Foundations of Modern Data Systems (FOMO)

Winter Semester 2025/2026

Exercise Session #2

23/10/25

Outline

1. Hands-on session: **Experiments & Profiling (con’t)**
2. The memory hierarchy – practical view
3. Project 1
4. Hands-on session: **Project 1 code walk-through**

Quick PSA

- The moodle form to provide us with your ssh-key closes the day of the course registration deadline, don’t be late!

Hands-on session

Experiments & Profiling (con’t)

A bit more about PMUs & profiling

- Using some slides from our friends at TU Dortmund, presented at BTW2025

https://tinyurl.com/nodmc-prof

Tutorial: Understanding Application Performance on Modern Hardware

Profiling Foundations and Advanced Techniques

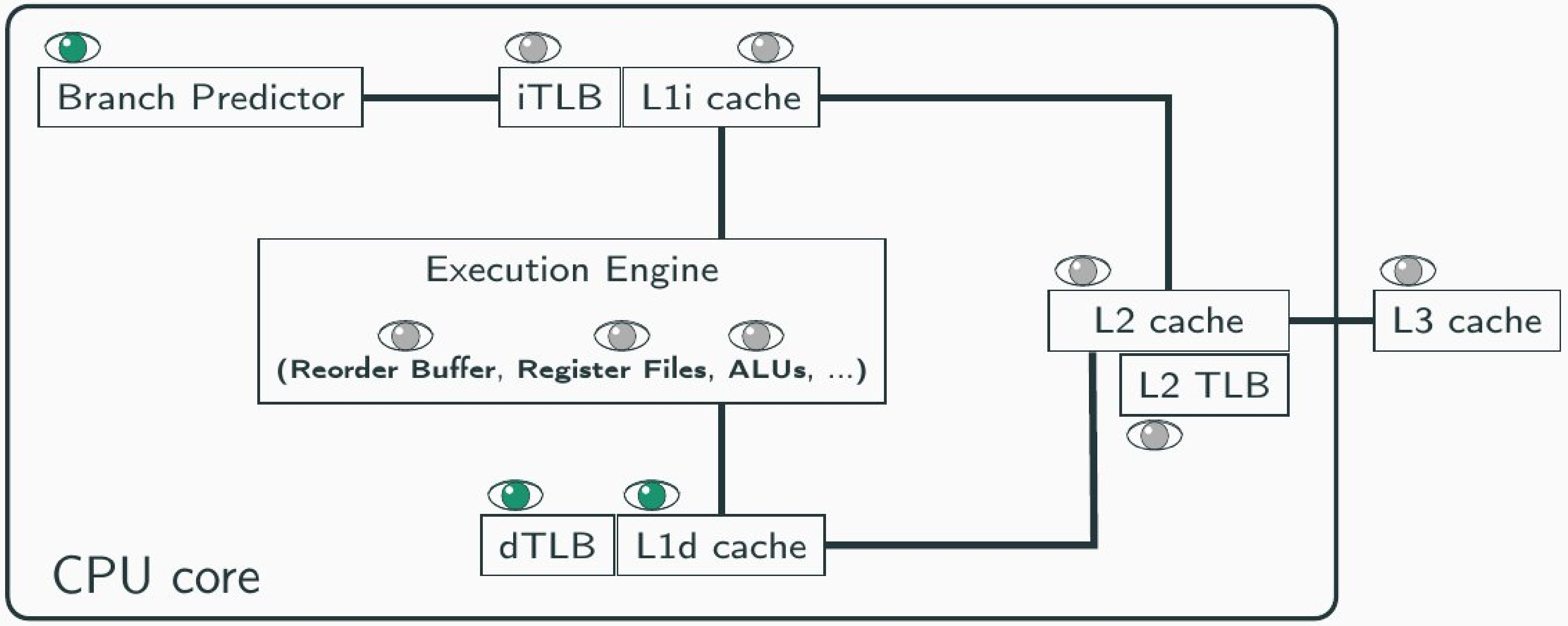
Jan Mühlig, Roland Kühn, and Jens Teubner

March 04, 2025

DBIS Group, TU Dortmund University

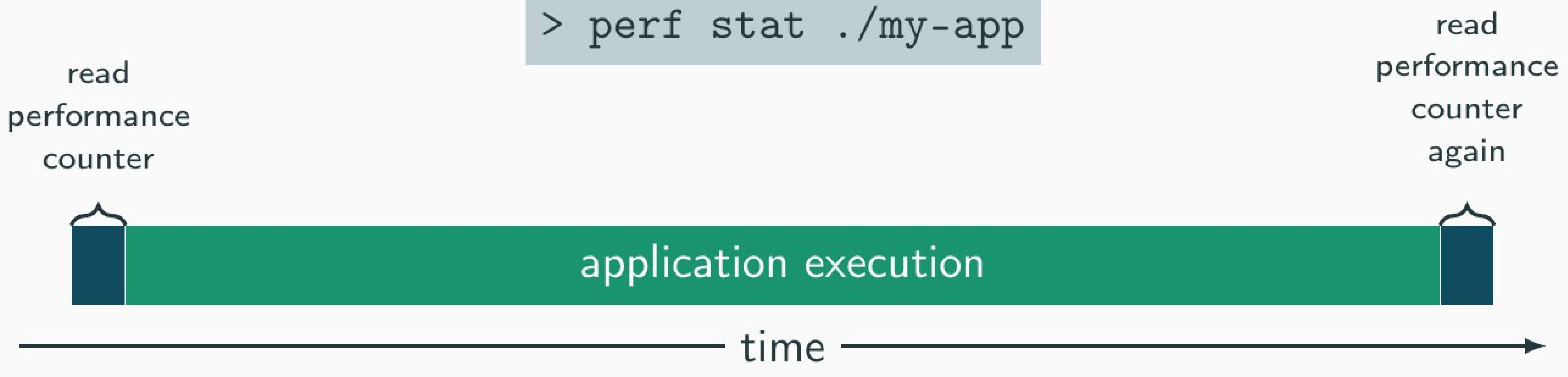
Performance Monitoring Units

- Modern CPUs are equipped with ***P****erformance* ***M****onitoring* ***U****nits*
- (Most) PMUs can **track** multiple *events* simultaneously
  - → E.g., `L1d` hits/misses, `dTLB` hits/misses, branch predictions, ...
  - → Hundreds to thousands events available (depending on the CPU vendor and model)



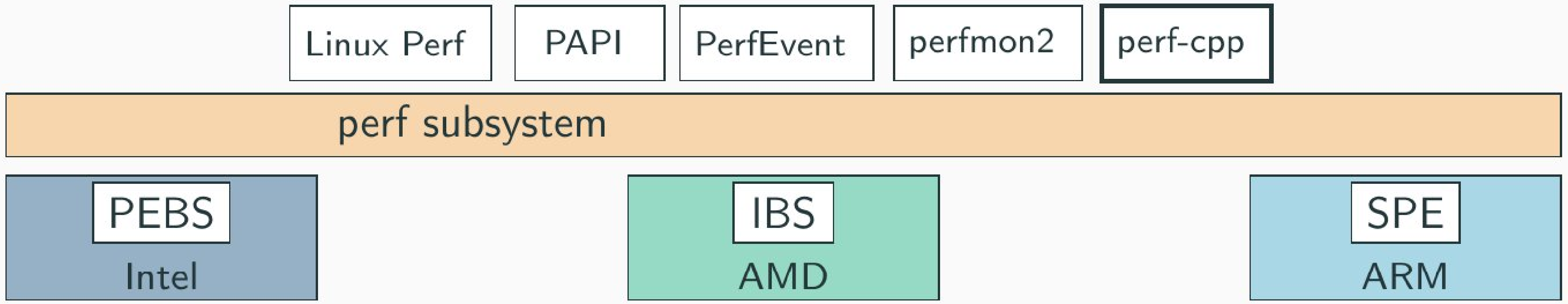
Profiling: Statistics

```
> perf stat ./my-app
```



```
170B   cycles
111B   instructions            # 0.66 insn per cycle
 23B   branches
  1B   branch-misses           # 8.22% of all branches
 62B   L1-dcache-loads
  4B   L1-dcache-load-misses   # 7.40% of all L1-dcache accesses
```

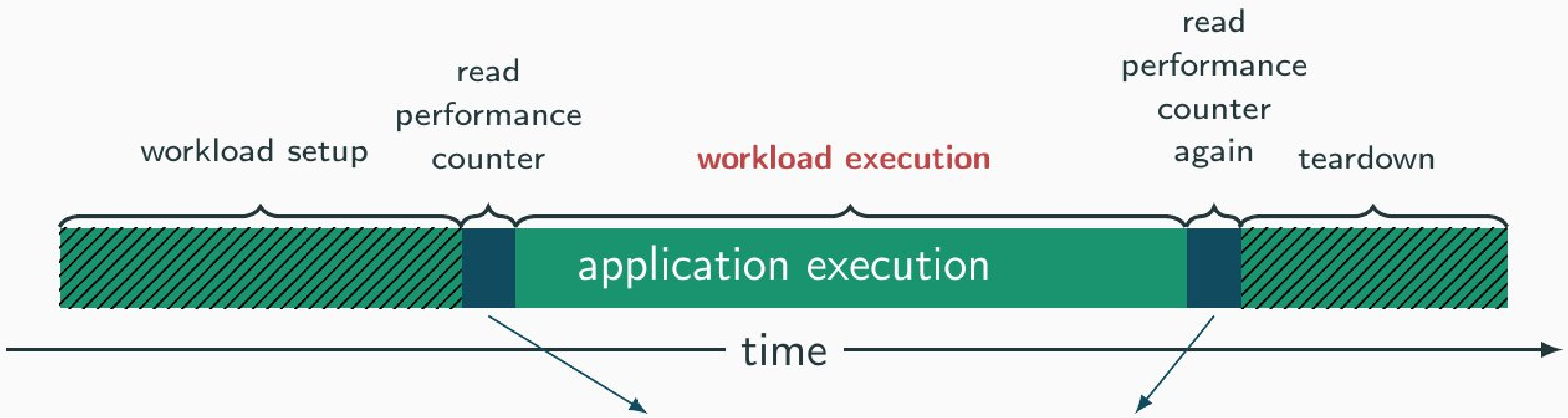
Abstractions



Takeaway

- ☹ Performance Monitoring Units are implementation-dependent
- ☺ The *perf subsystem* unifies configuration and access

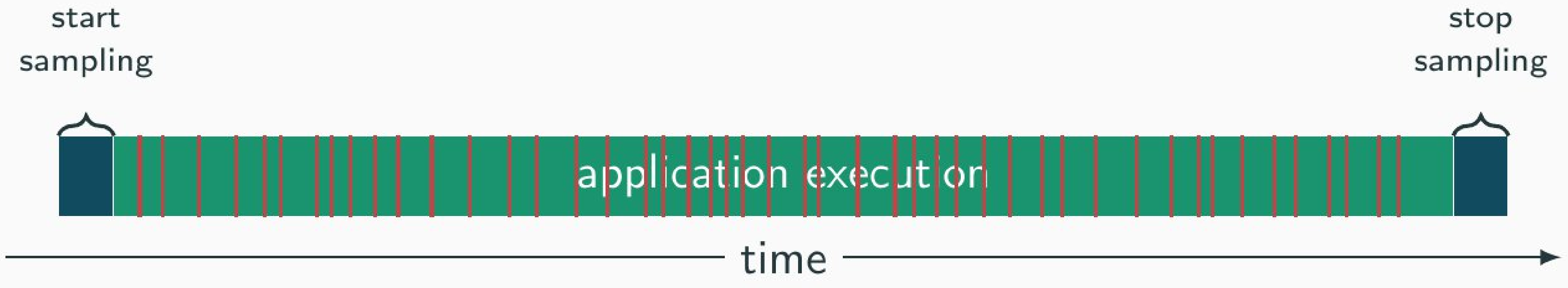
In-Application Profiling: Statistics



configure and read PMUs within the application e.g., via `PAPI`, `PerfEvent`, `perfmon2`, or `perf-cpp`

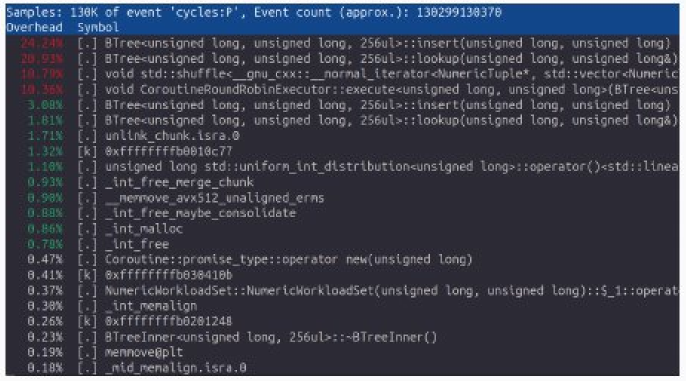
Profiling: Sampling

- **Idea**: Record periodic samples of the current execution state
- Needs to define
  - Period → *when* to record (e.g., every 4,000th *cycle* or *memory load*)
  - Sample → *what* to include (e.g., ***instruction pointer***, *memory address*, ...)



instruction pointer
0x58b140374b60
0x58b140374b64
0x58b140374b60
0x58b140374b6c
...

+ Binary & Debug Symbols =



```
Samples: 130K of event 'cycles:P', Event count (approx.): 130299130370
Overhead  Symbol
  24.24%  [.] BTree<unsigned long, unsigned long, 256ul>::insert(unsigned long, unsigned long)
  20.93%  [.] BTree<unsigned long, unsigned long, 256ul>::lookup(unsigned long, unsigned long&)
  10.79%  [.] void std::shuffle<__gnu_cxx::__normal_iterator<NumericTuple*, std::vector<Numeric
  10.36%  [.] void CoroutineRoundRobinExecutor::execute<unsigned long, unsigned long>(BTree<uns
   3.08%  [.] BTree<unsigned long, unsigned long, 256ul>::insert(unsigned long, unsigned long)
   1.81%  [.] BTree<unsigned long, unsigned long, 256ul>::lookup(unsigned long, unsigned long&)
   1.71%  [.] unlink_chunk.isra.0
   1.32%  [k] 0xffffffffb0010c77
   1.10%  [.] unsigned long std::uniform_int_distribution<unsigned long>::operator()<std::linea
   0.93%  [.] _int_free_merge_chunk
   0.90%  [.] __memmove_avx512_unaligned_erms
   0.88%  [.] _int_free_maybe_consolidate
   0.86%  [.] _int_malloc
   0.79%  [.] _int_free
   0.47%  [.] Coroutine::promise_type::operator new(unsigned long)
   0.41%  [k] 0xffffffffb030410b
   0.37%  [.] NumericWorkloadSet::NumericWorkloadSet(unsigned long, unsigned long)::$_1::operat
   0.30%  [.] _int_memalign
   0.26%  [k] 0xffffffffb0201248
   0.23%  [.] BTreeInner<unsigned long, 256ul>::~BTreeInner()
   0.19%  [.] memmove@plt
   0.18%  [.] _mid_memalign.isra.0
```

PerfEvent

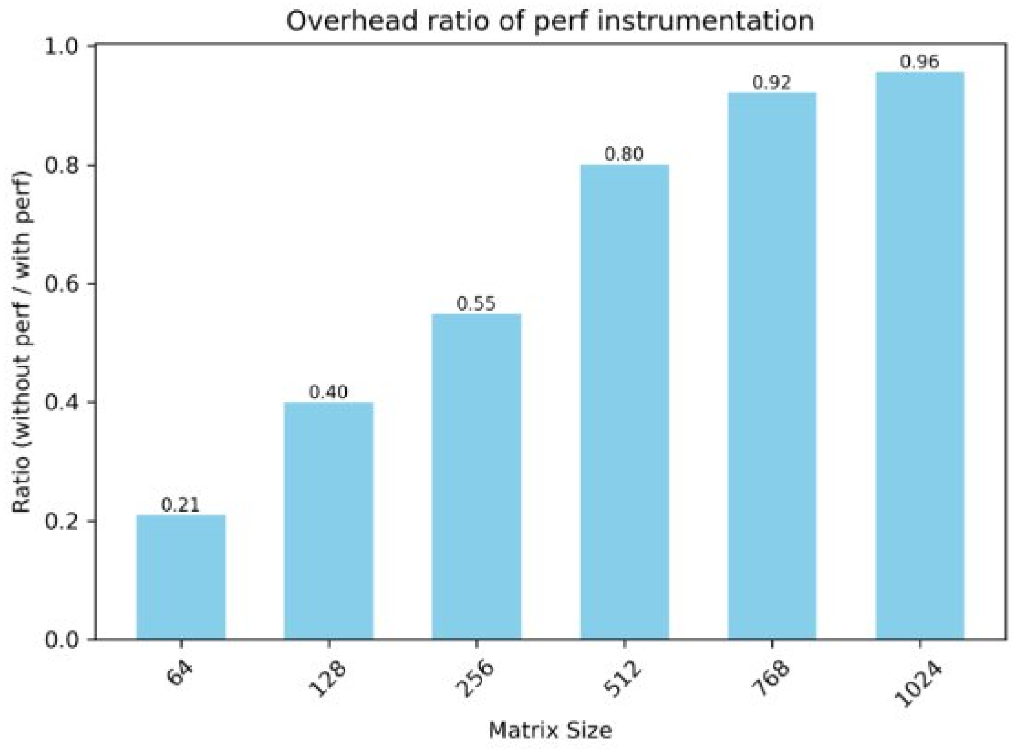
- Great drop-in for simpler projects to weave in perf profiling
- https://github.com/viktorleis/perfevent
- A single .hpp header to dump into your library with no complicated dependencies
- We provide it in FOMO/ADMS with the projects already, encourage using it

Likwid

- Another neat Perf-based tool (funded by BMBF btw!)
- https://github.com/RRZE-HPC/likwid
- Heavy-duty ‘all-in-one’ – energy consumption, NUMA topology analysis etc
- Can even weave into Fortran code!

Perf overheads

- Likwid overhead shrinks to 5%~10% (Runtime (without perf) / Runtime (with perf)) when matrix size is larger than 768x768.



Perf-cpp

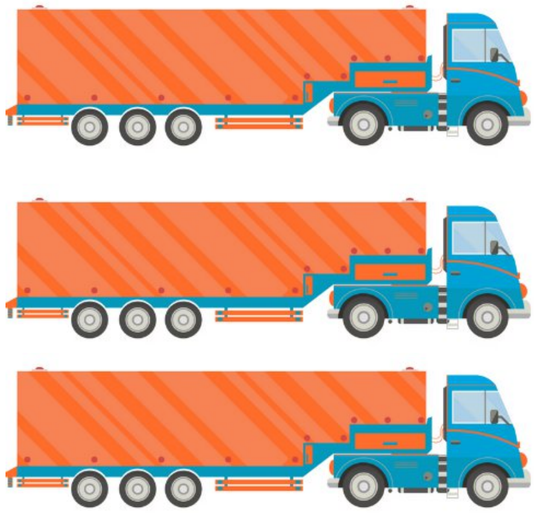
- https://github.com/jmuehlig/perf-cpp
- The project by TU Dortmund
- Includes ‘triggers’: conditions which trigger a PMU capture
- Powerful for more targeted profiling

The memory hierarchy

Practical view

Home cooking is one thing

A factory is a whole other beast



It starts with data

- Much like factories care about supply lines to feed the factory itself, data-intense algorithms need heavy-duty data acquisition mechanisms
- Most modern workloads are **memory-bound** and not compute-bound
- Maximizing memory performance will often speed up the workload

Data structure (DS) properties

5 core metrics

1. Insert Time
2. Update Time
3. Delete Time
4. Access Time
5. Space

Data structure (DS) properties

5 core metrics

1. Insert Time
2. Update Time
3. Delete Time
4. Access Time
5. Space

Severe implications on 1-4 due to the memory hierarchy

Data structure (DS) properties

5 core metrics

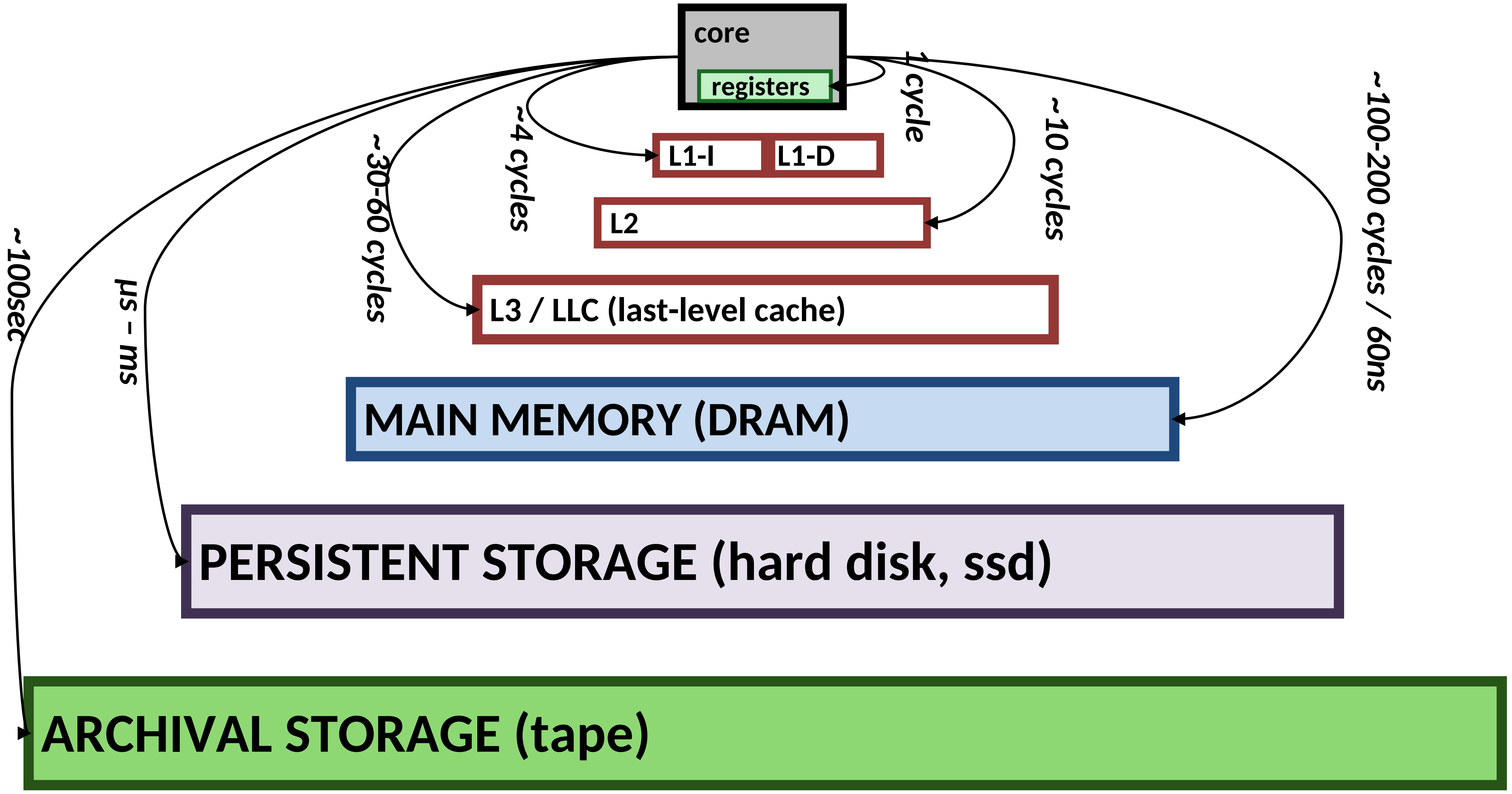
1. Insert Time
2. Update Time
3. Delete Time
4. Access Time
5. Space

What about these 4 for sequential access vs random?

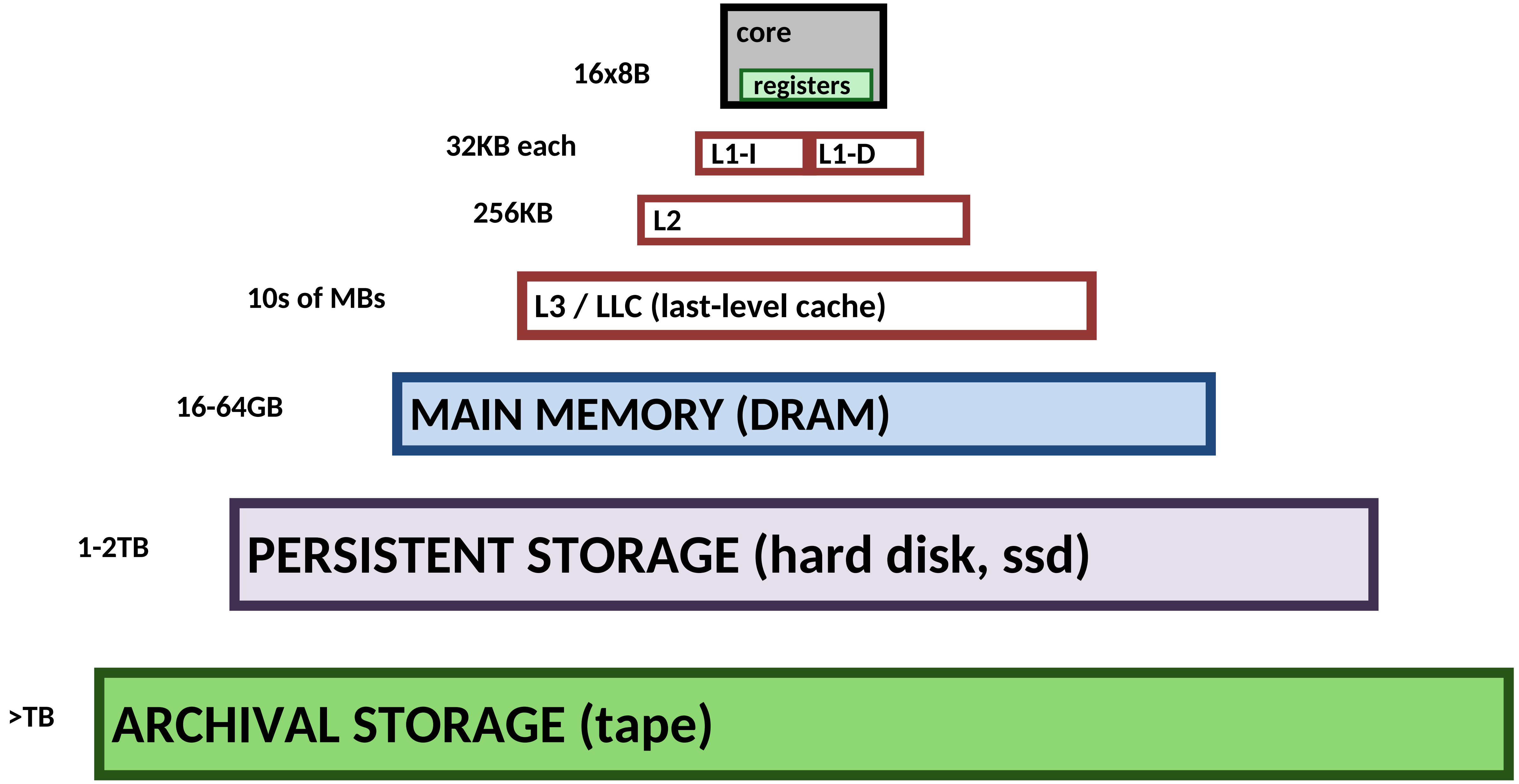
What about worst-case, average-case, best-case?

Severe implications on 1-4 due to the memory hierarchy

(typical) storage hierarchy – access latency



(typical) storage hierarchy – capacity



Why do we care?

- Two data structures with O(1) lookup are unlikely to have the same performance in the real world
- Random access behaves differently to sequential access
- access latency to a memory hurts some DS more than others
- We can trade-off space for decreased latency & higher bw
- We can try to maximize caching effects with reuse
- We can try to exploit locality by prefetching values we might need soon

Why do we care?

- Two data structures with O(1) lookup are unlikely to have the same performance in the real world
- Random access behaves differently to sequential access
- access latency to a memory hurts some DS more than others
- We can trade-off space for decreased latency & higher bw
- We can try to maximize caching effects with reuse
- We can try to exploit locality by prefetching values we might need soon

Key topics we cover next week

Array vs Vector

- Array
    - Static pre-allocated aligned piece of memory
    - Guarantees efficient sequential access
- Lists (vectors)
    - dynamically growing piece of memory
    - No guarantees that memory is aligned
    - Potentially garbage performance (caused by fragmentation)

- Great example: malloc’d arrays in C++ versus std::vector

Array vs Vector

```
vec.reserve(N);
for (size_t i = 0; i < N; ++i) {
    vec.push_back(i);
}
```

- **Without** reserve():
  - push_back() may trigger reallocations
  - Reallocations are costly in terms of performance

- **With** reserve():
  - Allocates required memory upfront
  - Eliminates need for reallocations during insertion

- **Benefits**:
  - Improves insertion performance
  - Reduces memory fragmentation

Cache Behavior - Access Patterns

- Access Patterns are key:
  - Sequential Access:
    - Data likely resides in cache & is getting prefetched -> faster access
  - Random Access:
    - Each access is a cache-miss -> slower access

- Use perf to measure cache misses:
  - perf stat -e cache-misses ./main

Summary

- Data size, memory alignment and access patterns are key points of consideration for high performance DS
- These are controlled **by you** (or your requirements)
- If you understand them well, you can start considering trade-offs with a hardware awareness

Project 1

Rundown of the projects

Starts today!

Points	**Project**
15	**1. Benchmarking & Memory Hierarchy** • Benchmark caches -> Benchmark Data Structures -> Report • *O(1) != O(1)*
15	**2. Vectorization & Parallelism** • Vectorized (SIMD) and Parallelized (multi-threading) implementation of an analytical algorithm • *Multi-threading isn't always a good idea*
20	**3. Accelerators** • Optimize a given application for both **CPU** and **GPU** and offload when appropriate • *Offloading isn't always a good idea*
Total: 50	

Learning Objectives

- Accurately measure and analyze hardware memory performance metrics.
- Evaluate and compare the performance of different data structures.
- Clearly and concisely communicate benchmarking results using visual and written formats.
- Explain how the modern hardware memory hierarchy influences the performance of data structures.

How will we do this?

- Benchmark some data structures!
- Make some cool plots!
- Dig into the weeds of how the CPU memory hierarchy works...

Our Data - the humble 64B Node

```
struct alignas(64) Node {
    uint64_t key;
    uint64_t data;
    Node *next;
    char padding[64 - 16 - sizeof(Node *)];
};
```

A 64 byte aligned Node
Each lookup = a cache line read

Makes lookups *very* memory intense

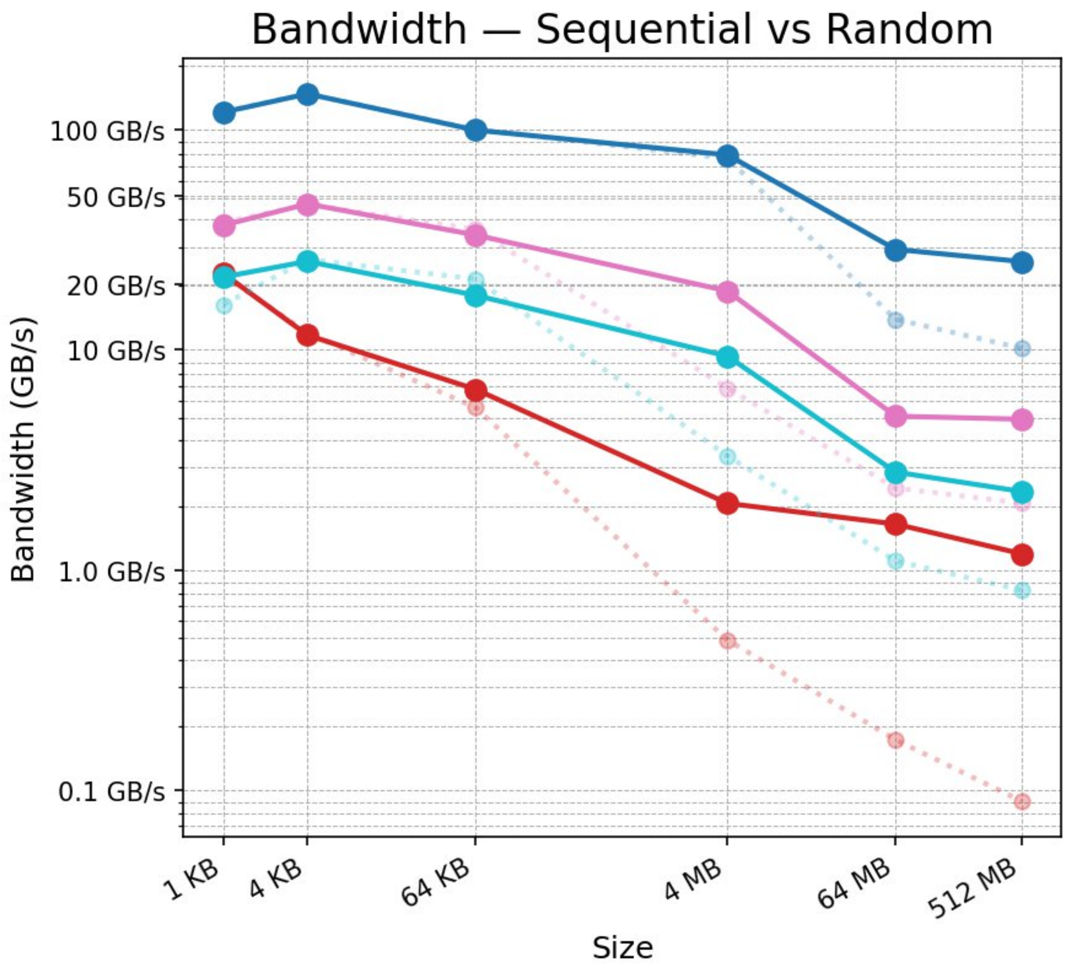
Our contestants

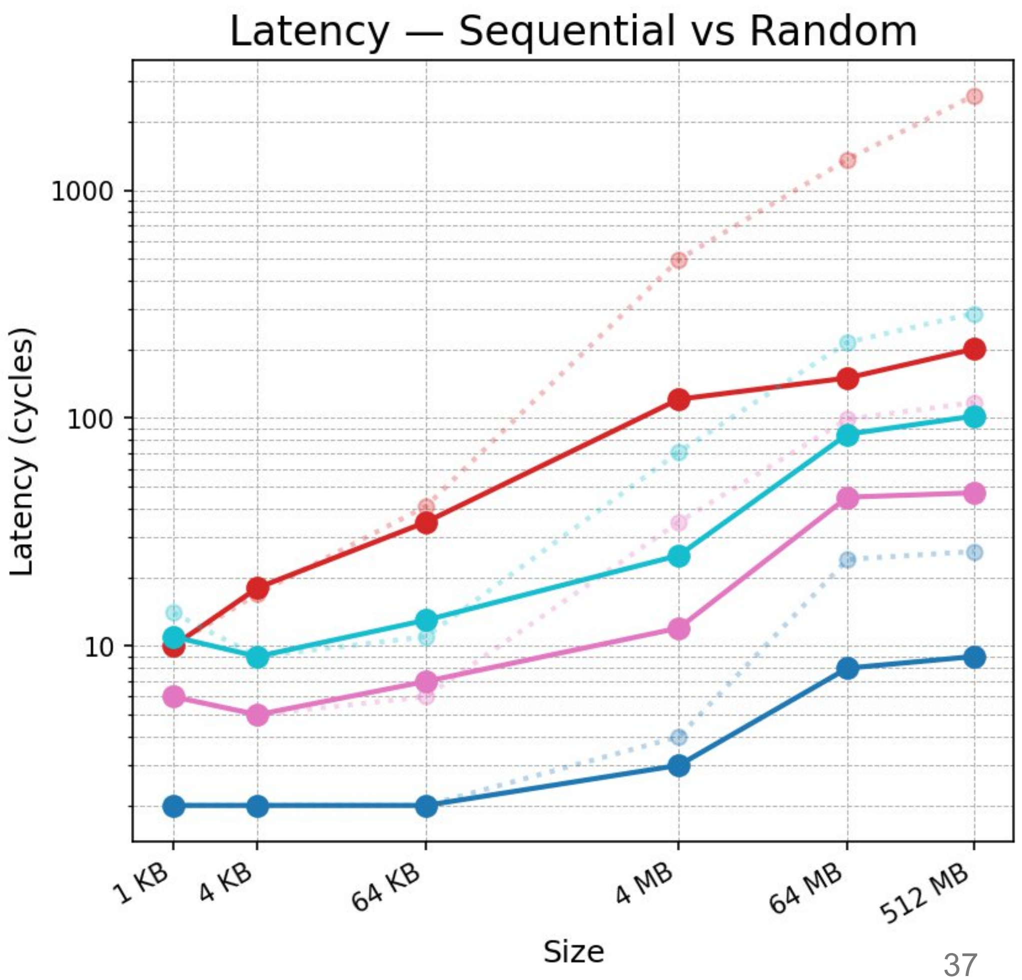
- **Directly accessed array**
  - Designed for dense datasets
  - Pre-initialized memory
  - Keys must be inserted in sequential order
- **Binary search over a list**
  - Designed for sparser datasets (gaps between keys)
  - Still pre-initialized and packed together as a vector for simplicity sake
  - Keys must be inserted in sequential order
- **Chained hash table** (implemented using a vector of vectors)
  - Dynamic size
  - Universal data structure
  - Various trade-offs in terms of chain depth (bin size) and total size of the table
  - Keys could be inserted in random order

Our metrics

- Lookup Bandwidth (average bytes retrieved per second)
- Lookup Latency (average cycles/lookup)

Your Goal


Bandwidth — Sequential vs Random
100 GB/s
50 GB/s
20 GB/s
10 GB/s
1.0 GB/s
0.1 GB/s
Bandwidth (GB/s)
1 KB
4 KB
64 KB
4 MB
64 MB
512 MB
Size


Latency — Sequential vs Random
1000
100
10
Latency (cycles)
1 KB
4 KB
64 KB
4 MB
64 MB
512 MB
Size

Your Goal - 3 Tasks

1. (3 pts) Evaluate the bandwidth and latency of your memory system
2. (6 pts) Evaluate the bandwidth and latency of 4 distinct data structures
   - Directly accessed array
   - Binary Search over a list
   - Chained Hash with a bin size of 1
   - Chained Hash with a bin size of 16
3. (6 pts) Report: explain how you obtained your results and **what do they mean**

Your benchmark

- Dataset Generation
- Data Structure Initialization
- Performance measurement (lookups)

- Consider some questions:
- How do I sufficiently ‘stress’ the data structures to really measure memory costs and not compute costs?
- How to ‘fairly’ simulate random access?

Some restrictions on key order for your sanity

- All data structures should have N Nodes inserted to them in ascending order by key (even the hash table)
- A key should not be looked up more often than once every N lookups
- All keys should be looked up the same amount of times at the end of the test.

A note on the chained hash table

We ask to evaluate 4 data structures but provide you with 3 classes. What gives?

The **ChainedHashTable** has a hyper parameter – bin size

Use this to initialize two variants of the data structure:

- Chained Hash with a bin size of 1
    - Is this still a hash table?
    - How will it perform relative to an array?
- Chained Hash with a bin size of 16
    - How does it compare to bin size 1?

Final outputs

1. DirectAccessArray
2. BinarySearchArray
3. ChainedHashTable(bin_size=1)
4. ChainedHashTable(bin_size=16)

return {bw_1, bw_2, bw_3, bw_4, lat_1, lat_2, lat_3, lat_4}

Tests

- 2 Basic tests (0.5 points each)
- 8 Advanced tests (1 point each)

The Report

1-2 pages, font size 11

1 point - **explanation** of the benchmarking setup (how the dataset is prepared, how the data structures are evaluated, etc)

1 point - **a plot** showing the bandwidth of all the data structures across varied access patterns and dataset sizes.

1 point - **a plot** showing the latency of all the data structures across varied access patterns and dataset sizes.

1 point - **Compare** the performance of the **DirectAccessArray** and the **ChainedHashTable** with bin size set to 1. Is the performance what you expected? If not, why not.

1 point - **Compare** the performance of the **BinarySearch** with random access versus sequential access. Is the performance what you expected? If not, why not.

1 point - **Compare** the performance of the **ChainedHashTable** with a bin size of 1 versus a bin size of 16. Is the performance what you expected? If not, why not.

Further points

- Full details are in the project description found under this week’s moodle section
- We will talk further about the data structures and specifically the effects of caches & memory alignment in closer detail next week

- FORK the project to your own namespace
- The report should be appended to your fork

Hands-on session

Project 1 code walk-through

QA