Introduction to Microarchitectural Attacks

Daniel Gruss

June 18, 2019

Graz University of Technology

Software-based Side-Channel Attacks

- security and privacy rely on secrets (unknown to attackers)
- secrets can leak through side channels

Software-based Side-Channel Attacks

- security and privacy rely on secrets (unknown to attackers)
- secrets can leak through side channels
- software-based $\rightarrow$ no physical access

1337 4242

FOOD CACHE

Revolutionary concept!

Store your food at home, never go to the grocery store during cooking.

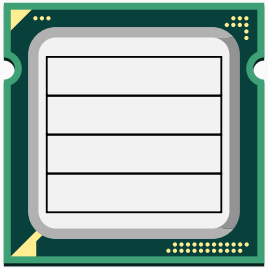
Can store **ALL** kinds of food.

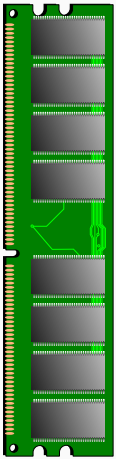
ONLY TODAY INSTEAD OF ~~$1,300~~

$1,299

ORDER VIA PHONE: +555 12345

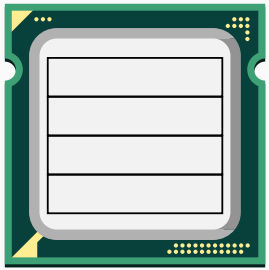
```
printf("%d", i);
printf("%d", i);
```

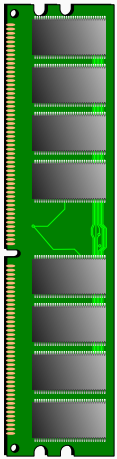




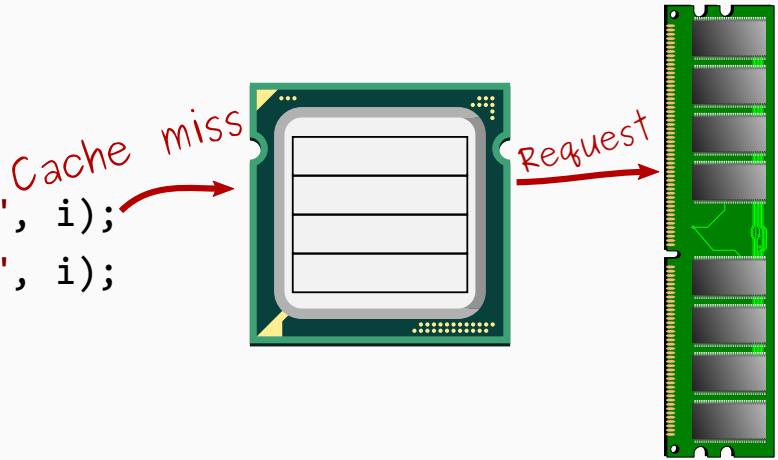
```
printf("%d", i);
printf("%d", i);
```

Cache miss

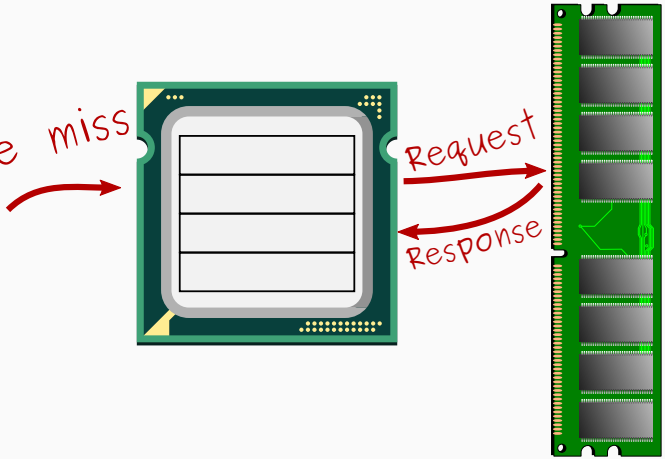




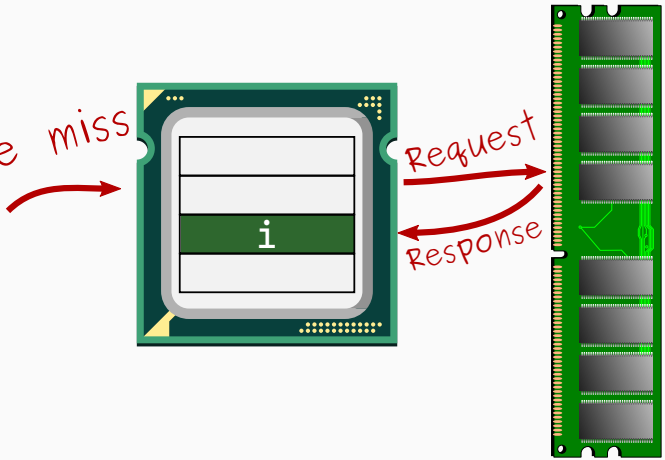
```
printf("%d", i);
printf("%d", i);
```

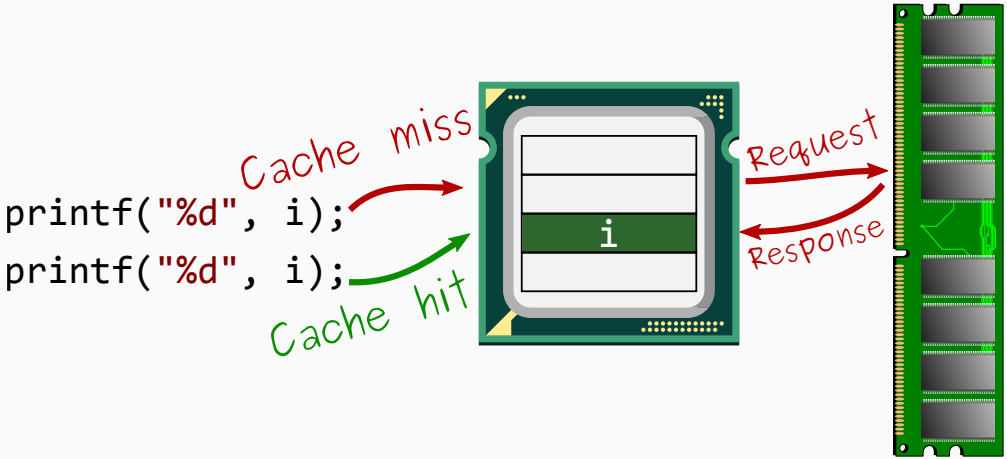


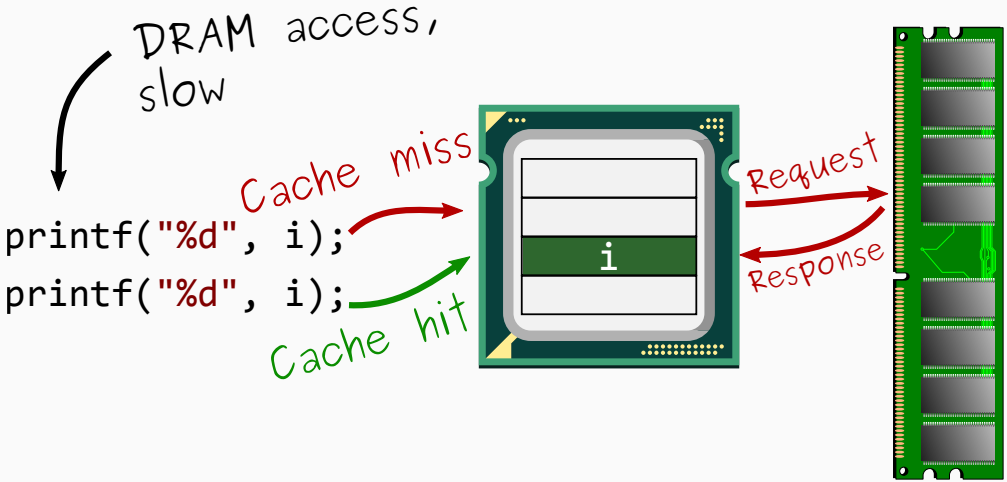
```
printf("%d", i);
printf("%d", i);
```

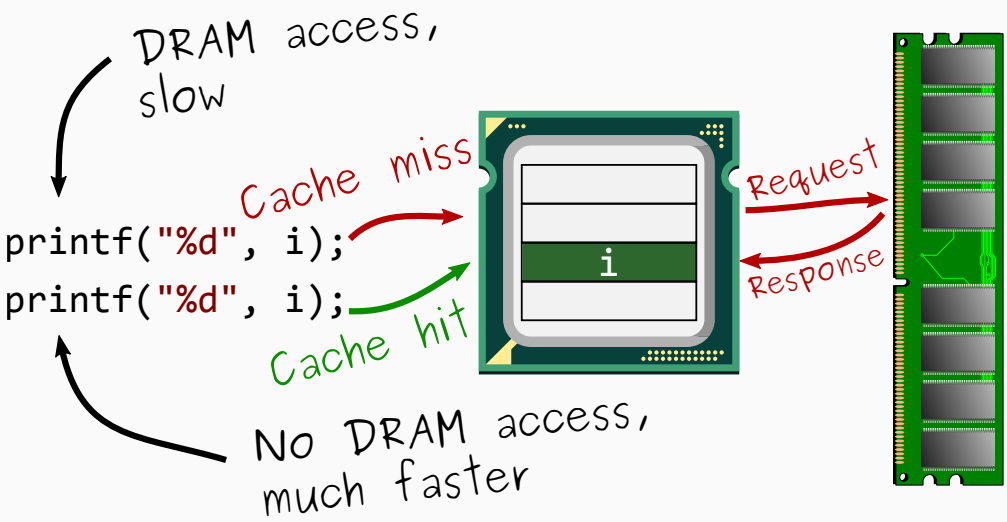


```
printf("%d", i);
printf("%d", i);
```




printf("%d", i);
printf("%d", i);
Cache miss
Cache hit
i
Request
Response


DRAM access, slow
printf("%d", i);
printf("%d", i);
Cache miss
Cache hit
i
Request
Response

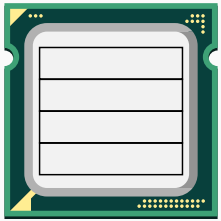

DRAM access, slow
printf("%d", i);
printf("%d", i);
Cache miss
Cache hit
i
Request
Response
No DRAM access, much faster

Shared Memory

ATTACKER

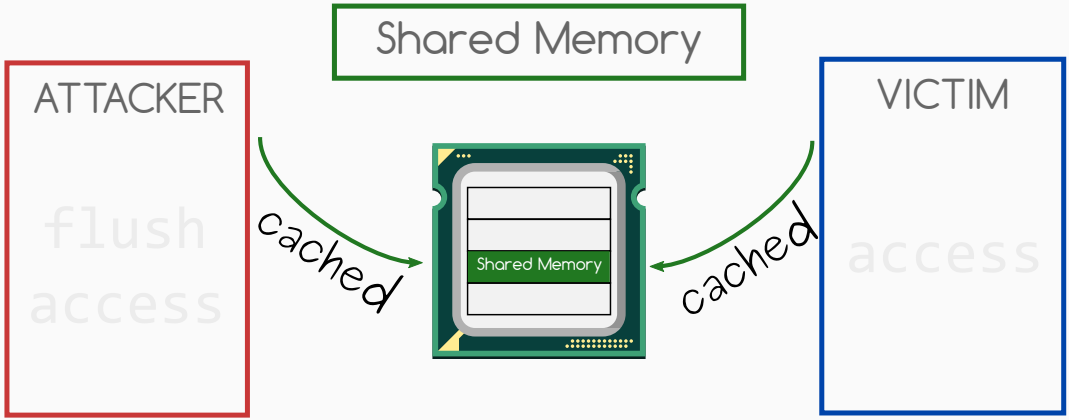
flush

access



VICTIM

access

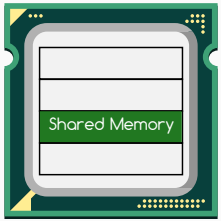

Shared Memory
ATTACKER
flush
access
cached
Shared Memory
cached
VICTIM
access

Shared Memory

ATTACKER

flush

access



VICTIM

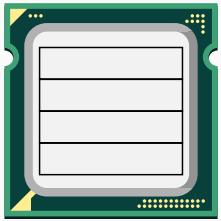
access

Shared Memory

ATTACKER

flush

access



VICTIM

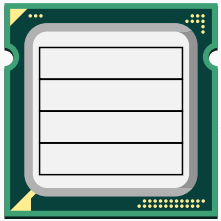
access

Shared Memory

ATTACKER

flush

access



VICTIM

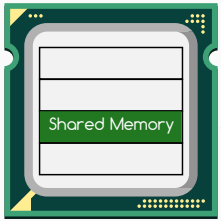
access

Shared Memory

ATTACKER

flush

access



VICTIM

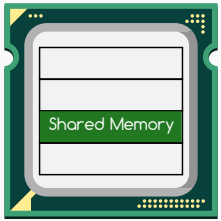
access

Shared Memory

ATTACKER

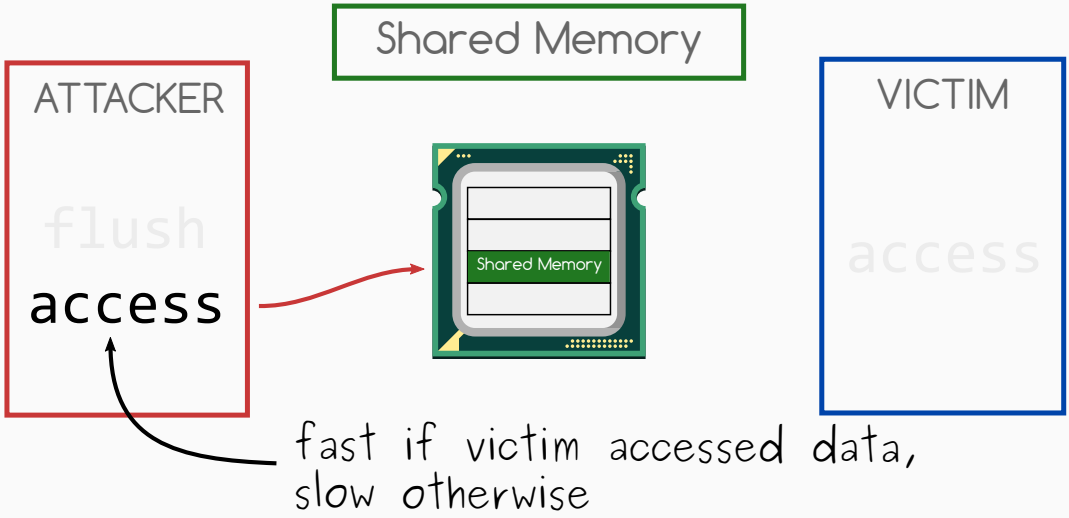
flush

access



VICTIM

access


Shared Memory
ATTACKER
flush
access
Shared Memory
VICTIM
access
fast if victim accessed data,
slow otherwise


FAIL
failblog.org

- Very short timings
- `rdtsc` instruction: “cycle-accurate” timestamps

```
[...]
rdtsc
function()
rdtsc
[...]
```

What are we measuring?

- Do you measure what you *think* you measure?
- *Out-of-order* execution → what is really executed?

```
rdtsc
function()
[...]
rdtsc
```

```
rdtsc
[...]
rdtsc
function()
```

```
rdtsc
rdtsc
function()
[...]
```

Accurate Microarchitecture Timing

- use pseudo-serializing instruction `rdtscp` (recent CPUs)

- use pseudo-serializing instruction `rdtscp` (recent CPUs)
- and/or use serializing instructions like `cpuid`

- use pseudo-serializing instruction `rdtscp` (recent CPUs)
- and/or use serializing instructions like `cpuid`
- and/or use fences like `mfence`

- use pseudo-serializing instruction `rdtscp` (recent CPUs)
- and/or use serializing instructions like `cpuid`
- and/or use fences like `mfence`

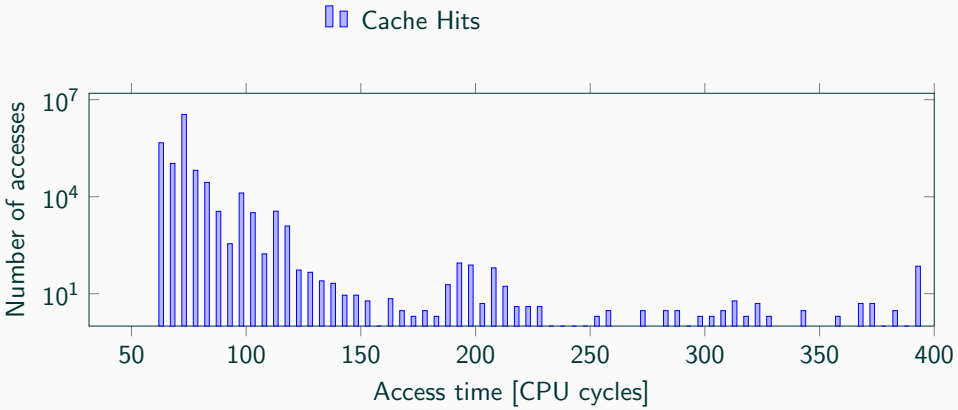
Intel, *How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures White Paper*, December 2010.

AUGUST 22, 2018 BY BRUCE

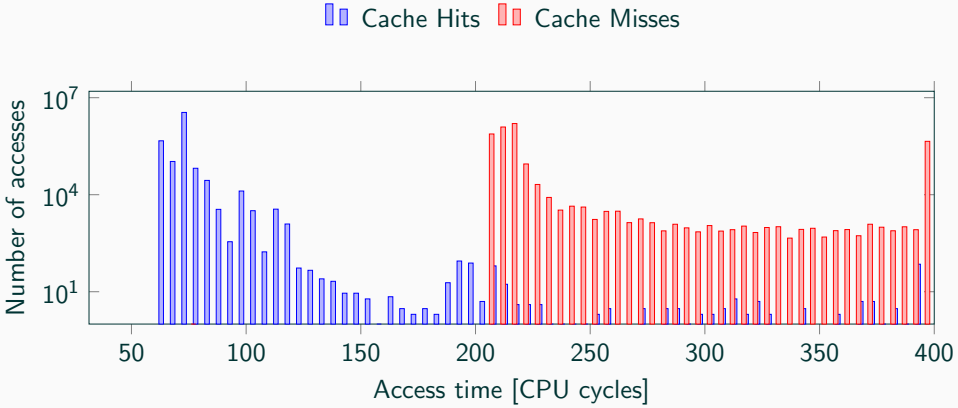
Intel Publishes Microcode Security Patches, No Benchmarking Or Comparison Allowed!

UPDATE: **Intel has resolved their microcode licensing issue which I complained about in this blog post.** The new license text is here.

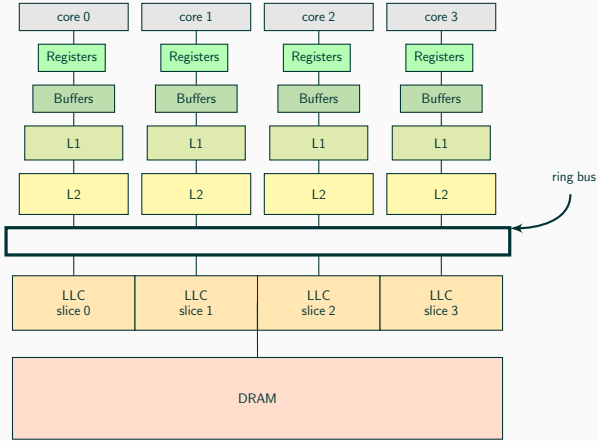
Memory Access Latency



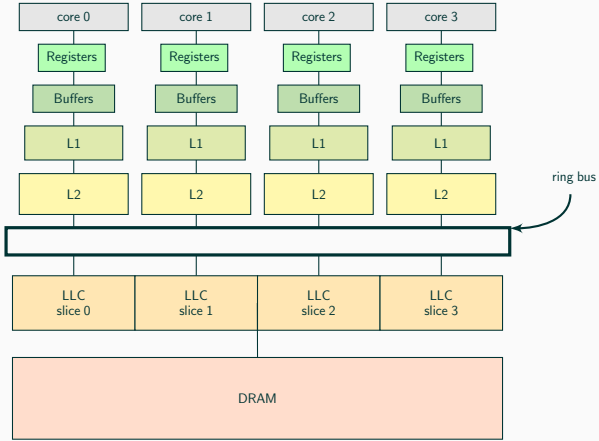
Memory Access Latency



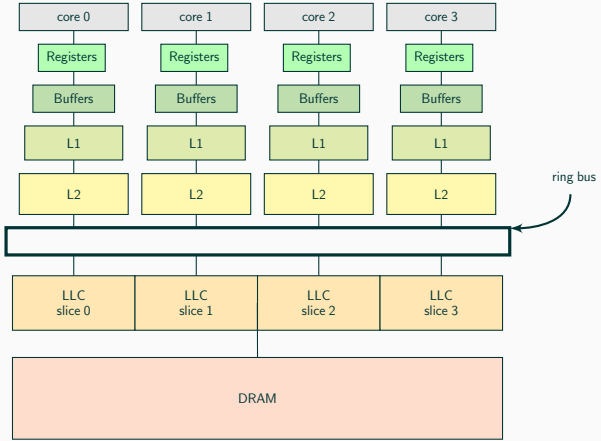
Memory Hierarchy



Memory Hierarchy

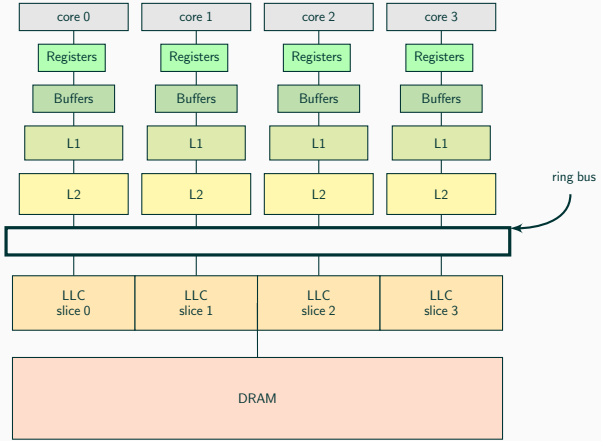


Memory Hierarchy



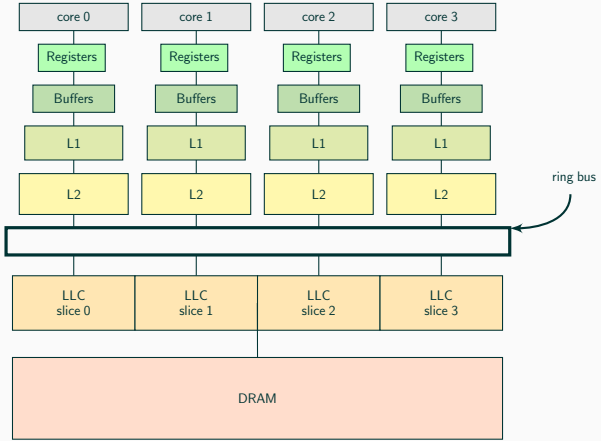
- L1 and L2 are private

Memory Hierarchy



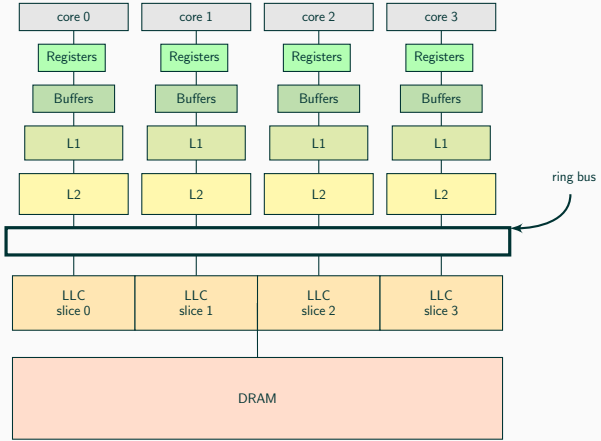
- L1 and L2 are private
- last-level cache:

Memory Hierarchy



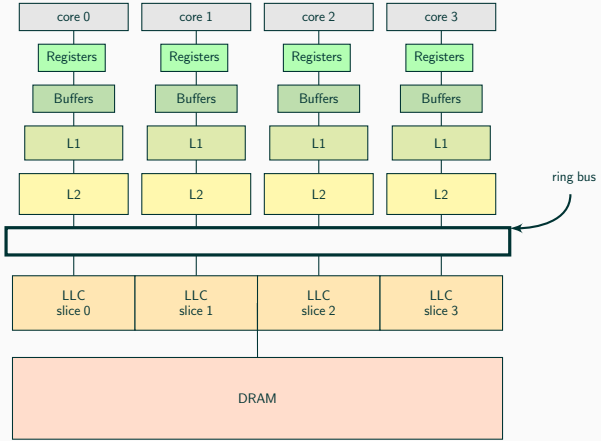
- L1 and L2 are private
- last-level cache:
  - divided in **slices**

Memory Hierarchy



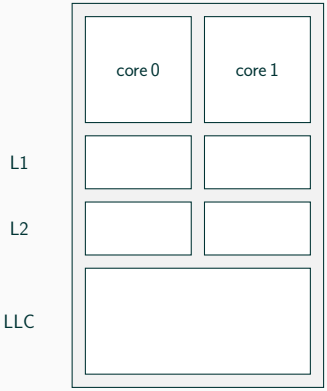
- L1 and L2 are private
- last-level cache:
  - divided in **slices**
  - **shared** across cores

Memory Hierarchy



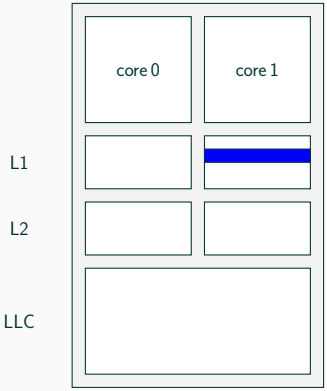
- L1 and L2 are private
- last-level cache:
  - divided in **slices**
  - **shared** across cores
  - **inclusive**

Inclusive property



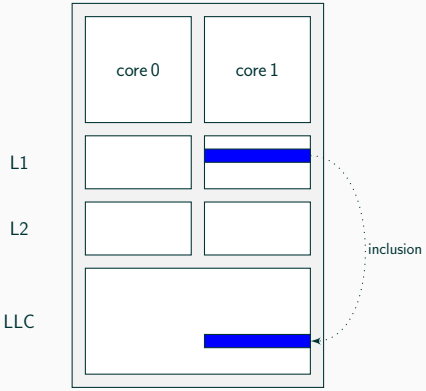
- **inclusive** LLC: superset of L1 and L2

Inclusive property



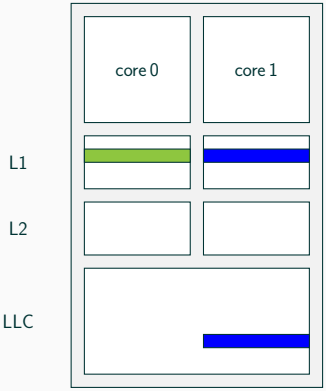
- **inclusive** LLC: superset of L1 and L2

Inclusive property



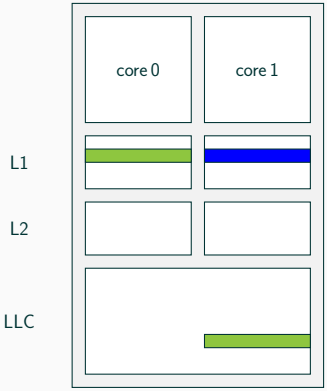
- **inclusive** LLC: superset of L1 and L2

Inclusive property



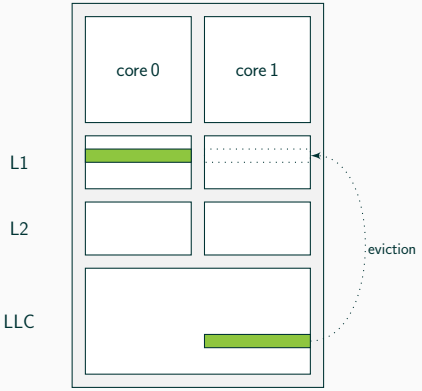
- **inclusive** LLC: superset of L1 and L2

Inclusive property



- **inclusive** LLC: superset of L1 and L2
- data evicted from the LLC is also evicted from L1 and L2

Inclusive property



- **inclusive** LLC: superset of L1 and L2
- data evicted from the LLC is also evicted from L1 and L2
- a core can **evict lines** in the private L1 **of another core**

Side-channel attack on user input

- Locate **key-dependent** memory accesses

Side-channel attack on user input

- Locate **key-dependent** memory accesses
- How?

Profiling Phase

- Preprocessing step to find exploitable addresses automatically

Profiling Phase

- Preprocessing step to find exploitable addresses automatically
  - w.r.t. “events” (keystrokes, encryptions, ...)

Profiling Phase

- Preprocessing step to find exploitable addresses automatically
  - w.r.t. “events” (keystrokes, encryptions, ...)
  - called “Cache Template”

Profiling Phase

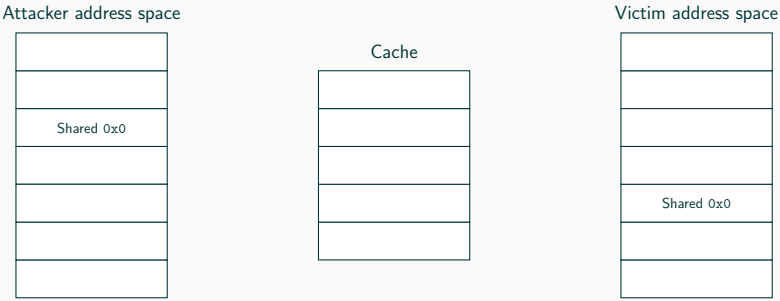
- Preprocessing step to find exploitable addresses automatically
  - w.r.t. “events” (keystrokes, encryptions, ...)
  - called “Cache Template”

Profiling Phase

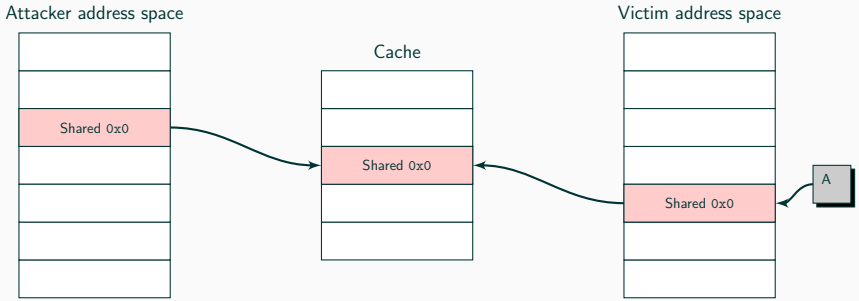
- Preprocessing step to find exploitable addresses automatically
  - w.r.t. “events” (keystrokes, encryptions, ...)
  - called “Cache Template”

Exploitation Phase

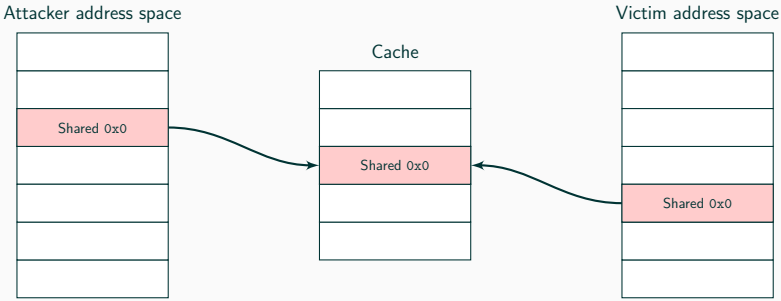
- Monitor exploitable addresses



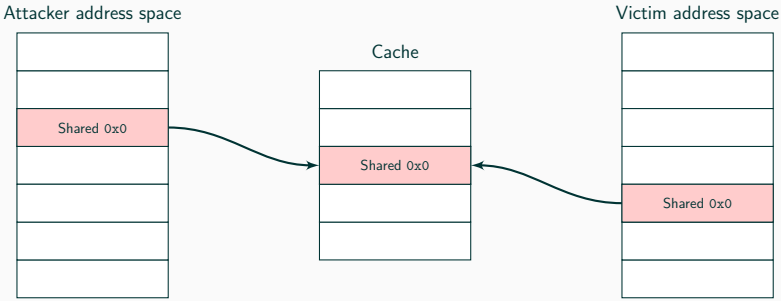
Cache is empty



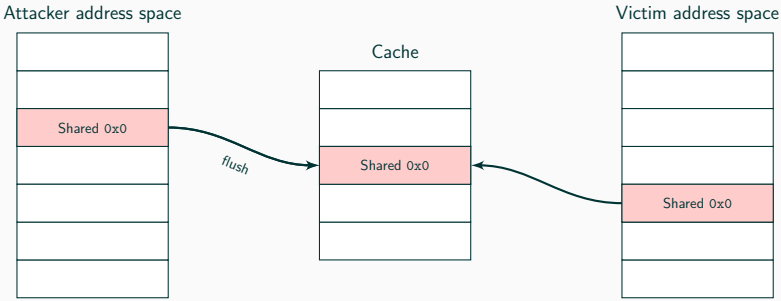
Attacker triggers an event



Attacker checks one address for cache hits ("Reload")

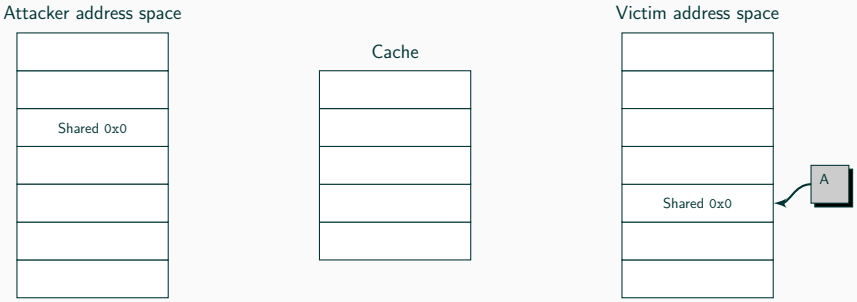


Update number of cache hits per event

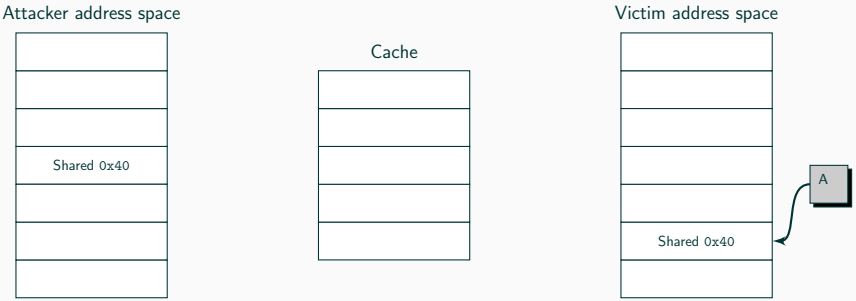


Attacker flushes shared memory

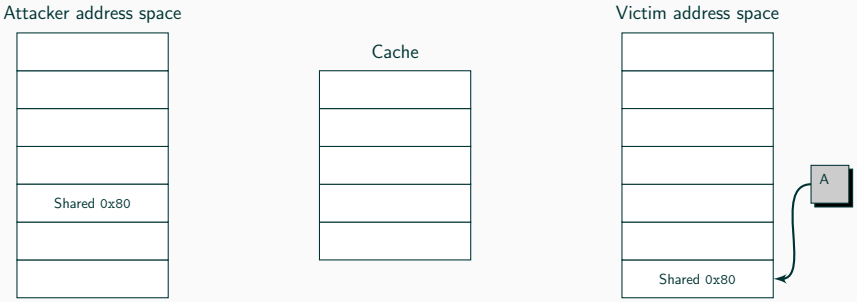
Profiling Phase



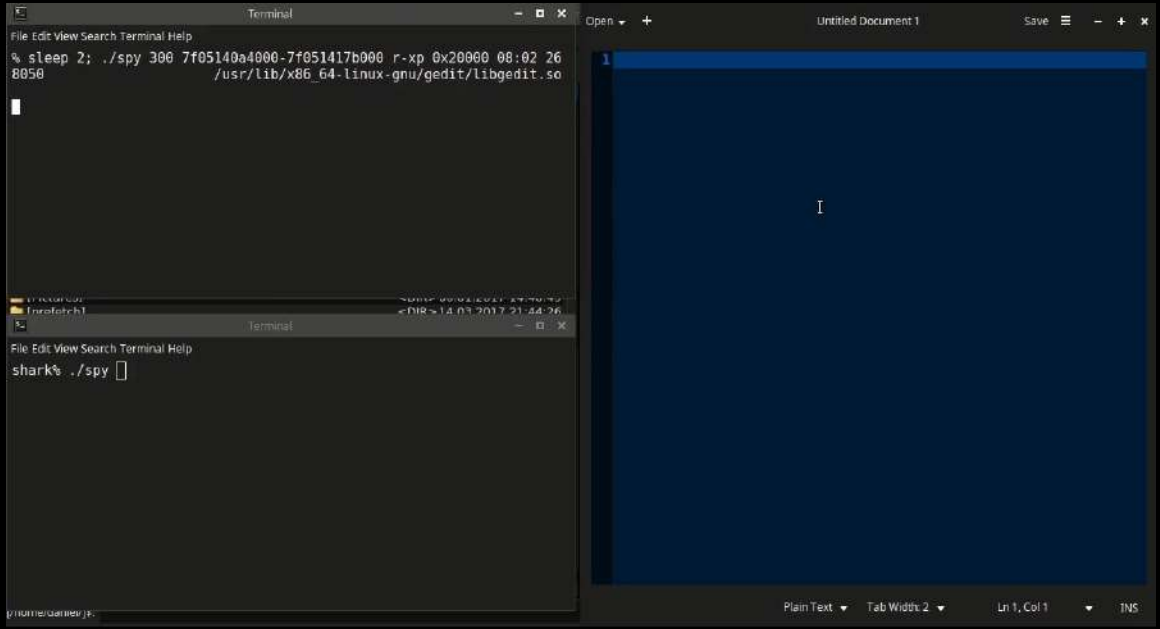
Repeat for higher accuracy



Continue with next address



Continue with next address


Terminal
File Edit View Search Terminal Help
% sleep 2; ./spy 300 7f05140a4000-7f051417b000 r-xp 0x20000 08:02 26
8850 /usr/lib/x86_64-linux-gnu/gedit/libgedit.so
Terminal
File Edit View Search Terminal Help
shark% ./spy
Open
Untitled Document 1
Save
1
Plain Text
Tab Width: 2
Ln 1, Col 1
INS

Profiling Phase: 1 Event, 1 Address

Profiling Phase: 1 Event, 1 Address

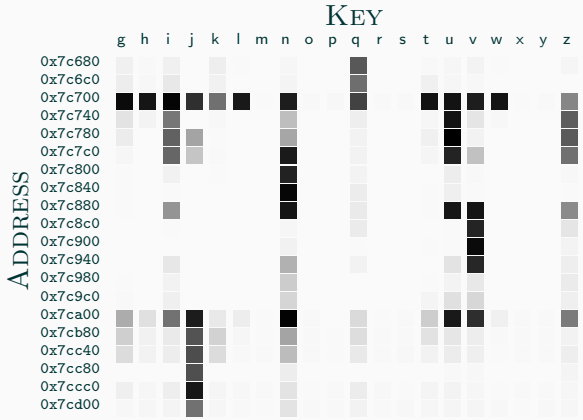
Example: Cache Hit Ratio for (`0x7c800`, `n`): 200 / 200

Profiling Phase: All Events, 1 Address

Example: Cache Hit Ratio for $(\texttt{0x7c800}, \texttt{u})$: 13 / 200

Distinguish `n` from other keys by monitoring `0x7c800`

Profiling Phase: All Events, All Addresses

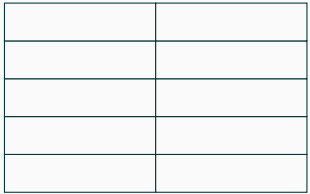


Directly mapped cache

Directly mapped cache

Memory Address

Cache



Directly mapped cache

Memory Address

Cache

Tag	Data

Directly mapped cache

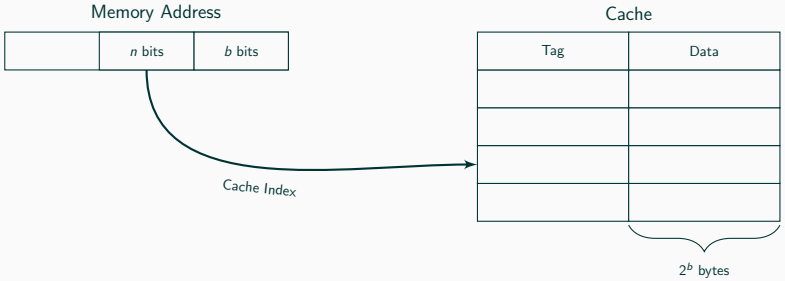
Memory Address

Cache

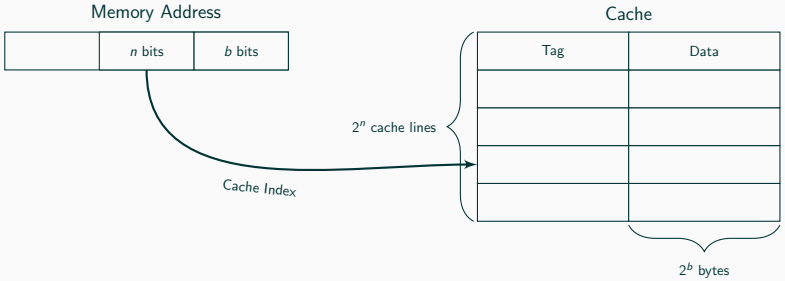
Tag	Data

Directly mapped cache

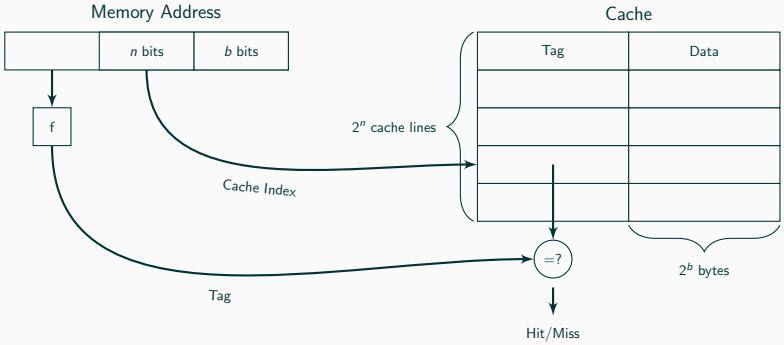
Directly mapped cache



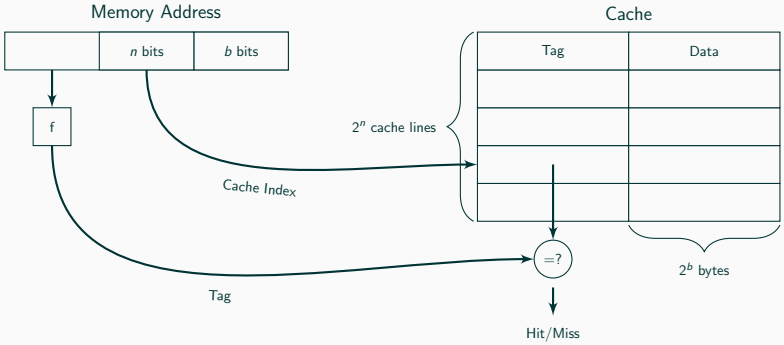
Directly mapped cache



Directly mapped cache

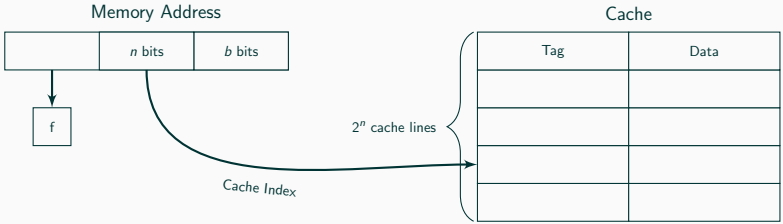


Directly mapped cache

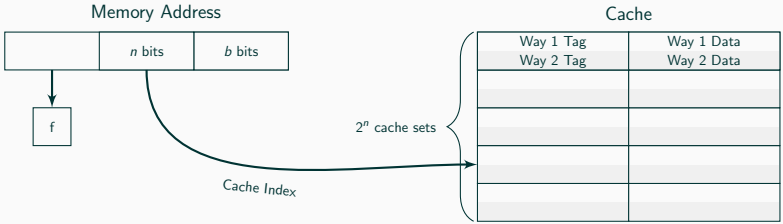


Problem: working on congruent addresses

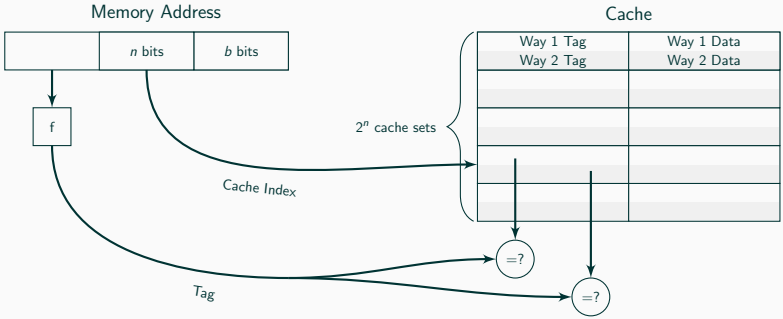
2-way set associativity



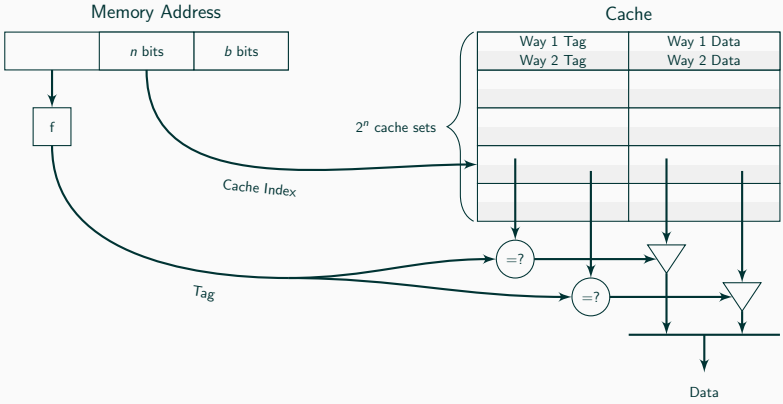
2-way set associativity



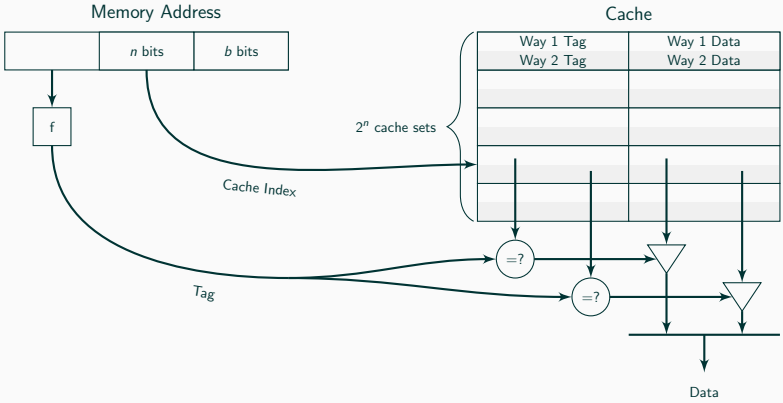
2-way set associativity



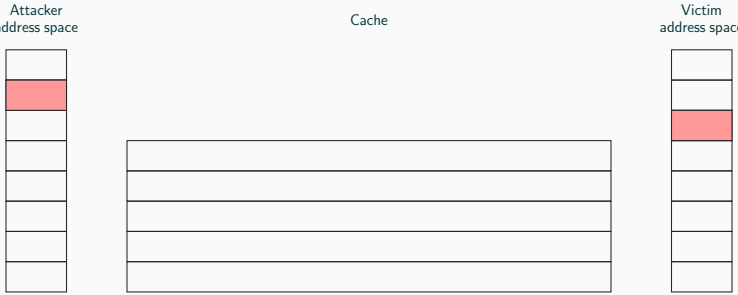
2-way set associativity

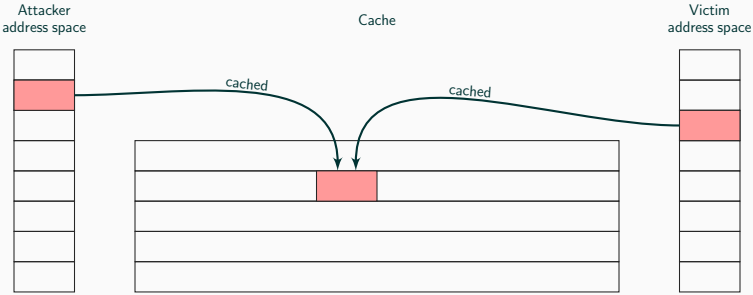


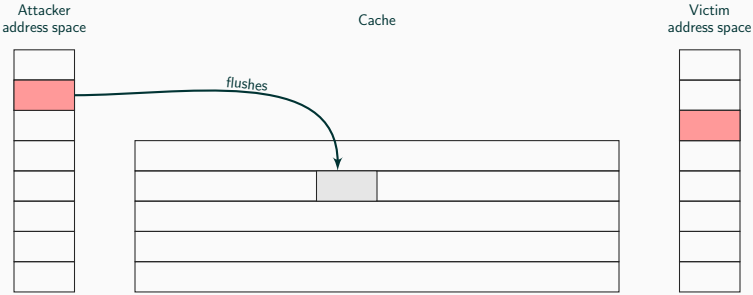
2-way set associativity

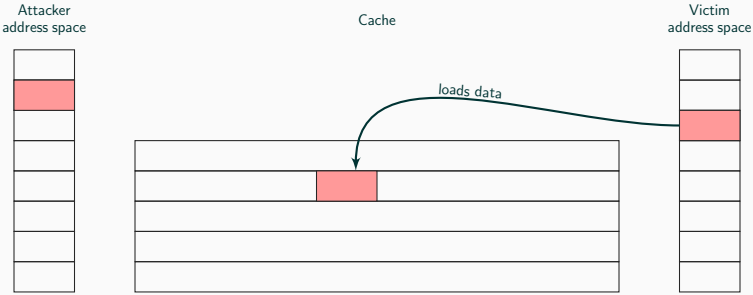


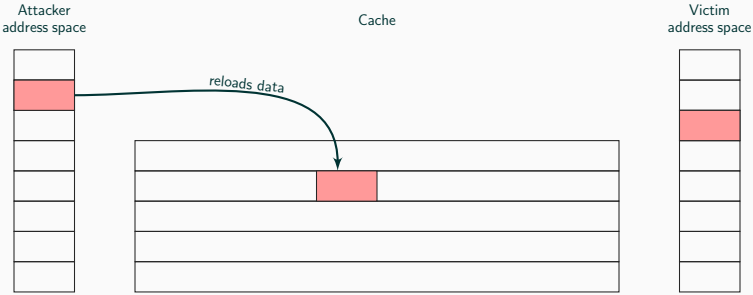
→ replacement policy


Attacker
address space
Cache
Victim
address space


Attacker
address space
Cache
Victim
address space
cached
cached


Attacker
address space
Cache
Victim
address space
flushes


Attacker
address space
Cache
Victim
address space
loads data


Attacker
address space
Cache
Victim
address space
reloads data

Pros: fine granularity (1 line)

Pros: fine granularity (1 line)

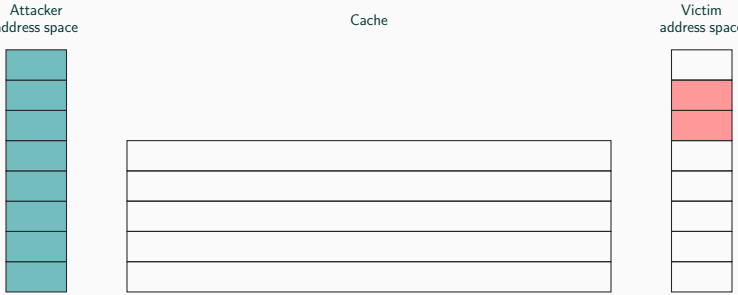
Cons: restrictive

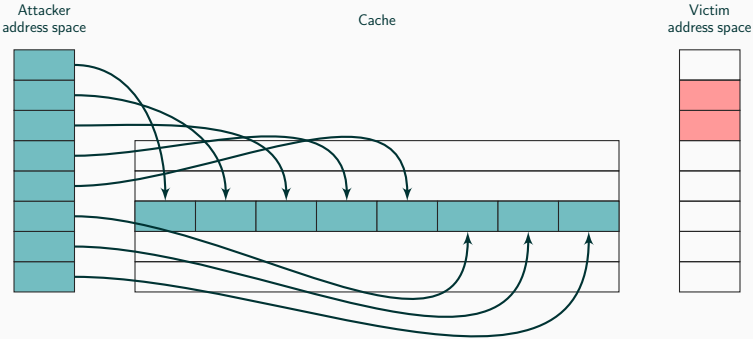
1. needs `clflush` instruction (not available e.g., in JS)

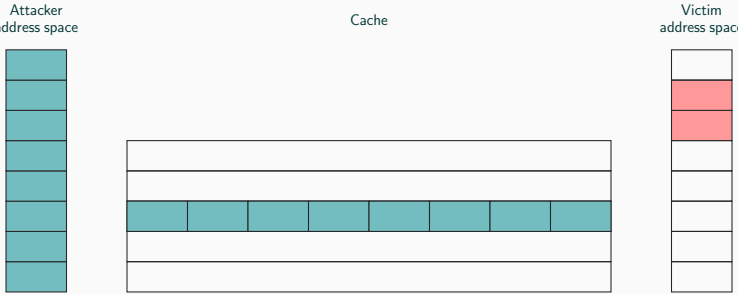
Pros: fine granularity (1 line)

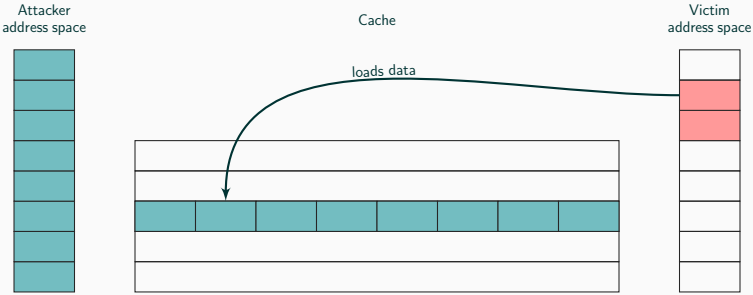
Cons: restrictive

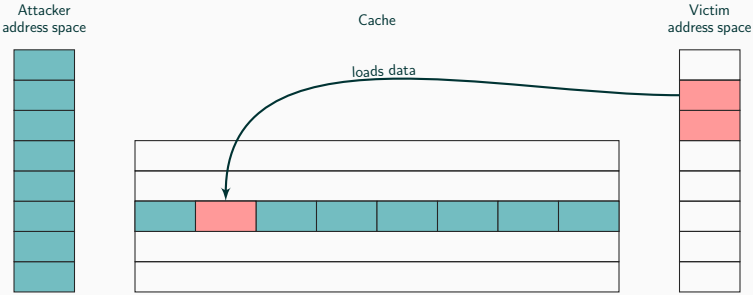
1. needs `clflush` instruction (not available e.g., in JS)
2. needs shared memory

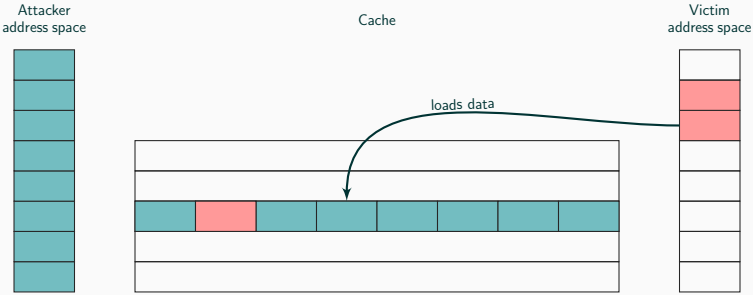

Attacker
address space
Cache
Victim
address space

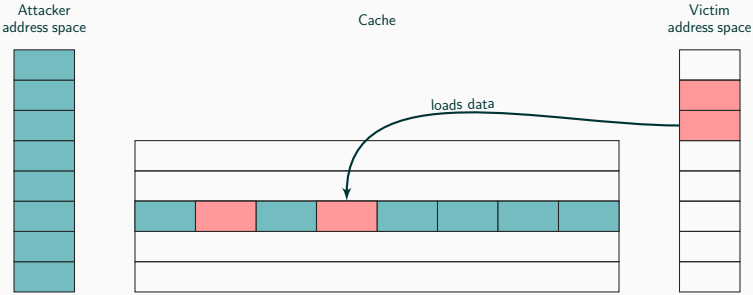

Attacker
address space
Cache
Victim
address space

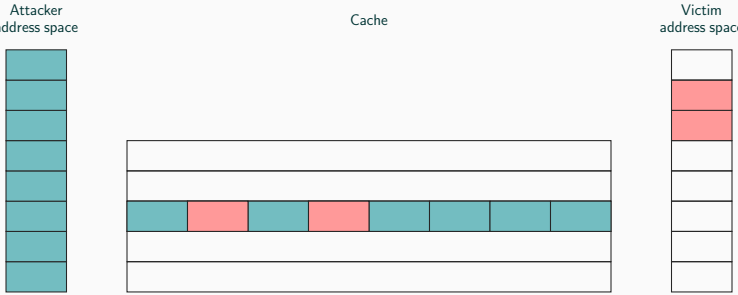

Attacker
address space
Cache
Victim
address space

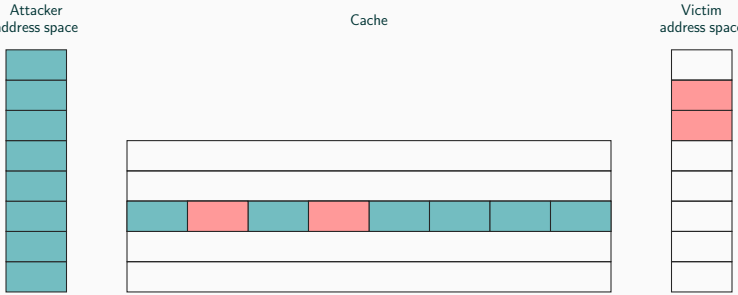

Attacker
address space
Cache
Victim
address space
loads data

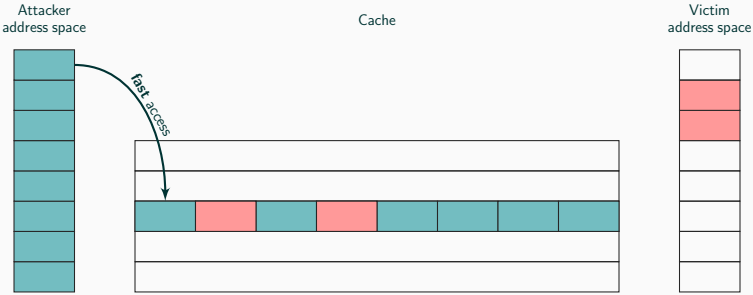

Attacker
address space
Cache
Victim
address space
loads data

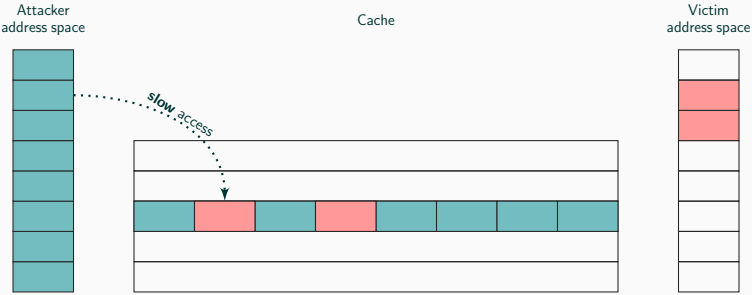

Attacker
address space
Cache
Victim
address space
loads data


Attacker
address space
Cache
Victim
address space
loads data


Attacker
address space
Cache
Victim
address space


Attacker
address space
Cache
Victim
address space


Attacker
address space
Cache
Victim
address space
fast access


Attacker
address space
Cache
Victim
address space
slow access

Pros: less restrictive

1. no need for `clflush` instruction (not available e.g., in JS)

Pros: less restrictive

1. no need for `clflush` instruction (not available e.g., in JS)
2. no need for shared memory

Pros: less restrictive

1. no need for `clflush` instruction (not available e.g., in JS)
2. no need for shared memory

Pros: less restrictive

1. no need for `clflush` instruction (not available e.g., in JS)
2. no need for shared memory

Cons: coarser granularity (1 set)

- Paging: memory translated page-wise from virtual to physical

TLB and Paging

- Paging: memory translated page-wise from virtual to physical
- TLB (translation lookaside buffer) caches virtual to physical mapping

TLB and Paging

- Paging: memory translated page-wise from virtual to physical
- TLB (translation lookaside buffer) caches virtual to physical mapping
- TLB has some latency

- Paging: memory translated page-wise from virtual to physical
- TLB (translation lookaside buffer) caches virtual to physical mapping
- TLB has some latency
- Worst case for Cache: mapping not in TLB, need to load mapping from RAM

- Paging: memory translated page-wise from virtual to physical
- TLB (translation lookaside buffer) caches virtual to physical mapping
- TLB has some latency
- Worst case for Cache: mapping not in TLB, need to load mapping from RAM
- Solution: Use virtual addresses instead of physical addresses

Cache indexing methods

- VIVT: Virtually indexed, virtually tagged

Cache indexing methods

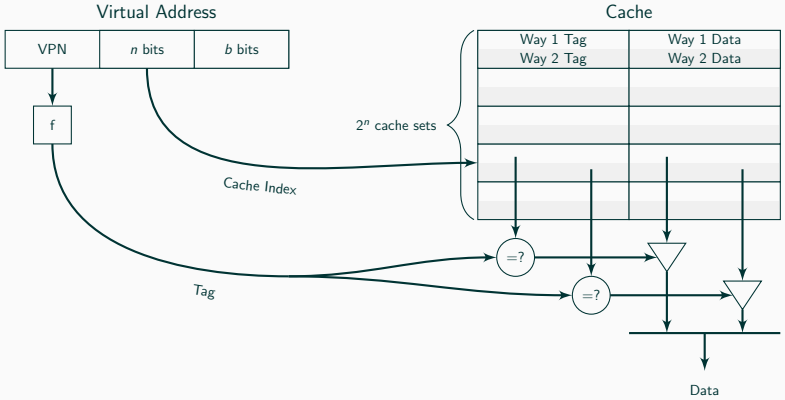
- VIVT: Virtually indexed, virtually tagged
- PIPT: Physically indexed, physically tagged

Cache indexing methods

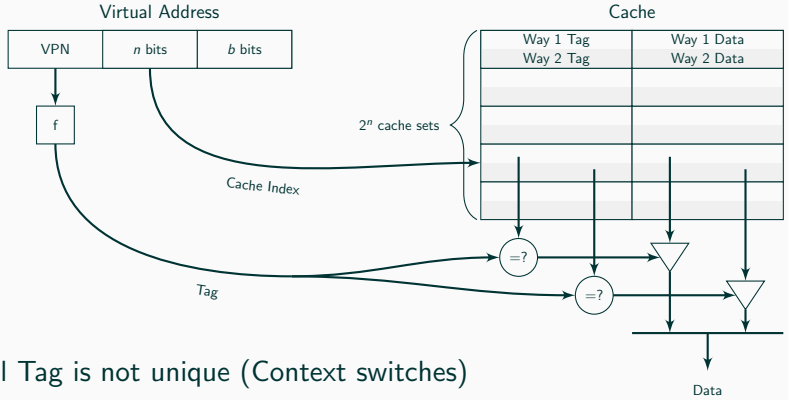
- VIVT: Virtually indexed, virtually tagged
- PIPT: Physically indexed, physically tagged
- PIVT: Physically indexed, virtually tagged

Cache indexing methods

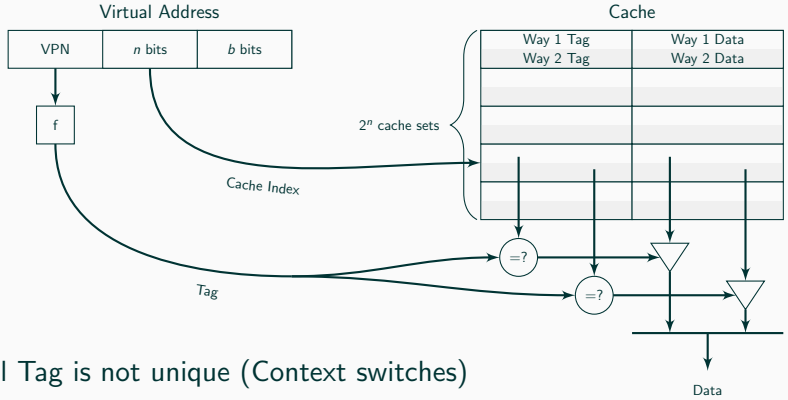
- VIVT: Virtually indexed, virtually tagged
- PIPT: Physically indexed, physically tagged
- PIVT: Physically indexed, virtually tagged
- VIPT: Virtually indexed, physically tagged


Virtual Address
VPN
n bits
b bits
f
Cache Index
Tag
2^n cache sets
Cache
Way 1 Tag
Way 2 Tag
Way 1 Data
Way 2 Data
=?
=?
Data

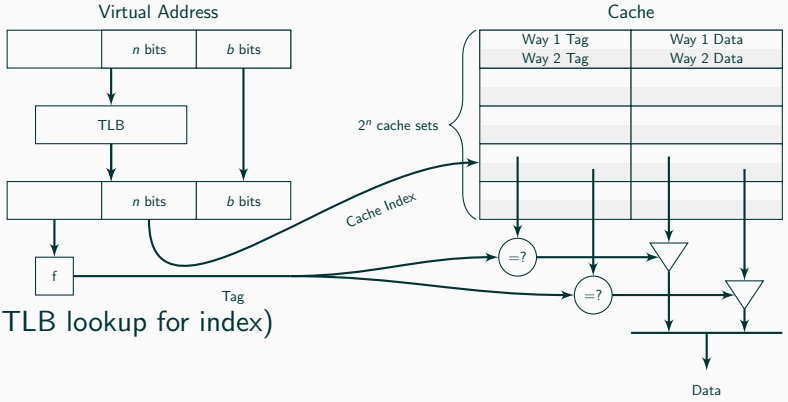
- Fast



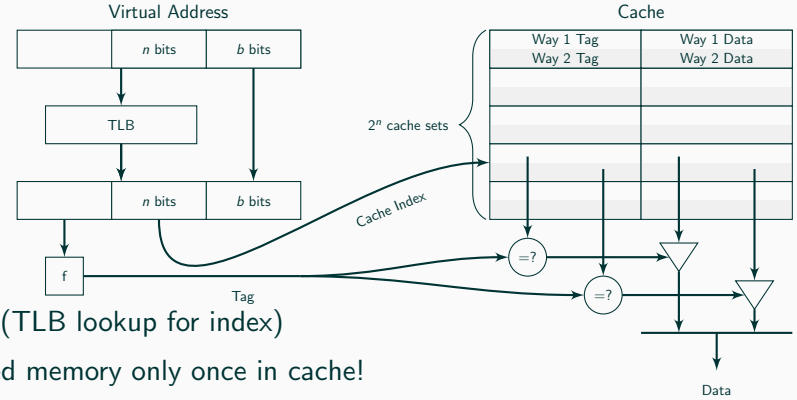
- Fast
- Virtual Tag is not unique (Context switches)



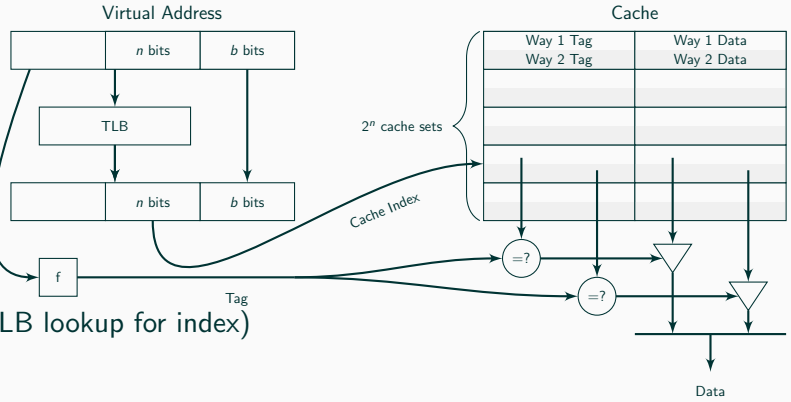
- Fast
- Virtual Tag is not unique (Context switches)
- Shared memory more than once in cache



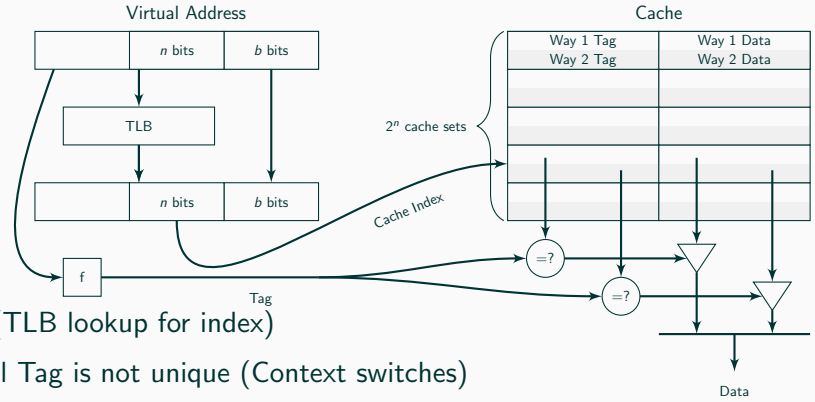
- Slow (TLB lookup for index)



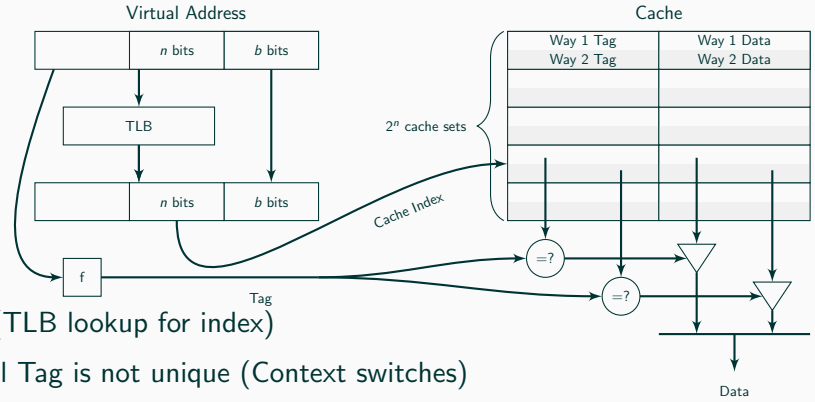
- Slow (TLB lookup for index)
- Shared memory only once in cache!



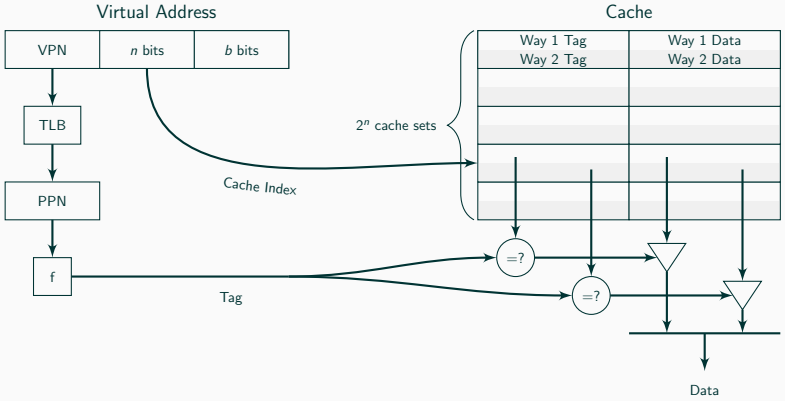
- Slow (TLB lookup for index)



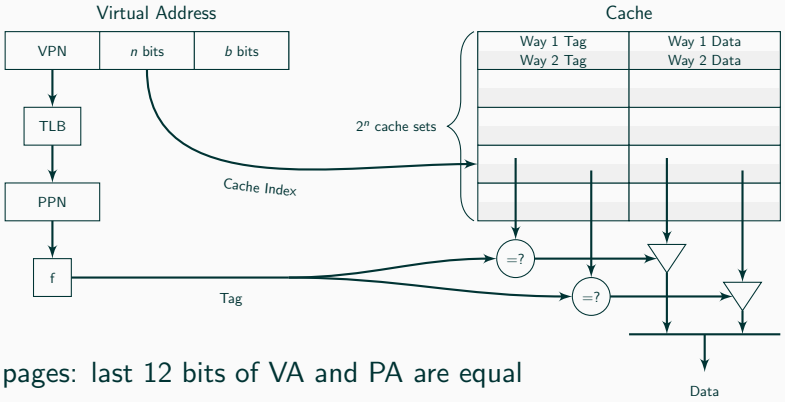
- Slow (TLB lookup for index)
- Virtual Tag is not unique (Context switches)



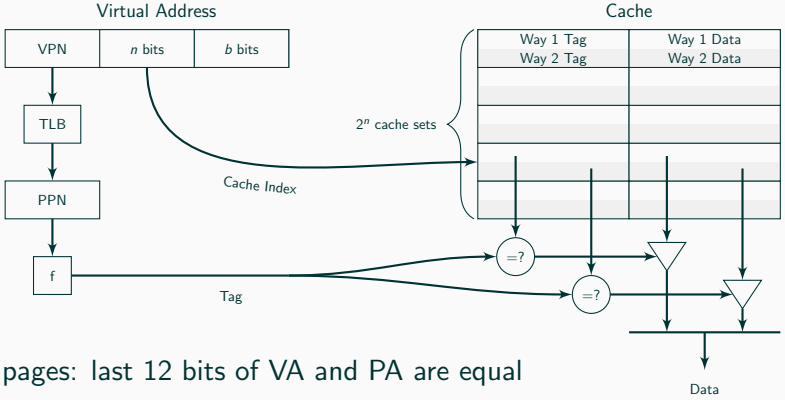
- Slow (TLB lookup for index)
- Virtual Tag is not unique (Context switches)
- Shared memory more than once in cache


Virtual Address
VPN
n bits
b bits
TLB
PPN
f
Tag
Cache Index
Cache
2^n cache sets
Way 1 Tag
Way 2 Tag
Way 1 Data
Way 2 Data
=?
=?
Data

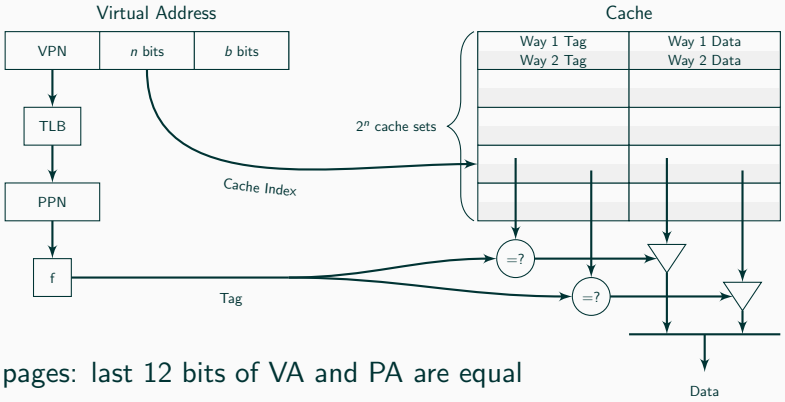
- Fast



- Fast
- 4 KiB pages: last 12 bits of VA and PA are equal



- Fast
- 4 KiB pages: last 12 bits of VA and PA are equal
- Using more bits is unpractical (like VIVT)



- Fast
- 4 KiB pages: last 12 bits of VA and PA are equal
- Using more bits is unpractical (like VIVT)

$\rightarrow$ Cache size $\leq$ # ways $\cdot$ page size

- L1 caches: VIVT or VIPT

- L1 caches: VIVT or VIPT
- L2/L3 caches: PIPT

We need to evict cache lines without `clflush` or shared memory:

1. which addresses do we access to have congruent cache lines?

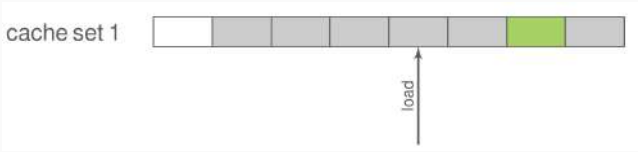
Issues with Prime+Probe

We need to evict cache lines without `clflush` or shared memory:

1. which addresses do we access to have congruent cache lines?
2. without any privilege?

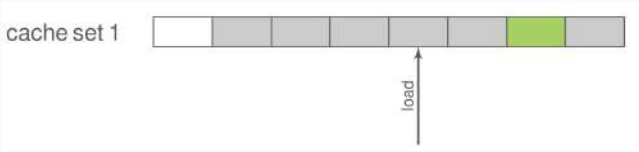
We need to evict cache lines without `clflush` or shared memory:

1. which addresses do we access to have congruent cache lines?
2. without any privilege?
3. and in which order do we access them?



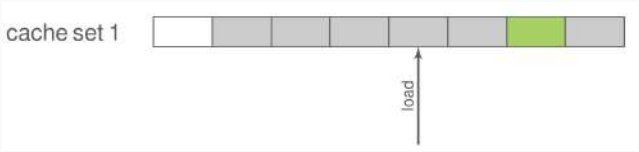
"LRU eviction":

- assume that cache uses LRU replacement



"LRU eviction":

- assume that cache uses LRU replacement
- accessing n **addresses from the same cache set** to evict an n-way set

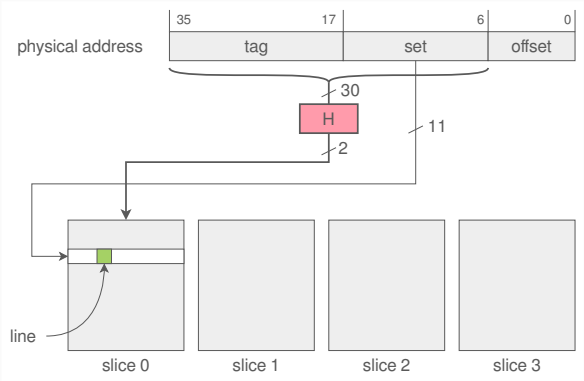


"LRU eviction":

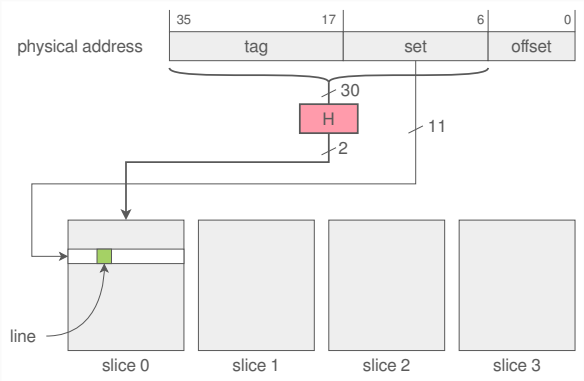
- assume that cache uses LRU replacement
- accessing n **addresses from the same cache set** to evict an n-way set
- eviction from last level $\rightarrow$ from whole hierarchy (it's inclusive!)

#1.2: Which addresses map to the same set?

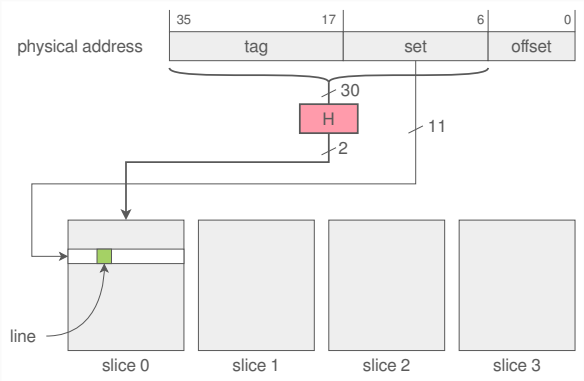
#1.2: Which addresses map to the same set?



#1.2: Which addresses map to the same set?

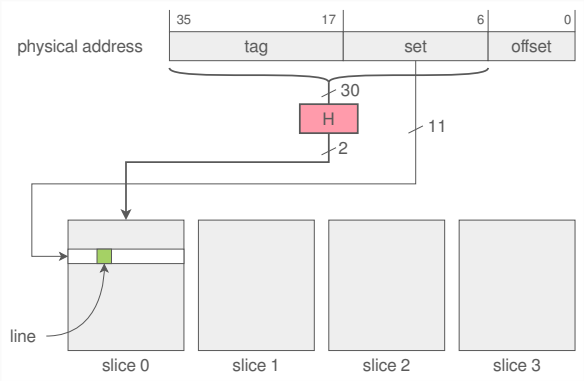


#1.2: Which addresses map to the same set?



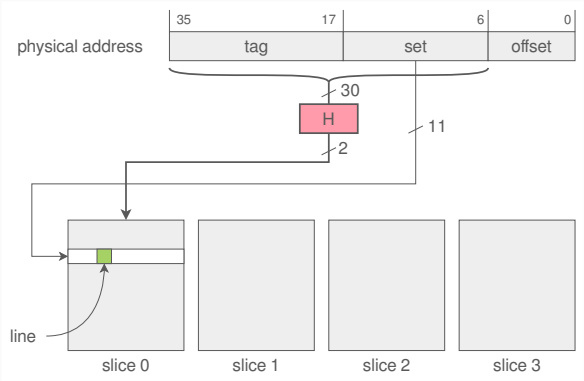
- function H that maps slices is undocumented

#1.2: Which addresses map to the same set?



- function H that maps slices is undocumented
- **reverse-engineered** by Maurice et al

#1.2: Which addresses map to the same set?



- function H that maps slices is undocumented
- **reverse-engineered** by Maurice et al
- hash function basically an XOR of address bits

#1.2: Which addresses map to the same set?

3 functions, depending on the number of cores

		Address bit																													
		37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06
2 cores	o_0						⊕		⊕		⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕	⊕	⊕		⊕		⊕		⊕				⊕
4 cores	o_0					⊕	⊕		⊕		⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕	⊕	⊕		⊕		⊕		⊕				⊕
	o_1				⊕	⊕		⊕		⊕	⊕		⊕		⊕	⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕		⊕			⊕		
8 cores	o_0		⊕	⊕		⊕	⊕		⊕		⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕	⊕	⊕		⊕		⊕		⊕				⊕
	o_1	⊕		⊕	⊕	⊕		⊕		⊕	⊕		⊕		⊕	⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕		⊕			⊕		
	o_2	⊕	⊕	⊕	⊕			⊕	⊕			⊕	⊕			⊕	⊕			⊕			⊕			⊕	⊕			⊕			

#2: Obtain information without root privileges

- last-level cache is physically indexed

#2: Obtain information without root privileges

- last-level cache is physically indexed
- root privileges needed for physical addresses

#2: Obtain information without root privileges

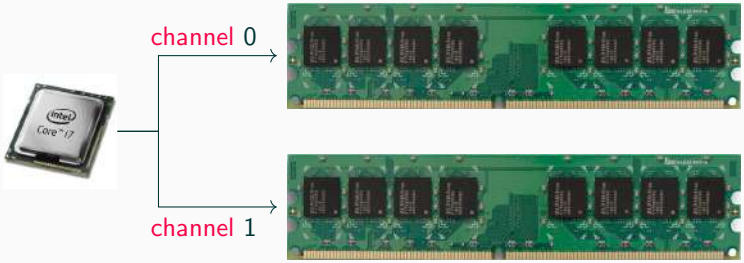
- last-level cache is physically indexed
- root privileges needed for physical addresses
- use 2 MB pages $\rightarrow$ lowest 21 bits are the same as virtual address

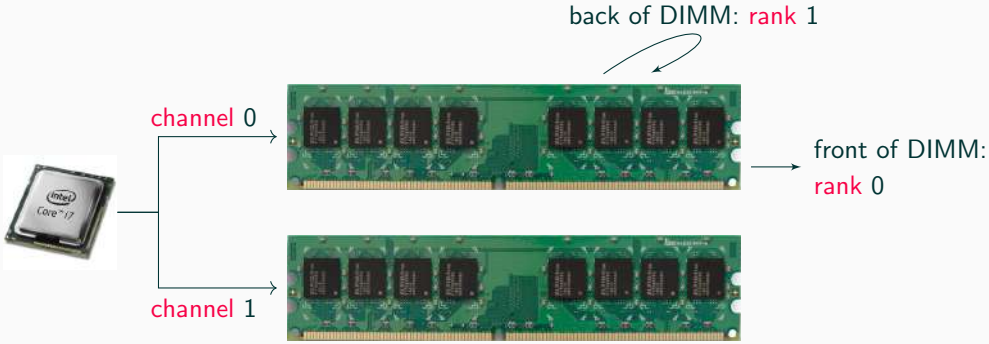
#2: Obtain information without root privileges

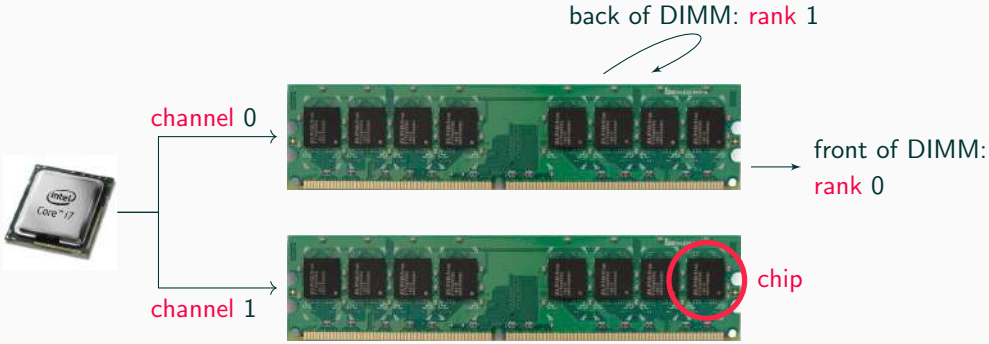
- last-level cache is physically indexed
- root privileges needed for physical addresses
- use 2 MB pages $\rightarrow$ lowest 21 bits are the same as virtual address

$\rightarrow$ enough to compute the cache set

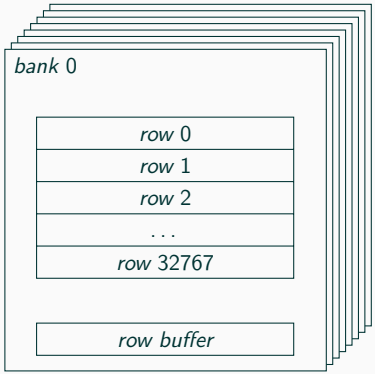
DRAM organization



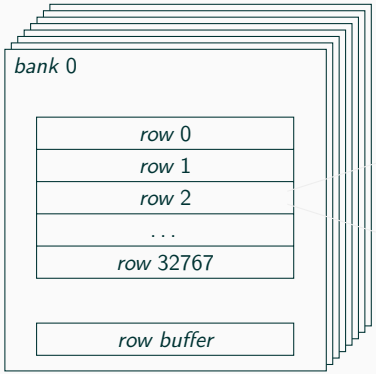

back of DIMM: rank 1
channel 0
front of DIMM:
rank 0
channel 1


back of DIMM: rank 1
channel 0
front of DIMM:
rank 0
chip
channel 1

DRAM organization

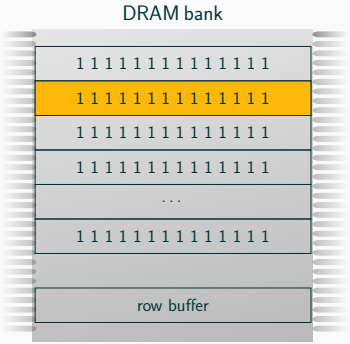


DRAM organization



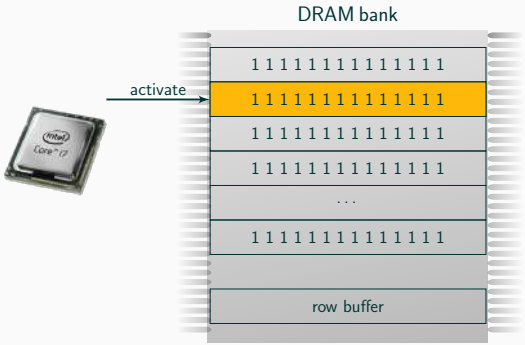
64k cells
1 capacitor,
1 transitor each

How reading from DRAM works



CPU wants to access row 1

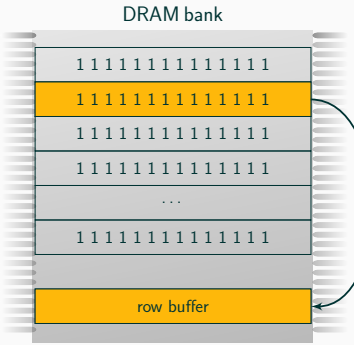
How reading from DRAM works



CPU wants to access row 1

$\rightarrow$ row 1 activated

How reading from DRAM works

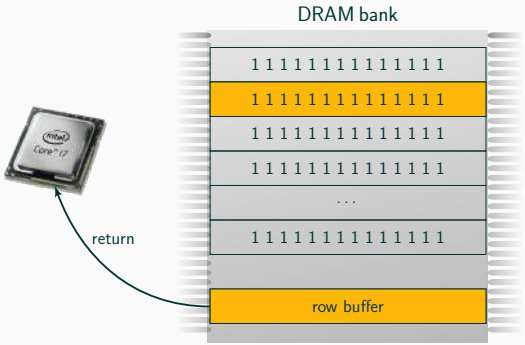


CPU wants to access row 1

$\rightarrow$ row 1 activated

$\rightarrow$ row 1 copied to row buffer

How reading from DRAM works

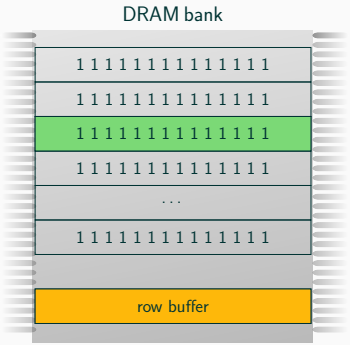


CPU wants to access row 1

$\rightarrow$ row 1 activated

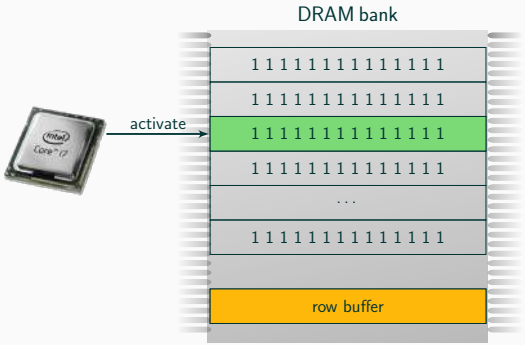
$\rightarrow$ row 1 copied to row buffer

How reading from DRAM works



CPU wants to access row 2

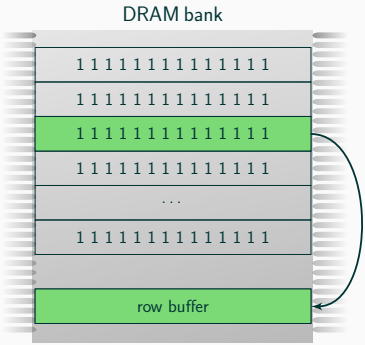
How reading from DRAM works



CPU wants to access row 2

$\rightarrow$ row 2 activated

How reading from DRAM works

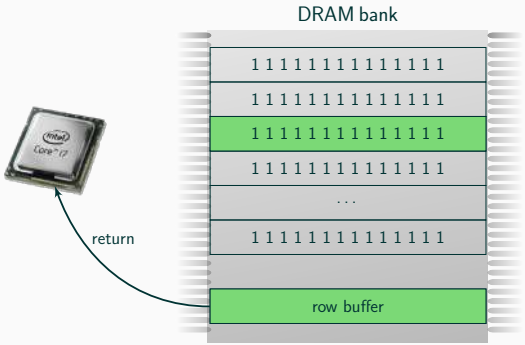


CPU wants to access row 2

$\rightarrow$ row 2 activated

$\rightarrow$ row 2 copied to row buffer

How reading from DRAM works

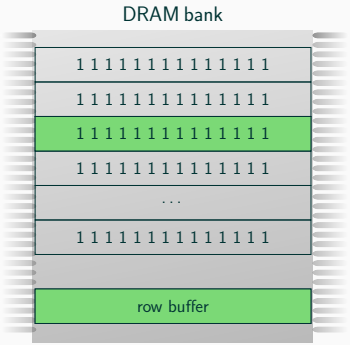


CPU wants to access row 2

$\rightarrow$ row 2 activated

$\rightarrow$ row 2 copied to row buffer

How reading from DRAM works



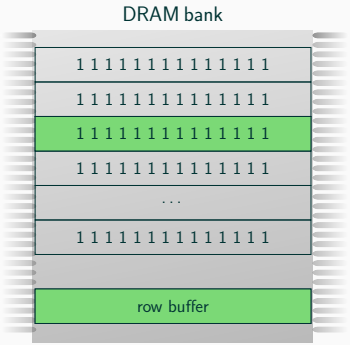
CPU wants to access row 2

→ row 2 activated

→ row 2 copied to row buffer

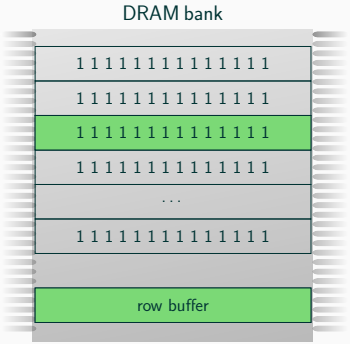
→ slow (row conflict)

How reading from DRAM works



CPU wants to access row 2—again

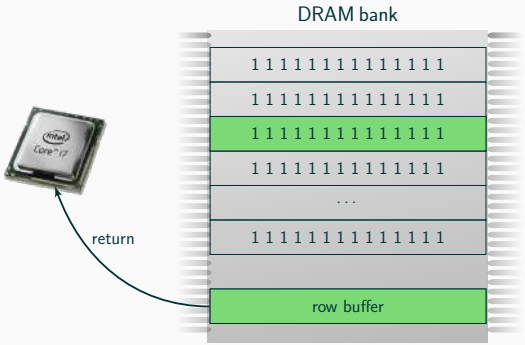
How reading from DRAM works



CPU wants to access row 2—again

→ row 2 already in row buffer

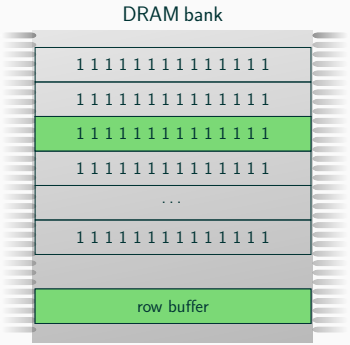
How reading from DRAM works



CPU wants to access row 2—again

→ row 2 already in row buffer

How reading from DRAM works

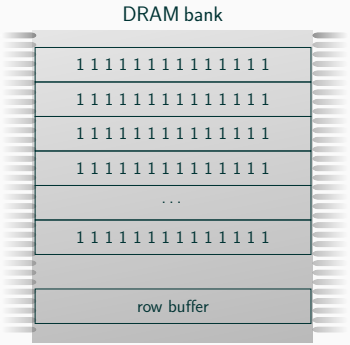


CPU wants to access row 2—again

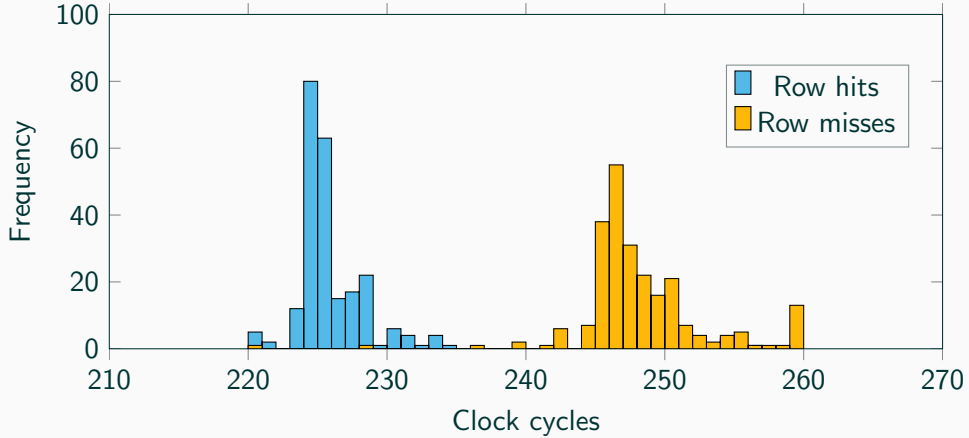
→ row 2 already in row buffer

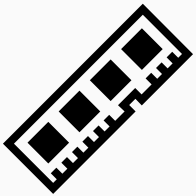
→ fast (row hit)

How reading from DRAM works

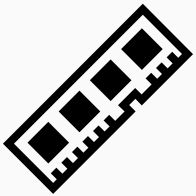


row buffer = cache

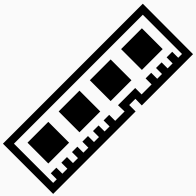

100
80
60
40
20
0
Frequency
210
220
230
240
250
260
270
Clock cycles
Row hits
Row misses



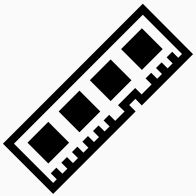
- Cache set is determined by part of physical address



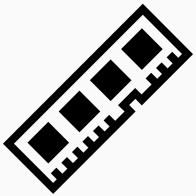
- Cache set is determined by part of physical address
- We have no knowledge of physical addresses



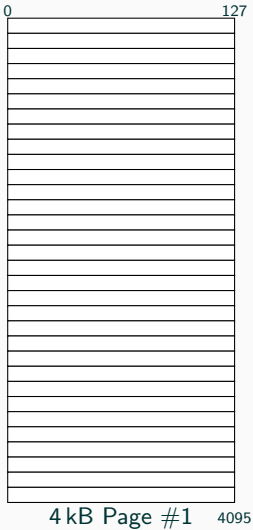
- Cache set is determined by part of physical address
- We have no knowledge of physical addresses
- Use the reverse-engineered DRAM mapping



- Cache set is determined by part of physical address
- We have no knowledge of physical addresses
- Use the reverse-engineered DRAM mapping
- Exploit timing differences to find DRAM row borders



- Cache set is determined by part of physical address
- We have no knowledge of physical addresses
- Use the reverse-engineered DRAM mapping
- Exploit timing differences to find DRAM row borders
- The 18 LSBs are '0' at a row border

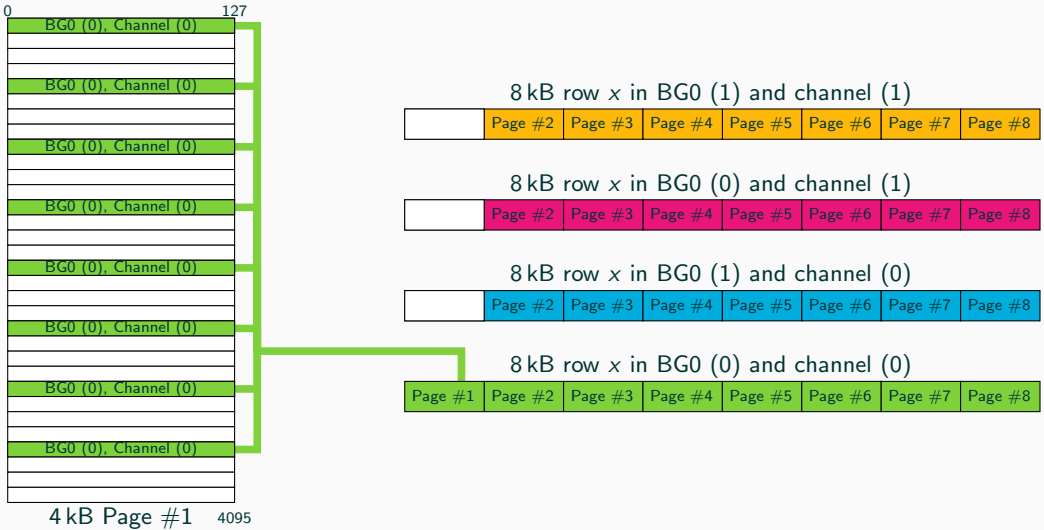


8 kB row x in BG0 (1) and channel (1)

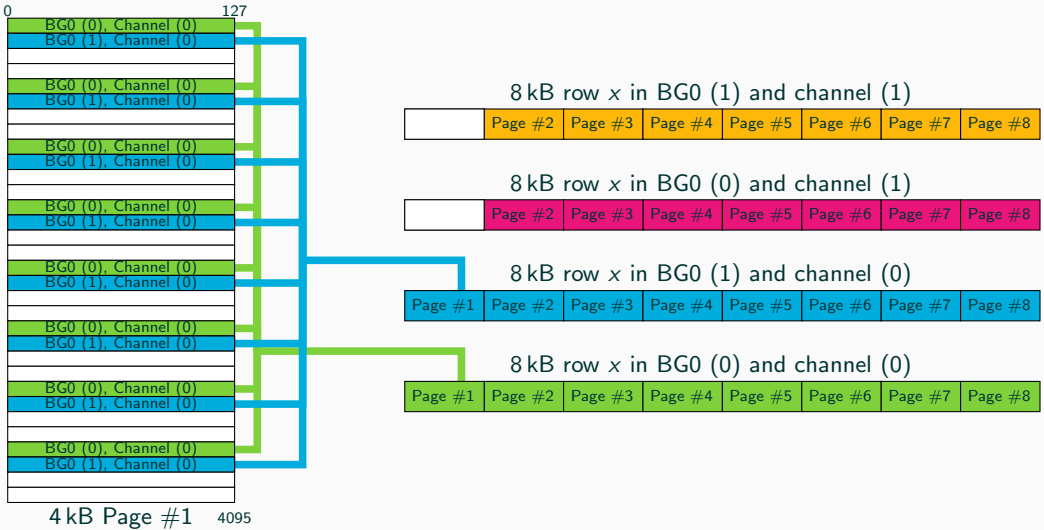
8 kB row x in BG0 (0) and channel (1)

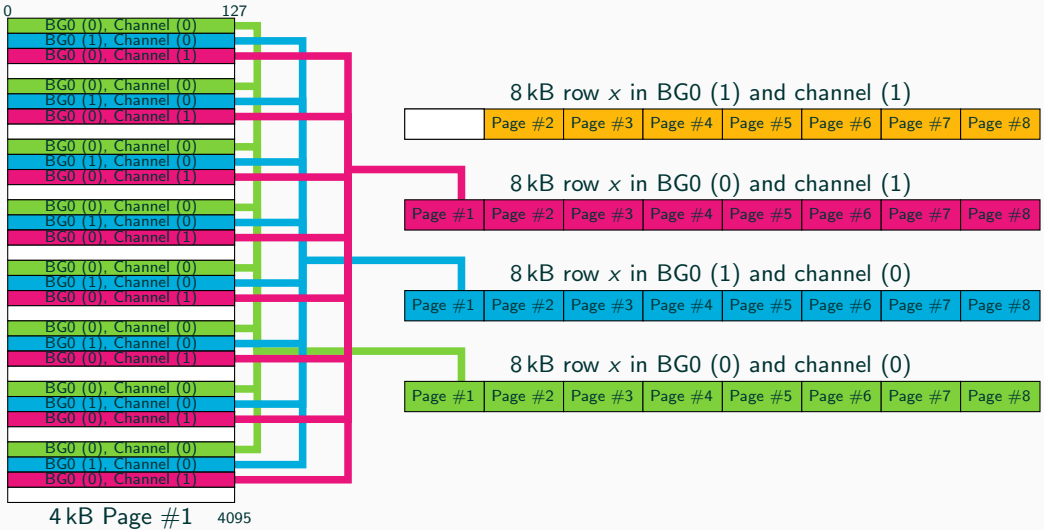
8 kB row x in BG0 (1) and channel (0)

8 kB row x in BG0 (0) and channel (0)

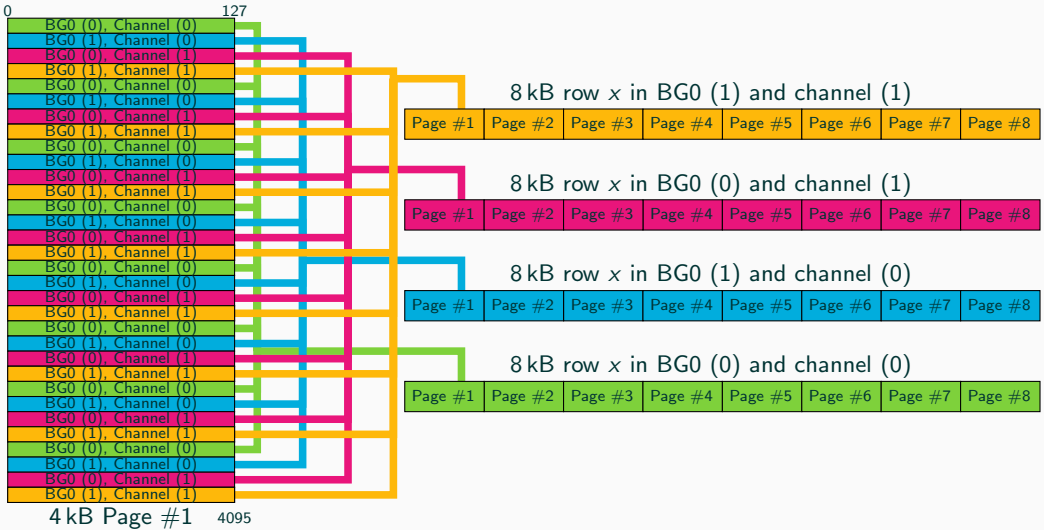

0
127
BG0 (0), Channel (0)
BG0 (0), Channel (0)
BG0 (0), Channel (0)
BG0 (0), Channel (0)
BG0 (0), Channel (0)
BG0 (0), Channel (0)
BG0 (0), Channel (0)
BG0 (0), Channel (0)
4 kB Page #1
4095
8 kB row x in BG0 (1) and channel (1)
Page #2
Page #3
Page #4
Page #5
Page #6
Page #7
Page #8
8 kB row x in BG0 (0) and channel (1)
Page #2
Page #3
Page #4
Page #5
Page #6
Page #7
Page #8
8 kB row x in BG0 (1) and channel (0)
Page #2
Page #3
Page #4
Page #5
Page #6
Page #7
Page #8
8 kB row x in BG0 (0) and channel (0)
Page #1
Page #2
Page #3
Page #4
Page #5
Page #6
Page #7
Page #8

Physical Addresses




0
127
BG0 (0), Channel (0)
BG0 (1), Channel (0)
BG0 (0), Channel (1)
4 kB Page #1
4095
8 kB row x in BG0 (1) and channel (1)
8 kB row x in BG0 (0) and channel (1)
8 kB row x in BG0 (1) and channel (0)
8 kB row x in BG0 (0) and channel (0)
Page #1
Page #2
Page #3
Page #4
Page #5
Page #6
Page #7
Page #8

Physical Addresses



0

127

BG0 (0), Channel (0)
BG0 (1), Channel (0)
BG0 (0), Channel (1)
BG0 (1), Channel (1)
BG0 (0), Channel (0)
BG0 (1), Channel (0)
BG0 (0), Channel (1)
BG0 (1), Channel (1)
BG0 (0), Channel (0)
BG0 (1), Channel (0)
BG0 (0), Channel (1)
BG0 (1), Channel (1)
BG0 (0), Channel (0)
BG0 (1), Channel (0)
BG0 (0), Channel (1)
BG0 (1), Channel (1)
BG0 (0), Channel (0)
BG0 (1), Channel (0)
BG0 (0), Channel (1)
BG0 (1), Channel (1)
BG0 (0), Channel (0)
BG0 (1), Channel (0)
BG0 (0), Channel (1)
BG0 (1), Channel (1)
BG0 (0), Channel (0)
BG0 (1), Channel (0)
BG0 (0), Channel (1)
BG0 (1), Channel (1)
BG0 (0), Channel (0)
BG0 (1), Channel (0)
BG0 (0), Channel (1)
BG0 (1), Channel (1)

4 kB Page #1

4095

8 kB row x in BG0 (1) and channel (1)

Page #1	Page #2	Page #3	Page #4	Page #5	Page #6	Page #7	Page #8

8 kB row x in BG0 (0) and channel (1)

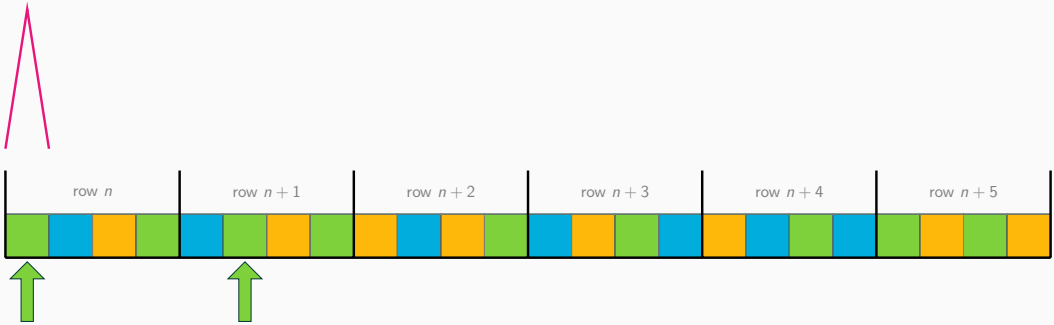
Page #1	Page #2	Page #3	Page #4	Page #5	Page #6	Page #7	Page #8

8 kB row x in BG0 (1) and channel (0)

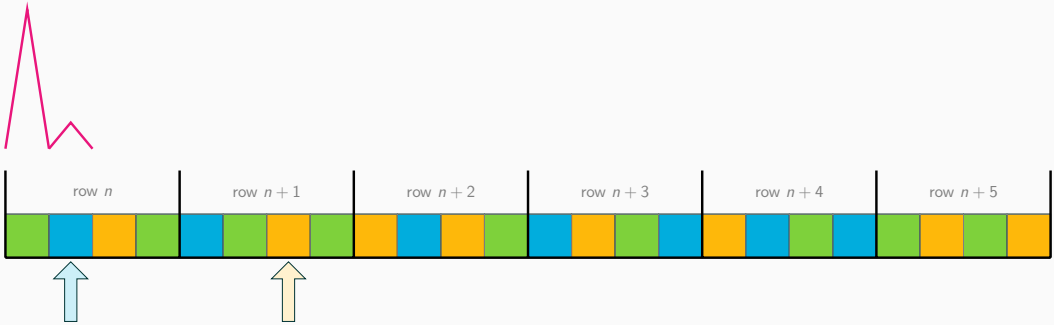
Page #1	Page #2	Page #3	Page #4	Page #5	Page #6	Page #7	Page #8

8 kB row x in BG0 (0) and channel (0)

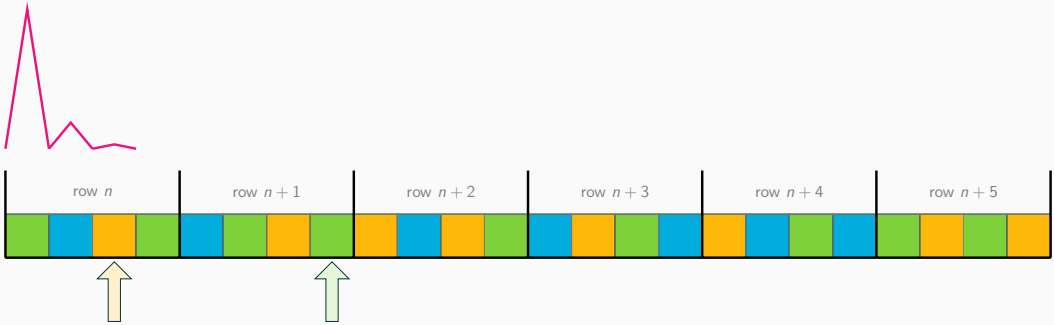
Page #1	Page #2	Page #3	Page #4	Page #5	Page #6	Page #7	Page #8

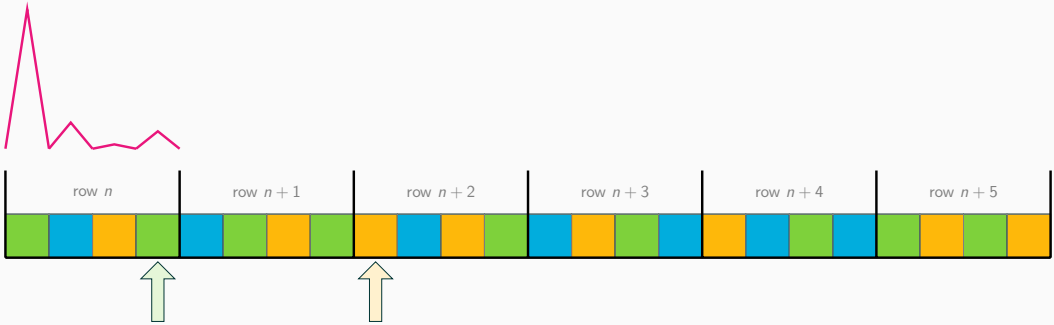

row n
row $n+1$
row $n+2$
row $n+3$
row $n+4$
row $n+5$

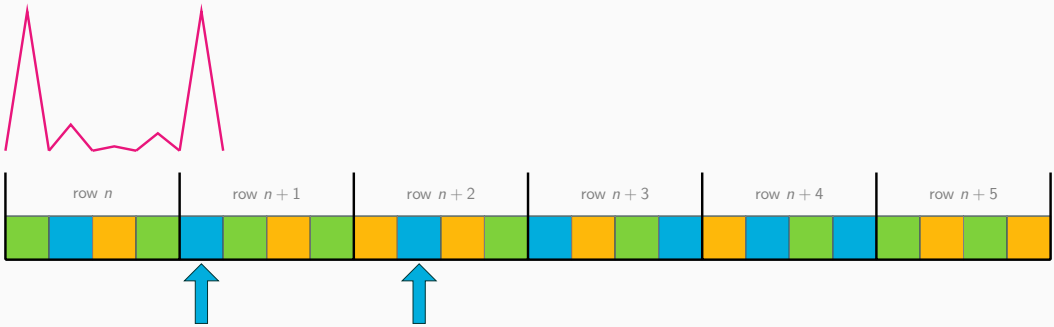
Physical Addresses



Physical Addresses

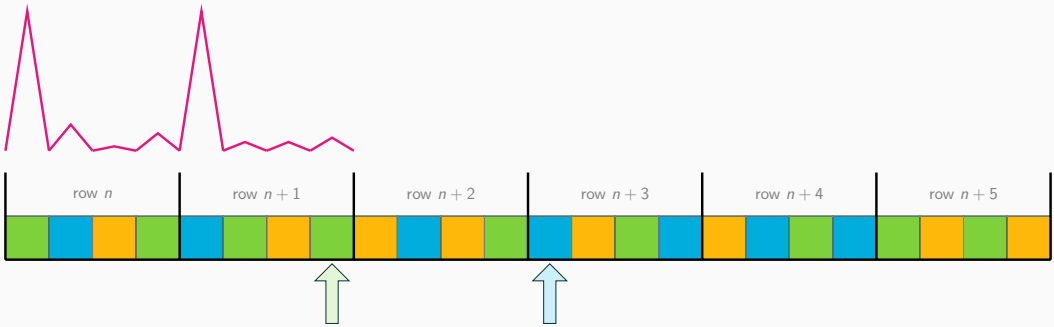



row n
row $n+1$
row $n+2$
row $n+3$
row $n+4$
row $n+5$

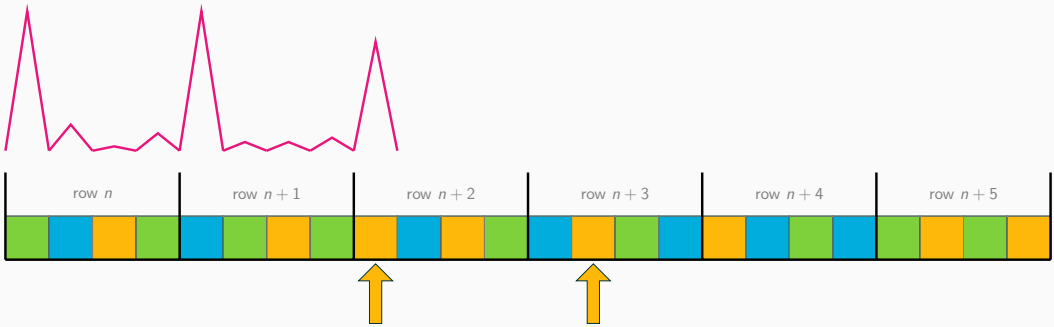

row n
row $n+1$
row $n+2$
row $n+3$
row $n+4$
row $n+5$


row n
row $n+1$
row $n+2$
row $n+3$
row $n+4$
row $n+5$

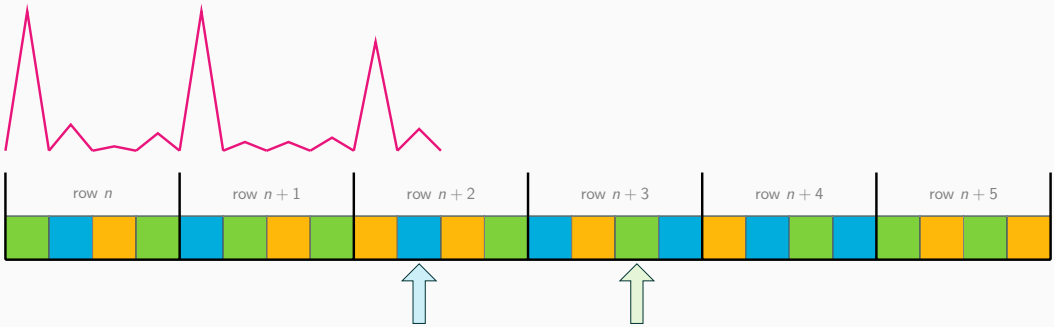

row n
row $n+1$
row $n+2$
row $n+3$
row $n+4$
row $n+5$

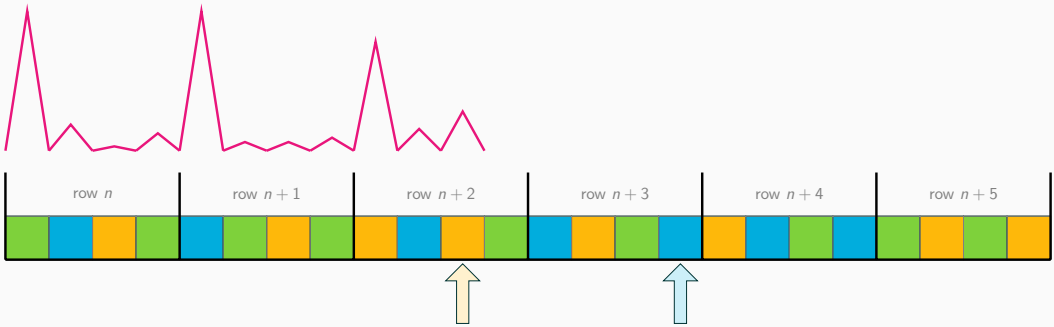

row n
row $n+1$
row $n+2$
row $n+3$
row $n+4$
row $n+5$

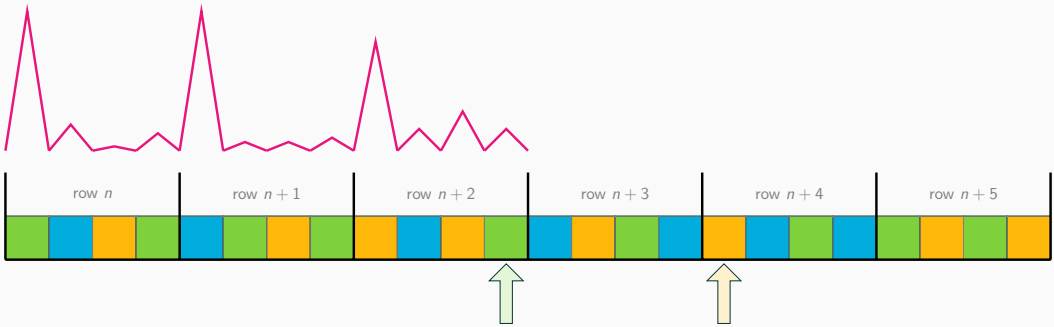
Physical Addresses

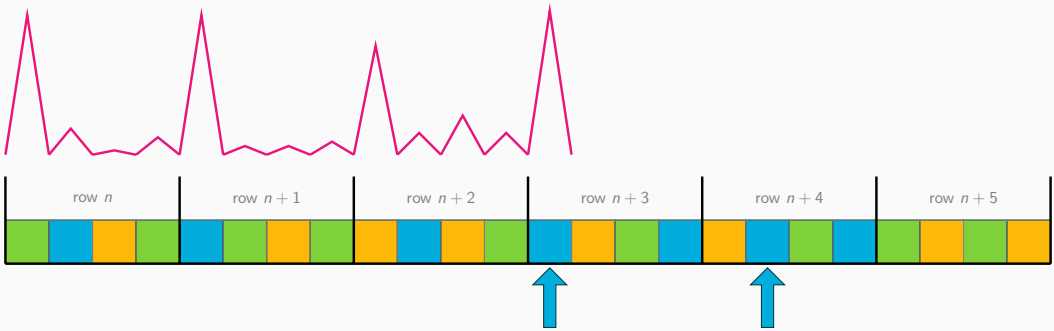


Physical Addresses

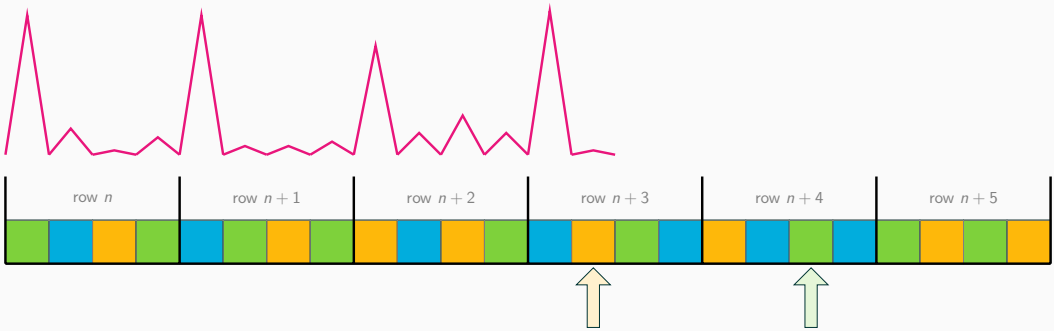


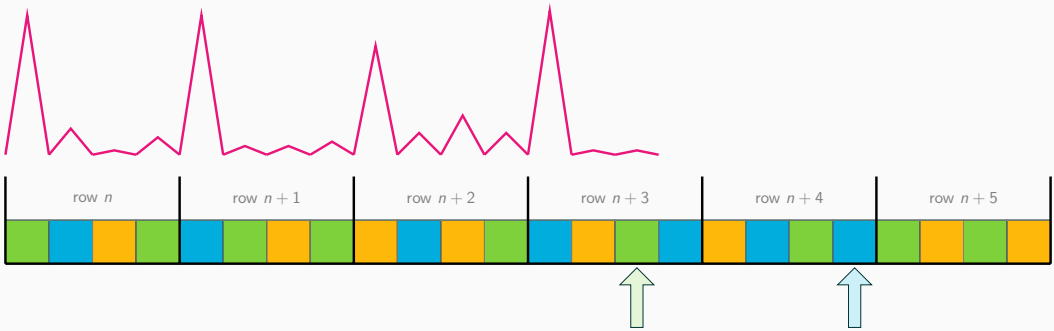

row n
row $n+1$
row $n+2$
row $n+3$
row $n+4$
row $n+5$


row n
row $n+1$
row $n+2$
row $n+3$
row $n+4$
row $n+5$

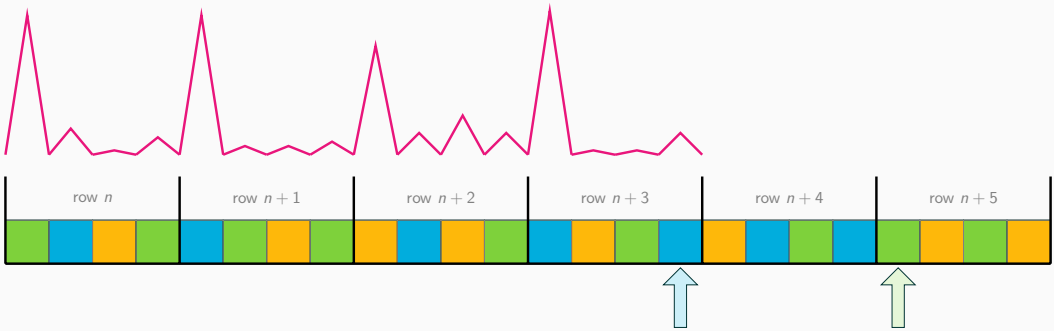

row n
row $n+1$
row $n+2$
row $n+3$
row $n+4$
row $n+5$

Physical Addresses




row n
row $n+1$
row $n+2$
row $n+3$
row $n+4$
row $n+5$

Physical Addresses

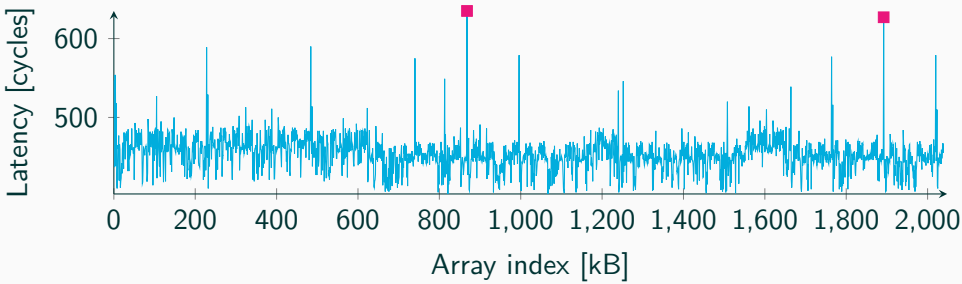



row n
row $n+1$
row $n+2$
row $n+3$
row $n+4$
row $n+5$

Physical Addresses

Physical Addresses

Result on an Intel i5-6200U



"LRU eviction" memory accesses

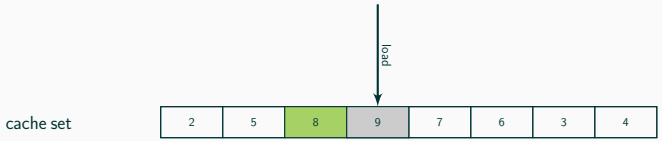
"LRU eviction" memory accesses

- LRU replacement policy: oldest entry first

"LRU eviction" memory accesses

- LRU replacement policy: oldest entry first
- timestamps for every cache line

"LRU eviction" memory accesses



- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

"LRU eviction" memory accesses

- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

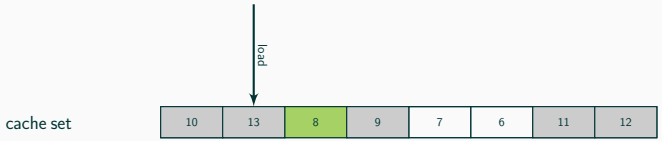
"LRU eviction" memory accesses

- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

"LRU eviction" memory accesses

- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

"LRU eviction" memory accesses



- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

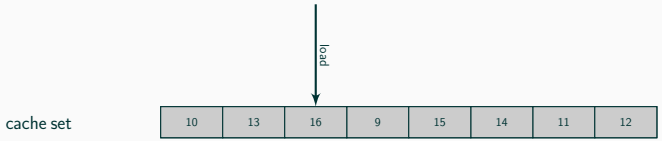
"LRU eviction" memory accesses

- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

"LRU eviction" memory accesses

- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

"LRU eviction" memory accesses

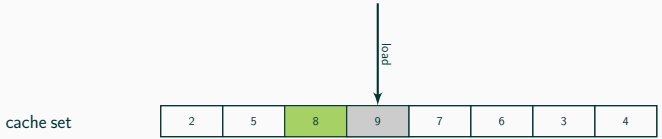


- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

"LRU eviction" memory accesses

- no LRU replacement

"LRU eviction" memory accesses

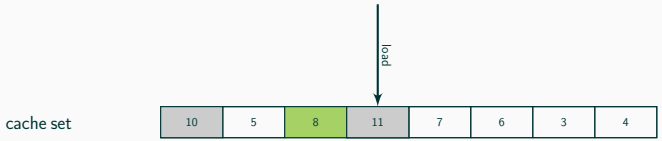


- no LRU replacement

"LRU eviction" memory accesses

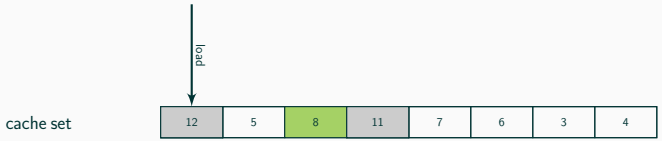
- no LRU replacement

"LRU eviction" memory accesses



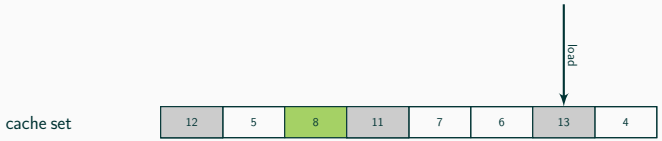
- no LRU replacement

"LRU eviction" memory accesses



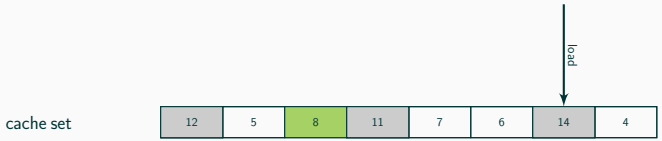
- no LRU replacement

"LRU eviction" memory accesses



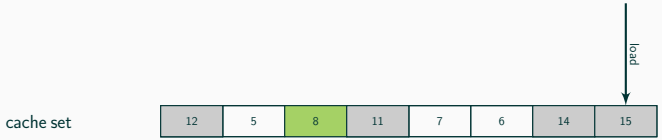
- no LRU replacement

"LRU eviction" memory accesses



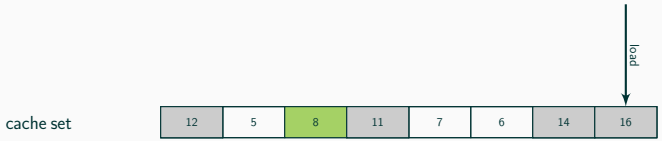
- no LRU replacement

"LRU eviction" memory accesses



- no LRU replacement

"LRU eviction" memory accesses



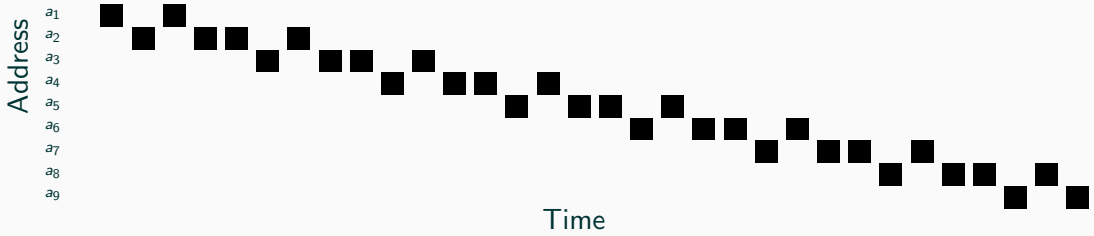
- no LRU replacement

"LRU eviction" memory accesses

- no LRU replacement
- only 75% success rate on Haswell

"LRU eviction" memory accesses

- no LRU replacement
- only 75% success rate on Haswell
- more accesses → higher success rate, but **too slow**



$\rightarrow$ fast and effective on Haswell: eviction rate >99.97%

Cache eviction strategy: New representation

- represent accesses as a sequence of numbers: $1, 2, 1, 2, 2, 3, 2, 3, 3, 4, 3, 4, \ldots$

Cache eviction strategy: New representation

- represent accesses as a sequence of numbers: $1, 2, 1, 2, 2, 3, 2, 3, 3, 4, 3, 4, \ldots$
- can be a long sequence

Cache eviction strategy: New representation

- represent accesses as a sequence of numbers: $1, 2, 1, 2, 2, 3, 2, 3, 3, 4, 3, 4, \ldots$
- can be a long sequence
- all congruent addresses are indistinguishable w.r.t eviction strategy

Cache eviction strategy: New representation

- represent accesses as a sequence of numbers: $1, 2, 1, 2, 2, 3, 2, 3, 3, 4, 3, 4, \ldots$
- can be a long sequence
- all congruent addresses are indistinguishable w.r.t eviction strategy

$\rightarrow$ adding more **unique** addresses can increase eviction rate

Cache eviction strategy: New representation

- represent accesses as a sequence of numbers: $1, 2, 1, 2, 2, 3, 2, 3, 3, 4, 3, 4, \ldots$
- can be a long sequence
- all congruent addresses are indistinguishable w.r.t eviction strategy

$\rightarrow$ adding more **unique** addresses can increase eviction rate

$\rightarrow$ **multiple** accesses to one address can increase the eviction rate

Cache eviction strategy: New representation

- represent accesses as a sequence of numbers: $1, 2, 1, 2, 2, 3, 2, 3, 3, 4, 3, 4, \ldots$
- can be a long sequence
- all congruent addresses are indistinguishable w.r.t eviction strategy

$\rightarrow$ adding more **unique** addresses can increase eviction rate

$\rightarrow$ **multiple** accesses to one address can increase the eviction rate

- indistinguishable $\rightarrow$ **balanced** number of accesses

Cache eviction strategy: Notation (1)

Write eviction strategies as: *P-C-D-L-S*

```
for (s = 0; s <= S - D; s += L)
  for (c = 0; c <= C; c += 1)
    for (d = 0; d <= D; d += 1)
      *a[s+d];
```

Cache eviction strategy: Notation (1)

Write eviction strategies as: *P-C-D-L-S*

S: total number of different addresses
(= set size)

```
for (s = 0; s <= S - D; s += L)
  for (c = 0; c <= C; c += 1)
    for (d = 0; d <= D; d += 1)
      *a[s+d];
```

Cache eviction strategy: Notation (1)

Write eviction strategies as: *P-C-D-L-S*

S: total number of different addresses (= set size)

D: different addresses per inner access loop

```
for (s = 0; s <= S - D; s += L)
  for (c = 0; c <= C; c += 1)
    for (d = 0; d <= D; d += 1)
      *a[s+d];
```

Cache eviction strategy: Notation (1)

Write eviction strategies as: *P-C-D-L-S*

S: total number of different addresses (= set size)

D: different addresses per inner access loop

```
for (s = 0; s <= S - D; s += L)
  for (c = 0; c <= C; c += 1)
    for (d = 0; d <= D; d += 1)
      *a[s+d];
```

L: step size of the inner access loop

Cache eviction strategy: Notation (1)

Write eviction strategies as: *P-C-D-L-S*

S: total number of different addresses (= set size)

D: different addresses per inner access loop

```
for (s = 0; s <= S - D; s += L)
  for (c = 0; c <= C; c += 1)
    for (d = 0; d <= D; d += 1)
      *a[s+d];
```

L: step size of the inner access loop

C: number of repetitions of the inner access loop

Cache eviction strategy: Notation (2)

```
for (s = 0; s <= S - D; s += L)
  for (c = 1; c <= C; c += 1)
    for (d = 1; d <= D; d += 1)
      *a[s+d];
```

Cache eviction strategy: Notation (2)

```
for (s = 0; s <= S-D; s += L)
  for (c = 1; c <= C; c += 1)
    for (d = 1; d <= D; d += 1)
      *a[s+d];
```

- P-2-2-1-4 $\rightarrow$ 1, 2, 1, 2, 2, 3, 2, 3, 3, 4, 3, 4

Cache eviction strategy: Notation (2)

```
for (s = 0; s <= S - D; s += L)
  for (c = 1; c <= C; c += 1)
    for (d = 1; d <= D; d += 1)
      *a[s+d];
```

- P-2-2-1-4 $\rightarrow$ 1, 2, 1, 2, 2, 3, 2, 3, 3, 4, 3, 4

$S = 4$

Cache eviction strategy: Notation (2)

```
for (s = 0; s <= S - D; s += L)
  for (c = 1; c <= C; c += 1)
    for (d = 1; d <= D; d += 1)
      *a[s+d];
```

- P-2-2-1-4 $\rightarrow$ 1, 2, 1, 2, 2, 3, 2, 3, 3, 4, 3, 4

Cache eviction strategy: Notation (2)

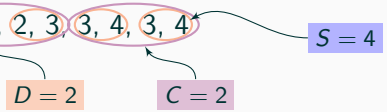
```
for (s = 0; s <= S - D; s += L)
  for (c = 1; c <= C; c += 1)
    for (d = 1; d <= D; d += 1)
      *a[s+d];
```

- P-2-2-1-4 $\rightarrow$ 1, 2, 1, 2, 2, 3, 2, 3, 3, 4, 3, 4

Cache eviction strategy: Notation (2)

```
for (s = 0; s <= S-D; s += L)
  for (c = 1; c <= C; c += 1)
    for (d = 1; d <= D; d += 1)
      *a[s+d];
```

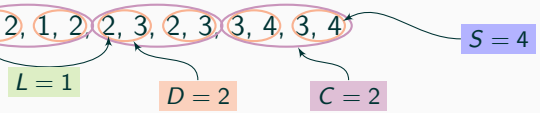
- P-2-2-1-4 $\rightarrow$ 1, 2, 1, 2, 2, 3, 2, 3, 3, 4, 3, 4



Cache eviction strategy: Notation (2)

```
for (s = 0; s <= S - D; s += L)
  for (c = 1; c <= C; c += 1)
    for (d = 1; d <= D; d += 1)
      *a[s+d];
```

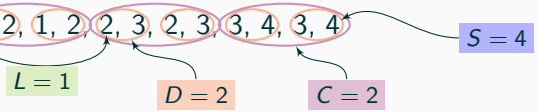
- P-2-2-1-4 $\rightarrow$ 1, 2, 1, 2, 2, 3, 2, 3, 3, 4, 3, 4



Cache eviction strategy: Notation (2)

```
for (s = 0; s <= S - D; s += L)
  for (c = 1; c <= C; c += 1)
    for (d = 1; d <= D; d += 1)
      *a[s+d];
```

- P-2-2-1-4 $\rightarrow$ 1, 2, 1, 2, 2, 3, 2, 3, 3, 4, 3, 4



- P-1-1-1-4 $\rightarrow$ 1, 2, 3, 4 $\rightarrow$ LRU eviction with set size 4

We evaluated more than 10000 strategies...[1]

strategy	# accesses	eviction rate	loop time
P-1-1-1-17	17		
P-1-1-1-20	20		

[1] Executed in a loop, on a Haswell with a 16-way last-level cache

We evaluated more than 10000 strategies...[1]

strategy	# accesses	eviction rate	loop time
P-1-1-1-17	17	74.46% ✗	
P-1-1-1-20	20	99.82% ✓	

[1] Executed in a loop, on a Haswell with a 16-way last-level cache

We evaluated more than 10000 strategies...[1]

strategy	# accesses	eviction rate	loop time
P-1-1-1-17	17	74.46% ✗	307 ns ✓
P-1-1-1-20	20	99.82% ✓	934 ns ✗

[1] Executed in a loop, on a Haswell with a 16-way last-level cache

We evaluated more than 10000 strategies...[1]

strategy	# accesses	eviction rate	loop time
P-1-1-1-17	17	74.46% ✗	307 ns ✓
P-1-1-1-20	20	99.82% ✓	934 ns ✗
P-2-1-1-17	34		

[1] Executed in a loop, on a Haswell with a 16-way last-level cache

We evaluated more than 10000 strategies...[1]

strategy	# accesses	eviction rate	loop time
P-1-1-1-17	17	74.46% ✗	307 ns ✓
P-1-1-1-20	20	99.82% ✓	934 ns ✗
P-2-1-1-17	34	99.86% ✓	

[1] Executed in a loop, on a Haswell with a 16-way last-level cache

We evaluated more than 10000 strategies...[1]

strategy	# accesses	eviction rate	loop time
P-1-1-1-17	17	74.46% ✗	307 ns ✓
P-1-1-1-20	20	99.82% ✓	934 ns ✗
P-2-1-1-17	34	99.86% ✓	191 ns ✓

[1] Executed in a loop, on a Haswell with a 16-way last-level cache

We evaluated more than 10000 strategies...[1]

strategy	# accesses	eviction rate	loop time
P-1-1-1-17	17	74.46% ✗	307 ns ✓
P-1-1-1-20	20	99.82% ✓	934 ns ✗
P-2-1-1-17	34	99.86% ✓	191 ns ✓
P-2-2-1-17	64		

[1] Executed in a loop, on a Haswell with a 16-way last-level cache

We evaluated more than 10000 strategies...[1]

strategy	# accesses	eviction rate	loop time
P-1-1-1-17	17	74.46% ✗	307 ns ✓
P-1-1-1-20	20	99.82% ✓	934 ns ✗
P-2-1-1-17	34	99.86% ✓	191 ns ✓
P-2-2-1-17	64	99.98% ✓	

[1] Executed in a loop, on a Haswell with a 16-way last-level cache

We evaluated more than 10000 strategies...[1]

strategy	# accesses	eviction rate	loop time
P-1-1-1-17	17	74.46% ✗	307 ns ✓
P-1-1-1-20	20	99.82% ✓	934 ns ✗
P-2-1-1-17	34	99.86% ✓	191 ns ✓
P-2-2-1-17	64	99.98% ✓	180 ns ✓

[1] Executed in a loop, on a Haswell with a 16-way last-level cache

Cache eviction strategies: Evaluation

We evaluated more than 10000 strategies...[1]

strategy	# accesses	eviction rate	loop time
P-1-1-1-17	17	74.46% ✗	307 ns ✓
P-1-1-1-20	20	99.82% ✓	934 ns ✗
P-2-1-1-17	34	99.86% ✓	191 ns ✓
P-2-2-1-17	64	99.98% ✓	180 ns ✓

→ more accesses, smaller execution time?

[1] Executed in a loop, on a Haswell with a 16-way last-level cache

P-1-1-1-17 (17 accesses, 307ns)

P-2-1-1-17 (34 accesses, 191ns)

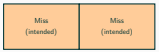
Cache eviction strategies: Illustration

P-1-1-1-17 (17 accesses, 307ns)

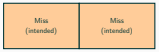
P-2-1-1-17 (34 accesses, 191ns)

Cache eviction strategies: Illustration

P-1-1-1-17 (17 accesses, 307ns)



P-2-1-1-17 (34 accesses, 191ns)



Time in ns

Cache eviction strategies: Illustration

P-1-1-1-17 (17 accesses, 307ns)

P-2-1-1-17 (34 accesses, 191ns)

Time in ns

Cache eviction strategies: Illustration

P-1-1-1-17 (17 accesses, 307ns)

P-2-1-1-17 (34 accesses, 191ns)

Time in ns

Cache eviction strategies: Illustration

P-1-1-1-17 (17 accesses, 307ns)

P-2-1-1-17 (34 accesses, 191ns)

Time in ns

P-1-1-1-17 (17 accesses, 307ns)

P-2-1-1-17 (34 accesses, 191ns)

Time in ns

P-1-1-1-17 (17 accesses, 307ns)

P-2-1-1-17 (34 accesses, 191ns)

Time in ns

P-1-1-1-17 (17 accesses, 307ns)

P-2-1-1-17 (34 accesses, 191ns)

Time in ns

P-1-1-1-17 (17 accesses, 307ns)

P-2-1-1-17 (34 accesses, 191ns)

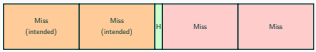
Time in ns

P-1-1-1-17 (17 accesses, 307ns)

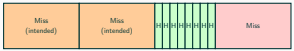
P-2-1-1-17 (34 accesses, 191ns)

Time in ns

P-1-1-1-17 (17 accesses, 307ns)

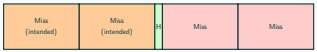


P-2-1-1-17 (34 accesses, 191ns)

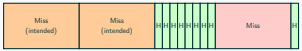


Time in ns

P-1-1-1-17 (17 accesses, 307ns)



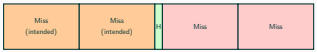
P-2-1-1-17 (34 accesses, 191ns)



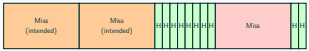
Time in ns

Cache eviction strategies: Illustration

P-1-1-1-17 (17 accesses, 307ns)

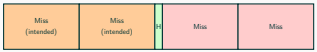


P-2-1-1-17 (34 accesses, 191ns)

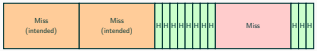


Time in ns

P-1-1-1-17 (17 accesses, 307ns)



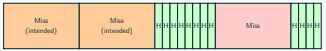
P-2-1-1-17 (34 accesses, 191ns)



Time in ns

P-1-1-1-17 (17 accesses, 307ns)

P-2-1-1-17 (34 accesses, 191ns)



Time in ns

P-1-1-1-17 (17 accesses, 307ns)

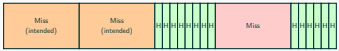
P-2-1-1-17 (34 accesses, 191ns)

Time in ns

Cache eviction strategies: Illustration

P-1-1-1-17 (17 accesses, 307ns)

P-2-1-1-17 (34 accesses, 191ns)

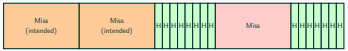


Time in ns

Cache eviction strategies: Illustration

P-1-1-1-17 (17 accesses, 307ns)

P-2-1-1-17 (34 accesses, 191ns)

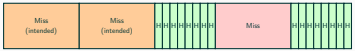


Time in ns

Cache eviction strategies: Illustration

P-1-1-1-17 (17 accesses, 307ns)

P-2-1-1-17 (34 accesses, 191ns)



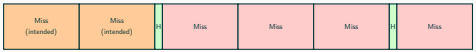
Time in ns

P-1-1-1-17 (17 accesses, 307ns)

P-2-1-1-17 (34 accesses, 191ns)

Time in ns

P-1-1-1-17 (17 accesses, 307ns)

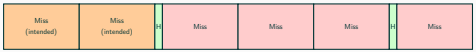


P-2-1-1-17 (34 accesses, 191ns)

Time in ns

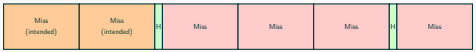
Cache eviction strategies: Illustration

P-1-1-1-17 (17 accesses, 307ns)



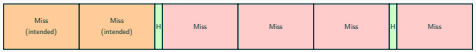
P-2-1-1-17 (34 accesses, 191ns)

P-1-1-1-17 (17 accesses, 307ns)



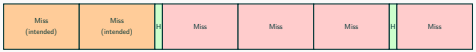
P-2-1-1-17 (34 accesses, 191ns)

P-1-1-1-17 (17 accesses, 307ns)

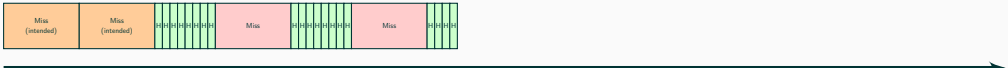


P-2-1-1-17 (34 accesses, 191ns)

P-1-1-1-17 (17 accesses, 307ns)

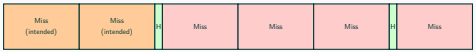


P-2-1-1-17 (34 accesses, 191ns)



Cache eviction strategies: Illustration

P-1-1-1-17 (17 accesses, 307ns)

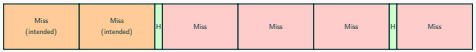


P-2-1-1-17 (34 accesses, 191ns)

Time in ns

Cache eviction strategies: Illustration

P-1-1-1-17 (17 accesses, 307ns)



P-2-1-1-17 (34 accesses, 191ns)

Time in ns

P-1-1-1-17 (17 accesses, 307ns)

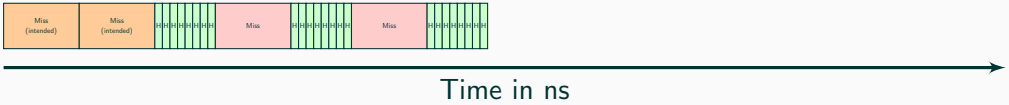
P-2-1-1-17 (34 accesses, 191ns)

Time in ns

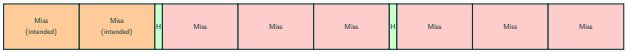
Cache eviction strategies: Illustration

P-1-1-1-17 (17 accesses, 307ns)

P-2-1-1-17 (34 accesses, 191ns)



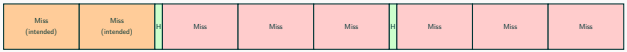
P-1-1-1-17 (17 accesses, 307ns)



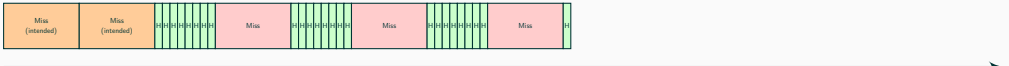
P-2-1-1-17 (34 accesses, 191ns)

Cache eviction strategies: Illustration

P-1-1-1-17 (17 accesses, 307ns)

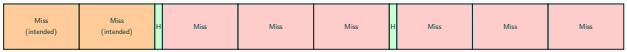


P-2-1-1-17 (34 accesses, 191ns)

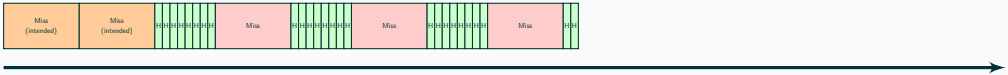


Cache eviction strategies: Illustration

P-1-1-1-17 (17 accesses, 307ns)

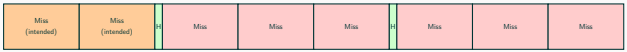


P-2-1-1-17 (34 accesses, 191ns)

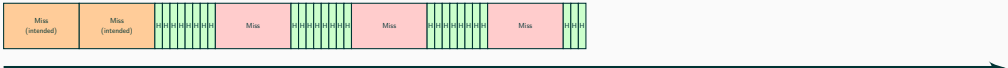


Cache eviction strategies: Illustration

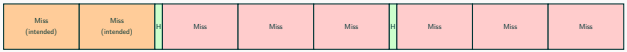
P-1-1-1-17 (17 accesses, 307ns)



P-2-1-1-17 (34 accesses, 191ns)

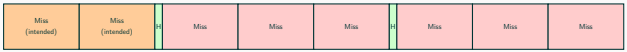


P-1-1-1-17 (17 accesses, 307ns)



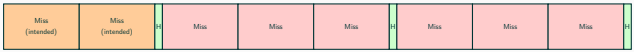
P-2-1-1-17 (34 accesses, 191ns)

P-1-1-1-17 (17 accesses, 307ns)

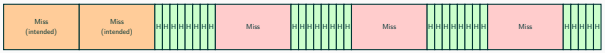


P-2-1-1-17 (34 accesses, 191ns)

P-1-1-1-17 (17 accesses, 307ns)



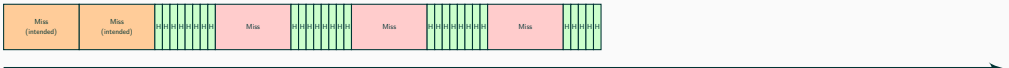
P-2-1-1-17 (34 accesses, 191ns)



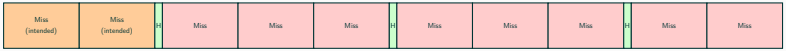
Time in ns

P-1-1-1-17 (17 accesses, 307ns)

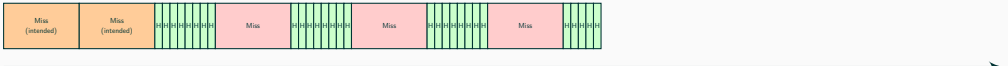
P-2-1-1-17 (34 accesses, 191ns)



P-1-1-1-17 (17 accesses, 307ns)

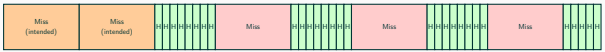


P-2-1-1-17 (34 accesses, 191ns)



P-1-1-1-17 (17 accesses, 307ns)

P-2-1-1-17 (34 accesses, 191ns)

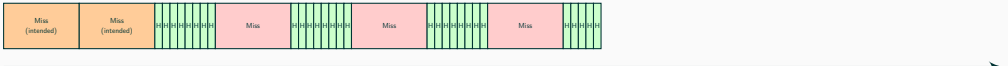


Time in ns

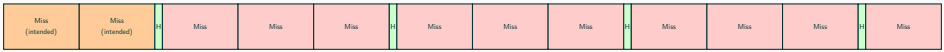
Cache eviction strategies: Illustration

P-1-1-1-17 (17 accesses, 307ns)

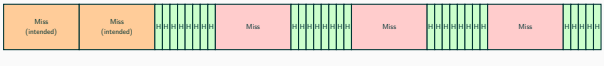
P-2-1-1-17 (34 accesses, 191ns)



P-1-1-1-17 (17 accesses, 307ns)



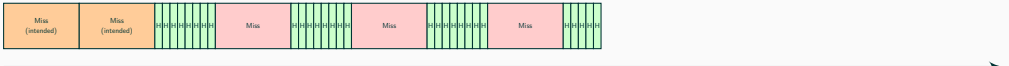
P-2-1-1-17 (34 accesses, 191ns)

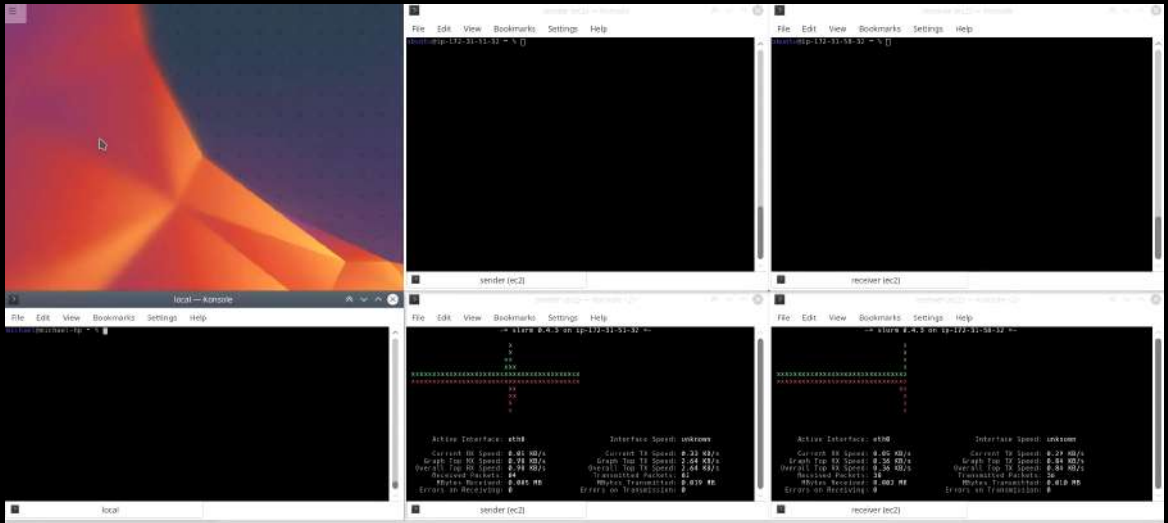


Cache eviction strategies: Illustration

P-1-1-1-17 (17 accesses, 307ns)

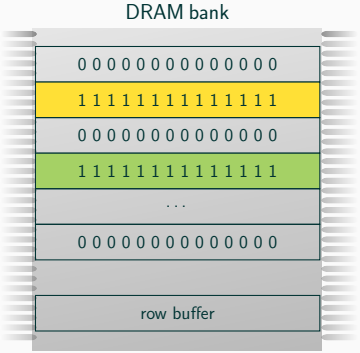
P-2-1-1-17 (34 accesses, 191ns)



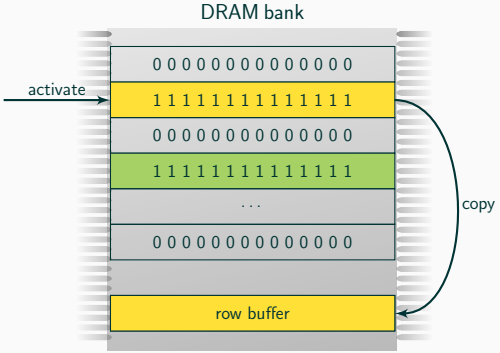


Hello from the other side (DEMO): Video streaming over cache covert channel

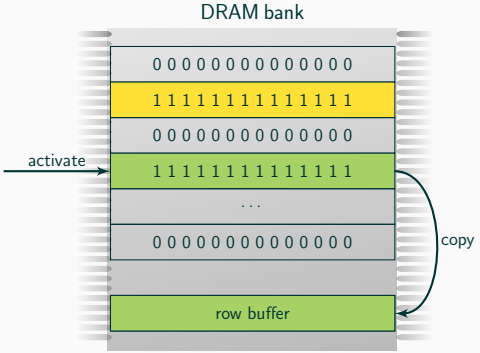
Line 17, Column 1 INSERT Soft Tabs: 4(8) UTF-8 Normal



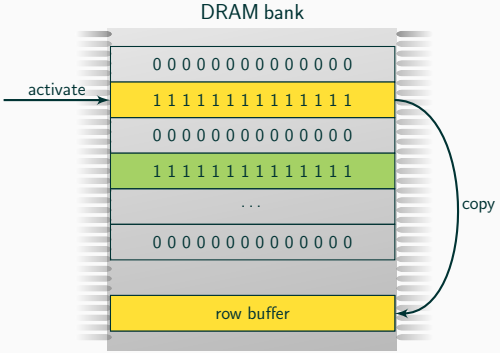
- Cells leak → repetitive refresh necessary
- Maximum interval between refreshes to guarantee data integrity
- Cells leak faster upon proximate accesses → Rowhammer



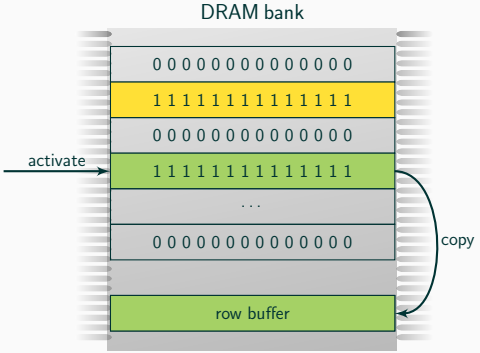
- Cells leak → repetitive refresh necessary
- Maximum interval between refreshes to guarantee data integrity
- Cells leak faster upon proximate accesses → Rowhammer



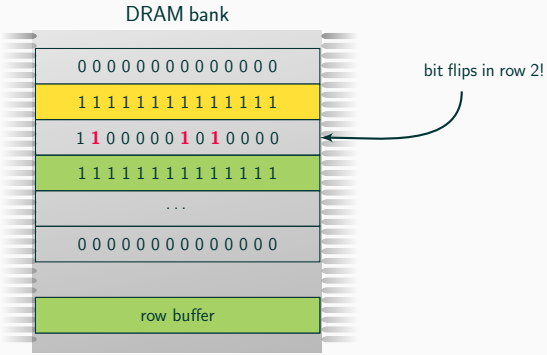
- Cells leak → repetitive refresh necessary
- Maximum interval between refreshes to guarantee data integrity
- Cells leak faster upon proximate accesses → Rowhammer



- Cells leak → repetitive refresh necessary
- Maximum interval between refreshes to guarantee data integrity
- Cells leak faster upon proximate accesses → Rowhammer



- Cells leak → repetitive refresh necessary
- Maximum interval between refreshes to guarantee data integrity
- Cells leak faster upon proximate accesses → Rowhammer



- Cells leak → repetitive refresh necessary
- Maximum interval between refreshes to guarantee data integrity
- Cells leak faster upon proximate accesses → Rowhammer

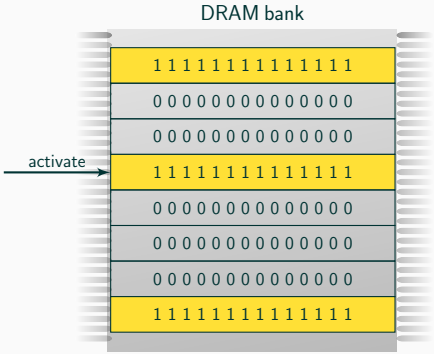
- There are two different hammering techniques

- There are two different hammering techniques
- #1: Hammer one row next to victim row and other random rows

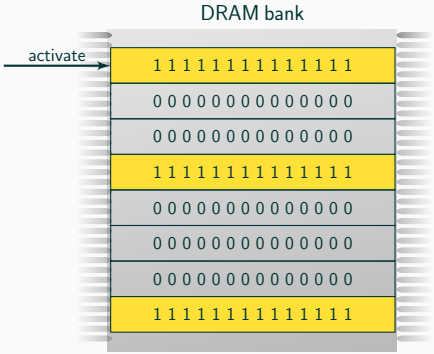
- There are two different hammering techniques
- #1: Hammer one row next to victim row and other random rows
- #2: Hammer two rows neighboring victim row

- There are **three** different hammering techniques
- #1: Hammer one row next to victim row and other random rows
- #2: Hammer two rows neighboring victim row
- #3: Hammer only one row next to victim row

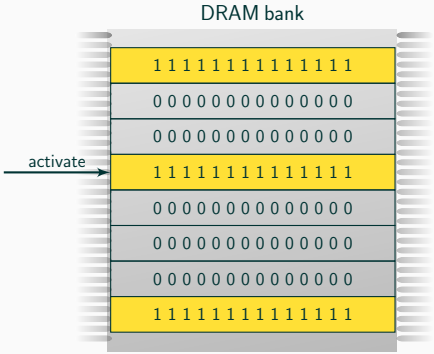
#1 - Single-sided hammering



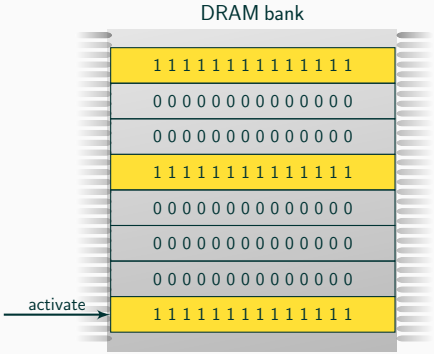
#1 - Single-sided hammering



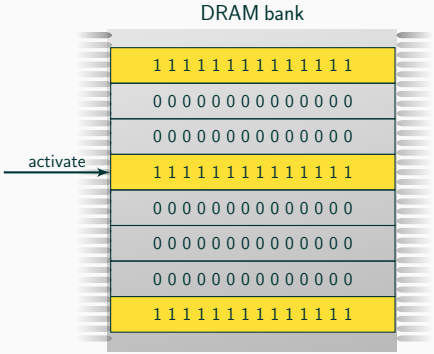
#1 - Single-sided hammering



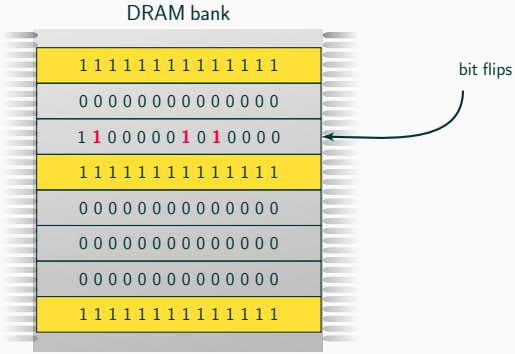
#1 - Single-sided hammering



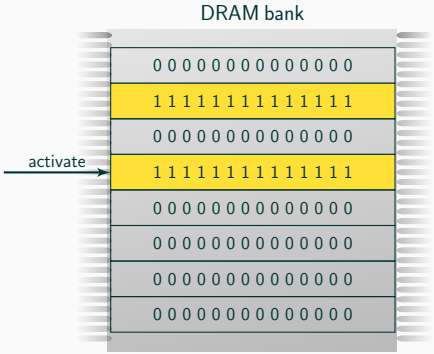
#1 - Single-sided hammering



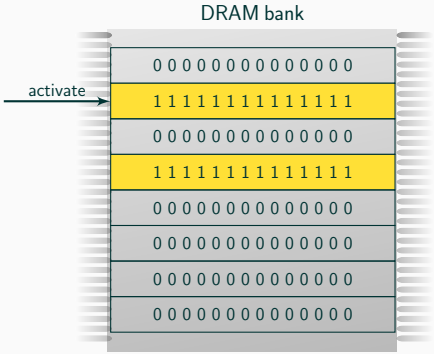
#1 - Single-sided hammering



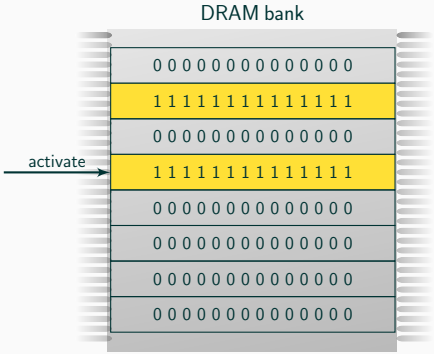
#2 - Double-sided hammering



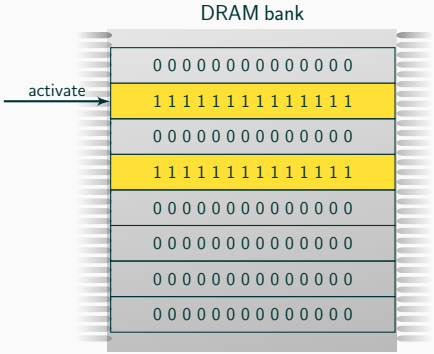
#2 - Double-sided hammering



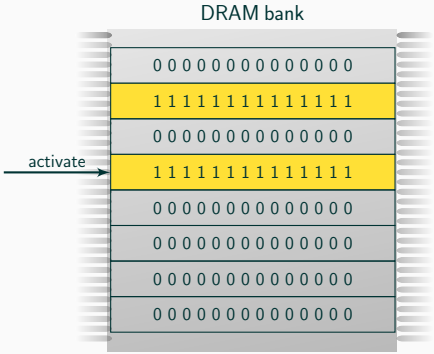
#2 - Double-sided hammering



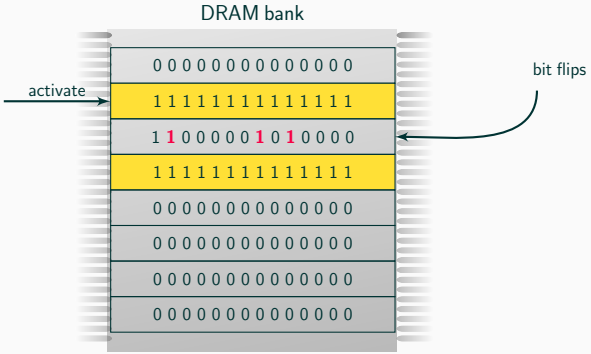
#2 - Double-sided hammering



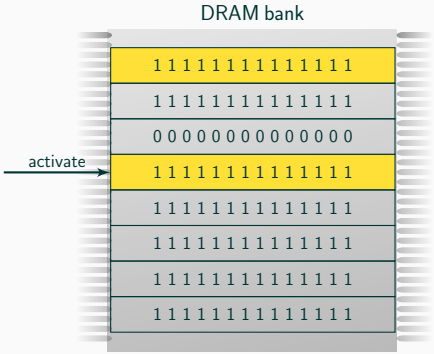
#2 - Double-sided hammering



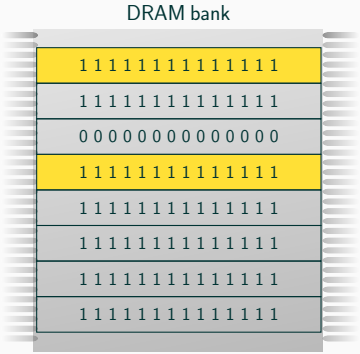
#2 - Double-sided hammering



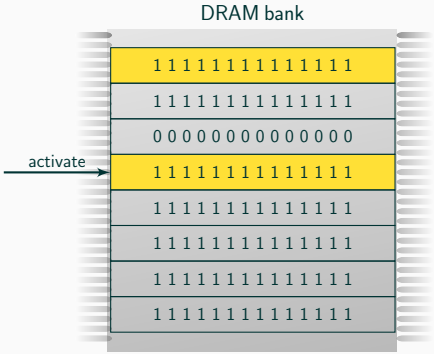
#3 - One-location hammering



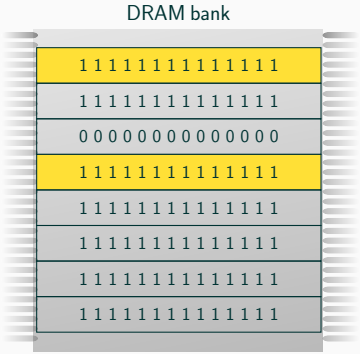
#3 - One-location hammering



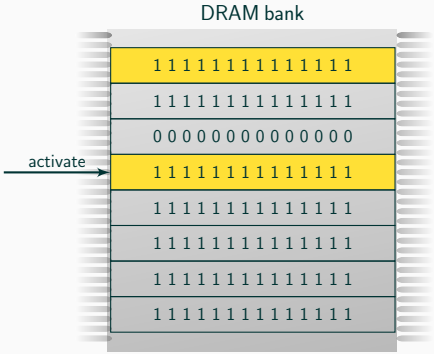
#3 - One-location hammering



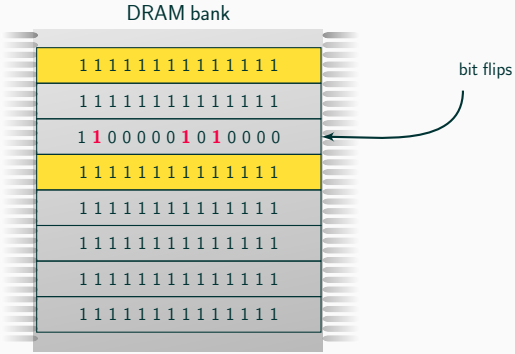
#3 - One-location hammering



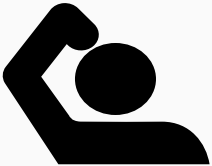
#3 - One-location hammering



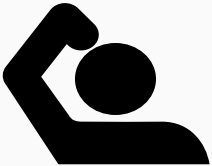
#3 - One-location hammering



How to exploit random bit flips?



How to exploit random bit flips?



- They are not random → highly reproducible flip pattern!

- They are not random → highly reproducible flip pattern!
  1. Choose a data structure that you can place at arbitrary memory locations

- They are not random → highly reproducible flip pattern!
  1. Choose a data structure that you can place at arbitrary memory locations
  2. Scan for “good” flips

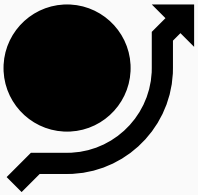
- They are not random → highly reproducible flip pattern!
  1. Choose a data structure that you can place at arbitrary memory locations
  2. Scan for “good” flips
  3. Place data structure there

How to exploit random bit flips?

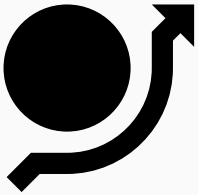
- They are not random → highly reproducible flip pattern!
  1. Choose a data structure that you can place at arbitrary memory locations
  2. Scan for “good” flips
  3. Place data structure there
  4. Trigger bit flip again

What if we cannot target kernel pages?

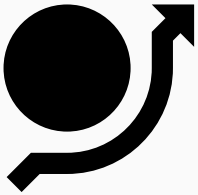
What if we cannot target kernel pages?



What if we cannot target kernel pages?

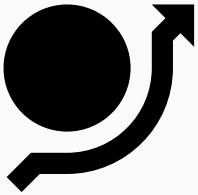


What if we cannot target kernel pages?



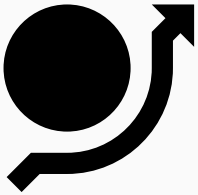
- Many applications perform actions as root

What if we cannot target kernel pages?



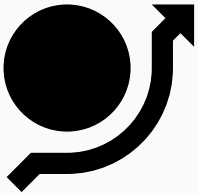
- Many applications perform actions as root

What if we cannot target kernel pages?



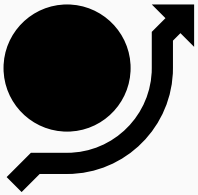
- Many applications perform actions as root
- They can be used by unprivileged users as well

What if we cannot target kernel pages?

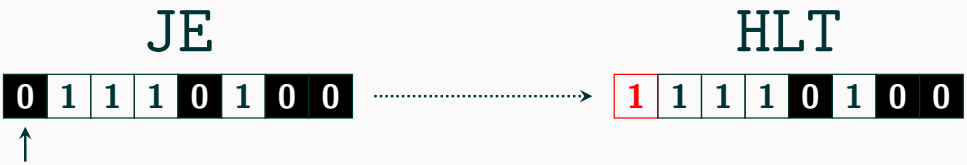


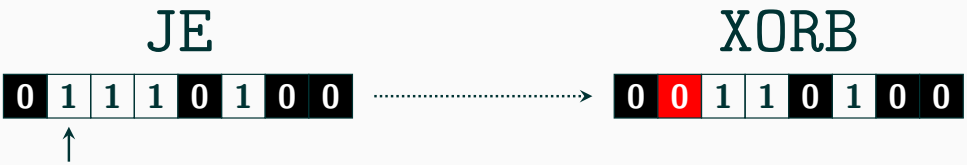
- Many applications perform actions as root
- They can be used by unprivileged users as well

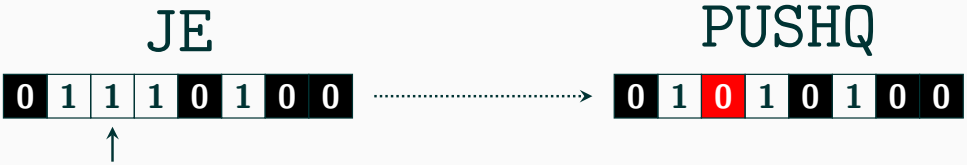
What if we cannot target kernel pages?



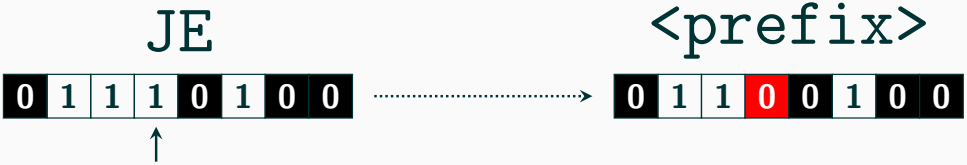
- Many applications perform actions as root
- They can be used by unprivileged users as well
- `sudo`

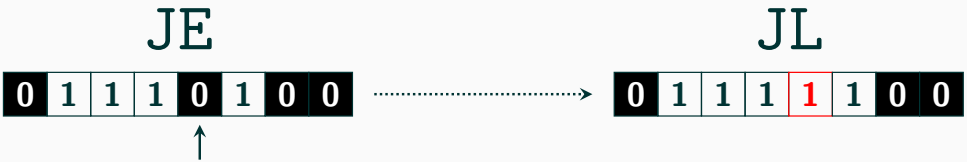

JE
0 1 1 1 0 1 0 0
HLT
1 1 1 1 0 1 0 0


JE
0 1 1 1 0 1 0 0
XORB
0 0 1 1 0 1 0 0


JE
0 1 1 1 0 1 0 0
PUSHQ
0 1 0 1 0 1 0 0

Opcode Flipping - Conditional Jump




JE
0 1 1 1 0 1 0 0
JL
0 1 1 1 1 1 0 0

Opcode Flipping - Conditional Jump


JE
0 1 1 1 0 1 0 0
JBE
0 1 1 1 0 1 1 0


JE
0 1 1 1 0 1 0 0
JNE
0 1 1 1 0 1 0 1

How widespread is the issue?

How widespread is the issue?

- 85% affected [Kim+14] (see Figure)

How widespread is the issue?

- 85% affected [Kim+14] (see Figure)
- 52% affected [SD15]

How widespread is the issue?

- 85% affected [Kim+14] (see Figure)
- 52% affected [SD15]

- First believed to be safe

- 85% affected [Kim+14] (see Figure)
- 52% affected [SD15]

- First believed to be safe
- We showed bit flips [Pes+16]

How widespread is the issue?

- 85% affected [Kim+14] (see Figure)
- 52% affected [SD15]

- First believed to be safe
- We showed bit flips [Pes+16]
- 67% affected [Lan16]

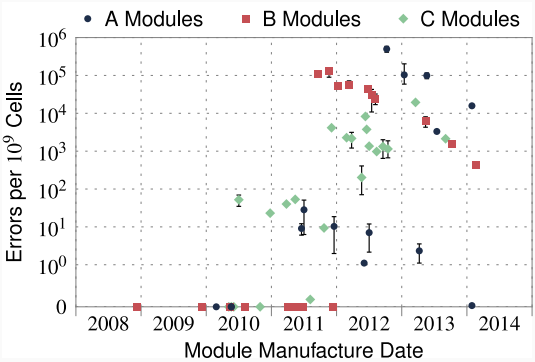
How widespread is the issue?

DDR3

- 85% affected [Kim+14] (see Figure)
- 52% affected [SD15]

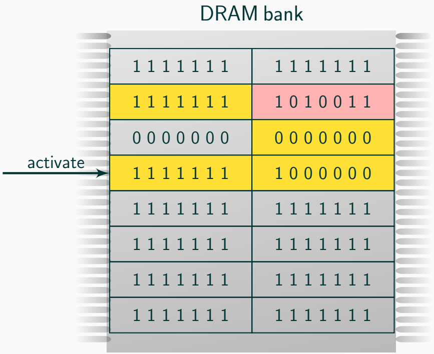
DDR4

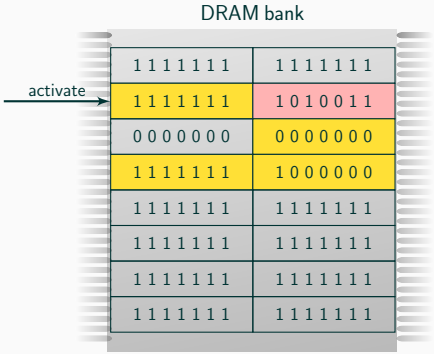
- First believed to be safe
- We showed bit flips [Pes+16]
- 67% affected [Lan16]

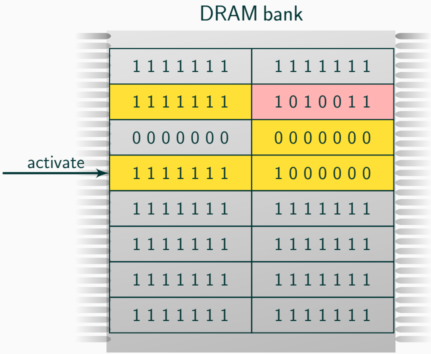


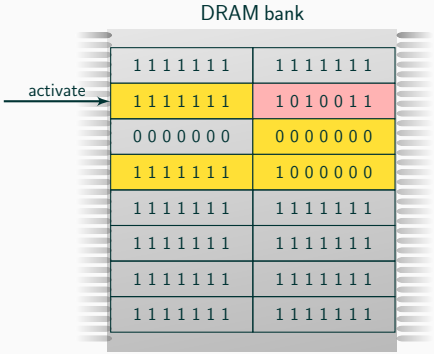

BIT FLIPS
BIT FLIPS EVERYWHERE

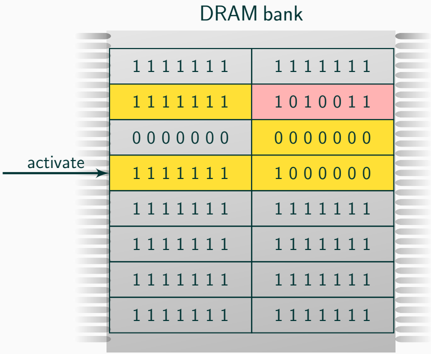
RAMbleed

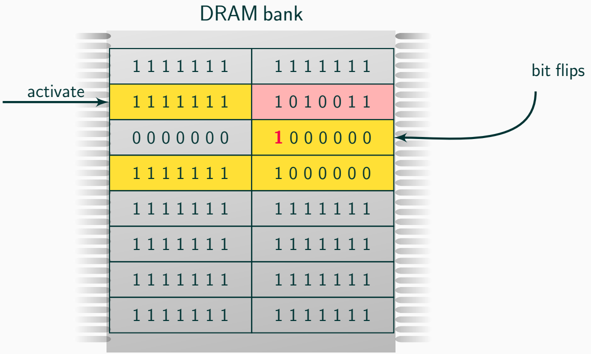

DRAM bank
activate
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 0 1 0 0 1 1
0 0 0 0 0 0 0
0 0 0 0 0 0 0
1 1 1 1 1 1 1
1 0 0 0 0 0 0
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1


DRAM bank
activate
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 0 1 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1


DRAM bank
activate
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 0 1 0 0 1 1
0 0 0 0 0 0 0
0 0 0 0 0 0 0
1 1 1 1 1 1 1
1 0 0 0 0 0 0
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1


DRAM bank
activate
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 0 1 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1


DRAM bank
activate
1 1 1 1 1 1 1 | 1 1 1 1 1 1 1
1 1 1 1 1 1 1 | 1 0 1 0 0 1 1
0 0 0 0 0 0 0 | 0 0 0 0 0 0 0
1 1 1 1 1 1 1 | 1 0 0 0 0 0 0
1 1 1 1 1 1 1 | 1 1 1 1 1 1 1
1 1 1 1 1 1 1 | 1 1 1 1 1 1 1
1 1 1 1 1 1 1 | 1 1 1 1 1 1 1
1 1 1 1 1 1 1 | 1 1 1 1 1 1 1


DRAM bank
activate
bit flips
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 0 1 0 0 1 1
0 0 0 0 0 0 0
1 0 0 0 0 0 0
1 1 1 1 1 1 1
1 0 0 0 0 0 0
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1

- We want the performance optimizations

Countermeasures are Difficult

- We want the performance optimizations
- Many side-channel attacks exploit intended behavior

Countermeasures are Difficult

- We want the performance optimizations
- Many side-channel attacks exploit intended behavior
- Often a trade-off between security and performance

Countermeasures are Difficult

- We want the performance optimizations
- Many side-channel attacks exploit intended behavior
- Often a trade-off between security and performance
- Every optimization is potentially a side channel

- We won't get rid of side channels

The Future

- We won't get rid of side channels
- More optimizations $\rightarrow$ more side channels

- We won't get rid of side channels
- More optimizations $\rightarrow$ more side channels
- But: low hanging fruits will disappear

Introduction to Microarchitectural Attacks

Daniel Gruss

June 18, 2019

Graz University of Technology