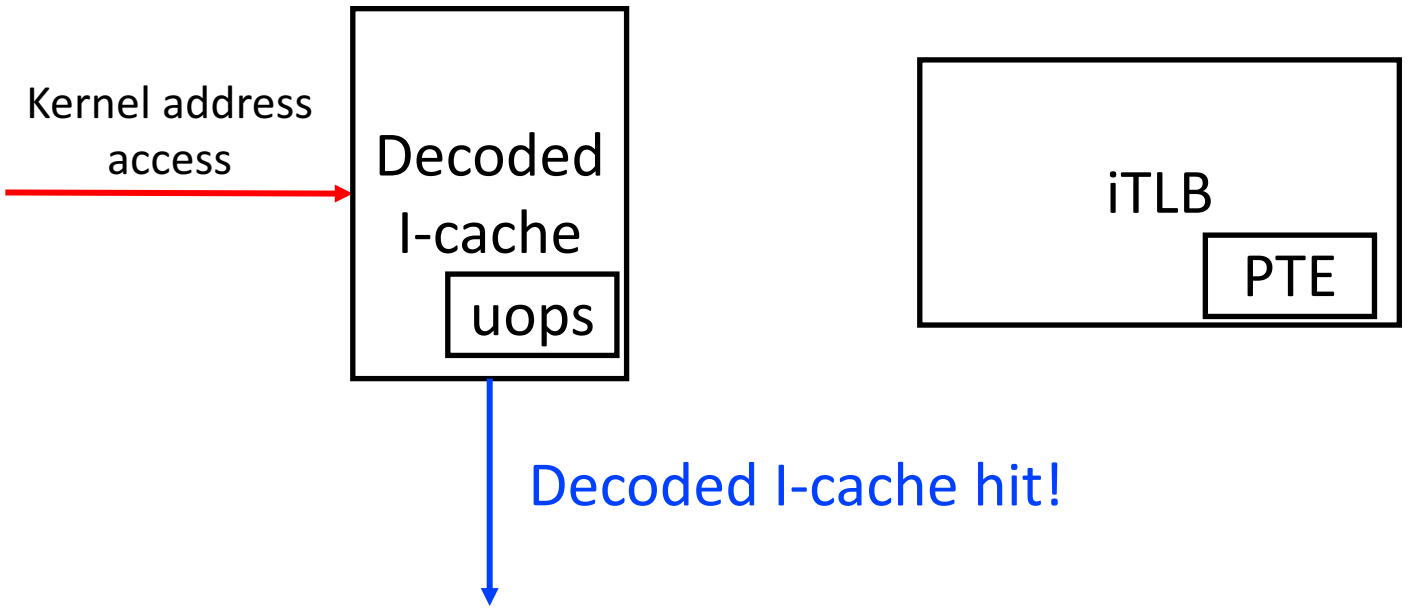
On the second access, 181 cycles

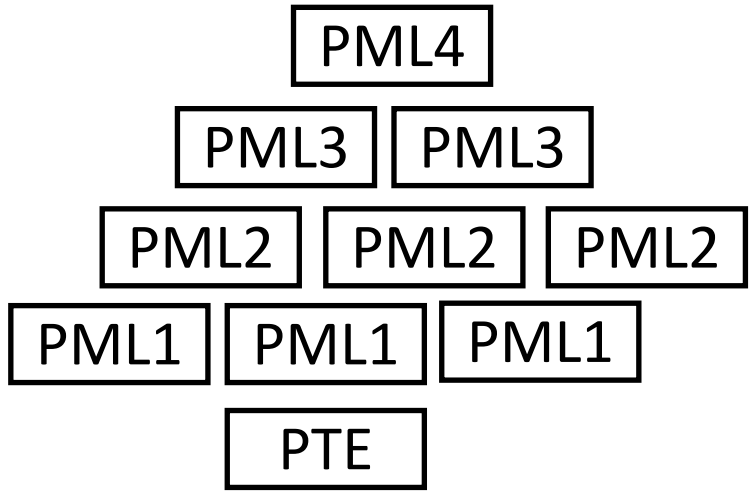


# Path for an Executable Page

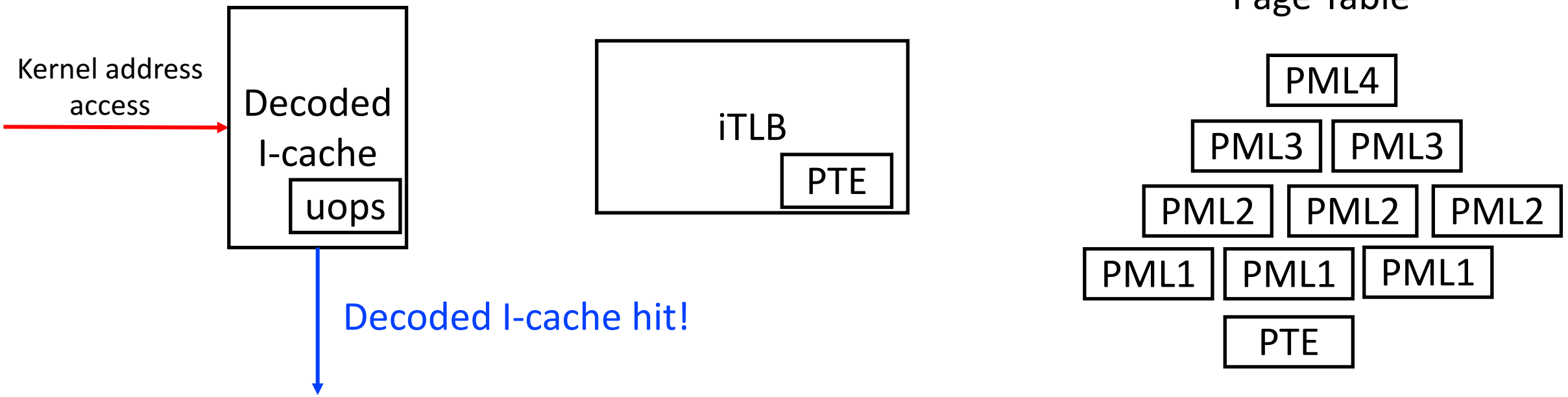
On the second access, 181 cycles



# Path for an Executable Page

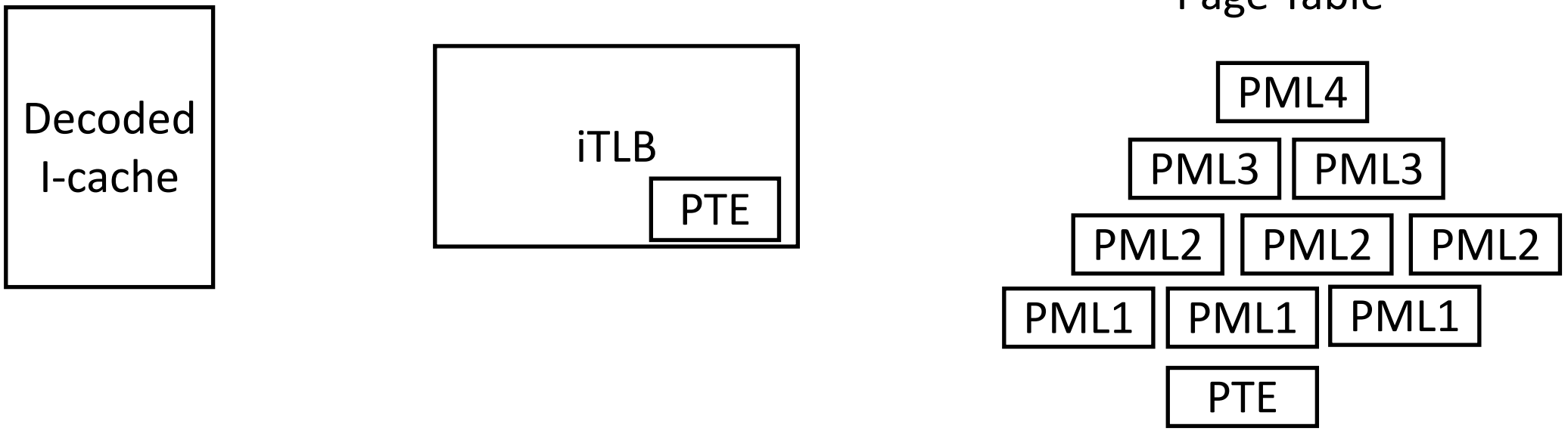
On the second access, 181 cycles




No TLB access, No page table walk (fast)

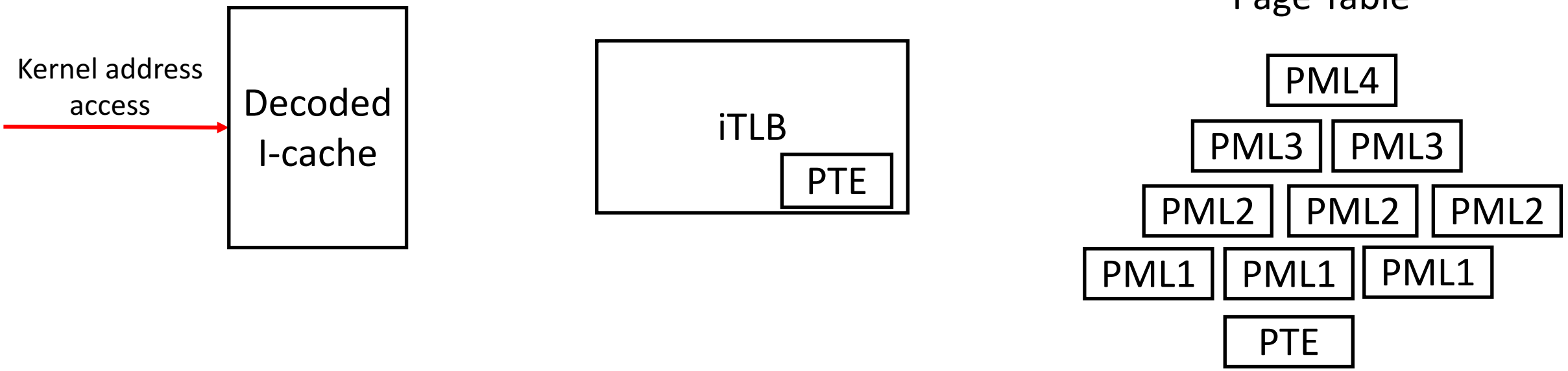
# Path for a non-executable, but mapped Page

On the second access, 226 cycles



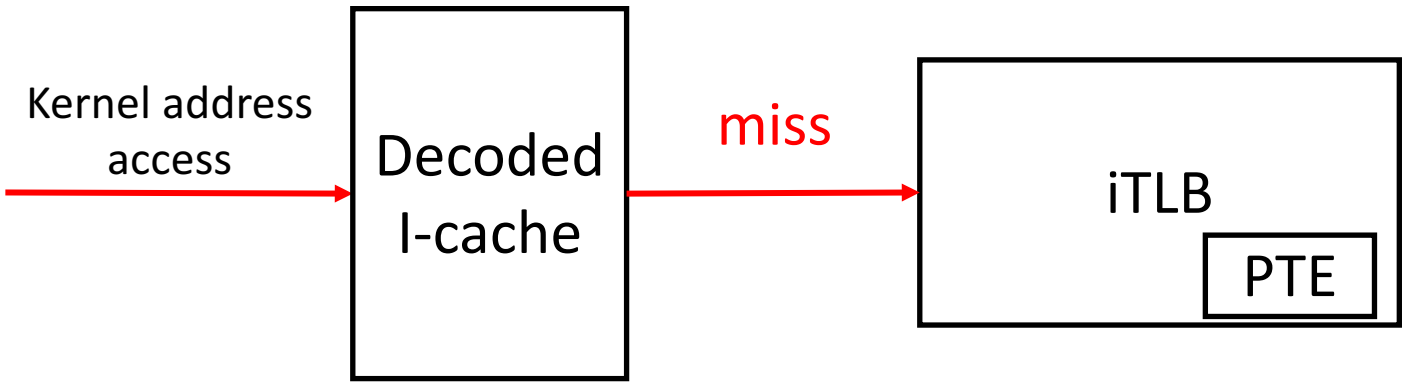
# Path for a non-executable, but mapped Page

On the second access, 226 cycles

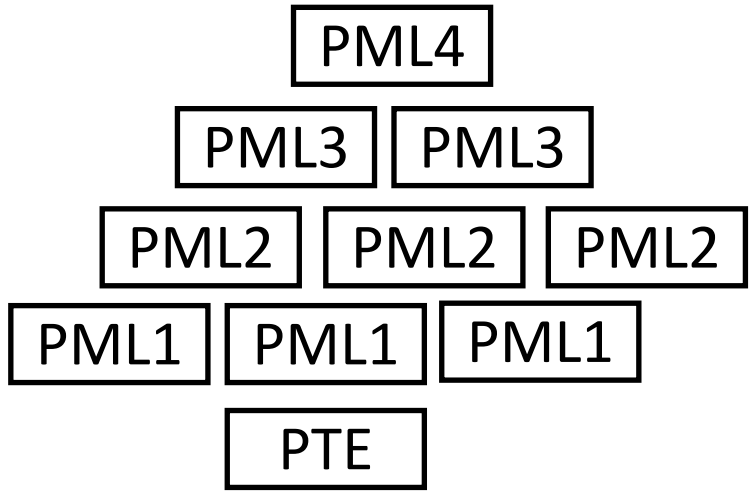


# Path for a non-executable, but mapped Page

On the second access, 226 cycles

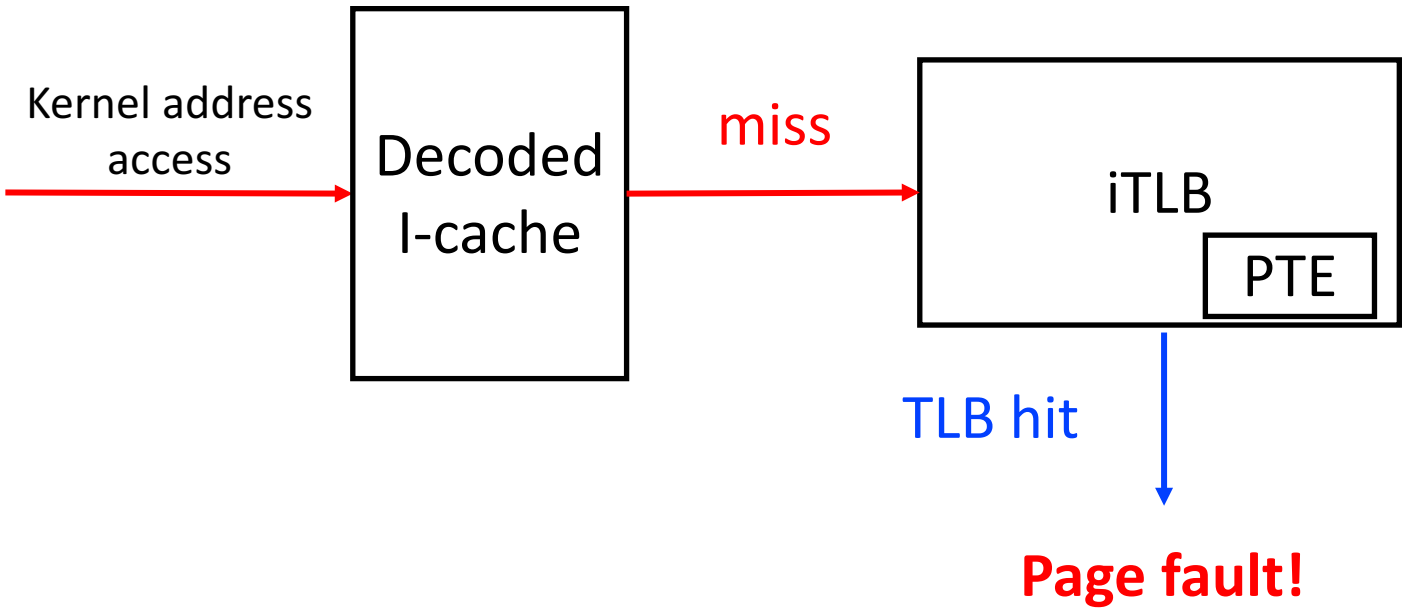


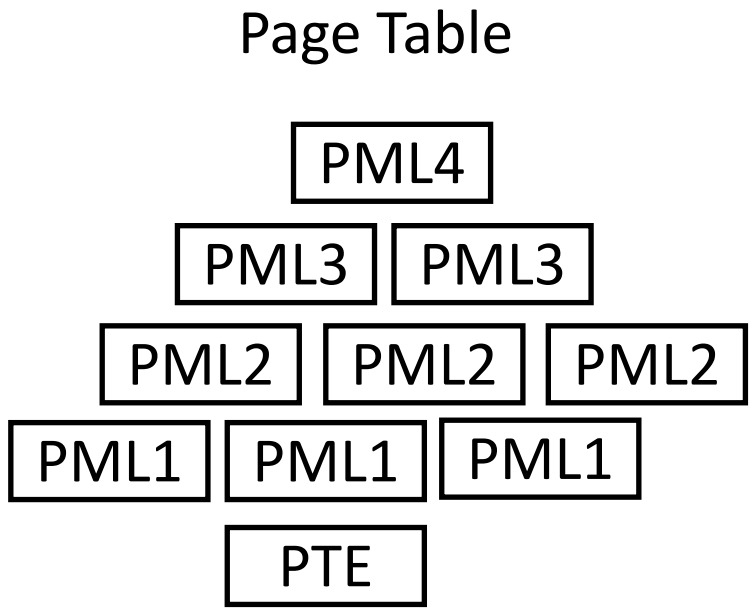

Page Table



# Path for a non-executable, but mapped Page

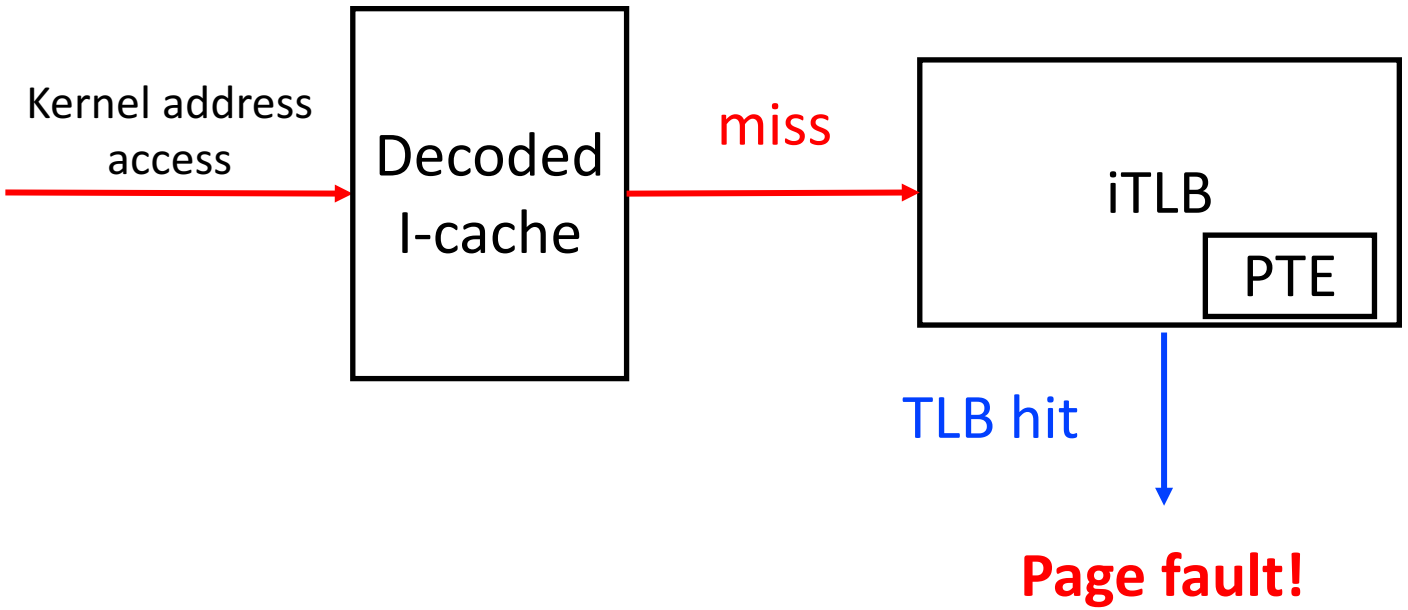
On the second access, 226 cycles

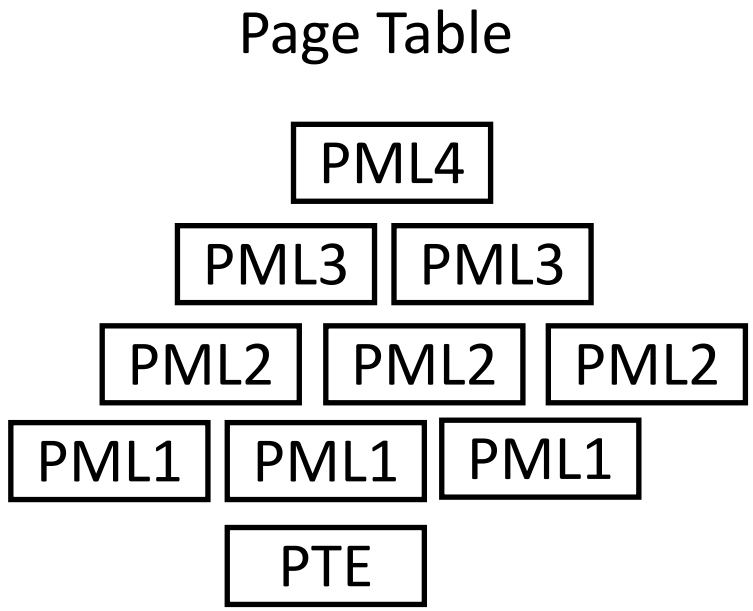


# Path for a non-executable, but mapped Page

On the second access, 226 cycles



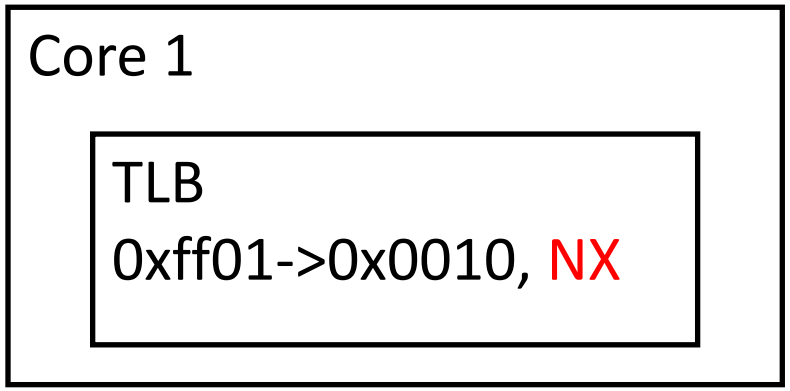

If no page table walk, it should be faster than unmapped (but not!)

# Cache Coherence and TLB

- TLB is not a coherent cache in Intel Architecture

# Cache Coherence and TLB

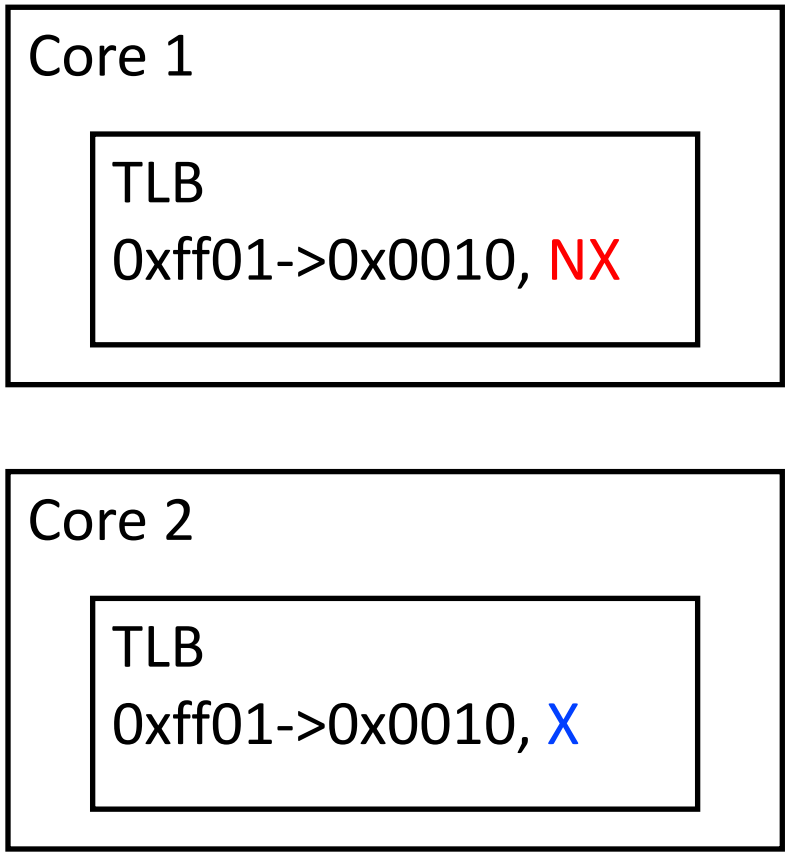
- TLB is not a coherent cache in Intel Architecture




1. Core 1 sets 0xff01 as Non-executable memory

# Cache Coherence and TLB

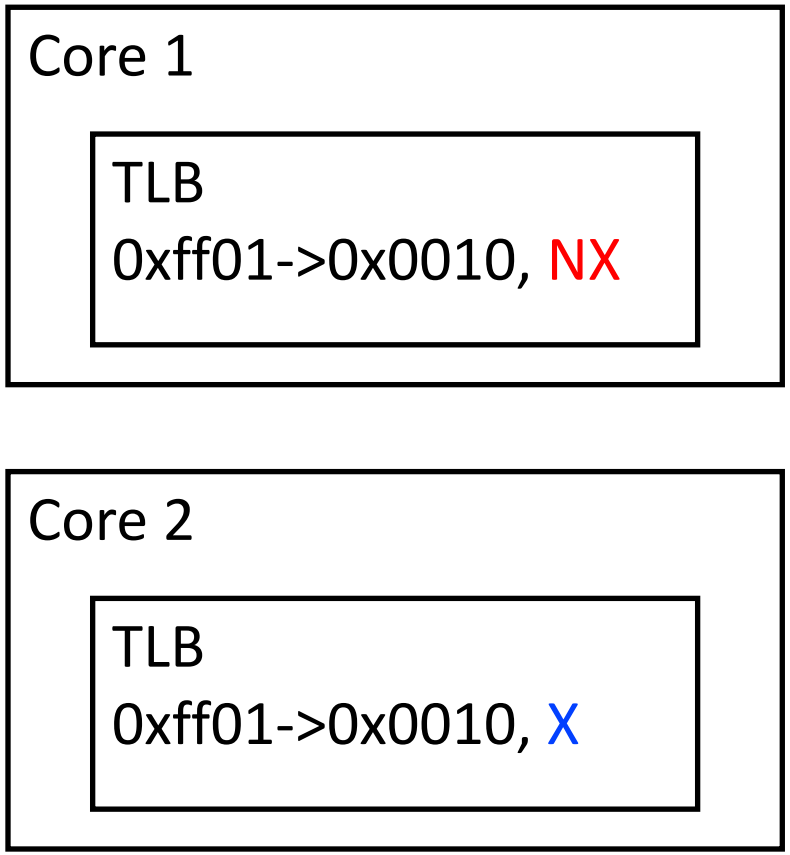
- TLB is not a coherent cache in Intel Architecture




1. Core 1 sets 0xff01 as Non-executable memory
2. Core 2 sets 0xff01 as Executable memory
   No coherency, do not update/invalidate TLB in Core 1

# Cache Coherence and TLB

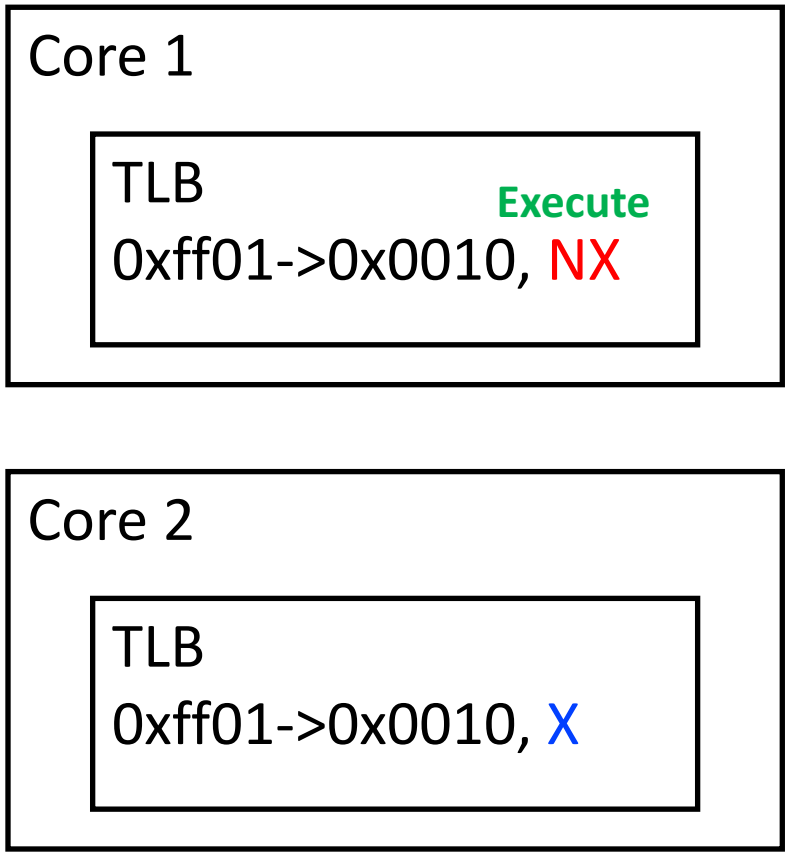
- TLB is not a coherent cache in Intel Architecture




1. Core 1 sets 0xff01 as Non-executable memory
2. Core 2 sets 0xff01 as Executable memory
   No coherency, do not update/invalidate TLB in Core 1
3. Core 1 try to execute on 0xff01 -> fault by NX

# Cache Coherence and TLB

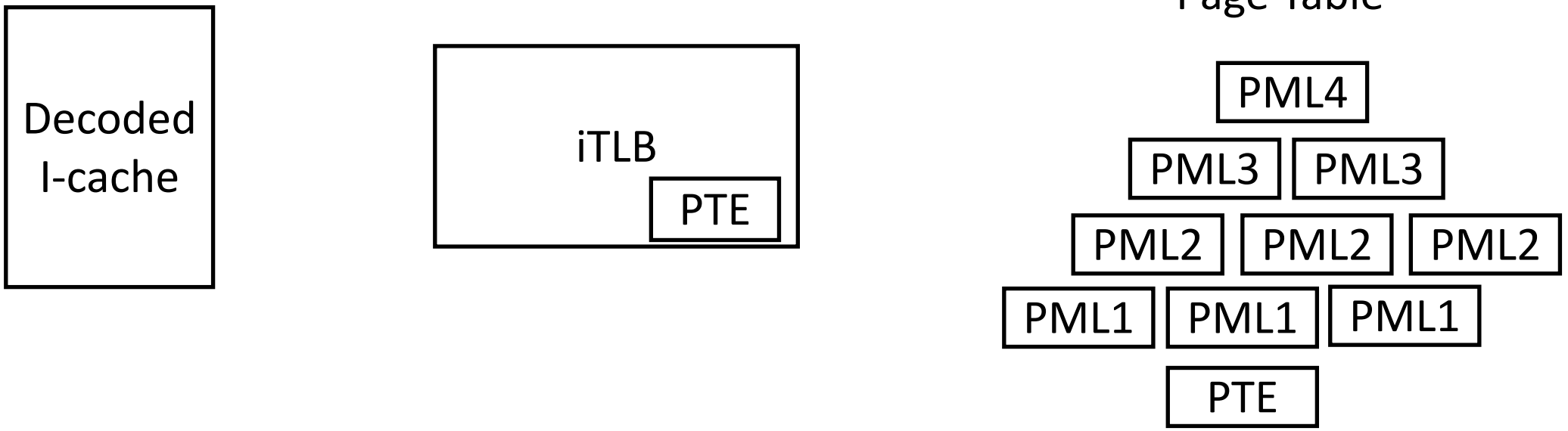
- TLB is not a coherent cache in Intel Architecture




1. Core 1 sets 0xff01 as Non-executable memory
2. Core 2 sets 0xff01 as Executable memory
   No coherency, do not update/invalidate TLB in Core 1
3. Core 1 try to execute on 0xff01 -> fault by NX
4. Core 1 must walk through the page table
   The page table entry is X, update TLB, then execute!

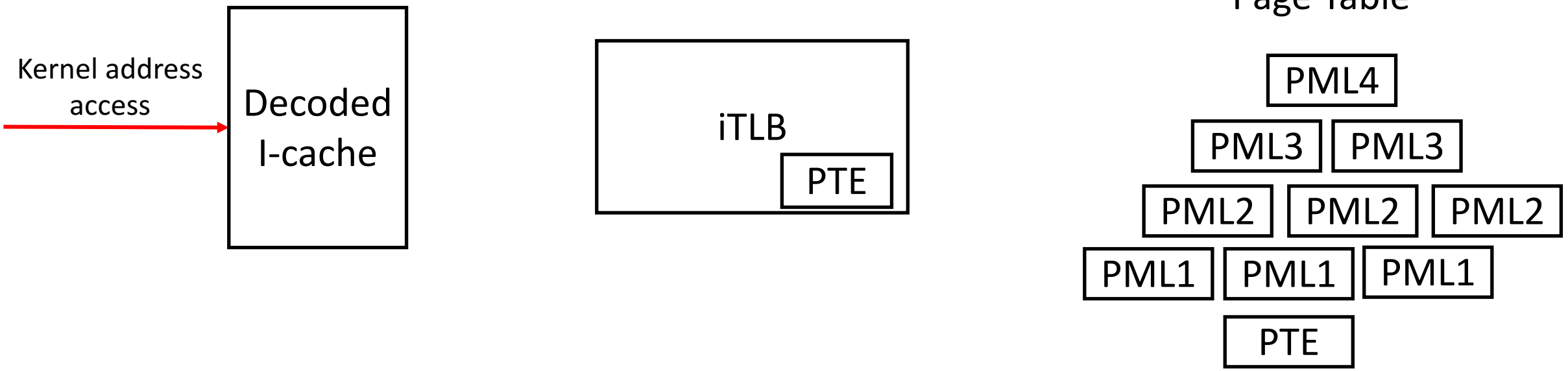
# Path for a Non-executable, but mapped Page

On the second access, 226 cycles



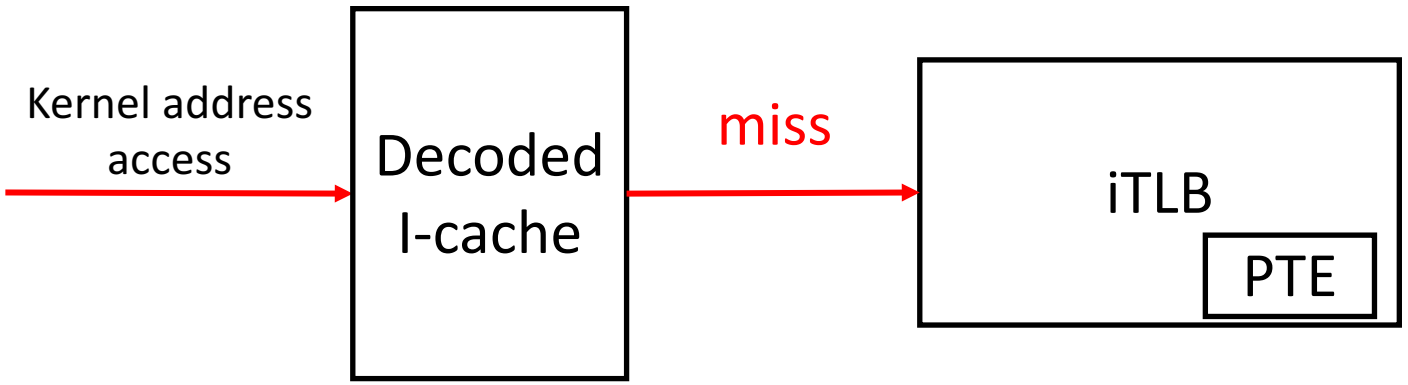
# Path for a Non-executable, but mapped Page

On the second access, 226 cycles

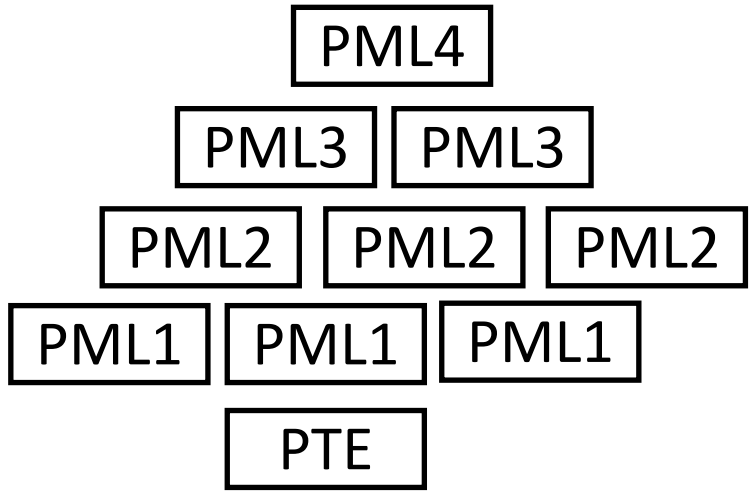


# Path for a Non-executable, but mapped Page

On the second access, 226 cycles

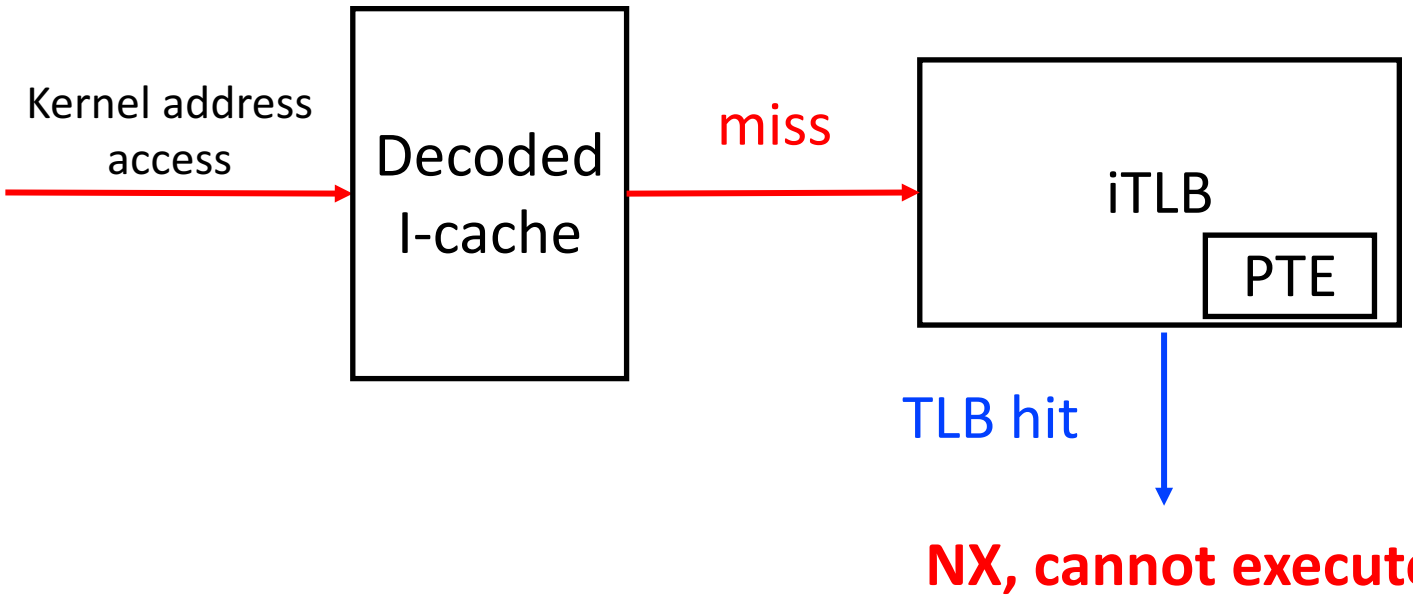


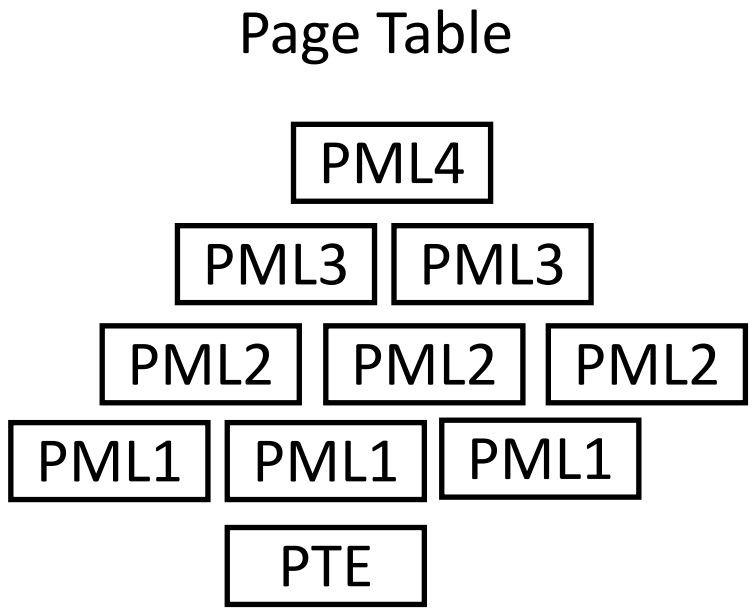

Page Table



# Path for a Non-executable, but mapped Page

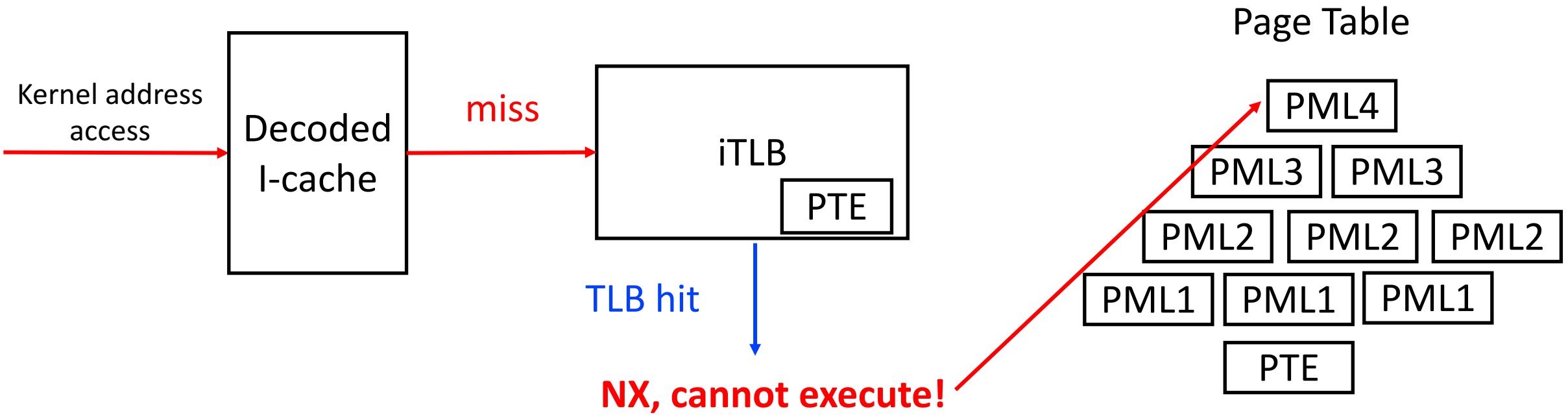
On the second access, 226 cycles



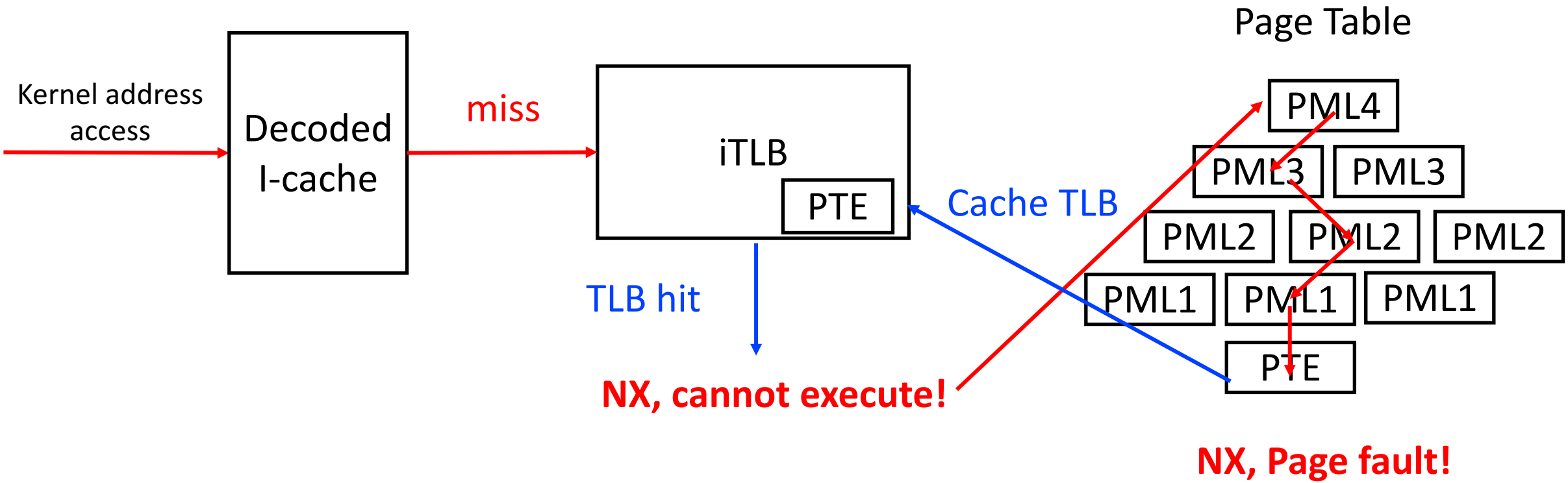
# Path for a Non-executable, but mapped Page

On the second access, 226 cycles



# Path for a Non-executable, but mapped Page

On the second access, 226 cycles



# Root-cause of Timing Side Channel (X/NX)

- For executable / non-executable addresses

| Fast Path (X) | Slow Path (NX) | Slow Path (U) |
|---|---|---|
| 1. Jmp into the Kernel addr<br>2. Decoded I-cache hits<br>3. Page fault! | 1. Jmp into the kernel addr<br>2. iTLB hit<br>3. Protection check fails, page table walk.<br>4. Page fault! | 1. Jmp into the kernel addr<br>2. iTLB miss<br>3. Walks through page table<br>4. Page fault! |
| Cycles: 181 | Cycles: 226 | Cycles: 226 |

- Decoded i-cache generates timing side channel

# Countermeasures?

- Modifying CPU to eliminate timing channels
  - Difficult to be realized ☹
- Turning off TSX
  - Cannot be turned off in software manner (neither from MSR nor from BIOS)
- Coarse-grained timer?
  - A workaround could be having another thread to measure the timing indirectly (e.g., counting i++;)

# Countermeasures?

- Using separated page tables for kernel and user processes
  - High performance overhead (~30%) due to frequent TLB flush
    - TLB flush on every copy_to_user()
- Fine-grained randomization
  - Compatibility issues on memory alignment, etc.
- Inserting fake mapped / executable pages between the maps
  - Adds some false positives to the DrK Attack

# Conclusion

- Intel TSX makes cache side-channel less noisy
  - Suppress OS Exception
- Timing side channel can distinguish X / NX / U pages
  - dTLB (for Mapped & Unmapped)
  - Decoded i-cache (for eXecutable / non-executable)
  - Work across 3 different architectures, commodity OSes, and Amazon EC2
- Current KASLR is not as secure as expected

# Any Questions?

- Try DrK at
  - https://github.com/sslab-gatech/DrK