A Comparative Analysis of Cache Replacement Policies: LRU, LFU, and SRRIP

Tarek Bessalah
Computer Science and Engineering
Texas A&M

Abstract—**When a program is running, the CPU constantly retrieves, decodes, and accesses memory, steps known as the instruction cycle. Memory access is slower than CPU speed, creating a significant bottleneck for overall CPU execution time. To address this, caches placed near processing units provide faster access to frequently used data, improving overall efficiency. Effective cache management is essential to optimizing performance, as cache misses introduce additional stalls. A cache miss is costly, so in an ideal scenario, the cache would contain all data values that the CPU will require in the future. Several replacement policies attempt to predict which cache entries are least likely to be accessed next. However, accurately forecasting future data access is inherently difficult, and no replacement policy is perfectly suited to all scenarios.**

This study provides a detailed analysis comparing Static Re-Reference Interval Prediction (RRIP), Least Recently Used (LRU), and Least Frequently Used (LFU) using ZSim simulator.

I. Introduction

Cache eviction algorithms or replacement policies are crucial for improving CPU performance in modern computing. Several cache replacement policies are available, such as LRU and LFU all of which have advantages and disadvantages better suited for certain scenarios.

The LRU cache replacement policy [3] assumes that recently accessed items are more likely to be accessed again, therefore, recently used data items would have a higher priority to be kept in the cache. When the cache reaches capacity, LRU makes space in the cache by selecting a victim that has been least recently used. LRU is well-suited for programs that perform repetitive operations within a certain period that tend to reuse data items. Most associative caches use LRU due to this principle of temporal locality. In associative caches, a block can be stored within any of the subset of cache locations otherwise known as sets. When the set is full, the block to be evicted would be the least recently used block. LRU is not effective when working with data that reoccurs in a pattern that exceeds cache capacity. For example, if a program iterates over a matrix multiplication with matrices larger than the cache size, the LRU keeps replacing rows as it cycles through them. By the time the cycle repeats, the data has already been evicted, which results in cache miss. This would lead to cache pollution. In addition, maintaining access order incurs overhead due to the additional metadata storage and computational recourses required for recording timestamps, which adds further complexity for larger-scale systems.

Computer Science and Engineering, Texas A&M University

On the other hand, LFU assigns a counter to every data item in the cache [4], whenever a data item is referenced it will increment a counter. When a cache reaches capacity, it will victimize data items with the lowest reference frequency. In the previous matrix multiplication example, LFU would retain rows that are accessed repeatedly, even if they haven't been accessed recently in the cycle, while less frequently used data is replaced, making it a more effective replacement policy for this scenario. The key drawback of LFU is that it may retain data items that have been accessed frequently in a short period of time, but have become no longer relevant, whilst evicting newer items that have more ”value”, leading to inefficient cache utilization.

Whilst LRU is suitable for applications with strong temporal locality, and LFU is suitable for applications that have predictable access patterns, a more balanced replacement policy for applications with mixed access patterns is needed for certain applications such as database systems or machine learning. Unlike LRU, which prioritizes recent data access, SRRIP implements the concept of a re-reference interval, which classifies each block in the cache based on its likelihood of being accessed again. This approach is dynamic and reduces unnecessary evictions by predicting which data blocks are likely to be accessed again in the future in the cache. In essence, SRRIP allows us to keep blocks in the cache with higher short-term relevance, this provides scan resistance by minimizing cache pollution from transient data. [1]

II. SRRIP Technique

A. Short description of SRRIP technique

For a more accurate eviction process, SRRIP assigns each block a prediction value, which is 2^M bits instead of being one binary value. For instance, if M=2, a block would have 4 levels of prediction $0 - 2^M$, where a higher prediction value represents a lower likelihood of future access, thus a contender to be victimized. When a new block is introduced into the cache, SSRIP does not assume it will be needed soon so it assigns it $2^M - 2$ value, this also would avoid an early eviction. In case of a cache miss, SRRIP evicts the blocks with the highest RRPV.

B. Implementation details

The SRRIPReplPolicy class implements the SRRIP policy detailed in [1]. Each block maintains an RRPV, giving us a prediction value on how soon the block is expected to be

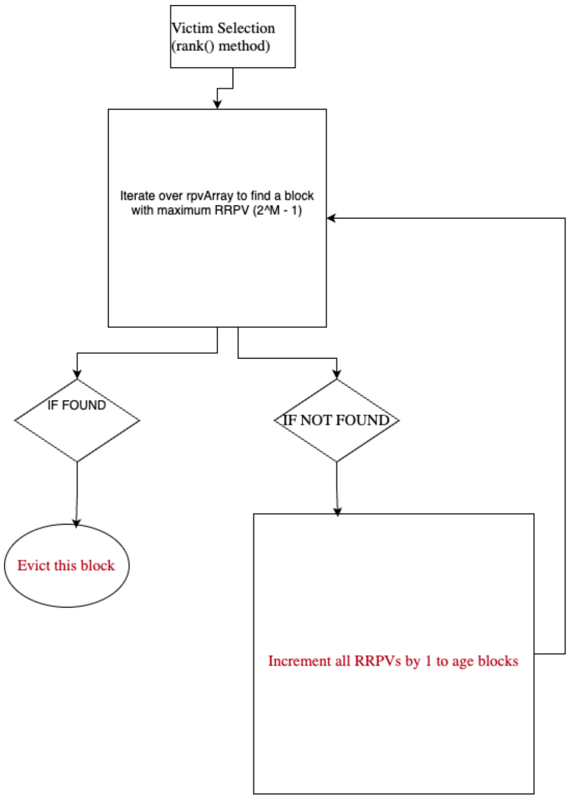
Victim Selection
(rank() method)
Iterate over rpvArray to find a block with maximum RRPV (2^M - 1)
IF FOUND
IF NOT FOUND
Evict this block
Increment all RRPVs by 1 to age blocks

Fig. 1. Victim Selection Process

accessed again. We maintain the array rpvArray which holds the RRPV for each cache line, in which each line is initialized to the maximum value (rpvMax), indicating a low probability of immediate re-access and avoiding new blocks from being victimized. The constant variable rpvMax is $2^M - 1$, where M is the number of bits allocated, in our implementation, rpvMax = 3 which implies that M = 2 ($2^2 - 1 = 3$). We also keep track of the number of cache lines (numLines).

[1] describes the SRRIP's victim selection as a process that "selects the victim block by finding the first block that is predicted to be re-referenced in the distant future (i.e., the block whose RRPV is 2^M − 1)". This victim selection process, represented in Fig. 1, has been implemented using the rank() method, which first iterates over rpvArray looking for a candidate with a maximum RRPV to evict. If no such block is found, all RRPVs are incremented to "age" each block until eventually an eviction candidate is found, this mechanism would remove blocks if they haven't been used recently. As a fallback, if the method still doesn't find a block to evict, it defaults to returning the first candidate in the set.

[1] explains that, when a cache hit occurs, "the RRIP-HP policy predicts that the block receiving a hit will be re-referenced in the near-immediate future and updates the RRPV of the associated block to zero". This means that blocks that have frequent hits should not be evicted. This is implemented via the update() method which assigns an RRPV of 0 to a cache block line whenever it's accessed, giving it the highest priority to remain in the cache.

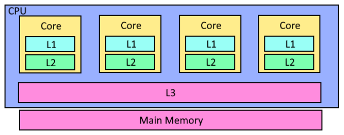

Fig. 2. [5] Multi-core system

To prevent early eviction of useful blocks while avoiding cache pollution, [1] indicates that "RRIP always inserts new blocks with a long re-reference interval", therefore blocks are given time to prove their relevance by assigning 2^M − 2 (rpvMax - 1), slightly below the maximum value in the replaced() method.

III. Methodology

For this study, ZSim, which is a C++ implementation optimized for x86 out-of-order (OOO) multi-core chips, shown in Figure 2, was used to conduct various benchmark simulations across SPEC (Standard Performance Evaluation Corporation) CPU2006 and PARSEC (Princeton Application Repository for Shared-Memory Computers) as shown in Table 1. Zsim is a high-performance simulator that achieves accuracy close to real-world system metrics, achieving 1,500 MIPS for simple cores and 300 MIPS for ooo multi-cores [2]. It has been developed to address the limitations of traditional simulation techniques, which are often too slow multicore systems.

To assess the performance of multi-core processors with applications commonly used in shared-memory systems, PARSEC is a benchmark suite that allows us to simulate workloads that reflect real-world, compute-intensive applications. In this study, the simulations have been run on 8 OOO cores. To measure CPU-intensive performance for single-threaded applications, the SPEC benchmark was used to measure this performance. The simulation has been run on a single OOO core. Simulations gcc, bzip2, mcf are for tasks that are integer calculations, whilst simulations namd, soplex are used for floating-integer calculations.

Both benchmarks have been configured with 32 KB L1 caches and a 256 KB L2 cache. Because PARSEC workloads are multi-threaded a larger L3 memory of 8 MB has been allocated, whilst SPEC uses 2 MB, which is sufficient for single-threaded benchmarks. The L3 cache contains the replacement policies studied, LFU, LFU, and SRRIP using 64-byte line size, 2400 MHz core frequency, and DDR3-1333-CL10 memory. Table II indicates the configurations used for this study. These benchmarks have been run on Texas A&M's Computer Science and Engineering linux.cse.tamu.edu server. This server is equipped with an Intel Xeon Silver 4216 CPUs, with a 2.10 GHz 64 logical CPUs spread across 2 sockets (16 cores per socket, 2 threads per core).

IV. Evaluation

Using the zsim.out files, generated after each simulation has been run, we will inspect the simulation results. The performance of the three replacement policies, LFU, LRU,

SPEC	**PARSEC**
bzip2	blackscholes
gcc	bodytrack
mcf	canneal
hmmer	dedup
sjeng	fluidanimate
libquantum	freqmine
xalan	streamcluster
lbm	swaptions
cactusADM	x264
namd	
soplex	
calculix	

TABLE I
SPEC AND PARSEC BENCHMARKS

and SRRIP is compared based on three metrics: the number of cycles, instructions per cycle (IPC), and misses per thousand instructions (MPKI).

The number of cycles indicate the total time taken for program execution, this metric would allow us to gain insight into how efficiently workload is handled. A lower cycle means that the processors requires fewer cycles to complete the instructions from a given application. Let cycles_i represent the cycle count for a westmere-core i. cCycles_i represents the additional cycle count for core i that stem from cache-related operations, like waiting for cache access or synchronization.

For each core i, the program computes the total cycles as:

$$total\_cycles_i = \text{cycles}_i + \text{cCycles}_i$$

For multi-threaded simulations (PARSEC), we take the maximum value from all cores' total cycles:

$$total\_cycles = \max(total\_cycles_1, total\_cycles_2, \ldots, total\_cycles_n) \quad (1)$$

where n is the number of cores. These results are presented in Figure 3 and Figure 4.

To understand efficiency, IPC measures the amount of instructions completed per cycle. A high IPC means that more instructions are being completed per cycle. Ideally, the pipeline is active with a continuous stream of instructions without any stalls. We calculate the IPC using

$$IPC = total\_instruction/total\_cycles \quad (2)$$

Where

$$total\_instruction = \sum_{i=1}^{n} instrs_i$$

Figure 5 and 6 highlight the results obtained for IPC.

For cache performance, we measured the value of MPKI for L3 across the various simulations. MPKI tells us how often the cache fails to retrieve needed data per 1000 instructions, a lower value of MPKI indicates better efficiency, meaning the cache holds frequently accessed data, whereas the latter indicate a high cache miss. Therefore, reducing MPKI can directly impact IPC. We calculate MPKI using

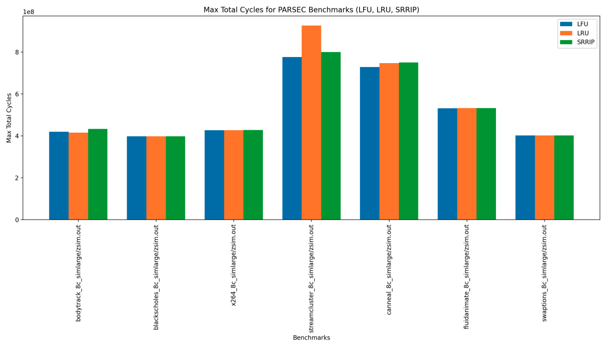

Fig. 3. PARSEC - Max Total Cycles

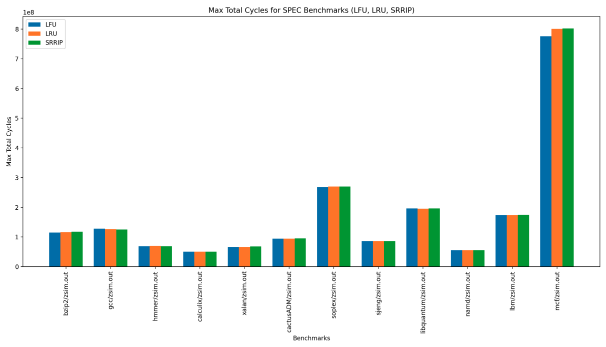

Fig. 4. SPEC - Max Total Cycles

$$MPKI = (total\_misses/total\_instruction) * 1000 \quad (3)$$

where

$$total\_misses = mGETS + mGETXIM + mGETXSM \quad (4)$$

mGETS is a variable that represents the amount of times a processor requests a shared copy of a data block. mGETXIM is the number of exclusive data accesses with permission to modify. mGETXSM represents the amount of times the processor requests exclusive access for modification. L3 cache is shared amongst the multiple cores, therefore we calculate the aggregated MPKI as the total sum of the both the total misses and the total instructions across all cores. Figures 8 and 9 show the results obtained for MPKI values.

ACKNOWLEDGMENT

I would like to thank Weston Cadena and Mohina for their assistance with understanding of this study. The other students on CSE 614 on the discussion portal were helpful.

CONCLUSIONS

SRRIP achieved lower total misses across both SPEC and PARSEC benchmarks indicating that it is a handles various patterns effectively. LRU and LFU have shown higher misses. For the IPC metric, LRU achieved the highest values across both single-threaded and multi-threaded workloads, which

Configuration Parameter	PARSEC (LFU, LRU, SRRIP)	SPEC (LFU, LRU, SRRIP)
Line Size	64 bytes	64 bytes
Frequency	2400 MHz	2400 MHz
Core Type	Out-of-Order (OOO)	Out-of-Order (OOO)
Number of Cores	8	1
L1 Instruction Cache	32 KB, 4-way	32 KB, 4-way
L1 Data Cache	32 KB, 8-way	32 KB, 8-way
L2 Cache	256 KB, 8-way	256 KB, 8-way
L3 Cache Size	8 MB	2 MB
L3 Associativity	16-way	16-way
L3 Hashing Function	H3	H3
L3 Replacement Policy	LFU, LRU, SRRIP	LFU, LRU, SRRIP
L3 Latency	24 cycles	24 cycles
Memory Type	DDR3-1333-CL10	DDR3-1333-CL10
Memory Controllers	4	4
Controller Latency	40 cycles	40 cycles
Phase Length	10,000 cycles	10,000 cycles
Max Instructions	5,000,000,000	500,000,000
Statistics Interval	1000 cycles	1000 cycles
Output Directory	LFU/LRU/SRRIP PARSEC	LFU/LRU/SRRIP SPEC

TABLE II
SYSTEM CONFIGURATION OVERVIEW

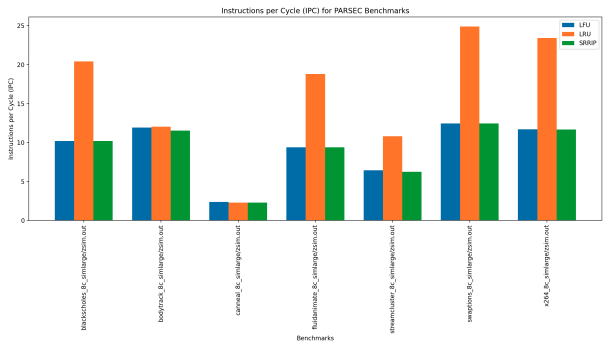

Fig. 5. PARSEC - IPC

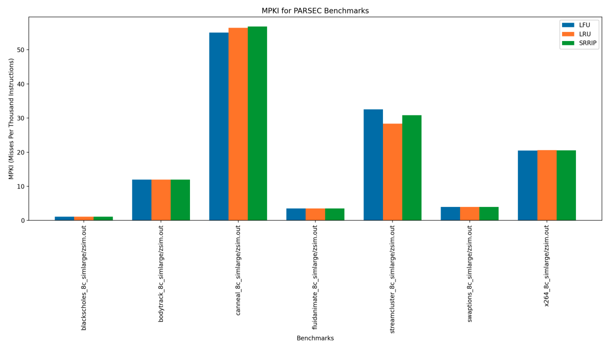

Fig. 7. PARSEC - MPKI

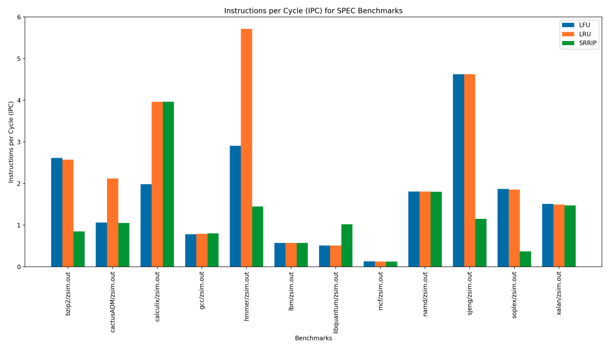

Fig. 6. SPEC - IPC

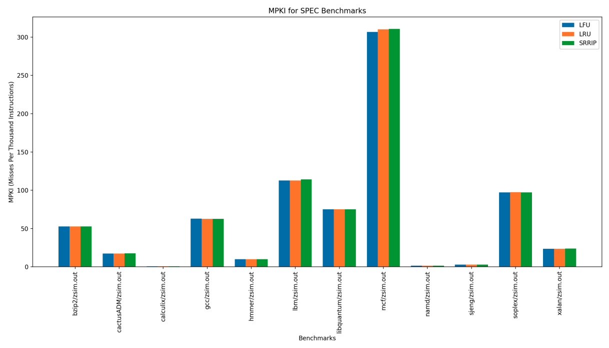

Fig. 8. SPEC - MPKI

means that it kept the pipeline busy. For the MPKI metric, SRRIP has a slightly lower MPKI in PARSEC benchmarks, making it sufficient for multi-threaded applications. In SPEC the differences are less apparent.

REFERENCES

[1] A. Jaleel, K. B. Theobald, S. C. Steely Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," Proc. 37th Annu. Int. Symp. on Computer Architecture (ISCA), Saint-Malo, France, pp. 60–71, June 2010.

[2] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," in Proceedings of the 40th Annual International Symposium on Computer Architecture, Tel-Aviv, Israel, 2013, pp. 475–486.

[3] X. Zhang, J. Liu, B. Li, and T.-S. P. Yum, "CoolStreaming/DONet: A Data-Driven Overlay Network for Efficient Live Media Streaming," in Multimedia Content Delivery, 1st ed., Burlington, MA, USA: Morgan Kaufmann Publishers, 2008, pp. 183-202.

[4] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "LRFU: A Spectrum of Policies that Subsumes the Least Recently Used

and Least Frequently Used Policies," IEEE Transactions on Computers, vol. 50, no. 12, pp. 1352-1361, Dec. 2001.

[5] O. Mendel, "Computer Memory Part 2," Available: https://www.oskarmendel.me/p/computer-memory-part-2. [Accessed: Oct. 30, 2024].