CSCE 614: HW4 REPORT (SRRIP)

Tex Altstaetter

Texas A&M Computer Science & Engineering

*Abstract*—*Write about the replacement techniques you are using in this report in short.*

Cache utilization has a great impact on the performance of a computing system. A cache miss can result in hundreds or even thousands of stalls in a program depending on the clock frequency. In this paper we compare the performance of three different cache replacement algorithms: LFU (Least Frequently Used), LRU (Least Recently Used), and SRRIP (Static Re-Reference Interval Prediction). This paper is an analysis of [1] which asserts that SRRIP has an average miss rate 4-10% lower than LRU.

# **Introduction**

*A short descriptions of each of the replacement policies and their references. Also compare the pros and cons of each of the techniques.*

The LRU replacement policy functions similarly to a first-in-first-out queue. When replacing an entry in the cache, it evicts the entry at the tail and inserts at the head. LRU predicts that more recent entries in the cache will be used in the immediate future and selects the oldest entry in the cache for replacement. The benefits of this replacement scheme is that it is simple to understand and can be implemented with a circular buffer. Despite its simplicity it performs well due to its dependence on temporal locality [2]. The downside is that it is also highly dependent on the cache size. For example, if there is a loop and the cache cannot hold all the instructions contained in that loop then at some point in the loop you will always have one or more cache misses and it could have significant impacts on performance if the loop count is large. In addition, the assumption that an instruction executed will be executed in the near-immediate future is based purely on heuristics and is most often not optimal.

In the LFU replacement policy, there is a counter associated with each entry that indicates how frequently it has been accessed. Upon adding a new entry, it evicts the entry with the lowest frequency [2]. This has some benefits over LRU in that LRU retains more history. An entry can be the least recently used but will be accessed with high frequency. This means that LFU contains more information of temporal locality than does LRU since it depends not only on the sequence of instructions but also their frequency. LFU is more resistant to when there is a mix of new entries in the cache in which infrequent instructions are executed. However, this also biases cache entries toward instructions with temporal locality that might not execute in the immediate future and for which other, more immediate entries are evicted from the cache thus causing cache misses.

SRRIP provides some balance between LFU and LRU since it does not assume that the most recent entry will be re-referenced in the near-immediate future, nor does it weight frequently used instructions as heavily as LFU. SRRIP has more flexibility than LRU and LFU in that you can choose what weight to assign it in the cache upon entry. This prevents new and infrequent entries from polluting the cache [1]. These entries are quickly replaced if they are not re-referenced in the near-future. This allows for other frequently used instructions to remain in the cache as these infrequent instructions are interspersed in the program execution. The algorithm implemented here is static which means that the value that is assigned to an entry is the same for any instruction, which is in contrast to Dynamic RRIP (DRRIP).

# **SRRIP technique**

## Short description of SRRIP technique

SRRIP provides more information by using M bits with a re-reference prediction value (RRPV) from zero up to 2M-1. Initially, when the cache is empty all entries are assigned a value of 2M-1. It then scans the cache for a match. If there is a cache hit, it sets the RRPV value of the cache hit to 0 indicating that it expects to be re-referenced soon. If it is a cache miss then it scans the block for a cache entry that has an RRPV value of 2M-1. It evicts that entry and places the new instruction in that location and assigns it an RRPV of 2M-2 indicating a distant re-reference prediction. However, if there is no entry of value 2M-1, then it increments the RRPV value of all entries by one and iteratively rescans until a value 2M-1 is found. If there are multiple entries with RRPV value 2M-1, it replaces the first entry it finds in a “Greedy” way.

## Implementation details

Zsim was used to simulate, analyze, and test the SRRIP implementation on various benchmarks. This allowed an easy comparison among LRU and LFU cache replacement algorithms which were already implemented and used for performance comparison against SRRIP. The two inputs passed into SRRIP are the cache size and the max RRPV value. The cache size is used to allocate two arrays, one for the instruction and the other to indicate the RRPV value corresponding to that instruction’s entry. In initialization, the instruction array is cleared and the RRPV array is set to the max RRPV value indicating that all entries are replaceable which simulates an empty cache.

There are three functions that need to be implemented: update(), rank(), and replaced(). First, a lookup is done on the current instruction to see if it exists in the cache. If it’s a cache hit then update() is called. Update() modifies the RRPV and returns. The instruction executes then fetches the next instruction to lookup. Otherwise, if lookup returns a miss then rank() is called which selects a victim in the cache to be evicted. After a victim is found replaced() removes the current entry and its RRPV value from the cache and sets a boolean indicating that this instruction is a new entry. In postinsert(), the calling function of replaced(), the new instruction is inserted in the instruction array. However, update() is called on both a cache hit or a miss which is why the boolean is needed to know whether the RRPV should be set to zero (update called on a cache hit) or 2M-1 (update called on a cache miss). It then executes the current instruction and is ready to fetch the next instruction.

Upon completion, the deconstructor is used to de-allocate the memory used for the instruction and RRPV value arrays. To see the source code for SRRIP implementation please see rrip\_repl.h.

Data flow for a cache hit:

fetch => lookup => update => done

Data flow for a cache miss:

fetch => lookup => preinsert => rank => postinsert => replaced => update => done

**Methodology**

To verify the results we ran twenty-three different benchmarks (7 integer, 7 floating-point, 9 multi-threaded) on each of the three different replacement policies (LRU, LFU, SRRIP). These benchmark are the SPEC CPU2006 and PARSEC.

SPEC CPU2006 Benchmarks

|  |  |
| --- | --- |
| Integer | Floating-Point |
| Bzip2 | Milc |
| Gcc | cactusADM |
| Mcf | Leslie3d |
| Hmmer | Namd |
| Sjeng | Soplex |
| Libquantum | Calculix |
| xalan | Lbm |

PARSEC Benchmarks

|  |
| --- |
| Blackscholes |
| Bodytrack |
| Canneal |
| Dedup |
| Fluidanimate |
| Freqmine |
| Streamcluster |
| Swaptions |
| x264 |

The configuration for the benchmarks were identical except for the cache replacement policy and the L3 cache memory size between the SPEC and PARSEC benchmarks. One reason the L3 cache is larger for the PARSEC benchmarks is that the cache is shared among all cores and a larger cache prevents one core from consuming too many resources and slowing the cache accesses of other cores. The configurations are similar to what would be seen on a computer in the consumer market.

SPEC CPU2006 Configuration

|  |  |
| --- | --- |
| Memory Page Size | * 8 KB |
| Threads | * 1 |
| L1-Instruction | * 4-way * Set Associative * LRU * 32 KB |
| L1-Data | * 8-way * Set Associative * LRU * 32 KB |
| L2 | * 8-way * Set Associative * LRU * 256 KB |
| L3 | * 16-way * Set Associative * LRU, LRU, SRRIP (RPV\_Max = 3) * 2 MB |

PARSEC Configuration

|  |  |
| --- | --- |
| Memory Page Size | * 8 KB |
| Threads | * 8 |
| L1-Instruction | * 4-way * Set Associative * LRU * 32 KB |
| L1-Data | * 8-way * Set Associative * LRU * 32 KB |
| L2 | * 8-way * Set Associative * LRU * 256 KB |
| L3 | * 16-way * Set Associative * LRU, LRU, SRRIP (RPV\_Max = 3) * 8 MB |

The SPEC benchmarks run for 100 million instructions and the PARSEC benchmarks run the whole parallel phase. Running a test suite of various benchmark types gives a broad perspective on the performance and limitations of each respective replacement policy. As a result, we are able to generalize and make recommendations based on the findings. These benchmarks were selected because it allows for direct comparison to [1], to verify reproducibility. These tests were run on the Texas A&M CSE Linux servers and the results for each test written to an output file, zsim.out for further analysis. These tests were run from existing scripts and therefore the test framework was already in place and was not modified except for implementing the SRRIP replacement algorithm.

It is concluded that SRRIP gives no benefit over LRU to L1-I, L1-D, or L2 caches due the limited size of the cache. This is why LRU was used as default for those caches.

Problems and challenges that were anticipated were that there is little way to know if there is a bug in the code unless it is obvious among all the tests. The best metric is to compare to existing LRU and LFU results and see if they were in range based on Figure 5 from [1]. In addition, the framework for setting up zsim is complex and careful attention is required when integrating SRRIP into the existing interface. Also, the benchmarks require a non-trivial time to complete and sufficient computing resources. Given the size of the class and limited TAMU CSE servers some tests errored out and needed to be restarted when computing loads were less heavily utilized.

My response to these challenges were to create bash scripts that could run each successive test and to use the bash command “screen” which allows running commands in the background after logging out of an ssh session. This allowed me to free up my computer for long-running test and not have to wait an unknown time for tests to complete. I also wrote post processing scripts to parse the output file and more easily analyze the data. These challenges do not in any way impact the results and analysis of my findings.

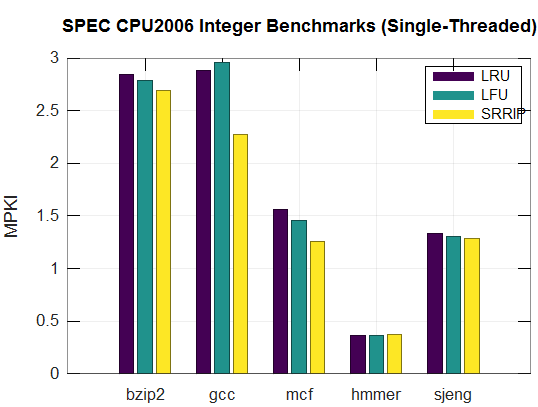
How was the data collected or generated? And, how was it analyzed? The writing should be direct and precise and always written in the past tense.

**You must explain how you obtained and analyzed your results for the following reasons:**

* Readers need to know how the data was obtained because the method you chose affects the results and, by extension, how you interpreted their significance.
* In most cases, there are a variety of different methods you can choose to investigate a research problem. The methodology section of your paper should clearly articulate the reasons why you chose a particular procedure or technique.
* The methodology should discuss the problems that were anticipated and the steps you took to prevent them from occurring. For any problems that do arise, you must describe the ways in which they were minimized or why these problems do not impact in any meaningful way your interpretation of the findings.

# **Evaluation**

In the evaluation section you have to compare the replacement techniques quantitatively. First produce the results. You have to compare three things about the program execution.

1. Number of cycles
2. IPC
3. MPKI

Draw graph (bar chart) for each of these for all the benchmarks across the techniques. Y-axis must be these parameters (#cycles, IPC, MPKI, etc.), and X-axis must be the benchmarks. For each of the benchmarks there should be “replacement technique” number of bars.

Q. How to calculate number of cycles?

*total\_cycles = cycles + cCycles*

For multi-threaded simulations (PARSEC) you have to add across all the processors (westmere-0 to westmere-7). The same thing is valid for all the calculations.

Q. How to calculate IPC?

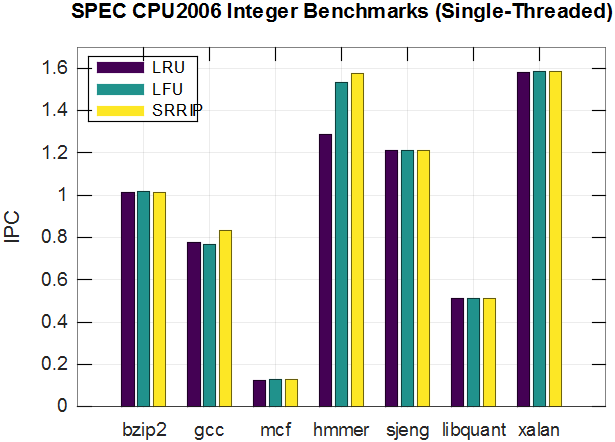
*IPC = #total\_instruction / #total\_cycles*

Q. How to calculate MPKI for L3?

t*otal\_misses = mGETS + mGETXIM + mGETXSM*

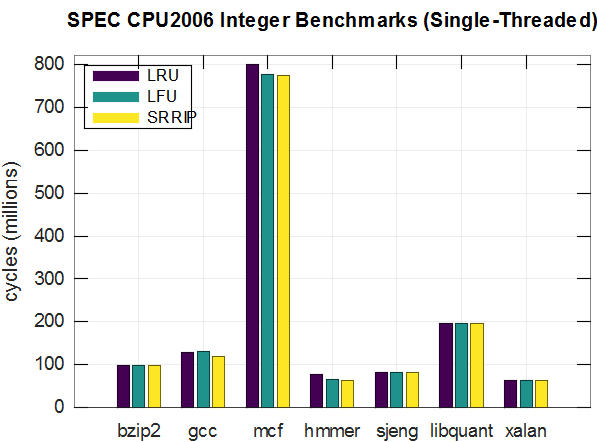
*MPKI = (#total\_misses / #total\_instruction) \* 1000*

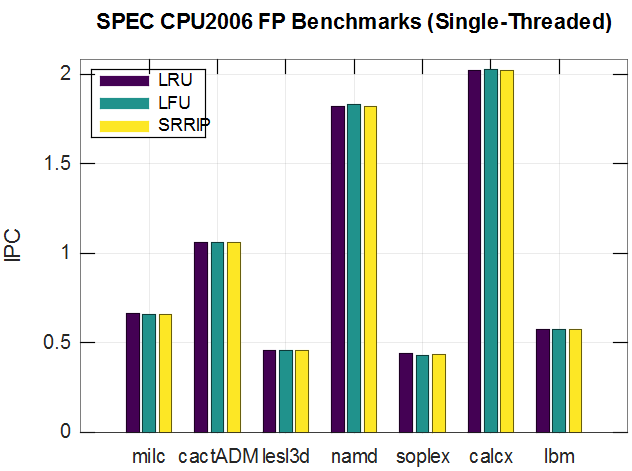
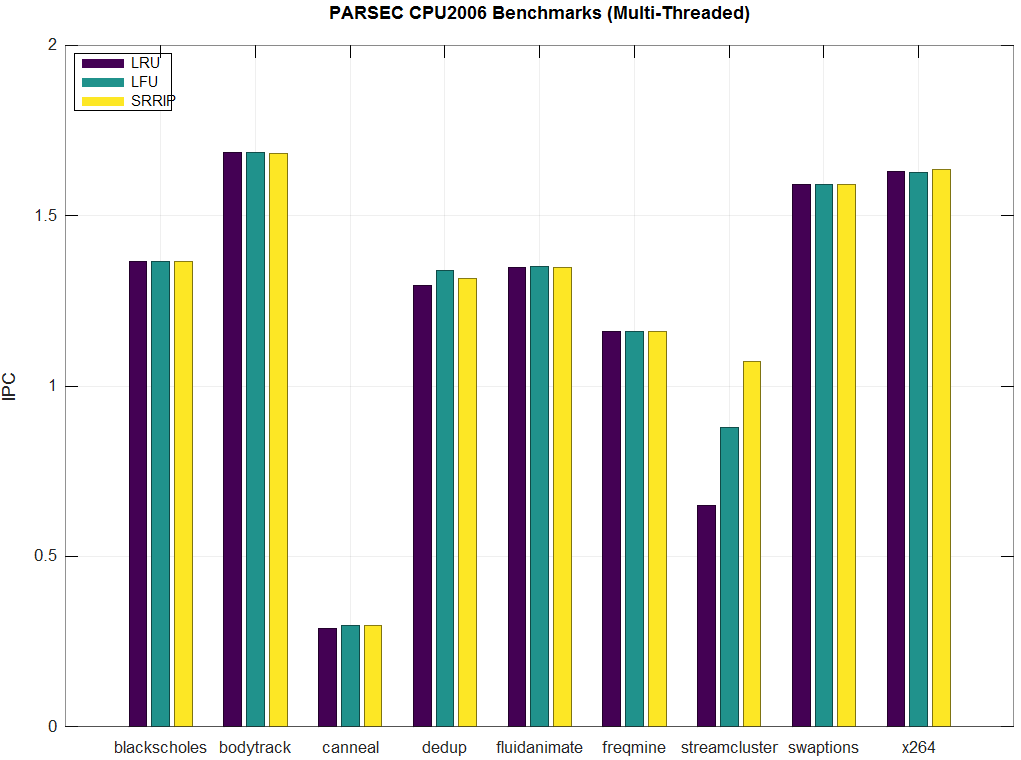
L3 is configured as shared and distributed across multiple cores. So you have sum them up for getting the total number of misses, for multi-threaded workloads (PARSEC).

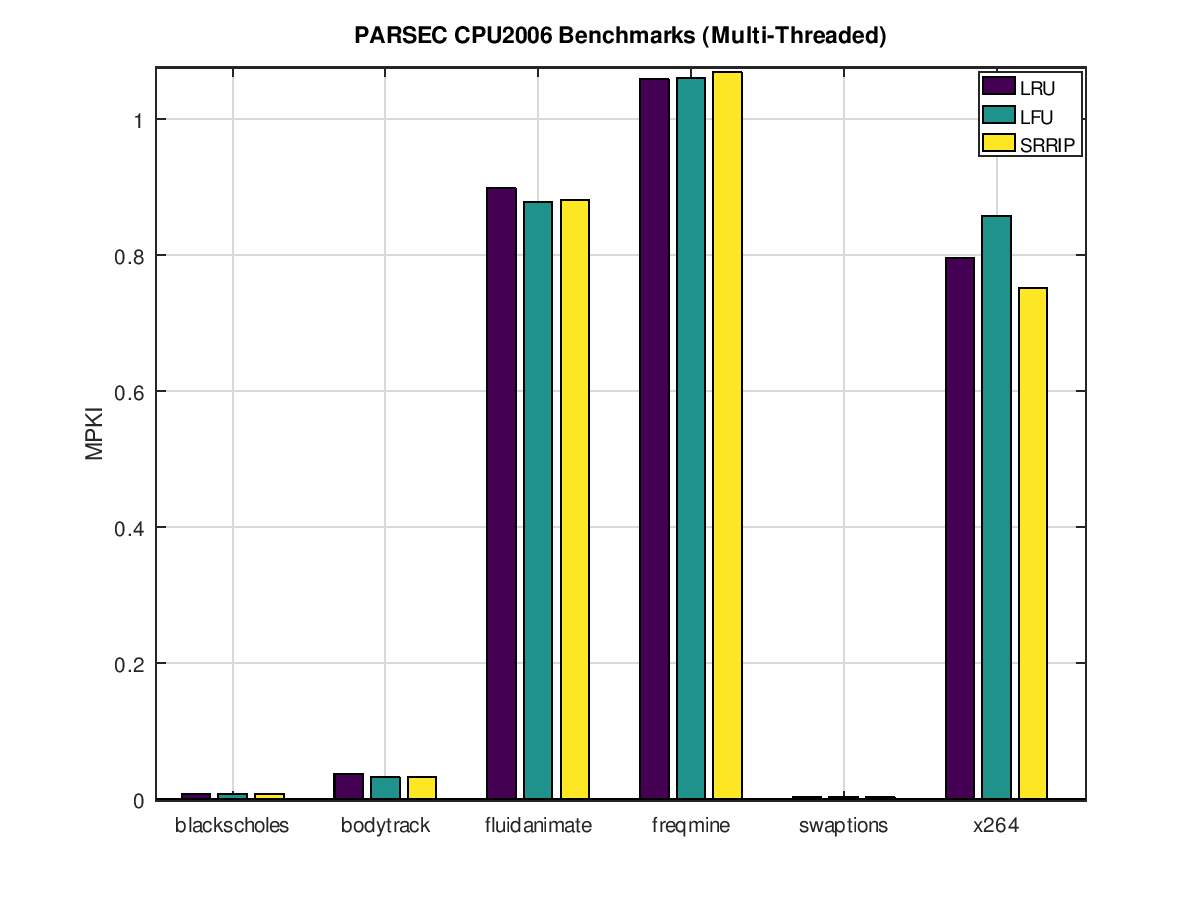


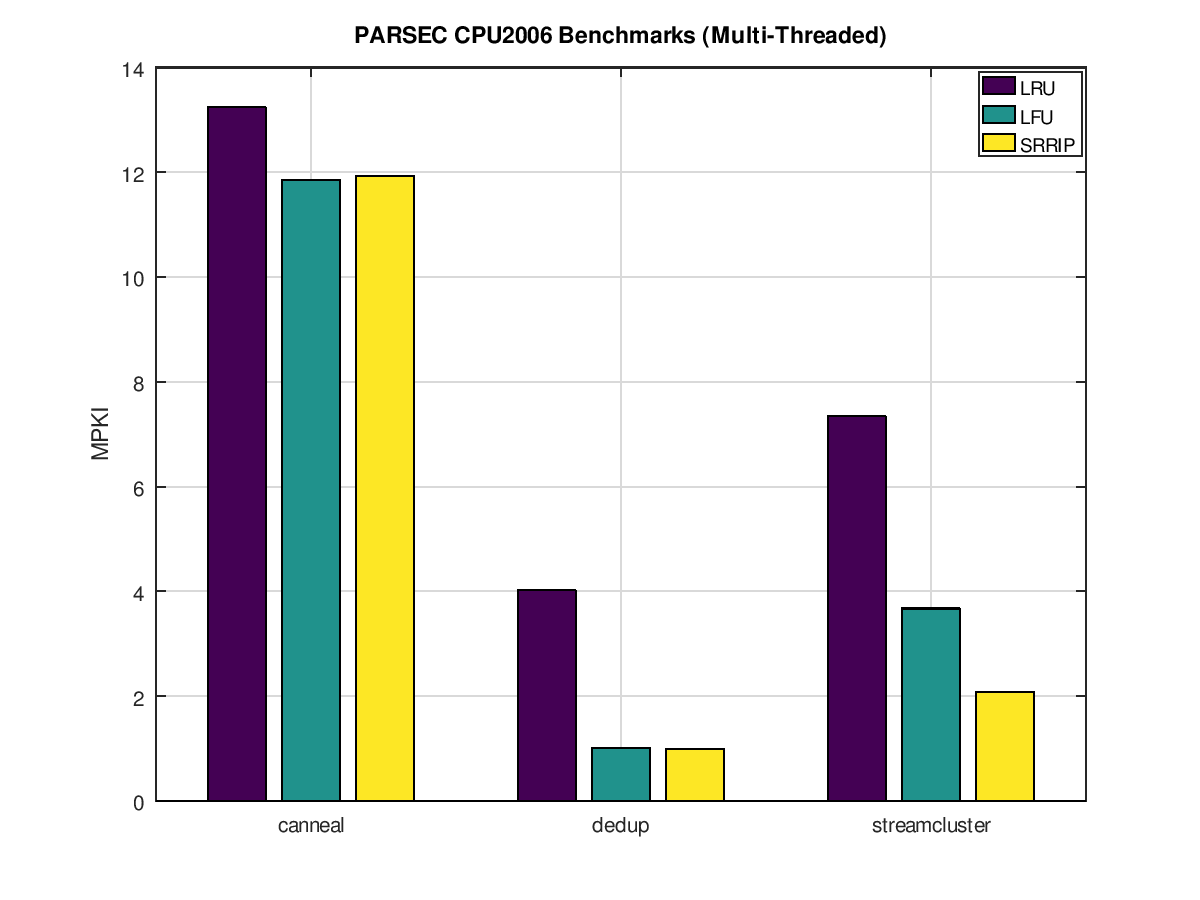
##### 

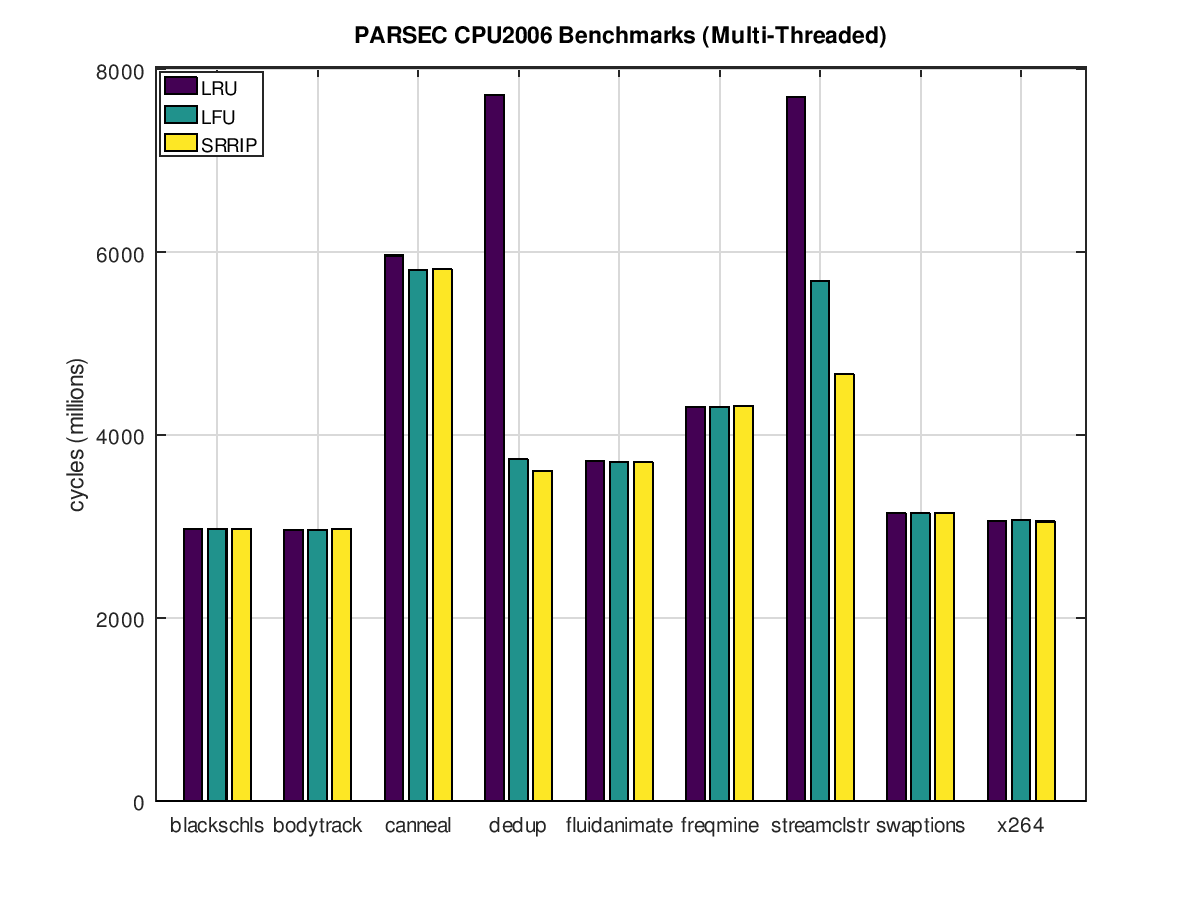
##### 











we ran SRRIP-HP set to zero after a rereference

SRRIP-FP decrements a re-ref instead of predicting

its a near-immed re-ref.

the PARSEC sims have a L3 cache that is 4X larger

8MB vs 2MB

We executed our sims with M=2 and the

insertion value = (2^M)-1 which actually

has the worst performance of the the SRRIP

configurations due to it quickly aging out

It is shown (as indicated in the paper) that

a long re-ref prediction has the best perf

long re-ref t >= 2^(M-1), whereas distant

re-ref t = (2^M)-2

we only evaluated at the L3 cache bc:

6.6. RRIP at Different Cache Levels

At the L1 cache,

SRRIP provides no opportunity to improve performance because the

cache size is too small and the temporal locality is too high. At the L2

cache, SRRIP provides no significant performance gains because the

L2 cache is small (256KB in our study). SRRIP did not degrade

performance of the L1 or L2 caches. To ensure that SRRIP performs

well at the LLC, we modified our hierarchy from a 3-level to a 2-

level hierarchy by removing the L2 cache. For this 2-level hierarchy,

both SRRIP and DRRIP outperform LRU by 4.8% and 10%

respectively. Thus, RRIP is most applicable at the LLC where the

temporal locality is filtered by smaller levels of the hierarchy7.

SRRIP works better when there is not as robust to temporal locality

due to its small cache size. WHen used in the larger L3 cache however,

it does perform relatively better by comparison. The tradeoff is that

scanning the cache for a hit takes longer for more entries, but it

is a low overhead. It is not affected considerably on an insertion bc

an it can replace any element that is of age to be replaced and finds

the first match and is "greedy" in selecting its victim.

8 KB Memory Page Size

The L1-I:

4-way Set Associative

LRU

32 KB

L1-D:

8-way Set Associative

LRU

32 KB

L2:

8-way Set Associative

LRU

256 KB

L3:

16-way Set Associative

SRRIP

RPV\_Max = 3

2 MB

##### **Acknowledgment**

Write about the discussions you have done with your friends (mention names), and also acknowledge their contributions. Also remember it is individual project, where you are allowed to discuss the concepts, but NOT allowed to share your implementations.

##### **Conclusions**

Your conclusion about the techniques.

##### **References**

The template will number citations consecutively within brackets [1]. The sentence punctuation follows the bracket [2]. Refer simply to the reference number, as in [3]—do not use “Ref. [3]” or “reference [3]” except at the beginning of a sentence: “Reference [3] was the first …

1. A. Jaleel, K. Theobald, S. Steely Jr. and J. Emer, “High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)” ISCA 2010 Proceedings of the International Symposium on Computer Architecture, vol. 37, pp. 60-71, June 2010.
2. H. Al-Zoubi, A. Milenkovic, M. Milenkovic “Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite.” In ACMSE, 2004.

**IEEE conference templates contain guidance text for composing and formatting conference papers. Please ensure that all template text is removed from your conference paper prior to submission to the conference. Failure to**

**remove template text from your paper/report may result in your report not being accepted.**