Memory Hierarchy

CS/COE 1541 (Fall 2020)
Wonsun Ahn

University of Pittsburgh

Experiment on kernighan.cs.pitt.edu

- 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
  - 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 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;
    }
  }
}
```

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 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;
}
```

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?

Code for array.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
}
```

kernighan.cs.pitt.edu specs

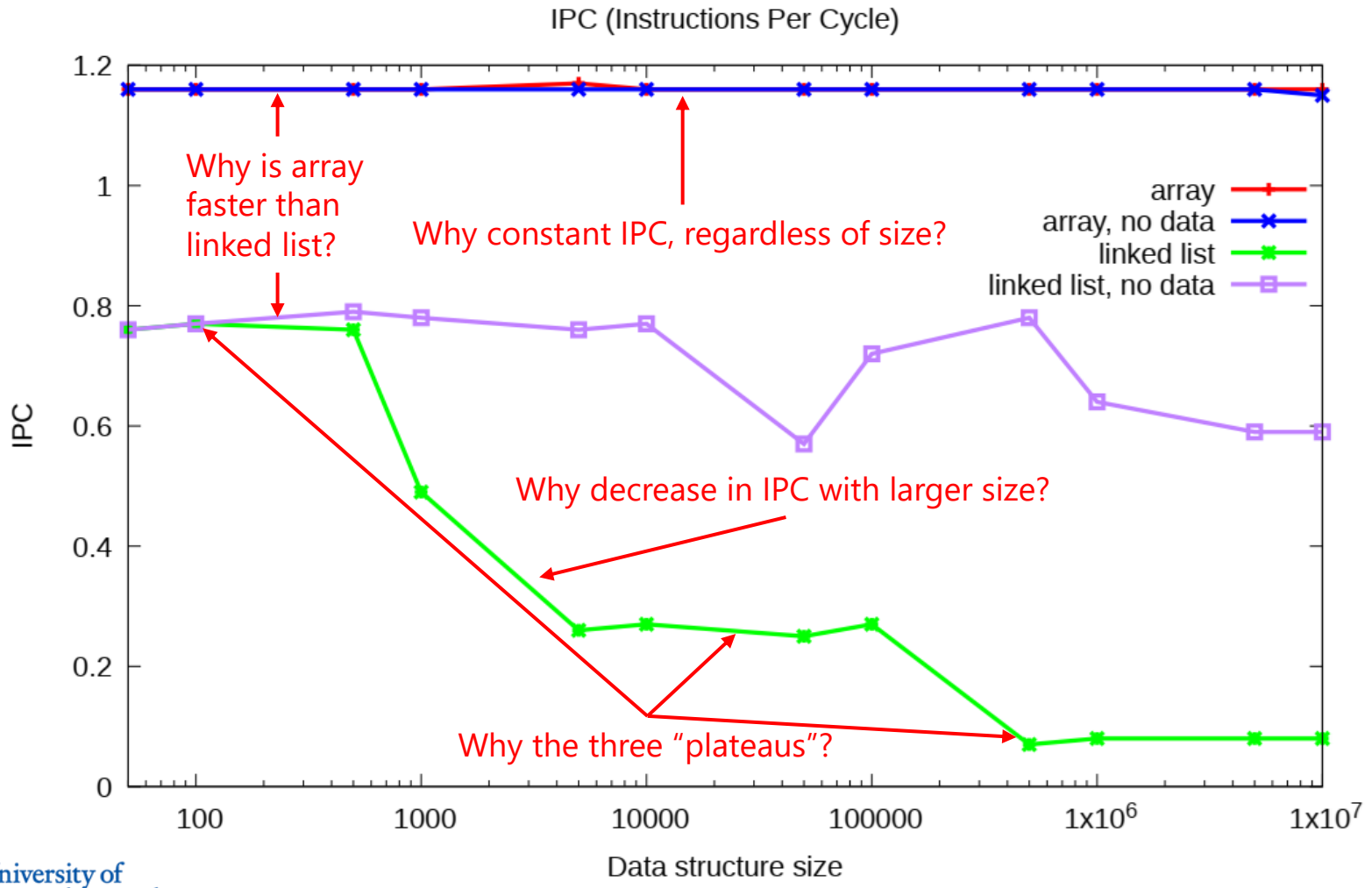
- Two CPU sockets. Each CPU:
  - Intel(R) Xeon(R) CPU E5-2640 v4
  - 10 cores, with 2 threads per each core (SMT)
  - L1 i-cache: 32 KB 8-way set associative (per core)
  - L1 d-cache: 32 KB 8-way set associative (per core)
  - L2 cache: 256 KB 8-way set associative (per core)
  - L3 cache: 25 MB 20-way 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, …

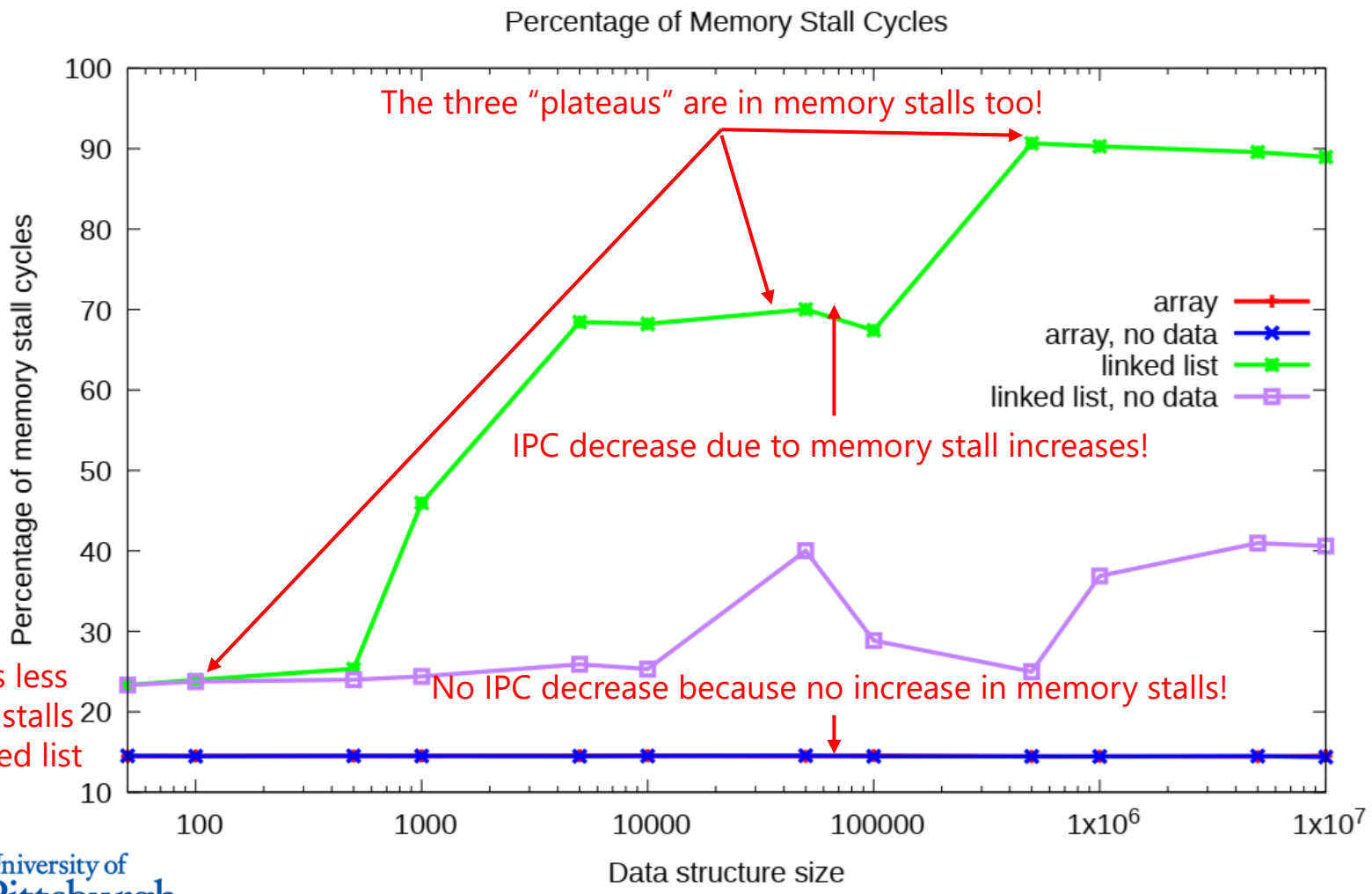
- Linux **perf** utility summarizes this info in easy to read format
  - https://perf.wiki.kernel.org/index.php/Tutorial

IPC (Instructions Per Cycle) Results



University of Pittsburgh

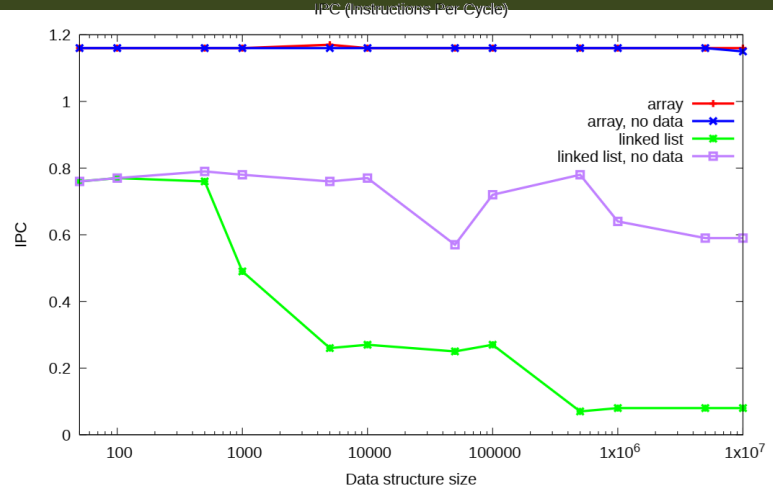
Memory Stall Cycle Percentage

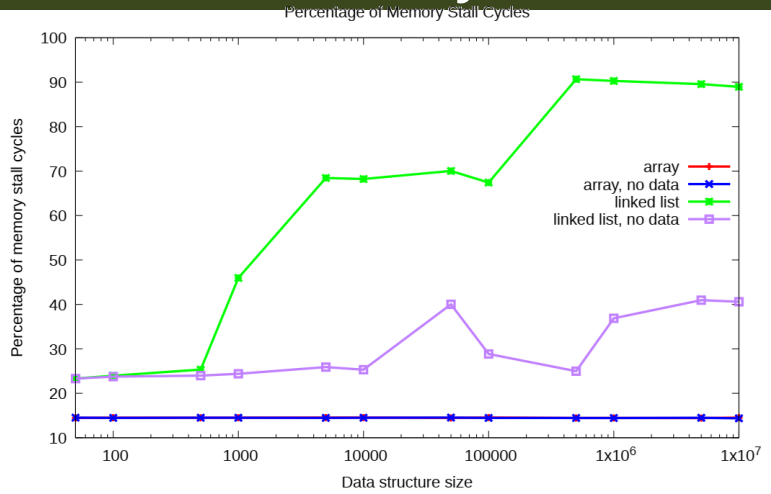


Array has less memory stalls than linked list

University of Pittsburgh

Data Structure Performance ∝ Memory Stalls



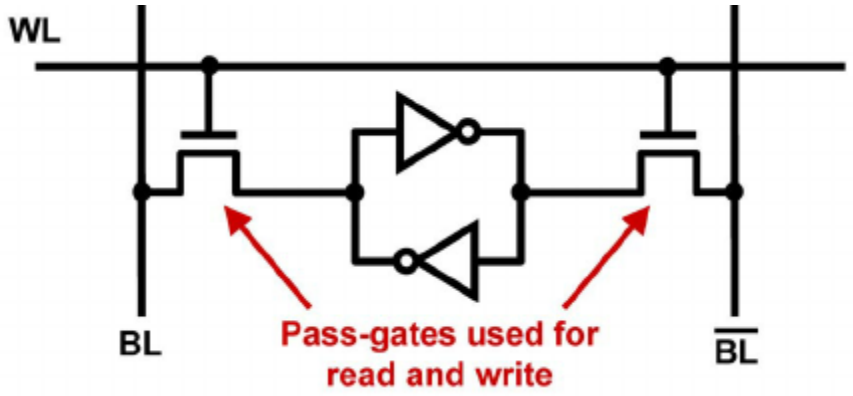


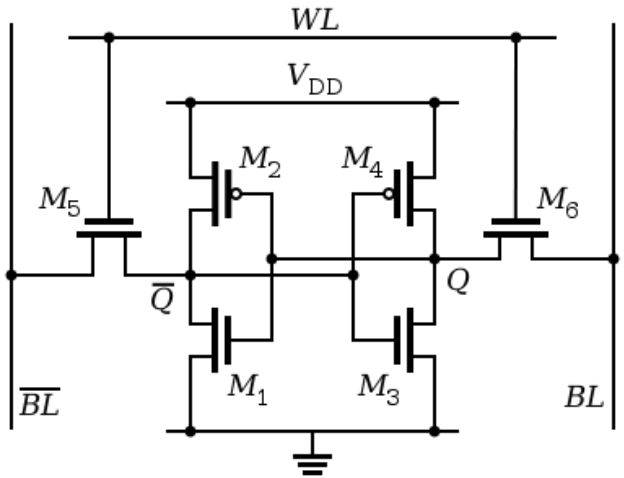
- **Data structure performance is proportional to memory stalls**
  - Applies to other data structures such as trees, graphs, …
- In general, **more data** leads to **worse performance**
  - But why? Does more data make MEM stalls longer? (Hint: yes)
  - And why is an array not affected by data size?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

Memory Technologies

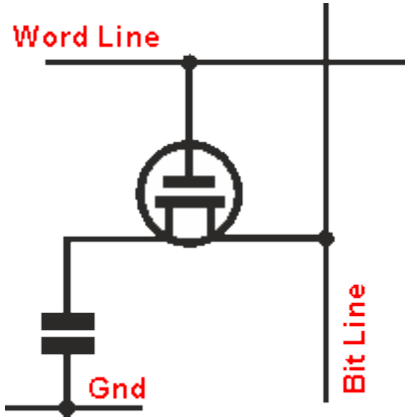
Static RAM (SRAM)

- 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**





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
    -> 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 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)

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

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."

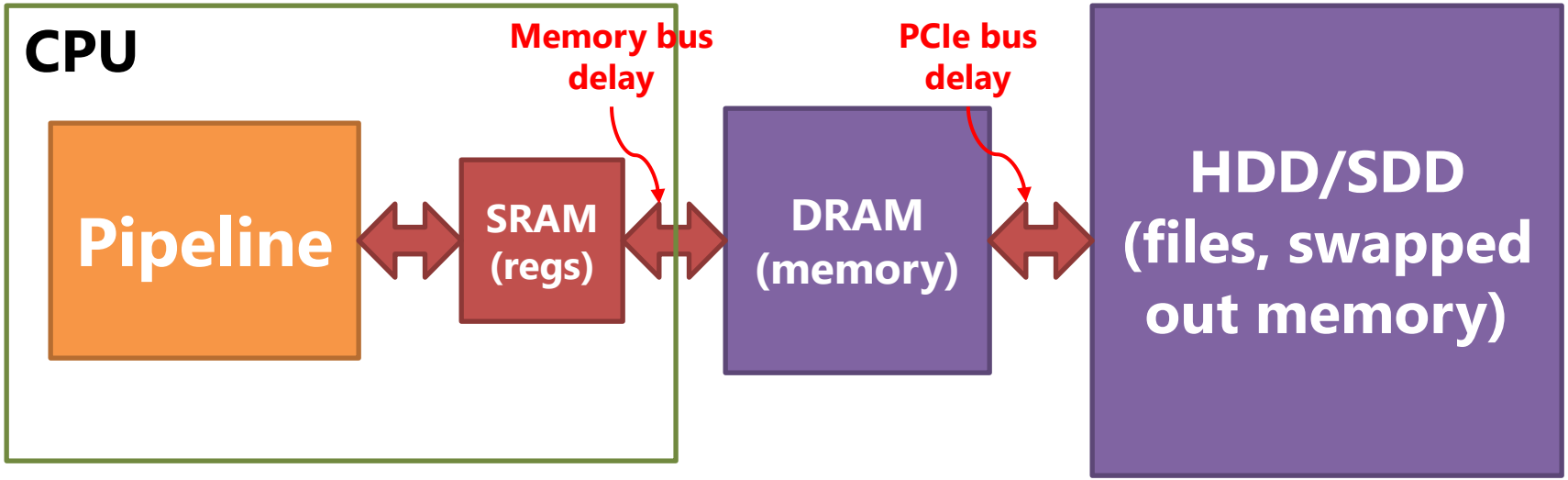
Do you notice a trend?

- 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

The memory hierarchy

System Memory Hierarchy

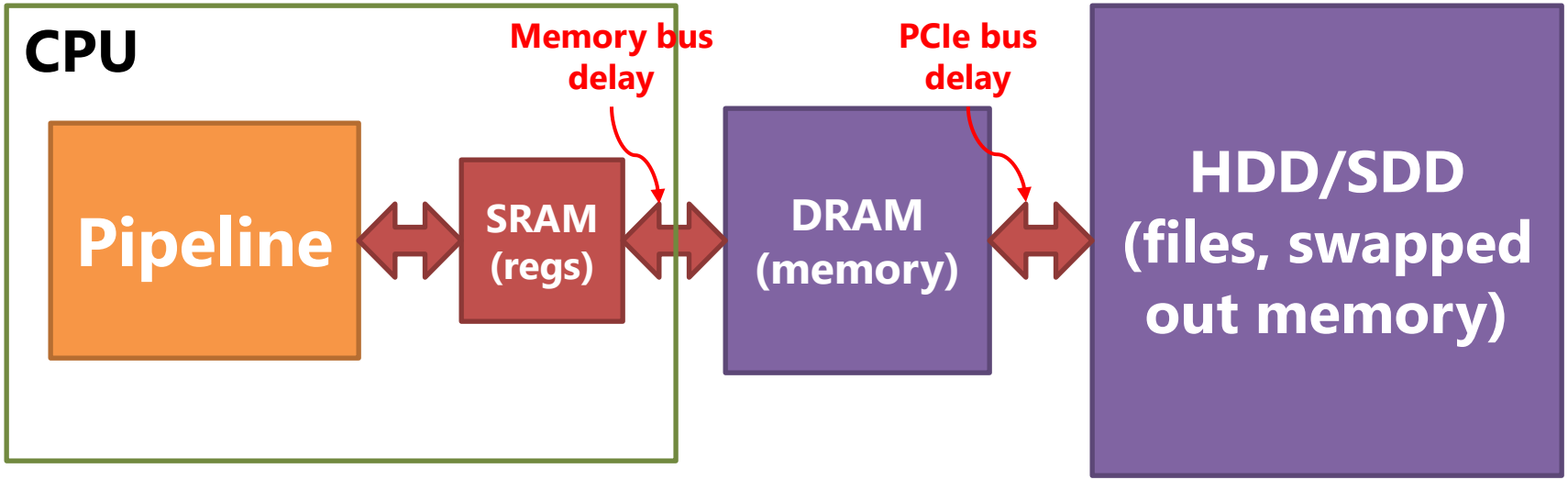
- Use fast memory (SRAM) to store frequently used data inside the CPU
- Use slow memory (e.g. DRAM) to store rest of the data outside the CPU



- 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

System Memory Hierarchy

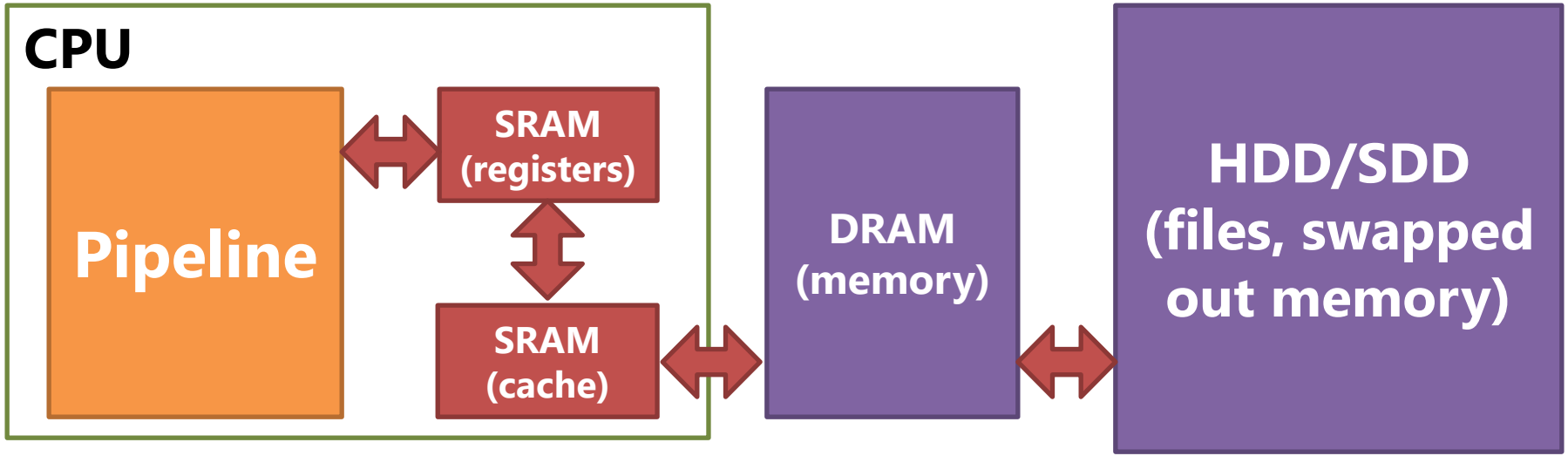
- Use fast memory (SRAM) to store frequently used data inside the CPU
- Use slow memory (e.g. DRAM) to store rest of the data outside the CPU



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

System Memory Hierarchy

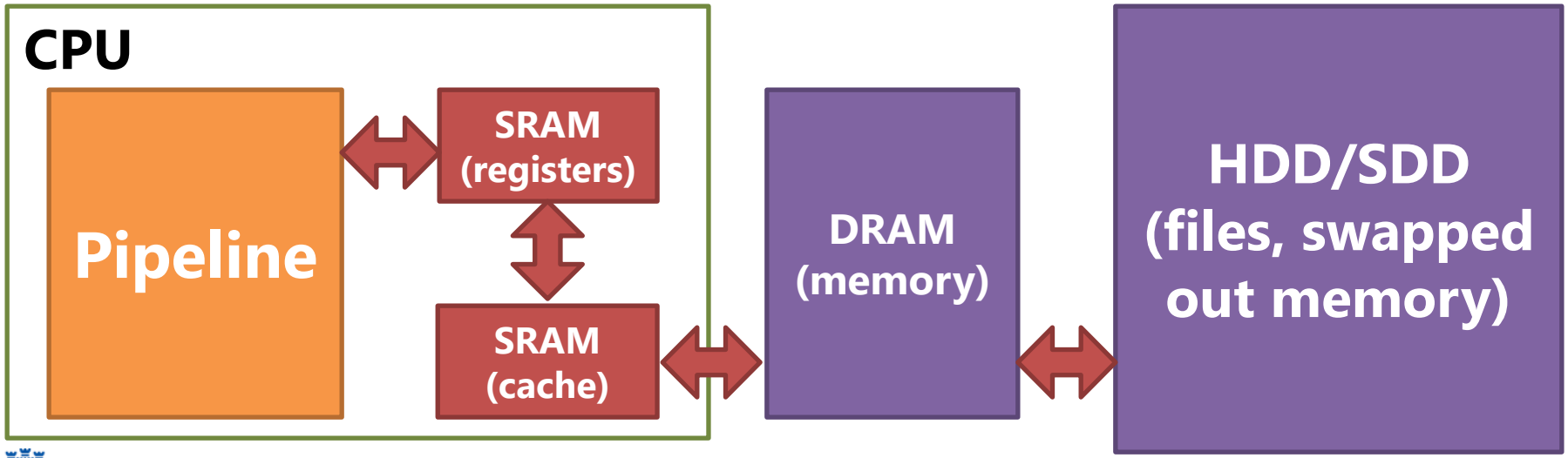
- Use fast memory (SRAM) to store frequently used data inside the CPU
- Use slow memory (e.g. DRAM) to store rest of the data outside the CPU



- 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**.

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

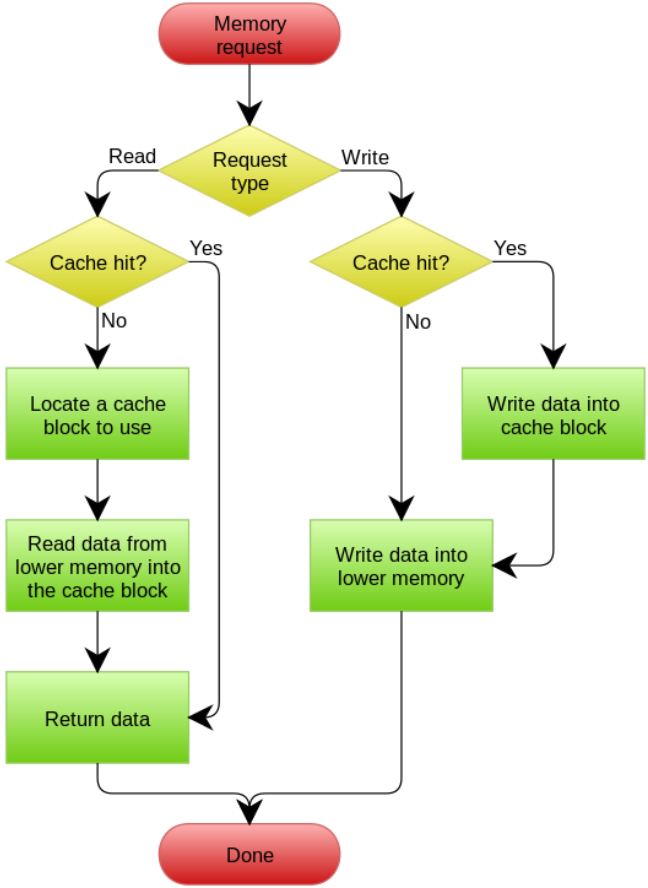


Cache Flow Chart

- **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**



Cache Locality: Temporal and Spatial

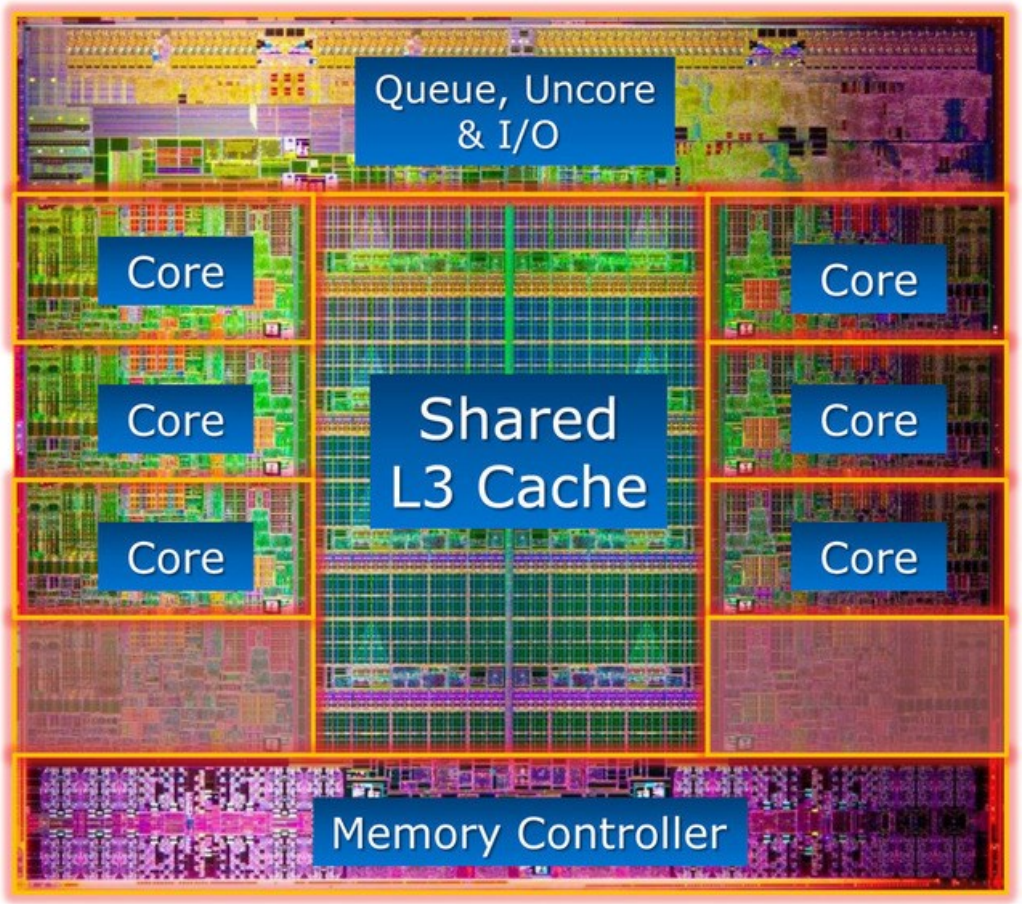
- Caching works because there is **locality** in program data accesses
  - **Temporal locality**
    - Same data is accessed many times in succession
    - **1st access** will **miss** but following accesses will **hit** in the cache
  - **Spatial locality**
    - Many data items that are spatially close 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**
- 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
  - Each program has a sweet spot. Architects choose a compromise.

So how big do we want the cache to be?

- Uber big!

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

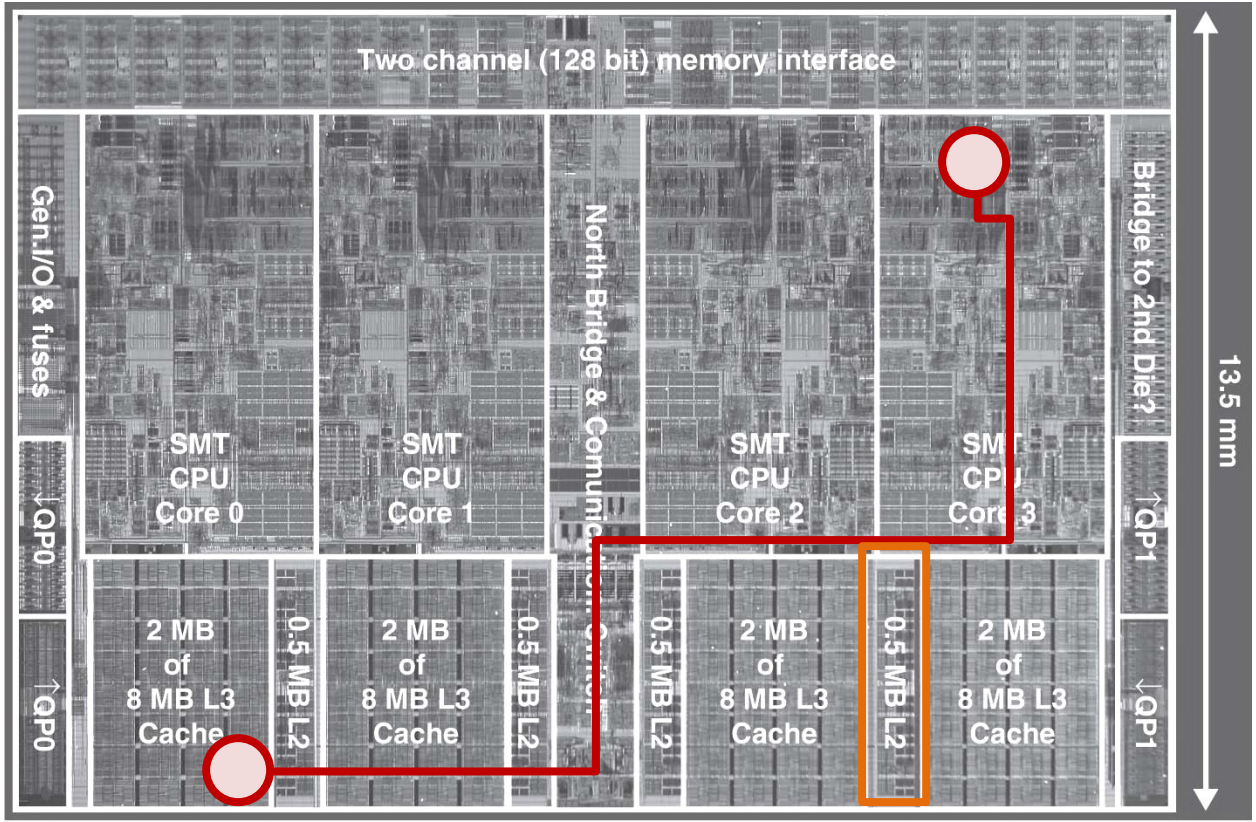
- But having a big cache comes with its own set of problems
  - Cache itself gets slower



Bigger caches are slower

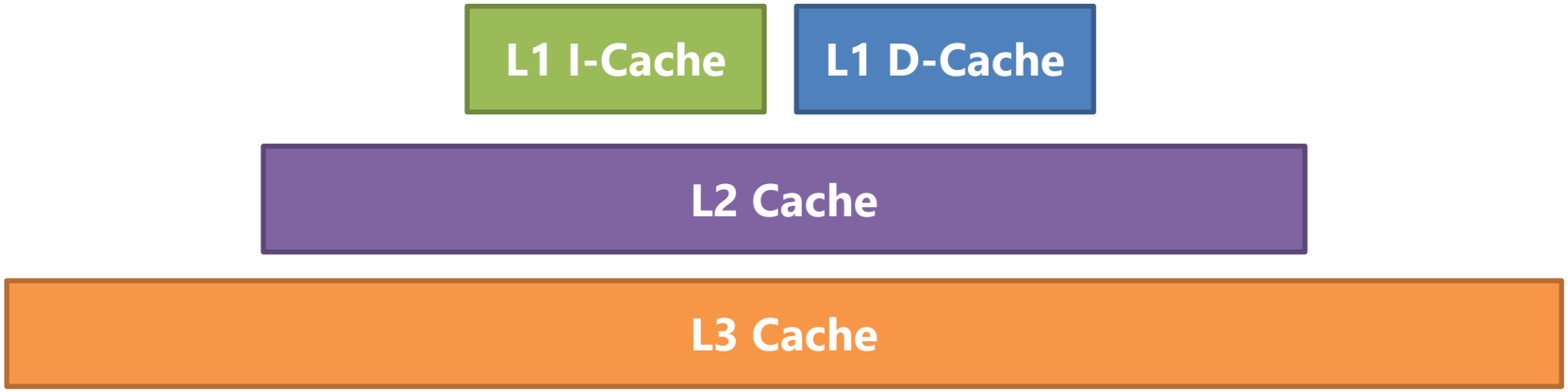
- Below is a diagram of a Nehalem CPU (predecessor of Sandy Bridge)

- 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:

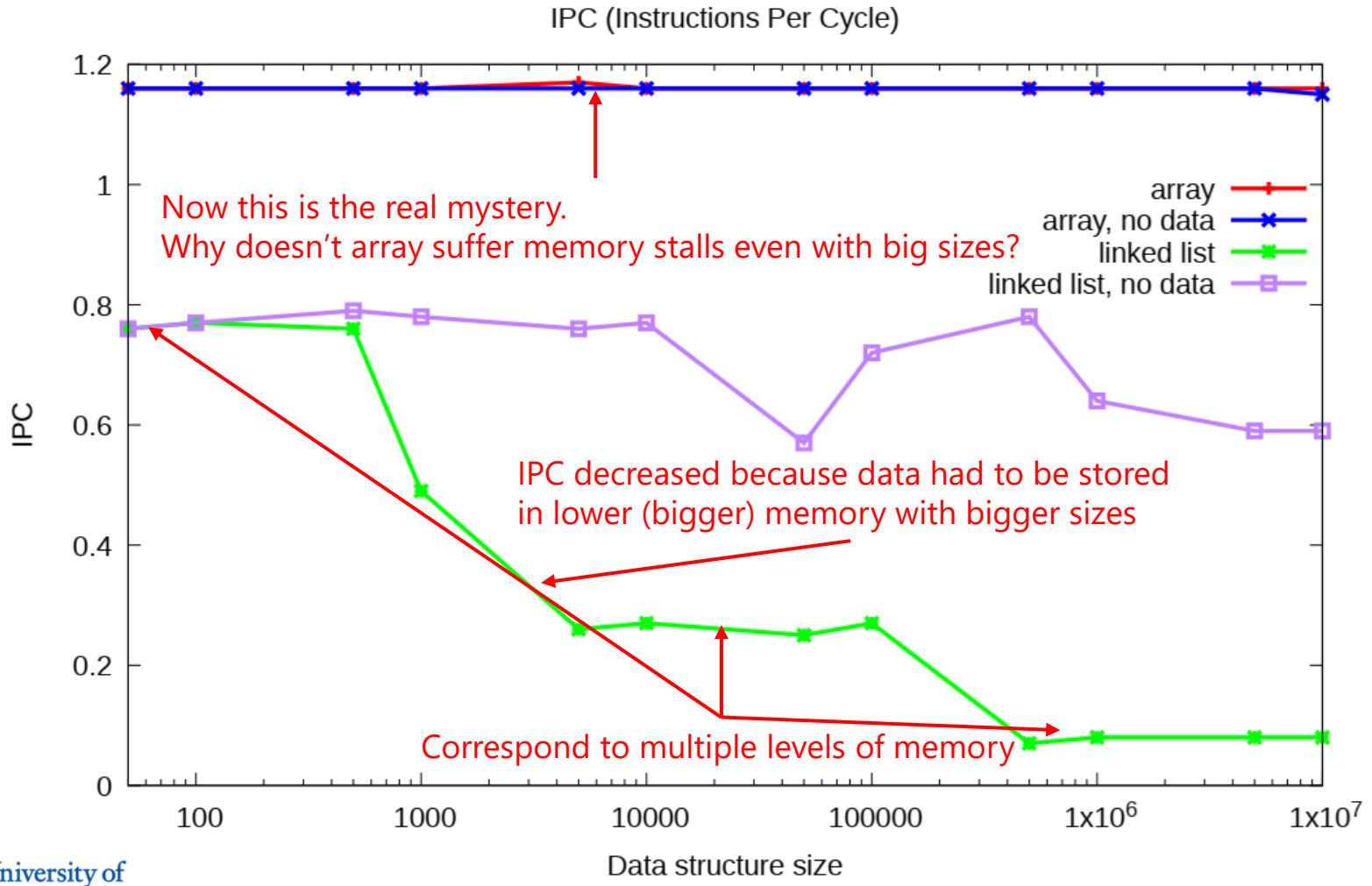


- L1 cache: Small but fast. Interfaces with CPU pipeline MEM stage.
  - Split to i-cache and d-cache to avoid structural hazard
- L2 cache: Middle-sized and middle-fast. Intermediate level.
- L3 cache: Big but slow. Last line of defense against memory access.
- Allows performance to degrade gracefully

kernighan.cs.pitt.edu cache specs

- On a Core i7-4400 Haswell:
  - 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)
- Access latencies (each level includes latency of previous levels):
  - L1: ~**3 cycles**
  - L2: ~**8 cycles**
  - L3: ~**16 cycles**
  - 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

Revisiting our IPC Results with new perspective



University of Pittsburgh