Comparision of Cache Replacement policies Using Gem5 simulator.

1st Abhiram Janagama
Department of Computer Science
UMBC
Arbutus,USA
ajanaga1@umbc.edu

2nd Pradeep Kumar Reddy Dasari Leela
Department of Computer Science
UMBC
Arbutus, USA
pdasari1@umbc.edu

3rd Lohith Prakash Konda
Department of Computer Science
UMBC
Arbutus, USA
lohithk1@umbc.edu

***Abstract*—When a cache exceeds its capacity, a cache replacement algorithm determines which cache elements to evict. There have been several replacement policies and cache configurations introduced over the years, but only a few have been proven to be effective. With the recent technology advances toward more associative caches, replacement policy, one of the fundamental variables affecting a cache's effectiveness, becomes even more crucial. Modern processors use a variety of policies, including Random, Least Recently Used (LRU), and Round-Robin, demonstrating that there is no consensus on which is the best. Cache memory block whose next reference is the farthest in the future, among all memory blocks existing in the set, would be replaced by an optimal but unreachable policy. In our project, we are primarily interested in gaining a better understanding of existing norms by experimenting with various cache configurations and analyzing their efficiency. This analysis will be carried out with the help of the GEM5 simulator, which will be used to run benchmarks with varying workloads and present the results. Cache associativity, memory hierarchy, cache replacement algorithm, and cache size are the four major components that affect cache output in this project. We've taken it a step further by experimenting with a variety of unconventional configurations.**

***Keywords*—Cache memory and performance, cache replacement policies, performance evaluation, Gem5, inclusive and exclusive caches.**

I. Introduction

Cache is a high-speed memory that is less expensive than registers yet faster than main memory. Cache memory is used to temporarily store data and information that is currently in use[1]. Because accessing data from main memory takes longer, a dedicated memory within the CPU is set aside to store small amounts of data for a period. When a CPU has cache memory, it takes less time for an instruction to be fetched from memory and processed. The lack of cache memory reduces the execution rate, lowering CPU performance.

Cache memory's main goal is to bridge the speed gap between slow memory and fast CPU at a minimal cost. It consists most of the most recently accessed portion of main memory. All data is saved in a memory device such as main memory. When the CPU or processor accesses certain data or information, it is copied to a faster storage medium, such as cache. System first checks its cache when a processor tries to access a piece of information again. If it's available in the cache, system uses it, else, it has to be fetched from main memory and copied into cache, expecting we'll need it again.

The performance of a cache can be quantified in terms of the Average Memory Access Time, which is calculated as follows:

Average Memory Access time = Hit Time + Miss Rate * Miss Penalty

Where, the time it takes to access a memory address in the cache is referred to as the hit time. Miss Rate denotes the percentage of total references in the cache that are missed to the total references, while Miss Penalty is the extra time necessary to service such a cache.

To lower the cache miss rate, compiler optimization techniques were implemented. Loop interchange, loop fusion, blocking, array merging, enhancing loop tiling with data alignment, and dynamic exclusion were among the techniques used. Nested loops are used in programs to access data in memory in a non-sequential fashion. Loop interchange essentially swaps the nesting of loops, allowing programs to access data in the order that it is stored. By enhancing spatial locality, loop interchange lowers misses. Loop fusion is the joining of two separate loops with the same looping and variables. The data in cache can be utilized repeatedly before being switched out by combining numerous loops into a single loop. Loop fusion improves temporal localization, which lowers misses.

The processor stalls for several clock cycles when accessing main memory. As a result, a faster memory that holds recently used data would increase the memory hierarchy's overall performance. Caches were created specifically for this reason. Caches are maintained small in order to allow for speedier access to them. However, the smaller the cache, the less data it can hold. As a result, the cache size has to be expanded without losing performance. As a result, caches are organized in hierarchies, with each level being labeled

There are usually three or rarely four levels of cache. They are usually referred as Level 1 (L1), Level 2 (L2), and Level 3 (L3) caches. They are classified as follows:

Level 1(L1) is a type of memory that stores and accepts data that's immediately stored in the CPU. Accumulator, address register and program counter, and are some of the most often used registers.

The Level2(L2) and Level3(L3) caches are larger than the L1 cache. Extra caches have been built between CPU and

RAM. L2 is sometimes incorporated into the CPU along with L1. The access time to the L2 and L3 caches is slightly longer than the L1 cache. The more L2 and L3 memory a computer has, the faster it can run.

Cache memories are still a hot topic in computer architecture research, with the ever-widening speed gap between CPU and memory, the need for a more efficient memory structure is emphasized [2]. As the number of cache levels in current processors grows, cache associativity grows. It's critical to reevaluate the efficacy of standard cache replacement policies. The cache controller must reject a cache memory line and replace it with fresh data from the main memory when all the lines in a cache memory become full and a new block of memory has to be inserted into the cache memory.

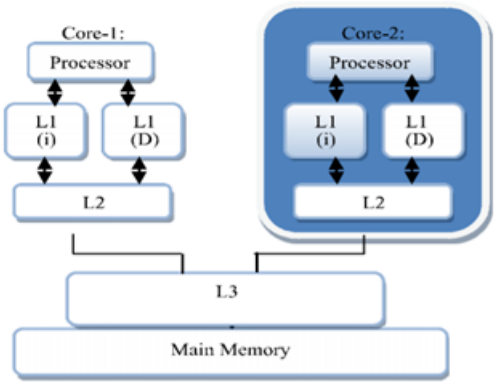

Fig. 1. Cache hierarchy

When a cache reaches its capacity, a cache replacement algorithm determines which cache elements to evict. The main factors influencing cache efficiency are latency and hit rate. Benchmark technologies are often used to calculate hit rates. The actual hit ratio varies greatly between applications. GEM5 is an object- oriented system architecture simulator platform that is open source and modular. GEM5 allows for quick and easy simulation of a complete system, including devices and operating system. Multiple Instruction Set Architectures (ISAs) and multiple processor cores are supported. GEM5 enables us to customize the functionality and design to suit our needs. The project's final deliverables will compare and analyze the performance of current cache replacement policies using benchmarks. We will gain a solid understanding of cache policies and how they affect overall CPU efficiency as a result of this activity.

II. Problem Statement

The rate of improvement in microprocessor speed is greater than that of DRAM (Dynamic Random Access Memory). So, while the performance gap between processor and memory is now a problem, it will get considerably worse somewhere downstream. The growing processor-memory performance gap is now the most significant roadblock to improve computer system performance. Cache replacement policies, which play an important role in cache performance and they are one of the key elements that can help close this performance gap. There are many replacement policies and cache configurations implemented over years and there are few policies and configurations that are proven to be efficient. In this project, we experiment with various cache designs and analyze the findings to acquire a better knowledge of established norms.

III. Proposed Approach

Cache replacement policies, which play an important role in cache performance and they are one of the key elements that can help close this performance gap. There are many replacement policies and cache configurations implemented over years and there are few policies and configurations that are proven to be efficient. In our project, we primarily focus on gaining a better understanding of established norms by experimenting with various cache configurations and analyzing their result.

The final deliverables will compare and evaluate the performance of current cache replacement policies with different configurations using benchmarks. We are going to analyze the four major components that affect cache efficiency to get better understanding :

1.Cache associativity
2.Memory hierarchy, cache levels
3.Cache replacement algorithms
4.Cache size.

We choose Gem5 over other simulators, as it is much easier to perform different measurements on cache replacement policies.

IV. Components that effect Cache efficiency

Even though much research has been done in the field of cache memories, some key questions about cache replacement policies applied to current workloads remain unanswered. The well-known observations about replacement policies, as well as related questions to which our study provides answers, are mentioned in this section.

A. Memory Hierarchy and Cache levels

Memory Hierarchy is a feature of computer system design that helps to arrange memory such that access time is reduced. The Memory Hierarchy was created using a programming technique called locality of references. The following diagram depicts the various layers of memory hierarchy.

Despite the memory delay of main memory access, this architecture was intended to enable CPU cores to process data faster. Since the CPU waits for data from main memory, accessing it may be a barrier for CPU core performance, and having all of main memory large speed can be incredibly costly

We can get the following details about the hierarchy from the above mentioned figure:

1.Capacity: It refers to the total amount of data that the memory can hold. The ability grows as we go down the Hierarchy from top to bottom.

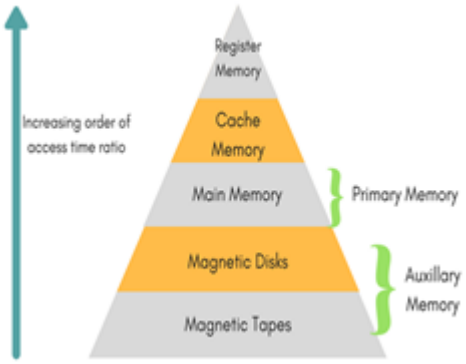

Fig. 2. Memory hierarchy

2.Performance: Owing to the significant disparity in access time between the CPU registers and the Memory when the computer system was configured without Memory Hierarchy architecture, the speed distance between the CPU registers and Main Memory increased. As a consequence, the system's efficiency suffers, necessitating enhancement.

3.Access time: It refers to the time between a read/write request and the data being accessible. The access time increases as we progress down the Hierarchy

B. Replacement Algorithms

Cache algorithms (also known as cache replacement policies) are optimization r algorithms, that a computer algorithm or a hardware-maintained configuration can use to handle a cache of data stored on the computer.

Caching boosts consistency by storing recent or often used data in memory locations that are easier or less expensive to reach than traditional memory stores. In order to achieve optimal cache utilization, replacement algorithms/policies are used. When the cache is completed, substitution policies determine which data is removed and make room for new data that is already being used.

The cache replacement algorithms have a clear bias toward the program's design and overall efficiency. To gain a deeper understanding of this, we decided to deviate from the norm of using the same replacement algorithms at all levels by experimenting with different cache replacement policies at different levels.

C. Associativity

Cache associativity is nothing but the number of locations in the cache where the block can be found. Since changing associativity has a huge effect on cache efficiency and expense, choosing the best associativity is critical. Increased associativity increases the chances of a block becoming a resident by lowering the chance that too many recently-referenced blocks map to the same location and allowing more blocks to be eligible for substitution.

- Direct mapping- The cache is structured into numerous sets with a single cache line per set in a direct-mapped cache structure. It can only occupy one cache line based on the memory block's address. A (n*1) column matrix can be utilized to address the cache. A location space is comprised of two sections: a record field and a label field. The label field is cached however, the remainder of the information is protected in the primary memory. The presentation of direct planning is straightforwardly relative to the hit proportion.
  For cache access, every primary memory address might be considered as having three fields. The most un-huge w bits inside a block of primary memory address a solitary word or byte. In most current machines, the location is at the byte level. The excess s bits address one of the 2s obstructs in the principle memory. This situation strategy saves power by keeping away from the need to examine across all cache lines.
- Associative Mapping – The cache is structured into a single cache set with many cache lines in a fully associative cache. Any of the cache lines can be occupied by a memory block. The cache can be coordinated as a (1*m) row network. The fully associative cache structure allows us to place memory blocks in any of the cache lines, allowing us to fully utilize the cache. The cache hit rate is improved by the placement policy. This arrangement is delayed as it needs to go through all the cache lines.
- Set- Associative Mapping – The set-associative cache is a trade-off between a Fully cooperative cache and a direct-planned cache. In this type of mapping, the content and addresses of the memory word are stored using associative memory. In any cache line, any block can be inserted. If a cache miss occurs, it gives you the option of using replacement algorithms.

V. Research Analysis on Cache

A. Research Idea 1: Cache levels

The memory hierarchy is designed in such a manner that having a lower, faster cache near the processor and a bigger, slower cache near the memory yields better performance. In this project, we swapped the L2 and L3 caches and discovered interesting behavior; the new configuration is L1 cache, L3 cache, and L2 cache, respectively.

Increasing the cache levels may improve the performance but the hardware and cost complexities comes into play. Intel tried to increase the cache levels to 4 but it is still not optimal for every computer because of hardware complexity.

B. Research Idea 2: Mixing algorithms in cache levels

We tried to evaluate with different configurations of cache levels. We mixed the cache replacement policies in three cache levels to understand how it works and to get the optimal configuration and analyze the performance. The detailed statistics is shown in evaluation section.

Fig. 3. Cache levels

In table 1 first column there are numerous configurations which we have analyzed. For the three levels we tried to use LRU for first level,FiFo for second etc..,

Fig. 4. Cache levels with varying policies

C. Research Idea 3: Cache Associativity

Larger sets and higher associativity result in less cache conflicts and lower miss rates, but they come at a higher cost in terms of hardware. As a result, coming up with the right associative value for each cache level is a challenge, so we decided to experiment with different values and provide the one that performs well.

Fully Associative – the best miss rate, (set size of 2^k)

Set Associative – (Intermediate)

Direct-Mapped (set size from 1)

D. Research Idea 3: Locality favoured by Cache replacement policy

Different types of locality:

Temporal – An object that has recently been accessed is likely to be accessed again quickly.

Spatial-If an object has recently been accessed, nearby objects are more likely to be accessed again.

VI. COMMON CACHE REPLACEMENT POLICIES

Cache memories are still a popular topic in computer architecture. The growing speed gap between processor and

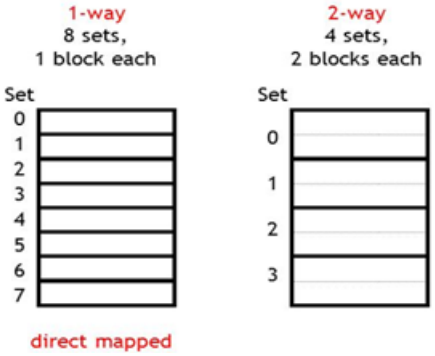

Fig. 5. Direct mapped cache

The figure above depicts a one-way associative cache and a two-way associative cache. The images below display four-way and eight-way associative caches.

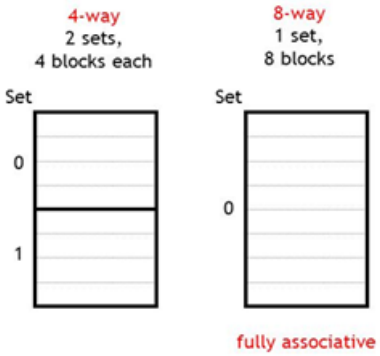

Fig. 6. Fully associative cache

memory highlights the need for an efficient memory hierarchy and a re-evaluation of common cache replacement policies. The cache controller must discard an existing cache memory line and replace it with the new line when the cache memory is complete, and a new block of memory needs to be inserted in the cache. The cache block to be evicted should ideally not be used again in the future. Since it is difficult to predict how long the information will be required in the future, current replacement policies predict based on cache activity.

Document placement/replacement algorithms have a huge impact on the performance of proxy caches. Cache placement algorithms attempt to position documents inside a cache, while cache replacement algorithms attempt to remove documents from one. Only if such algorithms can improve the hit ratio are they useful (see Glossary for a definition of hit ratio). Cache placement, in comparison to cache replacement, has received little attention.

Cache replacement algorithms often seek to reduce various parameters such as hit ratio, byte hit ratio, access cost, and latency (refer Glossary for definitions).

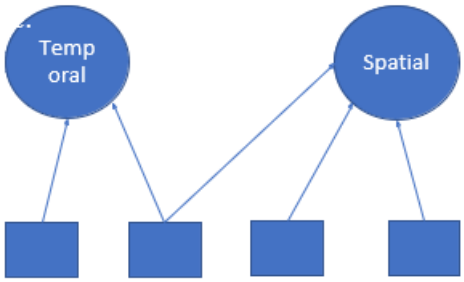

Fig. 7. temporal and spatial locality

The most commonly used cache replacement algorithms are Least Recently Used (LRU), Least Frequently Used (LFU), LRU-Min, LRU-Threshold, Pitkow/Recker, SIZE , Lowest Latency First , Hyper-G, Greedy-Dual-Size (GDS), Hybrid , Lowest Relative Value (LRV) , LNC Lowest Relative Value (LRV), LNC-R-W3 , Bolot/Hoscka , Size-adjusted LRU (SLRU), Least Unified-Value (LUV) , and Hierarchical Greedy Dual (HGD). Several of these attempt to maximize the hit ratio. Cache replacement algorithms can be classified in a variety of ways (for a representative categorization and a survey of Web cache replacement strategies).

LRU, LFU, Pitkow/Recker, and some of their variants are examples of traditional cache replacement algorithms. Least Recently Used (LRU) removes the item from the cache that has been requested the fewest times recently. Least Frequently Used (LFU) removes the item from the cache that has been retrieved the fewest times. Unless all objects are referenced in a single day, in which case the biggest object is substituted, expels objects in LRU order. When such parameters such as hit ratios, height, and so on are used to evaluate cache replacement algorithms, it's likely that they won't perform well. Proxy servers can reduce the number of requests sent to favorite servers, the amount of network traffic generated as a result of document requests, and the access latency. They use two metrics to look at the first two: cache hit ratio and byte hit ratio (see Glossary for definitions of these terms).

They calculate the overall feasible hit ratio and byte hit ratio using trace-driven simulations. They come to the conclusion that the norms used by LRU and Pitkow/Recker do the worst in their simulation. They argue that replacing documents based on their size is preferable because it maximizes the hit ratio in each of their workloads.

A. Least Recently Used (LRU)

The things that least as of late utilized things initially are disposed of. This calculation requires monitoring what was utilized when, which is expensive on the off chance that one needs to guarantee that the calculation consistently disposes of the item that has been utilized the least as of late. "age bits" for cache lines and the following of the "Least Recently Used" cache line dependent on age-bits.

Any time a cache line is utilized along these lines, the age of any remaining cache lines changes. After its last use, a line in LRU swap stays in the cache for quite a while prior to turning into the LRU line. Such cutoff times limit the cache space accessible for different lines superfluously. Moreover, in staggered caches, worldly reuse designs are regularly switched, showing up in the L1 cache however not in the L2 cache because of the L1 cache's separating impact.

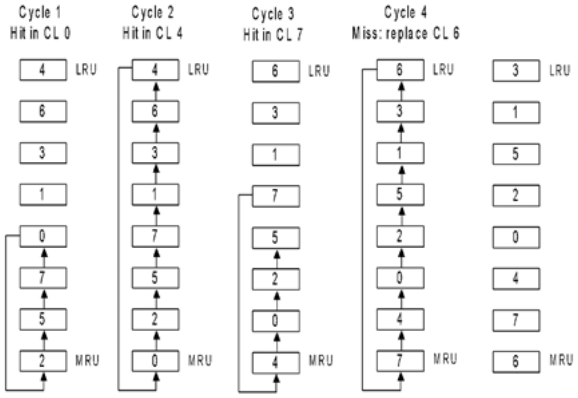

Fig. 8. Example of LRU policy

B. First In First Out (FIFO)

The Round Robin (or FIFO) substitution heuristic essentially replaces the cache lines in succession, with the most seasoned square in the set being supplanted first. Each cache memory assortment accompanies a roundabout counter that focuses to the following cache. that should be supplanted; the counter is refreshed after each cache miss. The cache behaves like a FIFO line when utilizing this calculation.

Page replacement algorithms are needed in working frameworks that utilization paging for memory the executives to figure out which page should be supplanted when another page shows up. At the point when another page is mentioned yet isn't accessible in memory, a page issue happens, and the one of the current pages replaces with the recently required page by the operating system.

Different page replacement algorithms propose various methods for determining which pages should be replaced. Both algorithms strive to reduce the number of page faults. This is the most basic page replacement method. The operating system utilizes this algorithm to monitor all pages in memory in a line, with the most seasoned page at the top. At the point when a page must be supplanted, the main page in the line is picked for substitution.

C. Random Replacement

This algorithm picks a data object from the cache at random and replaces it with the desired one. This algorithm does not need any data structure or to keep track of the contents of the data in the past. As a result, it uses less resources and therefore has a lower cost than other algorithms.

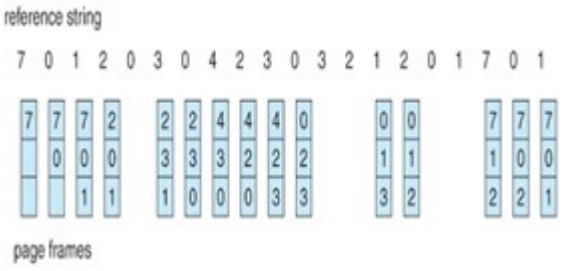

Fig. 9. Example of FIFO policy

This algorithm does not necessitate the storage of any access history data. It has been used in ARM processors because of its simplicity. A non-sequential victim counter is used in the selection algorithm. The handler of a pseudo - random replacing algorithm updates the counter by choosing an increment value at random and applying it to it. The counter is reset to a given value when it exceeds its maximum value.

When space is needed, picks a suitable item at random and discards it. This approach does not require the storage of any access history data. It has been employed in ARM processors because of its simplicity.

D. Most Recently used

MRU takes the same approach to LRU. The data or method with the highest score in recent usage is replaced with the current data or process, according to MRU. In a block-replacement technique, the previously used block is replaced with a new block. The Most Recently Used technique is a form of buffer-replacement strategy.

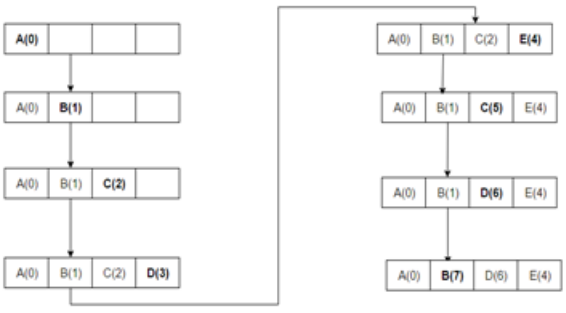

Fig. 10. Example of MRU policy

E. Pseudo LRU

Pseudo-LRU, also known as PLRU, is a cache algorithm that improves on the accuracy of the Least Recently Used (LRU) algorithm by replacing values with estimated measurements of age rather than keeping track of each value's exact age.

The implementation cost of LRU becomes expensive for CPU caches with high associativity (usually ¿4 ways). Because a system that nearly always discards one of the least recently utilized items is sufficient in many CPU caches, many CPU designers choose for the PLRU method, which only requires one bit per cache item to function. When compared to LRU, PLRU has a little lower miss ratio, slightly better latency, consumes slightly less power, and has reduced overheads.

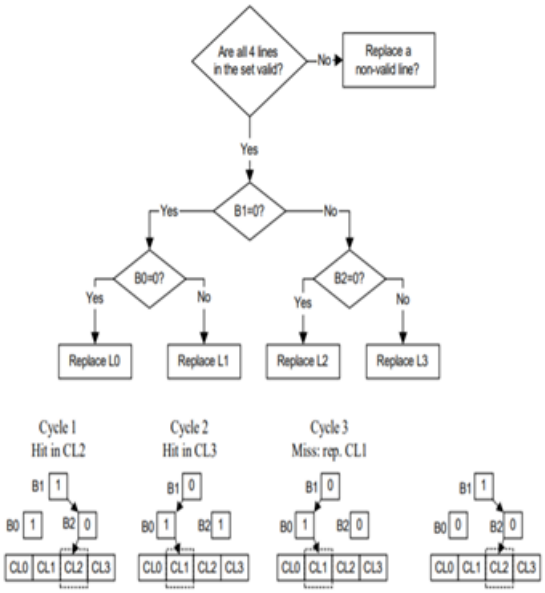

Fig. 11. Example of Pseudo LRU policy

VII. INCLUSIVE AND EXCLUSIVE CACHES

Contingent upon whether the substance of one store is available in another degree of caches, staggered stores can be developed in an assortment of ways. The lower-level cache ought to be comprehensive of the more significant level cache ifall squares in the more basic level cache are also present in the lower-level cache miss occurs.

The lower-level cache should be first- class to the more raised level cache if it joins impedes that are not in the more huge level cache miss occurs the benefit of an inclusive strategy is that if a cache miss occurs in parallel systems with per-processor private caches, other peer caches are checked for the block. On the off chance that the lower level cache contains the more significant level cache and the lower level cache contains a miss, the more elevated level cache doesn't need to be looked.

This means that an inclusive cache has a lower miss latency than an exclusive cache. The lower level cache determines the cache's unique memory capacity, which is a disadvantage of an inclusive policy. Unlike exclusive cache, where the unique memory capacity is sum of all caches in the hierarchy.

VIII. MODULES

cache-level.py = to update the cache size, associativity etc.

three-level-cache.py = main python file which will be run to get the results.

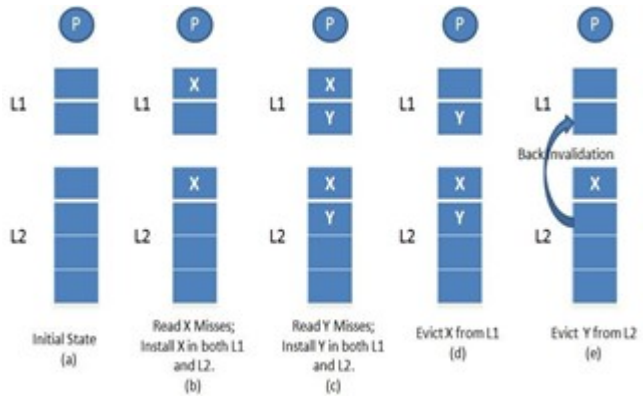

Fig. 12. Inclusive Cache example

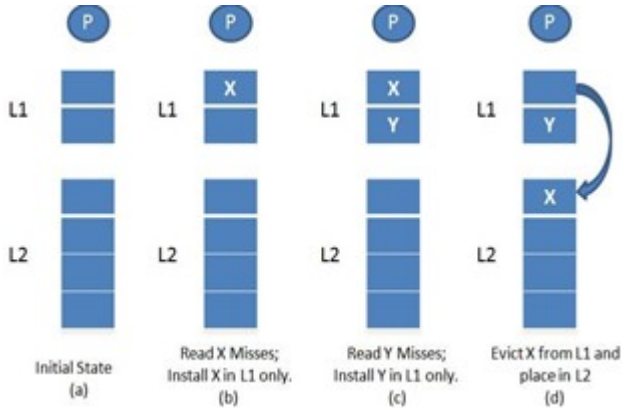

Fig. 13. Exclusive Cache example

two-level.py = main python file to run program with two caches.

Cache.py = Main configuration file to update which ALGORITHM to run on each level.

base.cc = Will have all the header files i.e all the algorithms which needs to be used.

fifo.cc/lru.cc/random-repl.cc = main C++ program

fifo.hh/lru.hh/random-repl.hh = header file to import all the settings to be used.

IX. EXPERIMENTAL ENVIRONMENT

The GEM5 and source code written in Python were used to evaluate the performance of various cache replacement policies. The benchmarks were taken from a study conducted by the University of Virginia, which looked at a variety of demands placed on the virtual processor with varying workloads.

Below is the link to the benchmark programs:

https://www.cs.virginia.edu/cr4bd/6354/F2016/h mework2.htmlpart-c-effective-cache-miss-penalty-versus-miss-latency-required-f or-checkpoint

A. GEM5 simulator

The gem5 simulator is a scalable, robust simulation device that can be used to evaluate a wide variety of structures and is freely accessible to all researchers. This infrastructure offers a wide range of CPU models, system execution configurations, and memory system configurations, allowing for maximum flexibility.

The gem5 simulator is an adaptable structure for examining PC framework design, including both framework level and processor microarchitecture. GEM 5 has a variety of CPU models that can be swapped out. The CPU models can be mixed and matched to create homogeneous and heterogeneous multi-core systems. GEM 5 decouples ISA semantics from CPU models, allowing it to accommodate multiple ISAs effectively. GEM 5 presently upholds the models SPARC, ARM, POWER, MIPS, RISC-V, Alpha and x86. For our project, we chose the X86 architecture.

Researchers will work on a certain feature of the technology without having to learn the whole code base thanks to a dedication to modularity and clean interfaces. The coding is open to all researchers thanks to the BSD-based licence, which removes all ethical stumbling blocks

1) Properties of Gem5:

- The gem5 simulator's main purpose is to serve as a community platform for architectural modeling. Flexible modeling to cater to a broader variety of consumers, wide availability and usefulness for the community, and high levels of developer engagement to promote collaboration are three main aspects of this aim.
- Every effective simulation infrastructure must have a high level of flexibility. Architects, for example, need a method that can test structures at different levels of detail while maintaining simulation speed and precision as an idea progresses from a high-level model to a particular specification. Different types of experiments can necessitate different levels of simulation.
- A comprehensive CPU model might be required for a fine-grain clock gating experiment, but modeling multiple cores is not. Meanwhile, while a highly scalable interconnect model can necessitate many CPUs, such CPUs do not need much information.
- In addition, by continuing to use the same infrastructure, an architect would be able to complete further work faster and with less overhead. The gem5 simulator has a wide range of capabilities and modules, giving it a great deal of versatility.
- The gem5 simulator can also run workloads in a variety of ISAs, including x86 and ARM, which are the most popular today. The number of workloads and configurations that gem5 can emulate has increased dramatically as a result of this. Simulators that simulate the whole device are difficult to use. The gem5 simulator has taken dozens of person-years to build, including both the infrastructure for modular modeling and the various comprehensive component models.

2) Design Features:

- Flexibility: The gem5 simulator's flexibility is a core aim and feature of its performance. Object-oriented architecture is the primary means of achieving flexibility. Advanced technologies such as multi-core and multi-system modeling follow directly from the ability to create structures from discrete, composable artifacts.
- SimObjects are used to represent all major simulation elements in the gem5 simulator, and they all have the same setup, configuration, statistics, and serialization behaviors. For system-call emulation, SimObjects provide models of specific hardware components such as cpu cores, caches, circuit elements, and modules, as well as more abstract elements such as a process and its related process context. Any SimObject is defined by two classes, one in Python and the other in C++, all of which are derived from the SimObject base classes in both languages. The SimObject's parameters are defined in the Python class specification.
- The Python base class includes standardized methods for parameter instantiation, labeling, and setup. The SimObject's state and remaining actions, as well as the performance-critical computer simulation, are all covered by the C++ class.

3) Python Integration: The close integration of Python in the gem5 simulator gives the simulator a lot of control. Although the simulator is coded in C++ for the most part, Python is used in every element of its running. In both Python and C++, all SimObjects are mirrored. Initialization, setup, and simulation power are all handled by Python. On initialization, the simulator starts executing Python code almost immediately; the basic main() method is written in Python, as is all control processing and launch code.

4) Benchmarks: We've chosen a few benchmark programs to test a variety of demands on the simulated processor.

BFS - solves a problem using the breadth-first search method. This software was chosen because its data cache locality is expected to be weak.

Sha - computes the input's SHA-1 cryptographic hash. The software was chosen because it should be heavy on integer operations and easy to branch predict and cache.

N-Queens - solves the N-Queen problem given a n as an argument. The Queens software was chosen because it appears to be both cache-friendly and branch prediction-challenging.

Blocked-matmul - A 2X2 register-blocked matrix multiplication of two 84X84 matrices is called Blocked-matmul. All sizes are hard coded, and the matrices are pseudo-randomly generated. This software was chosen because it could have both cache and floating-point operations.

5) Measurements: In this project, we break down the presentation of the new setup by estimating L1 Cache Overall Miss Rates and Program Execution Time. All insights are saved in the m5src organizer's stats.txt document. We utilize the equation underneath to decide the overall miss rate.

L1 Overall Miss Rates = (Total misses in d-cache + total misses in i-cache) / (Total accesses in d-cache + total accesses in i-cache)

Program Execution = (time taken in seconds to run the Benchmark Programs)

B. Environment setup

We'll use SCons to build gem5. SCons sets up a series of variables in the SConstruct file (gem5/SConstruct), then uses the SConscript files in each subdirectory to locate and compile all gem5 source. When SCons is run for the first time, it builds a gem5/build directory.

The files created by SCons, the compiler, and other tools can be found in this directory. Each collection of options that you use to compile gem5 will have its own directory. The build opts directory contains a variety of default compilation options. These files contain the parameters passed to SCons when gem5 was first created. We'll stick with the X86 defaults and tell the compiler to compile all of the CPU models. The default values for the Scons options can be found in the package build opts/X86.

The key argument forwarded to SCons is install/X86/gem5.opt, which specifies what you want to build. We're making gem5.opt in this situation (an optimized binary with debug symbols). Gem5 should be installed in the build/X86 directory. Since this directory does not yet exist, SCons can search in build opts for the X86 default parameters.

```
.... <lots of output>
....
[   SHCXX] nomali/lib/mali_midgard.cc -> .os
[   SHCXX] nomali/lib/mali_t6xx.cc -> .os
[   SHCXX] nomali/lib/mali_t7xx.cc -> .os
[      AR]  -> drampower/libdrampower.a
[   SHCXX] nomali/lib/addrspace.cc -> .os
[   SHCXX] nomali/lib/mmu.cc -> .os
[  RANLIB]  -> drampower/libdrampower.a
[   SHCXX] nomali/lib/nomali_api.cc -> .os
[      AR]  -> nomali/libnomali.a
[  RANLIB]  -> nomali/libnomali.a
[     CXX] X86/base/date.cc -> .o
[    LINK]  -> X86/gem5.opt
scons: done building targets.
```

Fig. 14. The output when build folder gets created

X. Experiment Results

The names in the first column of Table 2 are the x-axis parameters in the graphs below.To generate the graphs to be plotted, we generated a parser in Python.

The grouped bar chart below(Figure 15) depicts the results of four benchmark programs for various configurations. It's obvious that the execution times for all of the configurations shown above follow a similar pattern. BFS takes the longest

RAM size	2048 MB
Memory Mode	Timing
CPU Type	Timing Simple CPU
L2 Memory Bus	Coherent Cross Bar
L3 Memory Bus	Coherent Corss Bar
System Domain Clock	3GHz

TABLE I
GEM5 BASE CONFIGURATION

time, while n-Queen takes the shortest (for this experiment, we used n=10). However, it's difficult to say which configuration performs better. As a result, we've combined the program execution time and created a new bar chart as shown below. Table 2 provides more information on each configuration.

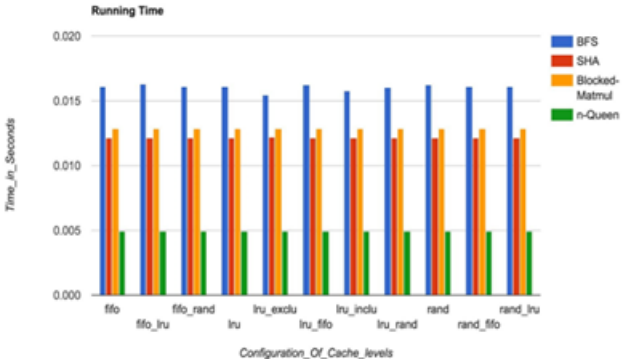

Fig. 15. Execution time for varying workloads for various configurations

The consolidated bar chart below (Figure 16) depicts the total execution time for the benchmark programs for the specified configuration. The lru exclu configuration clearly outperforms the others. The fifo lru configuration takes the longest to run.Same policy at all levels - When comparing the execution times of (lru, fifo, and rand), we find that lru wins by a small margin. Surprisingly, we discovered that rand does not have the fastest execution time. Different policies at different levels - When comparing (fifo lru, fifo rand, lru fio, lru rand, rand fifo, rand lru), we find that lru rand takes the shortest time to execute.

Changing the Memory Hierarchy - We discovered that lru exclu has the quickest execution time as compared to (lru exclu, lru inclu). Table 2 presents each configuration detail.

The L1 miss rates of four benchmark programs for various configurations are shown in the grouped bar chart (Figure 17) below. It's obvious that all of the above configurations have

Name	L1_policy	L2_policy	L3_policy	L2_size	L3_size	L1_assoc	L2_assoc	L3_assoc
fifo	fifo	fifo	fifo	256 kb	512 kb	2	8	8
lru	lru	lru	lru	256 kb	512 kb	2	8	8
rand	rand	rand	rand	256 kb	512 kb	2	8	8
fifo_rand	fifo	rand	lru	256 kb	512 kb	2	8	8
lru_rand	lru	rand	fifo	256 kb	512 kb	2	8	8
fifo__lr	fifo	lru	rand	256 kb	512 kb	2	8	8
rand_lru	rand	lru	fifo	256 kb	512 kb	2	8	8
rand_fifo	rand	fifo	lru	256 kb	512 kb	2	8	8
lru_fifo	lru	fifo	rand	256 kb	512 kb	2	8	8
lru_inclu	lru	lru	lru	512 kb	256 kb	2	8	8
lru_exclu	lru	lru	lru	512 kb	256 kb	2	8	8
2	lru	lru	lru	256 kb	512 kb	2	2	2
4	lru	lru	lru	256 kb	512 kb	4	4	4
8	lru	lru	lru	256 kb	512 kb	8	8	8
248	lru	lru	lru	256 kb	512 kb	2	4	8
842	lru	lru	lru	256 kb	512 kb	8	4	2

TABLE II
VARIOUS GEM5 CONFIGURATIONS

a common trend for miss rates in L1. BFS has the highest miss rate, while Sha encryption has the lowest. However, it is difficult to determine which configuration performs better. As a result, we've combined the L1 miss rates and produced a new bar chart below. To our surprise, having a lower miss rate at L1 does not necessarily imply faster execution. This is evident in the Sha software, which has a lower miss rate but takes longer to execute.

The L1 miss rates for various cache configurations are shown in the combined graph below(Figure 18). Of all of the configurations, lru exclu performs the best. lru incl is the

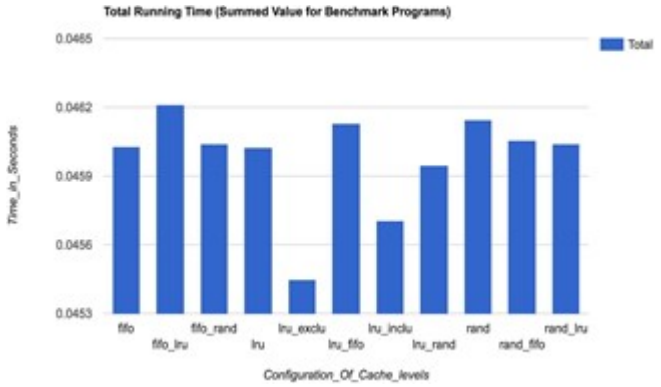

Fig. 16. Consolidate execution time

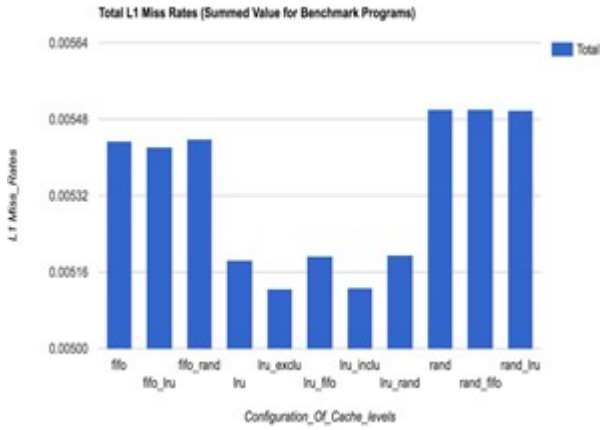

Fig. 18. L1 cache miss rate for varying set associativity

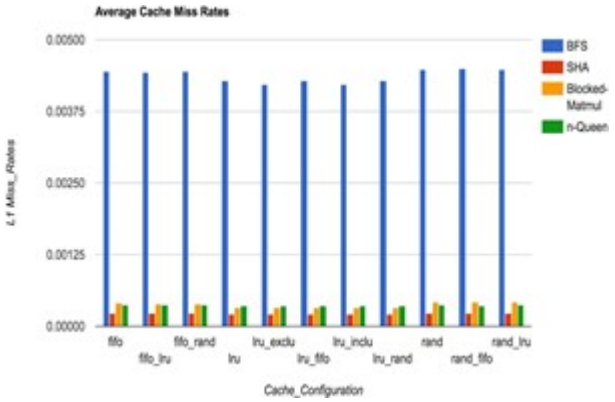

Fig. 17. L1 cache miss rate for different workloads

second configuration with lower miss rates, and lru is the third. While (rand, rand fifo, and rand lru) have similar miss rates, we can see from Graph 2 that rand lru has the shortest execution time. The majority of the configuration in the graph above is based on execution time.

In other words, lower miss rates are directly proportional to faster program execution. However, we have another configuration that indicates that, despite the higher miss rates, the software runs faster. Table 2 provides more information on each configuration.

For the four benchmarks, the below(Figure 19) bar chart depicts the average miss rate at L1 cache. The total miss rate is represented by the Y axis, while the associativity is represented by the X axis. At all three cache stages, 2 reflects set associativity. At all three levels of cache, 4 and 8 reflect 4-way set associativity and 8-way set associativity, respectively. 248 denotes the two way, four way, and eight way set associativity are introduced at the L1, L2, and L3 caches, respectively, while 842 denotes the opposite. It's clear that all of the programs work in the same way with all of the different configurations. There isn't much of a difference. Since the above bar graph makes it difficult to draw conclusions, we've included the consolidated miss rates in the graph below. Table 2 provides more information on each configuration.

The graph below depicts(Figure 19) the average miss rates at the L1 cache for various configurations. The configurations we checked are indicated by the X-axis, and the specifics are described in the configurable configurations table above. We can see that 8-way set associativity outperforms the others because it has the lowest miss rate. The explanation for this is well understood: greater set associativity at lower cache levels often favors better output. Table 2 provides more information on each configuration.

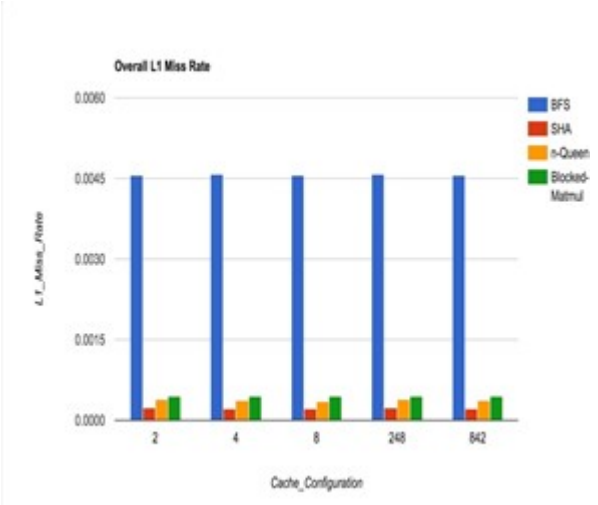

Fig. 19. L1 cache miss rate for varying associativity

Individual CPU Time for all of the different benchmarks at L1 cache is represented in the graph below. The x-axis depicts the various configurations of set associativity at the L1, L2, and L3 cache levels (details listed in the above configurable configurations table). The CPU Time in seconds is shown on

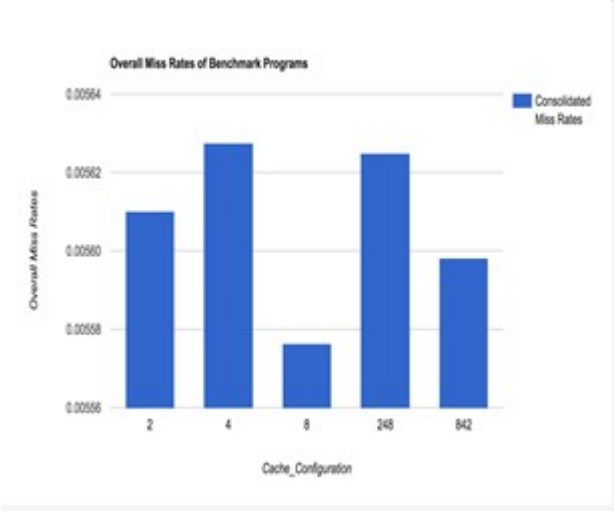

Fig. 20. Consolidated L1 miss rate

the y-axis. And when the associativity is modified, all of the programs work at the same pace.

We took a consolidated rate and plotted the graph below since these parameters do not yield any conclusions. Table 2 provides more information on each configuration.

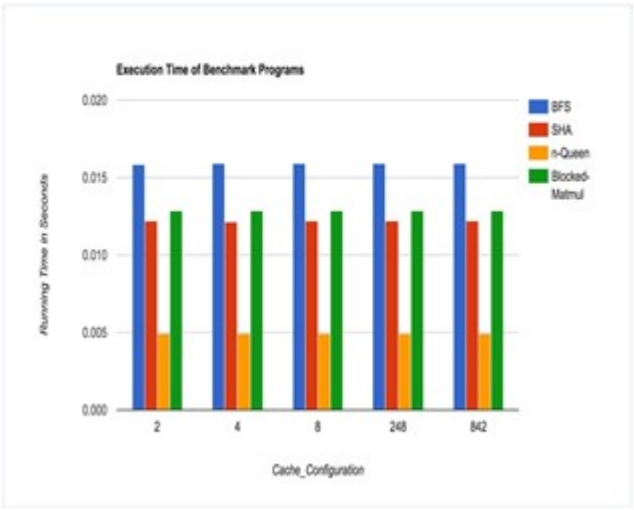

Fig. 21. Execution time for varying set for different loads

The consolidated CPU Times are shown in seconds in the graph above. We can see those programs with fixed associativity 2 perform better because they take up the least amount of CPU time. However, as shown in Graph 6, even though the miss rates at 8-way set associativity are very poor, the CPU does not perform any better. This is due to the fact that the CPU Time is influenced not only by miss rates, but also by other factors such as miss latency, tag latency, and so on, and thus a high CPU Time reduces overall CPU efficiency.

Table 2 provides more information on each configuration.

Fig. 22. Consolidate execution time

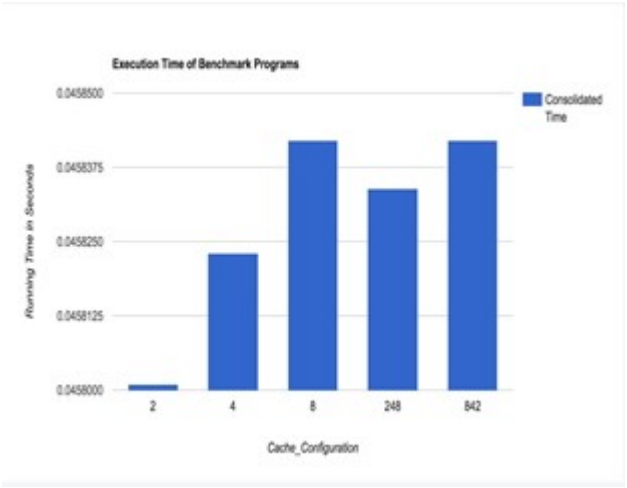

Fig. 23. overall consolidated execution time for all configurations

XI. CONCLUSIONS AND RECOMMENDATIONS

This project greatly aided us in learning and comprehending the cache hierarchy. Let's recap the study questions and wrap up with the findings we discovered during the experiments.

Various cache replacement policies that have been applied in the past have been analyzed and compared .The following parameters are used to compare policies:CPU time ,cache misses, cache hits, resource allocation, and extra parameters.

A. Swapping Memory Hierarchy

Conclusions are drawn from the graphs 2 and 4 above. Cache levels 2 and 3 are swapped in the lru inclu and lru exclu configurations. Tables 1 and 2 provide more information on the setup. Based on the results of all of the above configurations, we discovered that adjusting the memory hierarchy improves efficiency and reduces miss rates. The lru exclu has the shortest running time and the lowest

miss rate since the hierarchy is switched and level 2 and 3 are made exclusive. Better output and lower miss rates are provided by the memory hierarchy.

Conclusion 1: Swapping Cache Level 2 and 3 yields better performance.

Conclusion 2: Making Level 2 and 3 cache exclusive and level swapped shows the best performance among all configurations.

B. Cache Replacement Policy

The above graphs 2 and 4 are used to draw conclusions.

The cache replacement policy is swapped in the configuration of lru, rand, fifo, lru fifo, fifo lru, rand fifo, rand lru, fifo rand, and lru rand. Tables 1 and 2 provide more information on the setup. Surprisingly, rand performed admirably, while fifo rand had the slowest execution time. To summarize, having a different cache replacement policy does not improve efficiency.

Conclusion 3: Having different cache replacement policy doesn't provide any gain in performance.

Conclusion 4: Lesser Miss rates provides better performance, but the inverse is not true.

C. Associativity

Conclusions drawn from the data in Graphs 6 and 8.

Even though the miss rates are very low at 8-way set associativity, the CPU does not perform better, as shown in Graph 6. This is due to the fact that the CPU Time is influenced not only by miss rates, but also by other factors such as miss latency, tag latency, and so on, and thus a high CPU Time reduces overall CPU efficiency.

Conclusion 5: Having larger set associativity value yields lesser miss rate but that does not guarantee an increase CPU performance.

REFERENCES

- http://www.cs.ucf.edu/ neslisah/final.pdf
- http://www.cse.scu.edu/ mwang2/projects/Cache_replacement _10s.pdf
- https://www.geeksforgeeks.org/memory-hierarchy-design-and-its-characteristics/
- https://en.wikipedia.org/wiki/Cache_replacement_policies
- https://hal.archives-ouvertes.fr/hal-01700364/document
- http://www.ece.uah.edu/ milenka/docs/milenkovic_acmse_04r.pdf
- Ackland B., Anesko D., Brinthaupt D., Daubert S.J, Kalavade A.., Knoblock J., Micca E., Moturi M., Nicol C.J., O'Neill J.H., Othmer J., Sackinger E., Singh K. J., Sweet J., Terman C. J., and Williams J., "A Single-Chip, 1.6 Billion, 16-b MAC/s Multiprocessor DSP," IEEE Journal of Solid-state circuits, Vol. 35, No. 3, March 2000, pp. 412-423.
- Bansal,S.,and Modha,D.S.,"CAR: Clock with Adaptive Replacement", Proceedings of 3rd USENIX Conference on File and Storage Technologies, Volume 4, pp. 187-200, 2004.
- Butt, A.R.,Gniady, C., and Hu, Y.C., "The Performance Impact of Kernel Prefetching on Buffer Cache Replacement Algorithms", ACM SIGMETRICS Performance Evaluation Review, Volume 37, No. 1, pp. 157-168, 2005.
- Abdel F.A., and Samra, A.A., "Least Recently Plus Five Least Frequently Replacement Policy (LR+5LF)", International Arabic Journal of Information Technology, Volume 9, No. 1, pp. 16-21, 2012.
- Megiddo, N.,and Modha, D.S., "Outperforming LRU with an Adaptive Replacement Cache Algorithm", Computer, Volume 37, No. 4, pp. 58-65, 2004.
- Jaleel Aamer, B. Theobald Kevin, C. Steely Simon and Joel Emer, "High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)", ISCA10, pp. 19-23, June 2010.
- Liqiang He, Yan Sun and Chaozhong Zhang, "Adaptive Subset Based Replacement Policy for High Performance Caching", JWAC 2010-1st JILP Worshop on Computer Architecture Competitions: Cache Replacement Championship, 2010.
- Kathryn.S. McKinley, Steve Carr and Chau-Wen Tseng, Improving data locality with loop transformations, vol. 18, no. 4, July 1996.
- Preeti Ranjan Panda, Nakamura, Nikil D. Dutt and Alexandru Nicolau, "Augmenting Loop Tiling with Data Alignment for improved cache performance", IEEE Transactions on computers, vol. 48, no. 2, February 1999.
- https://ece752.ece.wisc.edu/lect11-cache-replacement.pdf
- http://www.ece.uah.edu/milenka/docs/milenkovic _acmse04 r.pdf
- http://www.cs.utexas.edu/users/mckinley/papers/evict-me-pa ct-2002.pdf
- http://snir.cs.illinois.edu/PDF/Temporal20and20Spatial 20Locality.pdf
- https://math.mit.edu/ stevenj/18.335/ideal-cache.pdf
- https://pdfs.semanticscholar.org/6ebe/c8701893a6770eb0e19 a0d4a732852c86256.pdf
- https://fD3hhNnfL6kwww.youtube.com/watch?v=
- http://pages.cs.wisc.edu/ david/courses/cs752/Spring2015/gem5-tutorial/index.html
- http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/
- https://github.com/dependablecomputinglab/csi3102-gem5-n ew-cache-policy
- Sawant, R., Ramaprasad, B.H., Govindwar, S., and Mothe, N., "Memory Hierarchies-Basic Design and Optimization Techniques Survey on Memory Hierarchies – Basic Design and Cache Optimization Techniques", 2010.
- Agarwal, A., and Pudar, S.D., "Column-Associative Caches: Caches A Technique for Reducing the Miss Rate of Direct-Mapped", ACM, Volume 21, No. 2, pp. 179-190, 1993.

- Ramaswamy, S., and Yalamanchili, S., “Improving Cache Efficiency via Resizing + Remapping”, Proceedings of IEEE 25th International Conference on Computer Design (ICCD), pp. 47–54, 2007.
- Podlipnig, S., and Böszörmenyi, L., “A Survey of Web Cache Replacement Strategies”,ACM Computing Surveys, Volume 35, No. 4, pp. 374-398, 2003. Chavan, A.S., Nayak, K.R., Vora, K.D., Purohit, M.D., and Chawan, P.M., “A Comparison of Page Replacement Algorithms”, International Journal of Engineering and Technology, Volume3, No. 2, pp. 171-174, 2011.