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

University of Pittsburgh

Code for linked-list.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 list over and over again 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
}
```

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

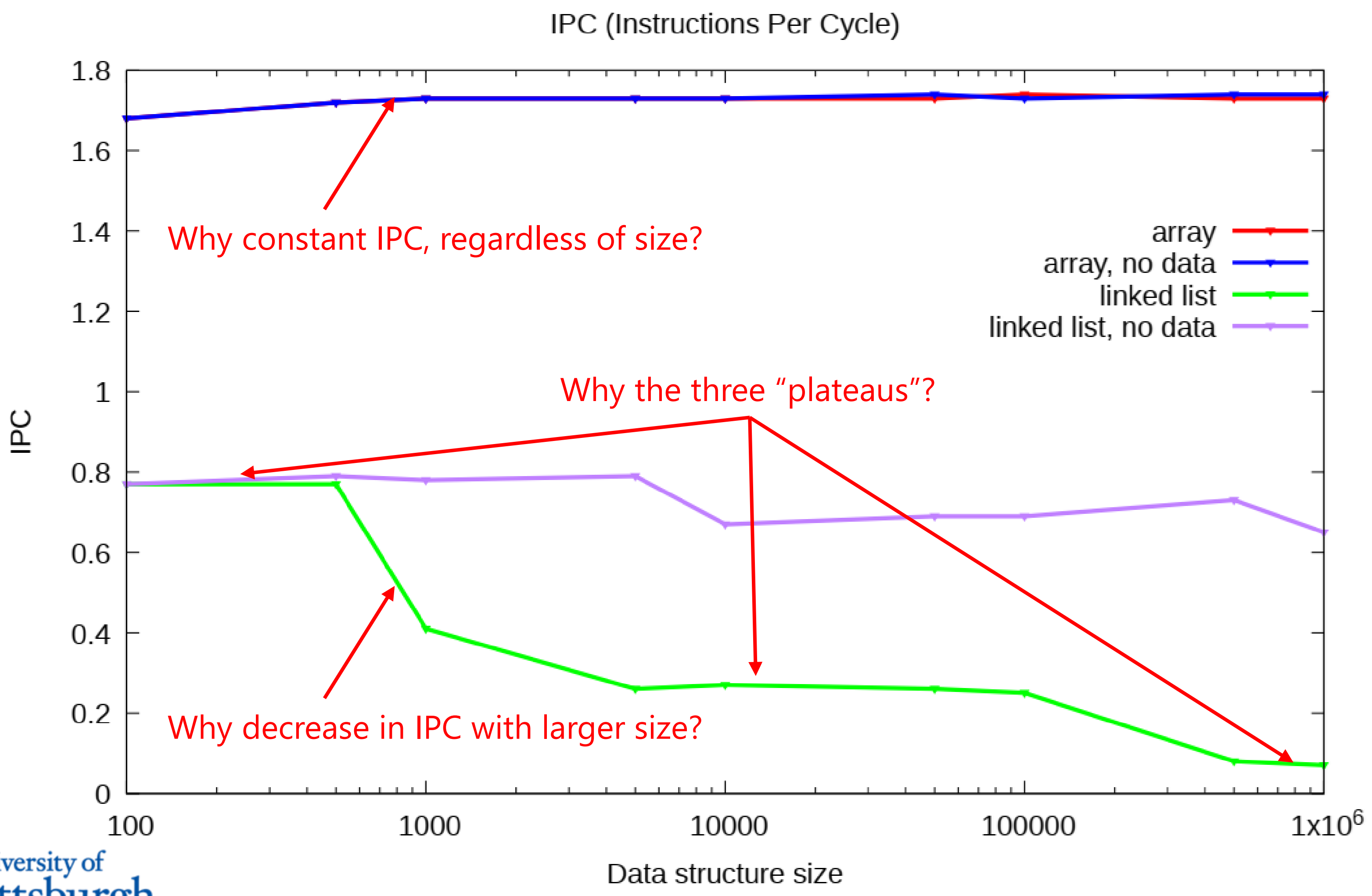
- Two CPU sockets. Each CPU:
  - Intel(R) Xeon(R) CPU E5-2640 v4
  - 10 cores, with 2 threads per each core (SMT)
  - L1 i-cache: 32 KB 8-way set associative (per core)
  - L1 d-cache: 32 KB 8-way set associative (per core)
  - L2 cache: 256 KB 8-way set associative (per core)
  - L3 cache: 25 MB 20-way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

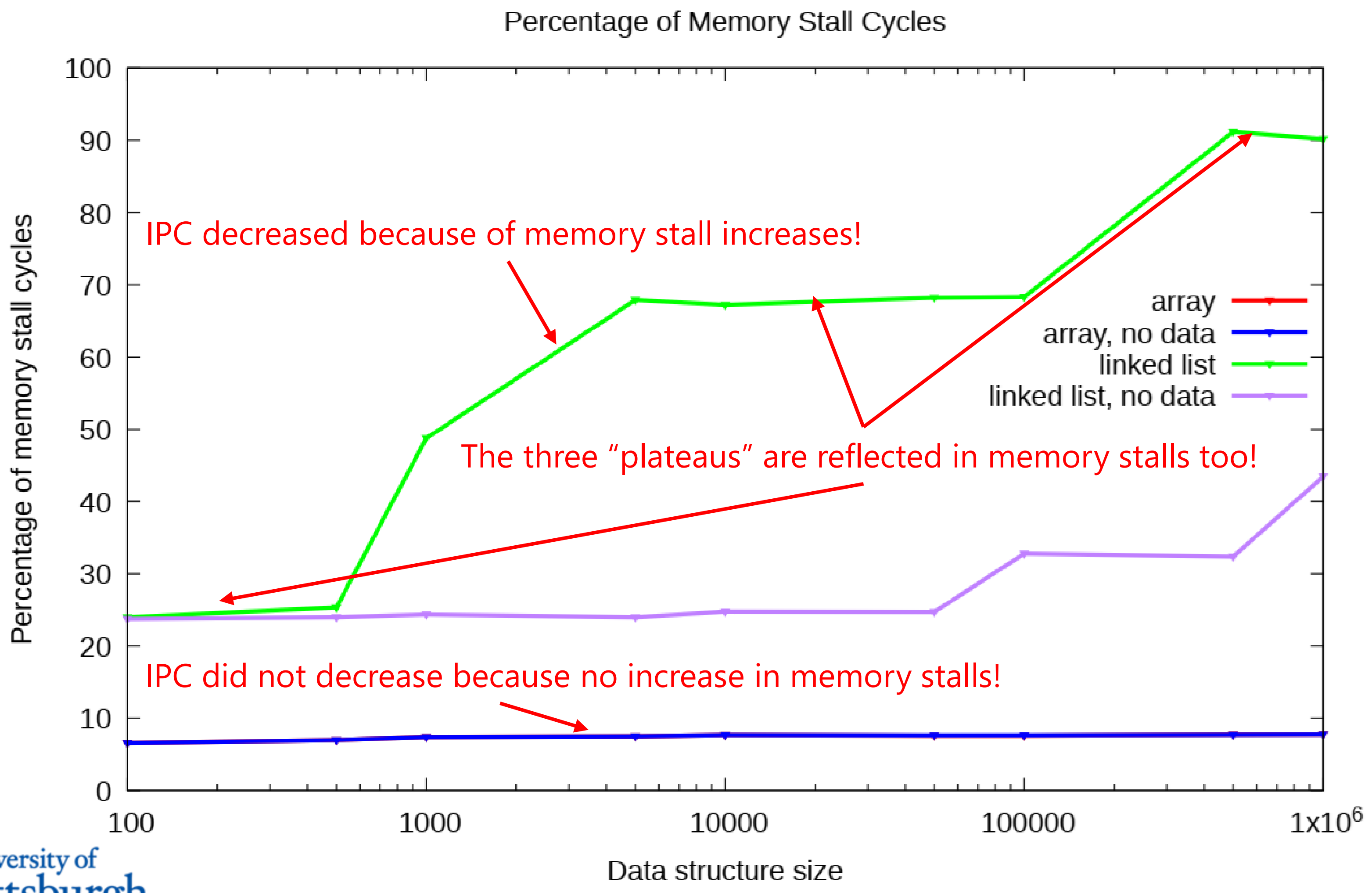
- Collected using CPU **Performance Monitoring Unit (PMU)**
  - PMU provides performance counters for a lot of things
  - Cycles, instructions, various types of stalls, branch mispredictions, cache misses, bandwidth usage, …

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

IPC (Instructions Per Cycle) Results



Memory Stall Cycle Percentage



IPC decreased because of memory stall increases!

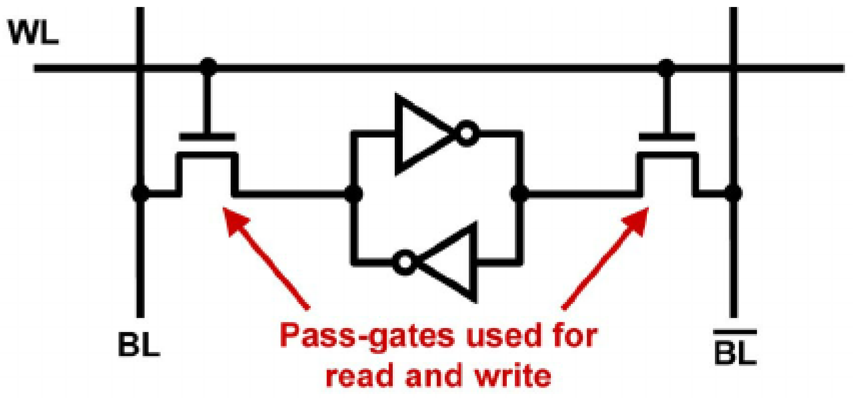
The three "plateaus" are reflected in memory stalls too!

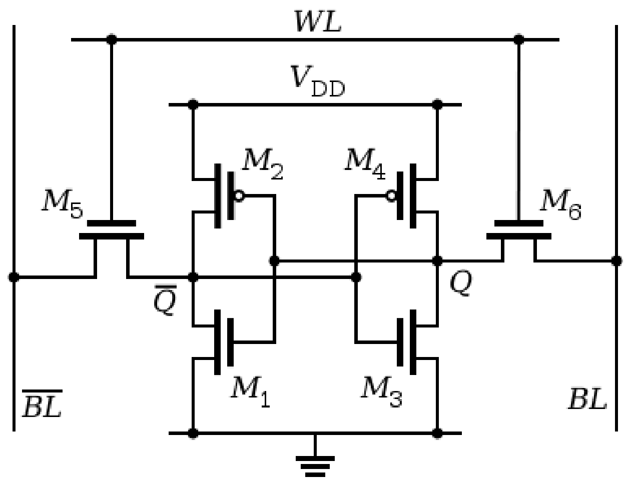
IPC did not decrease because no increase in memory stalls!

Memory Technologies

Static RAM (SRAM)

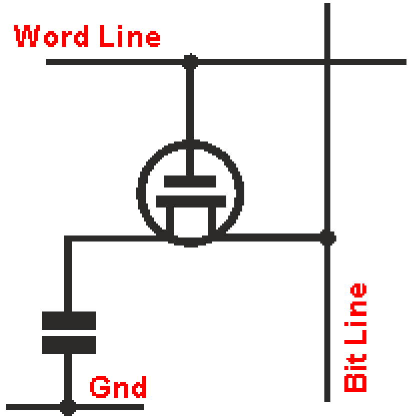
- SRAM uses a loop of NOT gates to store a single bit
- This is usually called a 6T SRAM cell since it uses... 6 Transistors!
- **Pros:**
  - **Very fast** to read/write
- **Cons:**
  - Volatile (loses data without power)
  - Relatively many transistors needed -> **expensive**





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

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.

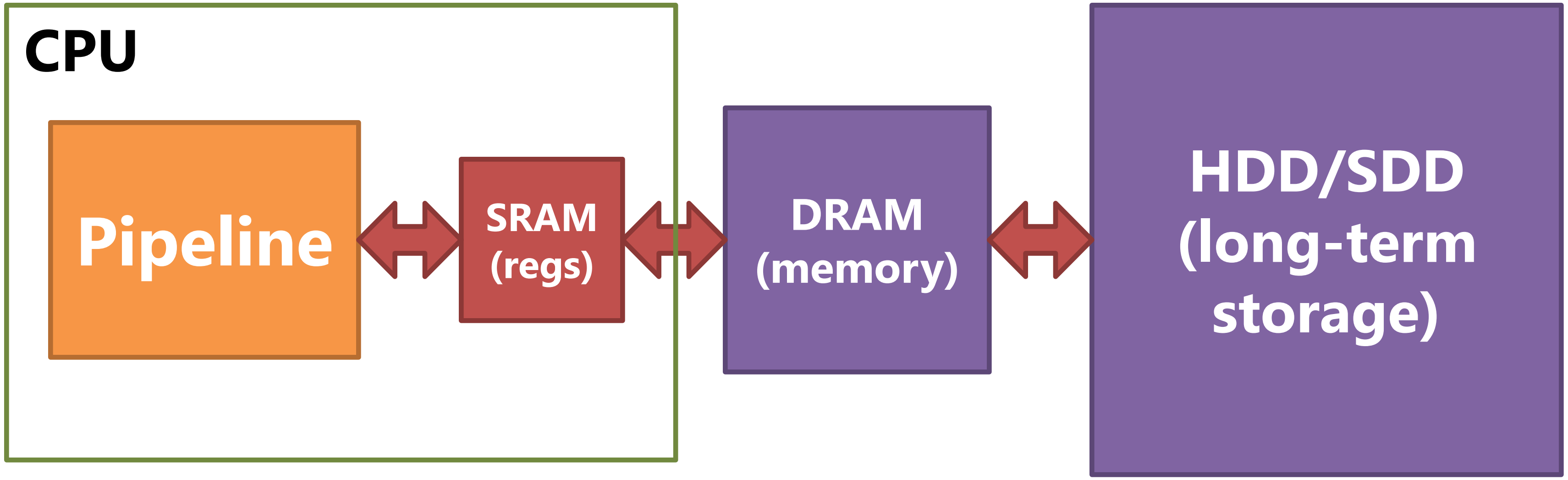
- Thus, memory is constructed as a hierarchy:
  - Fast and small memory at the upper levels
  - Slow and big memory at the lower levels

- 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

From each according to their ability…

- Use the **fast memory closest to the CPU,** and use the **big memory furthest from it.**



- Speed tends to be inversely proportional to both price and size.

Caching and Locality

- **Caching** is keeping a temporary copy of data for quick access.
- Each level in the hierarchy acts as a **cache** for the next lower level.
  - Registers are a temporary copy of data from memory.
  - Memory is a temporary copy of data from long-term storage.
  - Long-term storage is a temporary copy of... remote data?
- Caching exploits **locality.**
  - **Temporal locality** is the idea that you will access the same piece of data many times in succession.
  - **Spatial locality** is the idea that if you access a piece of data, you are likely to soon access another piece very close by
- By keeping data in a cache, we can greatly speed up data access by reusing the temporary copy.