Program for testing cache memory transfer rate performance

Student: Alexandru-Andrei Avram

Group: 30432

Table of Contents

1. Project Proposal

The primary objective of this project is to evaluate cache read capabilities using multiple programming languages, namely C, C++, Assembly (ASM), and Python.

The focus is on estimating cache hit/miss rates through execution time measurements, shedding light on how these languages interact with the cache subsystem. Understanding these interactions is important for making informed decisions when selecting languages for performance-critical applications.

2. Project Plan

The main part of the project is written in Python, controlling which script is run at what time, and integrating everything seamlessly using a graphical user interface. Python is the optimal language, since it can be used to run both C and C++ code.

Use of C/C++ with inline Assembly language integration

- Uses /clr (Common Language Runtime compilation).
- When an assembly function is called from C code, the C compiler handles the details of parameter passing and the function call.

Interpreting the results involves understanding the relationship between time, frequency, and clocks of latency. Specifically, if a processor with a clock frequency of 2.4 GHz takes 17 nanoseconds to access a particular level of cache, you can compute the latency in terms of clock cycles as follows:

$$17 \times 10^{-18} \text{ seconds} \times 2{,}400{,}000{,}000 \text{ Hz} = 17 \text{ ns} \times 2.4 \text{ GHz} \approx 41 \text{ Clocks}$$

This conversion allows you to relate cache access times to the number of clock cycles required for that access, providing a more intuitive understanding of cache performance.

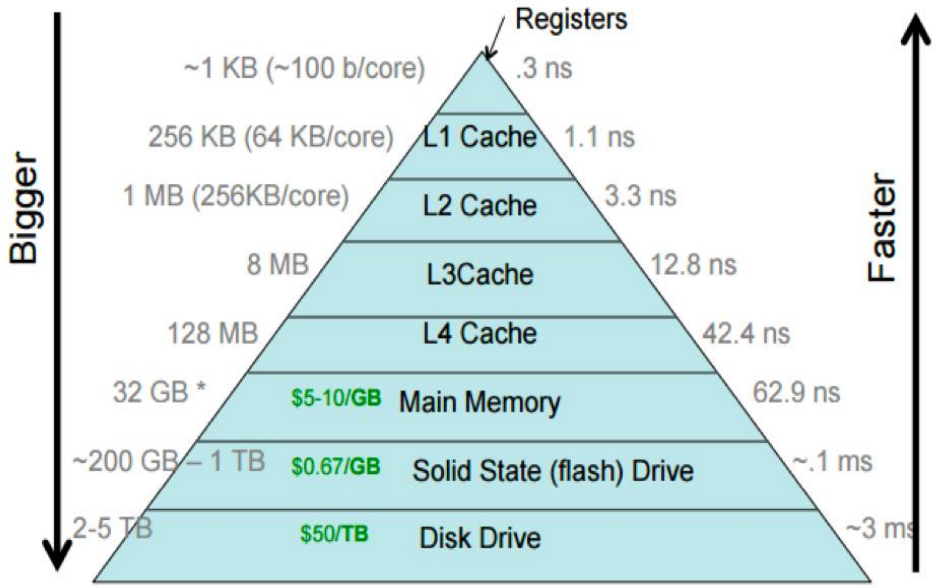
3. Bibliographic Study

a. Cache Hierarchy

L1 Cache: L1 cache is incredibly quick but fairly diminutive in size. It resides right inside the processor chip, ensuring that the CPU can access the most vital and frequently used data almost instantaneously.

L2 Cache: L2 cache acts as the dependable companion to L1. It has more capacity than L1 and can be found either on the CPU or residing on a nearby chip, connected by a high-speed bus. This arrangement spares L2 from the data-cluttered main system bus, allowing it to provide a steady stream of important data to the processor without getting slowed down.

L3 Cache: Despite having lower speed, L3 plays a pivotal role in enhancing L1 and L2, while still generally being twice as fast as the main memory (DRAM). In the realm of multicore processors, each core might have its personal L1 and L2 caches, but they often converge around a shared L3 cache. If an L3 cache references an instruction, it is usually elevated to a higher level of cache.



b. Benchmark Metrics

A cache hit occurs when the CPU successfully retrieves data from the cache without needing to fetch it from a slower level of memory (e.g., main memory).

Cache Hit Rate: Cache hit rate is the ratio of cache hits to the total memory accesses.

Cache Miss Rate: Cache miss rate is the ratio of cache misses to the total memory accesses.

Cache Access Latency: Cache access latency measures the time it takes to access data in the cache once a request is made. It reflects how quickly the CPU can retrieve data from the cache.

Memory Access Latency: Memory access latency is the time it takes to retrieve data from main memory (or a slower storage medium) when it's not available in the cache, thus will be measured when we surpass the cache memory size.

Idea: Read data from main memory. Measure the time it takes to access this data. Do the same for data that fits in cache and then data which does not fit in the cache memory.

Profiling tool: Valgrind

c. Compiler Optimization

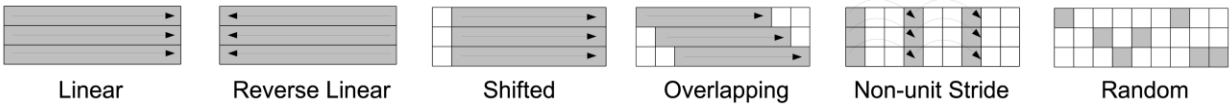
To ensure the benchmarks accurately reflect cache performance, compiler optimizations may be disabled so as not to interfere with the measurements.

Compiler flags like -O0 can turn off optimization.

-O1, -O2, -O3, or –Ofast are the optimization levels available with the GNU C++ compiler. Using -O0 will turn off those optimizations.

d. Data Access Patterns

Within the field of computing, a memory access pattern, often labeled as an IO access pattern, characterizes the specific manner in which a computing system or software application conducts its interactions with the reading and writing of data residing in secondary storage. These patterns exhibit distinct variations in the level of reference locality, and this variance exerts a notable impact on the efficacy of cache operations. Moreover, they are important for planning how to work on multiple tasks at the same time and for dividing the work efficiently in systems that use shared memory.



Sequential (Linear) Access Pattern: Sequential access is a data access pattern characterized by accessing a contiguous and ascending sequence of memory addresses without any omissions. This pattern follows a strict order, progressing in a linear and incremental fashion.

Reverse-Sequential (Reverse-Linear) Access Pattern: A reverse-sequential access pattern involves accessing a contiguous and descending sequence of memory addresses without any omissions. Similar to sequential access, this pattern maintains a strict order but follows a decreasing sequence.

Random Access Pattern: Random access represents a data access pattern where memory addresses are accessed in a non-sequential, erratic manner, exhibiting no particular order or predictability. This pattern entails abrupt shifts across address space.

Strided Access Pattern: Strided access entails accessing a sequence of memory addresses with a consistent and uniform gap (stride) between each referenced address. It can be characterized by the interval between addresses.

e. Other Variables

CPU frequency scaling, also known as dynamic voltage and frequency scaling (DVFS), is a technology that allows a CPU to adjust its clock speed and voltage to match the workload, conserving power and reducing heat production during periods of low demand and maximizing performance when needed. It can impact cache benchmarks by altering the timing of cache operations, potentially leading to variations in cache performance and benchmark results based on the CPU's clock frequency at a given moment.

4. Analysis

Functionalities:

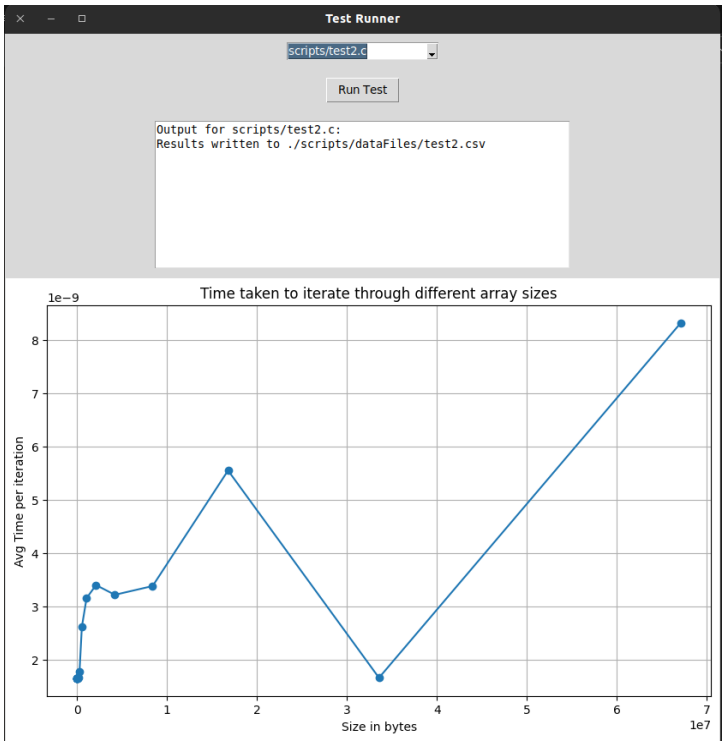
- Measuring the time it takes to read a given array of numbers and estimating if there were a lot or a few cache misses/hits (done by measuring time of execution multiple times – arrayTest.cpp)
- Comparing time taken for the same operation using sequential and random memory access (test.cpp).
- Benchmarking memory access times under different working memory sizes, using a random-access pattern (test2.c)

User-App Interaction:

Done through a simple GUI, which allows the user to select which test should be performed, and, after the selected test is run, displays a graph generated using the results of the run.

The graphs are created using the “pyplot” library. The python program calls one of the test scripts, which then outputs its results in csv format to a new file. After this step, python reads the data from the csv file and generates a graphical representation using the selected columns from the file. By incorporating various charts and graphs, the project presents raw metrics and allows users to

visually discern patterns and trends in the data. This visual approach adds a layer of accessibility, enabling a broader audience to grasp the nuances of cache performance across diverse programming languages.



5. Design

Studied processor and architecture:

AMD Ryzen 5 5600h 7 nm processor 3.3 - 4.2GHz

AMD Zen 3 microarchitecture

384 KB of L1 cache

3MB of L2 cache

16 MB of L3 cache

The chosen hardware architecture, the AMD Ryzen 5 5600h, serves as a fundamental element in shaping the cache performance tests, since cache memory sizes vary from chip to chip.

Considered tests and implementation methods:

There will be multiple C/C++ functions, to run each of the before mentioned tests (from "Analysis" section). Ideally, all these functions will be divided into separate files (one function per file) and accessed and ran through a main function/file which may be written in Python or C++.

A Command Line Interface will provide the user with a range of modular commands to run, in order to benchmark the cache memory, as well as with a help command, which lists all the possible commands and their syntax.

6. Tests

Three distinct tests are conducted, each providing valuable insights into cache performance. By systematically conducting these assessments, the project aims to uncover patterns and behaviors in cache utilization across C, C++, Assembly (ASM), and Python.

Test 1: Array access pattern test

In this test, the focus is set on benchmarking the impact of different array access patterns on cache memory. The code measures the time it takes to access array elements sequentially and randomly, providing insights into how cache behavior can affect performance.

Results for an array of 1 million elements:

Time taken for sequential access: 810 microseconds

Time taken for random access: 24210 microseconds

Test 2: Array access times test

This test is designed for benchmarking memory access times under different working memory sizes, using a random-access pattern. It aims to provide insights into how the performance changes with varying memory sizes.

It uses a piece of assembly code to implement a function named _access_random_place_to_place_memory_dep that performs random memory access through pointer chasing (repeatedly dereferencing pointers to access data at different memory locations).

The access pattern is such that each stride (memory chunk) depends on the result of the last read.

Output sample:

Size in bytes: 128 Iterations 10240000 Avg Time per iteration: 1.71992e-09 seconds

Size in bytes: 256 Iterations 10240000 Avg Time per iteration: 1.73496e-09 seconds

Size in bytes: 512 Iterations 10240000 Avg Time per iteration: 1.7249e-09 seconds

Size in bytes: 1024 Iterations 10240000 Avg Time per iteration: 1.71084e-09 seconds

Size in bytes: 2048 Iterations 10240000 Avg Time per iteration: 1.7127e-09 seconds

Size in bytes: 4096 Iterations 10240000 Avg Time per iteration: 1.71699e-09 seconds

Test 3: Array cache performance measurement

This test is meant to measure the time it takes to access elements in an array and calculate their sum to observe cache performance. If the array fits entirely within the cache, the access time is expected to be relatively fast. If the array size exceeds the cache capacity, there might be more cache misses, resulting in longer access times.

Results for an array of 1 million elements:

Sum: 499999500000

Time taken by the loop: 1030 microseconds

7. Compiler optimization

In the context of cache benchmarking or performance analysis, the absence of compiler optimizations is crucial because it allows you to observe and measure the raw, unaltered performance characteristics of your code. Compiler optimizations are designed to enhance the efficiency of the compiled code by applying various transformations that may change the structure and execution flow of the program. While these optimizations can significantly improve the overall performance of the application, they might obscure or alter the low-level details that cache benchmarking aims to investigate.

Thus, disabling compiler optimization is crucial in this case because:

- Compiler optimizations can reorganize code, inline functions, or perform loop unrolling, which may make it challenging to trace and interpret the exact memory access patterns, especially when investigating cache behavior.
- Cache performance is closely tied to memory access patterns. Optimizations might introduce prefetching, reordering, or other changes that mask the true nature of memory

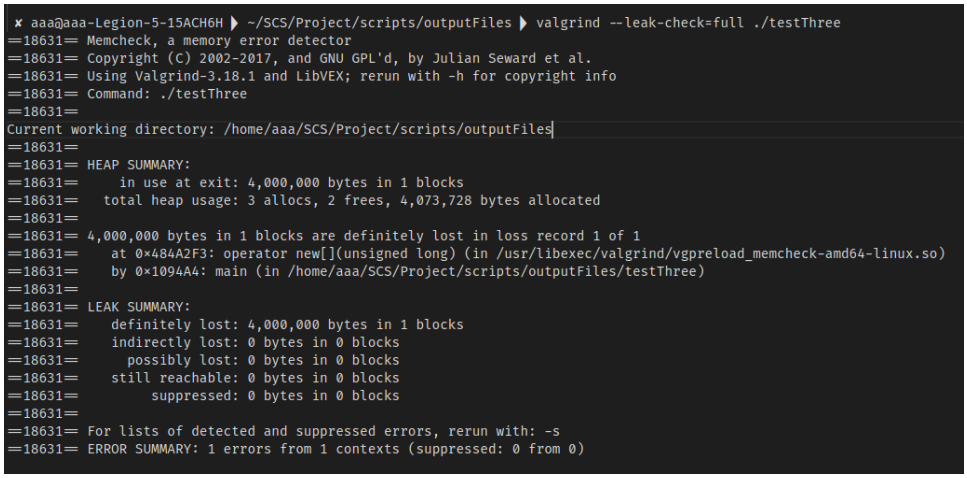
access. Disabling optimizations allows you to observe the real, unaltered memory access patterns.

- In the absence of optimizations, the generated assembly code more closely reflects the original source code. This facilitates easier debugging and profiling since the correspondence between the source code and the generated machine code is more straightforward.

8. Profiling

Tool used: Valgrind

Tools like Valgrind play a crucial role in cache benchmarking and performance analysis by providing detailed insights into the runtime behavior of programs, detecting memory-related issues, and offering profiling capabilities.



- Valgrind helps identify memory leaks, which can be particularly important in cache benchmarking where efficient memory usage is crucial. Memory leaks can lead to unnecessary memory consumption, impacting cache behavior and overall system performance.
- Valgrind provides cache profiling tools (such as Cachegrind) that help analyze cache usage patterns. These tools can reveal cache misses, hits, and other cache-related statistics, allowing developers to optimize code for better cache performance.
- Valgrind is extensible, allowing the development of custom tools for specific analyses. This extensibility makes it adaptable to various profiling and analysis needs, including those specific to cache benchmarking.

9. Reference List

(1) https://stackoverflow.com/questions/9412585/see-the-cache-missess-simple-c-cache-benchmark

(2) https://medium.com/applied/applied-c-memory-latency-d05a42fe354e
(3) https://learn.microsoft.com/en-us/answers/questions/522635/how-to-use-cache-in-c
(4) https://www.cl.cam.ac.uk/~nk480/C1819/lecture8.pdf
(5) https://learn.microsoft.com/en-us/cpp/build/reference/clr-common-language-runtime-compilation?view=msvc-170
(6) Joshua Ruggiero, "Measuring Cache and Memory Latency and CPU to Memory Bandwidth," Intel, 2008 [http://www.csit-sun.pub.ro/~cpop/Documentatie_SMP/Intel_Microprocessor_Systems/Intel_ProcessorNew/Intel%20White%20Paper/Measuring%20Cache%20and%20Memory%20Latency%20and%20CPU%20to%20Memory%20Bandwidth.pdf]
(7) https://patents.google.com/patent/US6754857B2/en
(8) https://www.shiksha.com/online-courses/articles/working-of-cache-memory/ (Needs VPN)
(9) https://www.techtarget.com/searchstorage/definition/cache-memory
(10) https://www.alibabacloud.com/blog/the-mechanism-behind-measuring-cache-access-latency_599384
(11) https://en.wikipedia.org/wiki/Memory_access_pattern
(12) https://www.embecosm.com/appnotes/ean6/html/ch07s03s02.html
(13) https://cs61.seas.harvard.edu/site/2019/Section5/
(14) https://www.vertica.com/docs/10.0.x/HTML/Content/Authoring/InstallationGuide/BeforeYouInstall/cpuscaling.htm
(15) https://www.amd.com/en/technologies/zen-core
(16) https://valgrind.org/docs/manual/cg-manual.html