Get-to-know your system

Martin Radev

January 14, 2019

Overview

Motivation

- Vendors typically don't share all of the info for a chip
- At a chip company you might implement similar internal verification tests
- Hands-on experience with how the system works

Determining cache line size

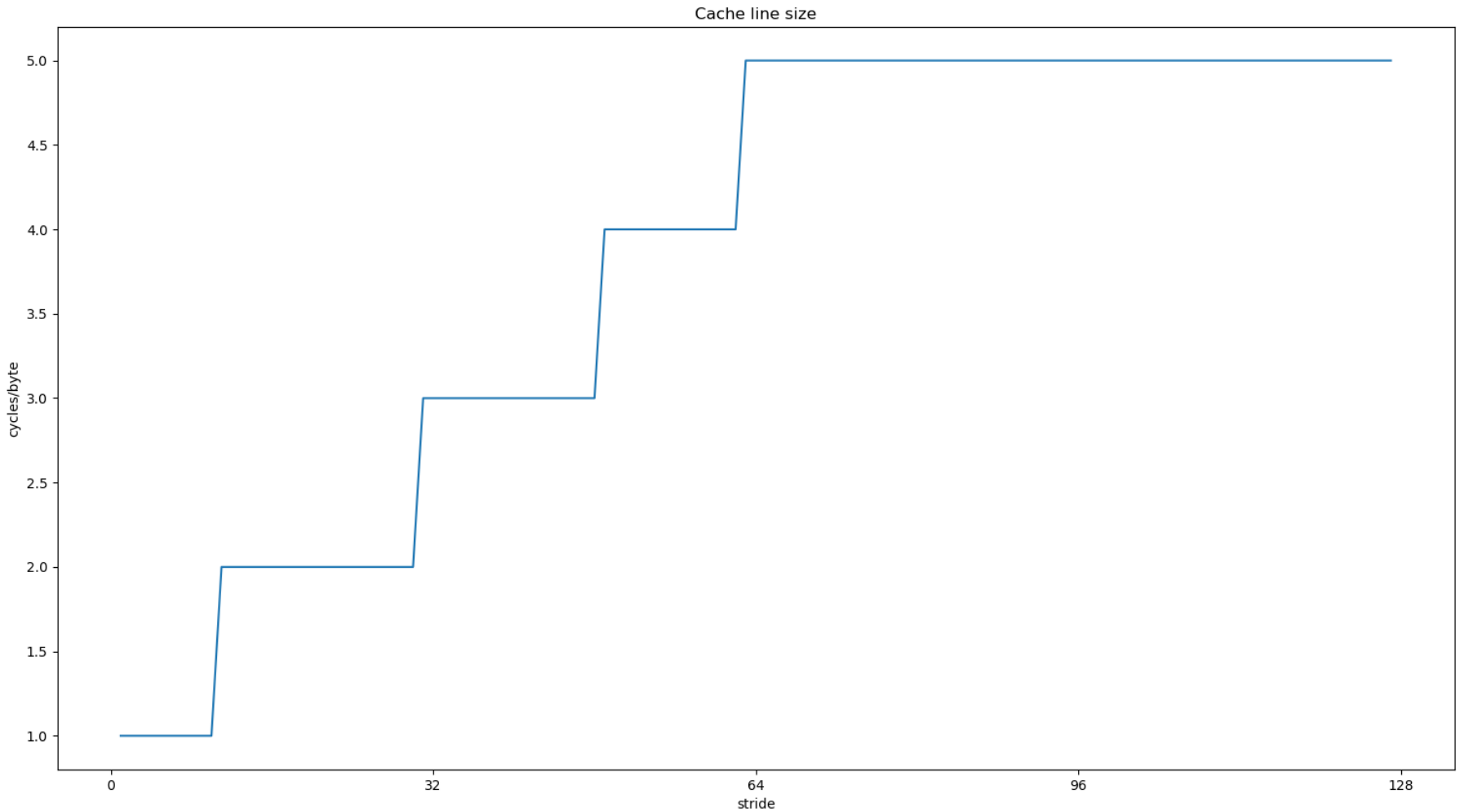
- Determine the amortized cost of reading a byte with a given stride
- Small stride $\rightarrow$ Cache line reuse $\rightarrow$ Small amortized cost
- Big stride $\rightarrow$ New cache line per read $\rightarrow$ cost of reading a line

Determining cache line size

Code

```
1   for (u32 stride = 1; stride < 128; ++stride)
2   {
3       u8 tmp;
4       u64 va = (u64)buffer;
5       u64 t1 = time_start();
6       for (c = 0; c < buffer_size; c += stride, va += stride)
7       {
8           asm volatile(".intel_syntax noprefix\n\t"
9                        "mov %0, BYTE [%1]\n\t"
10                       ".att_syntax prefix\n\t"
11                       : "=r"(tmp)
12                       : "r"(va));
13      }
14      u64 t2 = time_end();
15      u64 cyclesPerByte = (t2-t1) / (buffer_size / stride);
16  }
```

Cache line plot

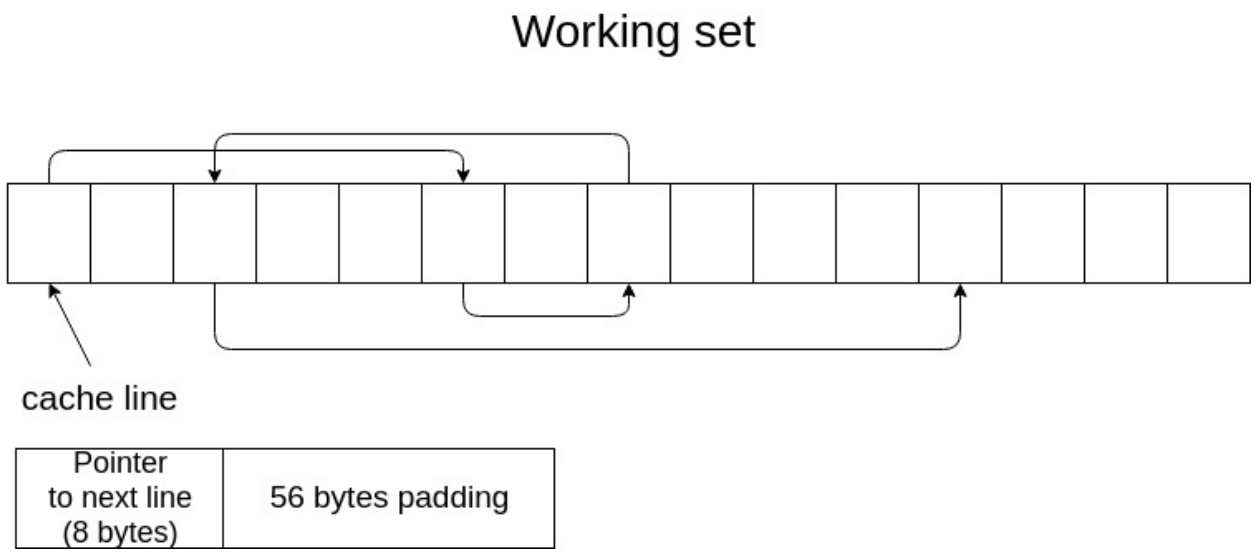


Determining data cache size - idea

- Vary size of the working set to exercise capacity misses
- Very small set $\rightarrow$ L1 data cache $\rightarrow$ kind of fast
- Small set $\rightarrow$ L2 data cache $\rightarrow$ slowish
- Medium set $\rightarrow$ L3 data cache $\rightarrow$ slow
- Big set $\rightarrow$ RAM $\rightarrow$ very slow
- etc

Determining data cache size

- Working set - continuous portion of virtual memory
  Lessen conflict misses
- Access randomly and uniformly to avoid prefetching
- Typically done via **pointer chasing**



Pointer chasing - sequence generation

Code

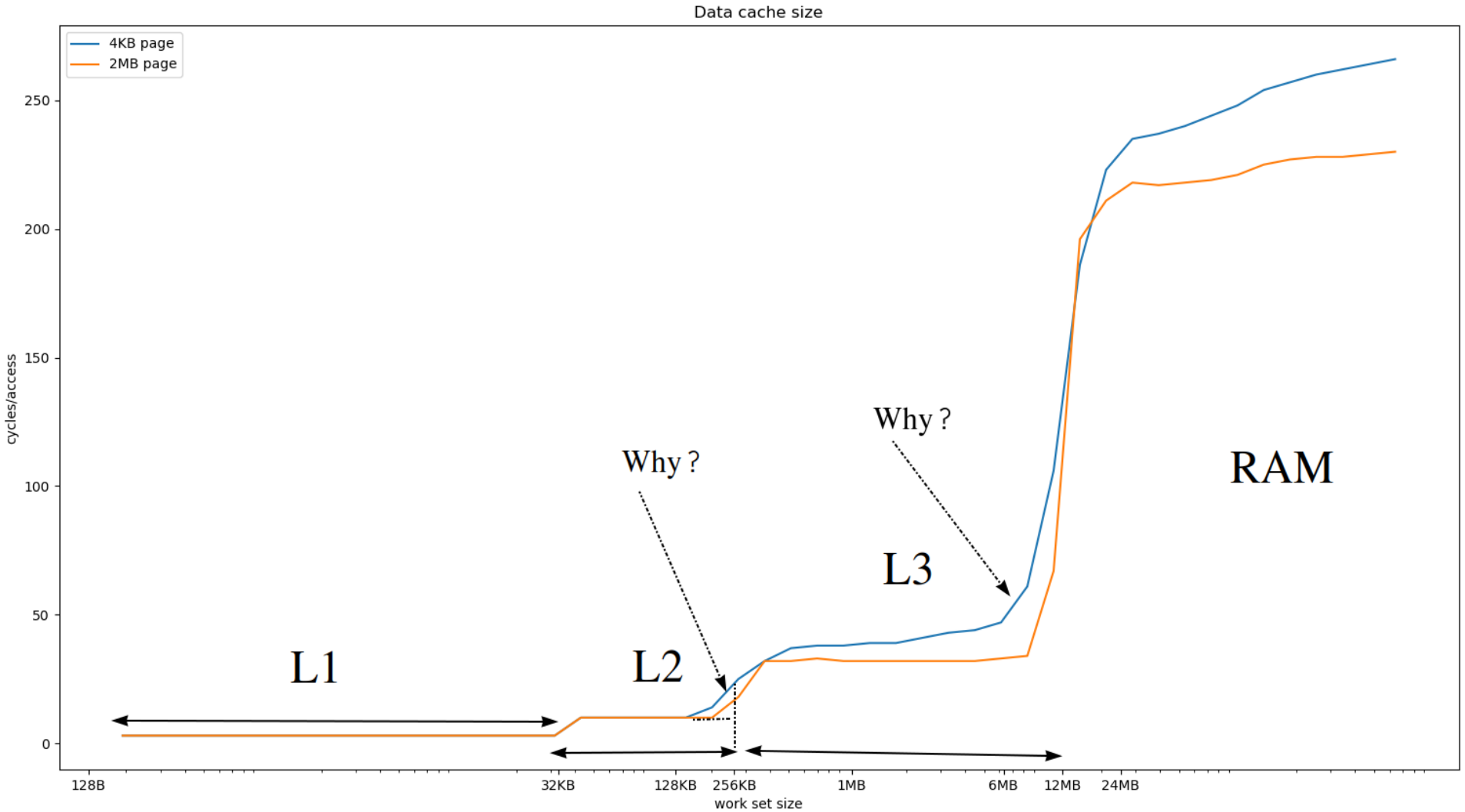
```
1   typedef struct BlockDecl
2   {
3       struct BlockDecl *next;
4       u8 padding[56U];
5   } Block;
6
7   void generateRandomSequence(Block *blocks, size_t numBlocks)
8   {
9       std::vector<size_t> seq(numBlocks-1);
10      for (size_t i = 1; i < numBlocks; ++i)
11      {
12          seq[i-1] = i;
13      }
14      std::shuffle(seq.begin(), seq.end(), std::default_random_engine(111U));
15      seq.push_back(0);
16      size_t prevAddr = 0U;
17      for (size_t i = 0U; i < numBlocks; ++i)
18      {
19          size_t nextAddr = seq[i];
20          blocks[prevAddr].next = &blocks[nextAddr];
21          prevAddr = nextAddr;
22      }
23  }
```

Determining data cache size

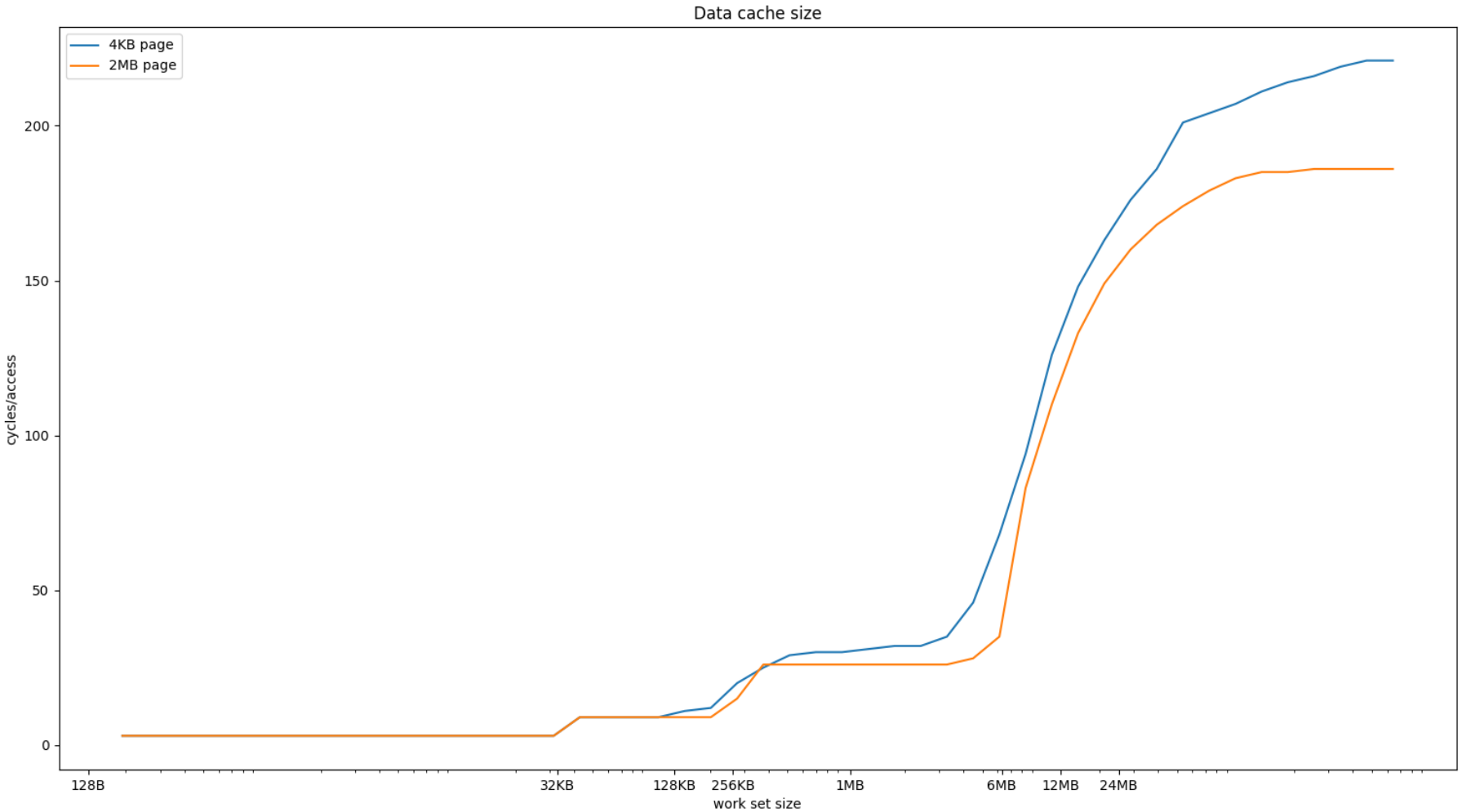
Code

```
1   const size_t kMaxBlocks = 12*1024U*1024U;
2   Block *blocks = (Block*)mmap(NULL, sizeof(Block) * kMaxBlocks, PROT_READ|PROT_WRITE,
3                                MAP_PRIVATE|MAP_POPULATE|MAP_ANONYMOUS, -1, 0U);
4   if (blocks == MAP_FAILED)
5   {
6       printf("Failed to allocate: %s\n", strerror(errno));
7       return;
8   }
9   for (size_t j = 3U; j < kMaxBlocks; j = (j*15U)/11U)
10  {
11      generateRandomSequence(blocks, j);
12      size_t kMaxAccesses = kMaxBlocks; // Make sure all blocks are accessed
13      u64 t1 = time_start()
14      Block *b = &blocks[0];
15      for (size_t i = 0U; i < kMaxAccesses; ++i)
16          b = b->next;
17      u64 t2 = time_end();
18      PRINT t2-t1
19      if (b == NULL) // Prevent compiler from optimizing out the loop above
20          exit(1);
21  }
```

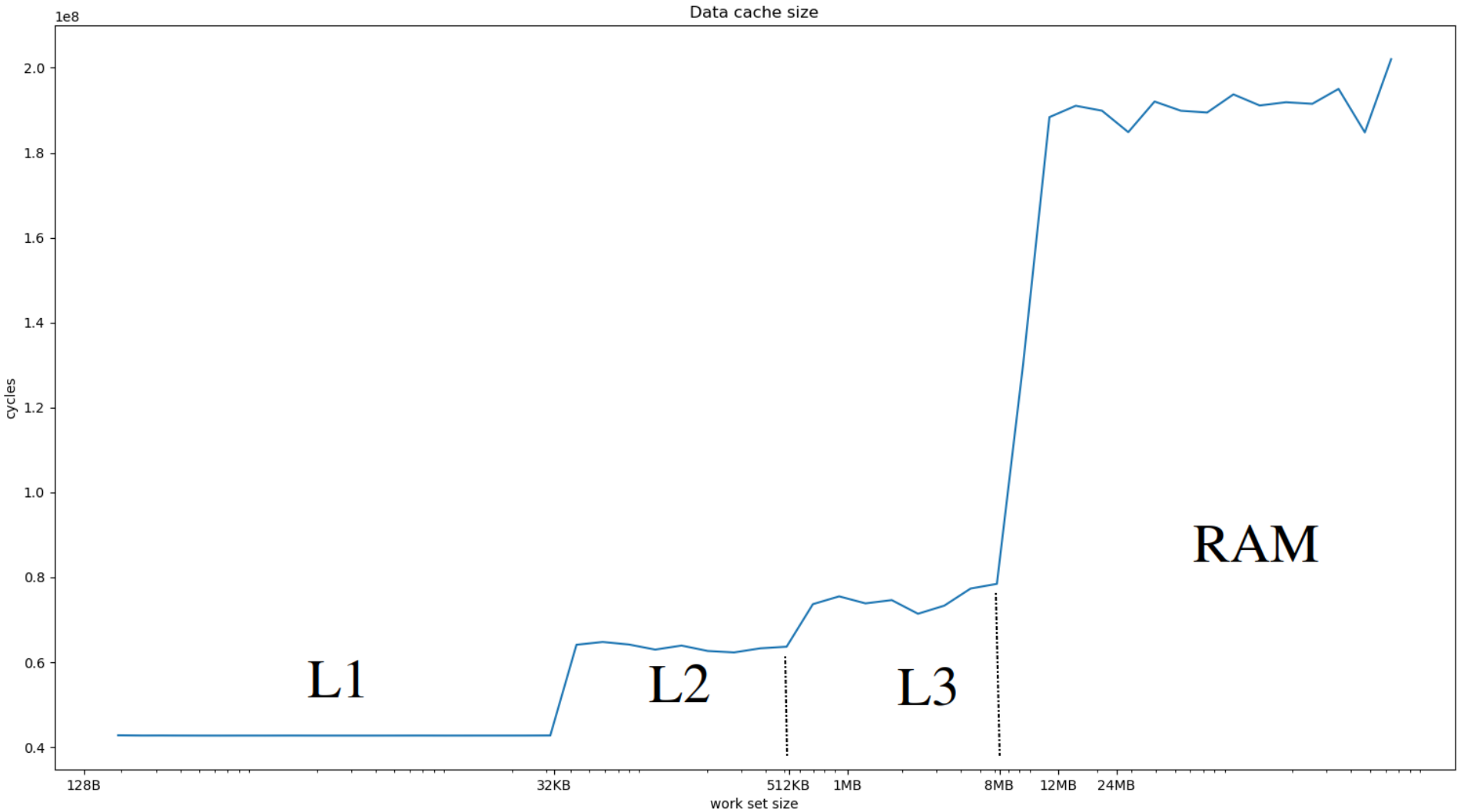
Data cache plot - I7 3930K (Sandy bridge)



Data cache plot - I7 4900MQ (Haswell)



Data cache plot - Ryzen 2700X



Some observations

- Using 2MB pages improves perf as buffer size increases
- Increase in cost/read as L2/L3 capacity is reached
  Possibly many reasons:
  - Increase in TLB capacity misses for 4KB pages
  - L2 is shared among I-L1,D-L1
  - L3 is shared among cores
  - Poor page coloring?
    This can be an issue for L2/L3 where the cache is very big!

What about the instruction cache?

- L1 split into I-cache and D-cache
- The I-cache could have different parameters
- Could be determined via pointer chasing again
- ... But requires some hackery

Executable virtual pages on Linux

Three types of page protection:

- Readable - R
- Writable - W
- Executable - E

Data is typically marked as RW for security
We can create a mapping tagged with RWE using mmap
Thus, we can dynamically generate code and get it to execute :)
Other ways out there too - modify the section headers in the exe
But your AV SW might complain :)

Example code

```
1   mmap(NULL, sizeof(Block) * kMaxBlocks,
2         PROT_READ|PROT_WRITE|PROT_EXEC,
3         MAP_PRIVATE|MAP_POPULATE|MAP_ANONYMOUS,
4         -1, 0U);
```

Pointer chasing in code

- Same idea
- BUT CPU must fetch instructions from different locations, not data
- Thus, each cache line has to contain instructions to force the PC (EIP) to change

Example code as it would be in memory

```
1   u64 t1 = begin_time()
2   Call function cache_line_1(counter)
3   u64 t2 = end_time()
4   print t2-t1
5   ... some other stuff ...
6
7   func Cache_line_1(counter)
8       decrement counter
9       if (counter != 0)
10      {
11          jump to next random cache line
12      }
13      return
14
15  ... some garbage here ...
16  ... many more lines here ...
```

Translating the code into assembly

Pseudo-code

```
1   func Cache_line_1(counter)
2       decrement counter
3       if (counter != 0)
4       {
5           jump to next random cache line
6       }
7       return
```

The real thing?

```
1   dec eax ; This modifies the flags
2   jne 0xAABBCCDD ; the address here varies depending on where the next line is located
3   ret ; in case we don't jump, then return
```

Not quite there yet!

The CPU doesn't understand mnemonic names

It understands machine code

Translating assembly into machine code

The formula:

- Google "dec x86 encoding"
- Be careful of addressing mode
- Copy-paste byte values into your C code

The real thing

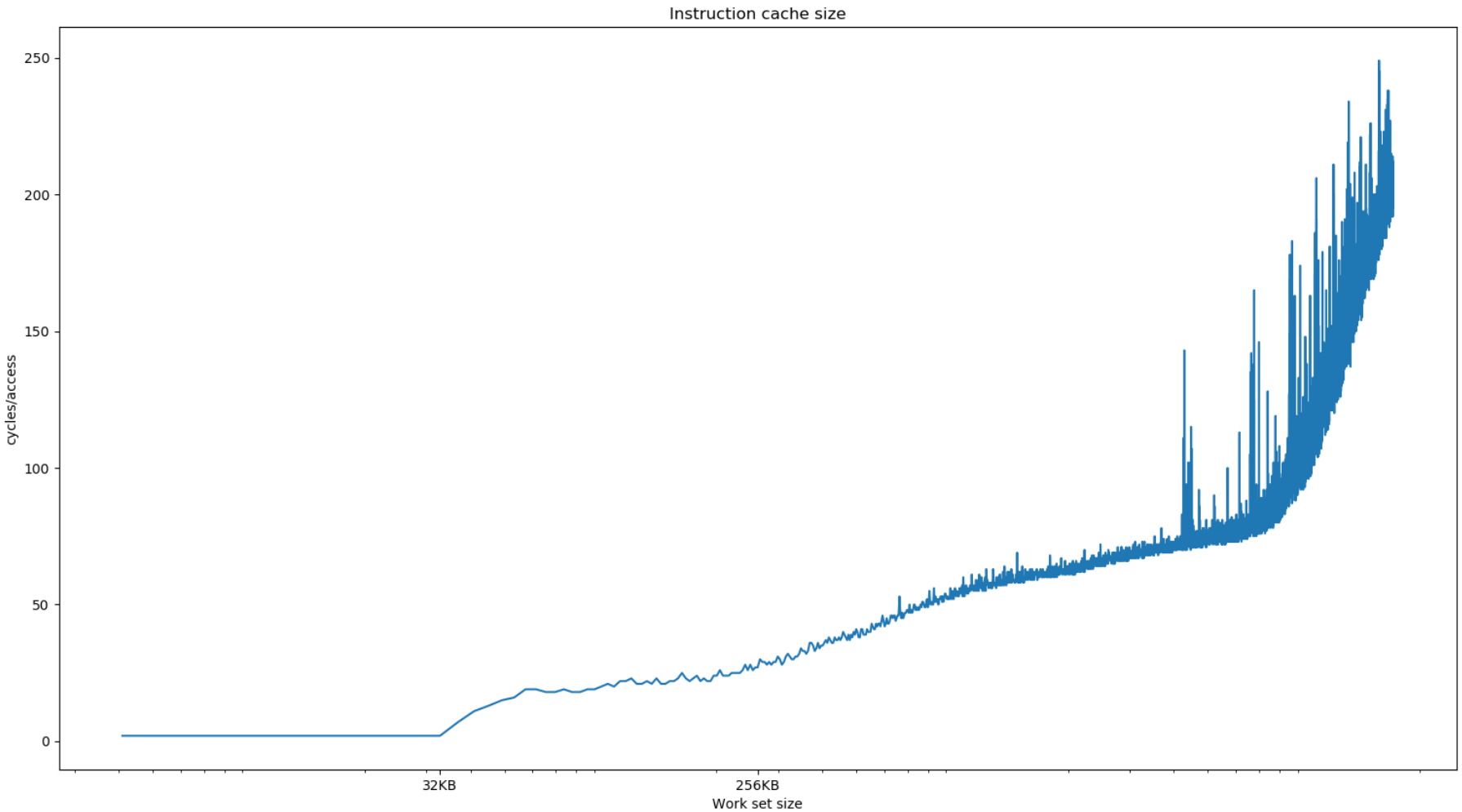
FF C8 $\rightarrow$ dec eax
0F 85 DD CC BB AA $\rightarrow$ jne 0xAABBCCDD
C3 $\rightarrow$ ret

Measuring L1 I-cache size

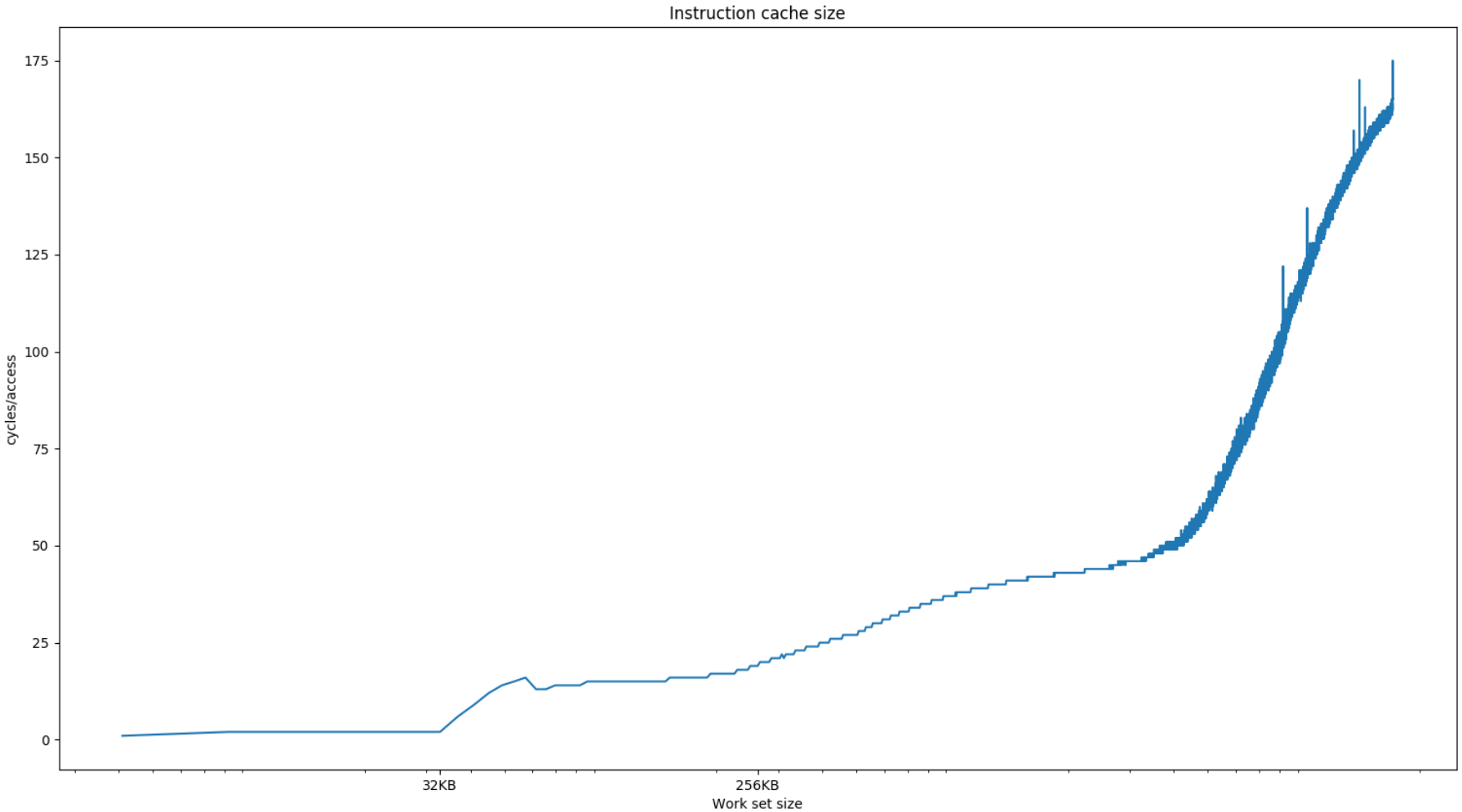
Code

```
1   for (size_t j = 64U; j < kMaxBlocks; j += 64) {
2       size_t prevAddr = 0U;
3       for (size_t i = 0; i < j; ++i) {
4           size_t nextAddr = (prevAddr + j - 1U) % j;
5           u64 nextBlockVA = (u64)(&blocks[nextAddr]);
6           u8 *block = (u8*)&blocks[prevAddr];
7           // Determine relative offset to next block
8           u64 offset = nextBlockVA - (u64)block;
9           u32 offsetU32 = (u32)(offset & 0xFFFFFFFFU);
10          u32 offsetU32Adj = offsetU32 - 8U; // 8 bytes since dec and jne take 8 bytes
11          block[0] = 0xFFU; block[1] = 0xC8U; // dec eax
12          block[2] = 0x0FU; block[3] = 0x85U; // jne
13          memcpy(&block[4], &offsetU32Adj, 4U); // relative offset
14          block[8] = 0xC3U; // ret
15          prevAddr = nextAddr;
16      }
17      u64 t1 = time_start();
18      asm volatile(".intel_syntax noprefix\n\t"
19                   "mov eax, %0\n\t"
20                   "call %1\n\t"
21                   ".att_syntax prefix\n\t"
22                   : /* no output */
23                   : "r"(kMaxAccesses), "r"((u64)blocks)
24                   : "eax", "flags");
25      u64 t2 = time_end();
26      PRINT t2-t1
27  }
```

I-cache plot (Sandy Bridge)



I-cache plot (Haswell)



Performance is not only about the I/D-cache

- Modern systems have to translate virtual addresses to physical
- Protection, privilege level have to be checked
- Requires caching again, but at a different granularity
  $\rightarrow$ page granularity
- TLB - translation lookaside buffer
- Typically L1 I-TLB, L1 D-TLB, L2 S(hared)TLB

Very fast intro to TLB

- an entry in a TLB can contain:
  - Physical address bits
  - Tag
  - Protection, privilege, other stuff
- 4KB page virtual address:
  0xAABBCCDD = AABBC (tag, virtual) . CDD (page offset, physical)
- Almost always you need the physical address
- The TLB gives you the uppermost physical bits

Cache varieties

Caches and TLBs may need to cooperate
Cache variations in terms of access:

- Virtually-indexed, virtually-tagged cache
  → No need to translate (maybe?)
  → Cache flush might be necessary on ctx switch → Cache aliasing is a problem
- Physically-indexed, physically-tagged cache
  → Translate, then read the cache
  → More latency, more flexible (?)
- Virtually-indexed, physically-tagged cache
  → TLB access and cache access often done in parallel
  → No aliasing potentially, no flush on ctx switch
  → This is a common approach

Page sizes

- We want to resolve TLB-misses ASAP
- But some memory systems are all about throughput
  Think your x86 CPU. Or even better, your GPU :)
- To reduce TLB misses, just have bigger pages
  On x86: **4KB, 2MB, 1GB**
- 4KB page: 0xAABBCCDD = AABBC (tag) . CDD (page offset)
  Important: CDD is also the concatenation of cache set index and line offset for the L1 cache
  Not true for L2 :)
- It's more complicated

Getting TLB size

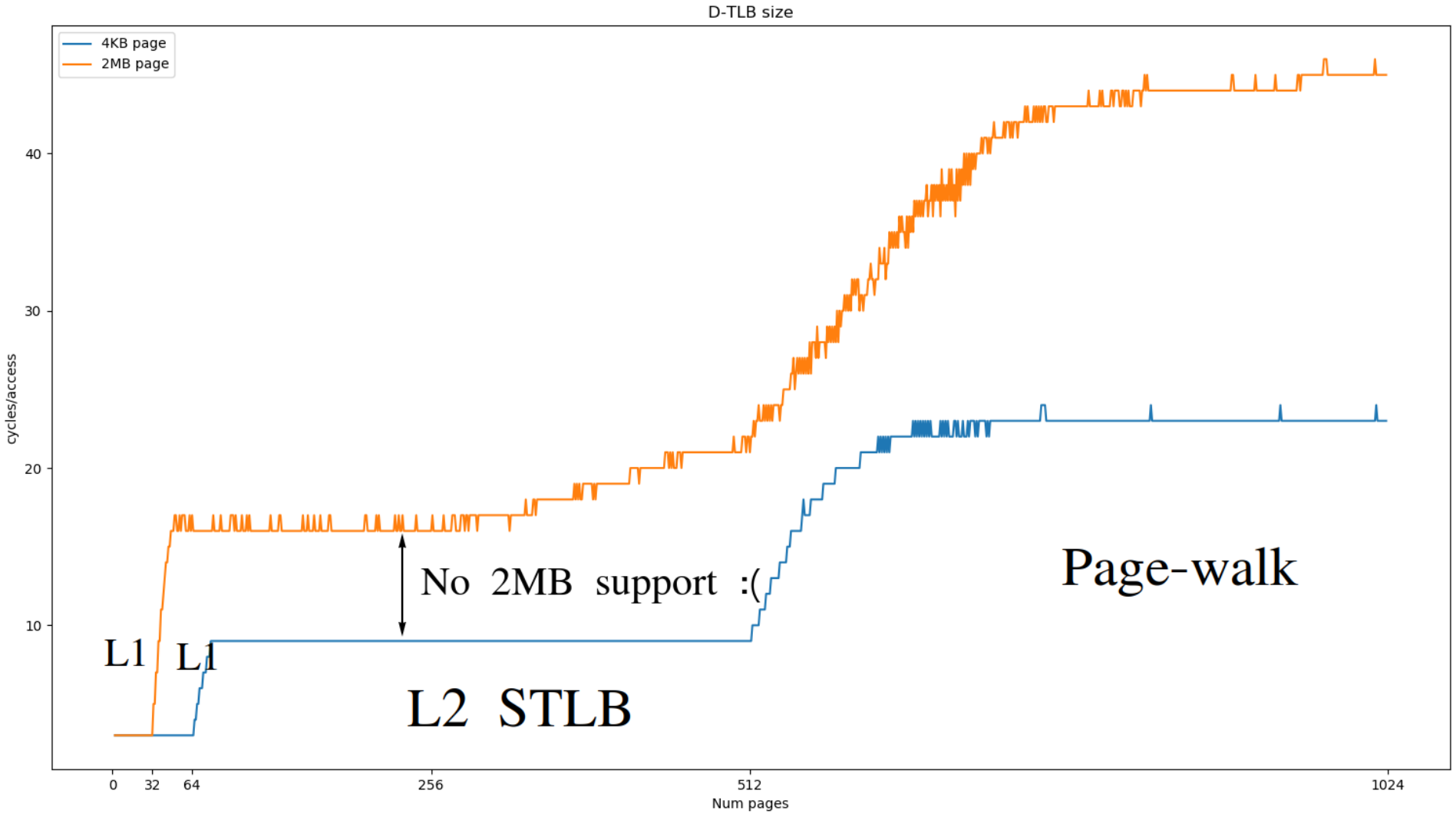
Simple:
Access many different pages but have all of the data in the L1 D/I cache.

4 KB page, 32 KB L1 D-cache, 8 way

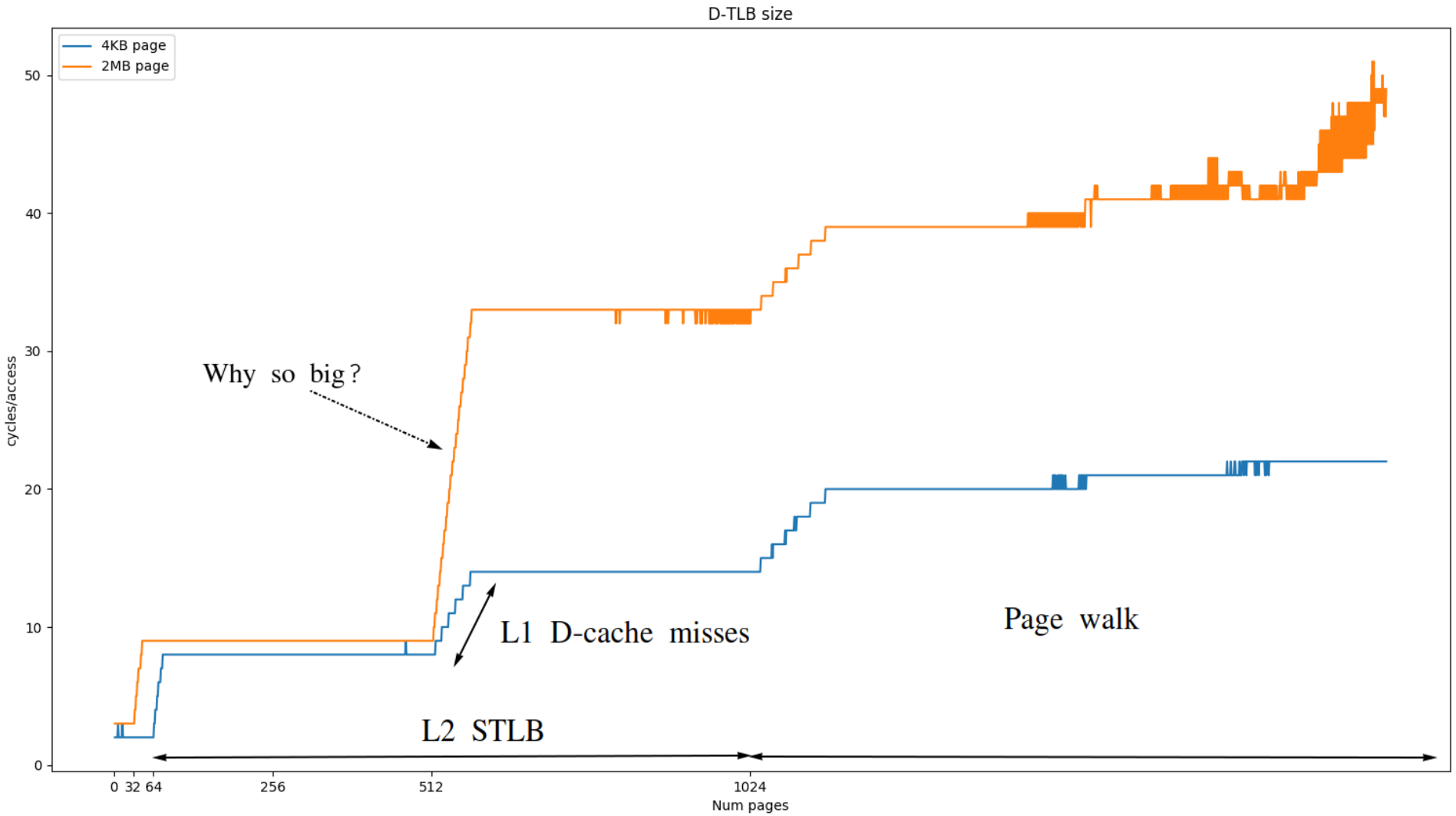
8 Different pages, but the data is all in set 0
0xFFF**0**0**0**0
0xFFF**1**0**0**0
...
0xFFF**7**0**0**0
Next page but now data in set 1:
0xFFF**8**0**4**0

As we increase the working set, we will start thrashing the L1/L2 TLBs

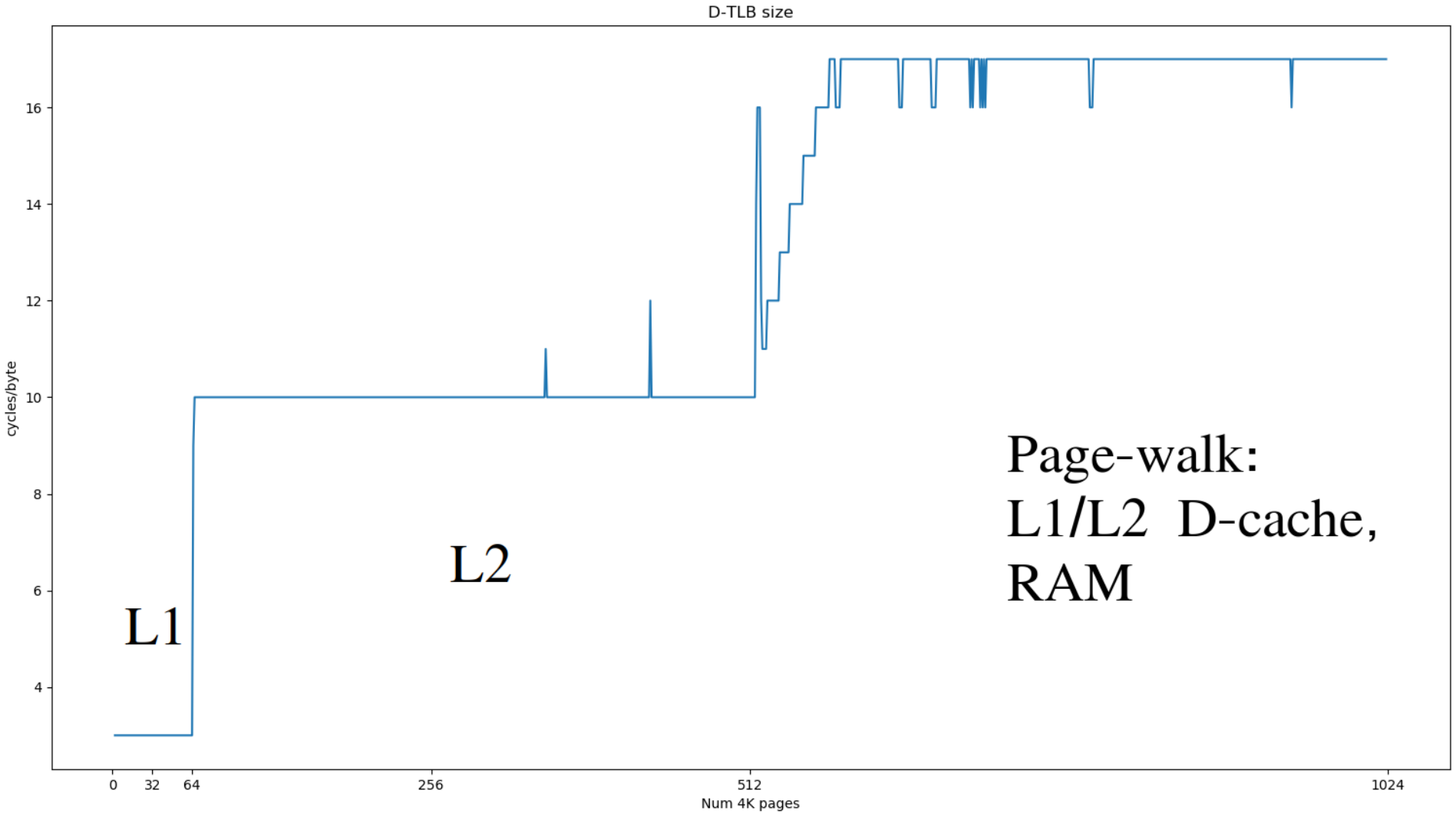
D-TLB - Sandy Bridge



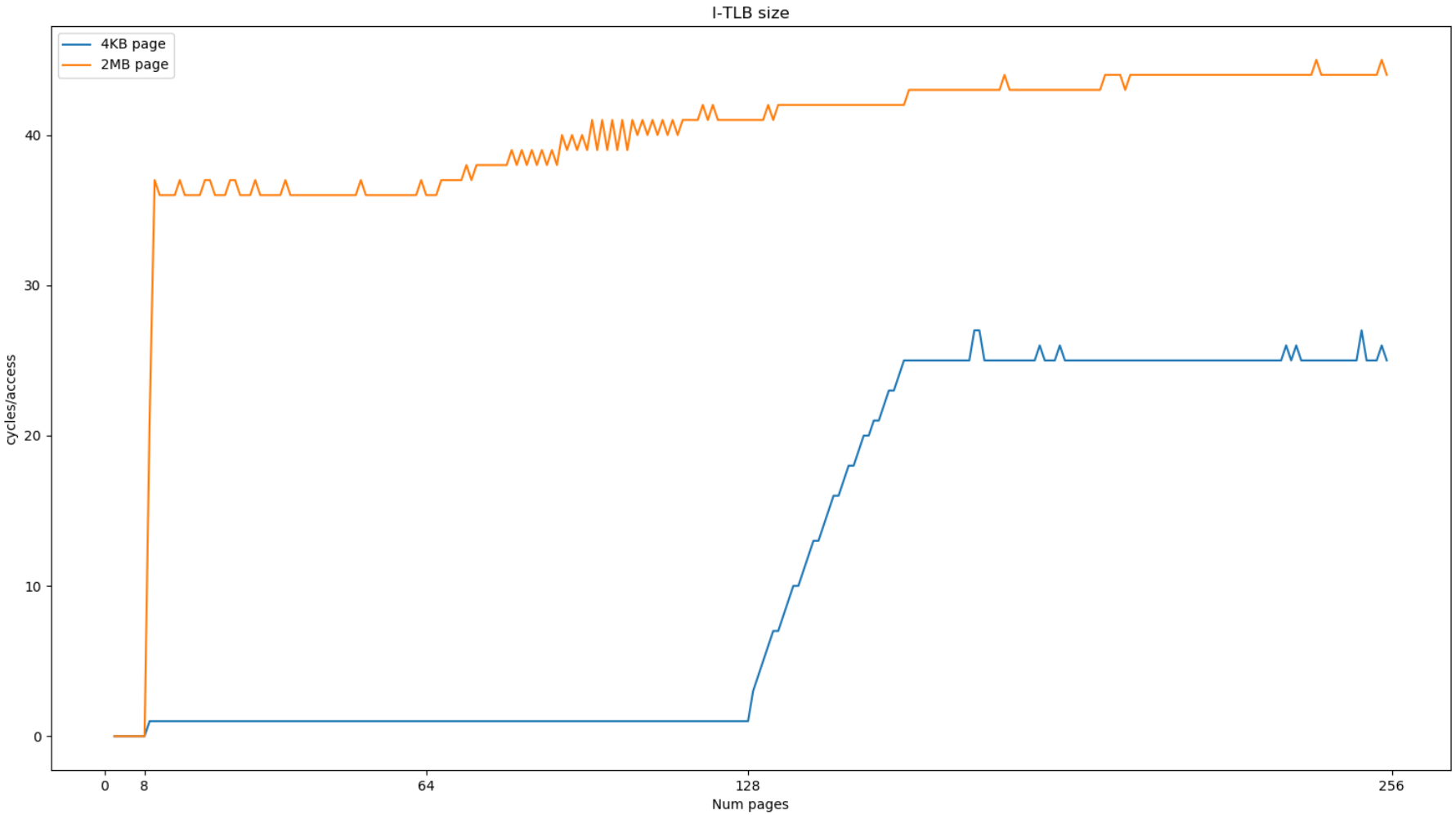
D-TLB - Haswell



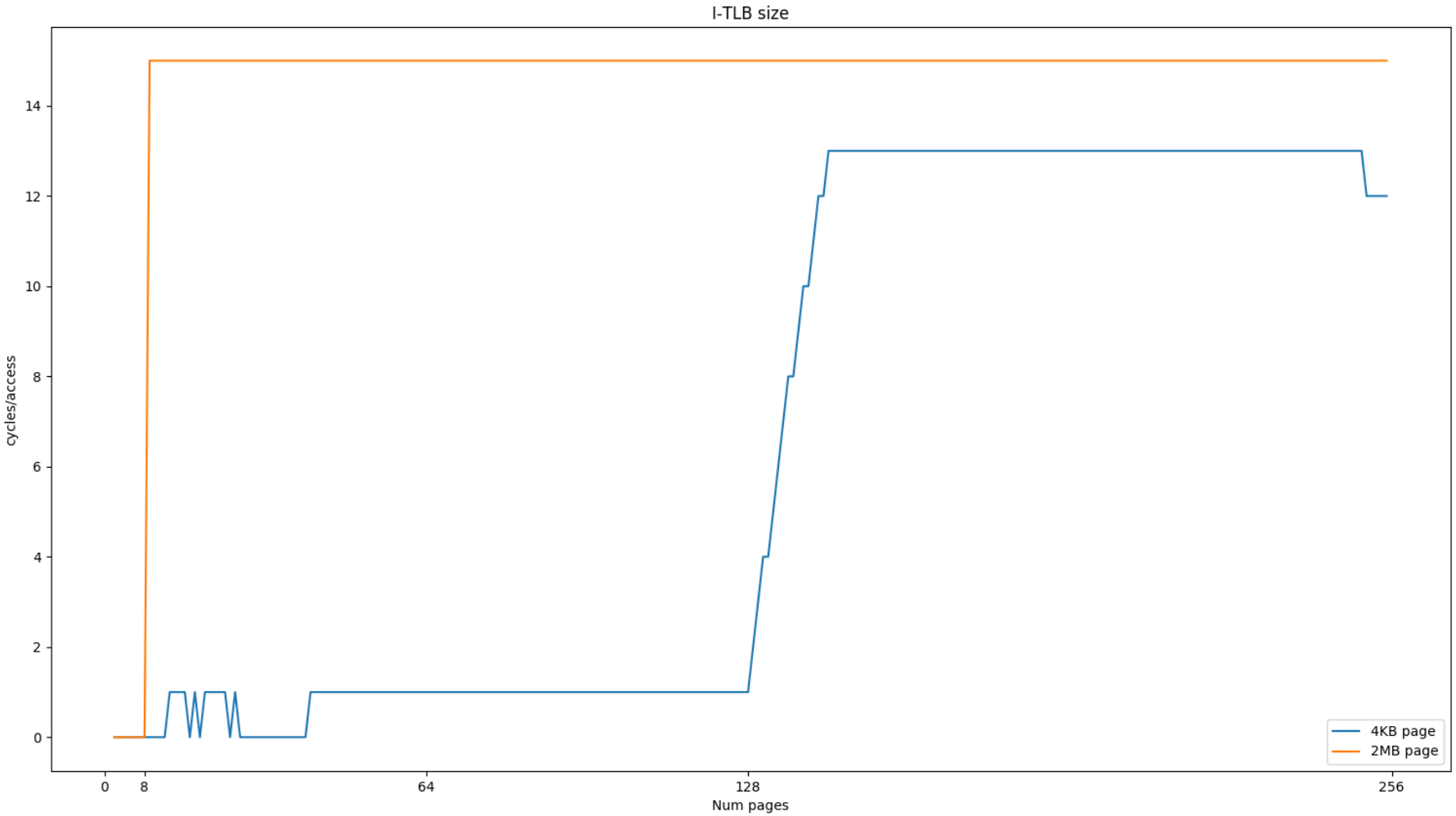
D-TLB - 4 KB page - Ryzen 2700X



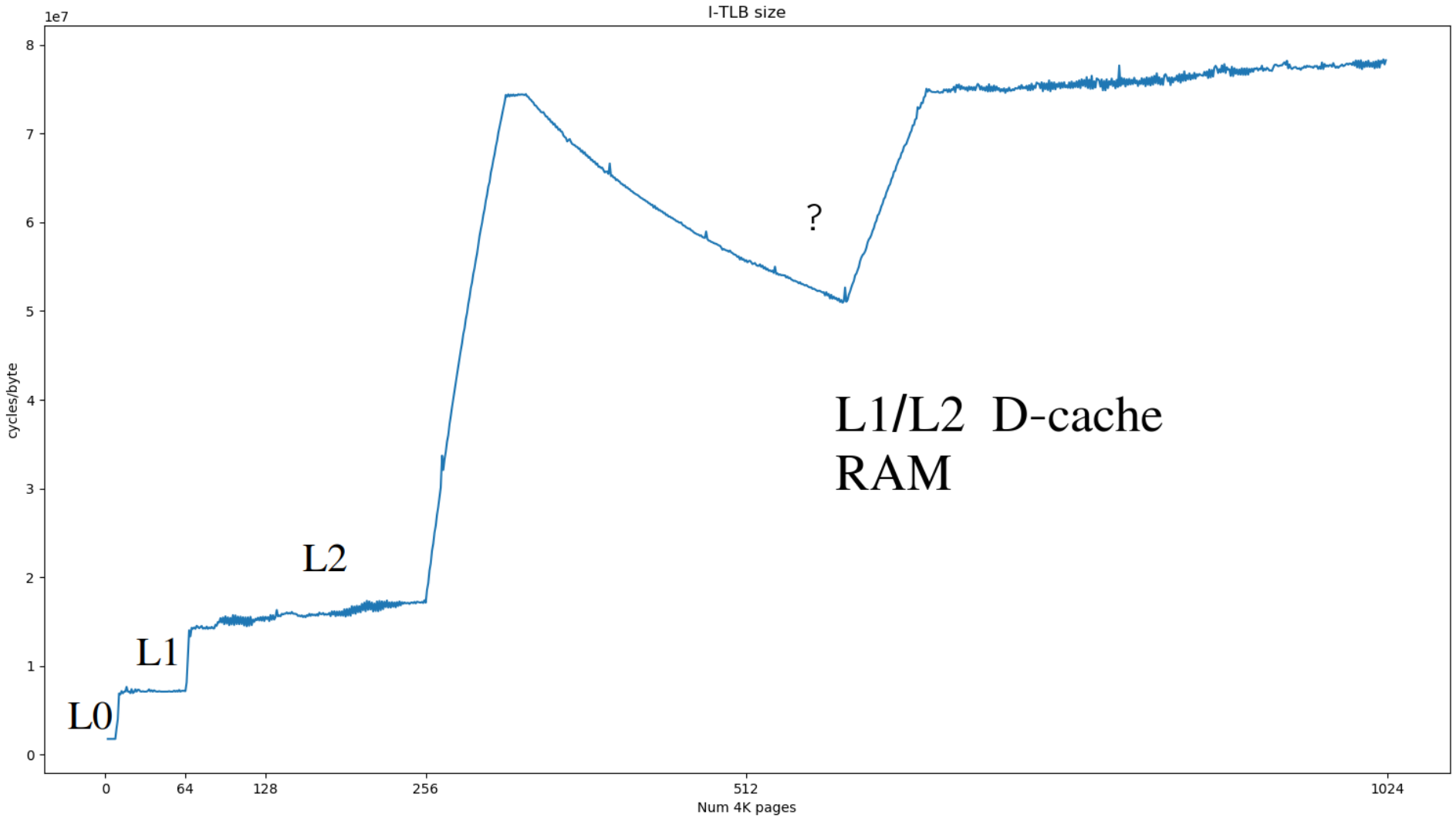
I-TLB - Sandy Bridge



I-TLB - Haswell



I-TLB - 4 KB page - Ryzen 2700X



Final points

- Performance is not only about the I/D caches
- Accessing memory is complex
- That complexity is to stay
- Kernel's management of memory is even more complex
- Many other things to measure for size - associativity, page-walk caches, ...

Thanks for the attention
Questions?