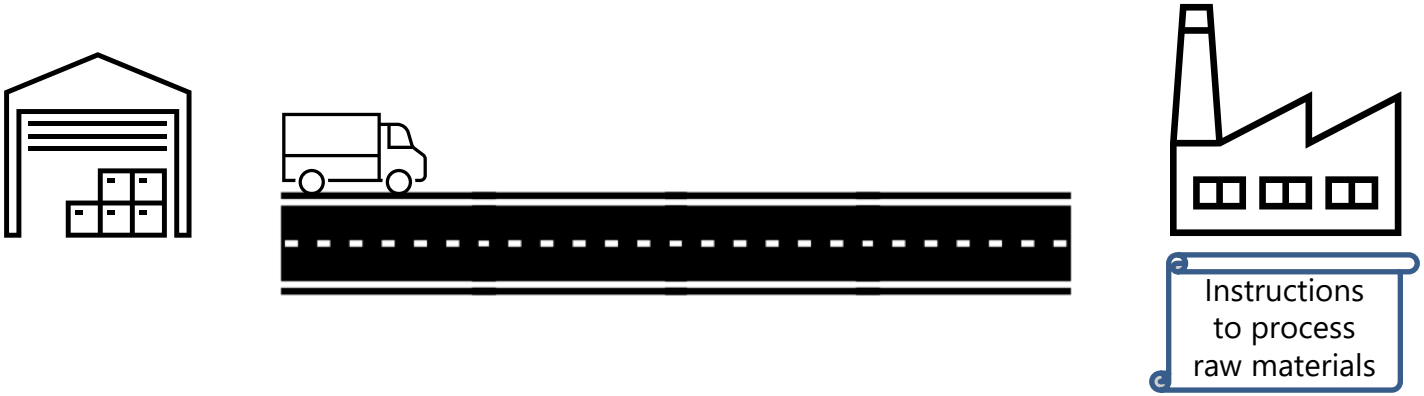
Memory Hierarchy

CS 1541
Wonsun Ahn

University of Pittsburgh

Impact of Memory On Performance

Remember our factory analogy? Memory is Logistics!



- If you can't move goods in a timely manner your output will suffer
- Factory efficiency doesn't matter if logistics is the bottleneck
  - If deliveries are slow (high memory latency)
  - Or do not arrive in sufficient quantities (low memory bandwidth)

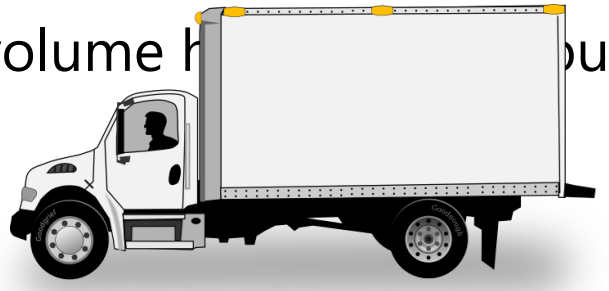
Memory Latency vs. Memory Bandwidth

- Memory bottleneck comes from two sources:
  - **Memory latency**: hours to handle a single delivery

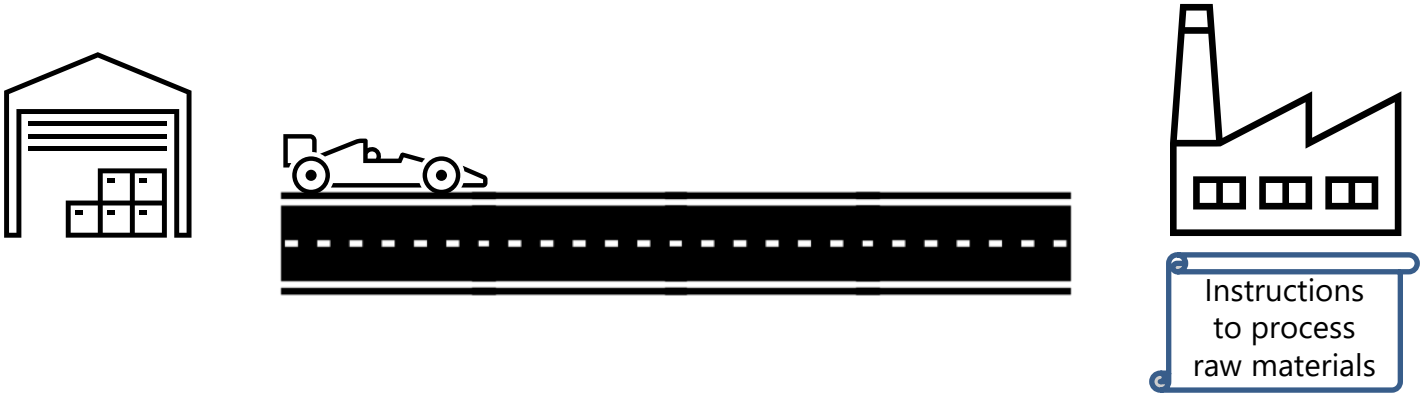
vs.

  - **Memory bandwidth**: maximum volume h…our

vs.



Memory Latency does not put a hard limit on performance

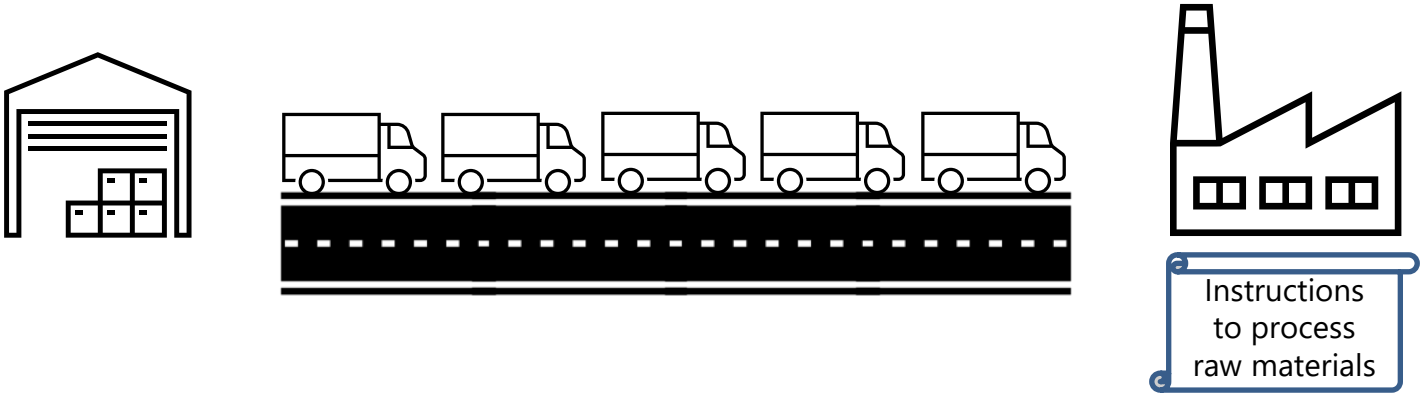


- Race cars are rarely used for logistics. Trucks are. Why?

1. The factory can produce something else while waiting (*scheduling*)
2. Can predict future orders based on past orders and fetch sooner (*prefetching*)

→ Memory Latency does not put a hard limit on performance.

But Memory Bandwidth does put a hard limit on performance



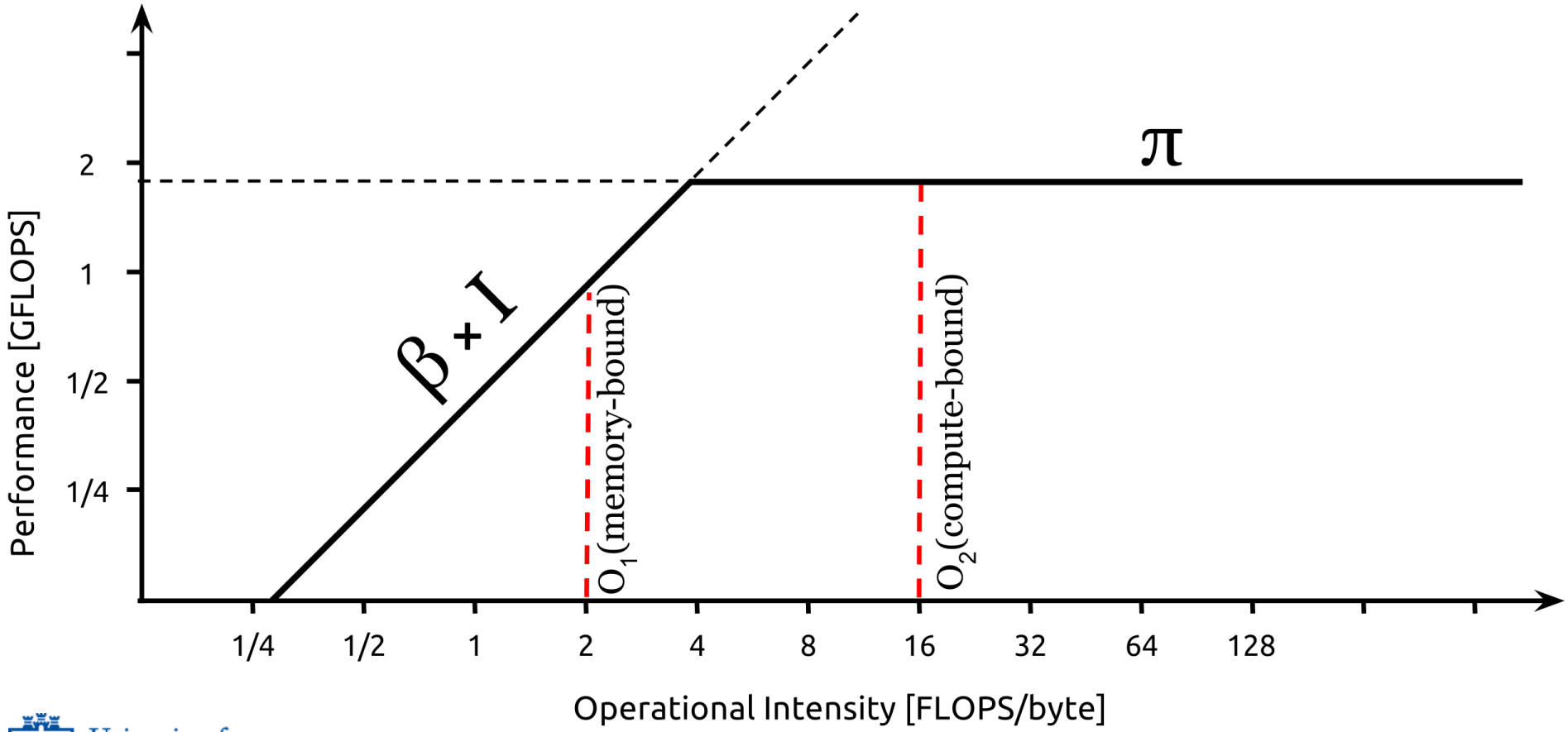
- If the link can only handle only 1 truck / minute, that will dictate throughput.

1. Per minute, factory can only do 1 truck load worth of work
2. Only way to increase amount of work done is to increase **operational intensity**.

→ Memory Bandwidth does put a hard limit on performance.

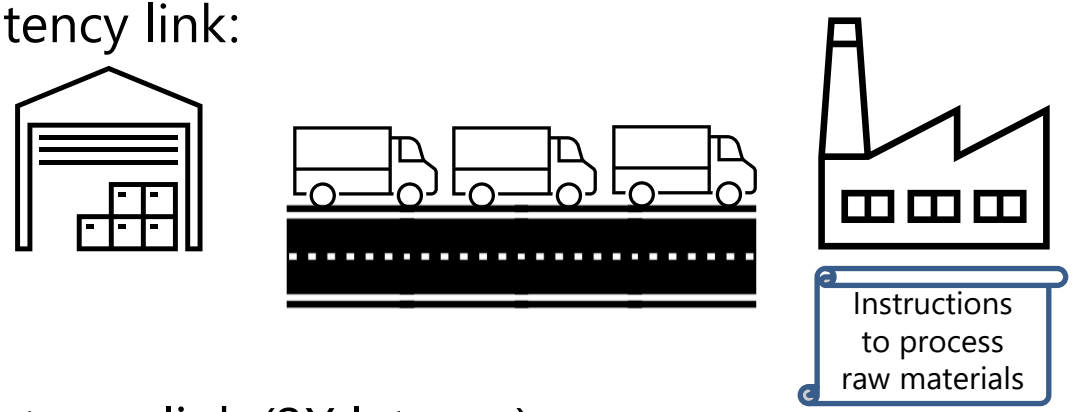
Memory Bandwidth forms left side of the Roofline Model

- I = intensity, β = peak bandwidth, π = peak performance

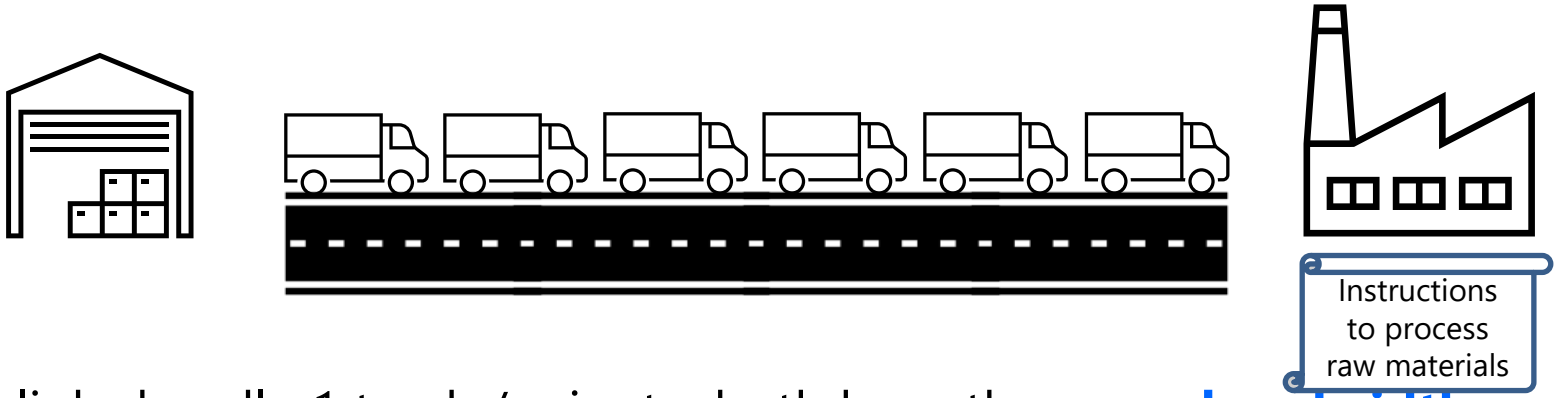


Note: Memory Bandwidth ≠ Memory Latency

- Low latency link:



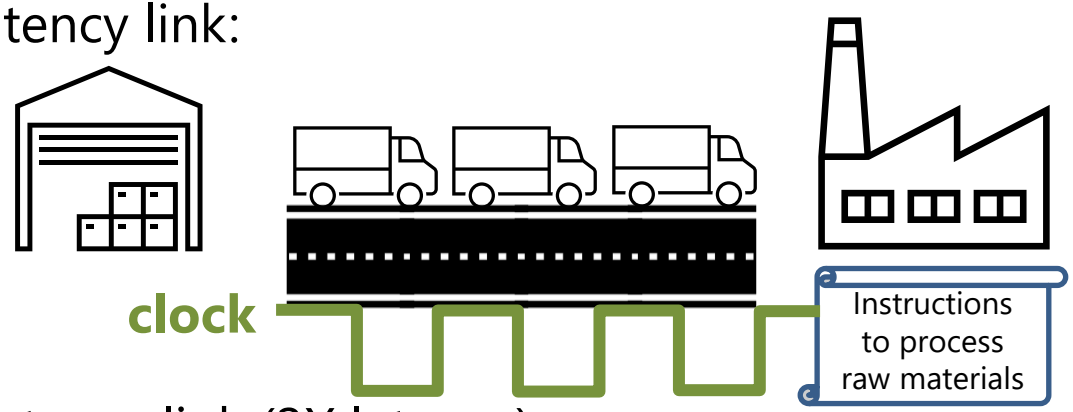
- High latency link (2X latency):



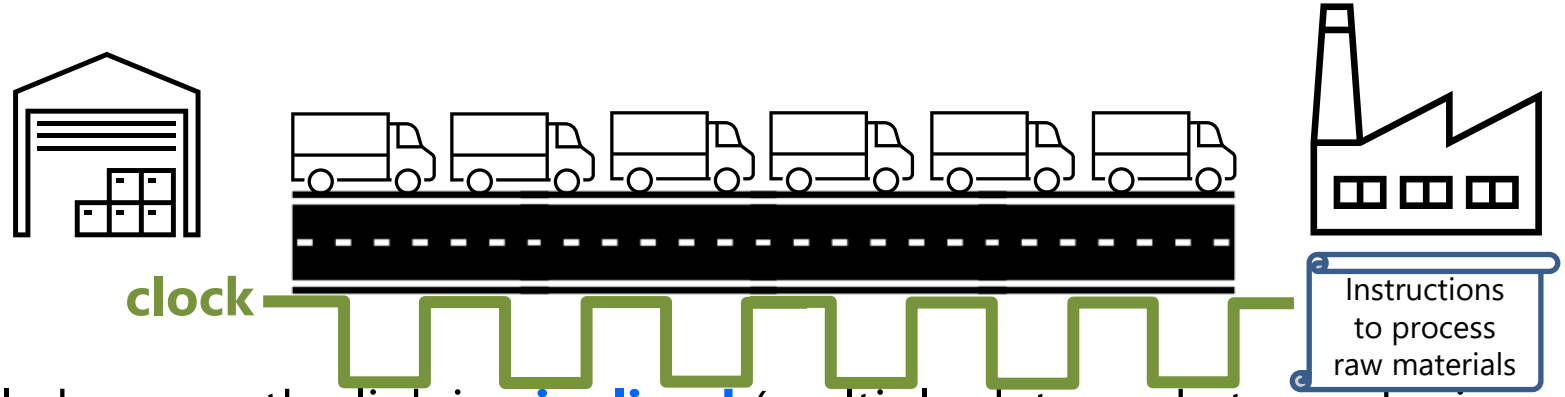
- If both links handle 1 truck / minute, both have the **same bandwidth**.

Bandwidth ≠ Latency, that is given a pipelined link

- Low latency link:



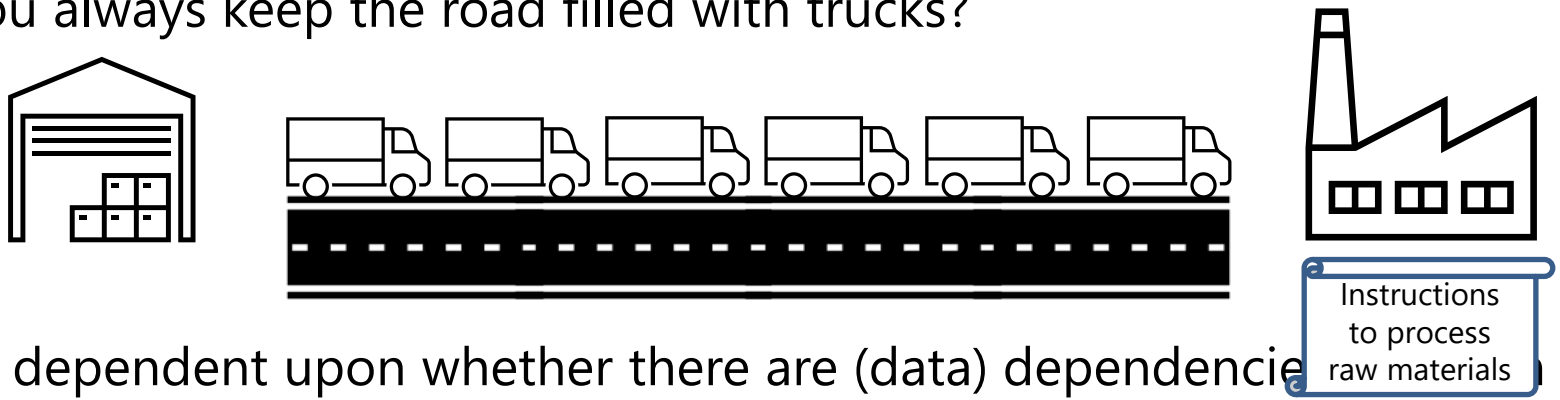
- High latency link (2X latency):



- Possible because the link is **pipelined** (multiple data packets can be in flight).

Keeping the Link Pipeline filled is Contingent Upon MLP

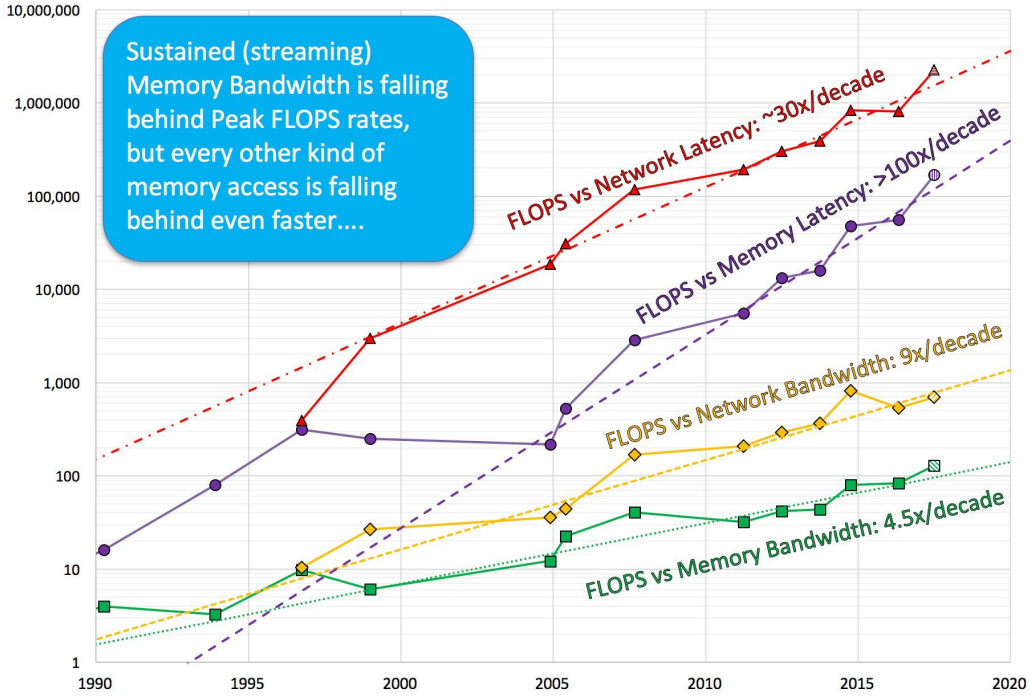
- Can you always keep the road filled with trucks?



- Highly dependent upon whether there are (data) dependencie[illegible] trucks!

- **Memory Level Parallelism** (MLP): degree of concurrency between data accesses
  - Just like ILP allows utilization of wide CPU, MLP allows utilization of wide link.
  - Just like ILP, MLP is an inherent property of the program. (E.g. linked-list traversal → low MLP vs. array traversal → high MLP)
  - Just like ILP, CPU needs a large instruction scheduling window to extract MLP.

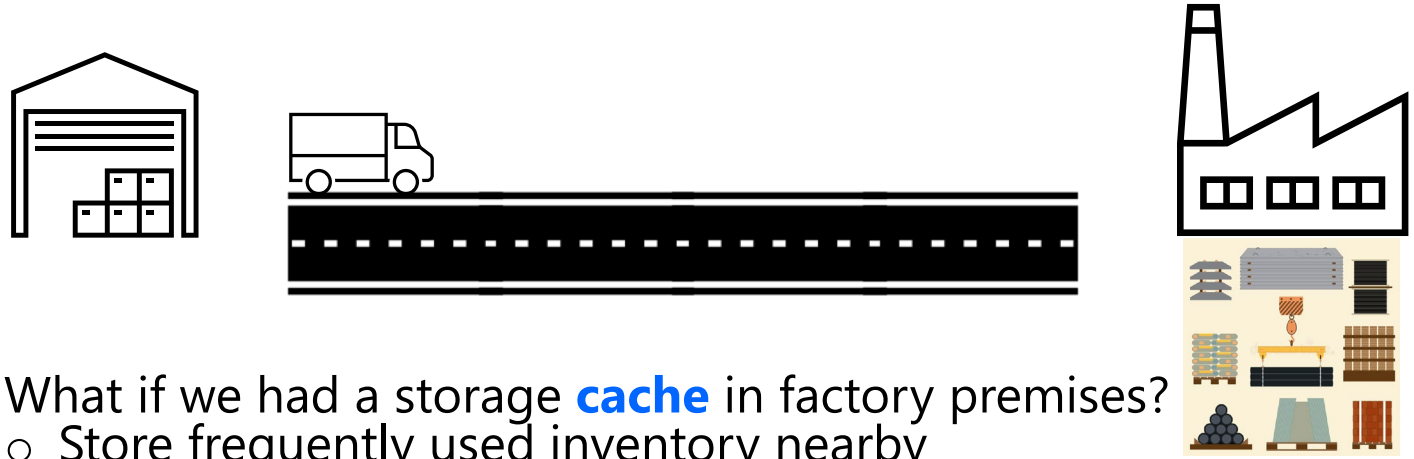
Memory Wall: widening gap between memory and CPU



Source: SC16 Invited Talk ""Memory Bandwidth and System Balance in HPC Systems" by John D. McCalpin

- FLOPS = floating point operations per second (CPU speed)

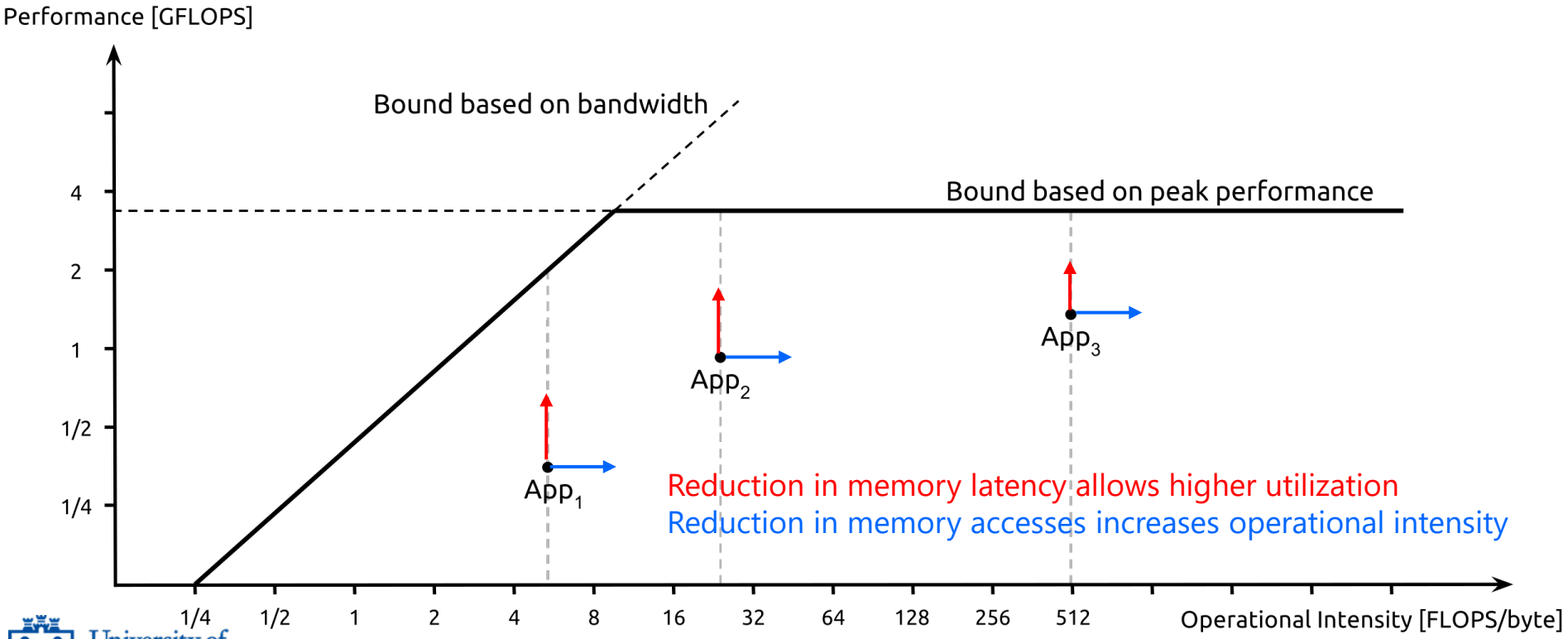
Cache: storage in CPU for frequently accessed data



- What if we had a storage **cache** in factory premises?
  - Store frequently used inventory nearby
- Then, both memory latency and bandwidth problems are alleviated
  - Latency: access is quick when material is found in the cache
  - Bandwidth: less need to go back and forth to the warehouse

Effect of Cache on Roofline Model

- With caching, apps shift upwards and rightwards



CPU Cycles = CPU Compute Cycles + Memory Stall Cycles

- **CPU Cycles = CPU Compute Cycles + Memory Stall Cycles**
  - CPU Compute Cycles = cycles where CPU is not stalled on memory
  - Memory Stall Cycles = cycles where CPU is waiting for memory

- HW/SW ways to improve **CPU Compute Cycles**:
  - HW: Pipelining, branch prediction, wide execution, out-of-order
  - SW: Redundant instruction elimination, scheduling, increasing ILP

- HW/SW ways to improve **Memory Stall Cycles**:
  - HW: Caches, write buffer, prefetcher (we haven't learned these yet)
  - SW: Optimizing memory access pattern to better use caches

Impact of overclocking (DVFS) on Memory Stall Time

- CPU Time = CPU Cycles * Cycle Time
  = CPU Compute Cycles * Cycle Time + Memory Stall Cycles * Cycle Time

- What if we halved the Cycle Time using DVFS?

- Impact on CPU Compute Cycles * Cycle Time:
  - CPU Compute Cycles remains constant (no change in bubbles)
  - CPU Compute Time is also reduced to half!

- Impact on Memory Stall Cycles * Cycle Time:
  - Memory Stall Cycles can increase by up to 2X (if cache always misses)
  - Why? DRAM speed remains same → Need twice the cycles per access
    - The bus (wire) that connects CPU to DRAM is not getting any faster
    - The DRAM chip itself is not getting any faster
  - End result may be there is no improvement in performance

Using PMUs to Understand Performance

University of Pittsburgh

Experiment on kernighan.cs.pitt.edu

- The source code for the experiments are available at: https://github.com/wonsunahn/CS1541_Spring2024/tree/main/resources/cache_experiments

- Or by copying the following directory at linux.cs.pitt.edu: /afs/cs.pitt.edu/courses/1541/cache_experiments/

- You can run the experiments by doing 'make' at the root
  - It will take a few minutes to run all the experiments
  - In the end, you get two plots: IPC.pdf and MemStalls.pdf

Four benchmarks

- linked-list.c
  - Traverses a linked list from beginning to end over and over again
  - Each node has 120 bytes of data

- array.c
  - Traverses an array from beginning to end over and over again
  - Each element has 120 bytes of data

- linked-list_nodata.c
  - Same as linked-list but nodes have no data inside them

- array_nodata.c
  - Same as array but elements have no data inside them

Code for linked-list.c

```
// Define a linked list node type with data
typedef struct node {
  struct node* next;    // 8 bytes
  int data[30];         // 120 bytes
} node_t;
...
// Create a linked list of length items
void *create(void *unused) {
  for(int i=0; i<items; i++) {
    node_t* n = (node_t*)malloc(sizeof(node_t));
    if(last == NULL) {  // Is the list empty? If so, the new node is the head and tail
      head = n;
      last = n;
    } else {
      last->next = n;
      last = n;
    }
  }
}
```

Code for linked-list.c

```
#define ACCESSES 1000000000

// MEASUREMENT BEGIN
// Traverse list over and over until we've visited `ACCESSES` nodes
node_t* current = head;
for(int i=0; i<ACCESSES; i++) {
  if(current == NULL) current = head;      // reached the end
  else current = current->next;            // next node
}
// MEASUREMENT END
```

- Note: executed instructions are equivalent regardless of list length
- So we expect performance to be same regardless of length. Is it?

Code for array.c

```
// Define a linked list node type with data
typedef struct node {
  struct node* next;    // 8 bytes
  int data[30];         // 120 bytes
} node_t;
...
// Create a linked list but allocate nodes in an array
void *create(void *unused) {
  head = (node_t *) malloc(sizeof(node_t) * items);
  last = head + items - 1;
  for(int i=0; i<items; i++) {
    node_t* n = &head[items];
    n->next = &head[items+1]; // Next node is next element in array
  }
  last->next = NULL;
}
```

Code for array.c

```
#define ACCESSES 1000000000

// MEASUREMENT BEGIN
// Traverse list over and over until we've visited `ACCESSES` nodes
node_t* current = head;
for(int i=0; i<ACCESSES; i++) {
  if(current == NULL) current = head;      // reached the end
  else current = current->next;            // next node
}
// MEASUREMENT END
```

- Note: same exact loop as the linked-list.c loop.
- So we expect performance to be exactly the same. Is it?

kernighan.cs.pitt.edu specs

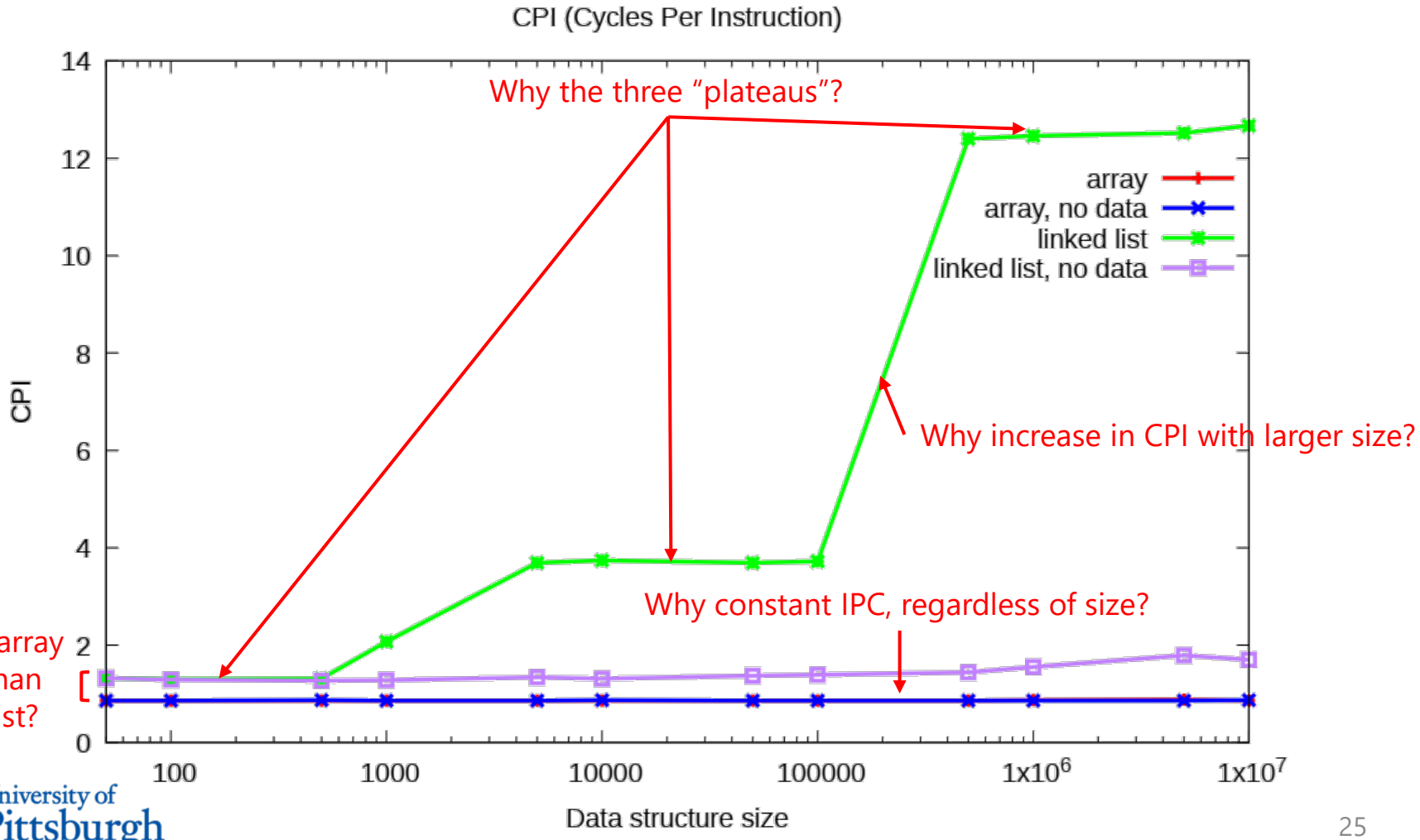
- Two CPU sockets. Each CPU:
  - Intel(R) Xeon(R) CPU E5-2640 v4
  - 10 cores, with 2 threads per each core (SMT)
  - L1 i-cache: 32 KB 8-way set associative (per core)
  - L1 d-cache: 32 KB 8-way set associative (per core)
  - L2 cache: 256 KB 8-way set associative (per core)
  - L3 cache: 25 MB 20-way set associative (shared)
- Memory
  - 128 GB DRAM
- Information obtained from
  - "cat /proc/cpuinfo" on Linux server
  - "cat /proc/meminfo" on Linux server
  - https://en.wikichip.org/wiki/intel/xeon_e5/e5-2640_v4

Experimental data collection

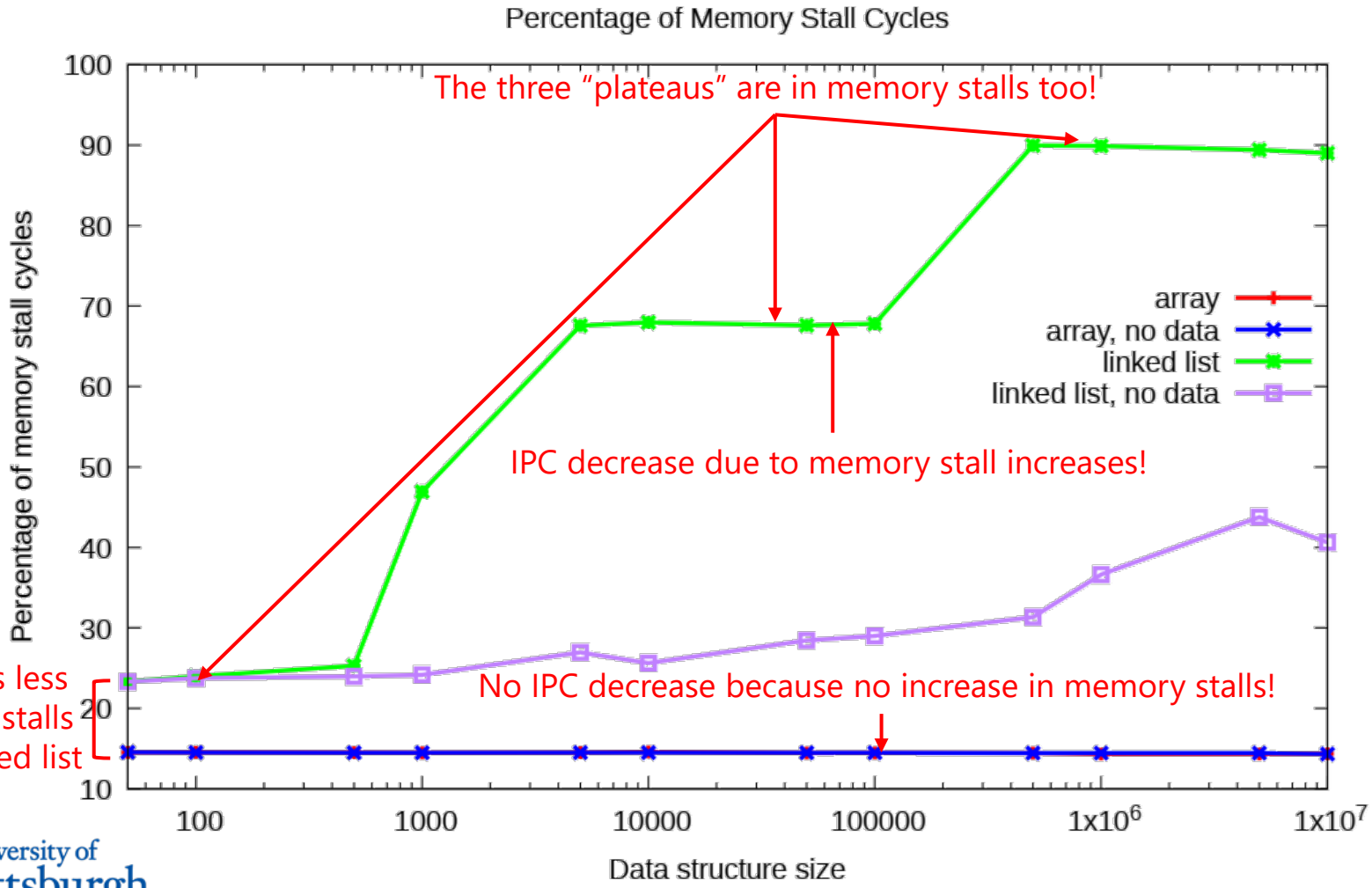
- Collected using CPU **Performance Monitoring Unit (PMU)**
  - PMU provides performance counters for a lot of things
  - Cycles, instructions, various types of stalls, branch mispredictions, cache misses, bandwidth usage, …

- Linux **perf** utility summarizes this info in easy to read format
  - https://perf.wiki.kernel.org/index.php/Tutorial

CPI (Cycles Per Instruction) Results

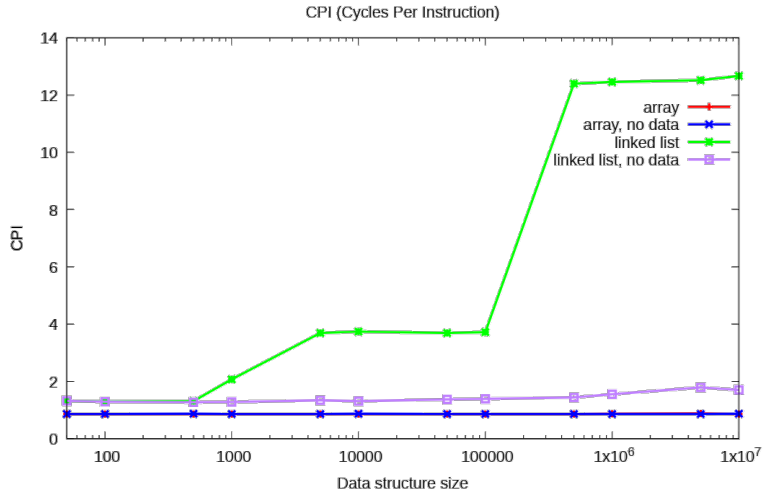


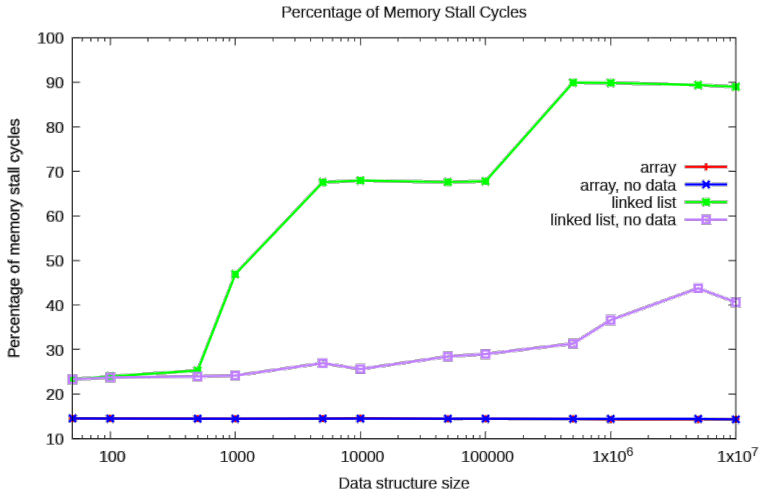
Memory Stall Cycle Percentage



University of Pittsburgh

Data Structure Performance ∝ Memory Stalls



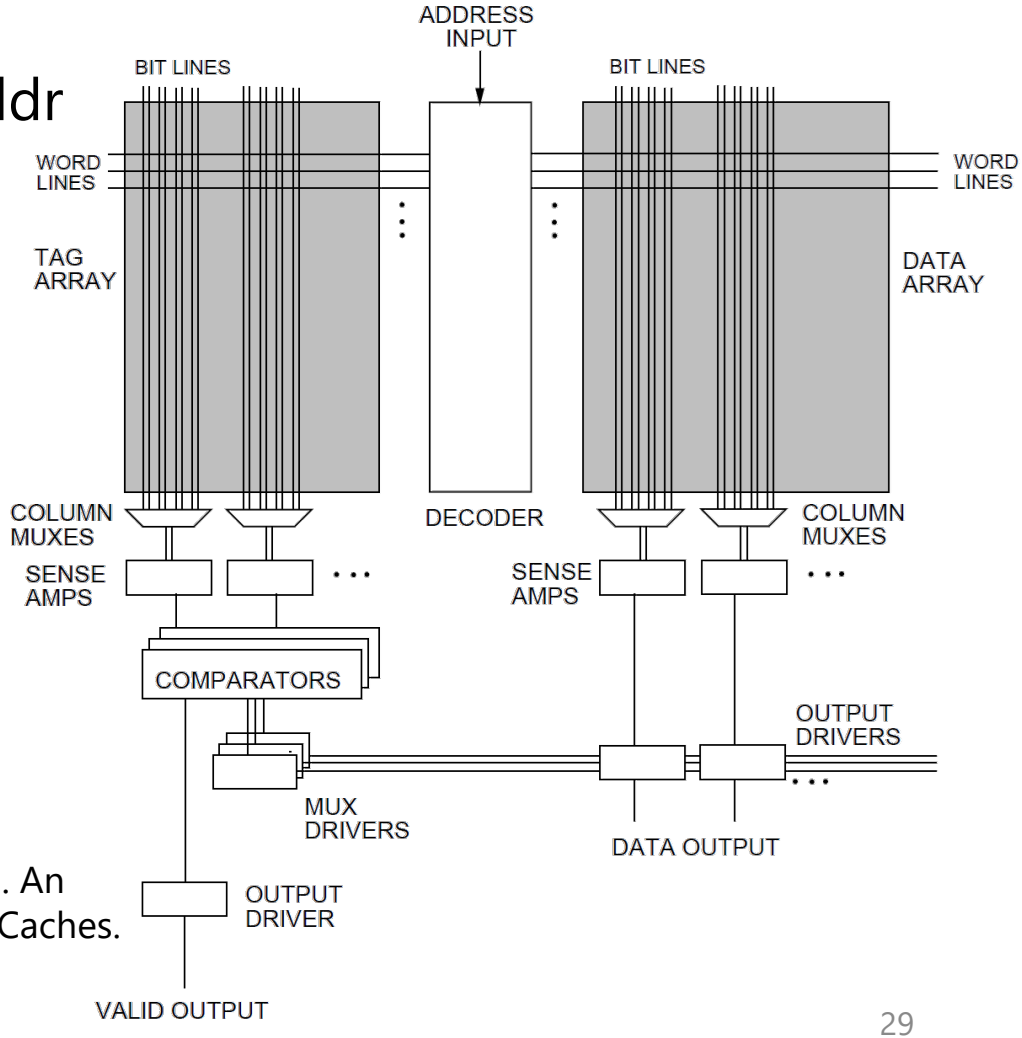


- **Data structure performance is proportional to memory stalls**
  - Applies to other data structures such as trees, graphs, …
- In general, **more data** leads to **worse performance**
  - But why? Does more data make MEM stalls longer? (Hint: yes)
  - And why is an array not affected by data size? (I wonder …)
- You will be able to answer all these questions when we are done.

Memory Technologies

Memory Structure

- Memory is a 2D array
  - Word lines select row from addr
  - Bit lines read out bits of word
- Each row contains
  - Tag bits (when needed)
    - Left shaded square
    - Resolves hash conflicts
  - Data bits
    - Right shaded square

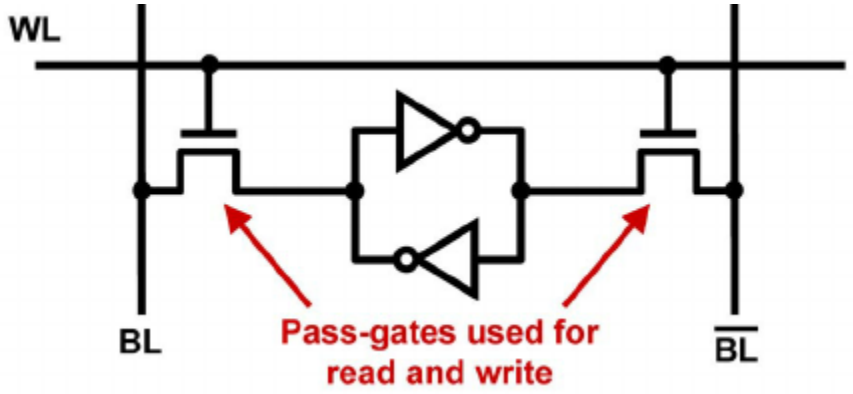


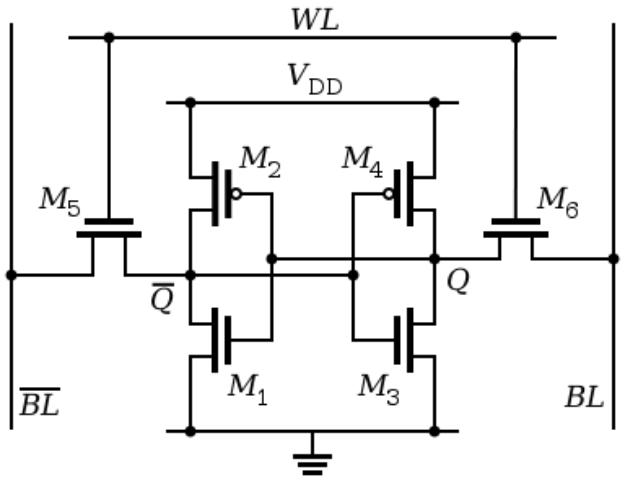
Courtesy: Winton, Steven and Jouppi, Norman. (1994). An Enhanced Access and Cycle Time Model for On-Chip Caches.

Static RAM (SRAM)

- SRAM uses a loop of NOT gates
  - In order to store a single bit
- This is called a 6T SRAM cell
  - Because it uses… **6 transistors**!

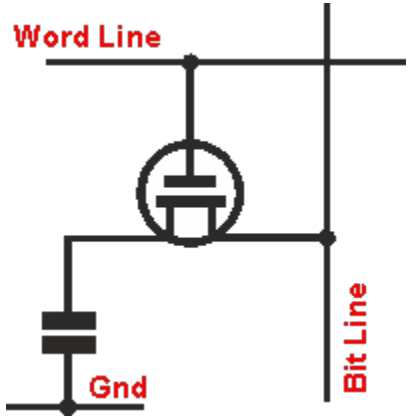
- **Pros:**
  - **Very fast** to read/write
- **Cons:**
  - Volatile (loses data without power)
  - Relatively many transistors needed
    -> **expensive**





Dynamic RAM (DRAM)

- DRAM uses **one** transistor and **one** capacitor
  - The bit is stored as a charge in the capacitor
  - Capacitor leaks charge over time
    -> Must be periodically recharged (called **refresh**)
    -> During refresh, DRAM can't be accessed
  - Accesses are slower
    -> Small charge must be amplified to be read
    -> Also after read, capacitor needs recharging again
  - Reading a DRAM cell is slower than reading SRAM



- **Pros:**
  - Higher density -> less silicon -> **much cheaper than SRAM**
- **Cons:**
  - Still volatile (even more volatile than SRAM)
  - **Slower access time**

Other technology

- Flash Memory
  - Works using a special MOSFET with “floating gate”
  - **Pros**: nonvolatile, much faster than HDD
  - **Cons**:
    - Slower than DRAM
    - More expensive than HDDs (1TB for $250)
    - Writing is destructive and shortens lifespan

- Experimental technology
  - Ferroelectric RAM (FeRAM), Magnetoresistive RAM (MRAM), Phase-change memory (PRAM), carbon nanotubes …
  - In varying states of development and maturity
  - Nonvolatile *and* close to DRAM speeds

Spinning magnetic disks (HDD)

- Spinning platter coated with a ferromagnetic substance magnetized to represent bits
  - Has a mechanical arm with a head
  - Reads by placing arm in correct cylinder, and waiting for platter to rotate

- **Pros:**
  - **Nonvolatile** (magnetization persists without power)
  - Extremely cheap (1TB for $50)
- **Cons:**
  - **Extremely slow** (it has a mechanical arm, enough said)

1. There is a trade-off between speed and cost

	Volatile		Nonvolatile	
	SRAM	DRAM	Flash	HDD
Speed	FAST	Pretty fast	OK	SLOW
Price	Expensive	OK	Pretty cheap	Cheap

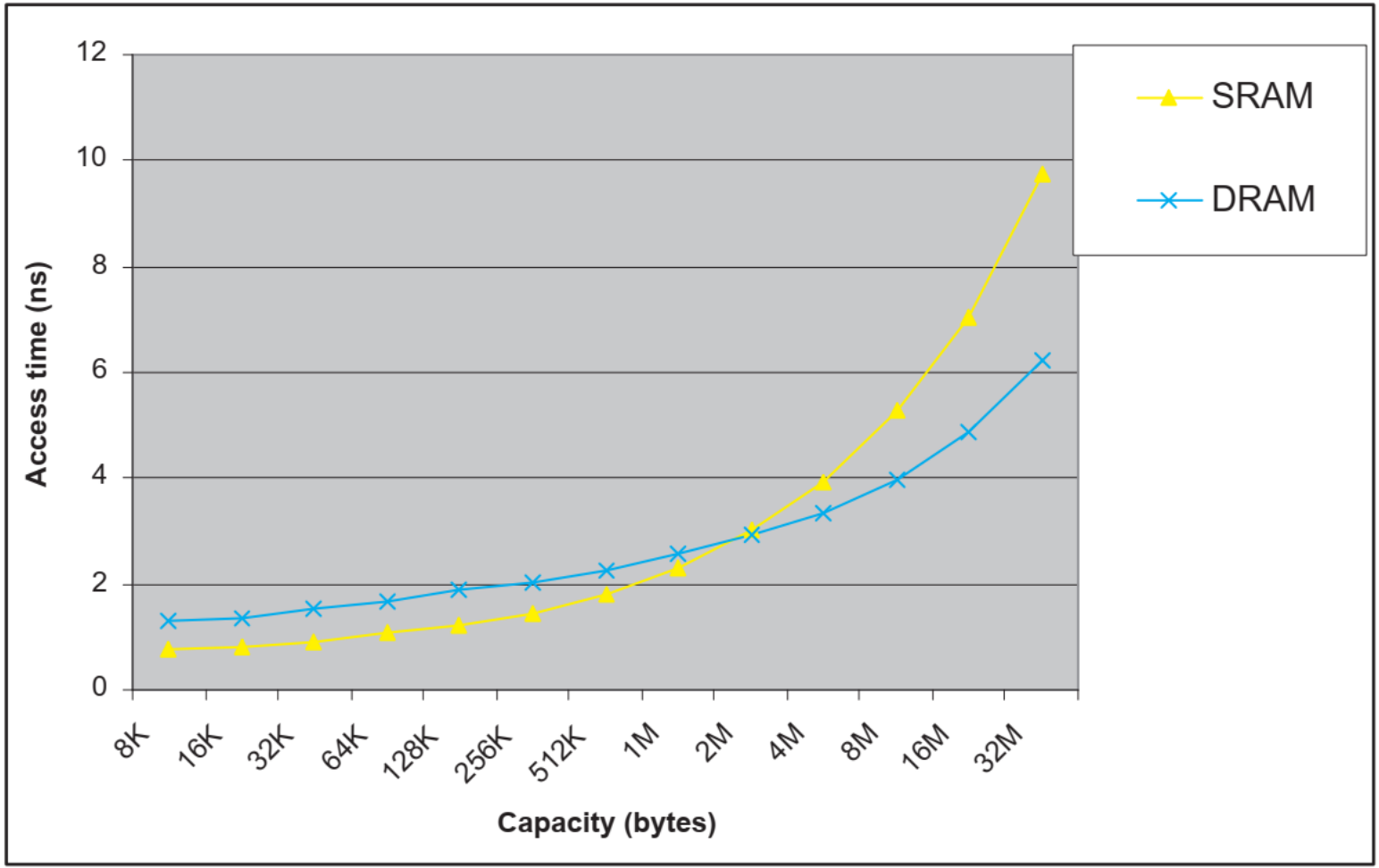
SLOWER!

CHEAPER!

- Slower memory tends to be cheaper.
- Faster memory tends to be more expensive.

- What kind of memory should we use for **caches**?

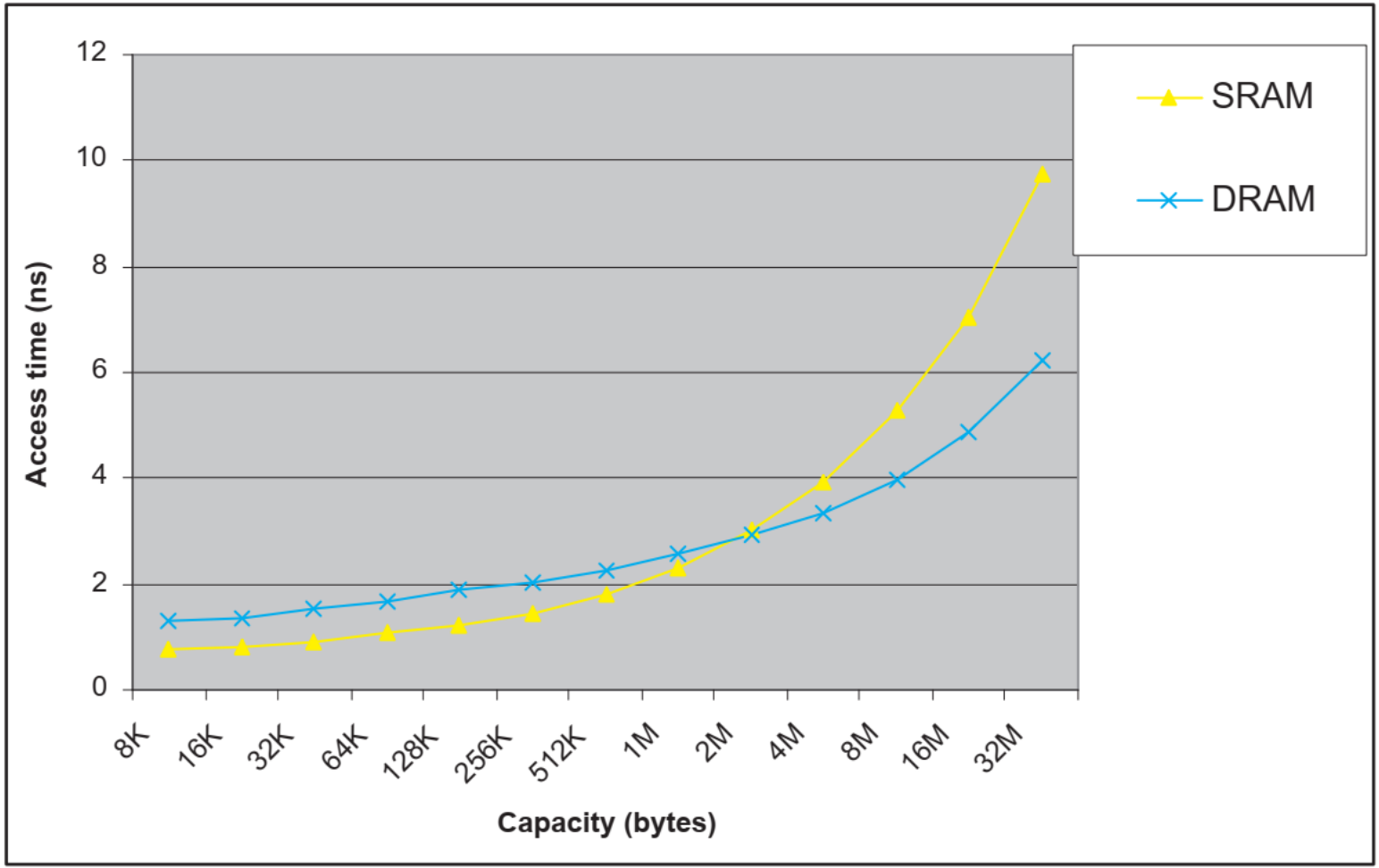
2. With same memory technology, larger is slower



University of Pittsburgh

Thoziyoor, Shyamkumar & Muralimanohar, Naveen & Ahn, Jung Ho & Jouppi, Norman. (2008). CACTI 5.1.

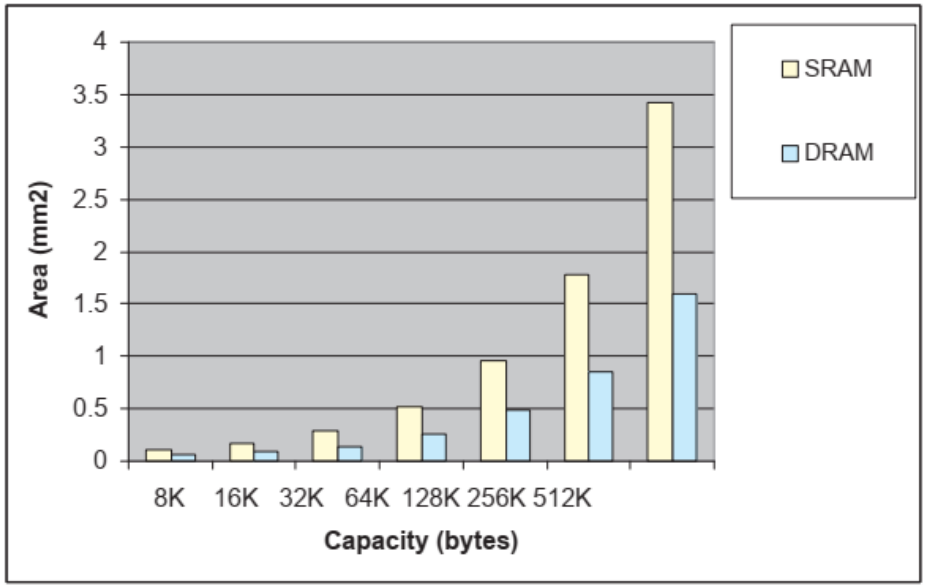
3. DRAM is faster than SRAM at high capacity

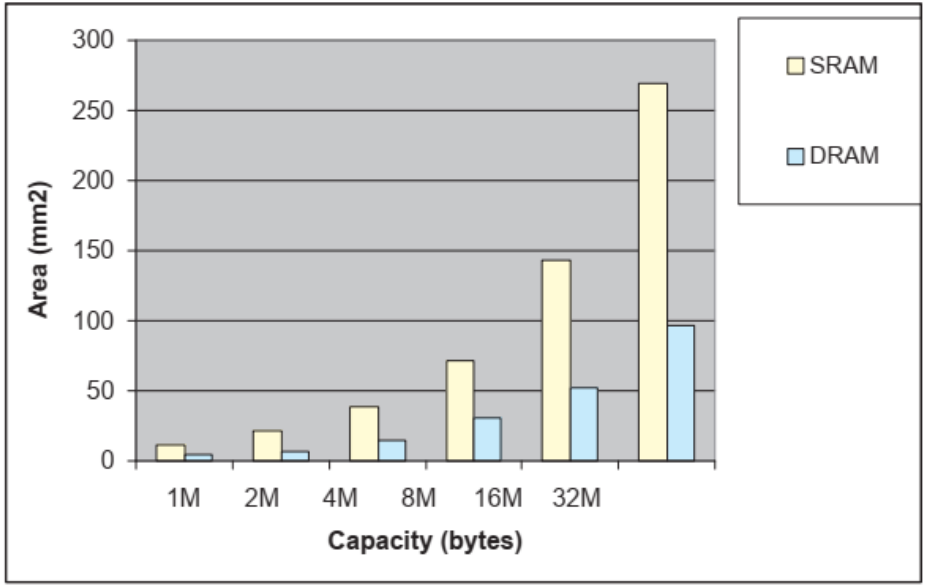


University of Pittsburgh

Thoziyoor, Shyamkumar & Muralimanohar, Naveen & Ahn, Jung Ho & Jouppi, Norman. (2008). CACTI 5.1.

Due to wire delay dominating at high capacities



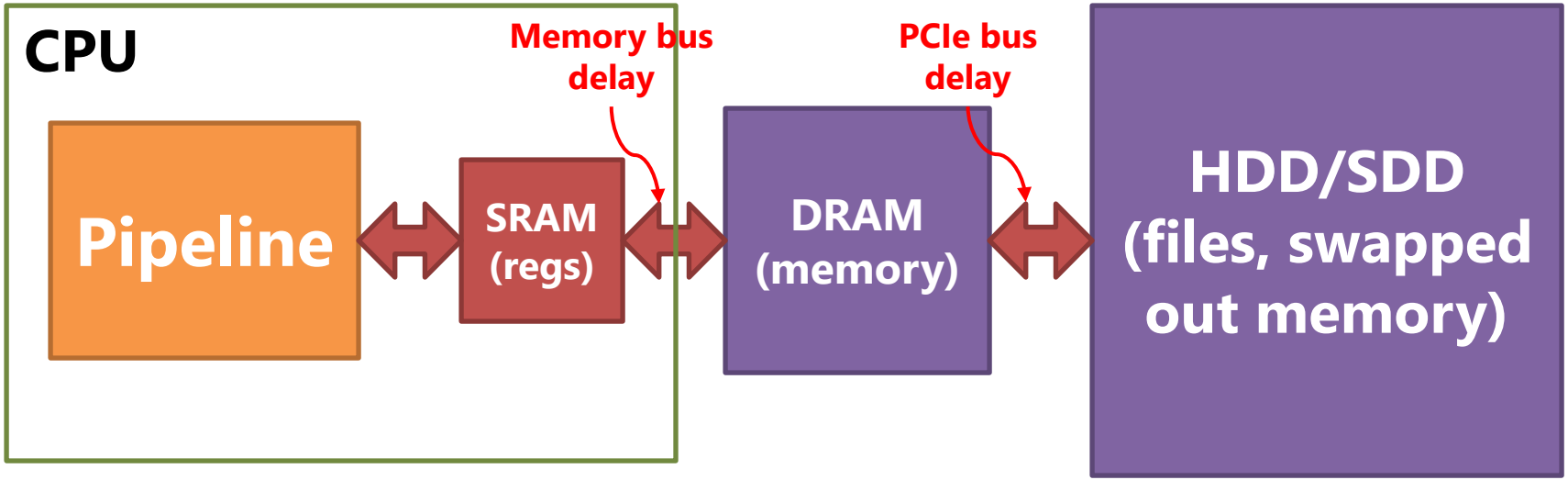


- Access Time = Memory Cell Delay + Address Decode Delay + **Wire Delay**
- With high capacity, Wire Delay (word lines + bit lines) starts to dominate
- Wire Delay ∝ Wire Length ∝ Memory Structure Dimensions
- DRAM density > SRAM density → DRAM Wire Delay < SRAM Wire Delay

The Memory Hierarchy

System Memory Hierarchy

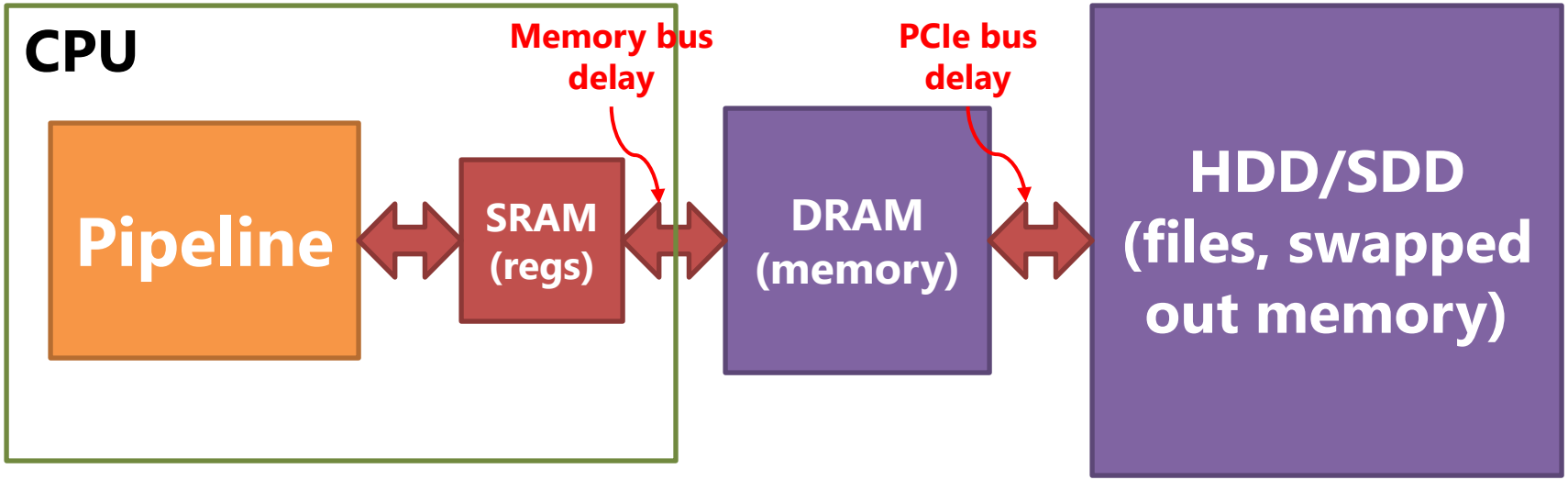
- Use small fast memory to store frequently used data
- Use big slow memory to store infrequently used data



- Registers: used frequently so stored in small SRAM
- Memory pages used frequently: stored in big DRAM
- Memory pages used infrequently: stored in HDD/SDD (in swap space)
- Note: Memories outside CPU chip suffers from bus delay as well

System Memory Hierarchy

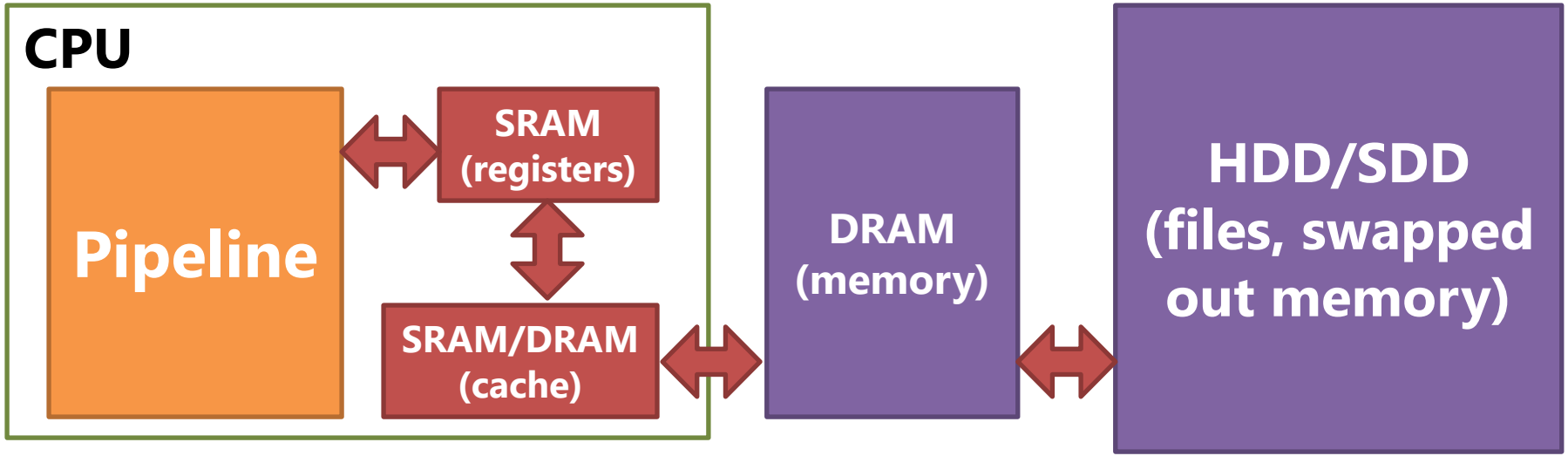
- Use small fast memory to store frequently used data
- Use big slow memory to store infrequently used data



- Memory Wall: memory access is much slower compared to registers
- Q: Can we make memory access speed comparable to register access?

System Memory Hierarchy

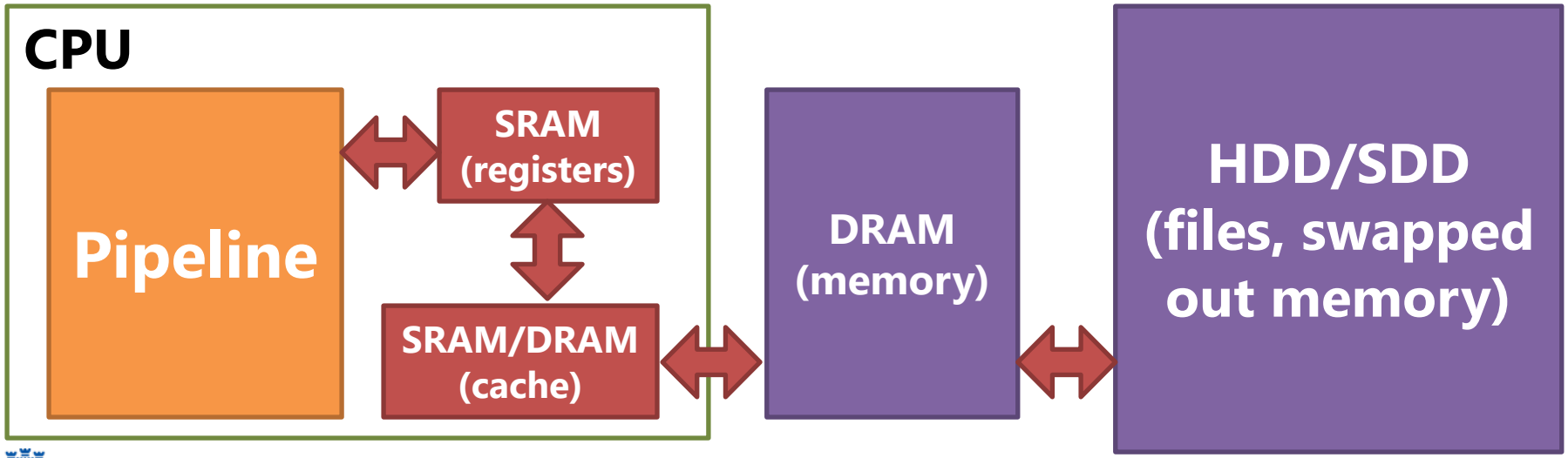
- Use small fast memory to store frequently used data
- Use big slow memory to store infrequently used data



- Memory Wall: Memory access is much slower compared to registers
- Q: Can we make memory access speed comparable to register access?
  - **Cache**: small, fast on-chip memory for frequently used data
  - Caches can be either SRAM or DRAM depending on its size.

Caching happens at multiple levels of the hierarchy

- **Caching**: keeping a temporary copy of data for faster access
- DRAM is in a sense also caching frequently used pages from swap space
  - We are just extending that idea to bring cache data inside the CPU!
- Now instructions like `lw` or `sw` never directly access DRAM
  - They first search the cache to see if there is a **hit** in the cache
  - Only if they **miss** will they access DRAM to bring data into the cache

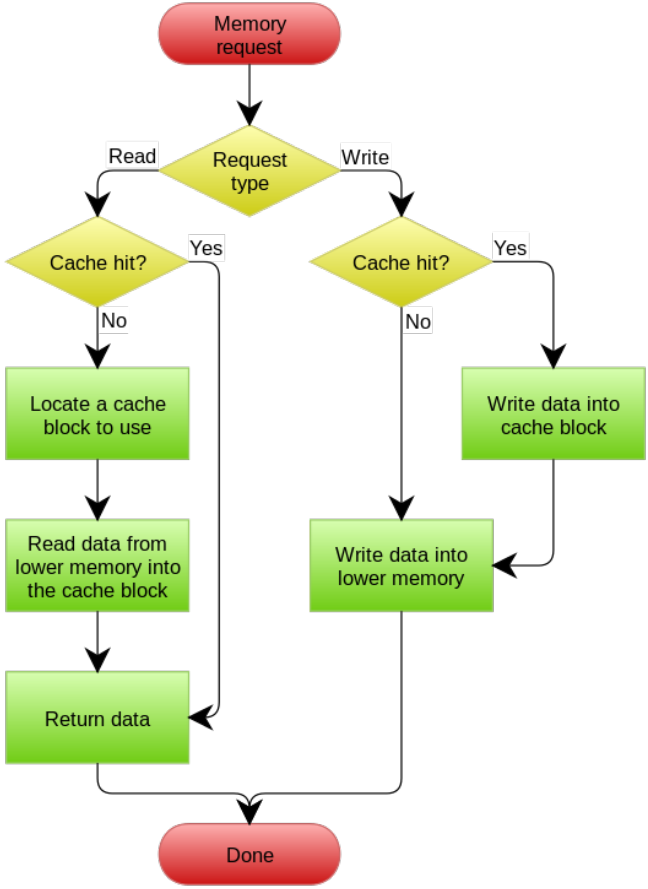


Cache Flow Chart

- **Cache block**: unit of data used to cache data
  - What **page** is to memory paging
  - Cache block size is typically multiple words (e.g. 32 bytes or 64 bytes. You'll see why.)

- **Good**: **Memory Wall** can be **surmounted**
  - On cache hit, no need to go to DRAM.

- **Bad**: MEM stage has **variable latency**
  - Typically, only a few cycles if cache hit
  - More than a 100 cycles if cache miss! (Processor must go all the way to DRAM.)
  - Makes performance very **unpredictable**



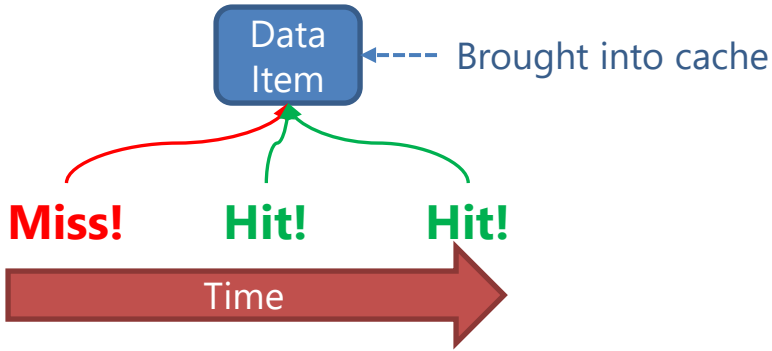
Temporal Locality and Spatial Locality

University of Pittsburgh

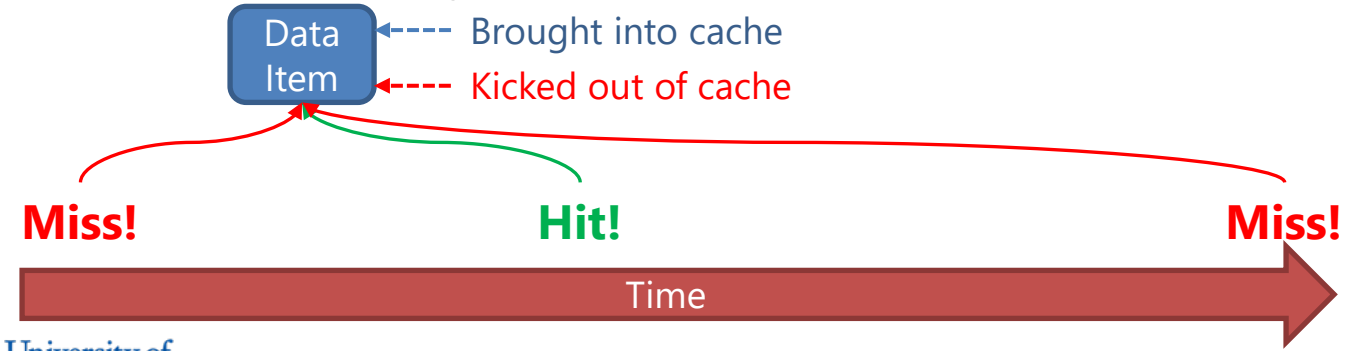
Cache Locality: Temporal Locality

- **Temporal Locality**: How close together are accesses in **time**?

- High temporal locality:



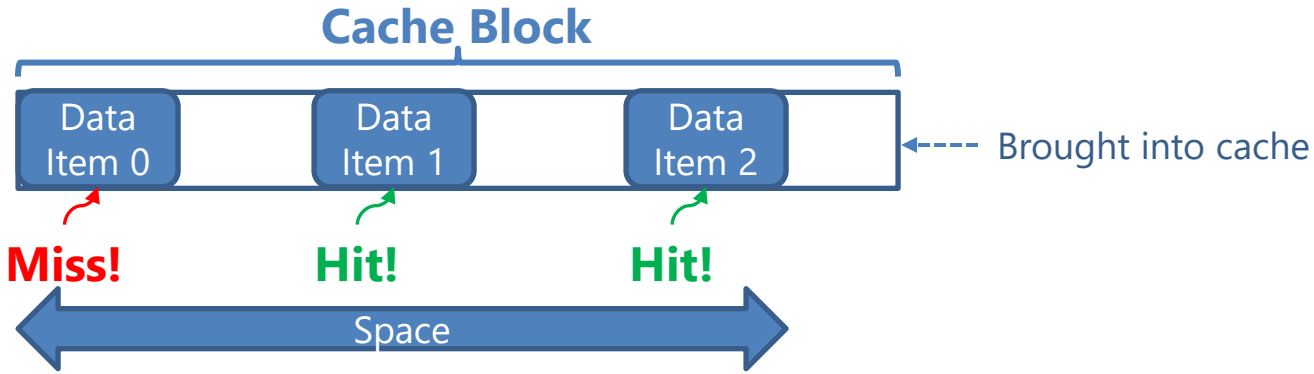
- Low temporal locality:



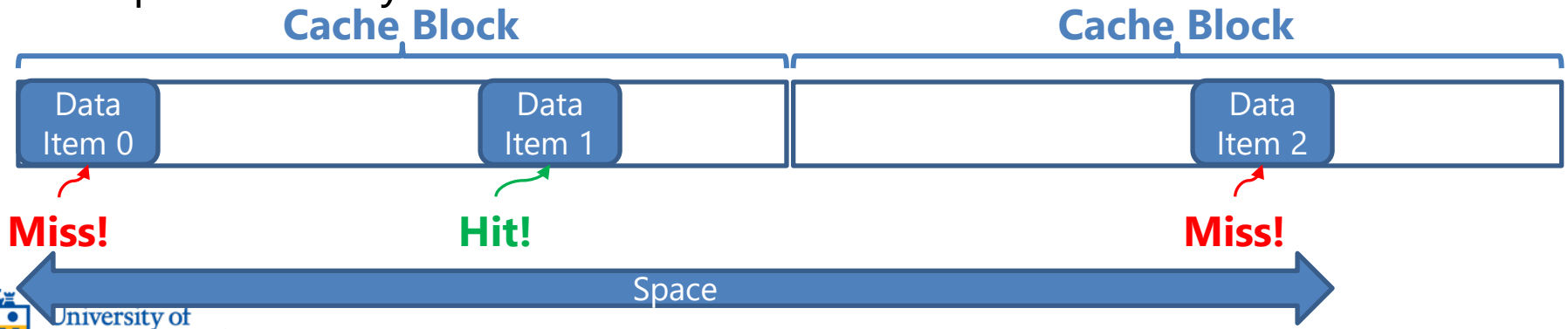
Cache Locality: Spatial Locality

- **Spatial Locality**: How close together are accesses in **space**?

- High spatial locality:



- Low spatial locality:



Cache Locality: Temporal and Spatial

- Caching works only if there is **locality** in program data accesses
  - **Temporal locality**
    - How close together in **time** are accesses to the **same item**
    - 1st access will miss but following accesses can hit in the cache
  - **Spatial locality**
    - How close together in **space** are accesses to **different items**
    - 1st access will miss but bring in an entire cache block
    - Accesses to other items within same cache block will hit
    - E.g., object fields or array elements often accessed consecutively

- Locality, like ILP, is a **property of the program**

Cold Misses and Capacity Misses

- **Cold miss** (a.k.a. **compulsory miss**)
  - Miss suffered when data is accessed for the **first time** by program
  - Cold miss since cache **hasn't been warmed up** with accesses
  - Compulsory miss since there is no way you can hit on the first access

- **Capacity miss**
  - Miss suffered when data is accessed for the **second or third times**
  - This miss occurs because **data was replaced** to make space
  - Not compulsory since with more capacity, miss wouldn't have happened
  - Capacity determines how much **locality** you can leverage

Reducing Cold Misses

- How do you reduce cold misses? Is it even possible?
  - Yes! By taking advantage of **spatial locality**.
  - Have a larger cache block so you bring in other data items on a miss.
  - If those other items are accessed, they will hit, even on first access!

- **Large cache blocks:**
  - \+ Potentially **reduce cold misses** (by leveraging spatial locality)
  - \+ Potentially **reduce capacity misses** (again, leveraging spatial locality)
  - – Potentially **increase capacity misses** (pollutes cache with useless items)

- So, larger or smaller cache blocks?
  - It all depends on how much spatial locality each program has.

Reducing Capacity Misses

- How do you reduce capacity misses?
  - Easy! Increase capacity to take advantage of **temporal locality**.

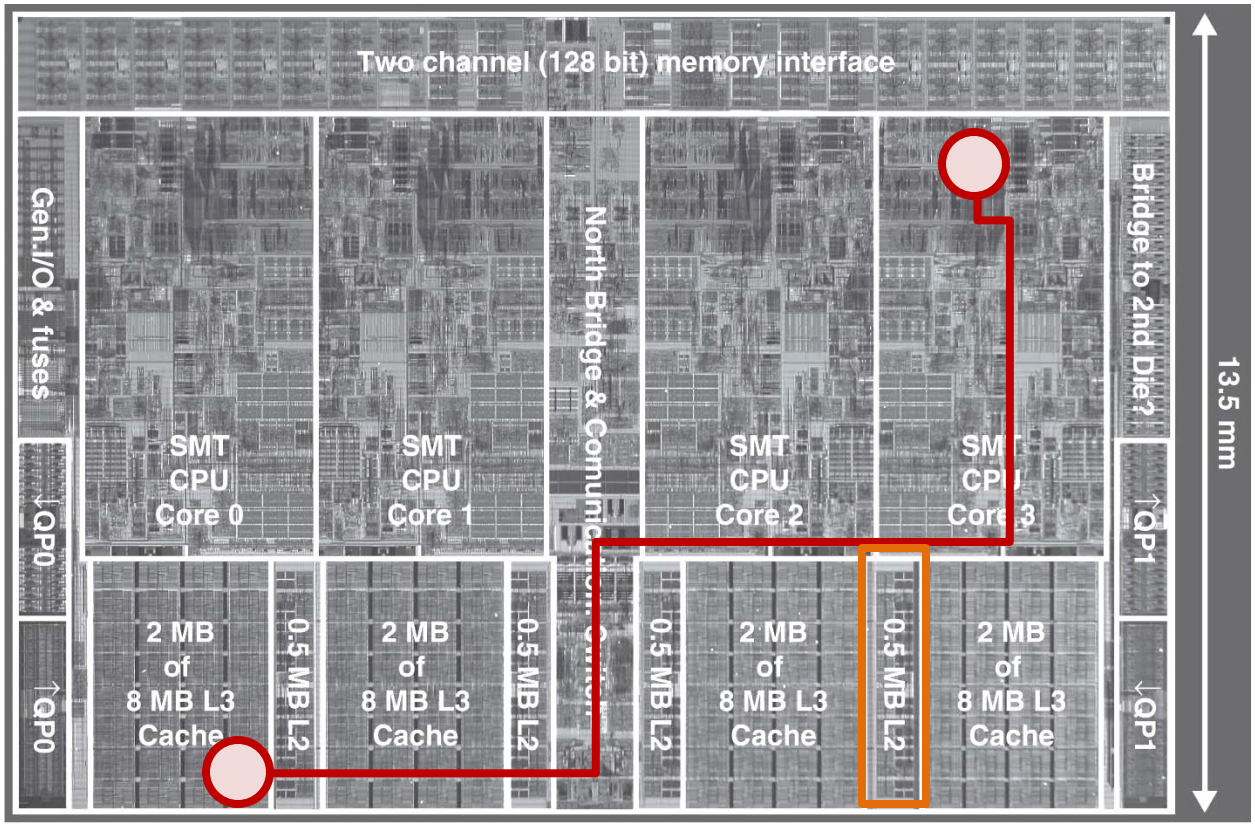
- **Large caches**:
  - \+ Potentially **reduce capacity misses** (by leveraging temporal locality)
  - – Potentially **increase cache access delay** (due to longer wires)

- So, larger or smaller caches?
  - Again, it depends on how much temporal locality each program has.

Bigger caches are slower

- Below is a diagram of an Intel Nehalem CPU

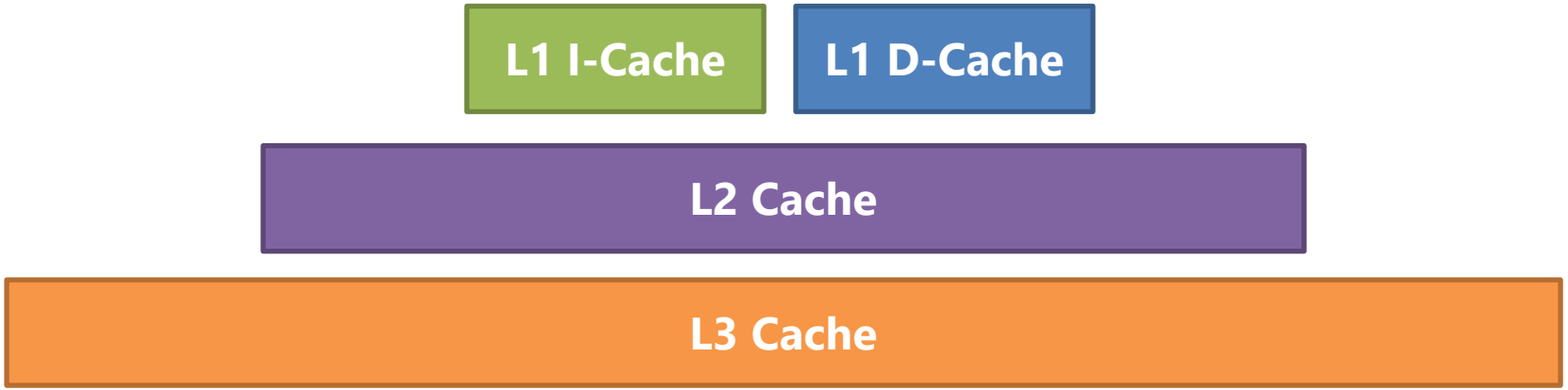
- How long do you think it takes for data to make it from here...
- ...to here?
- It must be routed through all this.
- Can we cache the data in the far away "L3 Cache" to a nearby "L2 Cache"?



Multi-level Caches

Multi-level Caching gives choice of where to put data

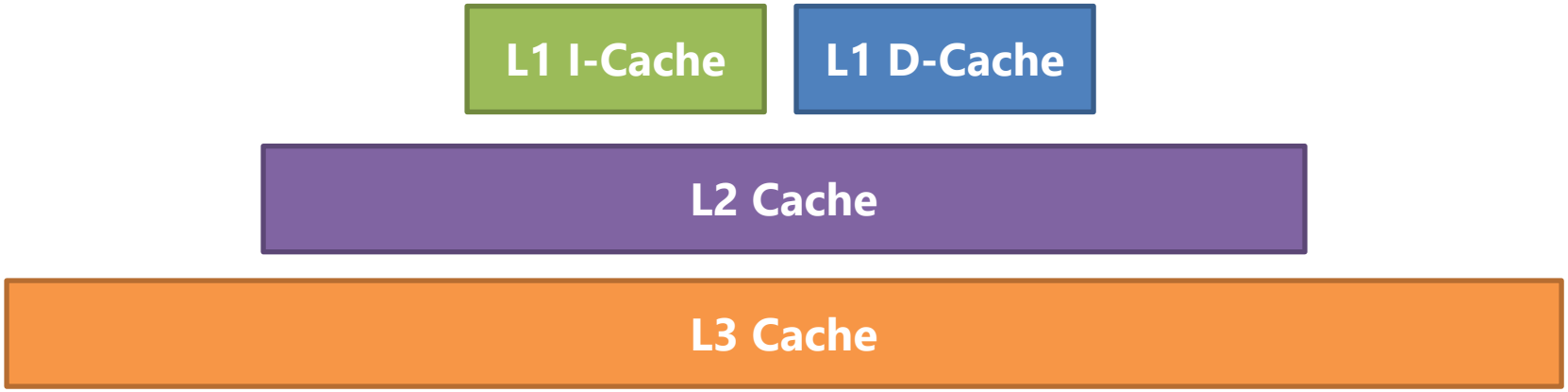
- Put data with **high temporal locality** in **small** caches.
- Put data with **low temporal locality** in **bigger** caches.



- At above is the structure of the kernighan.cs.pitt.edu Xeon CPU.
- L1 cache: **Small** but **fast**. Interfaces with CPU pipeline MEM stage.
  - Split to i-cache and d-cache to avoid structural hazard.
- L2 cache: Middle-sized and middle-fast. Intermediate level.
- L3 cache: **Big** but **slow**. Last line of defense against DRAM access.

Typical use of multi-level caches in software

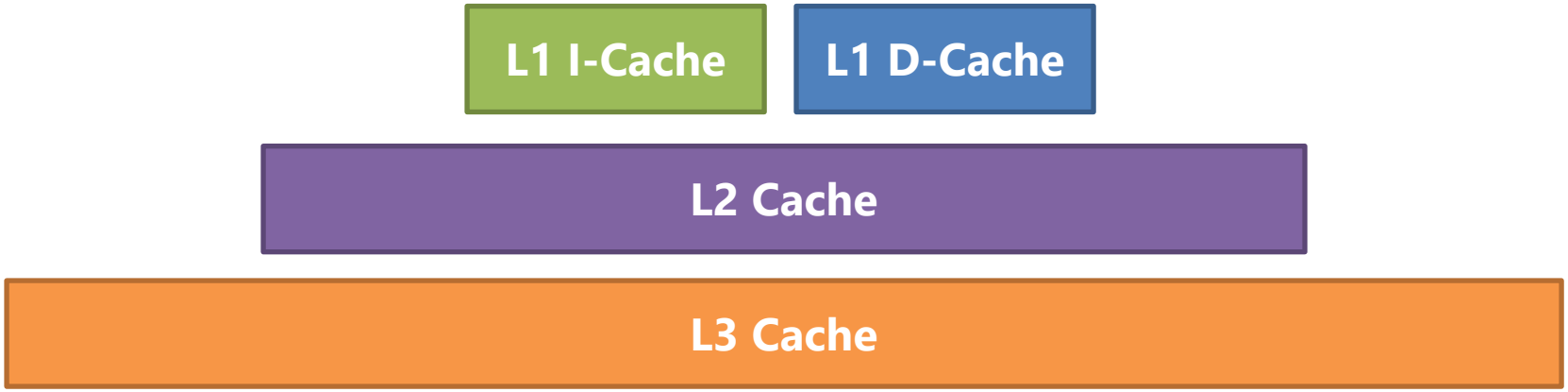
- Goal: place data at optimal cache level depending on temporal locality.



- L1 cache: **Small** data set that is accessed **frequently**
  - Local variables in stack memory, code within current loop
- L2 cache: **Medium** data set that is accessed **less frequently**
  - Currently accessed smaller data structures, code within current function
- L3 cache: **Big** data set that is accessed **even less frequently**.
  - Currently accessed bigger data structures, code within current module
- Up to HW cache replacement algorithm to place data sets appropriately

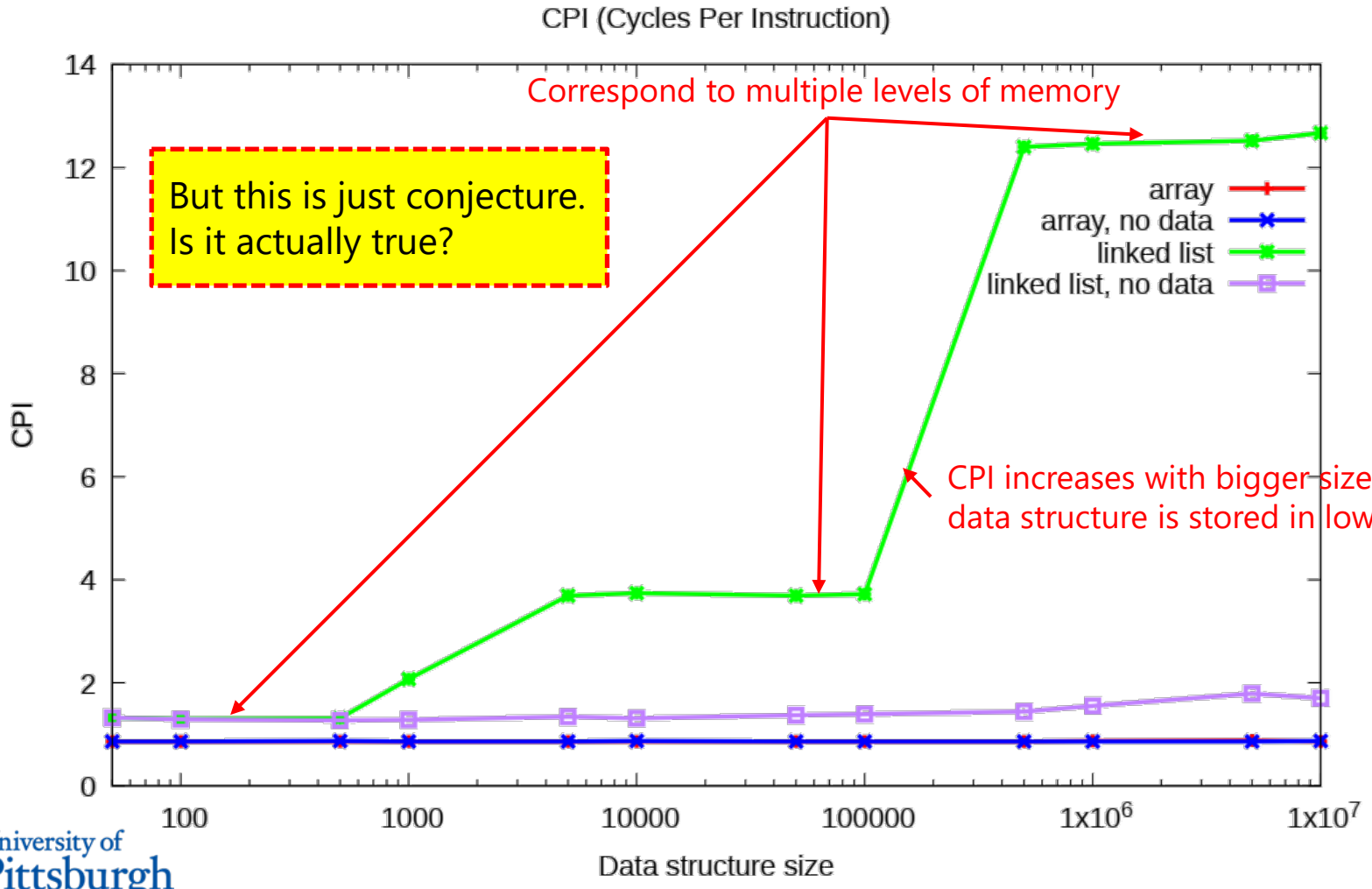
Multi-level caches may use hybrid memories

- Smaller caches use SRAM / bigger caches use DRAM.



- L1 cache, L2 cache: typically uses SRAM
  - Small enough that memory cell delay is important
- L3 cache: May use eDRAM (embedded DRAM) over SRAM
  - Can fit in much more eDRAM than SRAM for same silicon area
  - At high capacity, memory cell delay is insignificant

Revisiting our CPI Results with the new perspective



kernighan.cs.pitt.edu cache specs

- On a Xeon E5-2640 v4 CPU (10 cores):
  - L1 i-cache: **32 KB** 8-way set associative (per core)
  - L1 d-cache: **32 KB** 8-way set associative (per core)
  - L2 cache: **256 KB** 8-way set associative (per core)
  - L3 cache: **25 MB** 20-way set associative (shared)

  Ref: https://en.wikichip.org/wiki/intel/xeon_e5/e5-2640_v4

- Access latencies (each level includes latency of previous levels):
  - L1: ~**3 cycles**
  - L2: ~**8 cycles**
  - L3: ~**16 cycles**
  - DRAM Memory: ~**67 cycles**

  Ref: https://www.nas.nasa.gov/assets/pdf/papers/NAS_Technical_Report_NAS-2015-05.pdf

Cache Specs Reverse Engineering

- Why do I have to refer to a NASA technical report for latencies?
  - Ref: https://www.nas.nasa.gov/assets/pdf/papers/NAS_Technical_Report_NAS-2015-05.pdf
  - Because Intel doesn't publish detailed cache specs on data sheet
- In the technical report (does the step function look familiar?):

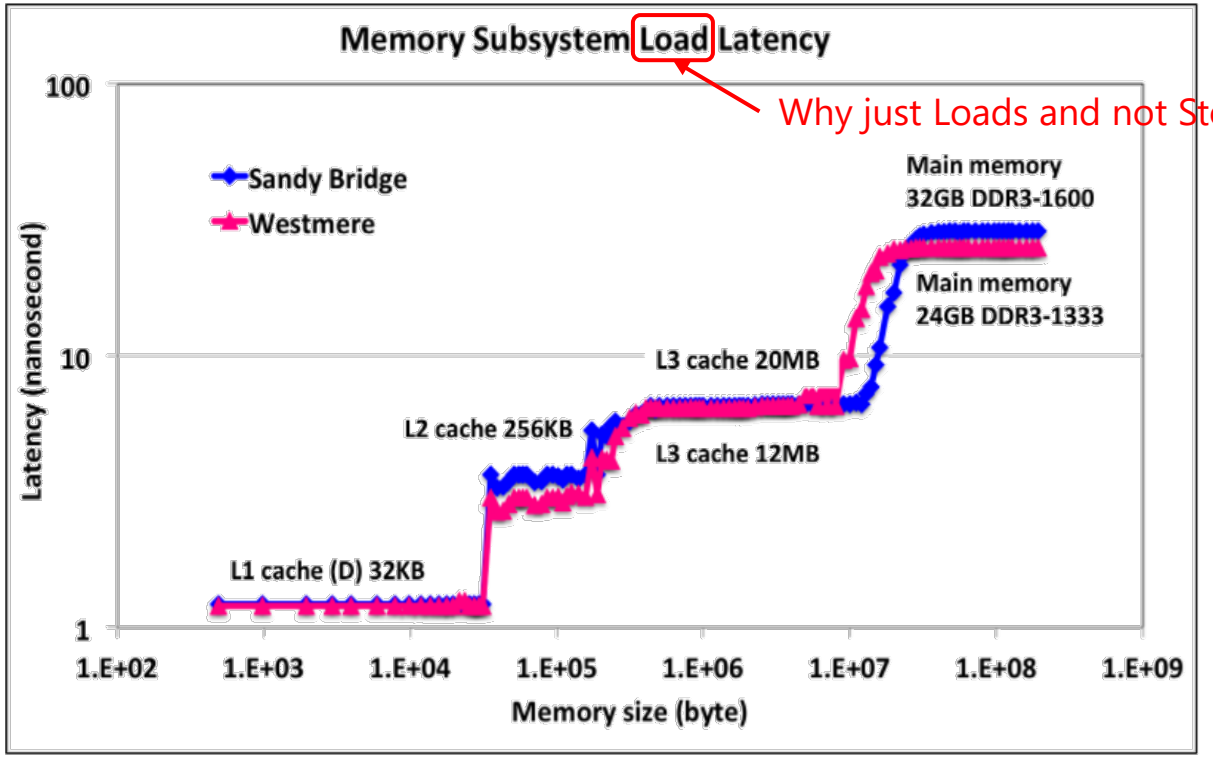


Figure 2. Memory latency of Westmere and Sandy Bridge.

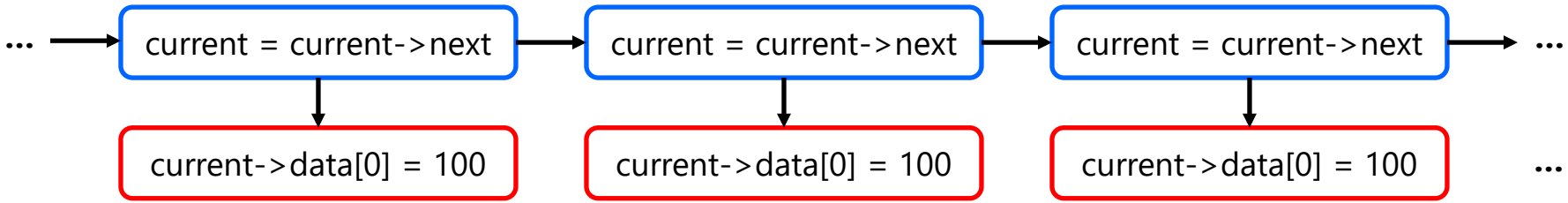
Store vs. Load Memory Accesses

Loads have more impact on performance

- Suppose we added a store to our original loop:

```
node_t* current = head;
for(int i=0; i<ACCESSES; i++) {
  if(current == NULL) current = head;          // reached the end
  else {
    current->data[0] = 100;                    // store to node data
    current = current->next;                   // load next node address
}
```

- Which would have more impact on performance? The load or the store?
  - A: The load because it is on the **critical path**.



Loads have more impact on performance

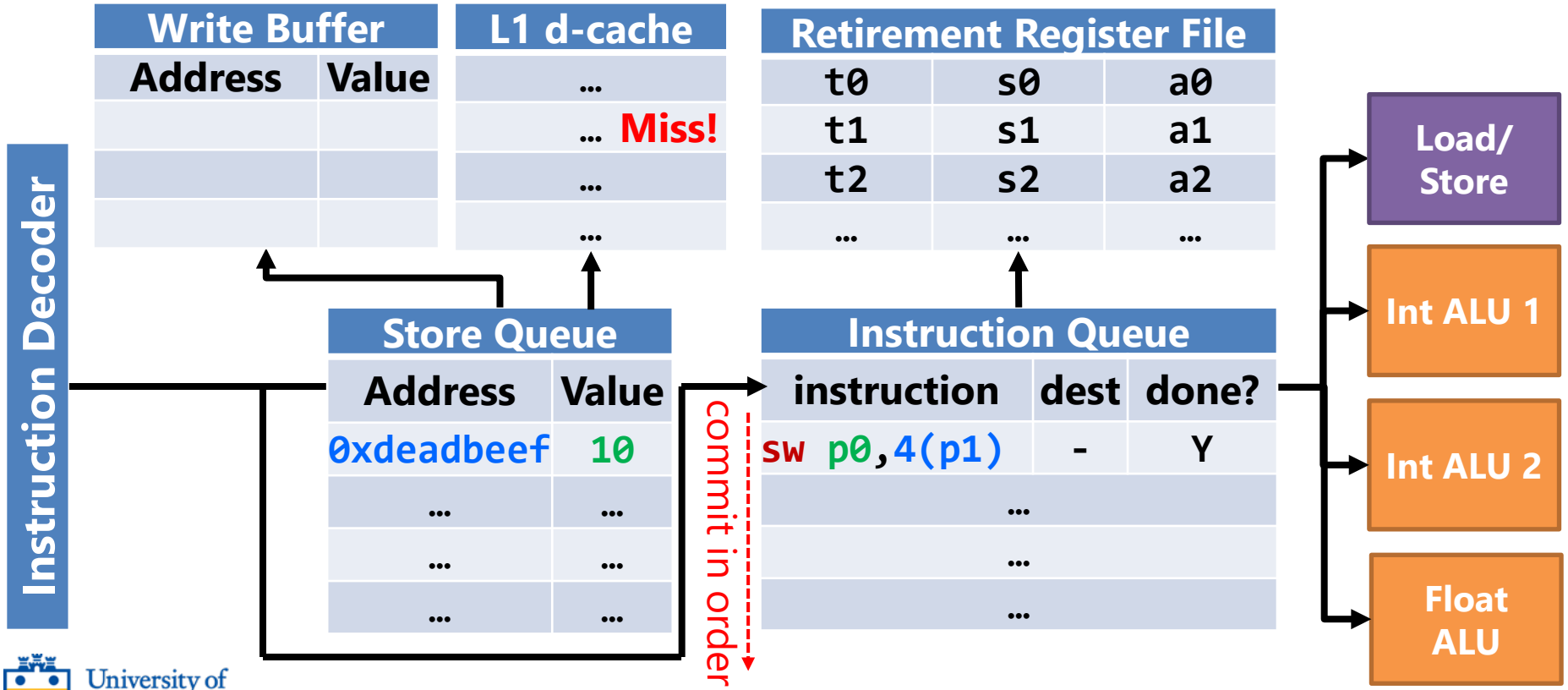
- Loads produce values needed for computation to proceed
  - **Stalled loads delay computation** and possibly the critical path

- Stores write computation results back to memory
  - As long as the results are written back eventually, no big hurry
  - If store results in a cache miss,
    store is marked as "pending" and CPU moves on to next computation
  - Pending stores are maintained in a **write buffer** hardware structure

- What if the next computation reads from a pending store?
  - First check the write buffer and read in the value if it's there
  - Again, performing the store is not on the critical path

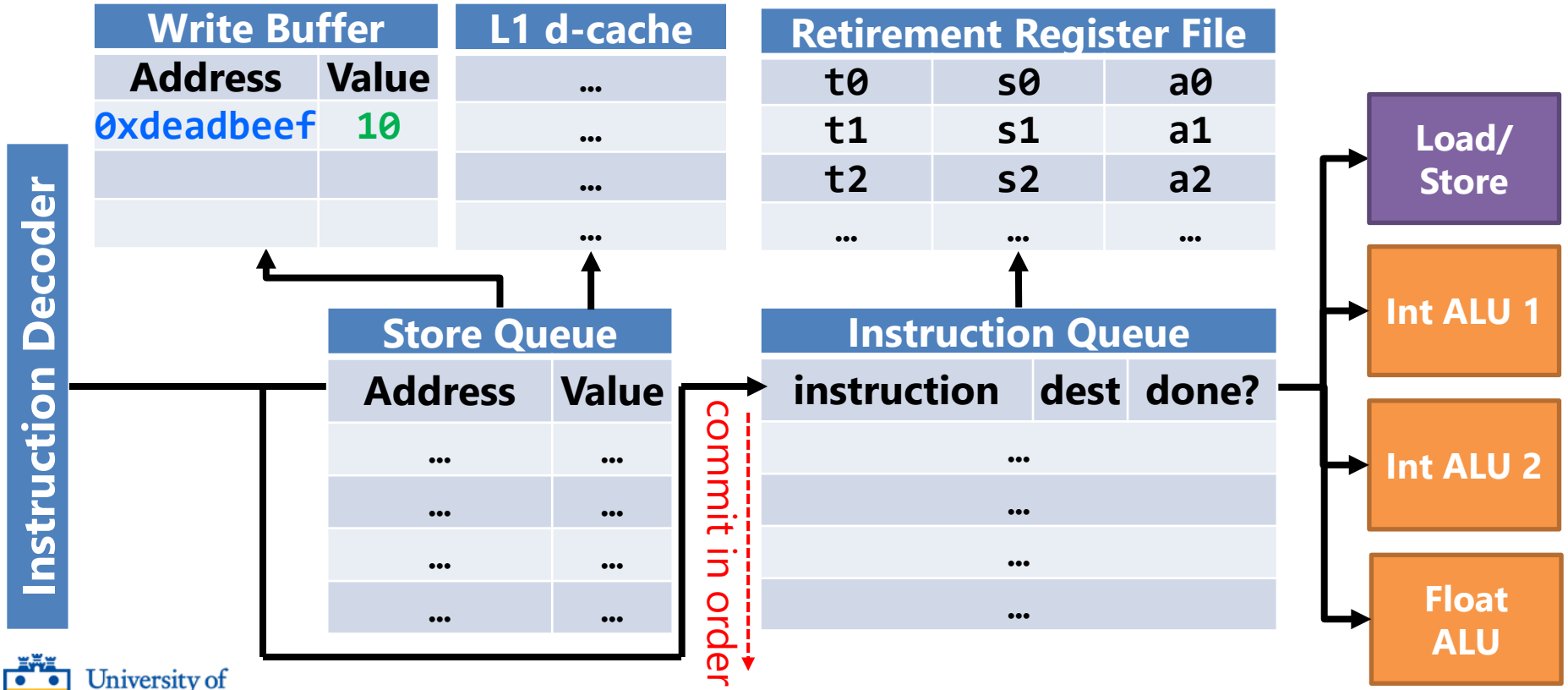
How Write Buffer Maintains Pending Stores

- `sw p0,4(p1)` is about to commit. `4(p1)` == `0xdeadbeef`, `p0` == `10`
- Unfortunately, address `0xdeadbeef` is not in the L1 d-cache and it misses



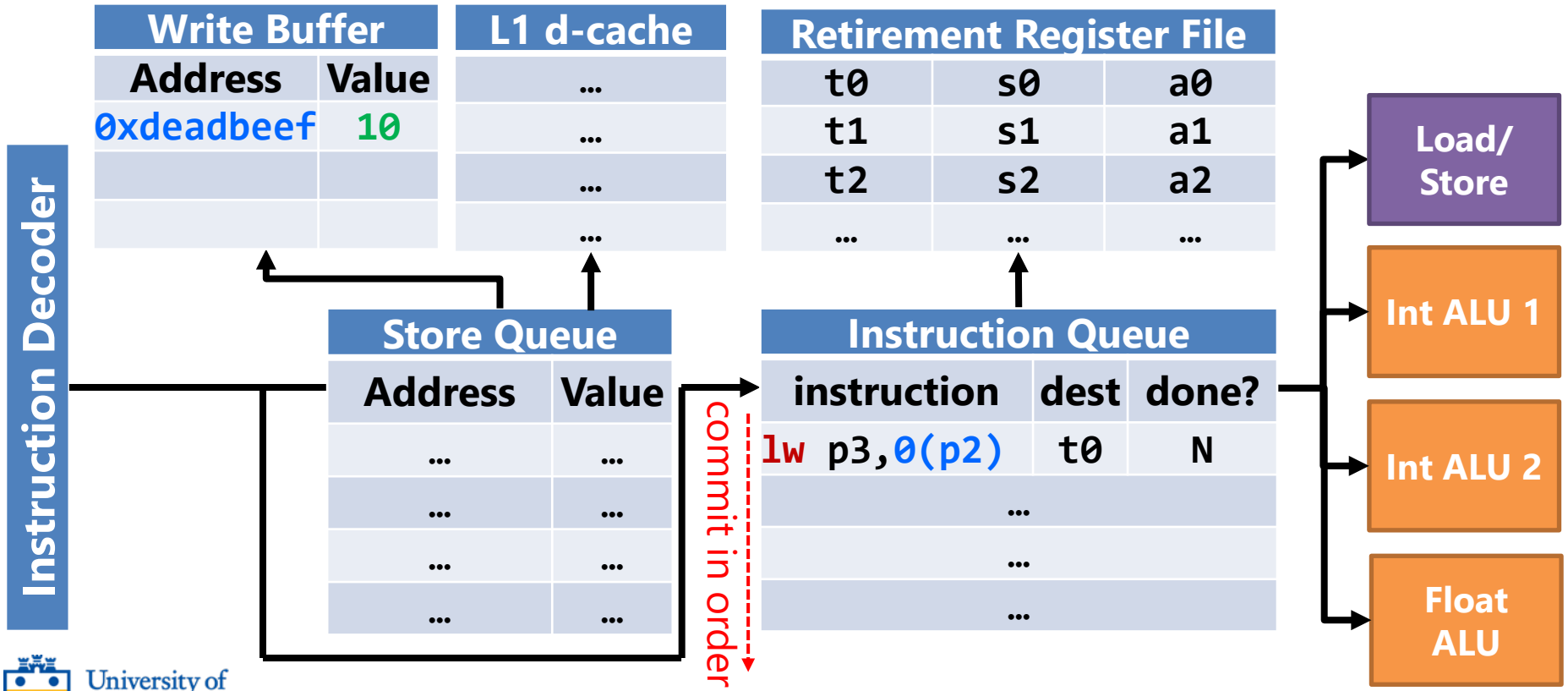
How Write Buffer Maintains Pending Stores

- `sw p0,4(p1)` commits successfully anyway
- The store is moved to the Write Buffer and stays there until store completes



How Write Buffer Maintains Pending Stores

- Later, when `lw p3,0(p2)` comes along, it checks Write Buffer first
- If `0(p2)` == `0xdeadbeef`, Write Buffer provides value instead of memory



So are stores never on the critical path?

- If we had an infinitely sized write buffer, no, never.

- In real life, write buffer is limited and can suffer structural hazards
  - If write buffer is full of pending stores, you can't insert more.

  → That will prevent a missing store from committing from i-queue

  → That will prevent all subsequent instructions from committing

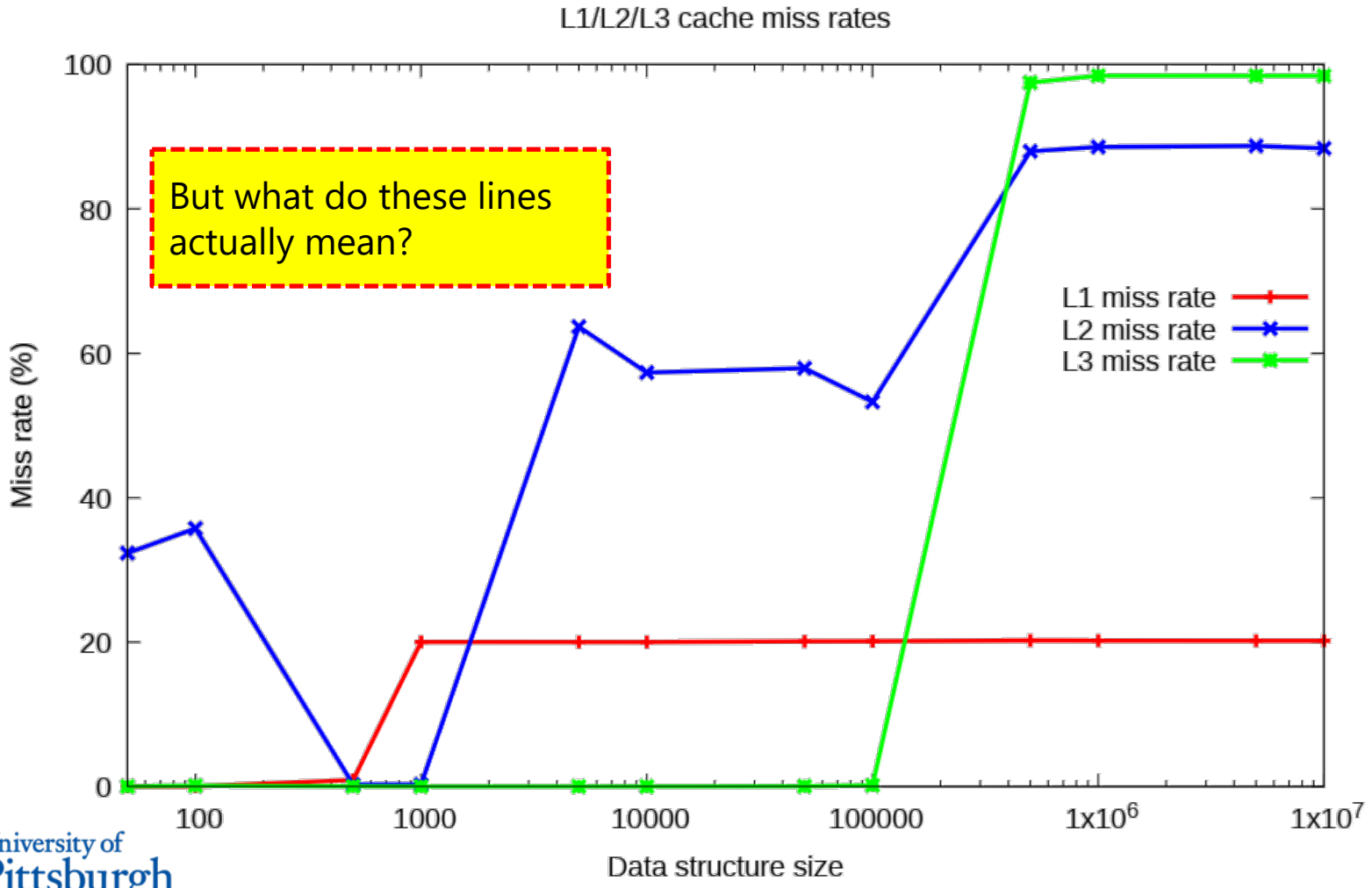
  → That will eventually stall the entire pipeline

- But with ample write buffer size, happens rarely
  - And if it does happen can be detected using PMUs

- Hence, we will focus on loads to analyze performance

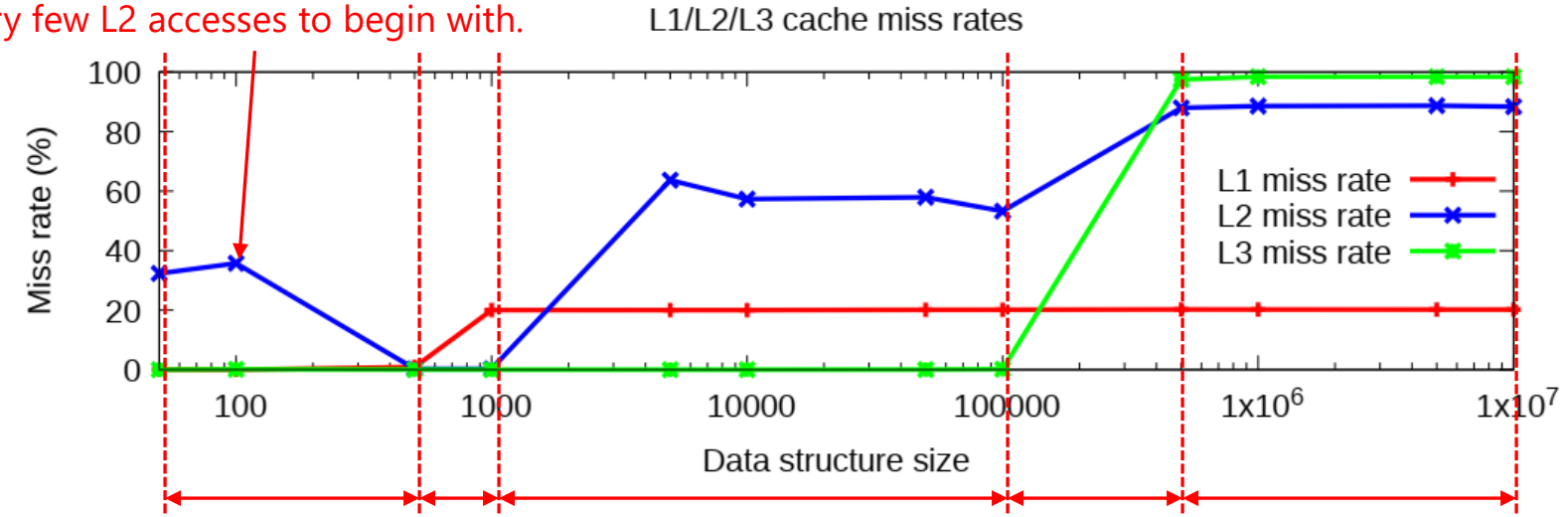
Analyzing Load Miss Rates

Linked List Cache Load Miss Rates

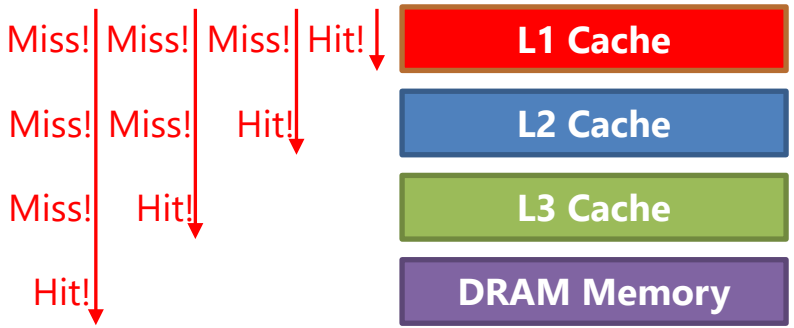


Linked List Cache Load Miss Rates

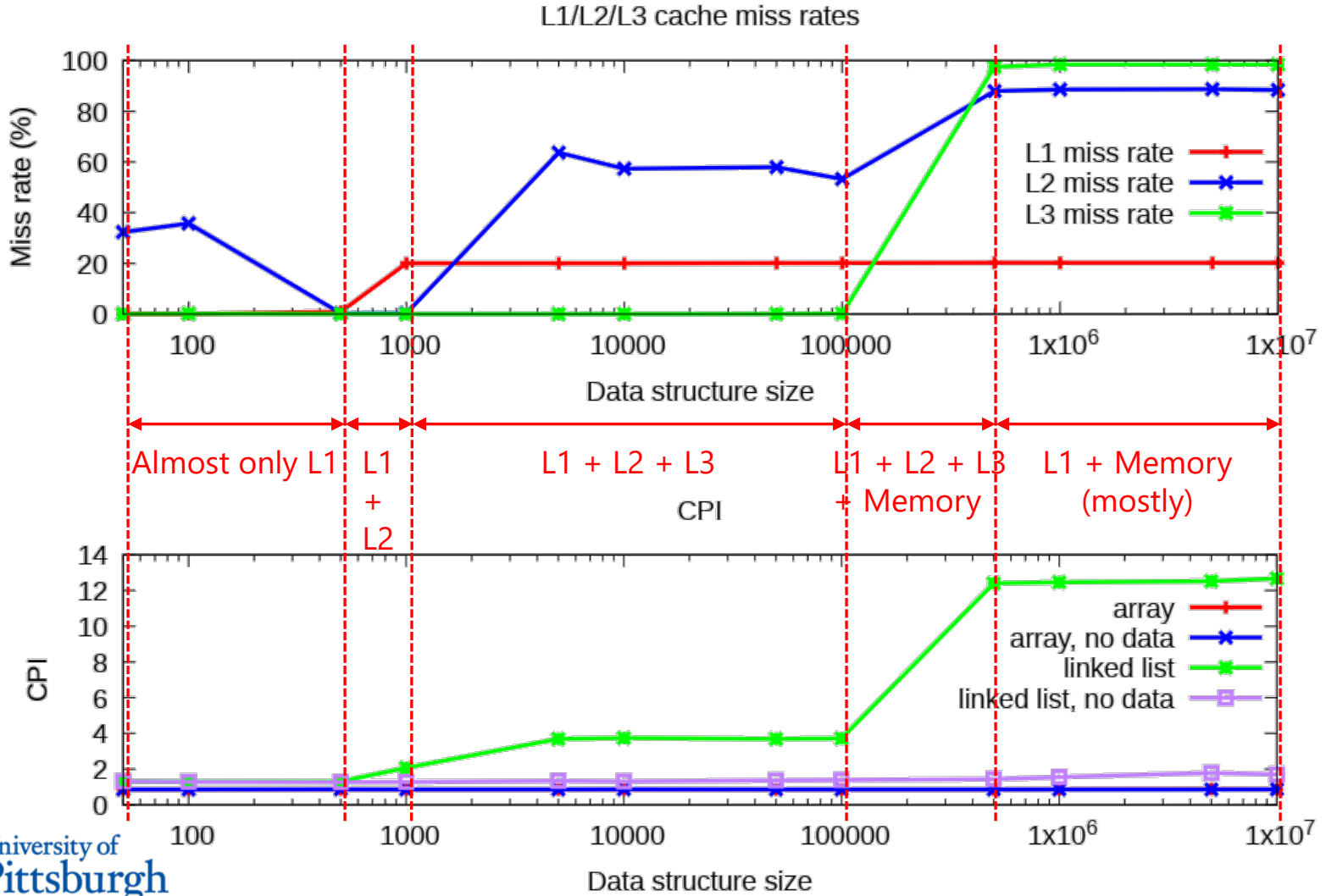
A few L2 cold misses result in high miss rate.
But very few L2 accesses to begin with.



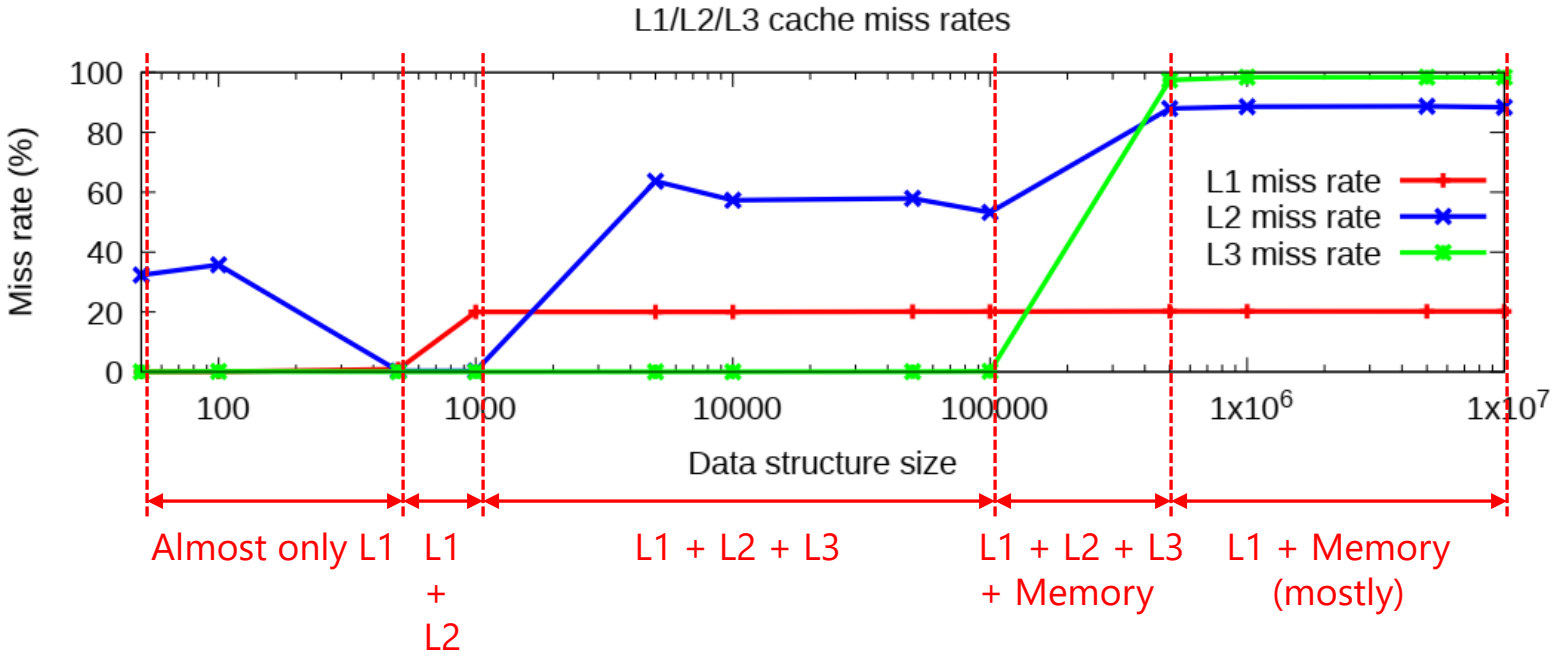
Do the miss rates correspond to CPI results?
Let's compare with our own eyes!



Linked List Cache Load Miss Rates vs CPI



Linked List Cache Load Miss Rates – Why steps?



- Why the step up in L1 cache misses between 500 – 1000 nodes?
- Why the step up in L2 cache misses between 1000 – 5000 nodes?
- Why the step up in L3 cache misses between 100 k – 500 k nodes?
- Also, why do cache miss increases look like **step functions** in general?

Working Set Overflow can cause Step Function

- The size of a node is 128 bytes:

```
typedef struct node {
  struct node* next;     // 8 bytes
  int data[30];          // 120 bytes
} node_t;
```

- **Working set**: amount of memory program accesses during a phase
  - For linked-list.c, working set is the entire linked list
    - Program accesses entire linked list in a loop over and over again
  - If there are 8 nodes in linked list, working set size = 128 * 8 = 1 KB
- When working set **overflows** cache capacity, start to see **cache misses**
  - Miss increase can be drastic, almost like a **step function**
  - Suppose cache size is 1 KB and nodes increase from 8 → 9
    - When **8** nodes: **always hit** (since entire list in contained in cache)
    - When **9** nodes: **always miss** (if least recent node is replaced first)

Least Recently Used (LRU) Fallacy

- When a cache block needs to be replaced, which one to choose?

- Least Recently Used (LRU) is usually a good policy
  - Premise: if a block hasn't been used recently, it won't be in the near future
- But it can cause performance cliffs on certain access patterns

- Ex: Cache has 8 blocks, working set size is 9 blocks accessed iteratively
  - Iteration 1: Accesses to Blocks 1 ~ 9 all result in cold misses.

Block 9	Block 2	Block 3	Block 4	Block 5	Block 6	Block 7	Block 8

  - Iteration 2: Accesses to Blocks 1 ~ 9 all result in capacity misses.

Block 8	Block 9	Block 2	Block 3	Block 4	Block 5	Block 6	Block 7

  - Iteration 3: All capacity misses again (same for all future iterations).

Block 7	Block 8	Block 9	Block 2	Block 3	Block 4	Block 5	Block 6

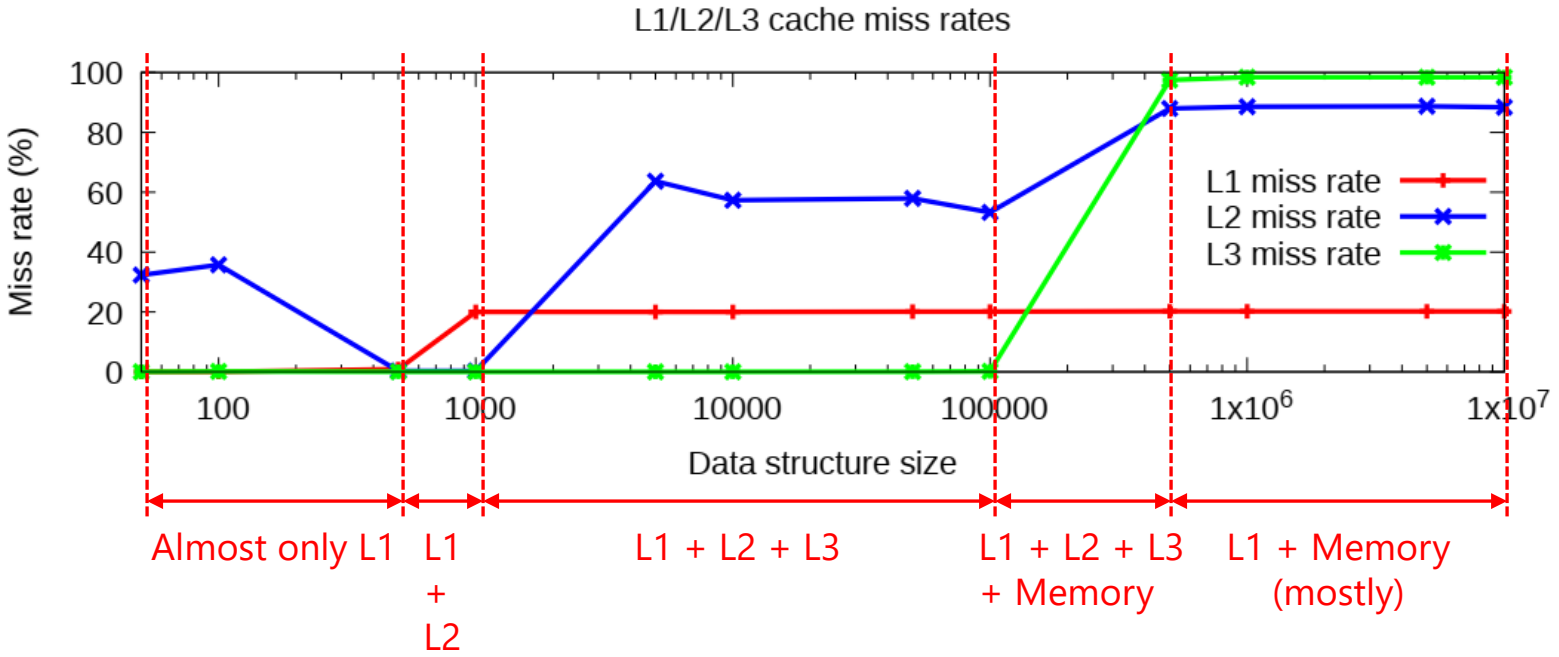
Linked List Cache Load Miss Rates – Why steps?

- Why the step up in L2 cache misses between 1000 – 5000 nodes?
  - L2 cache size is 256 KB
  - Number of nodes that can fit = 256 KB / 128 = 2048

- Why the step up in L3 cache misses between 100 k – 500 k nodes?
  - L3 cache size is 25 MB
  - Number of nodes that can fit = 25 MB / 128 ≈ 200 k

- Why the step up in L1 cache misses between 500 – 1000 nodes?
  - L1 d-cache size is 32 KB
  - Number of nodes that can fit = 32 KB / 128 = 256
  - So, in theory you should already see a step up at 500 nodes
  - Apparently, CPU doesn't use least-recently-used (**LRU**) replacement
  - According to another reverse engineering paper, Intel uses PLRU
    Ref: "CacheQuery: Learning Replacement Policies from Hardware Caches" by Vila et al.
    https://arxiv.org/pdf/1912.09770.pdf

Linked List – Why does L1 Miss Rate hold at 20%?



- Why did L1 cache miss rate saturate at around 20%?
  - Shouldn't it keep increasing with more nodes like L2 and L3?

Linked List – Why does L1 Miss Rate hold at 20%?

[linked-list.c]

```
void *run(void *unused) {
  node_t* current = head;
  for(int i=0; i<ACCESSES; i++) {
    if(current == NULL) current = head;
    else current = current->next;
  }
}
```

Within a typical iteration in for loop:
4 blue loads that hit in L1:
- 2 loads each of local vars current, i
- Read frequently so never replaced

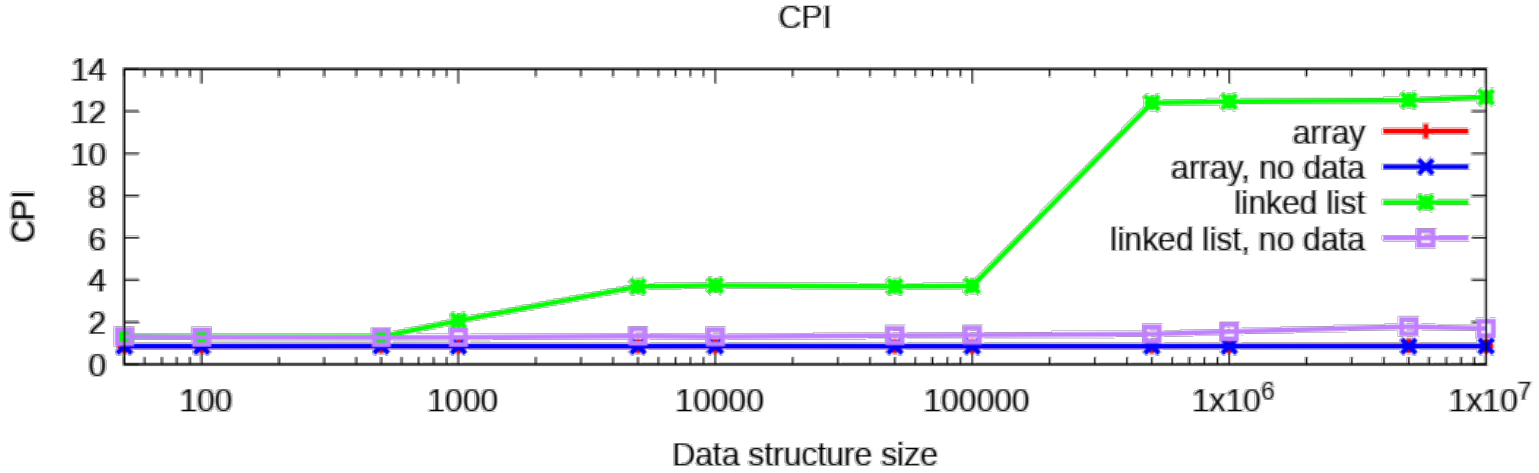
1 red load that misses in L1:
- current->next (next field of node)
- Node may not be in cache and miss (e.g. due to a capacity miss)

☛ 1 miss / 5 loads = 20% miss rate

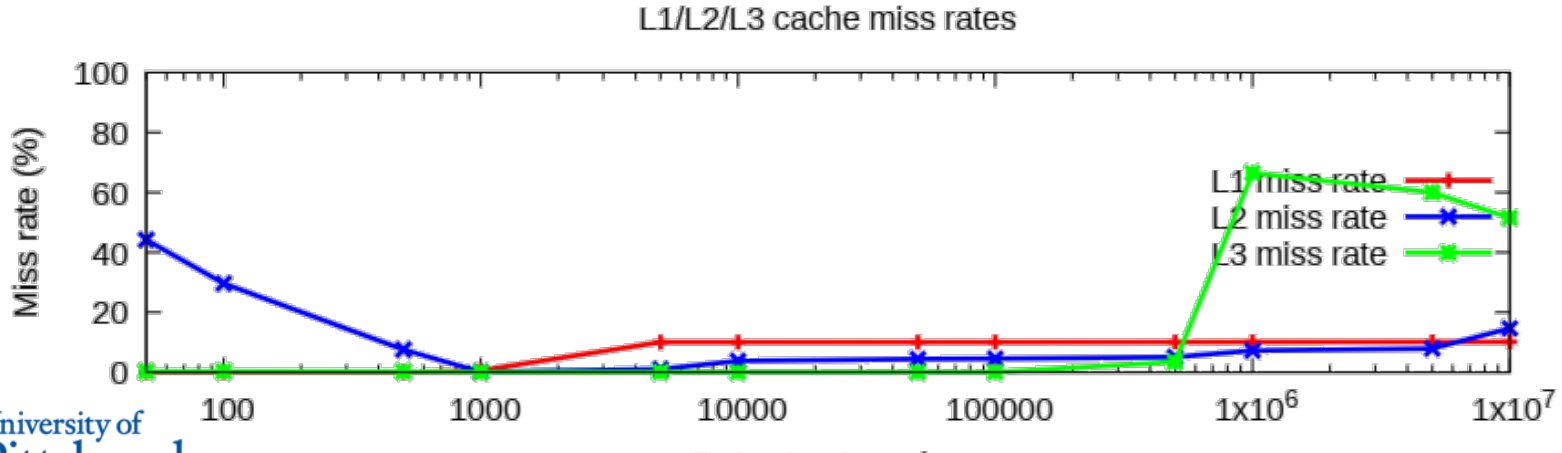
[objdump -S linked-list]

```
0000000000400739 <run>:
...
400741: mov 0x200920(%rip),%rax # %rax = head
400748: mov %rax,-0x8(%rbp) # current = %rax
40074c: movl $0x0,-0xc(%rbp) # i = 0
400753: jmp 400778 # jump to i < ACCESSES comparison
400755: cmpq $0x0,-0x8(%rbp) # current == NULL?
40075a: jne 400769 # jump to else branch if not equal
40075c: mov 0x200905(%rip),%rax # %rax = head
400763: mov %rax,-0x8(%rbp) # current = %rax
400767: jmp 400774 # jump to end of if-then-else
400769: mov -0x8(%rbp),%rax # %rax = current
40076d: mov (%rax),%rax # %rax = current->next
400770: mov %rax,-0x8(%rbp) # current = %rax
400774: addl $0x1,-0xc(%rbp) # i++
400778: cmpl $0x3b9ac9ff,-0xc(%rbp) # i < ACCESSES ?
40077f: jle 400755 # jump to head of loop if less than
```

Linked List w/o Data Cache Load Miss Rates

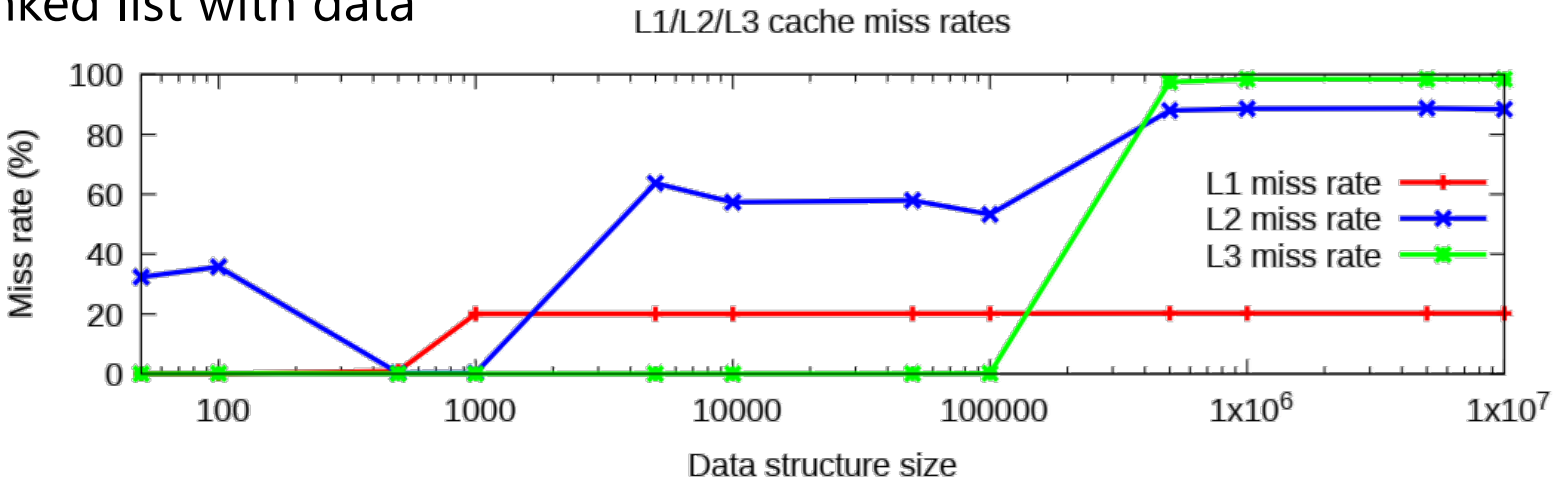


- Linked list, no data suffered almost no CPI degradation. Why?

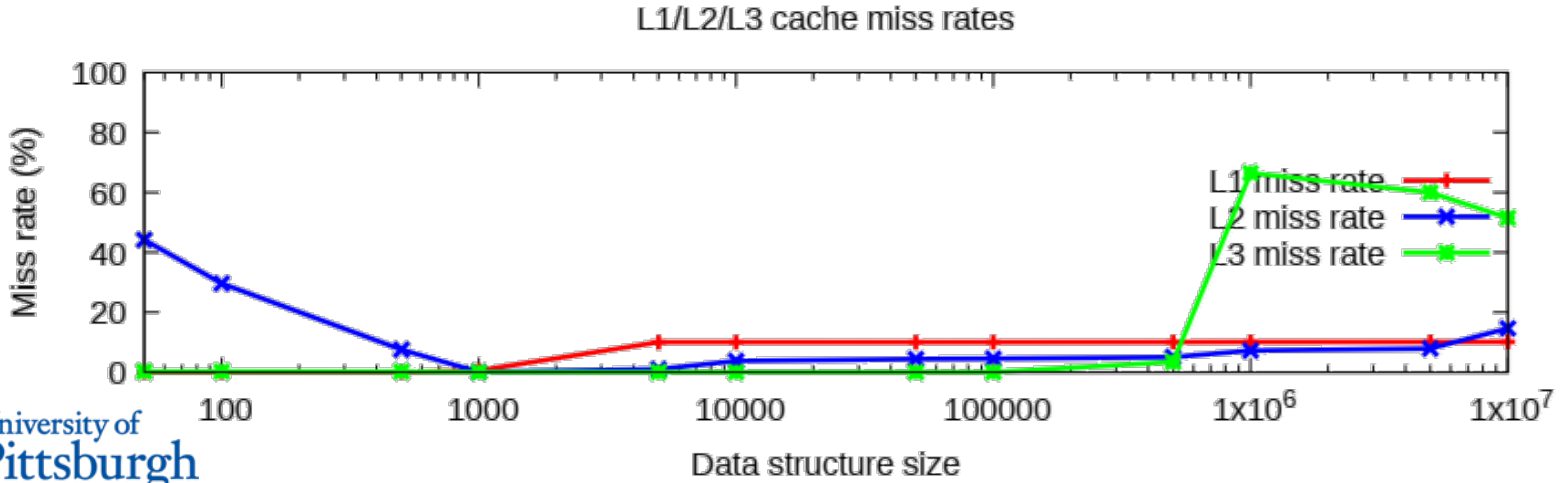


Linked List w/ Data vs. w/o Data

- Linked list with data



- Linked list with no data



Linked List w/ Data vs. w/o Data. Why?

- The size of a node with no data is only 8 bytes:

```
typedef struct node {
  struct node* next;     // 8 bytes
  // no data
} node_t;
```

- Compared to 128 bytes with data, can fit in 16X more nodes in cache
  - **Temporal locality**: More likely that a node will be present in cache

- How about L1 cache miss rate that hovers around 10% instead of 20%?
  - By 10^7 nodes, there is no temporal locality with respect to the L1 cache
  - **Spatial locality** must be responsible for the reduction in miss rate

Linked List w/ Data vs. w/o Data. Why?

- Nodes of the linked list are malloced one by one in a loop:

```
for(int i=0; i<items; i++) {
  node_t* n = (node_t*)malloc(sizeof(node_t));
  …
}
```

  - I have no idea where glibc malloc decides to allocate each node

- But knowing each cache block is 64 bytes long in the Xeon E5 processor
  - Let's say multiple nodes are allocated on same cache block:

  - Then even if access to node 1 misses, due to a capacity miss, accesses to nodes 37 and 23 that soon follow will hit!
  - This is assuming there is some **spatial locality** in how malloc allocates

Data structure with most spatial locality: Array

- Elements of an array are guaranteed to be in contiguous memory:

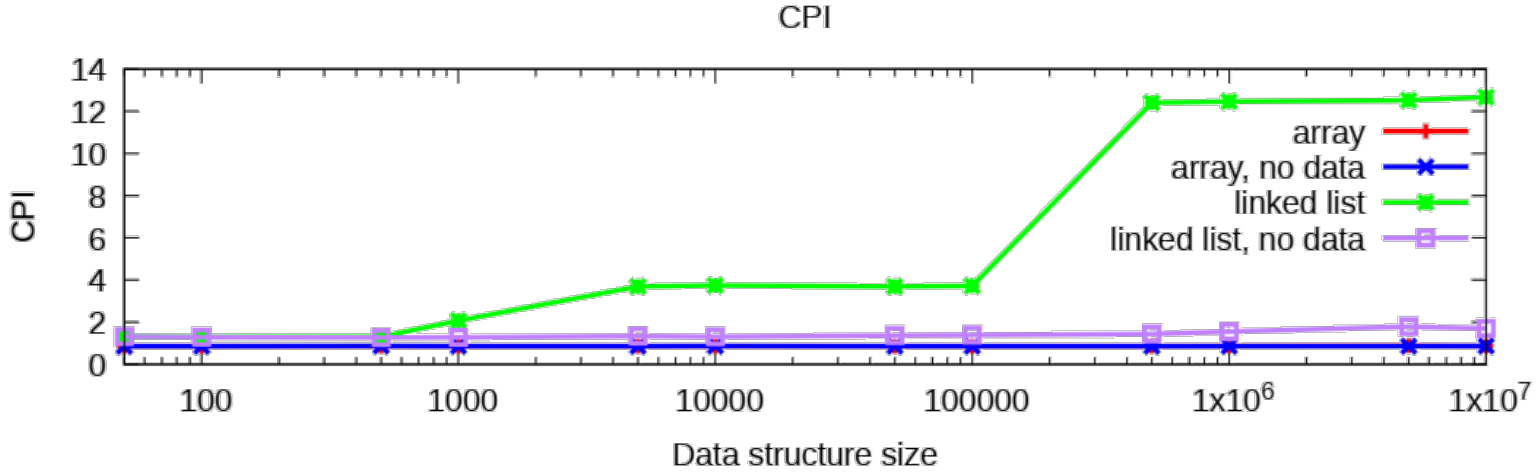
```
void *create(void *unused) {
  head = (node_t *) malloc(sizeof(node_t) * items);
  …
}
```

- Each cache block is 64 bytes long in the Xeon E5 processor
  - Now 8 elements are guaranteed to be on same cache line, in order:

node 0	node 1	node 2	node 3	node 4	node 5	node 6	node 7

  - Even with cold cache, only 1 out of 8 nodes miss (1/8 = 12.5% miss rate)
  - Assuming that nodes are accessed sequentially
    - If accessed in random order, no spatial locality, even with array
  - True regardless of capacity (even if cache contained only a single block)

Let's look at the CPI of arrays finally



- Array, no data did very well as expected
  - The most spatial locality of the four benchmarks (contiguous nodes)
  - Smallest memory footprint so can also leverage temporal locality the best
- Array performed the same as array, no data. How come?
  - No spatial locality since each node takes up two 64-byte cache blocks
  - Has much larger memory footprint compared to array, no data
  - This is the real mystery. We will learn more about it as we go on. ☺