MINISTRY OF EDUCATION AND RESEARCH

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA, ROMANIA

Performance Benchmark of Cache Transfer Rate

Mihali Vlad

Group:30434

Technical University of Cluj-Napoca

Cuprins

1. Introduction

1.1. Context

The goal of this project is to design and implement a benchmark for cache memory that evaluate the performance of transferring data in cache memory, intern memory and virtual memory. We test pure cache memory (without memory management).

The outcome of the program will be general information about memory size, charts that represents comparison of different memory transfer rate and tables with different miss and hit information.

1.2. Specifications

In this project, we will use C programming language to collect data about memory, especially cache memory. I decided to use C programming language because is low-level language (no automatic memory management) and the time measured will be more accurate.

Also, for the graphical user interface, we will use Java programming language to have an easy interface and better view over the charts and data about memory.

1.3. Objectives

The main objective of this project is to analyze performance of cache memory in comparison with internal and virtual memory.

The tasks that achieve the main goal are:

- Study about the subject;
- Create benchmarks in C programming language for:
    - Measure the cache size for every cache level and RAM size (internal memory)
    - Measure hit time for every cache level, RAM memory
    - Measure miss ratio of every cache level
    - Create graphical user interface in Java programming language that process data from benchmarks. It will include tables and also some options to run different benchmarks and to access this documentation to help you to understand this benchmark.

- Test final application (also tests will be done on every benchmark when implemented)

2. Bibliographic study

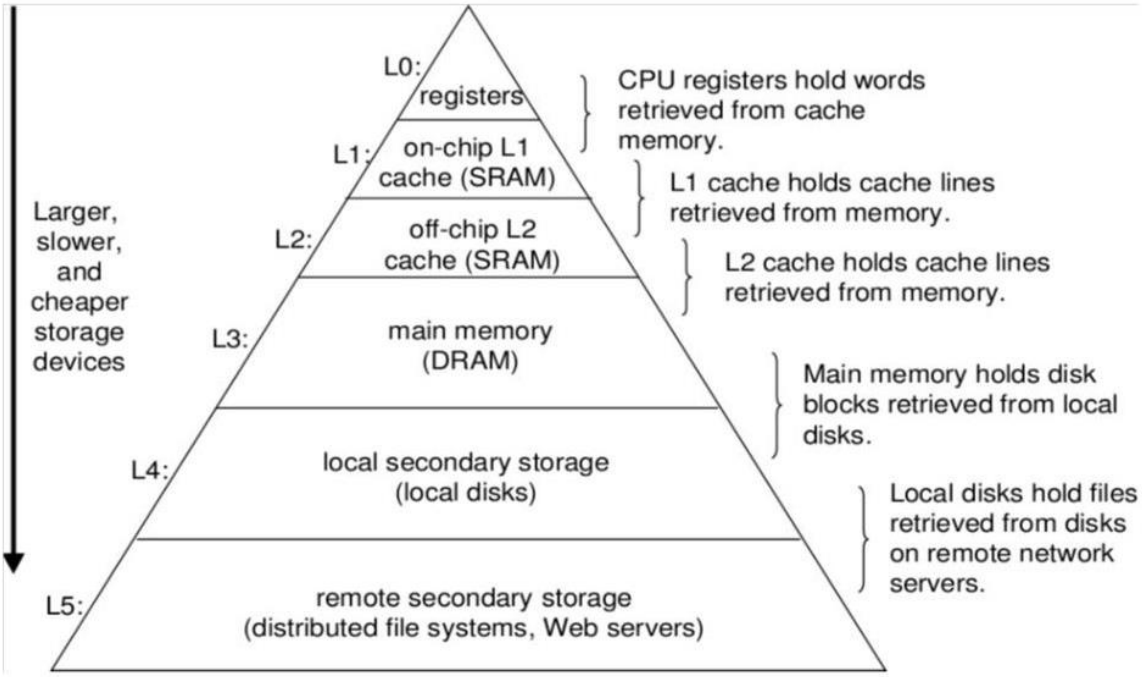
Memory in a device is structure on multiple levels, in a hierarchical manner (distance from CPU). By going further from CPU, memory levels will decrease in speed and increase in size.

Cache memory is the first level of memory, it is called also immediate memory. Cache memory is divided in multiple levels:

- L1 cache, smallest and fastest, it is inside the CPU;
- L2 cache, bigger than L1 cache but slower, it is mainly inside the CPU;
- L3 cache, biggest and slowest cache level (if exists), it is outside of the CPU. It is used to share data between cores, if CPU is single core than probably L3 cache is missing.

RAM (Random Access Memory) is a form of computer memory that can be read and changed in any order. It allows data items to be read or written in almost the same amount of time irrespective of the physical location of data inside the memory. It is also called main memory and it is 10 times slowest than L1 cache and typically 1GB – 64GB in size (> cache).

Virtual memory, also called secondary memory is not a volatile memory in contrast to others level mentioned above. Forms of this memory are: HDD, SSD, CD, DVD, BD, etc. .



A miss occurs when you need to request data from a specific memory level but that specific data it is not on that level. A memory miss occurs either because the data was never placed in the cache, or because the data was removed.

Types of miss:

- Compulsory misses – when we do on purpose;
- Conflict misses – when data whose there, but got deleted (evicted);
- Capacity misses – the data you trying to request is bigger than memory size.

Parameters	Compulsory misses	Conflict misses	Capacity misses
Larger memory size	No effect	No effect	Decrease
Larger block size	Decrease	Uncertain effect	Uncertain effect

Power law of cache misses: $M=M_0*C^{-\alpha}$

Where M is the miss rate for a cache of size C and M_0 is the miss rate of a baseline cache. The exponent α is workload-specific and typically ranges from 0.3 to 0.7, with an average of 0.5.

This delay that occurs due to cache misses are known as miss penalties.

Hit occurs when the data request it is in the specific memory level. It is opposite to miss. Hit ratio is calculated: 1-M where M is miss ratio.

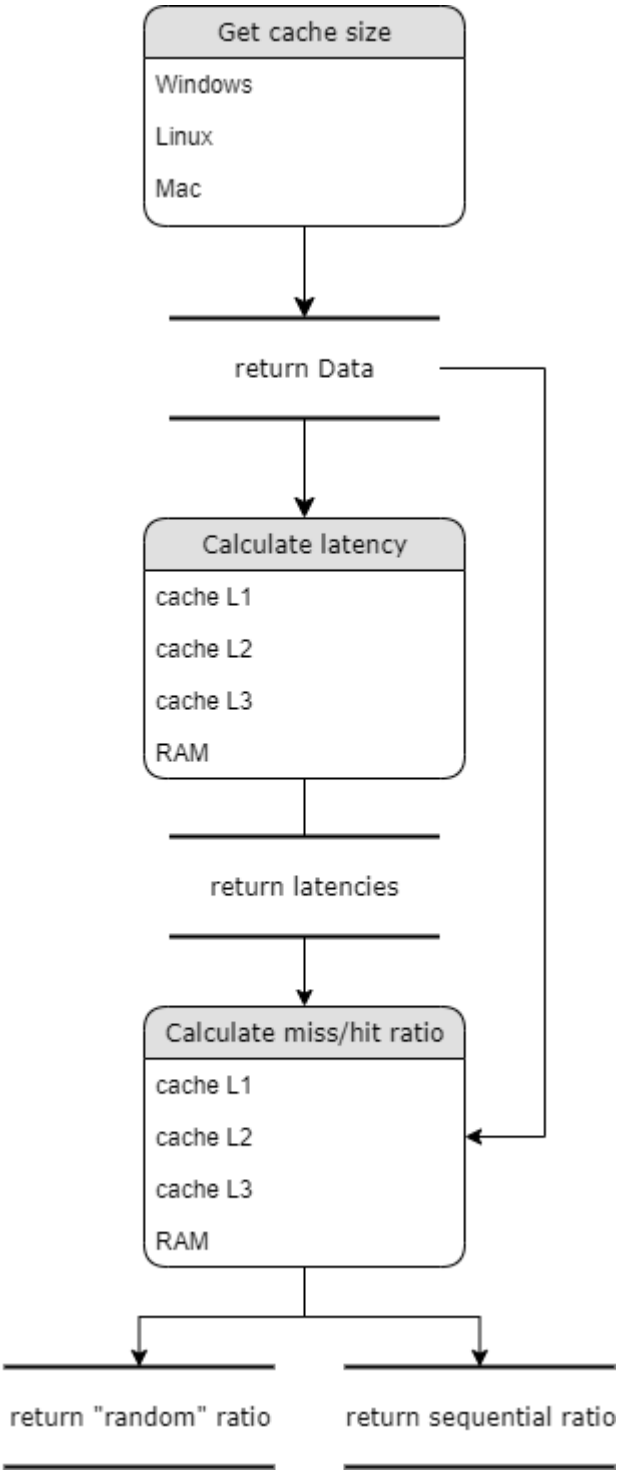
Average memory access time is calculated with:

AMAT = T + MR * MP, where T the access latency (time) of the memory that we apply this formula, MR is the miss ratio and MP is the average memory access time for the upper level.

3. Analysis

In this documentation of the Performance Benchmark of Cache Transfer Rate project, we are using chart to see the flow of the application and for a better understanding and a well structure presentation.

To the right of the page, it is presented a flowchart of the benchmarks. Each benchmark is represented by a big box. Inside of each box are rows that represent particularity or branch of the respective benchmark. Lines with data are inputs and outputs of the benchmarks that are interconnected.



Simply, the result of this project is to calculate average memory access time. So, to do this we need to use three benchmarks:

- Get cache size;
- Calculate latency;
- Calculate miss/hit ratio.

We use this to compute the result directly or indirectly.

Get cache size is more a retrieve information from the operating system then a benchmark. It is very difficult to get this data without operating system, because cache is well masked. Indeed, this is the scope of the cache, to not be visible to application. We tried to compute L1 cache size, but we received a much smaller size then in reality. The big problem was in L2 cache, every time we accessed it we got a peek and then cache prefetching guessed the next step and invalidate our data set. Same happened with L3 and RAM. We decided that is best to use the well build functions of the biggest three operating systems:

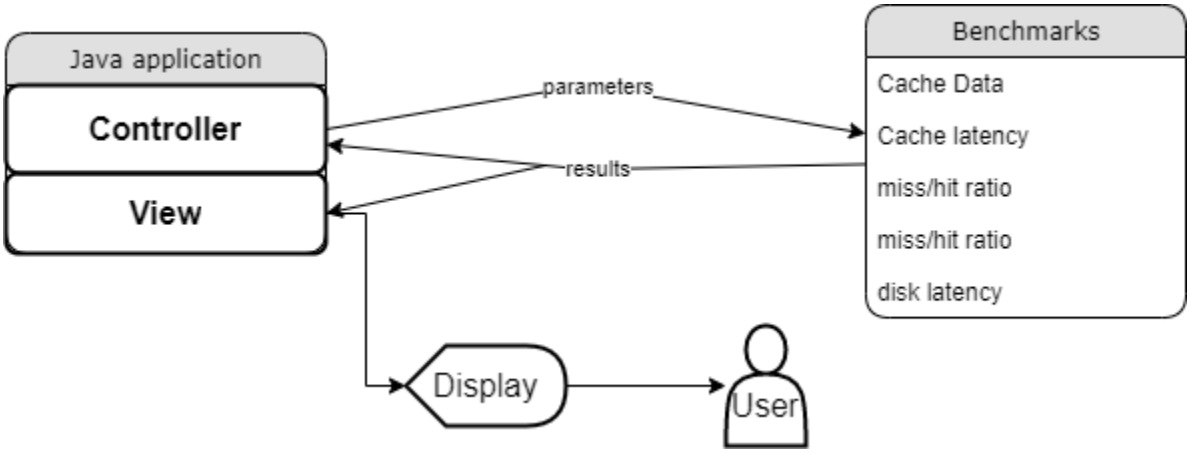
- Windows;
- Linux;
- Mac.

"Calculate latency" benchmark is the main benchmark of this project. Side note, all the benchmarks that refers to cache results in an approximation, because the cache is hidden from the program and can be tricky. To run this program, we need inputs that we obtain from previous benchmark. First, we measure for RAM memory and then for L1 cache, bypass L1 cache, L2 cache, bypass L2 cache and finally L3 cache. This data is measured for different access blocks and multiple time, to get the result more

accurate. Memory latency is the time (the latency) between initiating a request for a byte in memory until it is retrieved by a processor. If the data are not in the processor's cache, it takes longer to obtain them, as the processor will have to communicate with the external memory cells. Latency is therefore a fundamental measure of the speed of memory: the less the latency, the faster the reading operation.

Calculate miss/hit ratio benchmark is a copy of the previous one. The difference is what we measure. We need to add bypass to L3 cache also. When bypassing a cache, we measure how many times we hit that level. In addition, for L2 cache and L3 cache we need to keep track of how many times we hit an upper level. We have two types of results depending on how we access that memory: sequential or random.

In this java part of this application we create a Controller and a View. All data that is output from benchmarks is processed by the Controller and pass as input for the next benchmark. View also processed data from benchmark, but it is used to display in a friendly user interface in chart, tables to satisfy the user.



4. Design

For each operating system we reatrive date about cache. For Windows, we use GetLogicalProcessorInformation from the library windows.h. For Mac, we use sysctlbyname from the library sys/sysctl.h. For Linux, we use data from files in that are located in "/sys/devices/system/", more specifically in "/sys/devices/system/cpu/cpu0/cache".

We collect following information about internal structure of the device:

- total cache size in kb for each level
- Line size
- Number of CPU cores
- Number of logical processors
- RAM total size

If the platform is unrecognized the program stops here.

To calculate latency, we access data from an array. First, we calculate for RAM because we access it first time and it is majority in RAM. For time measurement, it is used clock_gettime function. Index is incremented by step on average of line size (from previous benchmark). Divide overall time by numbers of time we access that level. Store the result in an array (each level has its own array). To access L2 cache we need to bypass L1 cache, that means invalidate L1 by accessing all elements of array which is larger than cache. To access L3 cache we need to bypass L2 cache, that means invalidate L2 by accessing all elements of array which is larger than cache. We measure this for every block from size 100 bytes to L1 cache size (not to aced L1 cache size because we invalidate the result). Then, we make an average latency time for all cache level and RAM. Also, we need to divide L1 to processors number and L2 to cores number.

To calculate miss ratio, we have the same algorithm as in "calculate latency" benchmark. We will measure in the bypass part of the program. We measure the access time of every element and categorized in which type belongs. There are three type:

- If time << level latency we ignore
- If time ~= level latency it is a hit
- If time >> level latency it is a miss

Miss ratio is miss counter divided by sum of miss and hit counter and hit ratio is 1 – miss ratio.

It is one big problem with this algorithm, if the program is run more than once per boot then will affect the data set because of the cache prefetching. Cache prefetching is a technique used by computer processors to boost execution performance by fetching instructions or data from their original storage in

slower memory to a faster local memory before it is actually needed. In same cases we need to print same date to stop prefetching.

The other algorithm is it to access the memory before bypass in a random manner. Random need to be separate for CPU logic and out of the memory usage (maybe big pseudo-random array). Measure time for each element and compute miss ratio as previous algorithm. Unfortunately, we are limited in dimension because it needs to be significant smaller than size level. This means that the result is not so accurate. Also, in this part we measure RAM hit ratio or it will be very close to 100%.

In addition of the design of this work, we have another benchmark. It is a simple one. The scope is to measure the time it takes to read from file at first try, that it is indeed the latency time for the disk.

Therefore, we have all the data to calculate average memory access time:

AMAT = T + MR * MP, where T the access latency (time) of the memory that we apply this formula, MR is the miss ratio and MP is the average memory access time for the upper level.

5. Implementation

5.1. First Benchmark

It is meant to collect data from the system and outputs them in a file.

Most important data in this benchmark is the number of processors for each cache level. It is calculated as:

```
Cache = &ptr->Cache;
if (Cache->Level == 1)
{
    processorL1CacheCount++;
}
else if (Cache->Level == 2)
{
    processorL2CacheCount++;
}
else if (Cache->Level == 3)
{
    processorL3CacheCount++;
}
   break;
```

To find the information of the system we use:

```
(LPFN_GLPI) GetProcAddress(
                           GetModuleHandle("kernel32"),
                              "GetLogicalProcessorInformation");
```

5.2. Second Benchmark

This benchmark collects another data from the system. Most important are cache size for every level, ram size and line size.

Information is received by:

```
GetLogicalProcessorInformation(0, &buffer_size);
buffer = (SYSTEM_LOGICAL_PROCESSOR_INFORMATION *)malloc(buffer_size);
 GetLogicalProcessorInformation(&buffer[0], &buffer_size);
```

Cache size for each level is calculated as following:

```
for (i = 1; i != buffer_size / sizeof(SYSTEM_LOGICAL_PROCESSOR_INFORMATION); i++) {
        if(buffer[i].Cache.Level == 1){
            cachesize[0]+=buffer[i].Cache.Size;
        }else if(buffer[i].Cache.Level == 2){
            cachesize[1]+=buffer[i].Cache.Size;
            line_size = buffer[i].Cache.LineSize;
        }else if(buffer[i].Cache.Level == 3){
            cachesize[2]+=buffer[i].Cache.Size;
        }
       }
```

RAM size is from the following code:

```
MEMORYSTATUSEX statex;
statex.dwLength = sizeof (statex);
GlobalMemoryStatusEx (&statex);
 fprintf(fp,  "RAM %d MB \n\n",statex.ullTotalPhys/DIV);
```

5.3. Latency Benchmark

This benchmark computes the latency for each cache level.

The inputs are:

```
int line_size = atoi(argv[6]);
int cpu_cores = atoi(argv[5]);
int cpu_logical_processores = atoi(argv[4]);

const int L1_CACHE_SIZE = (atoi(argv[1])*KB/cpu_logical_processores);
const int L2_CACHE_SIZE =(atoi(argv[2])*KB/cpu_cores);
const int L3_CACHE_SIZE =(atoi(argv[3])*KB);
```

To measure time in ns we use:

```
clock_gettime(CLOCK_MONOTONIC, &startAccess); //start clock
……….//something you wont to measure
```

```
    clock_gettime(CLOCK_MONOTONIC, &endAccess); //end clock
      time = ((endAccess.tv_sec - startAccess.tv_sec) * SECONDS_PER_NS) +
   (endAccess.tv_nsec - startAccess.tv_nsec);
```

This is what we measure:

```
    while (index < j) {
        int tmp = arrayAccess[index];
        index = (index + tmp + ((index & 4) ? line_size-4 : line_size+4)); //skip
someware around cache line size
        count++;
       }
```

Variable j is an iteration of the benchmark from size 100 bytes to size of level 1 cache. It helps to make an average about the result (latency).

This is how to skip cache level:

```
//invalidate LX by accessing all elements of array which is larger than cache
    for(count=0; count < LX_CACHE_SIZE; count++){
        int read = arrayInvalidateL1[count];
        read++;
        readValue+=read;
       }
```

The flow of the benchmark is: measure RAM, measure level 1 cache, skip level 1 cache, measure level 2 cache, skip level 2 cache, measure level 3 cache.

5.4. Miss Ratio Benchmark

It is almost the same benchmark as latency benchmark, but the difference is what we measure.

Inputs are:

```
    int line_size = atoi(argv[6]);
    int cpu_cores = atoi(argv[5]);
    int cpu_logical_processores = atoi(argv[4]);
    int speedL1 = ((int) ceil(atof(argv[7])))*100;
    int speedL2 = ((int) ceil(atof(argv[8])))*100;
    int speedL3 = ((int) ceil(atof(argv[9])))*100;


    const int L1_CACHE_SIZE = (atoi(argv[1])*KB/cpu_logical_processores);
    const int L2_CACHE_SIZE =(atoi(argv[2])*KB/cpu_cores);
   const int L3_CACHE_SIZE =(atoi(argv[3])*KB);
```

This is what we measure:

```
for(count=0; count < L1_CACHE_SIZE; count++){
        clock_gettime(CLOCK_MONOTONIC, &startAccess);
        int read = arrayInvalidateL1[count];
        clock_gettime(CLOCK_MONOTONIC, &endAccess);
        read++;
        readValue+=read;
    double hit = ((endAccess.tv_sec - startAccess.tv_sec) * SECONDS_PER_NS) +
(endAccess.tv_nsec - startAccess.tv_nsec);
        if(hit>=speedL1){
            miss++;
        }
        printf( "%d---%.12f\n",count,hit);
       }
```

……

```
    for(count=0; count < L2_CACHE_SIZE; count++){
        clock_gettime(CLOCK_MONOTONIC, &startAccess);
        int read = arrayInvalidateL2[count];
        clock_gettime(CLOCK_MONOTONIC, &endAccess);
        read++;
        readValue+=read;
        double hit = ((endAccess.tv_sec - startAccess.tv_sec) * SECONDS_PER_NS) +
(endAccess.tv_nsec - startAccess.tv_nsec);
        if(hit>=speedL2){
            miss2++;
        }
        if(hit>=speedL1){
            count2++;
        }
        printf( "%d---%.12f\n",count,hit);
       }
```

……

```
   for(count=0; count < L3_CACHE_SIZE; count++){
        clock_gettime(CLOCK_MONOTONIC, &startAccess);
        int read = arrayInvalidateL3[count];
        clock_gettime(CLOCK_MONOTONIC, &endAccess);
        read++;
        readValue+=read;
        double hit = ((endAccess.tv_sec - startAccess.tv_sec) * SECONDS_PER_NS) +
(endAccess.tv_nsec - startAccess.tv_nsec);
        if(hit>=speedL3){
            miss3++;
        }
        if(hit>=speedL2){
            count3++;
        }
        printf( "%d---%.12f\n",count,hit);
```

```
}
```

For preventing the prefetch we must print some stuff for each iteration.

5.5. JavaFx program

We use javaFx to show data in gui and connect all the benchmarks.

How to run a c program in java (for miss ratio):

```
Process process = Runtime.getRuntime().exec("cmd /c start /wait scs3.exe "
    +cache1.getSize()+" "
    +cache2.getSize()+" "
    +cache3.getSize()+" "
    +numberOfProcessors+" "
    +numberOfCores+" "
    +lineSize+" "
    +cache1.getLatency()+" "
    +cache2.getLatency()+" "
    +cache3.getLatency()+" ");
process.waitFor();
```

This is the method to extract all doubles from a file:

```
private List<Double> getNumbersFromFile(String path){
  List<Double> list = new ArrayList<>();
  File file = new File(path);
  BufferedReader reader = null;

  try {
    reader = new BufferedReader(new FileReader(file));
    String text = null;

    while ((text = reader.readLine()) != null) {
      List<String> split = Arrays.asList(text.split("[ ,/:]+"));
      for (String maybeInt: split) {
        try {
          list.add(Double.parseDouble(maybeInt));
        } catch (Exception e) {
          continue;
        }
      }
    }
  } catch (FileNotFoundException e) {
    e.printStackTrace();
  } catch (IOException e) {
    e.printStackTrace();
  } finally {
```

```
    try {
      if (reader != null) {
        reader.close();
      }
    } catch (IOException e) {
    }
  }
```

We need this to parse files from benchmark.

Data are store as:

```
private int numberOfCores;
private int numberOfProcessors;
private int lineSize;
private CacheInfo cache1;
private CacheInfo cache2;
private CacheInfo cache3;
private int ramSize;
private double ramLatency;
private double amat;
```

and CacheInfo as:

```
private int size;
private double latency;
private double missRatio;
private double missPenalty;
```

The most important data is miss penalty for each cache size and AMAT which is calculated as:

```
cache3.setMissPenalty(ramLatency);
cache2.setMissPenalty(cache3.getLatency()+cache3.getMissRatio()/100*cache3.getMissPenalty());
cache1.setMissPenalty(cache2.getLatency()+cache2.getMissRatio()/100*cache2.getMissPenalty());
amat = cache1.getLatency()+cache1.getMissRatio()/100*cache1.getMissPenalty();
```

We establish a benchmark hierarchy, the higher benchmark needs to call the all the benchmark bellow him.

AMAT benchmark => Miss Ratio benchmark => Latency benchmark => Information benchmark

AMAT benchmark and Miss Ratio benchmark takes 10 to 15 minutes to run.

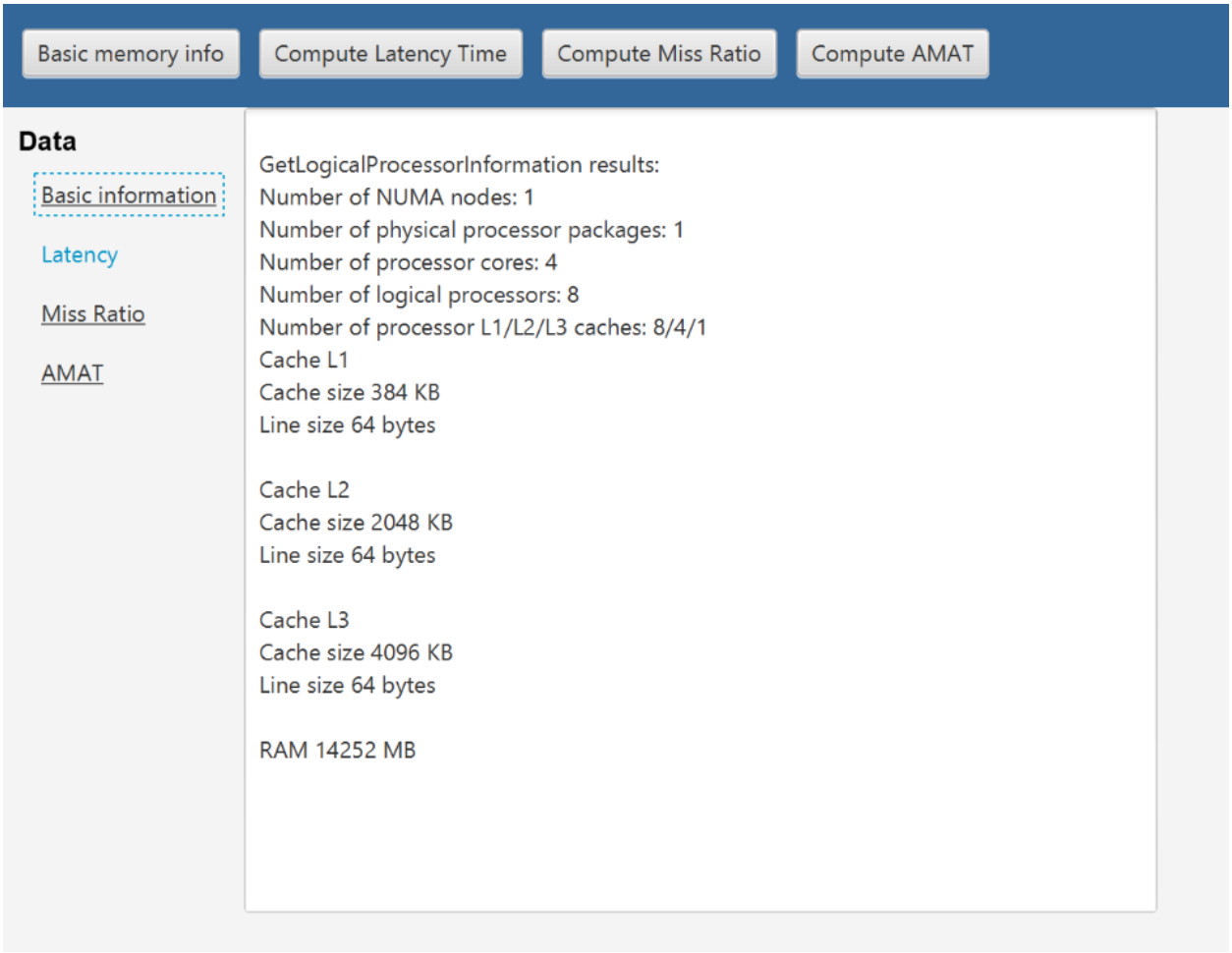
The others need no more than 10 seconds.

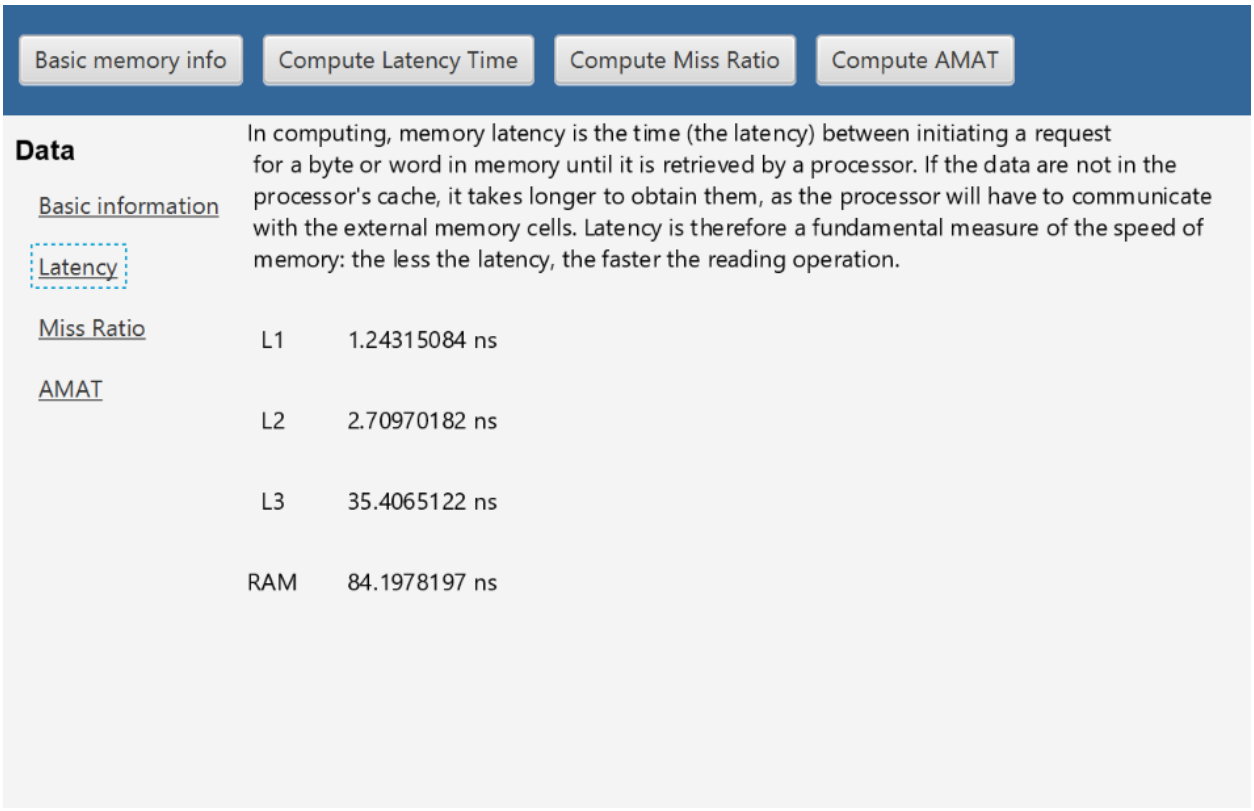
6. Testing

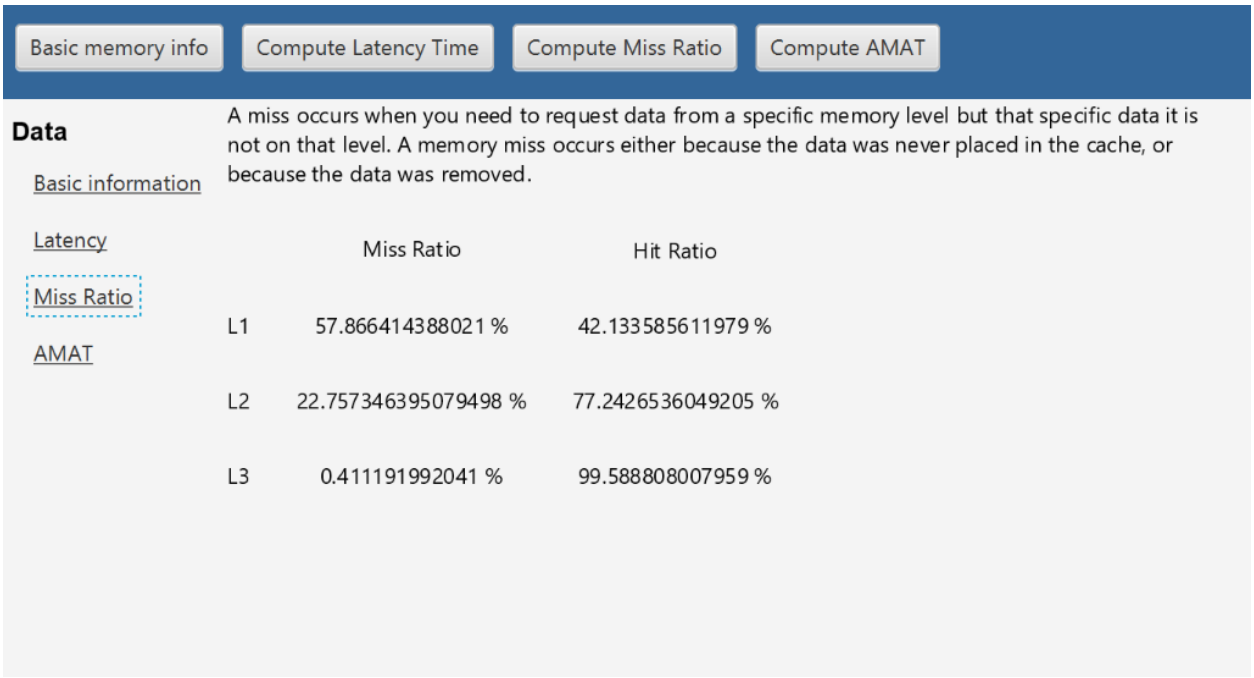
To have good performance we recommend following:

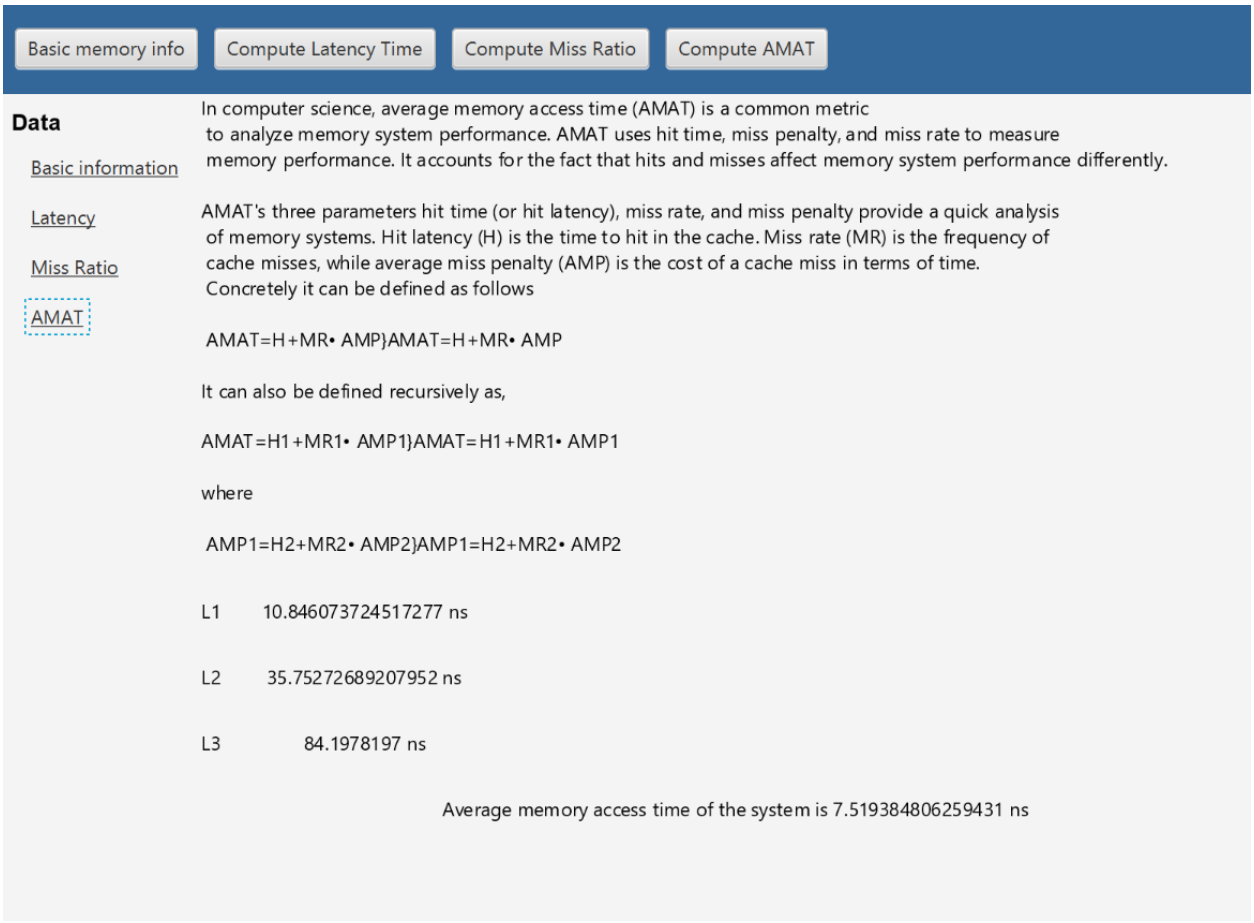
- Set Power Mode to Best Performance;
- Do not run multiple application;
- The test needs to be done ones per boot to have accurate data.

!!!! When testing please do not close the command-line that appears from the benchmarks (Miss Ratio benchmark and AMAT benchmark). There should be 2 command-line per benchmark.




Basic memory info
Compute Latency Time
Compute Miss Ratio
Compute AMAT
Data
Basic information
Latency
Miss Ratio
AMAT
In computing, memory latency is the time (the latency) between initiating a request for a byte or word in memory until it is retrieved by a processor. If the data are not in the processor's cache, it takes longer to obtain them, as the processor will have to communicate with the external memory cells. Latency is therefore a fundamental measure of the speed of memory: the less the latency, the faster the reading operation.
L1 1.24315084 ns
L2 2.70970182 ns
L3 35.4065122 ns
RAM 84.1978197 ns


Basic memory info
Compute Latency Time
Compute Miss Ratio
Compute AMAT
Data
Basic information
Latency
Miss Ratio
AMAT
A miss occurs when you need to request data from a specific memory level but that specific data it is not on that level. A memory miss occurs either because the data was never placed in the cache, or because the data was removed.
Miss Ratio
Hit Ratio
L1
57.866414388021 %
42.133585611979 %
L2
22.757346395079498 %
77.2426536049205 %
L3
0.411191992041 %
99.588808007959 %



7. Conclusions

In conclusion, the program measures some key information about the memory of the system. The application runs all the benchmarks described previously. Please take in consideration that Latency and Miss Ratio benchmarks are an approximation by measurement and sometimes is not reliable. The most important thing is that the benchmark computes the average memory access time.

This project can be improved in more points and we can do this by a deeply documentation about these subject that results in a more accurate result. In this project, we just touch the peak of the iceberg. Unfortunately, we made some assumptions and adjustments, but can be fixed in the future. In the next release, we can add to GUI the option to store benchmarks.

8. Bibliography

https://en.wikipedia.org/wiki/Memory_latency

https://www.extremetech.com/extreme/188776-how-l1-and-l2-cpu-caches-work-and-why-theyre-an-essential-part-of-modern-chips

https://wp-rocket.me/blog/calculate-hit-and-miss-ratios/

https://www.cloudflare.com/learning/cdn/what-is-a-cache-hit-ratio/

https://www.sciencedirect.com/topics/computer-science/cache-miss-rate

http://homepage.divms.uiowa.edu/~ghosh/4-1-10.pdf

https://blog.stackpath.com/cache-hit-ratio/

https://en.wikipedia.org/wiki/Average_memory_access_time

https://www.cs.uaf.edu/2011/spring/cs641/lecture/04_05_modeling.html

http://homepage.divms.uiowa.edu/~ghosh/2-28-06.pdf

https://stackoverflow.com/questions/40350872/calculating-average-time-for-a-memory-access

https://www.sciencedirect.com/topics/computer-science/average-access-time

http://home.ku.edu.tr/comp303/public_html/Lecture15.pdf

https://stackoverflow.com/questions/59257656/how-do-you-find-the-miss-penalty-of-a-single-level-cache

https://www.geeksforgeeks.org/multilevel-cache-organisation/