Memory Hierarchy

CS/COE 1541 (Fall 2020)
Wonsun Ahn

University of Pittsburgh

- The source code for the experiments are available at: https://github.com/wonsunahn/CS1541_Fall2020/tree/master/resources/cache_experiments

- Or on 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
  o It will take a few minutes to run all the experiments
  o In the end, you get two plots: IPC.pdf and MemStalls.pdf

University of Pittsburgh

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

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

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

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

```
// Define a linked list node type with no 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;
    }
  }
}
```

```
#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?

```c
// Define a linked list node type with no 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;
}
```

```
#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?

```c
#define ACCESSES 1000000000

// Define a linked list node type with no data
typedef struct node {
  struct node* next;    // 8 bytes
  int data[30];         // 120 bytes
} node_t;

...
// Traverse array over and over again until we've visited `ACCESSES` elements
node_t* current = array;
for(int i=0; i<ACCESSES; i++) {
  if(current == array + items) current = array;        // reached the end
  else ++current;                                       // next element
}
```
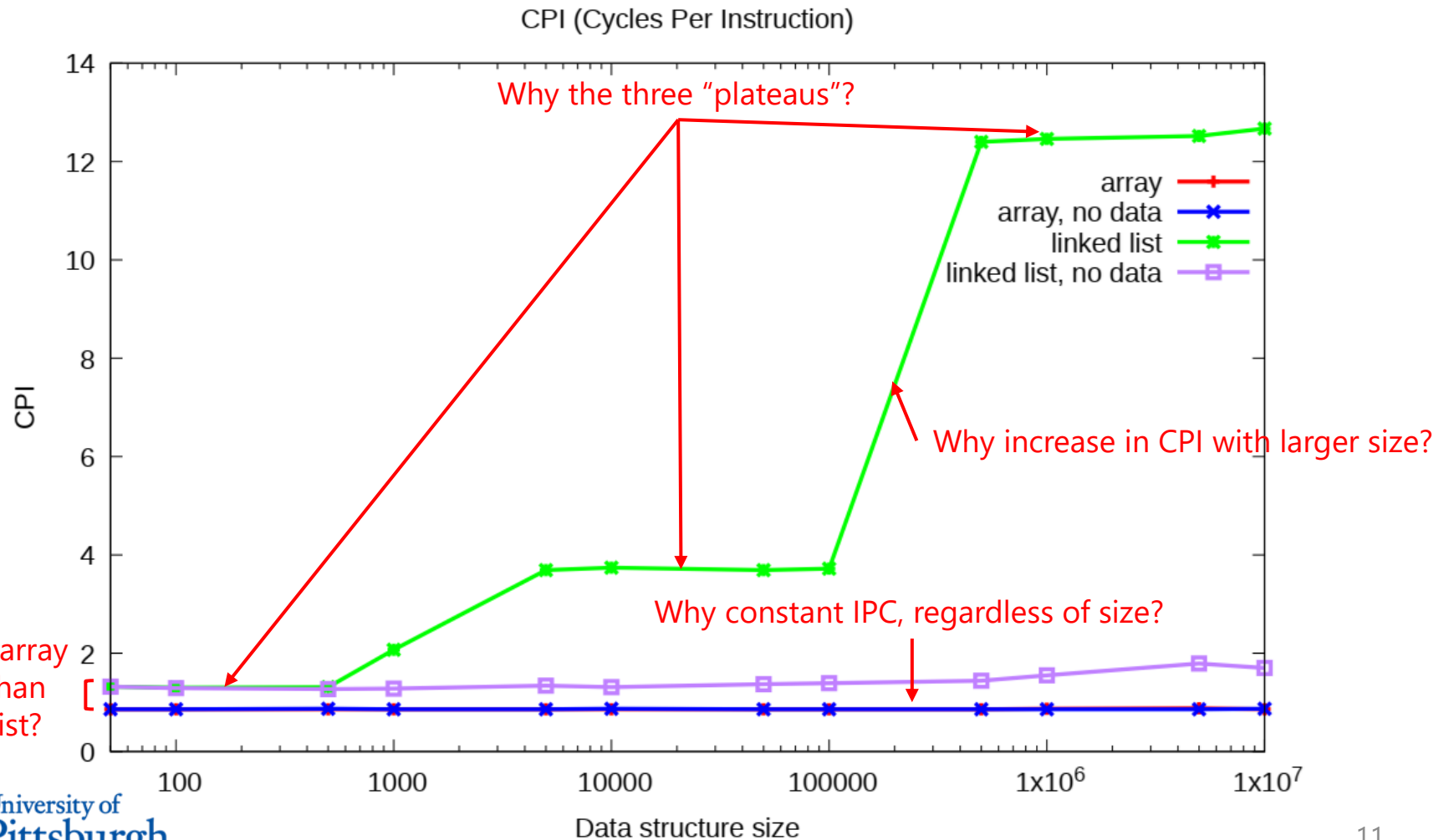
- Two CPU sockets. Each CPU:
  - Intel(R) Xeon(R) CPU E5-2640 v4
  - 10 cores, with 2 threads per each core (SMT)
  - L1 i-cache: 32 KB 8-way set associative (per core)
  - L1 d-cache: 32 KB 8-way set associative (per core)
  - L2 cache: 256 KB 8-way set associative (per core)
  - L3 cache: 25 MB 20-way set associative (shared)
- Memory
  - 128 GB DRAM
- Information obtained from
  - "cat /proc/cpuinfo" on Linux server
  - "cat /proc/meminfo" on Linux server
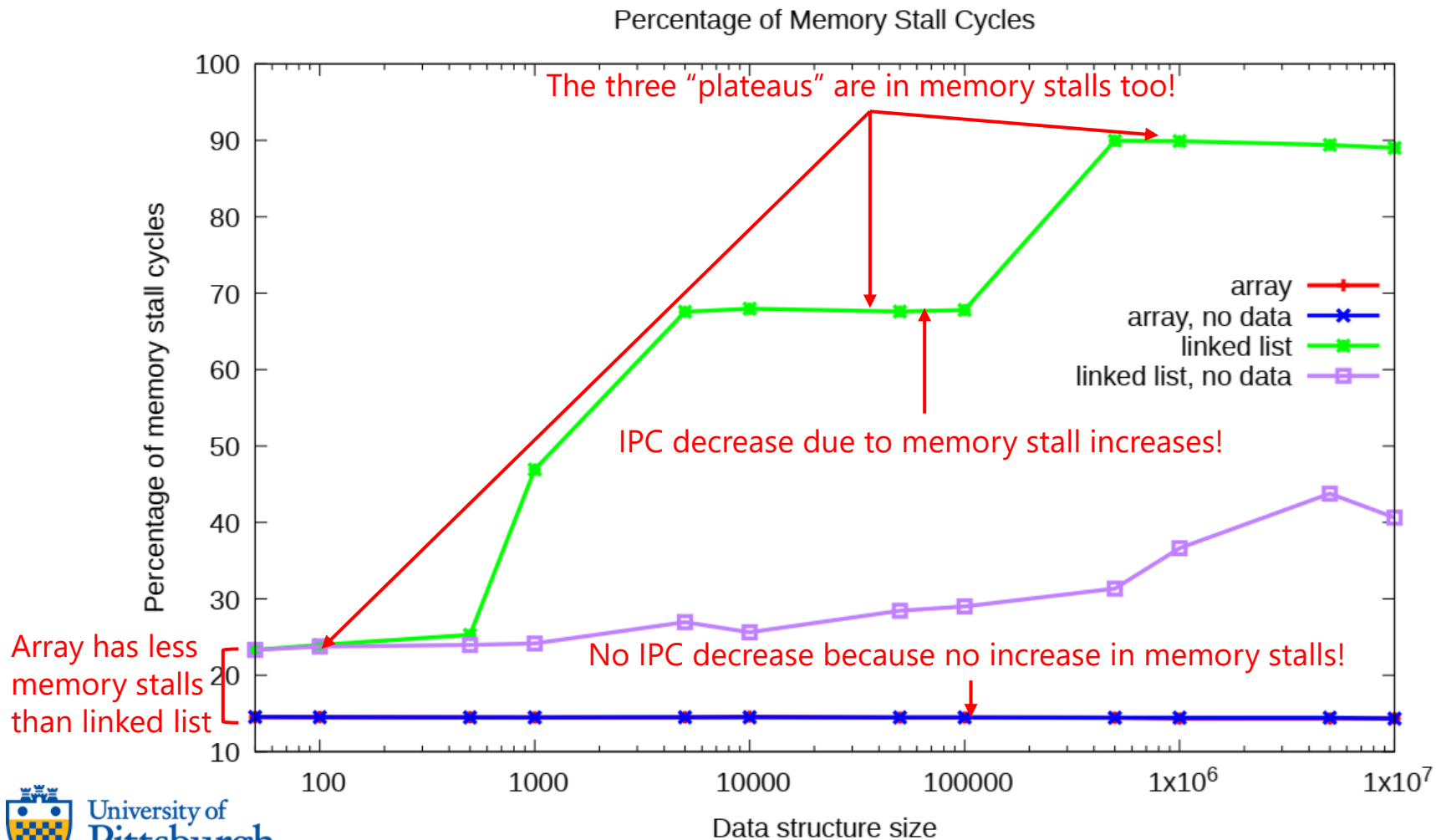 - https://en.wikichip.org/wiki/intel/xeon_e5/e5-2640_v4

University of Pittsburgh

- 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, …

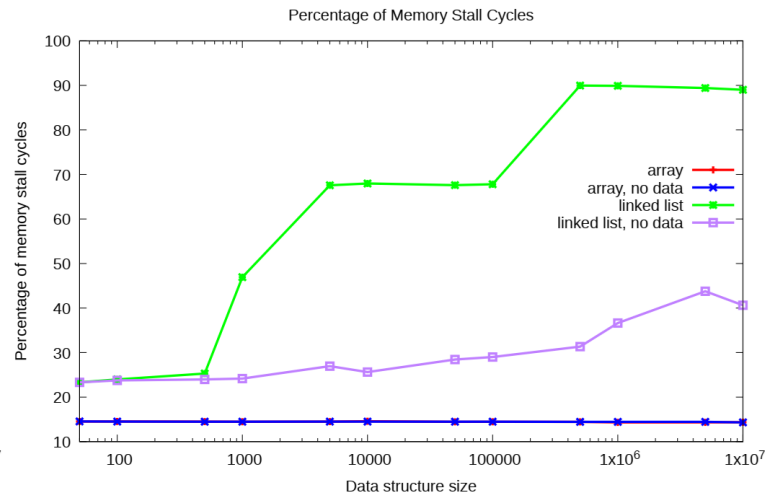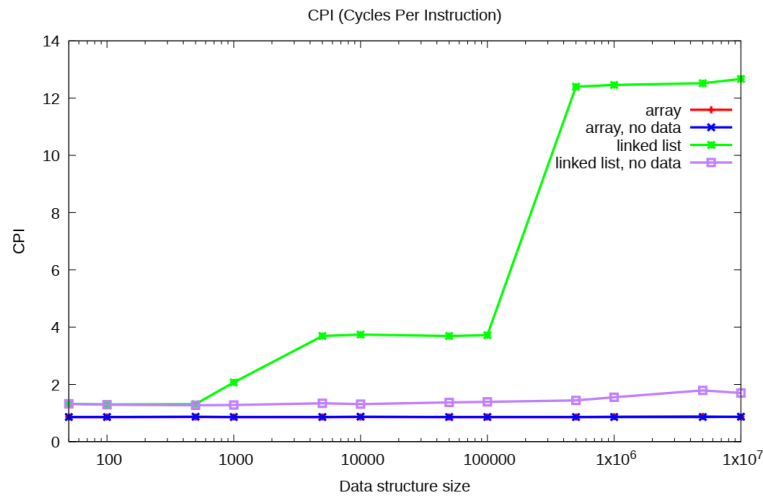- Linux **perf** utility summarizes this info in easy to read format
  - https://perf.wiki.kernel.org/index.php/Tutorial

University of Pittsburgh

CPI (Cycles Per Instruction)

Why the three "plateaus"?

Why increase in CPI with larger size?

Why constant IPC, regardless of size?

Why is array faster than linked list?

Legend:
- array
- array, no data
- linked list
- linked list, no data

CPI (y-axis)
Data structure size (x-axis)

Percentage of Memory Stall Cycles

The three "plateaus" are in memory stalls too!

IPC decrease due to memory stall increases!

No IPC decrease because no increase in memory stalls!

Array has less memory stalls than linked list

Legend:
- array
- array, no data
- linked list
- linked list, no data

Y-axis: Percentage of memory stall cycles

X-axis: Data structure size

University of Pittsburgh

- **Data structure performance is proportional to memory stalls**
  - o Applies to other data structures such as trees, graphs, …
- In general, **more data** leads to **worse performance**
  - o But why? Does more data make MEM stalls longer? (Hint: yes)
  - o 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

- SRAM uses a loop of NOT gates to store a single bit
- This is usually called a 6T SRAM cell since it uses… 6 Transistors!
- **Pros:**
  - **Very fast** to read/write
- **Cons:**
  - Volatile (loses data without power)
  - Relatively many transistors needed
    - -> **expensive**

WL

Pass-gates used for read and write

BL $\overline{BL}$
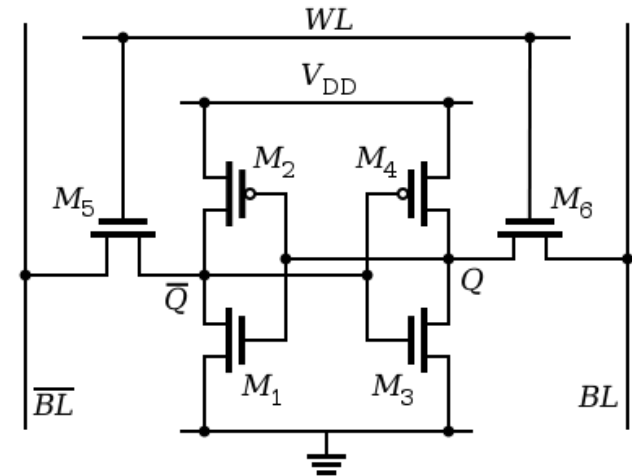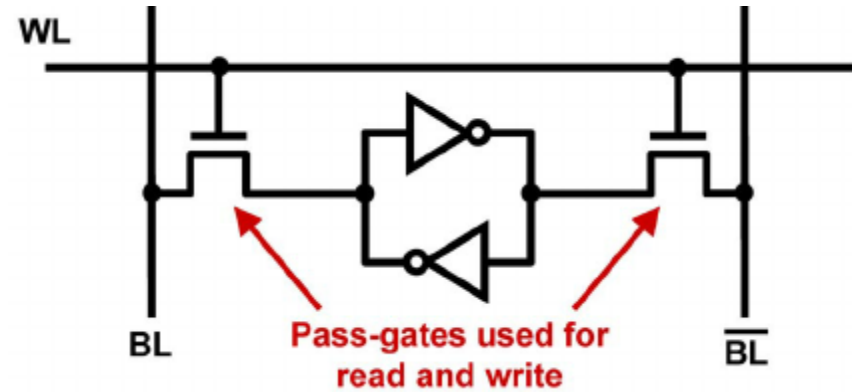
- DRAM uses **one** transistor and **one** capacitor
  - The bit is stored as a charge in the capacitor
  - Capacitor leaks charge over time
    - -> Capacitors must be periodically recharged
    - -> This is called **refresh**
    - **->** During refresh, DRAM can't be accessed
    - -> Also after read, 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 then SRAM)
  - **Slower access time**
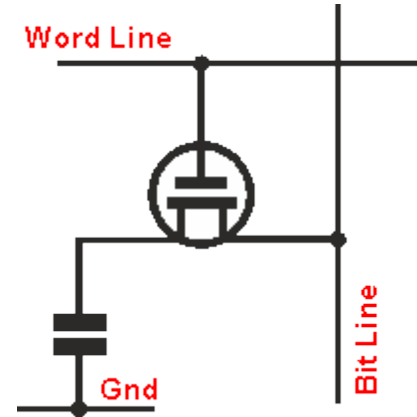
- Spinning platter coated with a ferromagnetic substance magnetized to represent bits
  - Has a mechanical arm with a head
  - Reads by placing arm in correct cylinder, and waiting for platter to rotate

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

University of **Pittsburgh**

- 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

University of **Pittsburgh**

Memory/storage technologies

	Volatile		Nonvolatile	
	SRAM	**DRAM**	**HDDs**	**Flash**
Speed	**FAST**	OK	**SLOW**	Pretty good!
Price	**Expensive**	OK	**Cheap!**	Meh
Power	**Good!**	Meh	**Bad**	OK
Durability	**Good!**	**Good!**	**Good!**	OK
Reliability	**Good!**	Pretty good!	Meh	Pretty good!

I'm using **Durability** to mean "how well it **holds data** after repeated use."
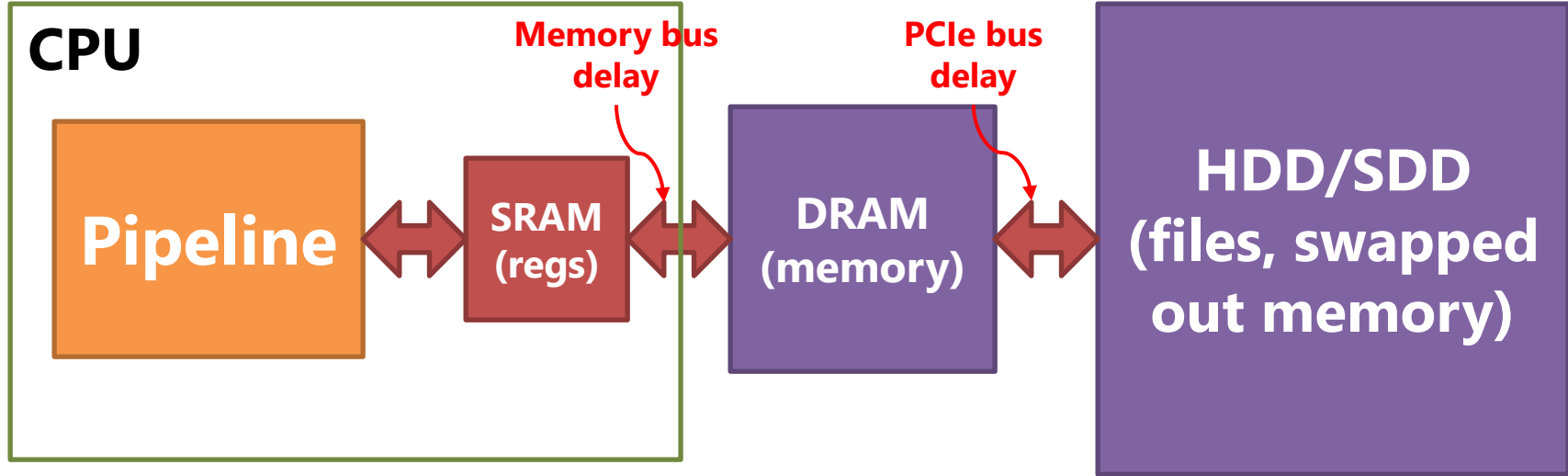I'm using **Reliability** to mean "how resistant is it to external shock."

- The **faster** the memory the more **expensive** and **lower density**.

- The **slower** the memory the **less expensive** and **higher density**.

- Thus, memory is constructed as a hierarchy:
  - Fast and small memory at the upper levels
  - Slow and big memory at the lower levels

- And also data is stored hierarchically:
  - Frequently accessed data is stored in the fast, upper levels
  - Infrequently accessed data is stored in the slow, lower levels
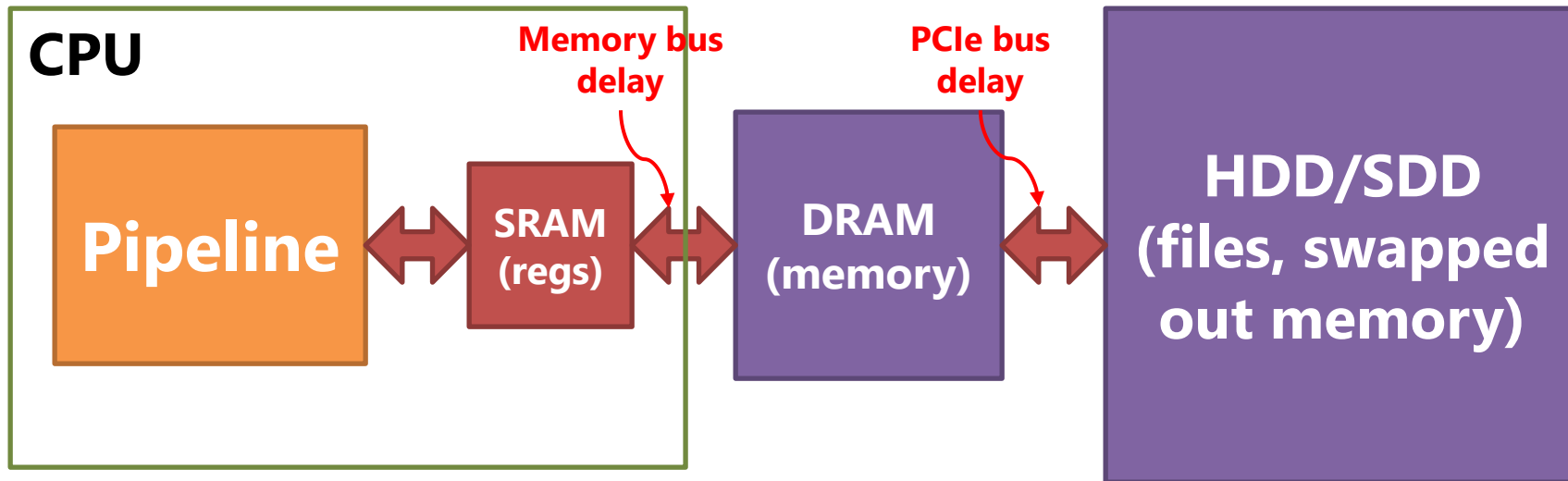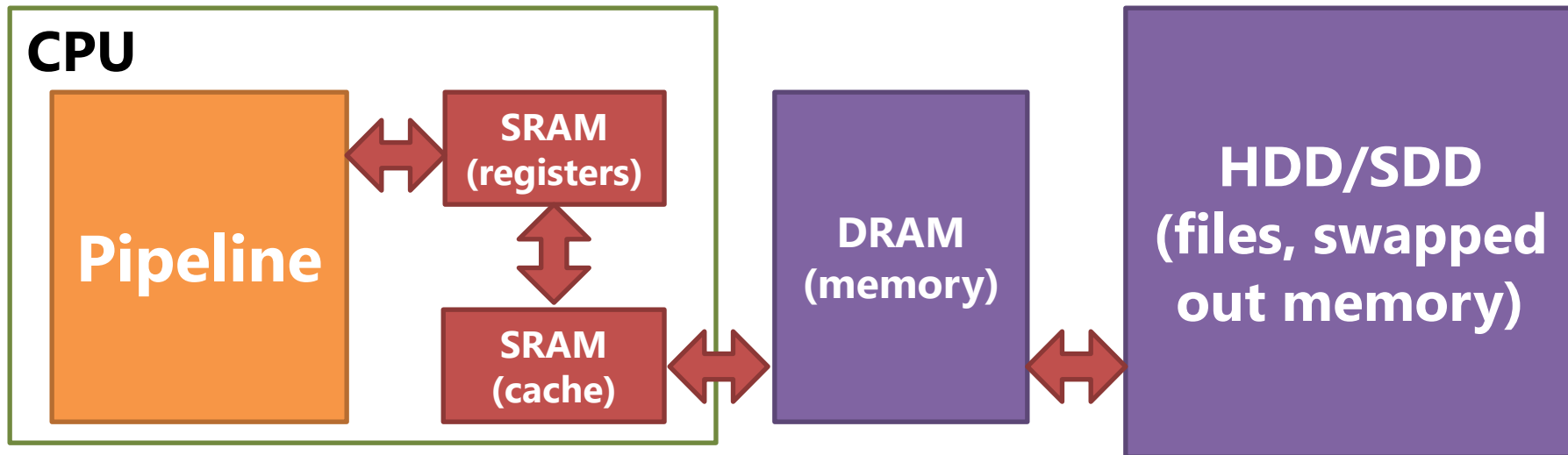
University of Pittsburgh

The memory hierarchy

- Use fast memory (SRAM) to store frequently used data inside the CPU
- Use slow memory (e.g. DRAM) to store rest of the data outside the CPU

CPU

Memory bus delay

PCIe bus delay

Pipeline ⬌ **SRAM (regs)** ⬌ **DRAM (memory)** ⬌ **HDD/SDD (files, swapped out memory)**

- Registers are used frequently for computation so are stored in SRAM
- Memory pages used frequently are stored in DRAM
- Memory pages used infrequently are stored in HDD/SDD (in swap space)
- Note: Memories outside CPU suffers from bus delay as well
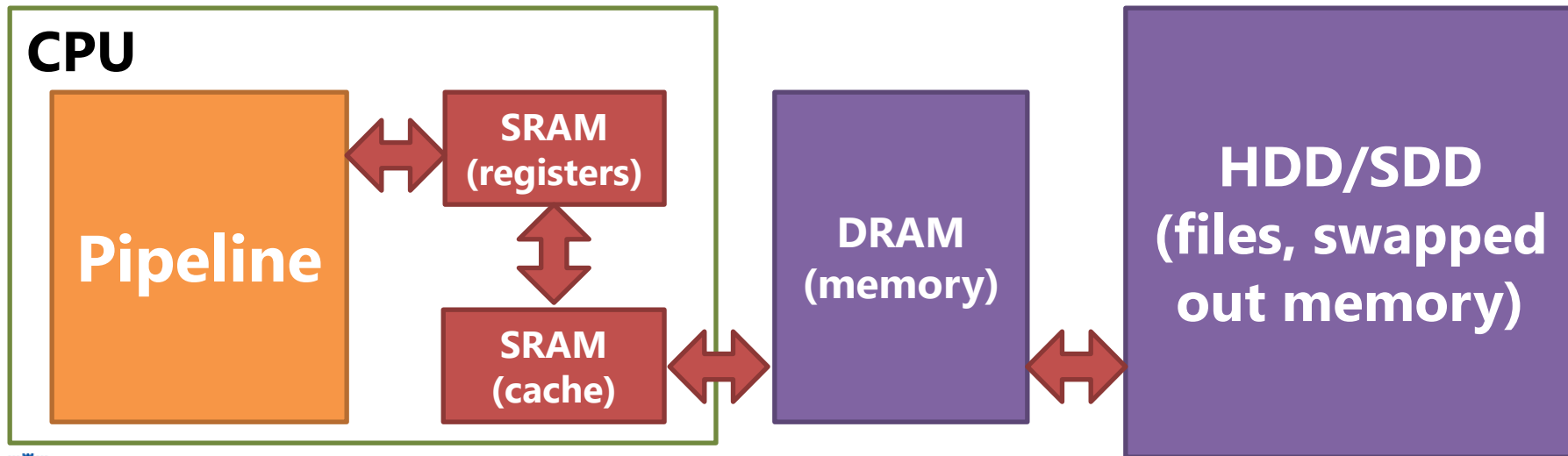
University of **Pittsburgh**

- Use fast memory (SRAM) to store frequently used data inside the CPU
- Use slow memory (e.g. DRAM) to store rest of the data outside the CPU

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

- Use fast memory (SRAM) to store frequently used data inside the CPU
- Use slow memory (e.g. DRAM) to store rest of the data outside the CPU

- Drawback: Memory access is much slower compared to registers
- Q: Can we make memory access speed comparable to register access?
  - How about storing frequently used memory data in SRAM too?
  - This is called **caching**. The hardware structure is called a **cache**.

University of
Pittsburgh

24

Caching

- **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

CPU

Pipeline	⟷	SRAM (registers)
		⟷
		SRAM (cache)

SRAM (cache) ⟷ DRAM (memory) ⟷ HDD/SDD (files, swapped out memory)

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

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

- **Bad**: MEM stage has **variable latency**
  o Typically only a few cycles if cache hit
  o More than a 100 cycles if cache miss! (Processor must go all the way to DRAM!)
  o Makes performance very **unpredictable**
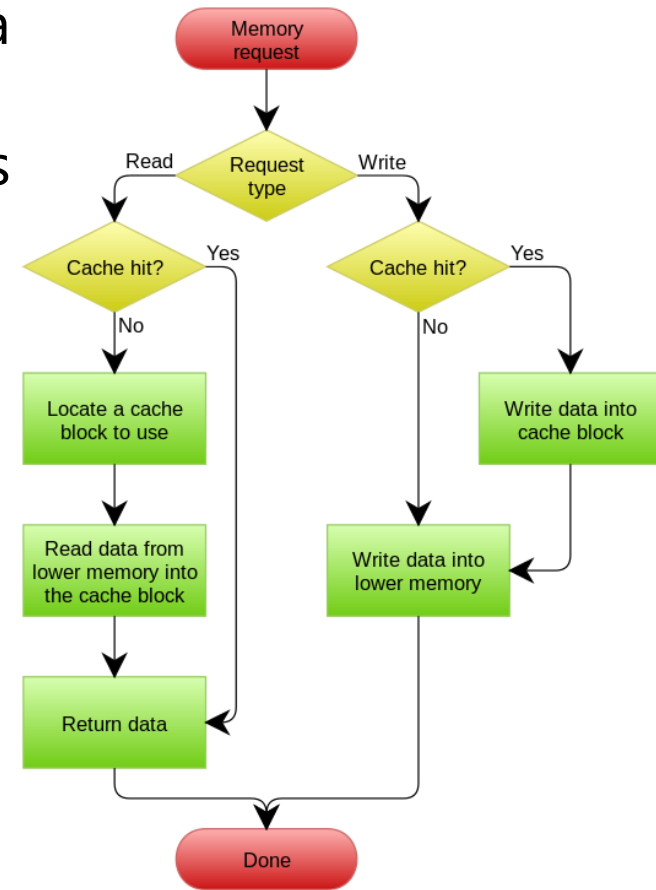
- Caching works because there is **locality** in program data accesses
  - **Temporal locality**
    - Same data is accessed many times in succession
    - **1st access** will **miss** but following accesses will **hit** in the cache
  - **Spatial locality**
    - Many data items that are spatially close are accessed together
    - E.g. fields of the same object, elements in an array, ...
    - Access to **1st item** will **miss** but bring in an entire **cache block**
    - Accesses to **other items** within same cache block will **hit**
  - Locality, like ILP, can only be exploited if it is there
- So does that mean having larger cache blocks is always better?
  - No, if block size is larger than spatial locality present in program
 - Then the space to bring in the extra unused data is wasted

- **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
  - Subsequent accesses will be hits since now data is fetched into cache
    - Unless it is replaced to make space for more frequently used data

- **Capacity miss**
  - Miss suffered when data is accessed for the **second or third times**
  - This miss occurred because **data was replaced** to make space
  - If there had been more capacity, miss wouldn't have happened
  - Capacity decides how much **temporal locality** you can leverage

- Reducing capacity misses is straightforward
  - Increase capacity, that is cache size!

- But how do you reduce cold misses? Is it even possible?
  - Yes! By taking advantage of **spatial locality**.
  - Have a large cache block so you bring in other items on a miss
  - Those other items may be accessed for the first time but will hit!

- Large cache blocks can …
  - Potentially exacerbate capacity misses
    (with no spatial locality, can store less data items in same capacity)
  - Potentially reduce cold misses
    (given some spatial locality, can bring in more data on a miss)
 - → Each program has a sweet spot. Architects choose a compromise.

- Uber big!

- On the right is a diagram of the Xeon Sandy Bridge CPU used in kernighan.cs.pitt.edu.
  o More cache real estate compared to cores!
  o A cache miss is that painful.

- But having a big cache comes with its own set of problems
 o Cache itself gets slower

University of Pittsburgh
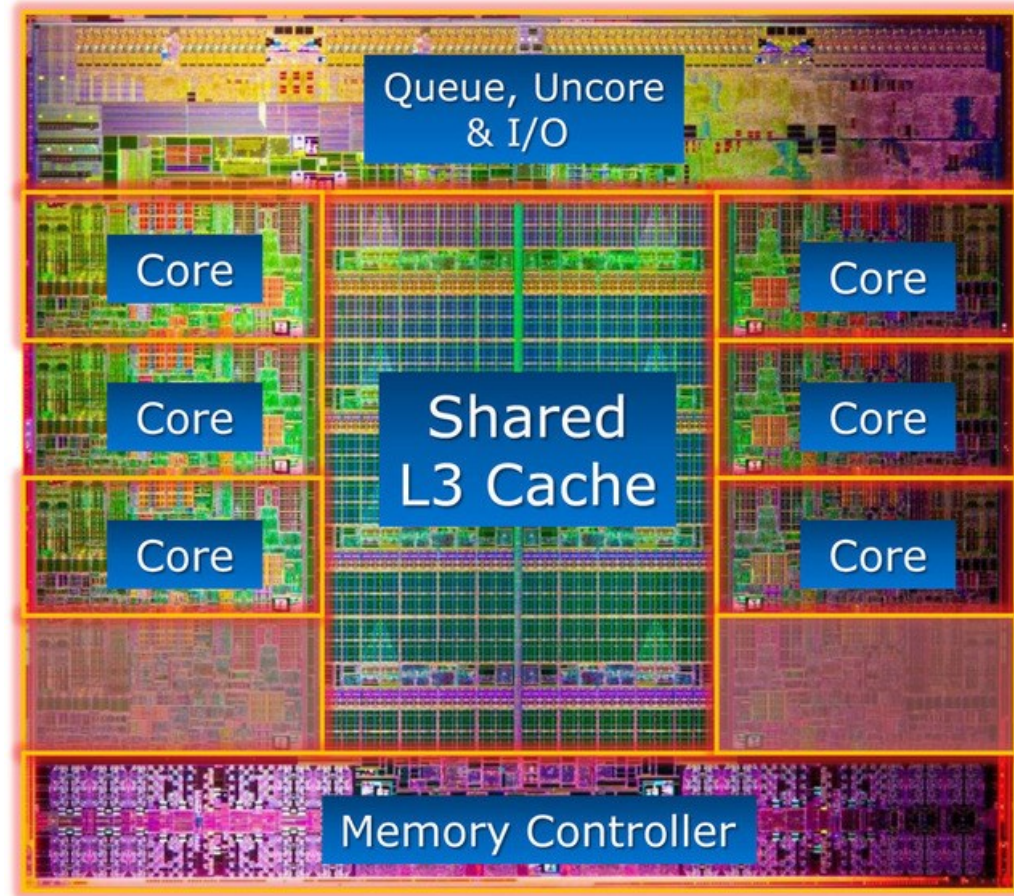
- Below is a diagram of a Nehalem CPU (predecessor of Sandy Bridge)

- 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"?
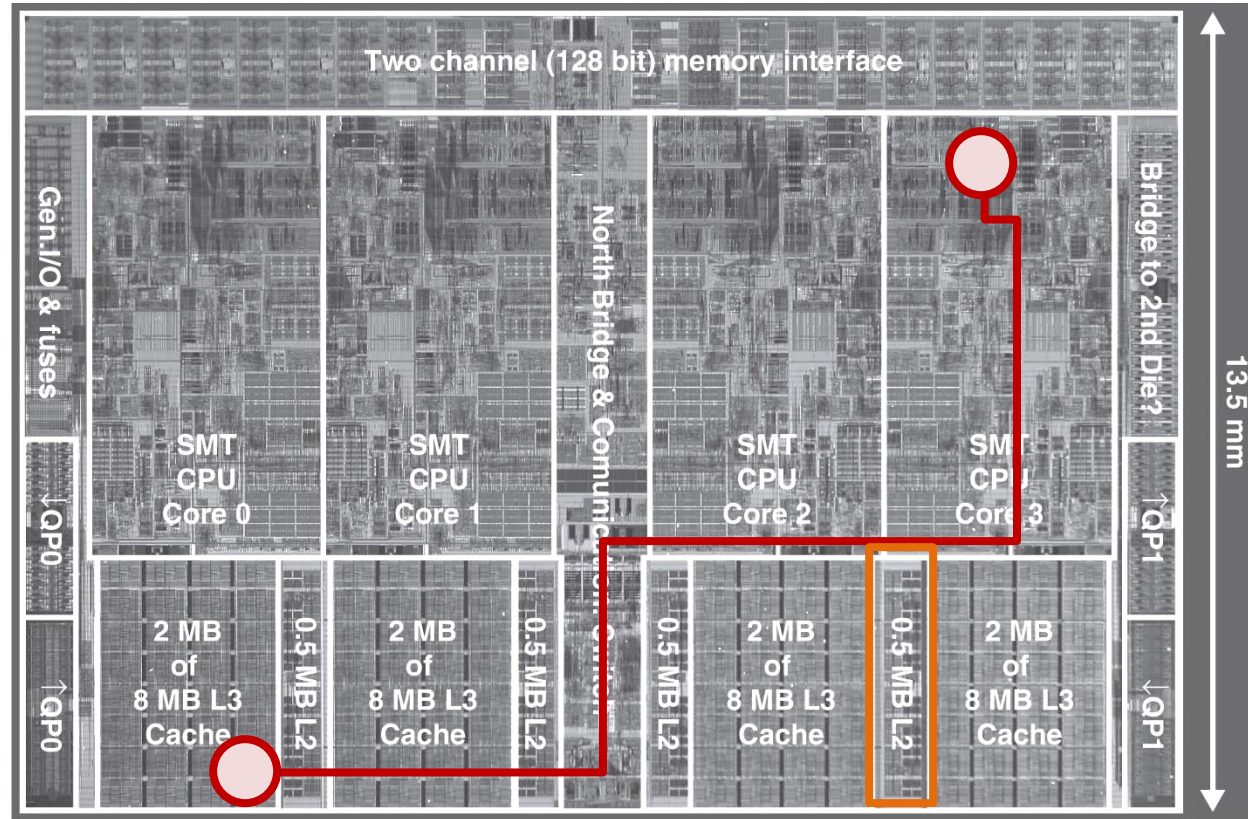
Multi-level Caching

- This is the structure of the kernighan.cs.pitt.edu Xeon CPU:

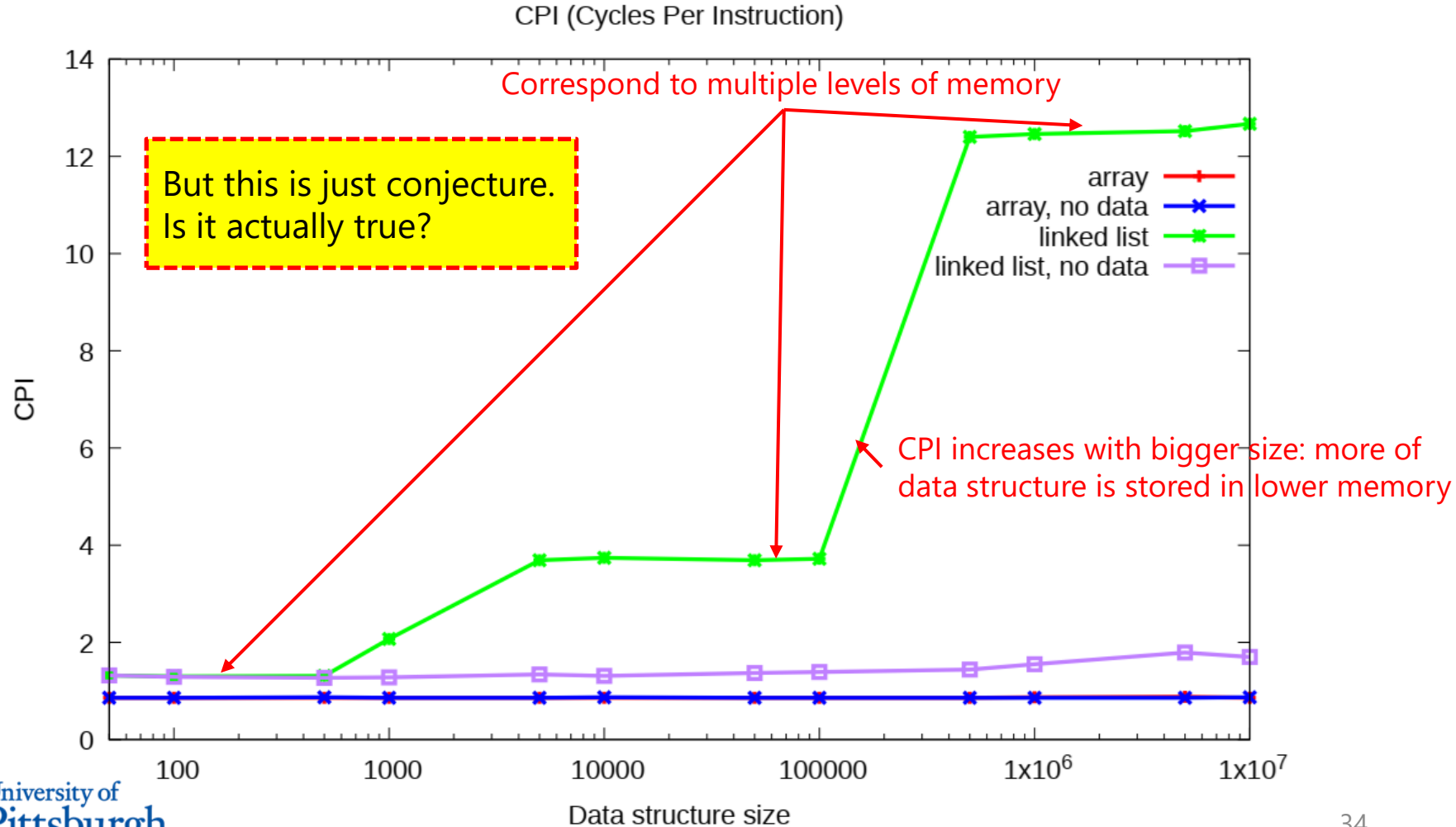| L1 I-Cache	L1 D-Cache
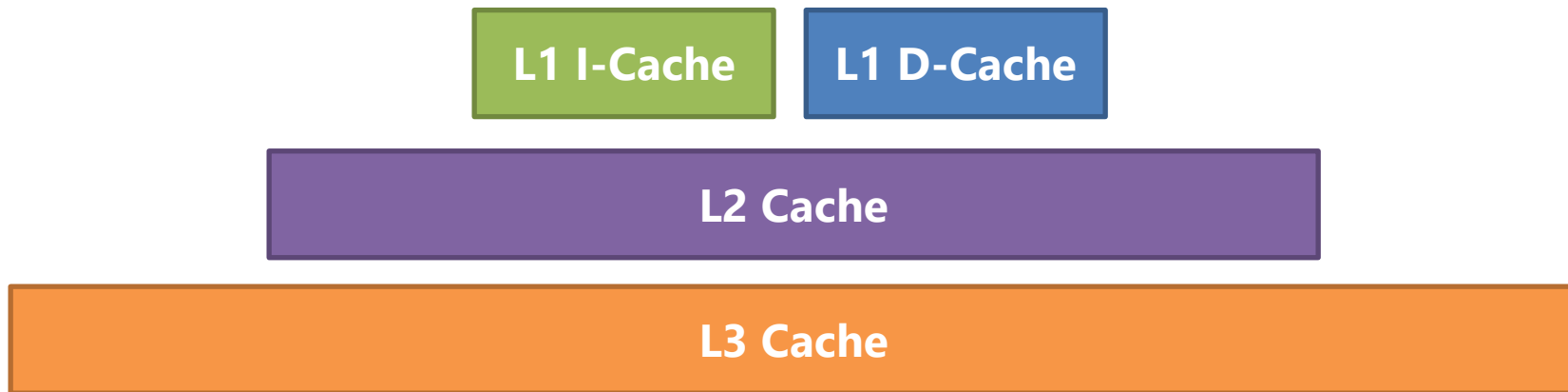
L2 Cache

L3 Cache

- 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 memory access.
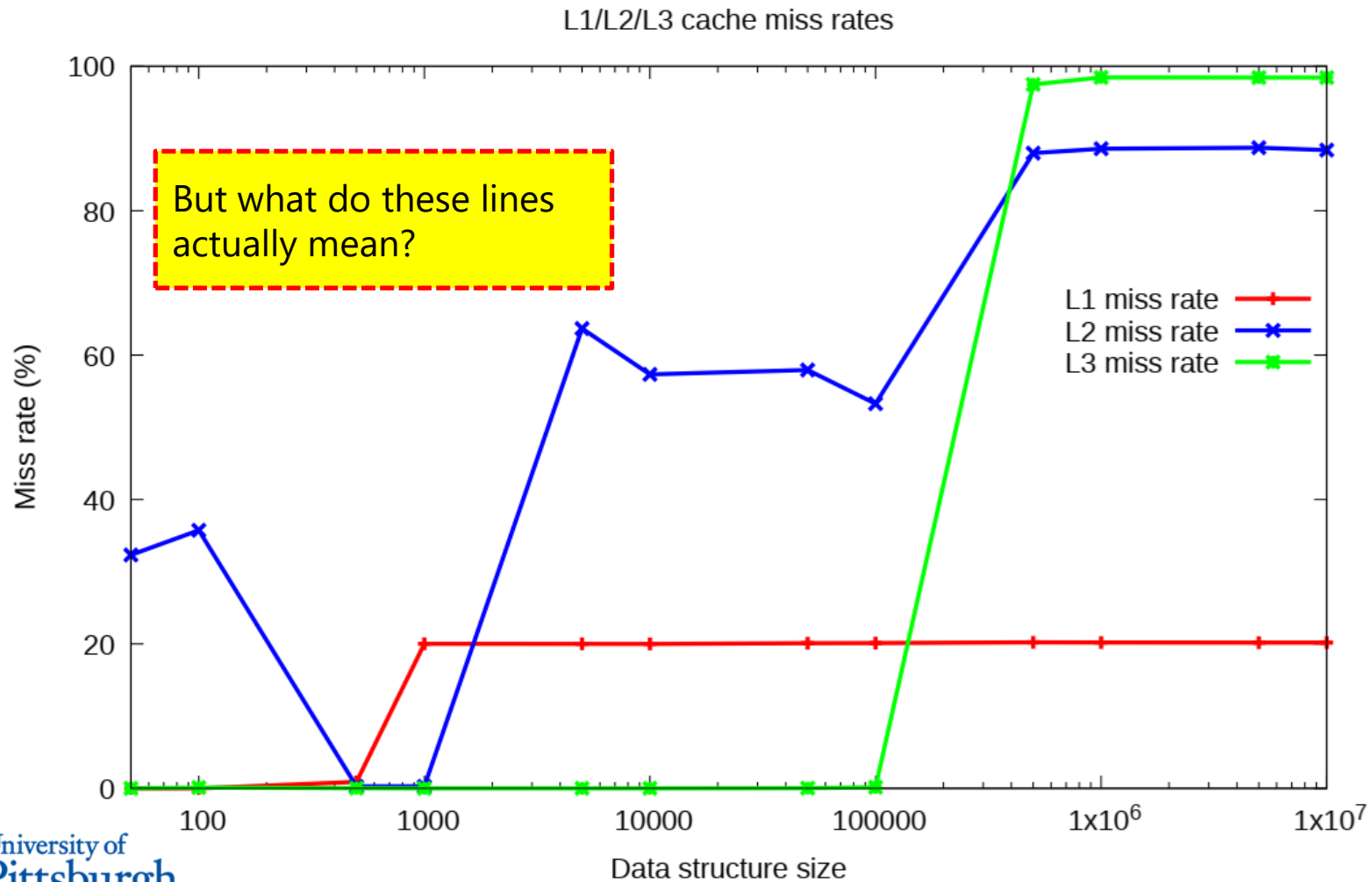- Allows performance to degrade gracefully

- 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
- Notice the gradual increase in cycles then the big jump to memory
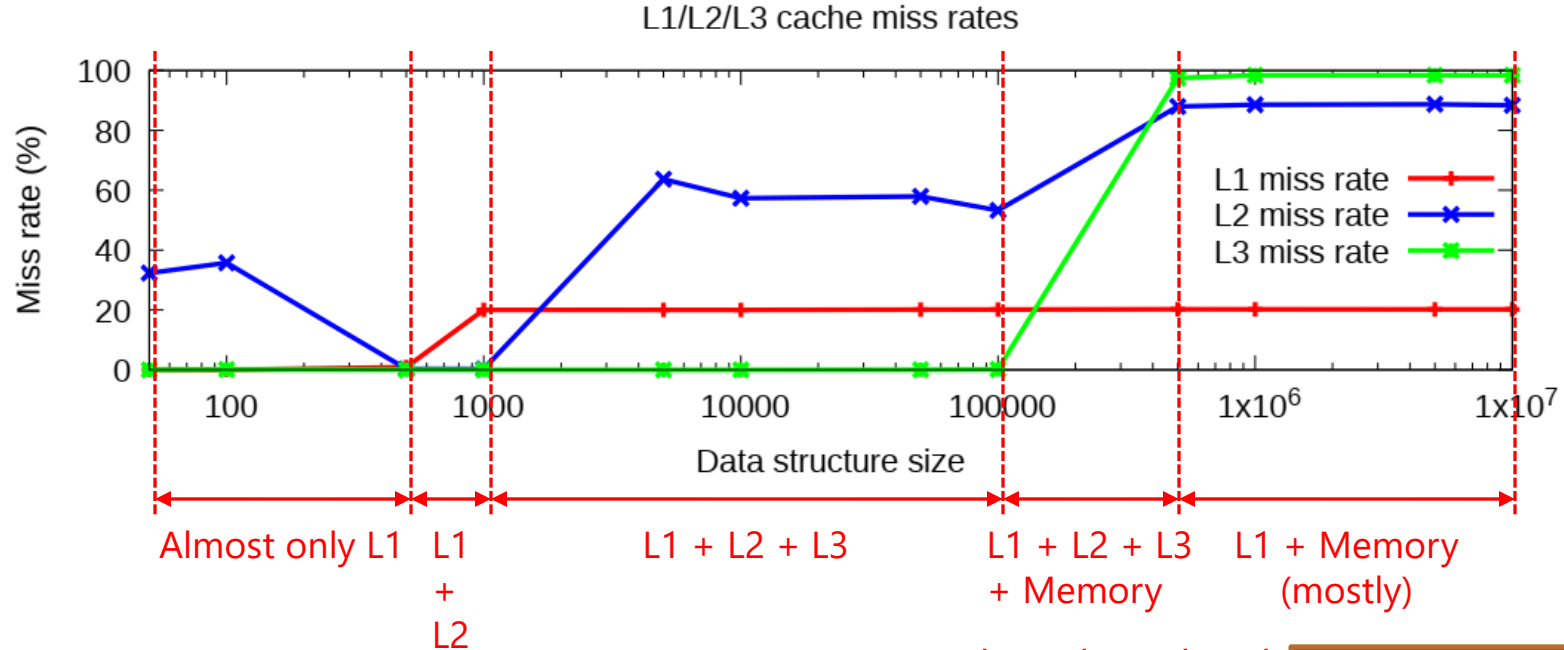 - That's why some processors have an L4 cache

CPI (Cycles Per Instruction)

Correspond to multiple levels of memory

But this is just conjecture.
Is it actually true?

array
array, no data
linked list
linked list, no data

CPI increases with bigger size: more of
data structure is stored in lower memory

CPI

Data structure size

University of Pittsburgh

34

L1/L2/L3 cache miss rates

But what do these lines actually mean?

L1/L2/L3 cache miss rates

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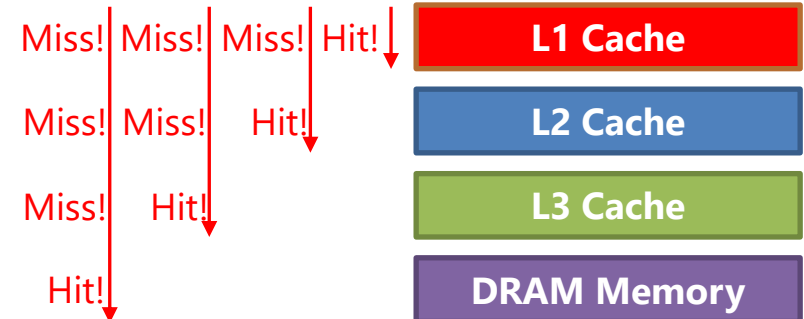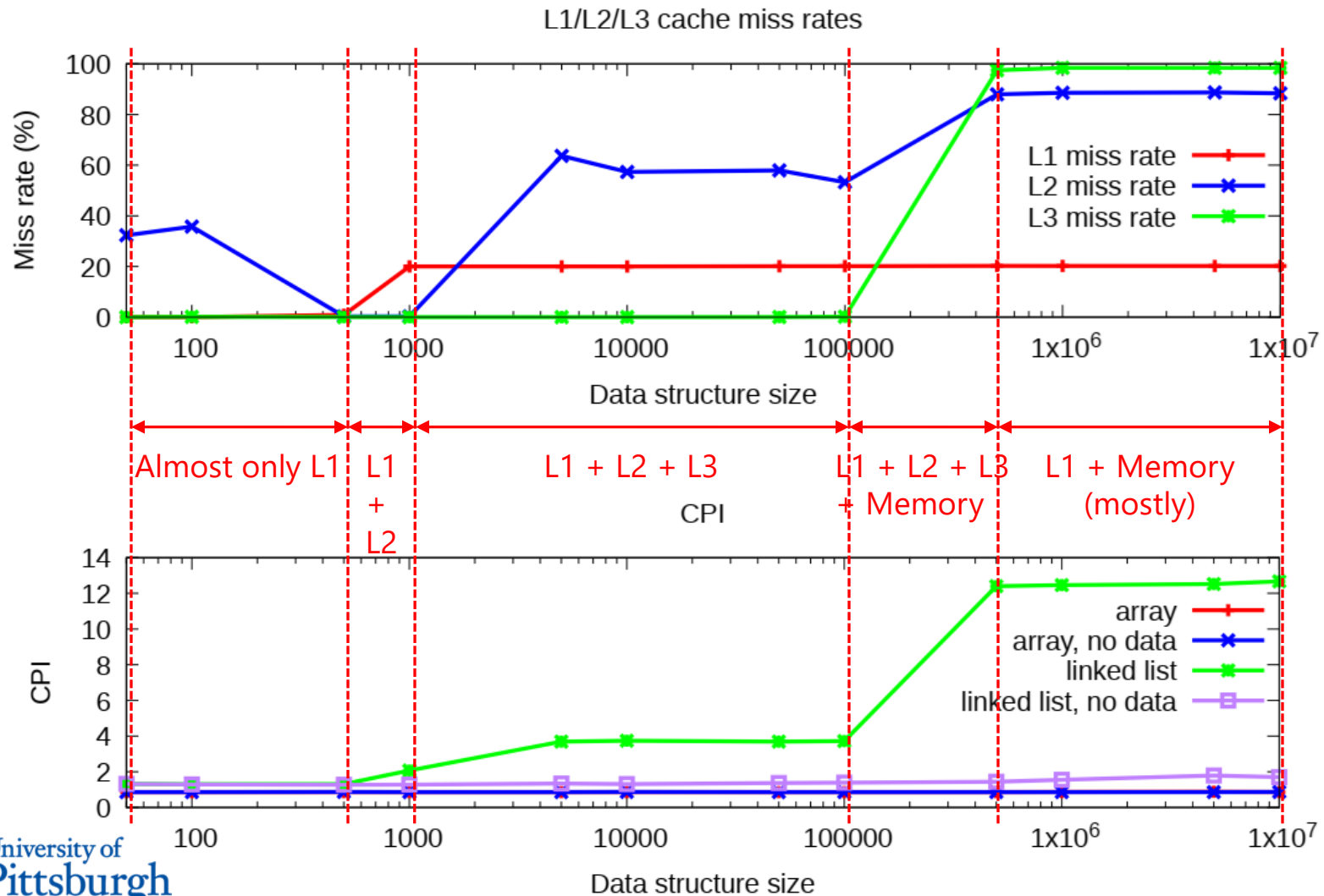
Miss! Miss! Miss! Hit!
Miss! Miss! Hit!
Miss! Hit!
Hit!

L1 Cache

L2 Cache

L3 Cache

DRAM Memory

University of Pittsburgh

36

L1/L2/L3 cache miss rates

Almost only L1   L1 + L2   L1 + L2 + L3   L1 + L2 + L3 + Memory   L1 + Memory (mostly)

L1/L2/L3 cache miss rates

Almost only L1 | L1 + L2 | L1 + L2 + L3 | L1 + L2 + L3 + Memory | L1 + Memory (mostly)
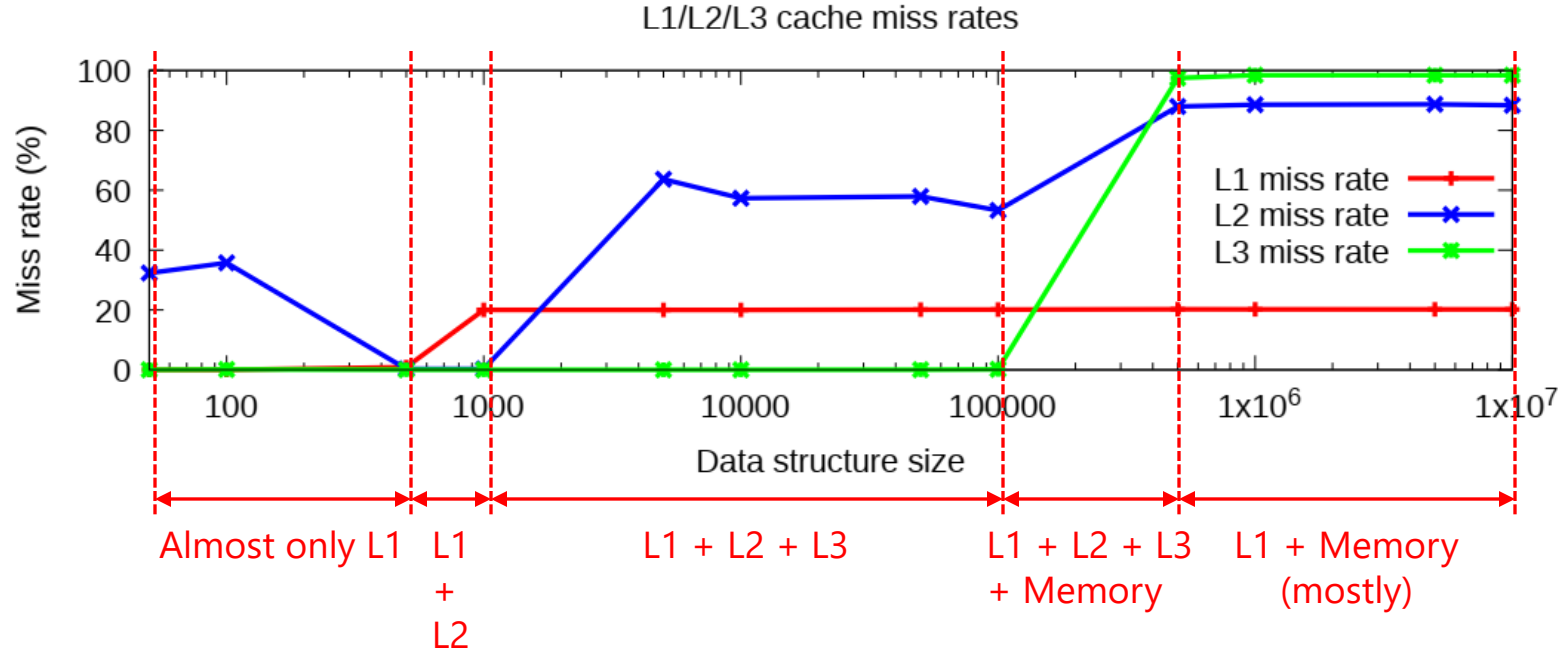
- Why did L1 cache miss rate increase between 500 – 1000 nodes?
- Why did L2 cache miss rate increase between 1000 – 5000 nodes?
- Why did L3 cache miss rate increase between 100 k – 500 k nodes?

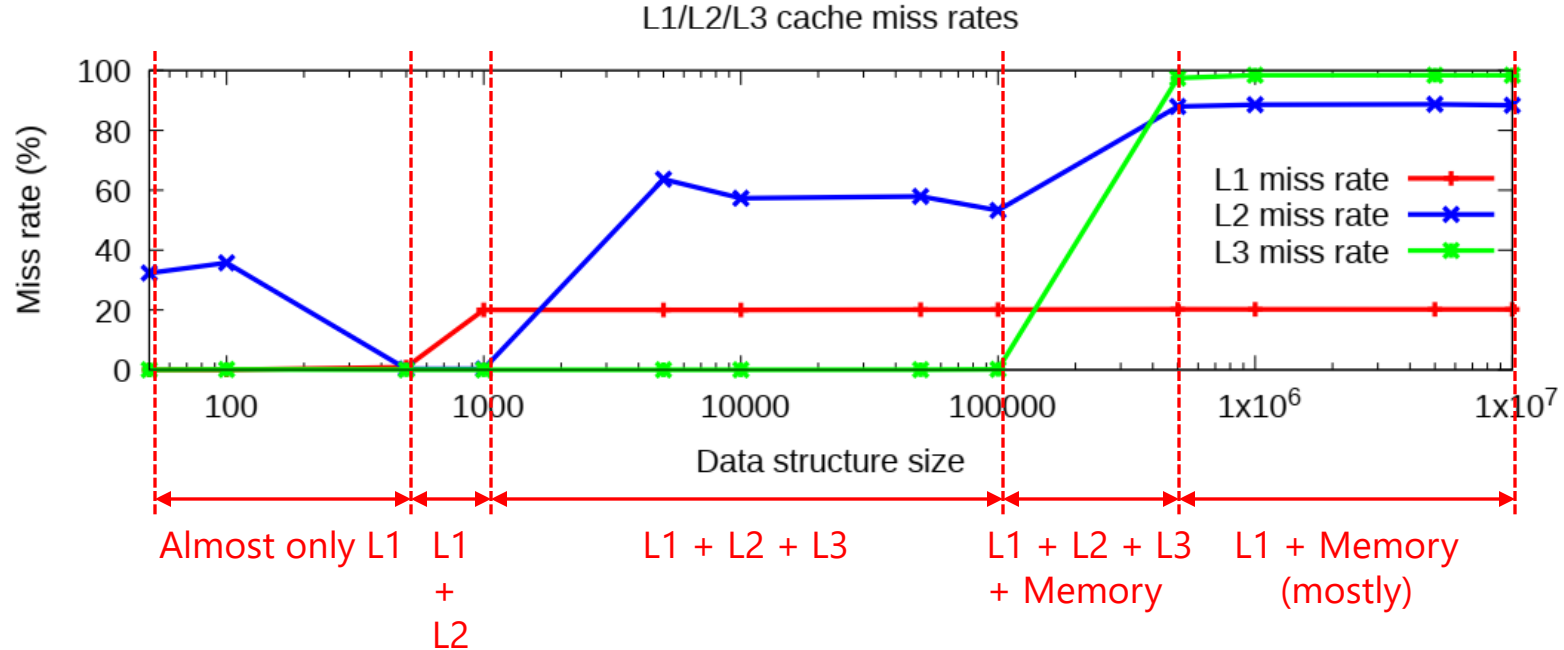- The size of a node is 128 bytes:
```
typedef struct node {
    struct node* next;      // 8 bytes
    int data[30];           // 120 bytes
} node_t;
```
- Why did L1 cache miss rate increase between 500 – 1000 nodes?
  - L1 d-cache size is 32 KB
  - Number of nodes that can fit = 32 KB / 128 = 256
  - At 500 nodes, L1 cache overflows and misses start to become significant
- Why did L2 cache miss rate increase between 1000 – 5000 nodes?
  - L2 cache size is 256 KB
  - Number of nodes that can fit = 256 KB / 128 = 2048
- Why did L3 cache miss rate increase between 100 k – 500 k nodes?
  - L3 cache size is 25 MB
  - Number of nodes that can fit = 25 MB / 128 ≈ 200 k

L1/L2/L3 cache miss rates

Almost only L1 · L1 + L2 · L1 + L2 + L3 · L1 + L2 + L3 + Memory · L1 + Memory (mostly)

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

[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;
  }
}
```

[objdump -S linked-list]

```
0000000000400739 <run>:
...
400741: mov 0x200920(%rip),%rax  # %rax = head
400748: mov %rax,-0x8(%rbp)  # current = head
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 = head
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 = current->next
400774: addl $0x1,-0xc(%rbp)  # i++
400778: cmpl $0x3b9ac9ff,-0xc(%rbp)  # i < ACCESSES ?
40077f: jle 400755  # jump to head of loop if less than
```

Within the for loop:

4 blue loads that hit in L1:
- current, i, head
- Read on each iteration so fetched into cache already and always hit

1 red load that misses in L1:
- current->next (next node)
- New node on each iteration so possibly not in the cache and miss
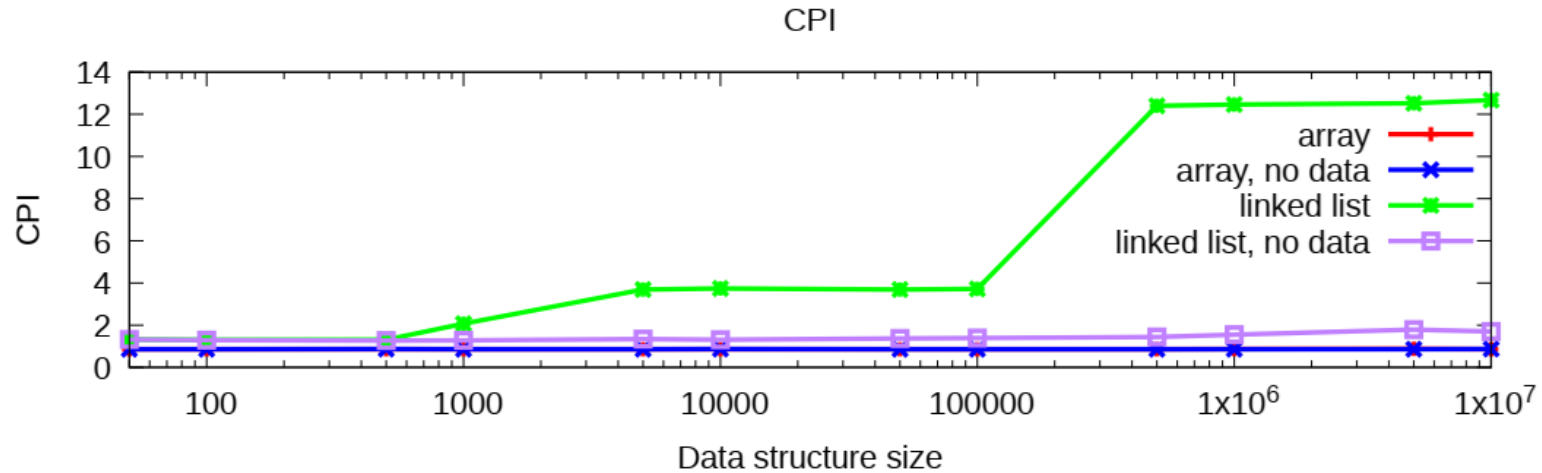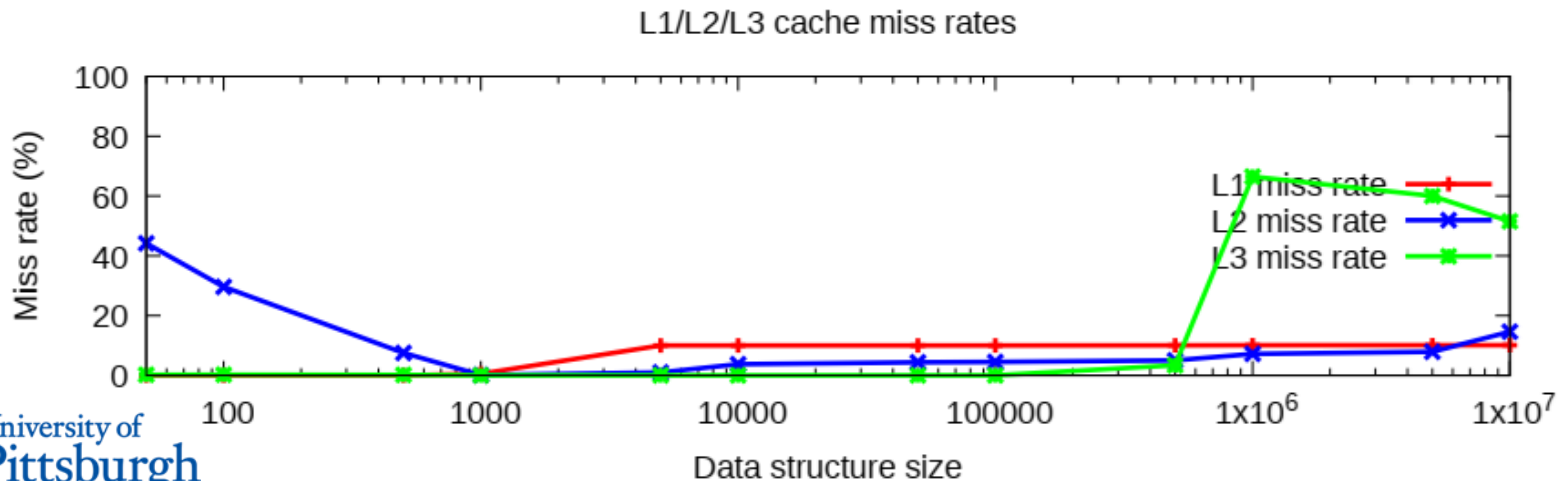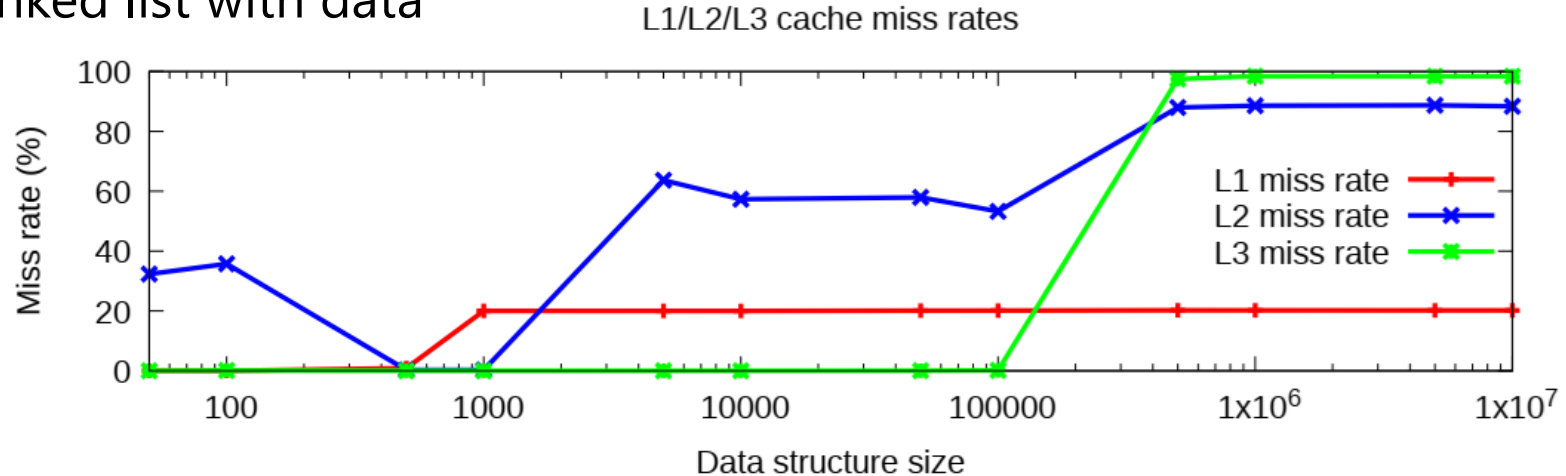
☛ Max 1 miss / 5 loads = 20% miss rate

CPI

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

L1/L2/L3 cache miss rates

- Linked list with data

- Linked list with no data
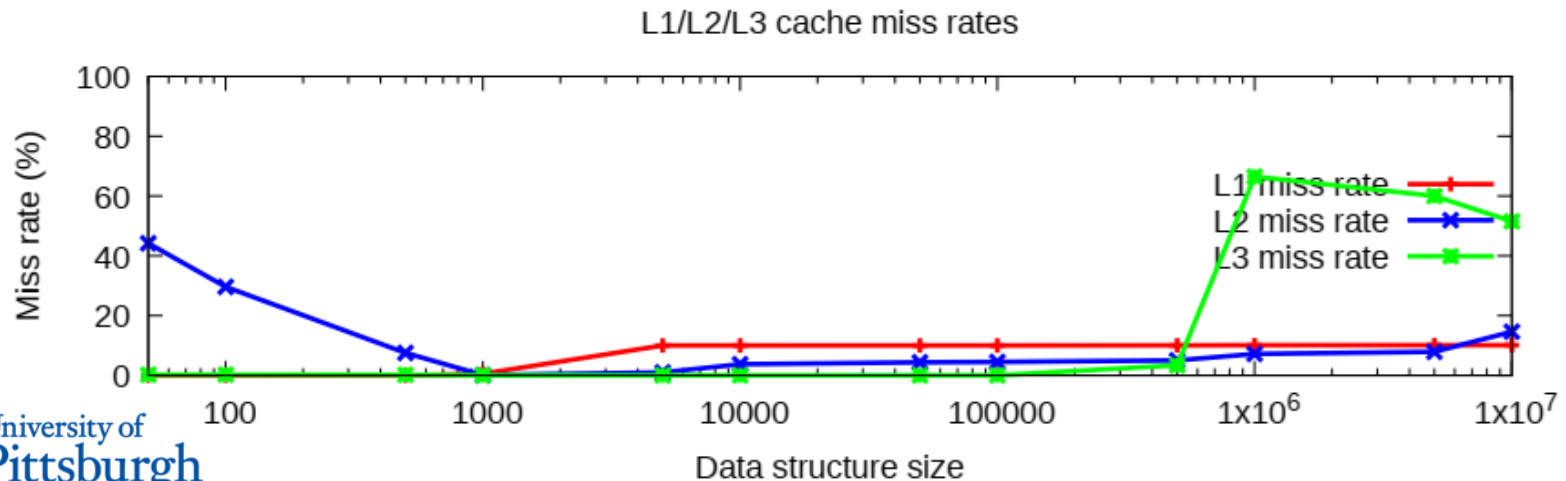
- The size of a node with no data is only 8 bytes:
```
typedef struct node {
    struct node* next;      // 8 bytes
    // no data
} node_t;
```

- Compared to 128 bytes with data, can fit in 16X more nodes in cache
  o More likely that a node will be present in cache next time you access it
  o Can take advantage of **temporal locality** over longer distances
  o But that alone cannot explain the wide discrepancy
    (w/o data with 10^6 nodes performs better than w/data with 10^4 nodes)

- What could explain the discrepancy?
  o Answer: **spatial locality** (neighboring nodes on same cache block)

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

```
void *create(void *unused) {
  for(int i=0; i<items; i++) {
    node_t* n = (node_t*)malloc(sizeof(node_t));
    ...
  }
}
```

- Each cache block is 64 bytes long in the Xeon E5 processor
  - Multiple adjacent nodes may be allocated on same cache line:

node 0			node 1			node 2	

  - Then even if block was replaced, node 0 may miss but nodes 1,2 will hit!
  - This is assuming there is some **spatial locality** in how malloc allocates

- 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
  o Now 8 elements are guaranteed to be on same cache line:

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

  o Even in a cold cache, only 1 out of 8 nodes miss (1/8 = 12.5% miss rate)
  o True regardless of capacity (even if cache contained only a single block)

CPI
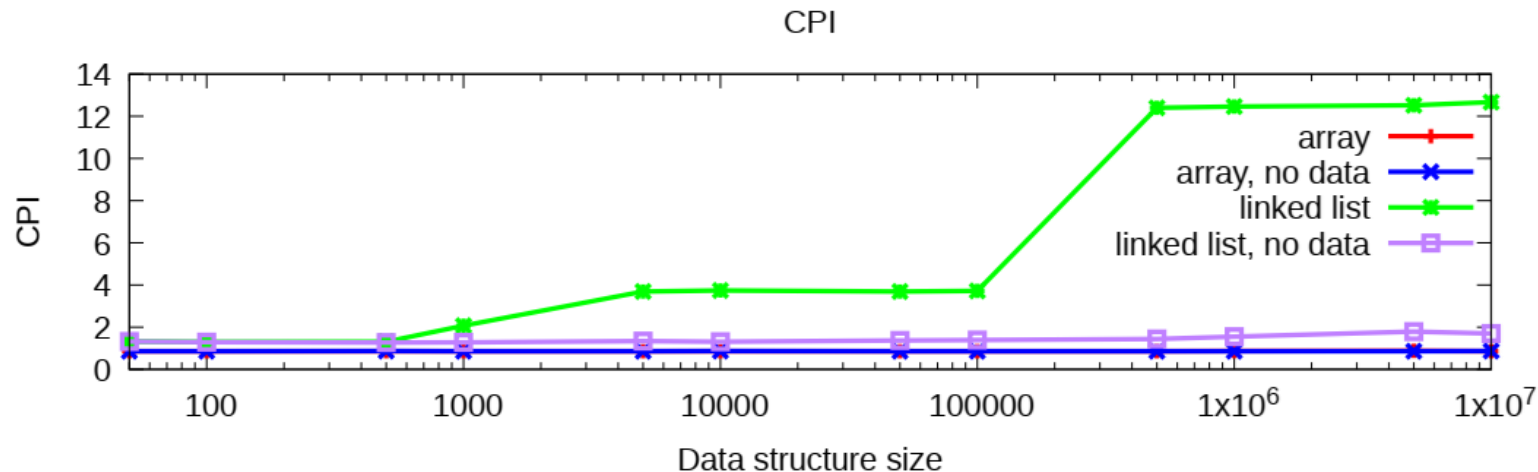
- Array, no data did very well as expected
  - The most spatial locality of the four benchmarks (contiguous access)
  - Smallest memory footprint so can also leverage temporal locality the best
- Array performed exactly the same as array, no data. How come?
  - No spatial locality since each node takes 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. ☺

University of
Pittsburgh