Analyzing the performance of different cache replacement policies using gem5 simulator

Debopriya Roy Dipta, Shubham Makwana, and Lohith Reddy Kalluru

ABSTRACT: For memory-intensive programs, a good cache replacement policy can exploit high performance. It is hard to decide on a particular algorithm that is highly efficient for a specific cache configuration, as trade-offs exist on every part. In this project, we carried out a detailed comparative analysis among some of the common and efficient cache replacement policies with an aim to study their individual effectiveness for different cache configurations, which will be performed by using the gem5 simulator. The evaluation is carried out by using benchmarks with varying workloads to determine the performance metrics individually. In this current study, we evaluated the performance of Random Replacement (RR), Least Recently Used (LRU), and First in First Out (FIFO) replacement policies with benchmark Nqueens and BFS. In addition, this project also intends to find out the impacts of the above-mentioned replacement policies on time-driven cache side-channel attacks' performance in terms of success rate. However, we have not been able to run such attack for different gem5 instances. So far, we have been able to write the source code to produce AES side-channel timing attack and have been able to verify the success rate of such attack on the local PC, not on the gem5 simulation environment. However, as we know that in the full-system mode, GEM5 runs a real operating system on it and allows users to interact with the OS. Hence, users can run any applications on GEM5 as running on real-world hardware and we can exploit this feature to launch the side-channel attack for different gem5 simulation instances to understand the impacts of different replacement policies on the success rate. Due to

limited time, we have not been able to try this out, but we are going to consider it as our future scope for this project.

INTRODUCTION

The cache memory aims to keep the frequently used memory blocks close to the processor. However, the program locality may demand more capacity that goes beyond the cache size while executing big data applications. In such scenarios, an efficient replacement algorithm can play a crucial role by preserving the frequently acceded blocks in the cache while evicting the deadlock immediately. Here, deadlock signifies the memory block that is very unlikely to be requested in the future. However, it is generally impossible to figure out or predict the necessity of specific information far ahead in the future. Due to this, the predictions of the current replacement policies mostly depend on the behavior of the cache. A lot of research is still ongoing on memory hierarchy; however, due to the increasing performance gap between memory and processors, it is important to revisit the efficacy of the current replacement policies.

The Time driven cache side-channel can be established by exploiting the time difference between cache hits and misses to leak important information. It can be assumed that different cache replacement algorithms may have individual impacts on the success rate of such time-driven side-channel attacks. By utilizing the gem5 simulator, we are interested to learn whether individual assessments can be performed in terms of their impact on such attacks' success. This assessment can be considered as an important factor while choosing a specific cache replacement algorithm.

RELATED WORK

In ref. [1], the authors provided a narrative on different replacement algorithms and presented a concise argument regarding the necessity of improving their efficiency. They also offered a comparative analysis among different existing replacement algorithms; however, they solely focused on narrative while providing their reasonings instead of any technical or simulation-based demonstrations. Some of the researches are focused on improving the individual algorithms by making changes; however, such improvements are achieved with a certain cost as well [2]. Therefore, trade-offs are also available in such cases. In this field, Belady's optimal algorithm is considered as the benchmark for all the replacement algorithms, which is impractical while considering implementations [3]. However, this algorithm is often used to evaluate the performance of other replacement algorithms. If it is considered in this sense, then the current replacement algorithms are still far away from reaching that benchmark.

In ref. [4], the authors presented a study to explain how the cache configurations impact the success rates of time-driven cache attacks. In their study, they considered several parameters, including replacement algorithms as well. In our project, we would like to build our own model using the gem5 simulator to reproduce a performance metric using different benchmarks of varying loads to realize the performance on an identical platform.

PROPOSED IDEA

We are interested in evaluating the performance of Random Replacement (RR), Least Recently Used (LRU), and First in First Out (FIFO) replacement policies for

different cache configurations. We are going to consider both two-level and three-level cache hierarchy along with different configurable parameters, such as cache size, associativity, line size, etc., for all level caches. The cache replacement algorithms have a strong bias to the nature of the program and its performance. In this project, we are going to observe the performance while trying different cache replacement policies at different levels instead of having the same replacement algorithms at all levels. The performance in terms of miss rates while utilizing benchmarks with varying loads will be performed. Another interesting part of this project is to evaluate the impact of individual algorithms on the success rate of time-driven side-channel attacks. This feature will also be considered as a performance metric in this study. For this entire study, the gem5 simulator will be utilized to produce the final performance metric.

METHODOLOGY

A. CONFIGURATION FOR TWO LEVEL HIERARCHY

In our first trial, we performed our analysis for two-level cache hierarchy. In Table 1, the configurations that we have chosen are listed which were simulated in gem5 for LRU, FIFO, and Random RP individually. We have customized the "src/mem/cache/Cache.py" file for configuring different replacement policies during the build. In Fig. 1, the system architecture for the two-level cache hierarchy is shown. For the ROI, we have utilized Nqueen benchmark only for the study of two-level cache hierarchy. For the later part, on three-level cache hierarchy, we are going to explore BFS benchmark as well. The output directories for all the simulated configurations are attached in the zip file.

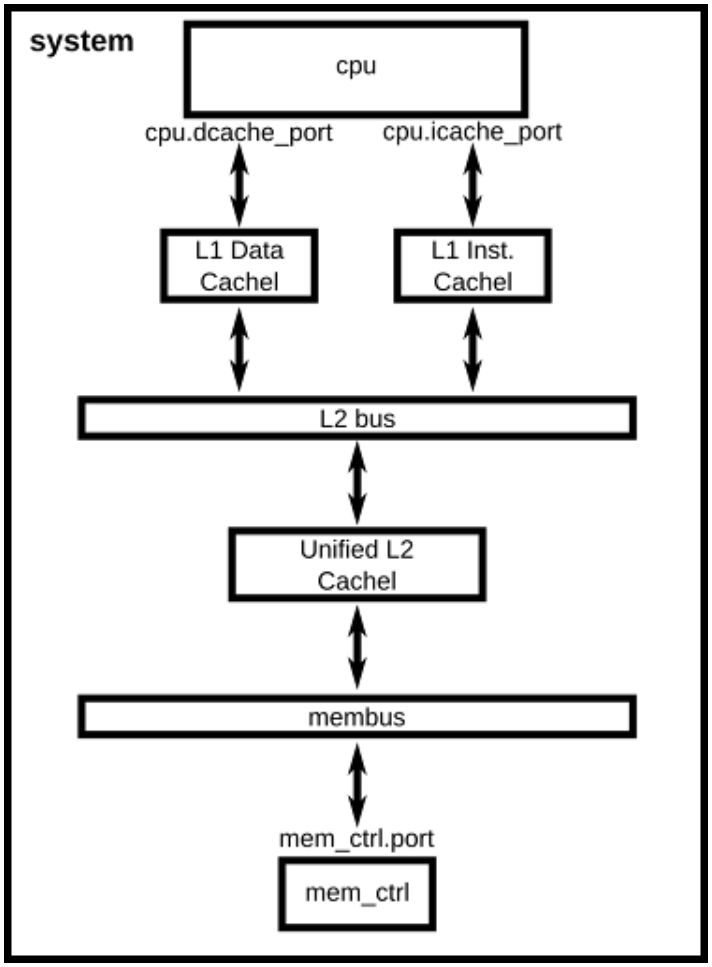


Fig. 1: Two-level cache hierarchy implemented in the GEM5 [5]

Here, we are interested to consider the L1 cache miss rate for individual cache configurations to realize the impacts of different RP. The formula that we have used to compute the L1 cache miss rate is presented below:

L1 Overall Miss Rates = (total misses in dcache + total misses in icache) / (total accesses in dcache + total accesses in icache)

TABLE 1: Different configurations for the two-level cache hierarchy to realize the impacts of individual parameters.

Configuration # 1	L1 icache (KB)	L1 dcache (KB)	L2 cache (MB)	L1 dcache associativity	L1 icache associativity	L2 cache associativity	Cache line size
1	128	128	4	2	2	1	64
2	256	256	4	2	2	1	64
3	512	512	4	2	2	1	64
4	128	128	8	2	2	1	64
5	128	128	16	2	2	1	64
6	128	128	4	4	4	1	64
7	128	128	4	4	4	4	64
8	128	128	4	8	8	8	64
9	128	128	4	2	2	1	128

B. CONFIGURATION FOR THREE LEVEL HIERARCHY

For three level hierarchy we had to change the following files:

Path: configs/common/Caches.py, → Created another copied version l3Caches.py

Added part:

```
class L3Cache(BaseCache):
    assoc = 16
    block_size = 64
    hit_latency = 20
    response_latency = 20
    mshrs = 512
    tgts_per_mshr = 20
    write_buffers = 256
```

Path: configs/common/CacheConfig.py → Created another copied version of l3CacheConfig.py

---

```
def config_cache( Options , System ):
   if Options . cpu_type == "arm_detailed":
      try:
         from O3_ARM_v7a import *
      except:
         print "Did you compile the O3 model?"
         SYS . Exit (1)
```

```
        dcache_class , icache_class , l2_cache_class , l3_cache_class =/
            O3_ARM_v7a_DCache , O3_ARM_v7a_ICache , O3_ARM_v7aL2 ,
O3_ARM_v7aL3
      else:
        dcache_class , icache_class , l2_cache_class , l3_cache_class =/
            L1Cache , L1Cache , L2Cache , L3Cache

    # Set the cache line size of the system
    system . Cache_line_size = options . Cacheline_size

    # set the shared l3 cache
    if options . l3cache :
         system . L3 = l3_cache_class ( clk_domain = system . cpu_clk_domain,
                  size = Options . l3_size ,Assoc = Options . l3_assoc )
         system . tol3bus = L3XBar ( clk_domain, system . cpu_clk_domain ,
                  width = 32)
        system . l3 . cpu_side = system . tol3bus . master
        system . l3 . mem_side = system . membus . slave

for I in xrange( Options . num_cpus ):
        if Options . Caches :
            ICache = icache_class ( size = Options . l1i_size ,
                              Assoc = Options . l1i_assoc )
            DCache = dcache_class ( size = Options . l1d_size ,
                              Assoc = Options .l1d_assoc )

            if buildEnv ['TARGET_ISA'] == 'x86':
```

```
            System . CPU [ I ]. addPrivateSplitL1Caches ( ICache , DCache ,
                    PageTableWalkerCache (),PageTableWalkerCache ())
        else:
            System . CPU [ I ]. addPrivateSplitL1Caches ( ICache , DCache )
    system . cpu [ i ]. createInterruptController ()


    #Configure secondary private cache if options . L2cache :
        system.Cpu [ i ]. L2 = l2_cache_class ( clk_domain = system . Cpu_clk_domain ,
                                size = options . L2_size ,
                                assoc = options . L2_assoc )

        system . cpu [ i ]. tol2bus = CoherentBus ()
        system . cpu [ i ]. l2 . cpu_side = system . cpu [ i ]. tol2bus . master
        system . cpu [ i ]. l2 . mem_side = system . tol3bus . slave

    if options . l3cache :

      system . cpu [ i ]. connectAllPorts ( system . cpu [ i ]. tol2bus , system . membus )

    else:
        if options . l2cache :
            system . cpu [ i ]. connectAllPorts ( system . tol2bus , system . membus )
        else:
            system . cpu [ i ]. connectAllPorts ( system . membus )
```

```
    return system
```

Path: configs/example/se.py → Created another copied version l3se.py

---

Replacing Commit.Caches to Commit.l3Caches

Replacing Commit.CacheConfig to l3CacheConfig

Path: /src/mem/XBar.py → Created another copied version L3XBar.py

---

```
class L3XBar(CoherentXBar):
    # 256-bit crossbar by default
    width = 32

    # Assume that most of this is covered by the cache latencies, with
    # no more than a single pipeline stage for any packet.
    frontend_latency = 1
    forward_latency = 0
    response_latency = 1
    snoop_response_latency = 1

    # Use a snoop-filter by default, and set the latency to zero as
    # the lookup is assumed to overlap with the frontend latency of
```

```
    # the crossbar
    snoop_filter = SnoopFilter(lookup_latency = 0)

    # This specialisation of the coherent crossbar is to be considered
    # the point of unification, it connects the dcache and the icache
    # to the first level of unified cache.
    point_of_unification = True
```

Path: /src/cpu/BaseCPU.py

```
from XBar import L3XBar
# Add three-level cache architecture configuration
def addThreeLevelCacheHierarchy(self, ic, dc, l3c, iwc=None, dwc=None,
                                xbar=None):
        self.addPrivateSplitL1Caches(ic, dc, iwc, dwc)
        self.toL3Bus = xbar if xbar else L3XBar()
        self.connectCachedPorts(self.toL3Bus)
        self.l3cache = l3c
        self.toL3Bus.master = self.l3cache.cpu_side
        self._cached_ports = ['l3cache.mem_side']
```

Path: configs/common/Options.py

---

parser.add_argument("--l3cache", action="store_true")

TABLE 2: Different configurations for the two-level cache hierarchy to realize the impacts of individual parameters.

Configuration # 1	L1 icache (KB)	L1 dcache (KB)	L2 cache (KB)	L3 cache (MB)	L1 icache & dcache associativity	L2 cache associativity	L3 cache associativity	Cache line size
1	128	128	4	8	2	2	2	64
2	256	256	4	8	2	2	2	64
3	512	512	4	8	2	2	2	64
4	128	128	4	8	4	4	4	64
5	128	128	4	8	8	8	8	64
6	128	128	4	8	2	4	8	64
8	128	128	4	8	8	4	2	64

A sample version of compiling with benchmark using the bash script

```
# -- an example to run SPEC 429.mcf on gem5, put it under 429.mcf folder --
export GEM5_DIR=/home/vm-user/gem5
export BENCHMARK=./qn
time $GEM5_DIR/build/X86/gem5.opt -d ~/Project/o3/FIFO/8 $GEM5_DIR/configs/example/se.py --cpu-type=O3CPU --caches --l2cache --l1d_size=128kB --l1i_size=128kB --l2_size=4MB --l1d_assoc=8 --l1i_assoc=8 --l2_assoc=8 --cacheline_size=64 --cmd=$BENCHMARK --options=11

#time $GEM5_DIR/build/X86/gem5.opt -d ~/work/gem5_output/nqueen/ $GEM5_DIR/configs/example/se.py -c --cmd=$BENCHMARK --options=11  -I 100000000 --cpu-type=MinorCPU --caches --l2cache --l1d_size=128kB --l1i_size=128kB --l2_size=1MB --l1d_assoc=2 --l1i_assoc=2 --l2_assoc=1 --cacheline_size=64
```

C. CACHE TIMING ATTACK on AES (T-TABLE)

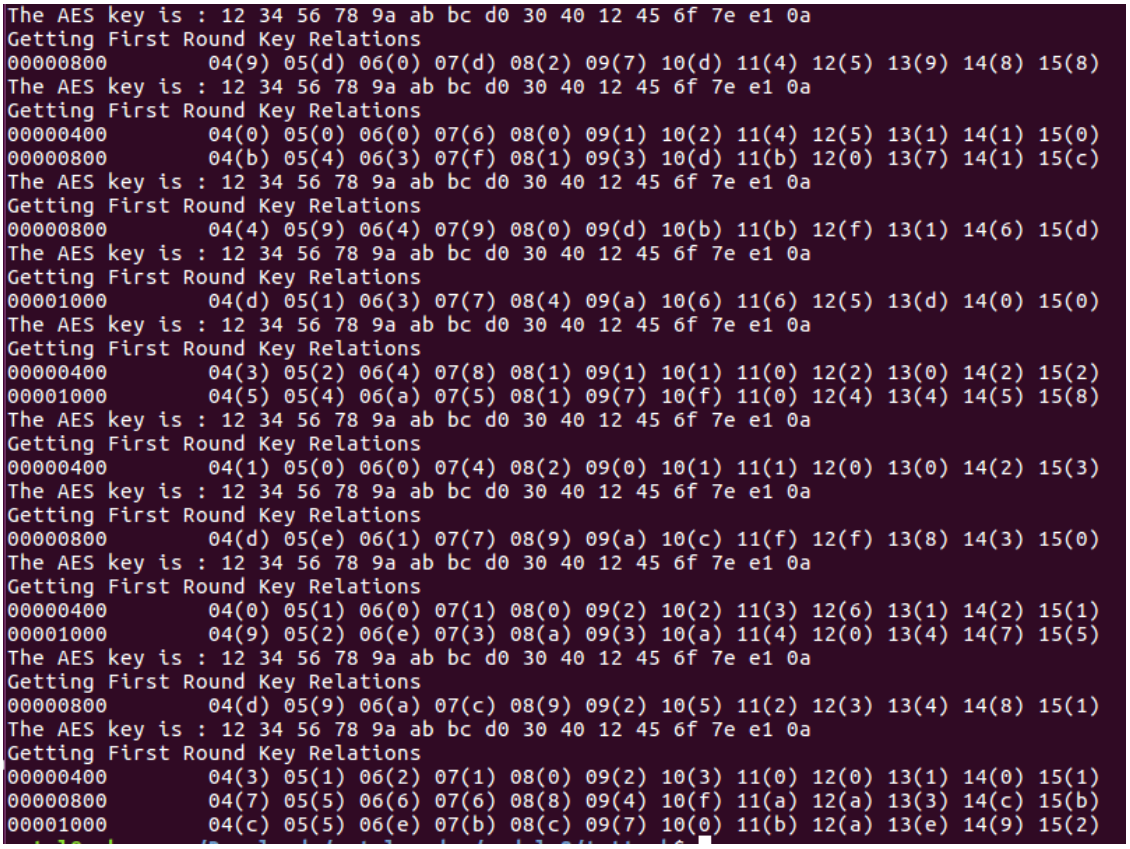
Here we are considering a variable index lookup T_i [k[i] ⊕ n [i]] at the starting of the AES computation. The time needed for the AES computation is related to the time required for the array lookup. The information leaked due to the AES timings indicates about k[i] ⊕ n[i] and this is being used to deduce the k[i] value as a function of n[i]. The attacker monitors the time required by the victim to handle n's. Then, adds the time required by AES for each n[i]and observes the overall max time for the AES. Also, the attacker performs the experiments based on the keys k which are already known with the same CPU and software. The attacker then observes the overall AES time to predict the victim's secret key.

Input Key: 12 34 56 78 9a ab bc d0 30 40 12 45 6f 7e e1 0a

Run time	Expected output 9 a b d 3 4 1 4 6 7 e 0	Successful Nibbles
1	9 d 0 d 2 7 d 4 5 9 8 8	3
2	b 4 b f 3 3 d b 6 7 1 c	4
3	4 9 4 d 0 d b 4 f 7 e d	4
4	d a 3 7 4 4 6 6 6 d e 0	5
5	9 a a d 1 7 1 0 4 7 5 8	5
6	1 a b d 2 0 1 1 0 0 2 3	4
7	9 e 1 7 3 4 c f f 8 3 0	4
8	9 a e 3 3 3 a 4 0 7 e 5	5
9	b 9 a d 9 2 1 2 6 4 e 1	4
10	c 5 b b 3 7 1 b a e e 2	4
	Total = 12 x 10 = 120	= 42

Accuracy = (42/120)x100% = 35%

Sample Output:



RESULTS

A sample stat.txt Output:

```
qn24b base version 1.0.1 2004-09-02
============================================
qn24b base version 1.0.1 2004-09-02
problem size n        : 11
total   solutions     : 2680
correct solutions     : 2680
million solutions/sec : 1.912
elapsed time (sec)    : 0.001
============================================
Exiting @ tick 1464837000 because exiting with last active thread context

real    0m18.173s
user    0m18.108s
sys     0m0.064s
```

Replacement Policy: FIFO

system.cpu.dcache.overallMisses::total (Count)	452	# number of overall misses
system.cpu.icache.overallMisses::total (Count)	1040	# number of overall misses
system.cpu.icache.overallAccesses::total (read+write) accesses (Count)	660769	# number of overall
system.cpu.dcache.overallAccesses::total (read+write) accesses (Count)	713013	# number of overall
system.cpu.dcache.overallMissRate::total accesses (Ratio)	0.000634	# miss rate for overall
system.cpu.icache.overallMissRate::total accesses (Ratio)	0.001574	# miss rate for overall

We are expecting to produce results in terms of graphs to relay the information in a more comprehensive way. Some of the important graphs that we have produced are shown below, although we did not get enough time to produce more comparative analysis based on the stat.txt output files.

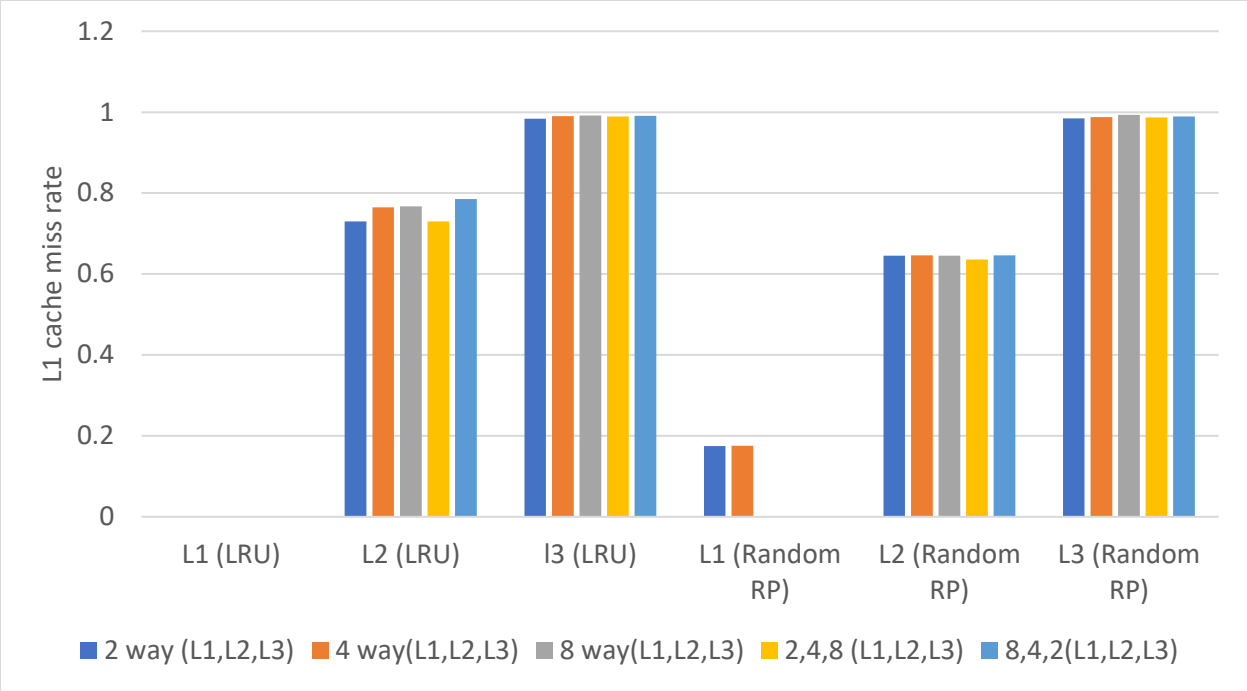


Fig. 2: Comparison of L1 cache miss rate with LRU and Random RP. Although we have the stat file for FIFO as well, we did not get the chance to calculate and put them in the analysis due to time constraint.

From Fig. 2, it seems that the associativity does not impact much for lower-level cache, L3. However, the variation can be much visible for higher-level cache like L1, especially for Random RP. As the workload of the benchmark is comparatively smaller, probably that's why such characteristic is visible.

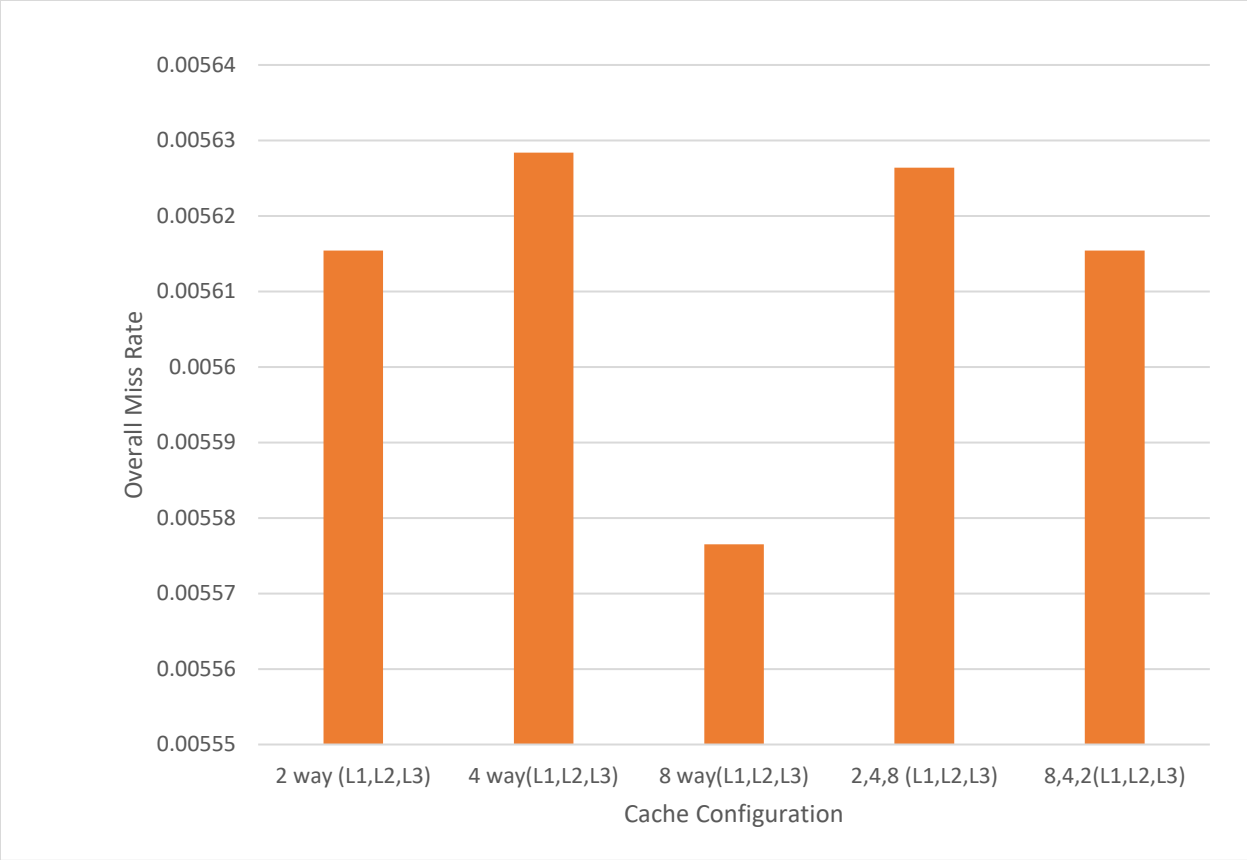


Fig. 3: Comparison of overall miss rate with cache configurations having varying associativity.

The result in Fig. 3 illustrates that, the higher associativity in all the cache levels may lead to the lowest overall miss rate. Increasing associativity in higher level cache or lower-level cache alone does not bring much effectiveness in terms of overall miss rate.

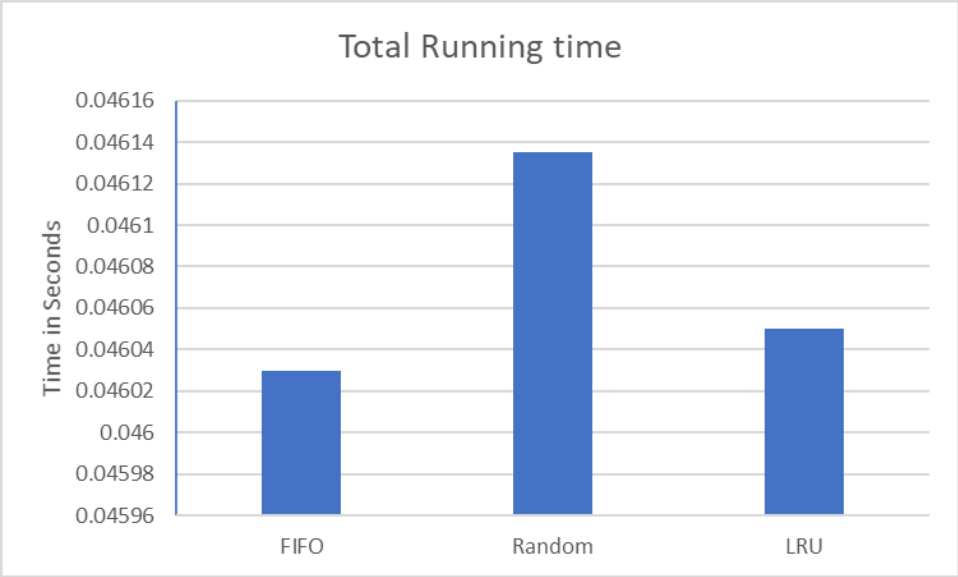


Fig. 4. Total running time with different RP while the other parameters remain constant.

According to Fig. 4, it can be seen that the total running time of the simulation while using benchmark takes comparatively more time while Random RP is utilized. We cannot come up with an explanation for such characteristic, but we are going to take deep look on this matter later.

FUTURE SCOPE

We have not been able to accomplish everything that we planned earlier. We would like to carry out the rest of the tasks in future. In below, we have listed these tasks as the future scope of this study:

- In the full-system mode, GEM5 runs a real operating system on it and allows users to interact with the OS. Hence, users can run any applications on GEM5 as running on real-world hardware. Therefore, we would like to launch the AES attack while simulating gem5 for different instances to observe the effects.

- Although we have run the simulations with different configurations, we did not get the chance to perform more detailed analysis, that we could. We would like to carry out rest of the analysis in future. Meanwhile, we would also like to explore the characteristics of different RP while considering multi-core system.

CONCLUSION

The main objective of this study is to provide a comparative analysis in terms of a performance metric to evaluate three different cache replacement algorithms. One of the interesting parts of this study was to realize the individual influence of the algorithms while the time-driven side-channel attack occurs, although we have not been able to finish that. However, through this project, we have been able to understand the impacts of different cache configurations. Moreover, we also learned about time-driven cache channel attack, which we have been able to successfully run, although we did not implement it with the simulation platform of gem5. The entire study was performed using gem5, and OpenSSL.

REFERENCES

[1] Das P., Roy B.R. (2021) A Categorical Study on Cache Replacement Policies for Hierarchical Cache Memory. In: Mandal J., Mukhopadhyay S., Roy A. (eds) Applications of Internet of Things. Lecture Notes in Networks and Systems, vol 137. Springer, Singapore. https://doi.org/10.1007/978-981-15-6198-6_19

[2] Daniel A. Jiménez. 2013. Insertion and promotion for tree-based PseudoLRU last-level caches. In Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46). Association for Computing Machinery, New York, NY, USA, 284–296.

[3] A. Jain and C. Lin, "Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement," 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), 2016, pp. 78-89, doi: 10.1109/ISCA.2016.17.

[4] Yu, X., Xiao, Y., Cameron, K. and Yao, D.D., 2019. “Comparative Measurement of Cache Configurations’ Impacts on Cache Timing Side-Channel Attacks. “In 12th USENIX Workshop on Cyber Security Experimentation and Test ({CSET} 19).

[5] Jason Lowe-Power, ‘gem5 Documentation.’ Available at: https://www.gem5.org/documentation.