FOMO Project #1
Benchmarking

October 23, 2025

1 Motivation

Typically, we think of data structures as having a space and time asymptotic complexity which should guide us as to when we should use each one. However, asymptotic complexity is just that - asymptotic and does not capture the additional constant overheads which are introduced in real-world scenarios.

A key source of these overheads is the memory hierarchy – a critical component of modern computer architecture, enabling to trade off memory capacity for performance (both in terms of latency and bandwidth).

Understanding this interplay between the hardware and data structures is critical to design efficient applications and benchmarking plays a key part.

2 Learning Objectives

The project's learning objectives are:

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

3 Tasks

3.1 Task Outline

In this project, you are provided a set of data structures and a set of scenarios for performing lookups on those data structures. Your task is to construct a benchmarking harness for the data structures such that you can determine how they interact with the provided hardware and the lookup scenario. You will be provided hardware to execute your experiments on and have to perform various evaluations to measure the performance of the data structures under the various scenarios.

The project is split into three sub-tasks:

1. Sub-task 1: Basic hardware benchmarking
2. Sub-task 2: Data structure benchmarking & evaluation
3. Sub-task 3: Report analyzing the obtained results in sub-task 2.

3.2 Sub-task 1: Basic hardware benchmarking (3 points)

Before evaluating more involved data structures on a fixed hardware configuration, it is important to first understand what the expected performance of the machine is. For the purposes of data structure lookups, the two key metrics are average latency and average bandwidth of each level of the memory hierarchy, from L1 cache down to DDR memory. To that end, we need a pair of data structures which are as simple as possible to truly be bottlenecked by nothing else than the memory performance. In this project, we use a lookup table stored as an array as well as a shuffled linked list.

Another consideration is the payload itself - we often need to fetch large chunks of data and not just individual integers. To that end, we use a struct called **Node** aligned to 64 bytes - a single cache line. This guarantees that every attempt to read this element will require loading a fresh cache line, making performance measurement much more precise.

In this task, you are provided most of the ingredients: the data structure, the benchmarking scenario and the workload itself. What is missing is the measurement logic. Your task is to introduce said logic such that you can evaluate how the provided hardware behaves in terms of average latency and average bandwidth across various levels of the memory hierarchy. You are encouraged to use the **perf** tool, but can use anything else that is exposed to you with the C++20 standard. Keep in mind that simply running the provided workload may not be enough to accurately measure the hardware performance and so further modifications to the code might be necessary!

For evaluating latency, we use the shuffled linked list as shown in Figure 1, the size of which you can incrementally increase until you can observe the degradation in access speed down to main memory access latency. The shuffled linked list is fit for the task, as it is a pointer-chasing problem and as such, the prefetching logic of the CPU does not come into play. Every element is accessed randomly. You are expected to return the latency in terms of clock cycles for each benchmarking scenario.

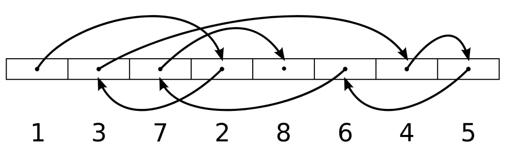

Figure 1: Randomly linked list without cycles

The results you are expecting should look similar to what is shown in Figure 2.

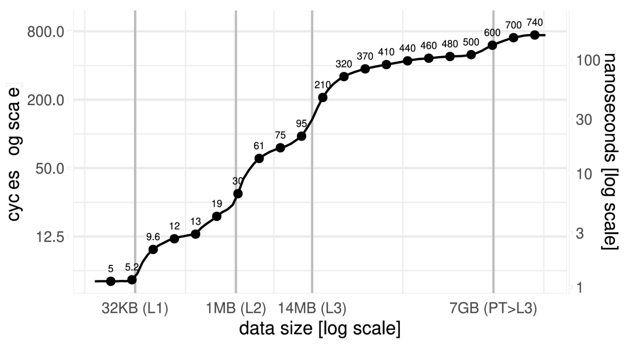

Figure 2: Cascade Lake X Data Access Latency

To measure the bandwidth of your various memories, you need an application which is dominated by sequential accesses to make use of hardware prefetching. This can be done by setting an array to a constant like zero, incrementing a sum or incrementing an array itself. The key is for the computation to be simple enough to not influence the bandwidth. Figure 3 features increments per second as an example of what the bandwidth results should roughly look like. Please return the bandwidth in MB/s (rounded to the nearest integer). In the provided benchmark, we perform a read on the array and 'throw away' the value by assigning it to a volatile.

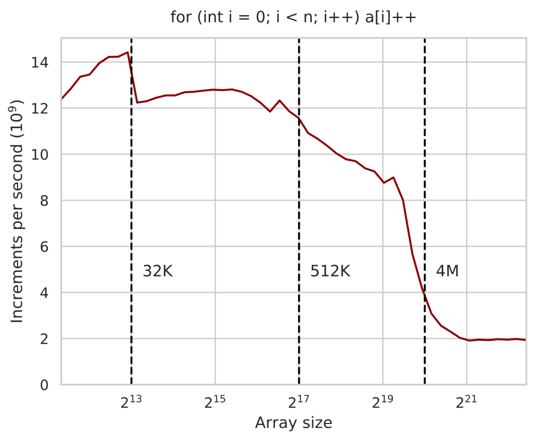

Figure 3: Cascade Lake X Data Access Throughput

To get points, you have to pass 2 basic test cases (1 point) as well as 2 advanced test cases (2 points) after pushing your code to GitLab. The basic test composes of comparing the bandwidth and latency of each cache level against each other, while the advanced test requires to return the values close to a 'ground truth' which is a reference implementation ran by the gitlab CI runner.

1. **0.5 point** Basic Bandwidth
2. **0.5 point** Basic Latency
3. **1 point** Advanced Bandwidth
4. **1 point** Advanced Latency

3.3 Sub-task 2: Data Structure benchmarking (6 points)

For the data structure evaluation, you are not provided any benchmarking logic, but have to implement it yourself. What you are given is a benchmarking scenario, consisting of two variables: the size of the dataset (from 16KB up to 512MB) and the access pattern (Sequential and Random) as well as a set of data structures - a **directly accessed array**, a **binary search over a list** and a **chained hash table**. The data structures are stored in **include/Benchmarking.hpp**

Your task is to implement the entire benchmarking logic, which should consist of:

1. Generator of the dataset for a given input size
2. Lookup logic to mimic the access pattern requested

3. Benchmarking logic which loads the dataset onto each data structure and measures their lookup performance in terms of bandwidth and latency.

At the end of your benchmark, for the advanced tests, you must return the bandwidths and latencies of each data structure for the given dataset size and access pattern scenario. The return order is written in the function definition in the repository.

There are 6 advanced test cases you are evaluated on, each worth 1 point:

1. **1 point** Random Access (Small Dataset)
2. **1 point** Random Access (Medium Dataset)
3. **1 point** Random Access (Large Dataset)
4. **1 point** Sequential Access (Small Dataset)
5. **1 point** Sequential Access (Medium Dataset)
6. **1 point** Sequential Access (Large Dataset)

3.4 Sub-task 3: The Report (6 points)

Your final task is to write a 1 to 2 page report (with a font size of 11), outlining your findings from sub-task 2. More specifically, your report should contain a series of plots evaluating the latency and bandwidth of the data structure. The dataset size should denote the x-axis, while the y-axis contains the bandwidth or latency. The report must contain explanations for each data structure as to why it performs better or worse than the others under each scenario.

All claims in the report should be backed up by your understanding of the behavior of the memory hierarchy as well as analysis done using performance counters on the CPU. Key metrics to use for your arguments should be LLC misses, L1 misses, branch miss-predictions, etc.

You gain points for fulfilling the following:

1. **1 point** - explanation of the benchmarking setup (how the dataset is prepared, how the data structures are evaluated, etc)
2. **1 point** - a plot showing the bandwidth of all the data structures across varied access patterns and dataset sizes.
3. **1 point** - a plot showing the latency of all the data structures across varied access patterns and dataset sizes.
4. **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.

5. **1 point** - Compare the performance of the BinarySearch with random access versus sequential access. Is the performance what you expected? If not, why not.

6. **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.

The results for the plots should be obtained using the development VM and not the runner. The runner is only for evaluating the successful completion of sub-tasks 1 and 2.

3.5 Submission guidelines

Provided that you have filled the form given in Moodle, you have been added to a gitlab group with a template project 1 present in the following repository: `https://tg.dm.informatik.tu-darmstadt.de/groups/fomo_ws2025_students`. If not, please contact the TAs.

Fork the repository and modify the code in **src/Benchmarking.cpp** to pass the tests in **test/**

The compilation flow can be found in the README.

Push to remote only when you are comfortable with getting tested by the CI runner.

The 1-page report should be pushed to your repository as a .pdf file. Your name and matriculation number should be written at the top of the report.