Performance Analysis of Cache Size and Set-Associativity Using gem5

Clayton Asada[1], Joseph Chorbajian[2], and Ilmaan Zia[3]

[1,2,3]Computer Engineering and Computer Science, California State University Long Beach

***Abstract*—Keywords - Cache, Size, Associativity, Replacement Policy, gem5, Simulation, Configuration, Miss rate**

I. Introduction

Cache performance is critical to the performance and efficiency of modern computer systems. As processors have become increasingly fast, the memory and storage hierarchy have struggled to keep up, leading to a performance gap between processors and memory systems. Caches are small, fast memories that sit between the processor and main memory to help bridge this performance gap and supply the processor with data and instructions quickly[1].

The design of caches involves navigating multiple trade-offs between cache size, associativity, block size, and replacement policies in order to achieve optimal performance for a given workload. Larger cache sizes can store more data, reducing the number of slow accesses to main memory, but have other trade-offs such as having higher access latencies and consuming more power. Associativity refers to the number of locations in the cache that can store a given memory block. Higher associativity reduces conflict misses that occur when two memory blocks map to the same cache location, exchanging for additional hardware cost. Block size determines how much data is brought into the cache at once on a cache miss. Larger block sizes can improve spatial locality and reduce miss rates, but they also bring in more unused data, wasting cache space. Replacement policies determine which cache line is evicted when the cache is full and a new memory block needs to be brought in. More effective replacement policies can improve cache miss rate by reducing the number of useful cache lines evicted[1].

Simulation tools, like gem5[2] and SimpleScalar[3], are useful for exploring the cache design space and evaluating trade-offs. These tools run simulated workloads and collect various cache metrics across different configurations. This allows designers to find the optimal balance of size, associativity, block size, and replacement policy for a target workload without having to build physical hardware prototypes.

In this paper, we show optimizations of cache miss rates through navigation of these trade-offs. We compare the miss rates on a total of 72 configurations per benchmark by varying cache size, associativity, block size, and replacement policies. Other cache parameters have remained constant. We use three benchmarks from the MiBench benchmark suite, designed for embedded processors. As such, the cache size, block size, and set associativity all match appropriate configurations for embedded processors.

II. Related Work

As cache performance is crucial for modern computer systems, computer architecture researchers community has recognized its significance. As a result, cache design has been extensively studied and optimized to improve system performance, with researchers exploring various cache configurations through simulation over decades.

For instance, researchers have explored trade-offs between cache replacement policies. In 2004, Al-Zoubi et. al.[4] evaluated cache replacement policies on the CPU2000 benchmark suite using SimpleScalar and found that LRU and LRU-based policies, specifically pseudo-LRU replacement policies, performed best. More recent studies by Panda et. al.[5] shows that modern LRU-based replacement policies outperforming pseudo-LRU using the CPU2006 benchmarks.

Cache size has also been extensively studied. Ma et. al.[6] used SimpleScalar to evaluate miss rates across multiple benchmarks in SPEC2000 and found that a larger L1 cache size significantly improves performance when the cache size is small, but gradually decreases as the cache gets bigger. Ullah et. al.[7] built on this work by proposing a 2:1 cache rule of thumb, which states that the miss rates of a cache with n cache size and x associativity are equivalent to the miss rates on a configuration with $n/2$ cache size and $2x$ associativity.

While most research on cache configurations and miss rates have been done in SimpleScalar, some research has used gem5. Reas et. al.[8] have simulated L2 cache miss rates on m5 (the precursor to gem5) and compared them to miss rates on hardware implementations of their design on the SPLASH-2 benchmark suite.

III. Motivation

As stated before, cache performance is a critical factor for any computer system as it plays a significant role in improving system performance and energy efficiency, particularly in memory-intensive workloads. This becomes a greater issue for embedded processors, where speed and energy efficiency have high priority. Therefore, understanding the trade-offs between cache size, associativity, block size, and replacement policies is crucial in the optimization process for processor design.

Furthermore, many simulations of miss rates for various processor configurations use SimpleScalar[3] as their simulator of choice. Original versions of SimpleScalar used its own

instruction set architecture derived from MIPS-IV ISA[3] and may not necessarily model real-world processors accurately. While there has been significant research on porting SimpleScalar to different architectures, such as PowerPC, x86, and ARM[4] (with ARM being verified as cycle-accurate[9]), most research papers on simulating miss rates do not specify the version of SimpleScalar used, the computing environment used, or the instruction set architecture it was ran on.

Therefore, we have developed key questions that our paper hopes to answer:

1) What is the best configuration for embedded processors on the selected benchmarks?
2) Does the 2:1 cache rule of thumb hold for the selected benchmarks?
3) How does gem5 and SimpleScalar differ on simulated cache miss rates?

Our research aims to contribute to the understanding of cache performance and design by using the gem5 simulator to model various single-core CPU cache configurations. To evaluate the effectiveness of gem5 in modeling cache performance compared to SimpleScalar, we base our configurations on configurations used in Ullah et. al.. In addition, we also aim to explore the impact of replacement policies on cache performance. By comparing the results of our study with those of Ullah et. al., we hope to provide a comprehensive picture of cache performance across different simulators and identify any differences in their respective results.

Our research has the potential to inform cache design and optimization, as well as guide future research in this area. By understanding the impact of cache size, associativity, block size, and replacement policies on performance, we can design more effective caches that improve system performance and energy efficiency. Furthermore, by evaluating the effectiveness of simulation tools such as gem5 and SimpleScalar, we can improve the accuracy and efficiency of cache performance modeling.

IV. Research methodology

In this research, we use the gem5 simulator to model various single-core CPU cache configurations and evaluate their performance under different workloads. We describe the experimental methodology below.

Hardware and Software Configuration: We use a Linux-based system for our experiments as that is a base requirement for the gem5 simulator. Additionally the research team purposefully used different types of hardware and operating systems to run the simulations to determine if there were any inconsistencies in the gem5 simulator. Two desktop computers, one operating on Ubuntu 22.04 and the other Debian distro, and a third laptop using WSL were used to run the gem5 simulations. We use gem5 version 20.0.0 for our simulations.

Workloads: We use a variety of benchmark workloads to evaluate the performance of different cache configurations. The specific benchmarks used in this research are Dijkstra, Sha, and Rijndael from the MiBench benchmark suite. The previous paper used two additional workloads, Go and Compress from the SPEC95 suite of benchmarks.

Cache Configurations: We evaluate several cache configurations with different sizes, different block sized, associativity, and replacement policies. As seen in the cache sizes configurations used were 16kB, 32kB, and 64kB. Cache block sizes were configuratble to 32 bytes and 64 bytes. The final configurations that were used for the original research was set associativities of 1, 2, 4, and 8. The additional configuration used in this study beyond what the original tested are following replacement policies, least recently used, random replacement, and first in first out replacement policy.

We use the findings of the previous research paper as a baseline for comparison of each unique combination of configurations.

Table I shows the various configurations tested. The three benchmarks come from the MiBench benchmark suite, designed for embedded processors**990739**. Least Recently Used (LRU), Random, and First-In First-Out (FIFO) are common replacement policies. While there are other replacement policies that may be better,

Benchmarks	**Cache Size**	**Block Size**	**Set Assoc.**	**Replacement Policy**
Dijkstra	16kB	32	1	Least Recently Used
Sha	32kB	64	2	Random
Rijndael	64kB		4	First-In First-Out
			8	

TABLE I
THE DIFFERENT CONFIGURATIONS TESTED.

Simulation Setup: We use gem5's full-system mode to simulate our experiments. In this mode, the entire system, including the CPU, cache, and memory hierarchy, is simulated. We run each workload multiple times to ensure the stability of the results and use a warm-up period to ensure that the cache is populated before we start collecting performance data.

Performance Metrics: We collect several performance metrics, including execution time, instruction throughput, and cache hit rate. We also collect cache miss rate, cache eviction rate, and average memory access latency to provide a more detailed analysis of cache behavior.

In summary, our experimental methodology involves using the gem5 simulator to evaluate various single-core CPU cache configurations under different workloads. We use a range of performance metrics to compare the performance of different configurations and compare our findings with those of other works to determine the significance of our results.

V. Results

In this section the results of each simulation, by benchmark, will be review and thoroughly discussed and analyzed. The target benchmarks are Sha, Dijkstra, and Rijndael.

A. Sha Benchmark

In order to evaluate the performance, we conducted a benchmark test using the SHA (Secure Hash Algorithm).SHA

is a family of cryptographic hash functions that are used to generate a fixed-size message digest from input data of any size. The message digest is designed to be unique and non-reversible, which makes it useful for verifying the integrity and authenticity of digital information.There are currently several variants of the SHA algorithm, including SHA-1, SHA-2 (which includes SHA-224, SHA-256, SHA-384, and SHA-512), and SHA-3.

From a total of 72 simulations run on the SHA benchmark with multiple configurations, different results were obtained on each iteration. The biggest simulated cache miss rate was 0.002082, while the smallest was 0.000007.

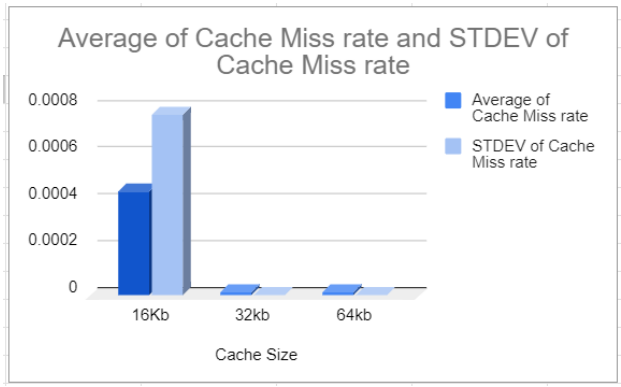

Fig. 1. The bar chart illustrates the cache miss rates for different cache sizes, including 16KB, 32KB, and 64KB. The x-axis represents the cache size, while the y-axis represents the cache miss rate.

The results show that as the cache size increases, the cache miss rate decreases. The cache miss rate for a 16KB cache size was the highest at 0.0004427916667 on average with a standard deviation of 0.0007721100946. However, as the cache size increased to 32KB and 64KB, the cache miss rates decreased significantly to 0.00001034782609 and 0.00001, respectively, with lower standard deviations of 0.000003083809563 and 0.000003064523511. These findings suggest that a larger cache size can significantly improve the performance of the SHA benchmark by reducing the number of cache misses, ultimately leading to faster processing times.

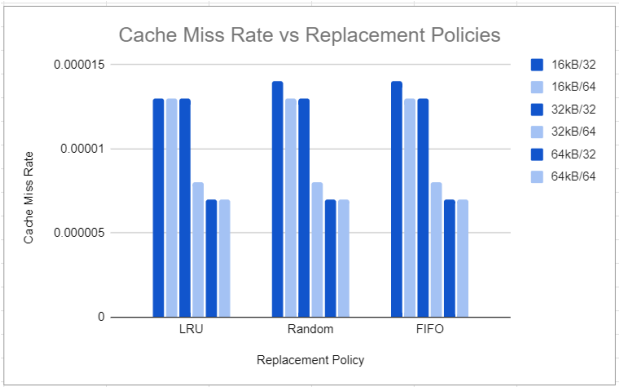

Fig. 2. Cache miss rates with different replacement policies (LRU, Random, and FIFO) with varying the cache sizes and block sizes.

Fig2 shows a bar chart indicating that all three replacement policies exhibit certain similarities in their cache miss rates across different variations of cache size and block size.

We observed that the Random and FIFO replacement policies had similar cache miss rates when using a 16kb cache size and a block size of 32. However, the LRU replacement policy had a slightly lower cache miss rate under these conditions. Additionally, we observed a significant drop in cache miss rate for all replacement policies when increasing the block size from 32 to 64 in a 32kb cache size. We also observed that the cache miss rate was lowest when using a 64kb cache size and a block size of 32 or 64, regardless of the replacement policy used. This finding was consistent across all three replacement policies (LRU, Random, and FIFO).

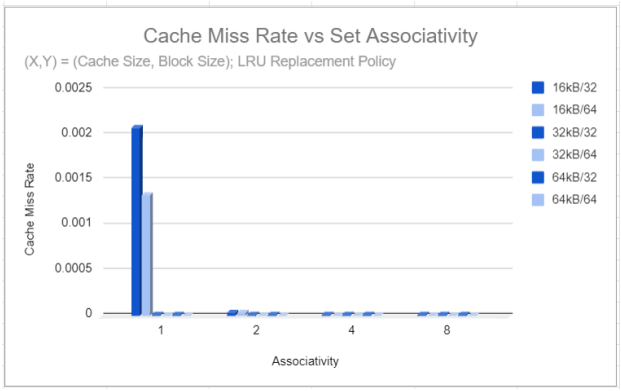

Fig. 3. Cache miss rate grouping with different set associativities (1, 2, 4, and 8) and different cache sizes and block sizes.

We observed that when the set associativity was set to 1, the cache miss rate for a 16kb cache size and block sizes of 32 and 64 was 0.002082 and 0.001342, respectively. However, when the cache size was increased to 32kb and 64kb and the block size was set to 32 or 64, the cache miss rate was significantly lower compared to the previous settings.

Furthermore, we found that with set associativity set as 2, 4, and 8, the cache miss rate was notably low across all configurations of cache size and block size. However, we noticed a slightly higher cache miss rate with a set associativity of 2 and a 16kb cache size compared to other configurations.

We observed that doubling the block size from 32 to 64 resulted in a significant improvement in performance across different cache sizes, set associativity, and replacement policies. On average, the performance improvement was 41.63

For instance, when using a 32kb cache size and a set associativity of 2 with the LRU replacement policy, the cache miss rate decreased from 0.000014 to 0.000008 when the block size was increased from 32 to 64. Similarly, when using a 64kb cache size and a set associativity of 4 with the Random replacement policy, the cache miss rate decreased from 0.000013 to 0.000007 with the same increase in block size. These results demonstrate the significant impact of block size on cache performance and highlight the importance of

carefully choosing appropriate block sizes when designing cache systems.

The 2:1 rule of thumb, which states that the miss rate of a cache stays the same if the cache size is halved and the block size is doubled, has been widely accepted in computer architecture. However, recent studies have shown that this rule may not hold true for certain workloads. Specifically, in the case of the SHA benchmark, our analysis of 32 tests revealed that the miss rate difference was observed in 18 cases after grouping. This suggests that the 2:1 rule of thumb may not be accurate for this particular workload.

B. Dijkstra Benchmark

In the following section contains the results of simulations using various caceh configurations and the Dijkstra benchmark. Each figure show a different aspect of the cache configurations.

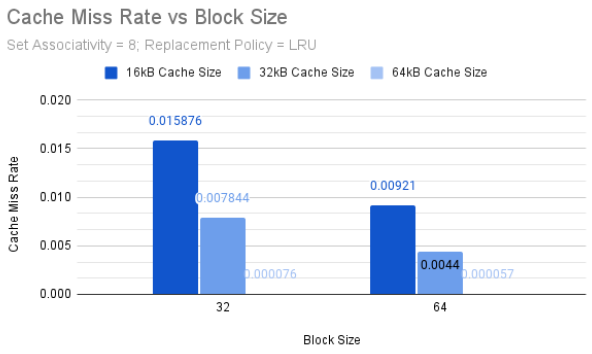

Fig. 4. Miss Rate Comparison of Block Size for Dijkstra Benchmark

The above figure depicts the D-cache miss rate with respect to different overall cache sizes and block size. In addition the set associativity is fixed at 8-way and and the replacement policy is set to Least Recently Used (LRU). These values have been fixed to show only the effects of cache size and block size on D-cache miss rate.

Significant difference are seen in performance from both cache size changes and block size. For a 16kB cache, changing the block size from 32 bytes to 64 bytes, is nearly as effective at reducing miss rate as doubling the cache size and maintaining the same block size. This finding did not true from 32kB cache to 64kB cache, but across each configuration, doubling the block size resulted in nearly a 50% reduction in D-cache miss rate.

While Dijkstra benchmark can be very compute heavy, having to calculate distances for many pairs of nodes, it appears that the 100x100 adjacency matrix was still too small test the effectiveness of cache configurations on a 64kB cache. The minute miss rate shown ing the above figure indicates that a significant majority of the D-cache misses consist of conpulsory cache misses, and once the data is loaded the cache can hold all of the data throughout execution.

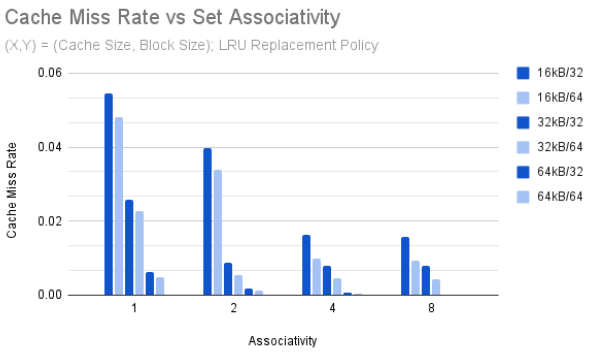

Fig. 5. Miss Rate Comparison of Set Associativity for Dijkstra Benchmark

Here we allow ourselves to vary more parts of the configuration. Having only cache replacement policy as a fixed value set to LRU. By comparing neighboring dark(left) and light(right) columns we can compare the effects of cache block size changes for a given set associativity. Looking at the first two columns on the left, we can see that doubling block size does not have nearly the same effect at low associativity, miss reduction of 16%, as it does with high associativity, reduction of 42%.

Additionally with this graph we can compare miss rates for the 2:1 rule of thumb in cache configurations. This rule states that doubling associativity and halving the cache size should result in equivalent D-cache miss rates. By inspecting column 3 of set associativity 2 groups, we find the following cache configuartion, Cache Size: 32kB, Block Size: 32 bytes and Replacement Policy: LRU. Next observe column 1 of group 4-way set associativy, the cache configuration is as follows, Cache Size: 16kB, Block Size: 32 bytes and Replacement Policy: LRU. By the rule of thumb we should see that these two columns and configurations resulting in equivalent cache miss rate. This is not what is obversed though. In our simulations using the gem5 simulator and Dijkstra benchmark, here specifically, there is no 2:1 rule of thumb present in any pair of applicable configurations.

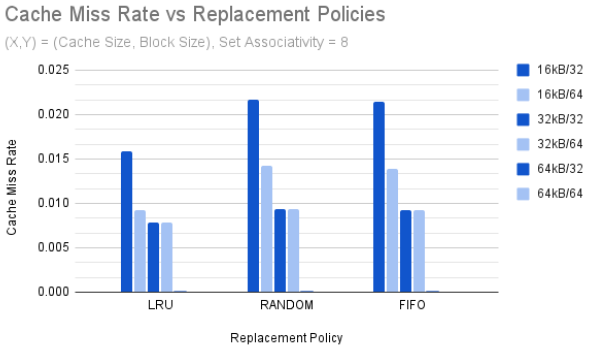

Fig. 6. Miss Rate Comparison of Replacement Policies for Dijkstra Benchmark

Finally we example replacement policy's effects on cache

miss rate. In the figure we have fixed set associativity at 8-way, specifically for the replacement policy to have as many options as possible for choosing a victim block. Here we observe significantly reduced returns as compared to other cache configuration changes. LRU does perform best in almost every case, there were two outlier scenarios of 16kB cache size, block size 32 bytes with 2-way and 4-way associativity having greater performance with Random replacement policy. Overall, First In First Out and Random replacement policies appear equivalent in efficacy.

While LRU policy does perform best in general across the board implementation of a more complex replacement policy, LRU vs FIFO or Random, may not be worth the effort for hardware architects. Particularly in embedded systems where size, power, and cost are off the utmost importance, these findings indicate that replacement policy would be a good starting point for savings across these three metrics.

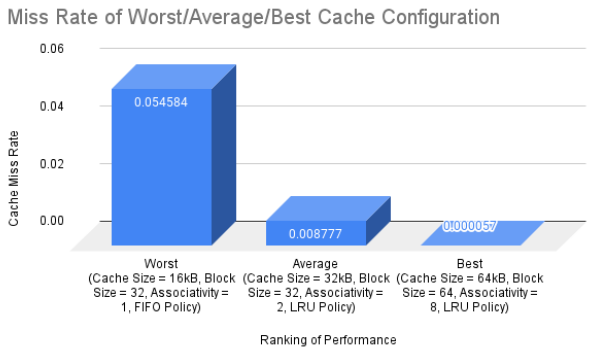

Fig. 7. Miss Rate of Worst, Average, and Best Performing Cache Configurations for Dijkstra Benchmark

In conclusion of Dijkstra benchmark analysis, the above chart plots the miss rate of the worst, average, and best performing cache configurations. For the Dijkstra benchmark, bigger meant better. Bigger Cache, bigger block size and higher associativity were all guarantees of increased performance and significant reductions in miss rate. Many of the previous charts were unable to effectively depict the miss rate of 64kB cache because its miss rates were so minute.

C. Rijndael Benchmark

MiBench provides an implementation of Rijndael. While modern desktop CPUs have instructions for Rijndael[10], they may not be implemented on embedded systems or on systems with small cache configurations. Figure 8 shows the miss rates on configurations with 16kB, 32kB, and 64kB of cache size and 1, 2, 4, and 8 associativity. The configuration with cache size of 16kB and an associativity of 1 shows the highest miss rate at 0.1529. It significantly improves if the cache size or the associativity changes. The miss rates with respect to cache size is monotonically decreasing. That is, a cache size of 16kB with associativity remaining the same will always perform worse than 32kB, and the same with 32kB compared to 64kB. Furthermore, miss rates with respect to associativity also monotonically decreases. The 2:1 cache rule of thumb applies for the configuration of 32kB cache size and 1 associativity compared to the 16kB cache size and 2 associativity, having miss rates 0.0424 and 0.0464, respectively. However, it quickly falls apart with configurations of 64kB cache size and 1 associativity and of 32kB cache size and 2 associativity, having miss rates 0.0353 and 0.0239. The gap continues to increase for larger configurations.

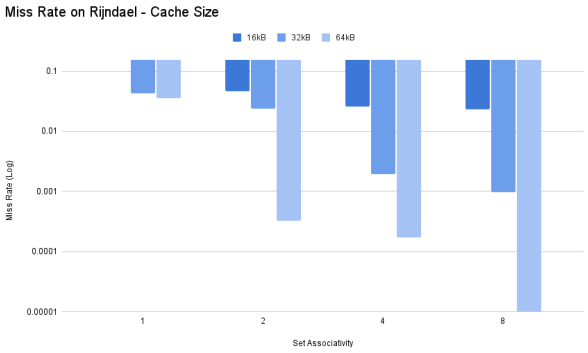

Fig. 8. The miss rate on configurations with different set associativities and cache sizes. The miss rates on the D-cache are on a logarithmic scale to represent the large disparities in the miss rates. Smaller values are better.

Figure 9 shows the miss rates on configurations with block size of either 32 or 64 and an associativity of 1, 2, 4, and 8. Unlike Figure 8, the miss rates are not monotonically decreasing as the cache size or associativity increases. For an associativity of 1, the miss rates for a configuration with a block size of 32 slightly loses over a configuration of a block size of 64 (miss rates 0.0780 and 0.0758, respectively). This pattern does not hold for larger associativities. However, with the exception of an associativity of 1, the larger the associativity, the smaller the difference between the block sizes.

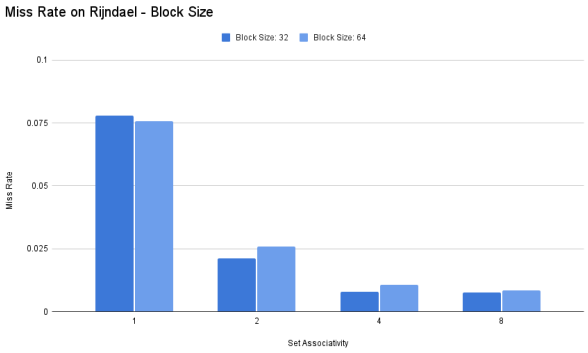

Fig. 9. The D-cache miss rate on configurations with different set associativities and block sizes. Smaller values are better.

This pattern is explored more in Figure 10. Looking solely at the cache size, increasing block size on larger cache size

often lowers miss rate. If we break down by associativity, an associativity of 1 always benefits from a larger block size, an associativity of 4 and 8 improve with larger cache sizes. The exception is a configuration with associativity of 2, as that will always benefit from a block size of 32.

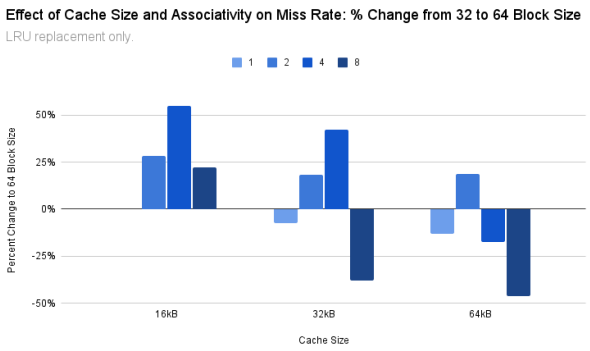

Fig. 10. The percent change of D-cache miss rate from configurations with 32 vs 64 block size. Only LRU replacement policy is shown, but the pattern remains the same with other replacement policies (although their extent differs). Smaller values are better.

Expanding more on the 2:1 cache rule of thumb, Figure 11 shows the percent difference of "cache rule of thumb" configurations. That is, the percent difference between a configuration with n cache size and x associativity and a configuration with $n/2$ cache size and $2x$ associativity. A value of zero indicates that the two values are the same. If many runs exhibit a percent difference close to zero, then the cache rule of thumb would hold for Rijndael benchmarks. However, 19 out of 35 runs exhibit a percent difference of greater than 50%, indicating a significant deviation from the supposed 2:1 cache rule of thumb.

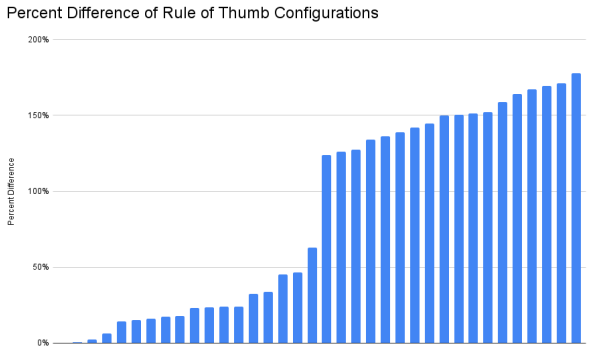

Fig. 11. The percent difference of D-cache miss rates of cache rule of thumb configurations. Values closer to 0 are better.

VI. Discussion

VII. Future Work

The use of gem5 to model various single-core CPU cache configurations provides a powerful tool for evaluating cache performance. Our research has shown that different cache configurations can have a significant impact on performance, and that using simulation tools such as gem5 can provide detailed insights into cache behavior.

There are several directions for future research in this area:

Multi-core CPU Cache Configurations: In this study, we focus on single-core CPU cache configurations. However, multi-core CPUs are becoming increasingly common, and cache design for multi-core CPUs is a complex and challenging area of research. Future studies could use gem5 to model various multi-core CPU cache configurations and evaluate their performance under different workloads.

Machine Learning Techniques: Recent research has explored the use of machine learning techniques to optimize cache performance. These techniques involve using machine learning algorithms to predict cache behavior and optimize cache configurations. Using gem5 to evaluate these techniques can help identify their strengths and weaknesses and guide future research in this area.

Emerging Memory Technologies: Emerging memory technologies, such as phase-change memory (PCM) and resistive random-access memory (RRAM), have the potential to revolutionize cache design. These technologies have different performance characteristics than traditional memory technologies, and designing caches that can take advantage of their unique properties is an area of active research. Future studies could use gem5 to evaluate cache designs that incorporate emerging memory technologies.

Energy-Efficient Caches: Energy efficiency is becoming an increasingly important consideration in cache design. Caches are power-hungry components of the memory hierarchy, and designing energy-efficient caches is essential to reduce power consumption and improve battery life. Future studies could use gem5 to evaluate energy-efficient cache designs, such as cache compression and cache bypassing techniques.

In conclusion, the use of gem5 to model various single-core CPU cache configurations provides a powerful tool for evaluating cache performance. Future research in this area could explore multi-core CPU cache configurations, machine learning techniques, emerging memory technologies, and energy-efficient caches. These studies could provide valuable insights into cache design and help improve the performance and energy efficiency of modern computer systems.

VIII. Conclusion

In this research, we have used the gem5 simulator to model various single-core CPU cache configurations and evaluated their performance using several benchmarks. We compared our results with those of a previous study that used the SimpleScalar simulator to evaluate cache performance. Our research aimed to contribute to the understanding of cache performance and design, as well as evaluate the effectiveness of different simulators in modeling cache behavior.

Our study found that the performance of different cache configurations varied significantly depending on the workload characteristics. We observed that the impact of replacement policies on cache performance was insignificant, with the LRU

policy outperforming the FIFO and Random policies in most cases.

Our results differed significantly from those of the previous study, which focused on the impact of cache size and associativity on performance using the SimpleScalar simulator. We found that the performance of cache configurations varied depending on the simulator. Our findings suggest that the gem5 simulator may be more effective in modeling cache behavior of modern CPU's compared to SimpleScalar.

Our research has several implications for cache design and optimization. By understanding the impact of different cache configurations on performance, designers can improve system performance and energy efficiency. Our findings also have implications for the use of simulation tools in cache performance modeling, highlighting the need for careful selection of simulator and benchmark suite.

In conclusion, our research has contributed to the understanding of cache performance and design using the gem5 simulator. Our results highlight the significant differences in cache performance between different simulators and benchmark suites. We hope that our findings will guide future research in cache design and optimization and improve the accuracy and efficiency of cache performance modeling.

References

[1] J. L. Hennessy, D. A. Patterson, and K. Asanović, *Computer architecture: a quantitative approach*, 5th ed. Waltham, MA: Morgan Kaufmann/Elsevier, 2012, OCLC: ocn755102367, ISBN: 978-0-12-383872-8.

[2] N. Binkert, B. Beckmann, G. Black, *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011, ISSN: 0163-5964. DOI: 10.1145/2024716.2024718. [Online]. Available: https://doi.org/10.1145/2024716.2024718.

[3] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," *SIGARCH Comput. Archit. News*, vol. 25, no. 3, pp. 13–25, Jun. 1997, ISSN: 0163-5964. DOI: 10.1145/268806.268810. [Online]. Available: https://doi.org/10.1145/268806.268810.

[4] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic, "Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite," in *Proceedings of the 42nd Annual Southeast Regional Conference*, ser. ACM-SE 42, Huntsville, Alabama: Association for Computing Machinery, 2004, pp. 267–272, ISBN: 1581138709. DOI: 10.1145/986537.986601. [Online]. Available: https://doi.org/10.1145/986537.986601.

[5] P. Panda, G. Patil, and B. Raveendran, "A survey on replacement strategies in cache memory for embedded systems," in *2016 IEEE Distributed Computing, VLSI, Electrical Circuits and Robotics (DISCOVER)*, 2016, pp. 12–17. DOI: 10.1109/DISCOVER.2016.7806218.

[6] H.-f. Ma, N.-m. Yao, and H.-b. Fan, "Cache performance simulation and analysis under simplescalar platform," in *2009 International Conference on New Trends in Information and Service Science*, 2009, pp. 607–612. DOI: 10.1109/NISS.2009.61.

[7] Z. Ullah, N. Minallah, S. N. K. Marwat, A. Hafeez, and T. Fouzder, "Performance analysis of cache size and set-associativity using simplescalar benchmark," in *2019 5th International Conference on Advances in Electrical Engineering (ICAEE)*, 2019, pp. 440–447. DOI: 10.1109/ICAEE48663.2019.8975563.

[8] R. M. Reas, A. B. Alvarez, and J. A. P. Reyes, "Simulation of standard benchmarks in hardware implementations of l2 cache models in verilog hdl," in *Proceedings of the 2010 12th International Conference on Computer Modelling and Simulation*, ser. UKSIM '10, USA: IEEE Computer Society, 2010, pp. 153–158, ISBN: 9780769540160. DOI: 10.1109/UKSIM.2010.35. [Online]. Available: https://doi.org/10.1109/UKSIM.2010.35.

[9] S. W. Chung, G. H. Park, H.-J. Suh, *et al.*, "Sim-arm1136: A case study on the accuracy of the cycle-accurate simulator," *Microprocessors and Microsystems*, vol. 30, no. 3, pp. 137–144, 2006, ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2005.07.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141933105000633.

[10] S. Gueron, "Intel advanced encryption standard (aes) instructions set," *Intel White Paper, Rev*, vol. 3, pp. 1–94, 2010.

APPENDIX

TABLE II
DIJKSTRA BENCHMARK

Cache Size	Block Size	Set Assoc.	Repl. Policy	Inst. Count	IC hit	IC Missed	IC MissRate	DC hit	DC Missed	DC Miss Rate
16kB	32	1	1	225882042	319363400	63949	0.0002	53844950	3108738	0.054584
16kB	32	1	2	225882042	319363400	63949	0.0002	53844950	3108738	0.054584
16kB	32	1	3	225882042	319363400	63949	0.0002	53844950	3108738	0.054584
16kB	32	2	1	225882042	319423331	4018	0.000013	54694862	2258826	0.039661
16kB	32	2	2	225882042	319422902	4447	0.000014	54905813	2047875	0.035957
16kB	32	2	3	225882042	319423032	4317	0.000014	54607742	2345946	0.04119
16kB	32	4	1	225882042	319424560	2789	0.000009	56021761	931927	0.016363
16kB	32	4	2	225882042	319424627	2722	0.000009	55728462	1225226	0.021513
16kB	32	4	3	225882042	319424033	3316	0.00001	55746866	1206822	0.02119
16kB	32	8	1	225882042	319426027	1322	0.000004	56049483	904205	0.015876
16kB	32	8	2	225882042	319425830	1519	0.000005	55715846	1237842	0.021734
16kB	32	8	3	225882042	319425977	1372	0.000004	55731617	1222071	0.021457
16kB	64	1	1	225882042	319392138	35211	0.00011	54210849	2740582	0.048121
16kB	64	1	2	225882042	319392138	35211	0.00011	54210849	2740582	0.048121
16kB	64	1	3	225882042	319392138	35211	0.00011	54210849	2740582	0.048121
16kB	64	2	1	225882042	319424577	2772	0.000009	55027664	1923767	0.033779
16kB	64	2	2	225882042	319424252	3097	0.00001	55123643	1827788	0.032094
16kB	64	2	3	225882042	319424290	3059	0.00001	54898513	2052918	0.036047
16kB	64	4	1	225882042	319425030	2319	0.000007	56390810	560621	0.009844
16kB	64	4	2	225882042	319425019	2330	0.000007	56156876	794555	0.013951
16kB	64	4	3	225882042	319424507	2842	0.000009	56181918	769513	0.013512
16kB	64	8	1	225882042	319426555	794	0.000002	56426880	524551	0.00921
16kB	64	8	2	225882042	319426390	959	0.000003	56139222	812209	0.014261
16kB	64	8	3	225882042	319426520	829	0.000003	56161253	790178	0.013875
32kB	32	1	1	225882042	319423518	3831	0.000012	55489194	1464494	0.025714
32kB	32	1	2	225882042	319423518	3831	0.000012	55489194	1464494	0.025714
32kB	32	1	3	225882042	319423518	3831	0.000012	55489194	1464494	0.025714
32kB	32	2	1	225882042	319425809	1540	0.000005	56453802	499886	0.008777
32kB	32	2	2	225882042	319425775	1574	0.000005	56369181	584507	0.010263
32kB	32	2	3	225882042	319425820	1529	0.000005	56368002	585686	0.010284
32kB	32	4	1	225882042	319426035	1314	0.000004	56502425	451263	0.007923
32kB	32	4	2	225882042	319425985	1364	0.000004	56402575	551113	0.009677
32kB	32	4	3	225882042	319426025	1324	0.000004	56407995	545693	0.009581
32kB	32	8	1	225882042	319426035	1314	0.000004	56506962	446726	0.007844
32kB	32	8	2	225882042	319425972	1377	0.000004	56422545	531143	0.009326
32kB	32	8	3	225882042	319426026	1323	0.000004	56425358	528330	0.009276
32kB	64	1	1	225882042	319424696	2653	0.000008	55663535	1287896	0.022614
32kB	64	1	2	225882042	319424696	2653	0.000008	55663535	1287896	0.022614
32kB	64	1	3	225882042	319424696	2653	0.000008	55663535	1287896	0.022614
32kB	64	2	1	225882042	319426453	896	0.000003	56644758	306673	0.005385
32kB	64	2	2	225882042	319426421	928	0.000003	56580462	370969	0.006514
32kB	64	2	3	225882042	319426470	879	0.000003	56585160	366271	0.006431
32kB	64	4	1	225882042	319426573	776	0.000002	56694001	257430	0.00452
32kB	64	4	2	225882042	319426521	828	0.000003	56614011	337420	0.005925
32kB	64	4	3	225882042	319426561	788	0.000002	56619458	331973	0.005829
32kB	32	8	1	225882042	319426035	1314	0.000004	56506962	446726	0.007844
32kB	32	8	2	225882042	319425972	1377	0.000004	56422545	531143	0.009326
32kB	32	8	3	225882042	319426026	1323	0.000004	56425358	528330	0.009276
64kB	32	1	1	225882042	319425599	1750	0.000005	56589626	364062	0.006392
64kB	32	1	2	225882042	319425599	1750	0.000005	56589626	364062	0.006392
64kB	32	1	3	225882042	319425599	1750	0.000005	56589626	364062	0.006392
64kB	32	2	1	225882042	319425821	1528	0.000005	56844976	108712	0.001909
64kB	32	2	2	225882042	319425849	1500	0.000005	56812936	140752	0.002471
64kB	32	2	3	225882042	319425848	1501	0.000005	56817912	135776	0.002384
64kB	32	4	1	225882042	319426036	1313	0.000004	56918063	35625	0.000626
64kB	32	4	2	225882042	319426034	1315	0.000004	56905687	48001	0.000843
64kB	32	4	3	225882042	319426035	1314	0.000004	56903637	50051	0.000879
64kB	32	8	1	225882042	319426036	1313	0.000004	56949333	4355	0.000076
64kB	32	8	2	225882042	319426035	1314	0.000004	56947976	5712	0.0001
64kB	32	8	3	225882042	319426036	1313	0.000004	56947966	5722	0.0001
64kB	64	1	1	225882042	319426340	1009	0.000003	56679489	271942	0.004775
64kB	64	1	2	225882042	319426340	1009	0.000003	56679489	271942	0.004775
64kB	64	1	3	225882042	319426340	1009	0.000003	56679489	271942	0.004775
64kB	64	2	1	225882042	319426463	886	0.000003	56882379	69052	0.001212
64kB	64	2	2	225882042	319426480	869	0.000003	56858757	92674	0.001627
64kB	64	2	3	225882042	319426492	857	0.000003	56863394	88037	0.001546
64kB	64	4	1	225882042	319426574	775	0.000002	56929917	21514	0.000378
64kB	64	4	2	225882042	319426570	779	0.000002	56920955	30476	0.000535
64kB	64	4	3	225882042	319426573	776	0.000002	56919509	31922	0.000561
64kB	64	8	1	225882042	319426574	775	0.000002	56948181	3250	0.000057
64kB	64	8	2	225882042	319426572	777	0.000002	56947025	4406	0.000077
64kB	64	8	3	225882042	319426574	775	0.000002	56946405	5026	0.000088