CSCE 614: HW4 REPORT (SRRIP)

Nikhil Kulkarni   
Dept. of ECEN, TAMU

*Abstract* – Cache is a high-speed memory layer that enables quick data access for processors, thus streamlining efficiency and retrieval speed. It holds commonly used data and instructions, cutting down the CPU's main memory access time. Structured in levels from L1 to L3, with L1 being the quickest and smallest, cache is vital for system performance enhancement. Its contents are ephemeral, clearing with reboots or when superseded, which ensures the CPU has immediate access to pertinent data. In our homework, I applied an advanced cache replacement strategy to enhance cache performance, benchmarking it against traditional policies such as Least Recently Used (LRU) and Least Frequently Used (LFU). I assessed the new algorithm's effectiveness using SPEC2006 benchmarks to understand how the changes perform with compute intensive applications and PARSEC benchmarks to evaluate its performance on chip-multiprocessor (CMP) architecture.

# **Introduction**

The Least Recently Used (LRU) cache replacement strategy is predicated on the principle of temporal locality, positing that data items accessed in the recent past are more probable to be requested again shortly. To realize this approach, LRU keeps a log of the cache's utilization history, typically employing a data structure similar to a linked list, to monitor the frequency of item usage. Upon the cache reaching its maximum capacity and the necessity to incorporate a new item arises, the item positioned at the end of this log, the one that has remained unused for the longest duration, is selected for removal. The simplicity of LRU's implementation accounts for its widespread adoption across numerous systems. Nonetheless, its dependence on past access trends can be disadvantageous. For example, in scenarios where access patterns are evenly distributed or the active dataset exceeds the cache's size, LRU may induce less than ideal performance due to the recurrent ejection and reloading of cache entries. Moreover, the continual upkeep of the usage log can incur significant computational costs, especially in hardware environments where resources and processing power are limited.

Conversely, the Least Frequently Used (LFU) policy prioritizes the frequency of access over the time of access. LFU maintains a tally of the number of times each cache item is accessed. When the cache capacity necessitates the eviction of an item, the policy targets the item with the minimal frequency count for removal. This method can be exceptionally advantageous for workloads characterized by a limited and consistent set of items that are accessed frequently, as it ensures that such frequently utilized data persists within the cache. Nevertheless, LFU may exhibit a diminished adaptability to shifts in data access patterns. If an item is used a lot for a while but then isn't needed anymore, it could still take up space in the cache because it was used so much before. This means it's using space that could be better used for items that are currently important. This issue of data staleness is further compounded by the 'cold start' dilemma, wherein new items enter the cache with a null frequency count and stand a high risk of eviction before they can prove their potential for frequent access. The system is more complex because it needs a good way to keep track of and change how often data is accessed. This can take up more resources than the simpler method of tracking recent use that LRU (Least Recently Used) uses.

Not Recently Used (NRU) is another cache replacement policy that makes decisions based on a single bit of information to track whether a block of data in the cache has been accessed during a certain period. When the cache is full and a new block needs to be inserted, NRU looks at this bit to determine which block to evict. If the bit is set (indicating the block has not been used recently), that block is considered a candidate for replacement. This approach is straightforward and easy to implement because it doesn't require detailed usage statistics or complex algorithms to decide which data to keep and which to discard.

One of the advantages of NRU is its simplicity and low overhead. It is less resource-intensive compared to more sophisticated algorithms because it does not need to keep detailed records of how frequently or recently each block of data has been accessed. This makes it suitable for systems where memory and processing resources are limited, or where the cost of implementing a more complex algorithm is not justified by the needs of the application.

However, NRU also has significant drawbacks. Its binary approach to tracking access (either recently used or not) can lead to suboptimal cache utilization, especially in systems with diverse access patterns. For instance, if the policy is set to favor keeping recently used blocks, then data that is accessed infrequently but regularly might be unnecessarily evicted. Conversely, if the policy favors block not used recently, it might retain data that is never accessed again at the expense of data that is accessed sporadically but more frequently. This lack of granularity means that NRU might not always make the most efficient use of the cache, leading to a higher rate of cache misses than more nuanced algorithms.

Considering the limitations inherent in the LRU, LFU and NRU policies, a novel approach known as Re-Reference Interval Prediction (RRIP) has been formulated and is explained in detail in the subsequent section.

# **SRRIP technique**

## Static Re-Reference Interval Prediction

The Static Re-Reference Interval Prediction (SRRIP) employs a predictive mechanism for cache replacement, utilizing multiple bits (M-bits) to indicate the predicted re-reference interval of cache blocks, encoded as Re-reference Prediction Values (RRPV). Two approaches are described within SRRIP: *Hit Priority (HP)* and *Frequency Priority (FP).*

SRRIP with Hit Priority (HP) is a cache replacement strategy that employs an algorithm to predict the likelihood of a cache block's re-use. It assigns Re-reference Prediction Values (RRPV) to each cache block, with a numerical value indicating the block's re-reference likelihood. In the HP approach, when a cache block is accessed (a cache hit occurs), the RRPV of that block is reset to zero. This reset indicates the system's prediction that the block will be needed again imminently, effectively prioritizing it to remain in the cache over others with higher RRPV values.

Conversely, The Frequency Priority (FP) model within SRRIP updates a block's RRPV by decrementing it with each cache hit, unless it's already at zero. This method refines re-reference predictions, maintaining blocks in the cache based on how often they're accessed. Frequent accesses lead to lower RRPVs, which signals a sooner expected re-reference, ensuring that blocks in high demand stay in the cache longer. My work for this homework is to implement hit priority of 2 – bit SRRIP algorithm only and hence I shall talk about the HP implementation details in the next subsection.

## Implementation details

The algorithmic implementation for a 2-bit SRRIP-HP is given in [1]. Every block in the cache is initialized with distant re-reference prediction value which will signify that block is a good candidate for eviction. The replacement policy is divided into two parts one if cache is hit and the other is if cache is missed. If cache is hit, then the re-reference value of that block is set to 0 indicating the block might be used in near immediate future. If the cache is missed, then the re-reference value is set to intermediate re-reference value. The algorithmic flow and methods implemented are described in next section.

Computer Science and Engineering, Texas A&M University

Instructor: E. J. Kim,

CSCE 689-700, HW-4 Report, Date: 04/01/2021

# **Methodology**

The zsim simulator was used as required for enacting the replacement policy. This simulator sets up the cache with the specified number of lines. Each replacement policy utilizes pre-insert and post-insert functions to refresh the cache blocks upon each request for memory access. Depending on the criteria specified in the replacement policies, each block is either swapped out or refreshed with the appropriate values.

*Cache Hit:*

When a data block in the cache is accessed (a cache hit), immediately set the Re-reference Prediction Value (RRPV) of that block to '0'. This step marks the block as highly likely to be used again soon, giving it the highest priority to stay in the cache.

*Cache Miss:*

1. Start a search for a block with an RRPV of '3' from the left side of the cache. This represents looking for a block that is predicted to be the least likely to be used again soon.
2. If a block with an RRPV of '3' is found, move to step 5.
3. If no such block is found (meaning no block is marked with an RRPV of '3'), increment the RRPV of all blocks by '1'. This increases their predicted re-reference interval, effectively aging the data as less likely to be used soon.
4. After incrementing RRPVs, go back to step 1 of the Cache Miss process, beginning the search for an RRPV of '3' again.
5. When a block with an RRPV of '3' is found, it is selected for replacement. The incoming block takes its place, and the RRPV of the newly inserted block is set to '2', suggesting it has a moderate likelihood of being needed again soon, but not as immediate as an RRPV of '0'.

Using above methodology, the SRRIP-HP is implemented and compared with LRU and LFU. The details of implementation and evaluation are provided in next section.

# **Evaluation**

Zsim is a C++ based simulator which simulates memory hierarchy of x86-64 architectures. The simulator already consists of LRU and LFU implementations. SRRIP-HP is implemented and tested with PARSEC and SPEC benchmarks. I have evaluated performance based in IPC (instructions per cycle) and MPKI (misses per kilo instruction) to compare performance of each algorithm. IPC measures the number of tasks a computer can do in one go. More is usually better because it means the computer is working at full speed. SRRIP really shines in certain benchmarks because its way of guessing which data will be needed next works well there. But there are few BMs where SRRIP doesn't do as well, which might mean its guessing method doesn't match how that task uses data. MPKI tells us how often the computer reaches for data that it hasn't stored for quick access, which slows things down because it must get it from a slower type of memory. Generally, SRRIP is better at avoiding this slow down compared to LRU and LFU. For example, in tasks that need a lot of computing power like ‘stream cluster,’ SRRIP does a great job of keeping the right data ready to go. But this isn't the same for all tasks. some show only a small benefit, and others don't do as well with SRRIP as they do with LRU or LFU. Let us delve deeper into individual benchmarks results in terms of IPC, Cycles and MPKI in subsequent sections and compare performance.

Total number of cycles in different benchmarks is calculated as:

*total\_cycles = cycles + cCycles*

Figure 1 illustrates the *total cycles* in PARSEC benchmarks for each policy. The simulation is run on Westmere CPU which has 8 cores and total cycles will be sum of above calculation for each core. fewer cycles indicate a policy is more effectively managing the cache to provide quick data access, thereby speeding up the computation.

A graph of different colored lines

Description automatically generated

Fig.1 Total cycles in PARSEC Benchmarks.

All three policies perform similarly in most cases except for *stream cluster* where the cycles for SRRIP is lowest indicating it can effectively predict the block for eviction. LRU, though it performed close to remaining prediction schemes, still took a slightly higher number of cycles to perform tasks. On the other hand, LFU performance is mostly like SRRIP and has been better in some scenarios like *fluidanimate* and *canneal*. Figure 2 gives total cycles for SPEC benchmarks where the performance of SRRIP remained dominant in most of the cases taking lesser number of cycles to complete tasks. LRU remained to have largest number of cycles in all three policies in SPEC benchmarks as well. While LFU outperformed SRRIP in some cases, the difference is too small to be considered. Even in the *mcf***,** SRRIP outperformed LRU and LFU indicating effectiveness.

A graph of different types of cycles

Description automatically generated

Fig. 2 Total Cycles in SPEC Benchmarks.

Next, I have considered the IPCs of different benchmarks in figure 3 and found that it shows greater results for different replacement policies and yet SRRIP is highly better than remaining. This is because of the different workloads in given benchmarks. For example, *stream cluster*has shown drastic improvement with SRRIP at 74.82% better than LRU and 21.84% better than LFU due to the nature of instructions.

A graph with numbers and a bar chart

Description automatically generated

Fig. 3 IPC of different Benchmarks.

SRRIP outperformed LFU in all scenarios but there are a few cases where LRU performed similar and negligibly better than SRRIP which can be found in *sjeng* and *libquantum* benchmarks. This means that the workloads in these benchmarks do not update as fast as predicted by SRRIP.

To further my understanding on how each replacement policies work in terms of cache misses, I have analyzed data for MPKI (Misses per Kilo Instructions) which indicates total misses in 1000 instructions. Since lesser misses indicate better performance, I analyzed data from figure 4 for MPKI. As defined by RRIP algorithm, the number of misses is considerably smaller in SRRIP with respect to LRU or LFU. Notable improvement is observed in *stream cluster* performance with respect to SRRIP is 73.11% better than LRU and 42.01% better than LFU. Similar trend is observed in SPEC benchmarks as well in *Xalan* with 25.17% better performance with respect to LRU and 21.06% better performance with respect to LFU. There are minor degradations for SRRIP when compared with LRU and LFU which is prominently seen in SPEC benchmarks, but the performance degradation is negligibly smaller in SRRIP. Overall, we can see that SRRIP has outperformed in terms of misses making it better approach than other replacement policies.

A graph with numbers and lines

Description automatically generated

Fig. 4 MPKI for Benchmarks.

The MPKI calculation for each benchmark is done as below:

*MPKI = (#total\_misses / #total\_instruction) \* 1000*

To get further understanding, figure 5 indicates the total misses in PARSEC benchmarks and figure 6 indicates the total misses in SPEC benchmarks.

A graph of different colored lines

Description automatically generated

Fig. 5 Misses in PARSEC Benchmarks.

For a given number of instructions, SRRIP has always had lesser number of misses owing to lesser MPKI with respect to other replacement policies.

A graph with different colored bars

Description automatically generated

Fig. 6 Misses in SPEC Benchmarks.

##### **Acknowledgment**

I extend my deepest gratitude to my friends who have contributed to this work through their support and expertise. Special thanks go to Vaibhavi Rastogi (UIN: 834005560), whose insights into SRRIP implementation were invaluable, I thank her for her support in critical analysis and data verification that greatly enhanced the quality of this report. I also thank my TA, Sabuj Laskar for providing prompt responses to my queries. Their willingness to give their time so generously has been very much appreciated. This report has been made possible not only by the direct contributions of my TA and friends but also by the invaluable resources available online.

##### **Conclusions**

The comparative analysis of the LRU, LFU, and SRRIP cache replacement policies across various PARSEC benchmarks has provided valuable insights into their performance characteristics. SRRIP generally demonstrated superior efficiency, due to its advanced nature, achieving the lowest cycle counts in most benchmarks, which underscores its strength in environments where frequency of access is a consistent predictor of data reusability. LRU, with its simplicity and focus on recency rather than frequency, displayed moderate performance, making it a reliable choice for workloads with strong temporal locality. While the data pointed out numerical variations, method of implementing the SRRIP has notably improved its performance. The coding of the algorithm was explored through various methods, with the specifics documented in the ‘*rrip\_repl.h*’ file. Applying the same benchmarks yielded different results for SRRIP, which suggests that the instructions used in the existing algorithm were a factor. Therefore, the careful implementation of the algorithm was key to uncovering and addressing any coding issues effectively.

These findings highlight the importance of tailoring cache replacement strategies to match specific workload behaviors. While SRRIP emerged as the most efficient in given specific set of benchmarks, it is critical to consider the unique access patterns of each application when selecting a cache replacement policy. The observed performance variations also emphasize that more complex or nuanced algorithms like SRRIP do not necessarily guarantee improved performance over simpler ones such as LRU or LFU, and they must be evaluated within the context of their intended operational environment.

The DedUp benchmark is causing buffer overflow and could not be run. I have included snippet of the error message in the zip file with name “DedUp BM Error”.

##### **References**

1. A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10), June 2010.
2. D. Sanchez and C. Kozyrakis, "ZSim: fast and accurate microarchitectural simulation of thousand-core systems," ACM SIGARCH Computer Architecture News, June 2013.
3. Zsim tutorial - <http://zsim.csail.mit.edu/tutorial/slides/core.pdf>
4. Zsim turorial - <http://zsim.csail.mit.edu/tutorial/slides/memory.pdf>