CSCE 614: HW4 REPORT (SRRIP)

Tex Altstaetter

Texas A&M Computer Science & Engineering

***Abstract*—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.**

I. INTRODUCTION

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 it provides a more careful weighting for new entries and doesn't allow them to pollute the cache if it's the first time it is added to 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).

II. SRRIP TECHNIQUE

A. Short description of SRRIP technique

SRRIP provides more information by using an M bits re-reference prediction value (RRPV) from zero to 2^M-1. Initially, when the cache is empty all entries are assigned a value of 2^M-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 2^M-1. It evicts that entry and places the new instruction in that location assigning it an RRPV value of 2^M-2 indicating a distant re-reference prediction. However, if there is no entry of value 2^M-1, then it increments the RRPV value of all entries by one and iteratively rescans the cache until an entry with RRPV value 2^M-1 is found. If there are multiple entries with RRPV value 2^M-1, it replaces the first entry it finds in a "Greedy" manner.

B. Implementation details

When implementing SRRIP it requires two inputs: the cache size and the max RRPV value. Additionally, the cache insertion value is another parameter but was fixed at 2^M-2 for this implementation. The cache size is used to allocate two arrays, one to store the instruction cache and the other to store the RRPV values 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 on program start which simulates an empty cache.

There are three key functions that need to be implemented. The first is updating the cache, which sets the entry's RRPV value to zero on a cache hit, while a cache miss sets the new cache entry's value to 2^M-2. If a cache hit occurs no more processing is needed and the next instruction can be fetched. On a cache miss, the first entry in the cache found with the max RRPV value is replaced. If no entry is found all RRPV elements increment until a candidate is found. Finally, the old instruction is evicted and the new entry is inserted in its place. To see the source code for the 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 => find candidate => replace => update => done

Methodology

Zsim was used to simulate, analyze, and test the SRRIP implementation on various benchmarks which is a C++ based simulator created out of Stanford University [3]. Zsim is described as a fast and scalable x86-64 multicore simulator and when compared to a real system the IPC is within 10% in 18 out of 29 benchmarks [3]. When comparing MPKI (Misses Per Thousand Instructions) of the simulated and real system the percent error is 0.3% for the L3 cache [3]. When evaluating performance zsim is biased towards overestimation which is a result of a more simplified memory model and features that zsim currently lacks [3]. When comparing the simulation instruction decoding to a real system the absolute error rate is 1.3% [3]. Given these error rates, it provides confidence about how results correlate with a real system. Therefore, zsim allows for an easy comparison of SRRIP against LRU and LFU cache replacement algorithms within the average error specified above.

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). The test suites selected are the SPEC CPU2006 and PARSEC, two widely used and well-known test suites.

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.

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 of 5 billion instructions. 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 verify reproducibility SRRIP. 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 until can be compared among all the tests. The best way to tell is to compare to the existing LRU and LFU results and see if they are 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. There were issues with the compatibility of the library dependencies and versions of software packages. Also, despite zsim being 2-3 times faster than other simulators the benchmarks require a non-trivial time to complete and sufficient computing resources [3]. 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.

III. Evaluation

Evaluation of the three replacement techniques was done by quantitative analysis of three key metrics:

1. Number of cycles
2. IPC
3. MPKI

For the single-threaded SPEC CPU2006 benchmark the cycles were calculated as:

$$total\_cycles = cycles + cCycles$$

Where cycles are the number of simulated unhalted cycles and cCycles are the number of cycles spent in contention stalls.

For multi-threaded PARSEC simulations, the cycles are being executed in parallel across eight different cores. Therefore, the total cycles are calculated as follows:

$$cycles\_core1 = cycles1 + cCycles1$$
$$cycles\_core2 = cycles2 + cCycles2$$
$$\ldots$$
$$cycles\_core8 = cycles8 + cCycles8$$

$$total\_cycles = max(cycles\_core1, cycles\_core2, \ldots, cycles\_core8)$$

The max among each core is taken because the core that executes the greatest number of cycles defines the critical path.

Calculating the Instructions per Cycle (IPC) is as follows:

$$IPC = total\_instruction / total\_cycles$$

Calculating Misses per Thousand Instructions (MPKI) is defined as:

$$total\_misses = mGETS + mGETXIM + mGETXSM$$

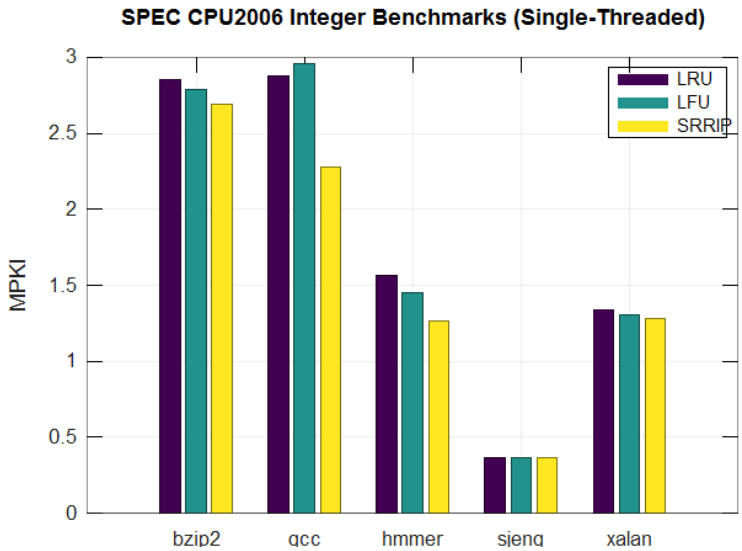
$$MPKI = (total\_misses / total\_instruction) * 1000$$

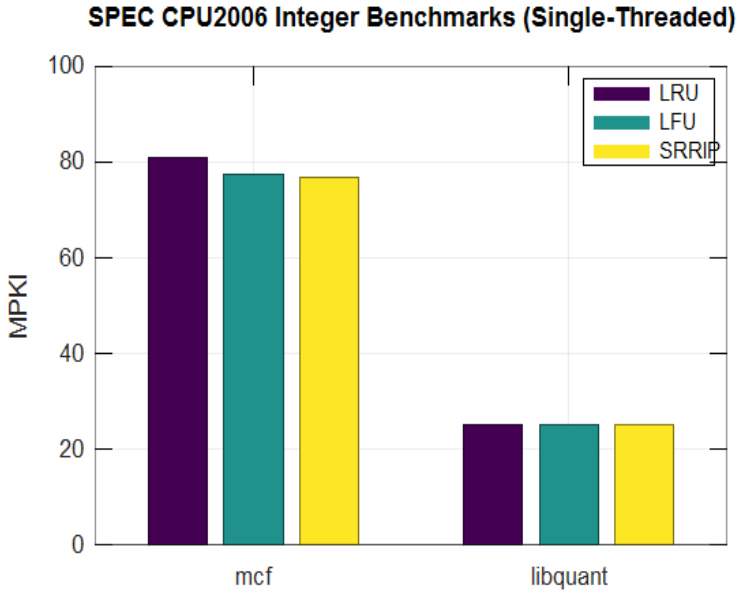
Where mGETS are the GETS misses, mGETXIM are the GETX I->M misses, and mGETXSM are the GETX S->M misses.

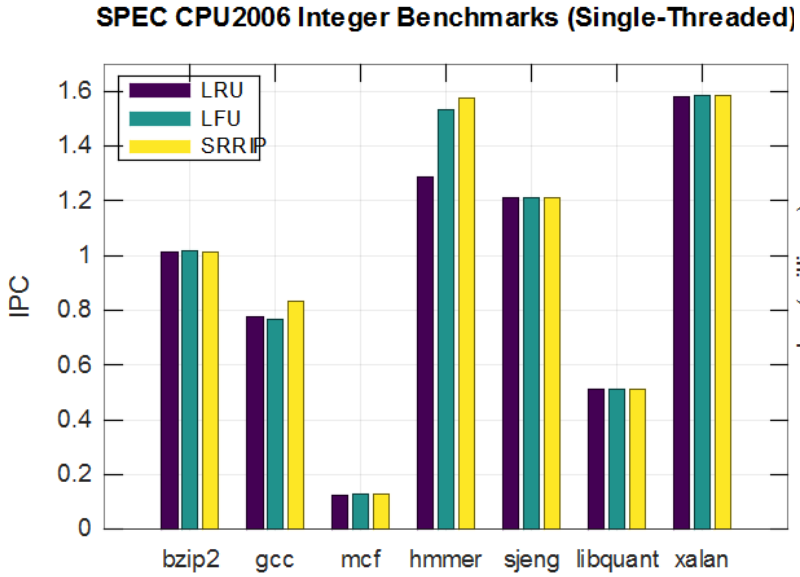
L3 is configured as shared and distributed across multiple cores. For the multithreaded simulations the total misses and total instructions needs to be summed across all the cores to correctly calculate MPKI.

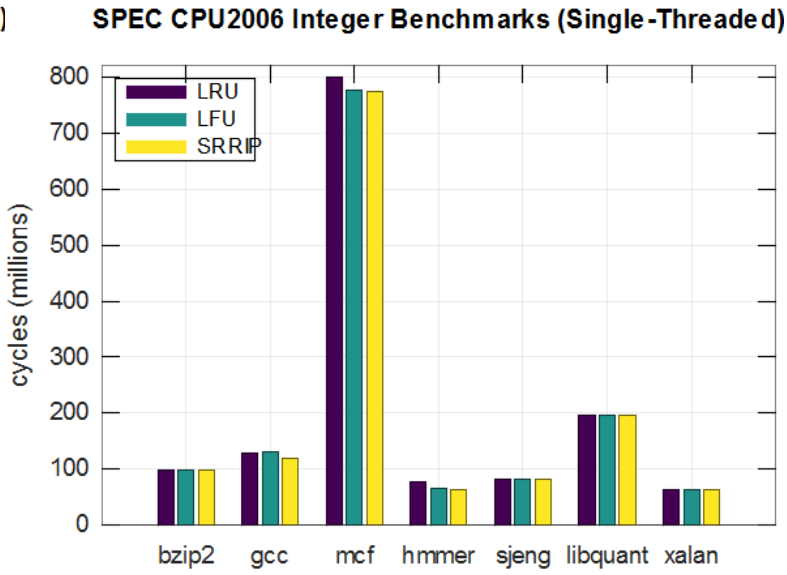
The bar graphs of the performance is shown below. Where possible similar tests are on a single graph. If the dynamic range of too great those graphs are shown on a separate plot. This is to allow for proper precision to be easily seen when viewing the data. The format has been sacrificed for clarity.

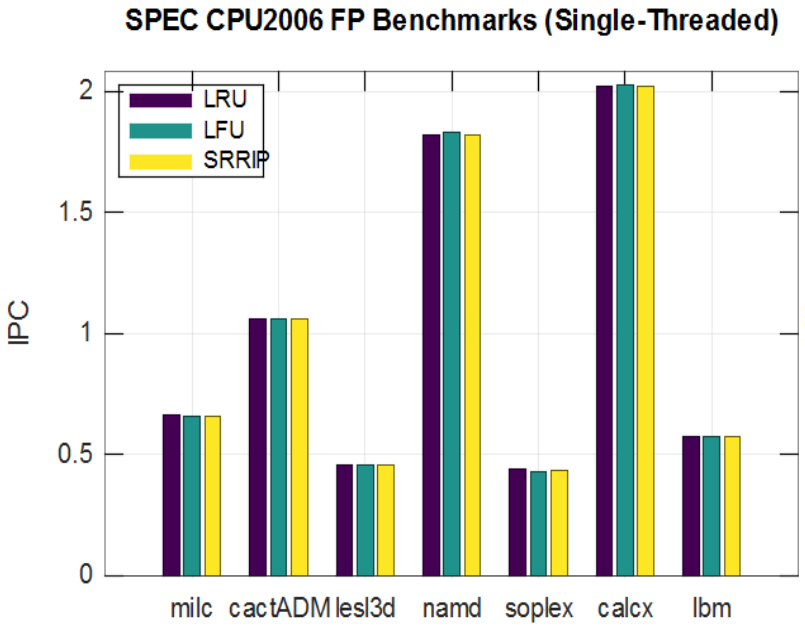
There is a distinct relationship between the performance metrics plotted for the benchmarks. The IPC and the total cycle count are inversely related. Therefore, for benchmarks with a high cycle count we expect them to have a low IPC to be low, relatively speaking. For example, the mcf benchmark illustrates this point well. In addition, higher miss rates are directly proportional to cycle count. Despite the miss

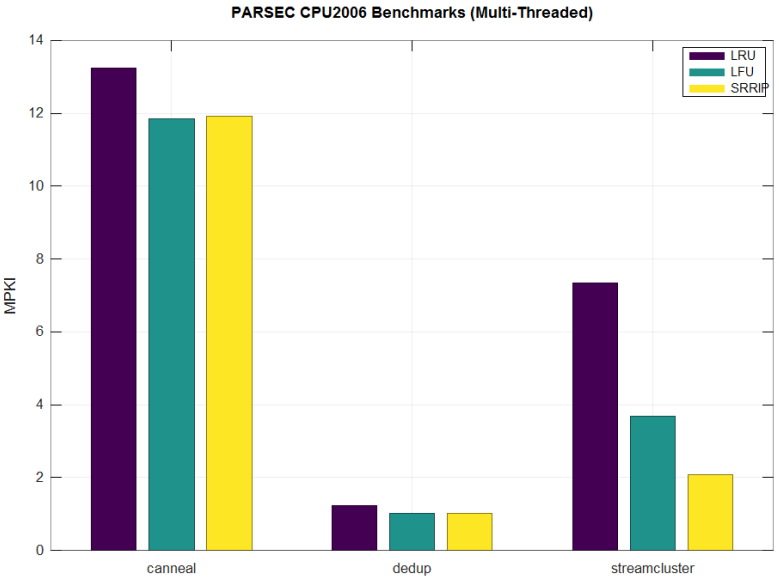


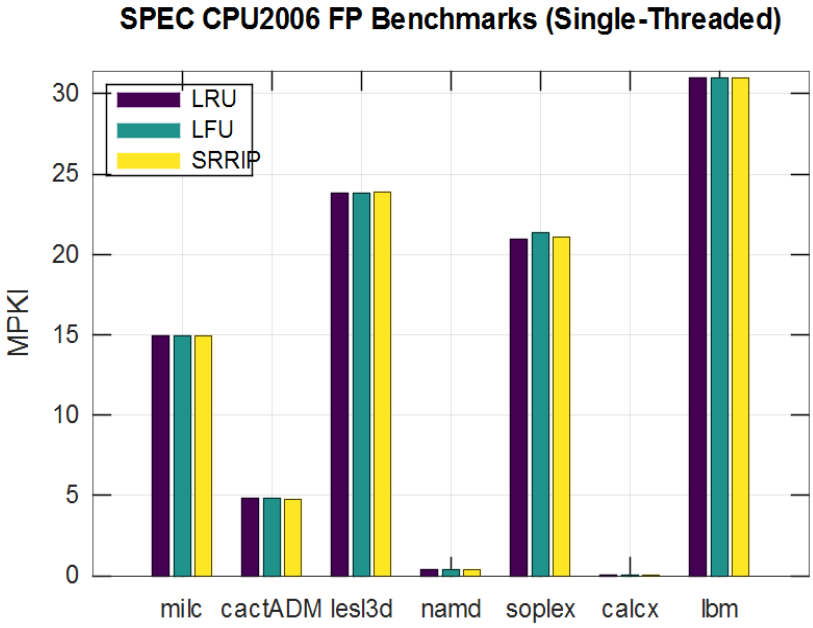


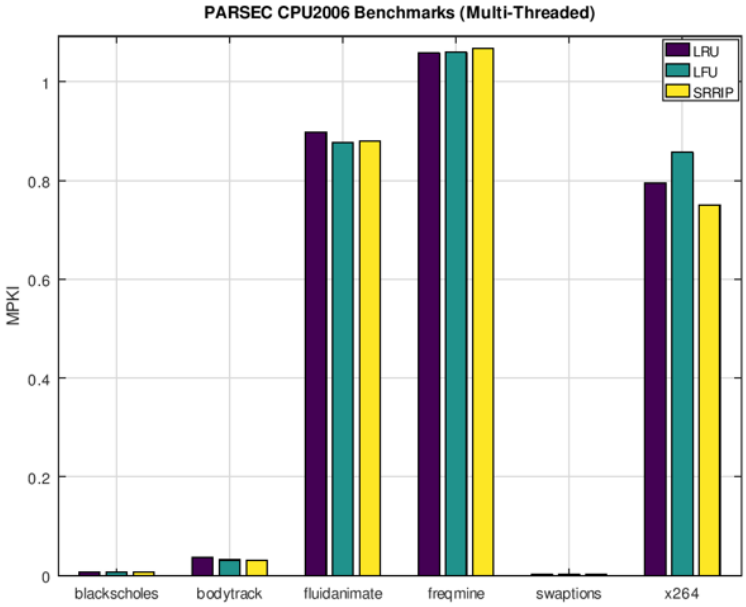


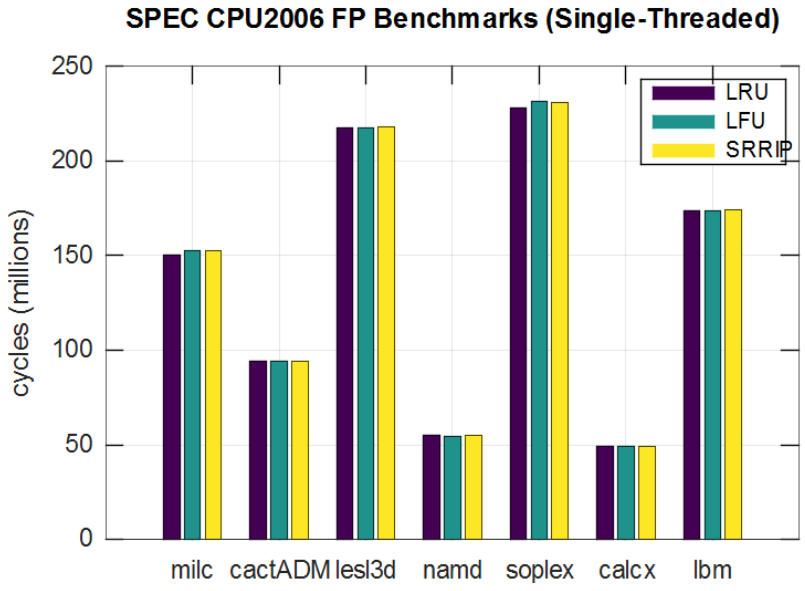




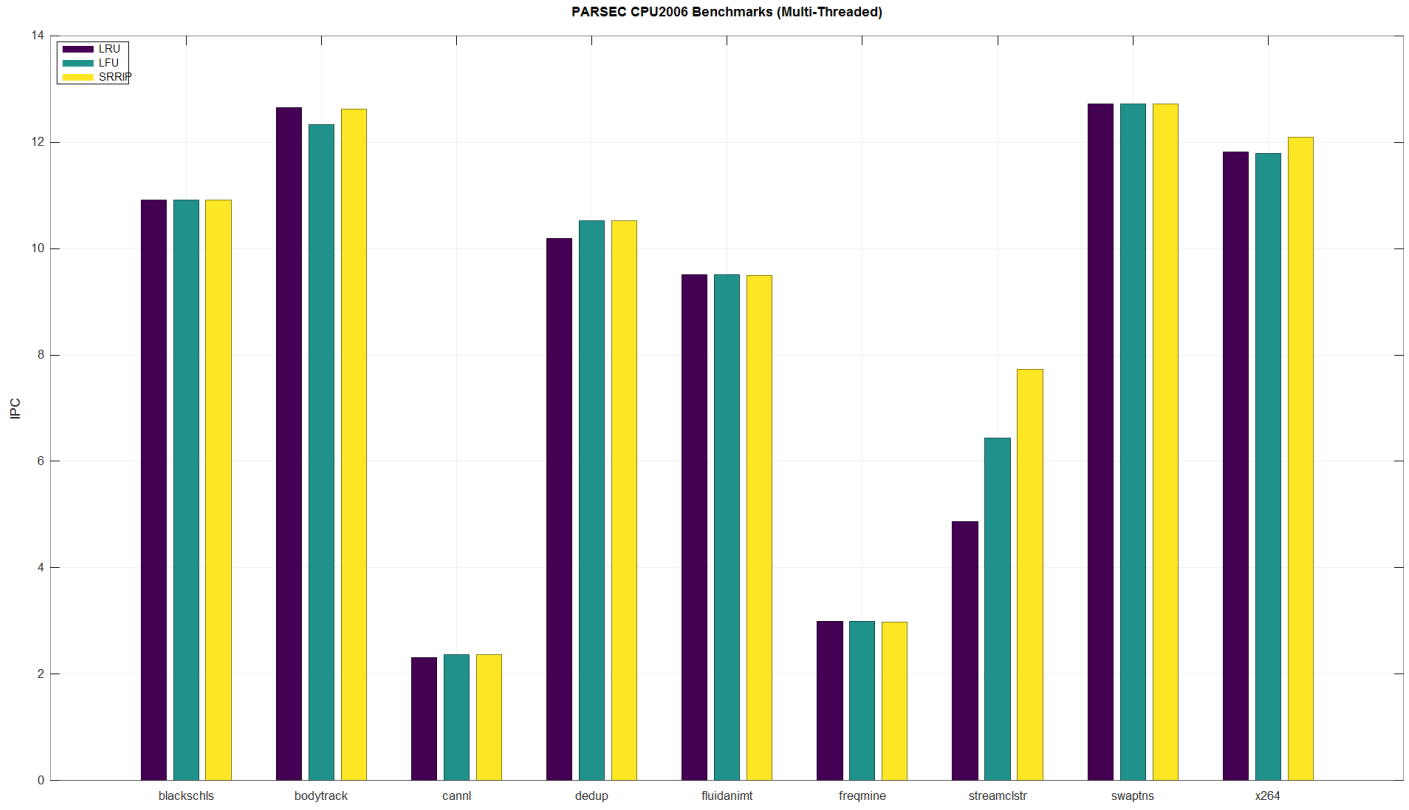


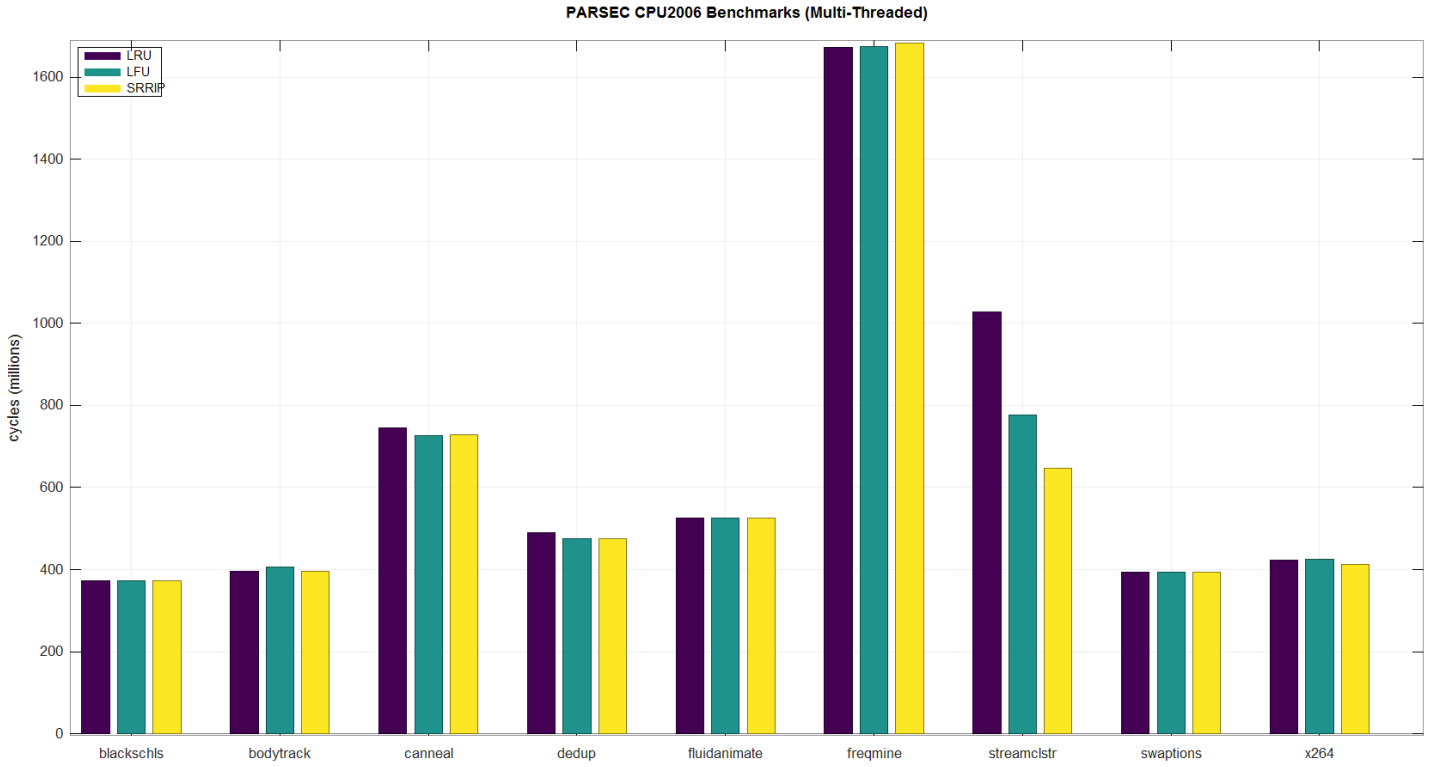






rates being relatively low it can be seen that cache misses have a significant impact on the total cycle count of the benchmark. This quality is seen very clearly by benchmarks cactusADM, namd, and calculix which have low miss rates. For example between cactusADM and milc, the MPKI difference is 10 %, however, the total cycle count is more than 50% greater for milc relative to cactusADM. Conversely, lbm has twice the MPKI miss rate of milc but is only 15% more cycles. To conclude, miss rates are directly proportional to cycle count but IPC greater correlates to total performance since the overall efficiency of instructions in aggregate is more important that cache misses even when occurring at a high rate. It should be noted that for shorter simulations this principle does not hold due to the short viewing window of the simulation. This analysis holds for both integer and


PARSEC CPU2006 Benchmarks (Multi-Threaded)
LRU
LFU
SRRIP
IPC
0
2
4
6
8
10
12
14
blackschls
bodytrack
cannl
dedup
fluidanimt
freqmine
streamclstr
swaptns
x264


PARSEC CPU2006 Benchmarks (Multi-Threaded)
LRU
LFU
SRRIP
cycles (millions)
0
200
400
600
800
1000
1200
1400
1600
blackschls
bodytrack
canneal
dedup
fluidanimate
freqmine
streamclstr
swaptions
x264

floating point simulations. For the PARSEC multithreaded simulations, patterns are more recognizable. In the multicore simulations, the IPC is less of a determining factor of performance. This is because there needs to be a balanced loading of the cores in order to get the most efficient CPI, so a good load balance algorithm is as important as the cache replacement policy. The effect of scheduling the cores was not analyzed. The IPC of the multicore simulations is in the expected range when you scale the single-threaded processes by eight threads indicating that the benchmarks run instructions that are relatively independent between cores. Therefore, the increase in IPC is a direct result of a superscalar architecture.

As far as the relationship between the individual cache replacement policies, LRU, LFU, and SRRIP it can be seen, on the aggregate, that SRRIP has an IPC that is as good as or better than LRU and/or LFU. LFU and SRRIP are much closer in performance than LRU, this is due to a more balanced weighting for cache entries as opposed to LRU which gives equal weighting to each new entry. This difference is most profound in the streamcluster benchmark. This could be attributed to a less consistent instruction access with more mixed-access instruction. When there is high repeatability of instructions LRU, LFU, and SRRIP perform identically.

ACKNOWLEDGMENT

In this assignment many people contributed to this discussion. Namely Dr. EJ Kim, Pritam Majumder, the students of CSE 614 on student discussion forums, and Michael Nelson. They helped significantly on verifying the simulation setup procedure.

CONCLUSIONS

The benefit of SRRIP is that it is more robust to new instructions polluting into the cache whereas LRU, and to a lesser extent LFU, are more sensitive. Generally speaking, LFU performed better than LRU, and SRRIP performed better than both LFU and LRU. It has been verified that the claims made in [1] are valid and the data supports their assertion that SRRIP outperforms e

REFERENCES

[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.

[3] D. Sanchez and C. Kozyrakis. 2013. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA). 475–486