Cache Performance in Memory Hierarchy: A Cross-Simulators Analysis using SimpleScalar and Gem5

Elvis Fernandez
dept. Electrical and Computer Engineering
Florida International University
Miami, Florida
efern207@fiu.edu

Kevin Lopez
dept. Electrical and Computer Engineering
Florida International University
Miami, Florida
klope108@fiu.edu

Narek Vanetsyan
dept. Electrical and Computer Engineering
Florida International University
Miami, Florida
nvane007@fiu.edu

Abstract—**This paper focusses on Cache Performance in Memory Hierarchy and provides an cross-complier simulation analysis using two open source simulators: Gem5 and SimpleScalar. Memory hierarchy organization is a complex subject and requires the understanding of how all pieces interact together to provide the best possible performance. Cache performance is an important metric in modern computer systems and it is considered to be crucial for many applications, especially real time applications. Thus, it is important to understand what parameters impact cache performance and how they do it. To help understand cache performance in memory hierarchy, this work explores cache performance through simulations using two different simulators: SimpleScalar and Gem5.**

Keywords—Memory Hierarchy, Cache, Performance, Simulators, SimpleScalar, Gem5

I. INTRODUCTION

Access time to main memory is one factor that negatively impacts performance in modern computer systems. Typically, Random Access Memory (RAM) is physically placed far away from the cores, which incurs excess latency. This, in addition to an accentuated gap between CPUs and Memory speeds, contributes to increased core access time to memory space. To reduce such memory access time, more expensive and smaller memories are placed between the CPUs and RAM, creating a Memory Hierarchy as shown in Figure 1. The farther away from the CPUs, the longer it takes to access that memory space. Conversely, the closer that memory space is to the CPUs, the more expensive it becomes.

The memory space between CPU and RAM is known as Cache memory. Cache memory organization has evolved from a single level cache to a multilevel cache memory subsystem where cache sharing is possible in the presence of multicore systems. However, while cache contributes to lower the access time to main memory, it is considered the primary performance bottleneck for processing real-time and hard real-time applications on single-core and multi-core systems [1]. Thus, it is important to understand what parameters impact cache performance and how they do it. To help understand cache performance in memory hierarchy, this work explores cache performance through simulations using two different simulators: SimpleScalar and Gem5.

This paper is organized as follows. In Section II, the paper discusses the team's motivation to work on the subject. Section III reviews related articles and literature that provide a background in Memory Hierarchy, focusing in Cache memory. The paper details the team methodology to address the research in Section IV. The Results of the simulations are discussed in Sections V and VI. Finally, the work is concluded in Section VII while Section VIII highlights the future work.

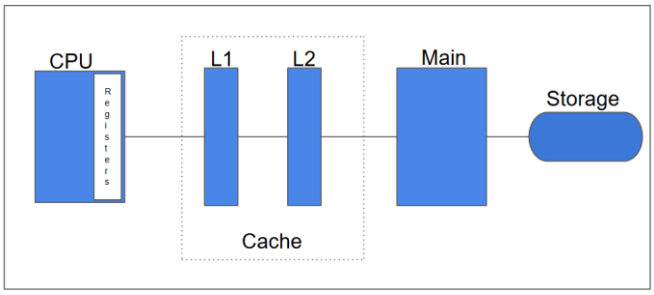

Fig. 1. Cache in Memory Hierarchy.

II. MOTIVATION

Memory hierarchy organization is a complex subject and requires the understanding of how all pieces interact together to provide the best possible performance. For example, single core CPUs from twenty years ago would have a single unified cache whereas today's multicore systems deploy multilevel cache systems where some levels of cache are even shared. A lot has changed in the way designers implement memory architecture and there are several proven techniques or combinations of them that help achieve improved performance.

There is plenty of Literature describing such techniques and fortunately there are several simulators that allow benchmarking of computer systems including cache performance. The caveat is that not all Instruction Set Architectures (ISAs) are supported in a single simulator, or they are limited by the number of CPUs, etc. We started investigating several available simulators and found that in some instances there is not a lot of information regarding how to get started with them. Thus, this work focuses on describing how to get started with two of the more popular simulators we found and compares simulation results from both in a cross-simulation analysis.

III. BACKGROUND

In this section we will review the main concepts behind cache memory and the parameters that impact its performance from updated literature and published research articles.

A. *Cache Memory*

Cache memory in computer systems defines a fast, but relatively small memory space compared with the next level in the Memory Hierarchy, that can speed up the access time between a level and the one above that is cache. It can speed up access to all manners of storage devices, including tape drives, disk drives, main memory, servers on the network (e.g., web servers are a type of storage device), and even other caches [2]. However, the term is mainly utilized to describe the memory space between the CPUs and the RAM in the Memory Hierarchy and it is widely accepted and referenced as such in the Literature as illustrated in Figure 1.

The physical composition of cache are SRAM cells and there are many SRAM bit-cells architecture being implemented, mainly to solve the native issues a simple 6T SRAM bit-cell possesses, such as half select disturb, read/write conflicts, voltage constraints which all translate into reliability issues. There are 6T, 8T, 9T, 10T, 12T, and 14T SRAM architectures, each with a benefit but the more reliable as of this work our team found is the 7T/14T Dependable SRAM described in [4]. Figure 2 illustrates a simple 6T SRAM cell which is the most basic SRAM bit-cell [3] and Figures 3 illustrates a 7T/14T SRAM Dependable cell as described in [4] .

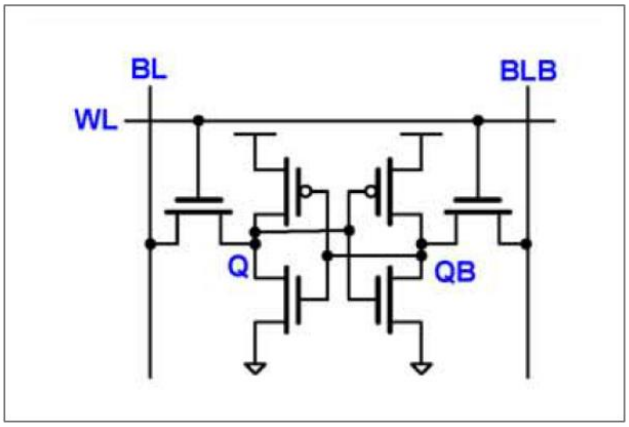

Fig. 2. 6T SRAM bit-cell [3].

Cache memory concepts are based on the principles of locality which point to that once a datum is referenced in a program it is likely that the program references the same datum again or a nearby data as well in the near future [2]. The tendency of programs to use data items over and again during the course of their execution is known as Temporal Locality. The tendency of programmers and compilers to cluster related objects together in the memory space is known as Spatial Locality [2]. Are the principles of locality what makes caches so useful because once data is requested from the CPU, there will be nearby data brought to cache as well. So, future requests can be fulfilled from cache instead, reducing access to main memory.

Per the CPU request data is brought into cache in blocks or cache lines (chunks of data from RAM), and organized in sets (a group of blocks). Then, a tag is added to every data entry to indicate whether a particular datum is present in cache and once the allocated cache space is full there are mechanisms also known as replacement policies to return data to RAM and bring new data to cache. Thus, cache is constrained by its size and the size of the blocks: the larger the cache the better the hit rate, but then hit time is trade off. Large cache sizes can and should exploit large block sizes, and this couples well with the tremendous bandwidths available from modern DRAM architectures [2].

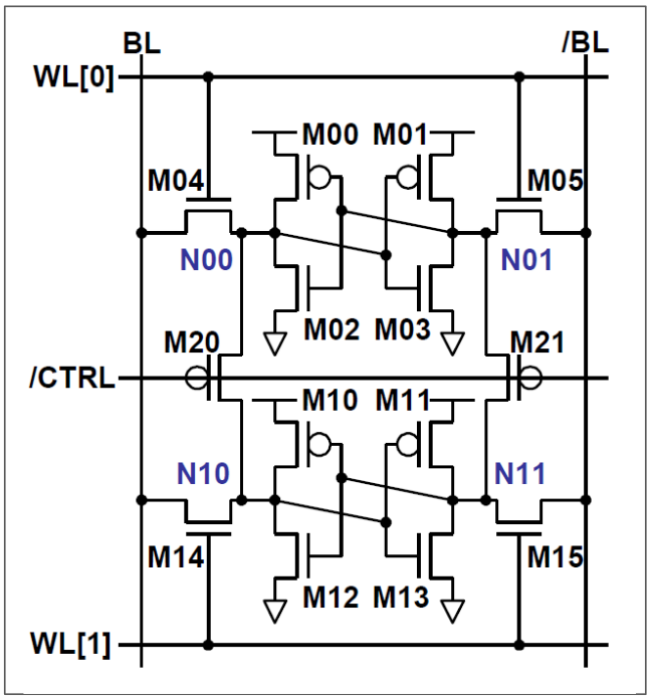

Fig. 3. 7T/14T Dependable SRAM described in [4].

B. *Cache Organization*

There are three basic cache organizations that address the issue of where to place every block brought from main memory: direct mapped, fully associative, and set associative. Figure 4 illustrates all three organizations.

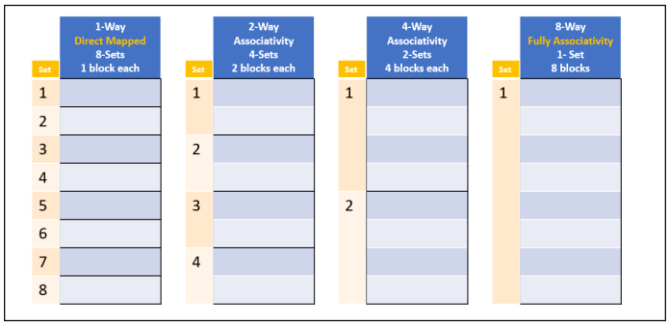

Fig. 4. Cache Organization. From Direct mapped to the left to Fully Associative to the right. Set-Associative in the middle.

Direct mapped (1-way associativity) is the simplest way of cache organization: there is only one block per set. It is a simple technique and it is easy to implement in hardware, but it can impact hit time, for example, if multiple blocks are mapped to the same space in cache they might have to be referenced continuously and brought in and out, which is known as trashing.

Fully associative caches (also called content-addressable memories) have only one set that encompasses all blocks. It is expensive to implement, complex hardware and it consumes more power, but it provides flexibility, full utilization of cache and It can positively impact hit rate.

Set Associativity is the tradeoff between both direct mapped and fully associative. Therefore, any attempt to increase performance will naturally start with the assumption that a set associativity will provide the optimal performance. Out of the three possible approaches i.e. direct, associative and set associative, set associative caches are considered best due to their better hit rate and less access time. However, it has been reported that beyond a certain point, increasing cache size has more of an impact than increasing associativity. [5].

C. *Replacement Policies*

Replacement policies or algorithms are used to attain optimized use of Cache. When cache is full, then replacement policies decide which piece of data is replaced in order to make space for new data that is currently being used. An efficient algorithm is that which can take less time and number of cache misses are low and also balancing cost [6]. There are many replacement policies for cache, including Least Recently Used (LRU), Least Frequently Used (LFU), History Least Recently Used (HLFU), Segmented LRU (SLRU), First In First Out (FIFO), Random Replacement (RR), Clock with Adaptive replacement (CAR), Greedy Dual Size (GDS), LR+5LF Algorithm, and Lowest Latency First (LLF). These replacement algorithms are sometimes classified in several classes as shown in Figure 5

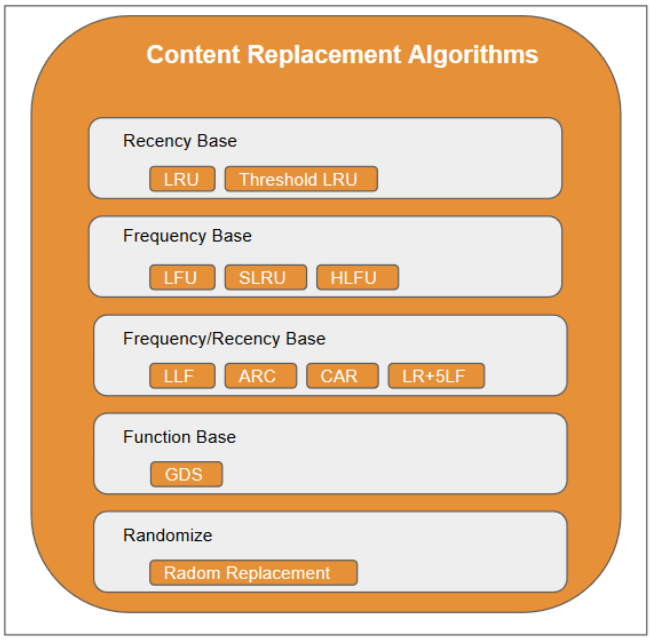

Fig. 5. Contect Replacement Algorithms listed by classes.

Current generation of multiprocessors implement mainly four of these policies: Random, FIFO, LFU and LRU. Out of these, LRU is commonly the most used replacement algorithm because it is based on the principle of temporal locality of access, which makes it appropriate for most applications. Recent studies describe cache design space with relatively finite associativity, and consider only true Least Recently Used replacement policy [7]. The Least Recently Used replacement policy uses access pattern of a program memory to predict that most recently accessed cache line will most likely to be accessed again, and the cache line which has been Least Recently Used (a block in the set that is in cache for the longest and having no reference to it) will be replaced by cache controller. [8] To keep track of the entries, the LRU algorithm uses a status, but as the associativity increases the number of bits also increases to maintain a record for each block. Therefore, some designers trade of performance for less complex algorithms such as Random replacement. Random replacement policy chooses a candidate block to be discarded randomly from all the cache lines in the set. This policy does not need to keep any information of access history.

LFU policies use a counter to keep records for each block that is loaded into cache. Once the cache is full, the block with the lowest counter is removed. LFU is good for keeping certain regularly used data, such as the root directory of a file system, in the cache. However, without some modification, it would not be very effective in handling data references due to locality of access [2]

The first in first out algorithm removes the page that has not been used for a long time. It treats the pages as a circular buffer, and pages are removed in around robin fashion [8]. FIFO is probably no better than random selection in choosing the least likely to be used data for replacement [2].

D. *Cache Optimization*

Cache performance can be directly related to its impact in memory access time. Cache optimization is achieved by reducing hit time, reducing miss penalty, increasing cache bandwidth and by reducing miss rate. There are several optimization techniques to improve cache performance, but it is important to highlight that increased performance in an area, might mean trading off some other. Thus, a tradeoff must be made by the designer, so the final result ends up in reducing memory access time.

To reduce miss rate, a simple solution is to increase cache and block size. However, while both can reduce miss rate, it can also increase miss penalty, which in terms may increase memory access time. Larger block size cache conjointly increases the conflict misses. In distinction smaller block size cache reduces cache hit times. Multi-level caches are quite economical in up the overall system performance but hierarchy style of multi-level caches is extremely complicated [9].

Higher associativity has also been pursued to reduce miss rate. Set-associativity is a tradeoff between direct mapped and fully associativity. With the increase in associativity, there is also an increase in complexity and power consumption and there is an increase in latency.

Miss penalty can be reduced by implementing multilevel cache. Adding another level, or multiple levels of cache, contributes to reducing the miss penalty, but it is more complex and comes at a power cost as well. Today's systems typically implement 2 levels of cache and more advanced complex ones introduce three levels. In multicore systems, multiple levels of cache are present and even cache resources are shared. Cache coherence protocols are employed to solve the problem of inconsistent data that may arise from updating or writing on the same memory location by different multiprocessor cache [10]. Another way to reduce miss penalty is to give priority to read misses over writes through a write buffer. However, write buffers complicates memory accesses [11].

To reduce hit time, a solution is to avoid address translation during indexing of the cache. Full virtual addressing for both indices and tags eliminates address translation time from a cache hit [11]. Other techniques are used in current designs to increase cache performance, including hybrid and adaptive multilevel cache architectures, scalable software-defined caches, non-uniform caches, etc [5]

IV. METHODS

A. *Simulators Options*

Our project consists of taking two simulation tools and running a series of experiments to verify cache performance across various factors. To get to this point we started by experimenting with SimpleScalar, GEM5, Dinero IV and Moola. When attempting to use these cache performance simulators we noticed that there were some major differences between tools. Moola was created by the University of North Texas. Since this simulation tool was a dissertation project, we found it very difficult to find resources and examples regarding install instructions and simulations. This tool was created as a subset of Dinero IV. Which is where we decided to take a step back and look into the tool encompassing Moola. Dinero IV was created by the University of Wisconsin-Madison. This tool, though very powerful, has tons of limitations as it is still in its primitive stages of development. The tool is open source but requires a lot of programming to get a simple simulation output. As stated before, many students take up years to develop programs with Dinero IV to encompass a simulator to their liking. Since we do have a time constraint we decided to move on with our research and find a tool that was ready for simulations.

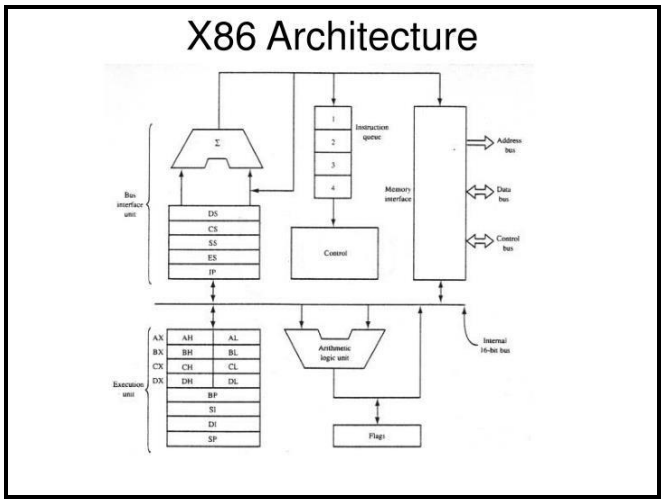

Fig. 6. Block Diagram for an x86 Instruction Set Architecture.

B. *Gem5 Simualtor*

GEM5 is a CPU simulation tool that, when constrained properly, can simulate different aspects of cache. Though we had a lot of research to do, we found that many of the parameters we wanted to change were possible with GEM5. The GEM5 website provides a manual which helps with installation and getting started with your first CPU simulation. With extensive research we found a set of commands that allowed us to simulate cache performance. Note that we were looking to change parameters such as cache size for both level 1 and level 2, block size and associativity. All of which were possible with Gem5. The instruction set architecture used on GEM5 is x86. x86 is a family of instruction set architectures initially developed by Intel based on the Intel 8086 microprocessor and its 8088 variants. Figure 6 illustrate the basic architecture for x86 ISA.

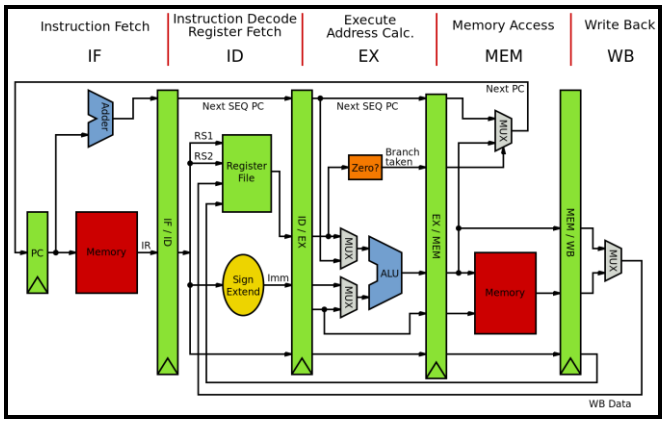

Fig. 7. Block Diagram for MIPS Instruction Set Architecture.

C. *SimpleScalar Simulator*

SimpleScalar is a simulation tool which calls an option that allows targeted simulation. The simulation tool has a lot of tools to choose from such as sim-fast, sim-cache and sim-cheetah. In our case we decided to use the sim-cache option to simulate the different cache parameters discussed in the paragraph above. Since this tool is highly leveraged for educational purposes, finding resources was very simple. We were able to find sample commands to change parameters such as cache size for both level 1 and level 2, block size and associativity. The instruction set architecture used for SimpleScalar is PISA. PISA is a MIPS-like instruction set architecture. MIPS is a reduced instruction set computer instruction set architecture developed by MIPS Computer Systems, now MIPS Technologies, based in the United States. Figure 7 shows a block diagram for a MIPS ISA.

D. *The Bechmark Program (Binary)*

One of the goals for the simulation was to be able to simulate the same benchmark binary on both simulators, so the analysis was done with as many fixed variables as possible. The team first explored SPEC benchmark programs; however, encountered several challenges, including the need for paid licenses. Some of the binaries the team was able to construct from SPECs using open sources code were bzip2, mcf, hmmer, sieng, and lbm, but they would present cross-compiling issues over different ISAs. Then, a generic matrix multiplication (matmul.c) program was considered and tested in both simulators. The matmul.c micro-benchmark program cross-complied successfully in SimpleScalar and Gem5, so it was selected for the simulations. The micro-benchmark matmul.c is a simpler program, not as demanding as other SPEC benchmarks, yet it is a suitable open source option to understand the impact of the optimization parameters.

E. *Golden Command*

Time is always a scarcity, so the team planned to find a common command that would allow repeated iterations through both simulators, and if possible, the use of scripting to run the simulations. Looking into what parameters would provide a representation of the cache performance on both SimpleScalar and Gem5. As previously stated, level 1 and level 2 cache size, along with its block size and associativity will be changed would provide a sizable variable in cache behavior. Both GEM5 and SimpleScalar allow to do this with a simple command parameter change. Since it's as simple as changing a number, we decided that it would be time beneficial to create a script to iterate between the value we want to vary. Figure 7 shows the "Golden" command that was used to iterate for both GEM5 and SimpleScalar between all the variable parameters.

SimpleScalar Sample Command:

```
./sim-cache      -cache:dl1:128:16:1:l
                 -cache:dl2
                 ul2:1024:256:4:l
                 matmul.ss 50 50 50
```

GEM5 Sample Command:

```
./build/X86/gem5.fast configs/example/se.py --ruby
                 --num_cpu=1
                 --l1d_size=128
                 --cacheline_size=16
                 -c matmul.x86 --options='50 50 50'
```

Fig. 8. Sample commands used during the simulation.

In addition to identify a the golden commands, the team leverage python to script the iterations. Figure 8 illustrates a sample code for Level 1 cache and block size.

```
import os

l1size=["2kB","4kB", "8kB", "16kB", "32kB", "64kB"]
cachelinesize=["16", "32", "64", "128", "256"]

os.chdir("/home/<user>/gem5")

for x in l1size:
   for y in cachelinesize:
      os.system("./build/X86/gem5.fast configs/example/se.py --ruby
--l1d_size=" + x + " --cacheline_size=" + y + " -c matmul.x86
--options='50 50 50'")
      os.system("cp /home/narek/gem5/m5out/stats.txt
/home/<user>/Desktop/Results/" + x + "_" + y + ".txt")
      print("done with " + x + "_" + y)
```

Fig. 9. Python sample code for Level 1 cache and block size iterations in Gem5

V. RESULTS

The simulations were grouped as follows: cache and block size, cache associativity, multi level cache and cache misses. The cache and block size section will iterate level 1 cache size between 2KB, 4KB, 8KB, 16KB, 32KB and 64KB. It will also iterate the block size between 16, 32, 64, 128 and 256. The cache associativity section will iterate level 1 cache with a size of 8KB through 2, 4, 8, and 16 of an associativity parameter. The multi level cache section will keep level 1 cache to 8KB while iterating level 2 cache between 8KB, 16KB, 32KB, 64KB, and 128KB. Lastly, the cache misses section will iterate level 1 cache size between 8KB, 16KB, 32KB, 64KB and 128KB while also iterating between 2, 4, and 8 of associativity.

A. *Cache and Block Size*

Figure 10 shows the results of the cache and block size section. We note that when cache size goes up, miss rates fall. We can also see that when block size goes up, miss rates go up. This leads us into thinking that is it a better idea to enlarge cache size while keeping the block size to its minimum. When it comes to comparing GEM5 versus SimpleScalar we note that GEM5 has better cache miss rates. This can be due to the instruction set architecture of GEM5. As we can recall from the previous section, GEm5 has an instruction set architecture of x86 while SimpleScalar has an instruction set architecture of PISA or MIPS.

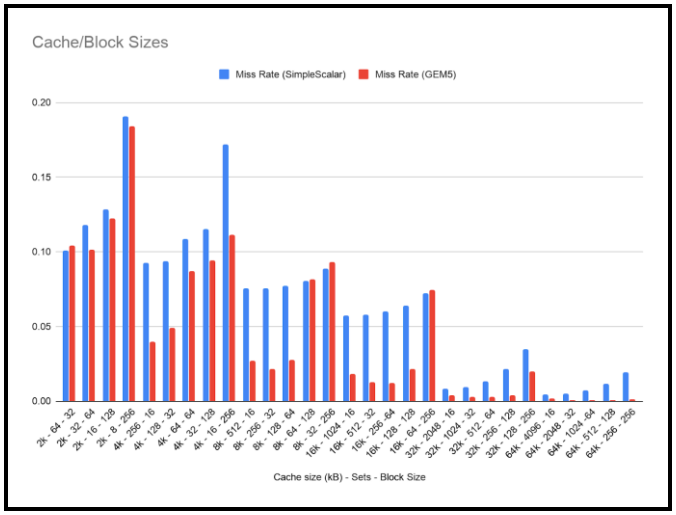

Fig. 10. Cache and block size results

B. *Cache Associativity*

Figure 11 shows the results for cache associativity simulation. We note that there is very little change in the results for SimpleScalar. We believe that the instruction set of MIPs has to do with this small change in miss rates. As for GEM5 we note that there is more change in comparison to SimpleScalar but this change is still within a magnitude of reason. The miss rate for GEM5 is still smaller in all cases in comparison to SimpleScalar.

C. *Multi Level Cache*

Figure 12 shows the results for the multi level cache simulation. We note that adding an extra level of cache itself helps reduce the miss rate heavily on both the SimpleScalar and GEM5 simulation. It is almost by a factor of 2 that these results better in. From Figure 12 we can also confirm the finding in part A. As cache size goes up the miss rate goes down.

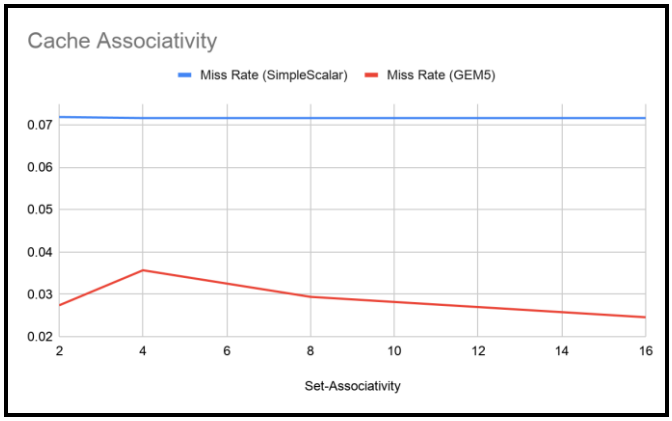

Fig. 11. Cache Associativity results.

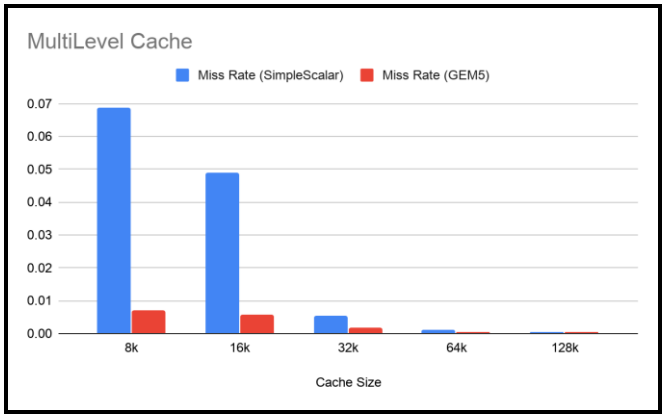

Fig. 12. Multi-Level Cache results.

D. *Cache Misses*

Figure 13 shows the results for the different types of cache misses. The figure, though cryptic, shows three types of cache misses. The first one is compulsory; this means that every first access is to a block. This is due to the cache being empty, therefore, no hits are possible. They are also known as cold-start misses or first-reference misses. The second one is capacity miss which occurs when cache cannot contain all the blocks needed during execution of a program. Therefore, capacity misses will occur because of blocks being discarded and later retrieved. The last type of cache miss is conflict miss which occurs if the block placement strategy is set-associative or direct-mapped. Conflict misses will occur because a block may be discarded and later retrieved if too many blocks map to its set. The idea is that hits in a fully associative cache that become misses in an n-way set-associative cache are due to more than n requests on some popular sets. They are also known as collision misses.

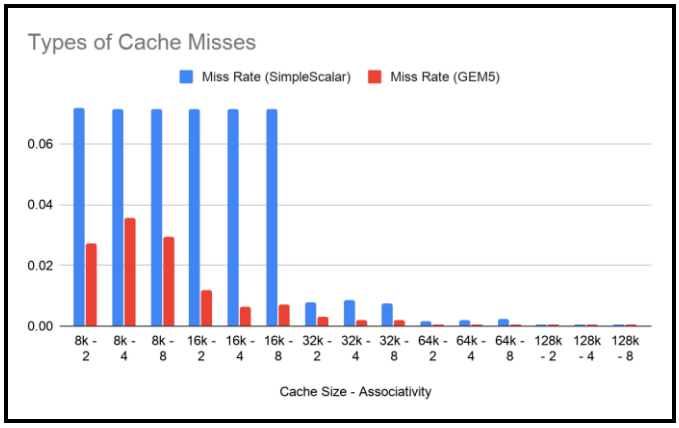

Fig. 13. Types of Cache Misses results.

VI. Discussion

Cache is used as a faster means of accessing data that would otherwise be stored in RAM, which, although significantly larger, is further away and much slower. Cache's role in overall processing speed is crucial because its aim is to allow more frequently used instructions and data to be more readily available to the CPU. However, the configuration of cache makes a big difference in how well the system can operate.

A. *Overview*

In our work, we have primarily focused on comparing two cache simulation programs running very similar or identical simulations: SimpleScalar and Gem5. The former is from a previous assignment and we wished to find another program so that we could perform a cross-simulators analysis. After some research, we settled on Gem5; it was able to install and run the way we needed it to, and we were able to understand how to set up various cache configurations inside it. In SimpleScalar, the PISA ISA is used, and in Gem5 we compiled for x86. We did investigate other ISAs, such as Alpha (64-bit RISC), and MIPS (also RISC), but we were unable to compile them or to build with them. Not every ISA has full support in the tool. x86 worked for us and we were able to compile binaries with it. Of course, this also lets us see potential differences between ISAs when running simulations.

B. *Simulation Results*

We ran a large number of simulations and changed various options, such as cache sizes and associativities. Our primary objective was to capture hit and miss rates, as this would allow us to compare the results of the two simulation programs as closely as possible. With many simulations being run, we could also establish patterns with the collected data and see if they fall within expected ranges.

Because quite a few simulations had to be run, it was worth trying to automate the process. We wrote some Python scripts that ran commands in Terminal and copied the Gem5 output files to our own directory. We could then open the files and extract the necessary data. See Fig. 14 for a screenshot of a directory with simulation output files automatically created by one of the Python scripts.

As we saw in the Results section above, there is a strong connection between cache/block sizes and miss rate: the larger the cache and block sizes, the lower the miss rate becomes. There were some minor inconsistencies, but this can be expected because the complexity of cache and the many unknowns behind the simulation tools leaves many variables out of our control.

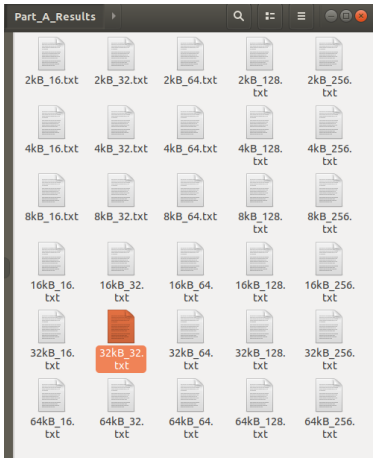

Fig. 14. Directory showing 30 output files automatically created.

C. *Simulator Details*

There is a little more detail we can discuss about the simulators and what we obtained from them. There were a few differences, but we kept everything as similar to each other as we could in the two tools.

In our work, SimpleScalar operated with the PISA ISA and Gem5 operated with the x86 ISA. The configurations for each simulator were set up a bit differently; this involved going through many configuration files (mostly Python scripts) and we even ended up changing a few default values so that they matched SimpleScalar and allowed us to use our automation efforts with the correct values. Gem5's results did have some more inconsistencies: the miss rate had a little bit of a fluctuation to it that was not there for SimpleScalar's results. The difference was minor but worth making note of.

We made sure to keep cache sizes and associativites the same during each simulation. The number of simulations were also the same for both tools so that we could compare the data that way. One major factor to note is that every simulation on both tools ran on a single CPU core. Gem5 does have options for configuring multiple CPUs and cores, but we needed to adhere to the configurations used in the SimpleScalar runs.

VII. CONCLUSION

Our work was meant to showcase the similarities and differences between two simulation programs: SimpleScalar and Gem5. We learned a lot about configuring the two programs (namely, Gem5 for this work), and about creating simulations as identical to each other as possible in both tools. We clearly showed that simulation results followed each other very closely, thus proving that regardless of the tool or the ISA, the concept of hit and miss rates were presented correctly.

VIII. FUTURE WORK

We have not exhausted the possibilities that this work has started. There are many extra combinations of commands for cache simulation that are available in both tools. As an example, a huge cache size with low associativity could be simulated just for comparison. Another possible future work is investigating the possibility of running these simulations on multiple CPUS or cores. Gem5 does have built-in support but it may require some extensive work to set up. Other future possibilities for continuing this work are setting up other simulation tools, finding other binaries and programs to run cache simulations on, and trying more than two cache levels. Not every tool supports more than two levels and it does become much more difficult to configure those extra levels.

REFERENCES

[1] A. Asaduzzaman, "An efficient memory block selection strategy to improve the performance of cache memory subsystem," *14th International Conference on Computer and Information Technology (ICCIT 2011)*, Dhaka, 2011, pp. 12-17.

[2] Jacob, Bruce, David Wang, and Spencer Ng. Memory systems: cache, DRAM, disk. Morgan Kaufmann, 2010.G.

[3] Y. Chiu et al., "40 nm Bit-Interleaving 12T Subthreshold SRAM With Data-Aware Write-Assist," in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 61, no. 9, pp. 2578-2585, Sept. 2014.

[4] Y. Chen, Y. Tang, Y. Liu, A. C. -. Wu and T. Hwang, "A Novel Cache-Utilization-Based Dynamic Voltage-Frequency Scaling Mechanism for Reliability Enhancements," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 25, no. 3, pp. 820-832, March 2017.

[5] M. T. Banday and M. Khan, "A study of recent advances in cache memories," 2014 International Conference on Contemporary Computing and Informatics (IC3I), Mysore, 2014, pp. 398-403.

[6] Qaisar Javaid, Ayesha Zafar, Muhammad Awais, Munam Shah. Cache Memory: An Analysis on Replacement Algorithms and Optimization Techniques. Mehran University Research Journal of Engineering and Technology, Mehran University of Engineering and Technology, Jamshoro, Pakistan, 2017, 36 (4), pp.831-840. hal-01700364

[7] Cantin J. F, Hill M. D., Cache Performance of the SPEC CPU2000Benchmarks,http://www.cs.wisc.edu/multifacet/misc/spec2000 Cachedata

[8] S. Kumar and P. K. Singh, "An overview of modern cache memory and performance analysis of replacement policies," 2016 IEEE International Conference on Engineering and Technology (ICETECH), Coimbatore, 2016, pp. 210-214.

[9] M. A. Kumar and G. A. Francis, "Survey on various advanced technique for cache optimization methods for risc based system architecture," 2017 4th International Conference on Electronics and Communication Systems (ICECS), Coimbatore, 2017, pp. 195-200.

[10] Z. Al-Waisi and M. O. Agyeman, "An overview of on-chip cache coherence protocols," 2017 Intelligent Systems Conference (IntelliSys), London, 2017, pp. 304-309.

[11] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Cambridge (Mass.): Elsevier, 2019.

[12] A. Asaduzzaman, "An efficient memory block selection strategy to improve the performance of cache memory subsystem," 14th International Conference on Computer and Information Technology (ICCIT 2011), Dhaka, 2011, pp. 12-17.

[13] M. F. Mridh, A. Asaduzzaman and A. K. Saha, "An effective measurement technique of level-2 cache performance for multicore embedded systems," 2013 International Conference on Informatics, Electronics and Vision (ICIEV), Dhaka, 2013, pp. 1-4.

[14] S. Priya, et. al., "OPTIMIZED DIRECT METHOD ROUTING FOR CACHE MEMORY IN RISC ARCHITECTURE," in The IIOAB Journal, Issue: Engineering and Technology, India, 2018, pp. 30-35.

[15] B. Batson and T. N. Vijaykumar, "Reactive-associative caches," Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques, Barcelona, Spain, 2001, pp. 49-60.

[16] Dasgupta, Sanjoy and Edouard Servan-Schreiber. “Cache Behaviour of the SPEC95 Benchmark Suite.” (1996).

[17] S. Shen et al., "TS Cache: A Fast Cache With Timing-Speculation Mechanism Under Low Supply Voltages," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 28, no. 1, pp. 252-262, Jan. 2020.

[18] S. Zhou, S. Katariya, H. Ghasemi, S. Draper and N. S. Kim, "Minimizing total area of low-voltage SRAM arrays through joint optimization of cell size, redundancy, and ECC," 2010 IEEE International Conference on Computer Design, Amsterdam, 2010, pp. 112-117.

[19] “SimpleScalar LLC,” *SimpleScalar LLC*. [Online]. Available: http://www.simplescalar.com/. [Accessed: 21-Apr-2020].

[20] “Standard Performance Evaluation Corporation,” *SPEC*. [Online]. Available: http://spec.org/. [Accessed: 21-Apr-2020].

[21] “Sponsors,” gem5. [Online]. Available: https://www.gem5.org/. [Accessed: 21-Apr-2020]