CSCE 614: HW4 REPORT (SRRIP)

Nader Ghassemi
Computer Science & Engineering
Texas A& M University

***Abstract*—In the cases where several programs are co-scheduled to processed on a system there is opportunity to modify system performance. This chance can be exploited by the hardware.The hardware could improve system by better replacement policies by allocating more cache resources to programs that benefit from the cache and less to those programs that do not.**
Here we study a detailed analysis of static RRIP cache replacement policies. As mentioned above, cache consumption has a huge impact on the efficiency of a computing system. When cache miss happens it based on the frequency of the clock, it could lead to several thousands stalls. We go through the different algorithms on cache replacement and contrast the performance of a few algorithms including Least Recently Used (LRU), Least Frequently Used (LFU) and finally Static Re-Reference Interval Prediction (SRRIP). We use Zsim simulation to investigate the Ameer's paper that claims Static Re-Reference Interval Prediction has more hit rate than LRU.

I. Introduction

Traditionally, data transformation between cache and CPU had been implemented in the first come first served algorithm or more commonly used term first-in-first-out (FIFO). The LRU replacement algorithm behaves similarly to a FIFO queue. When replacing an entry in the cache, it evicts the entry at the end of the queue and puts at the head. Least recently used algorithm forecasts that more recent entries in the cache would be implemented in the soon future and chooses the oldest entry in the cache for replacement. The advantages of this replacement method is that it is simple to follow and can be utilized with a circular buffer. On the other hand, it executes good due to its dependence on temporal locality as explained in Al Zoubi and et. al [2]. The drawback of this algorithm is that it directly depends on the cache size. For instance, lets suppose we have a loop and the cache couldn't hold all the instruction set included in that loop therefore eventually in the loop we will run into cache misses and it will affects execution in the case the loop repeats too many times. Besides, we presume that an instruction who executed recently will be executed in the near-immediate future. Normally, in transferring process of 64 byte blocks, there will be a counter associated with each entry in the least frequently used algorithm which implies how often it has been implemented. Whenever we want to add the new entry, it evicts the entry with the lowest using history [2]. There are several advantages for LRU, first, this algorithm includes more history. An available entry could be the least recently used yet would be obtained with high frequency. In another words, we can include more data about temporal localityin LFU than LRU because it depends not only on the sequence of instructions but also their frequency. Meanwhile, the LFU algorithm is more opposing to the situation where there are bunch of new entries stored in the cache in which infrequent instructions are being performed. Nevertheless, this also biases cache entries toward instructions with temporal locality that might not processed in the near time and for which other, more immediate entries are replaced from the cache which cache capacity misses will be the consequence. While LRU provides good performance for workloads with high data locality, LRU limits performance when the prediction of a near immediate re-reference interval is incorrect. Applications whose rereferences only occur in the distant future perform badly under LRU. Such applications correspond to situations where the application working set is larger than the available cache or when a burst of references to non-temporal data discards the active working set from the cache. In both scenarios, LRU inefficiently utilizes the cache since newly inserted blocks have no temporal locality after insertion.

Re-reference Interval Prediction (RRIP), on the other hand,

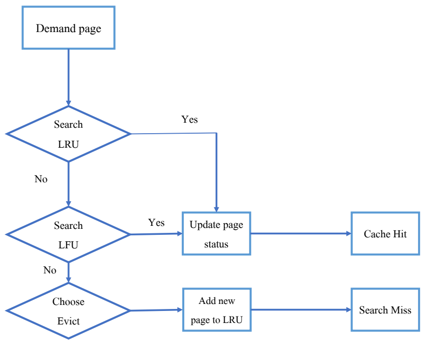

Fig. 1. Flowchart

creates 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]. SRRIP focuses on creating high performance scan-resistant replacement policy that requires low hardware overhead, retains the existing cache structure, and most importantly integrates easily into existing hardware approximations of LRUThese 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. In dynamic RRIP the value assigned to each entry is equal for each instruction while in SRRIP, each instruction get different value which is assigned to the entry.

II. SRRIP Technique

A. Short description of SRRIP technique

This method creates more information by assigning a re-reference bit prediction value (RRPV) between 0 to the $2M-1$. Where in the case that the cache is empty all entries are getting a value of $2M-1$. Then, It will looks the cache up to find a match. Whenever there is a hit, it sets the RRPV value of the cache hit to the zero which shows that it expects to be re-referenced in the near future. If it is a cache miss then it searches the block for a cache entry that has an RRPV value of $2M-1$. It replaces that entry and places the new instruction in that location and giving it an RRPV value of $2M-2$ which shows a distant re-reference prediction. In the case, there is not any entry of value $2M-1$ we wil increment the RRPV value of all entries by one nuimber and iteratively research all blocks of the cache till an entry matches with RRPV value of $2M-1$. In the case that there are several entries with RRPV value equal to $2M-1$, it replaces the first entry it finds in a process similar to RR.

B. Implementation details

For each implementation we need to set the RRPV value as well as the cache size and the cache insertion value which was already fixed at $2M-2$ for our 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. $rrip_repl.h.$ file includes the source code of SRRIP. In this implementations we needed to update the cache in order to set the entry's RRPV value to zero on a cache hit and in the case a cache miss happens it sets the new cache entry's value to $2M-2$. Nonetheless, in the case a cache hit happens, no more processing is required and the next instruction will be fetched and processed.
When a cache miss occurs, the first entry with the maximum RRPV value which the cache found will be evicted. However, it will increments all RRPV values in the case that no entry being found. At the end, the new entry being replaced to the old instruction.
In the case cache hit happens the data flow shown in figure 2 and in the case miss happens the data flow shown in figure 3.

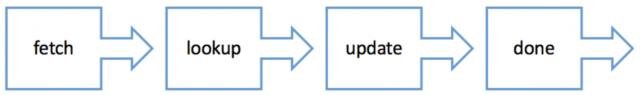

Fig. 2. DAta flow for cache hit.

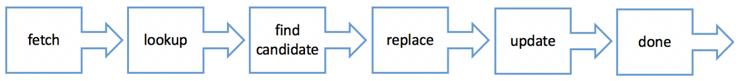

Fig. 3. Data flow for cache miss.

III. Methodology

We have used Zsim to produce the result and to compare and examin the SRRIP algorithm utilization on several benchmarks. ZSim is a fast, scalable, and accurate microarchitectural simulation infrastructure. It is based on $C++$ language and can simulate multi-core systems with detailed out-of-order cores and complex, heterogeneous memory hierarchies at speeds of 10s to 100s of MIPS [4]. 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]. If we want to compare the simulation instruction decoding to a real system the absolute error rate is 1.3% [3]. Considering all aforementioned rates, Zsim is a proper software that allows (in the error rates range) for a convenient analyze of SRRIP versus the other two cache replacement policies LRU and LFU. In simulation, suites selected are the SPEC for single core CPU2006 and PARSEC for multi-core threads, which are two widely used and well-known test suites. Besides, we have executed 69 simulations based on twenty-three benchmarks given which seven of them were integer and seven were floating-point and nine were multi-threaded and we implemented each of above benchmarks on the three LRU, LFU, SRRIP replacement policies.

Benchmarks List		
SPEC Integer	SPEC Floating Point	PARSEC
bzip2	milc	blackscholes
gcc	cactusADM	bodytrack
mcf	leslie3d	canneal
hmmer	namd	dedup
sjeng	soplex	fluidanimate
libquantum	calculix	freqmine
xalan	Lbm	streamcluster
		swaptions
		x264

TABLE I
SPEC CONFIGURATION

Memory Page Size	8 KB
Threads	1
L1-Instruction	Set Associative LRU 32 KB 4 way
L1-Data	Set Associative LRU 32 KB 8 way
L2	Set Associative LRU 256 KB 8 way
L3	Set Associative LRU, LFU, SRRIP 2 MB 16 way

TABLE II
PARSEC CONFIGURATION

Memory Page Size	8 KB
Threads	8
L1-Instruction	Set Associative LRU 32 KB 4 way
L1-Data	Set Associative LRU 32 KB 8 way
L2	Set Associative LRU 256 KB 8 way
L3	Set Associative LRU, LFU, SRRIP 8 MB 16 way

The configuration for the benchmarks were pretty similar except for the cache eviction algorithm and the size of $l3$ cache memory between the SPEC single core and PARSEC multi-core 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 as shown in figure 4 .

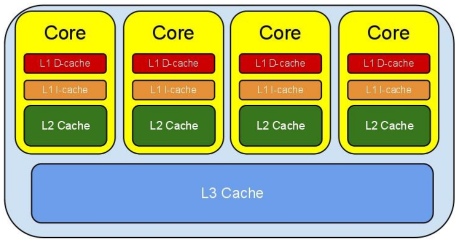

Fig. 4. Cache hierarchy. We see that $l3$ is shared between all cores and schematically is larger.

For the single core SPEC benchmarks it executes in hundred million instructions and the multi-core PARSEC benchmarks executes the total parallel phase for around five billion different instructions. executing the test suite of several kinds of benchmarks brings a broad perspective on the execution and restrictions of every LRU and LFU and SRRIP algorithms. Therefore, it makes sense to generalize and predict based on the results we are getting. We are able to see performance of SRRIP and compare it with other two replacement policies based on these benchmarks. I used the linux.cse.tamu.edu server to run these simulations and results were saved on zsim.out for each benchmark.

After running the simulation, we realized that SRRIP has no advantage over LRU in L1-I, L1-D, or L2 caches because of the small restricted size of the cache L1 and L2 caches and since they have a small size LRU is used for these caches.

There were several issues while running the simulation. There were a few measures to check our results to whether if there is any bug in our code and we had to wait till we compare our LRU LFU and SRRIP results together and with reference[1]. Even though, Zsim is faster than other simulators software, the benchmarks needed a non-trivial length of time to finish and necessary resources [3]. Considering the number of students in the class and restricted tamu cse servers a few benchmark simulations errored out and we restarted again when the load of severs were not heavy.

IV. EVALUATION

The investigation of the LRU, LFU and SRRIP were done by quantitative examination of the following measures

- Number of Cycles
- IPC
- MPKI

Where for the single-threaded SPEC benchmark, we used the following equation to compute the total number of cycles

$$total\_cycles = cycles + cCycles \tag{1}$$

In the above equation $cCycles$ are the number of cycles took in $cycles$ are simulated unhalted cycles. While since in the PARSEC cases cycles are being ran in 8 cores, total number of cycles derived by summing all different cycles of each different core. Then, since the max number of cycles describes the critical path, maximum amount between each core is considered as follows:

$$total\_cycles0 = cycles0 + cCycles0$$

$$total\_cycles1 = cycles1 + cCycles1$$

$$total\_cycles2 = cycles2 + cCycles2$$

$$\dots$$

$$total\_cycles7 = cycles7 + cCycles7$$

$$total\_cycles = Max(total\_cycles0, total\_cycles1,$$

$$\dots, total\_cycles7)$$

Instructions Per Cycle (IPC) and Misses Per Kilo Instructions (MPKI) can be derived as follows:

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

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

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

Where mGETS are misses of the GETS , mGETXIM are the misses of $GETXI- > M$, and mGETXSM are the misses of the $GETXS- > M$.

As illustrated in figure 4, the L3 is stabilized as shared and divided among several cores. In order to correctly calculate MPKI , in the multi-threaded case the total number of misses and total number of instructions required to be accumulated across all around the cores. to correctly calculate MPKI.
We have shown in the bar graph forms of the executions. Some of the test have identical results in some of the figures. In some cases the dynamic range of graphs are shown on a separate plot. We do that in order to show the precision more properly.

There is a distinct relationship between the performance metrics plotted for the benchmarks. We can see that there is a relation between the these metrics. The total cycle count and the IPC are inversely proportional. Hence, whenever there are benchmarks that have a high cycle numbers, speaking loosely, we can predict they have less IPC. Such as, the mcf benchmark confirms this point. Besides, cycle numebrs are directly proportional to major miss rates.

The cache misses have important influence on the total number of cycles in the benchmark, even though the rate of misses is relatively low. This feature can be viewed clearly in cactusADM, namd, and calculix benchmarks where miss rates are really small. This could be seen between cactusADM and milc, the MPKI difference is around 8%, although the total number of cycles is relatively more than45% larger for milc relative to cactusADM if we compare. In a converse manner, lbm has twice the MPKI miss rate of milc but is only 15% more cycles.
To recapitulate, the cycle numbers are directly proportional miss rates, however, IPC entangled to total efficiency because the overall performance of instructions in total is more significant that cache misses even when it happens at a high rate. It is noted that if the simulations run for short amount of time we couldn't see these principles and this investigations are true for floating points and integer simulation. In the PARSEC tests, more patterns are

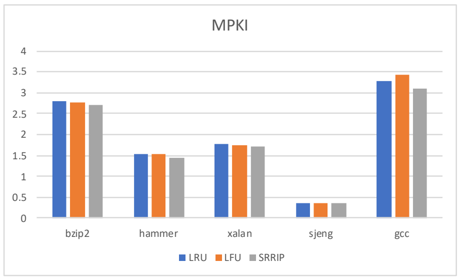

Fig. 5. SPEC integer benchmark.

For the PARSEC multithreaded simulations, patterns are more distinctive. 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 IPC of the multicore simulations is in the expected range when you scale the single-threaded processes by eight threads indicating that the benchmarks executes instructions that are relatively independent among cores.

To the extent that the connection between the individual cache replacement algorithm, LRU, LFU, or SRRIP, we can see that, in general (totally), that SRRIP methods 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, the reason is a more equal weighting for cache entries as opposed to LRU which brings balanced weighting for each individual new coming entry. This distinction could be seen profoundly in the streamcluster benchmark. This could be assigned to a less solid instruction access with more mixed-access instruction. In the case there is high repeatability of instructions LRU, LFU, and SRRIP perform similarly.In this study, we have applied RRIP on cache misses and learn the re-reference interval of the missing block without any external information. Re-reference interval prediction on cache hits ideally requires knowledge of when a cache block receives its last hit. RIPP can use such information to update the re-reference prediction of the re-referenced cache block to intermediate, long or distant re-reference interval.

Acknowledgment

Several people attributed in this assignment. Namely Pritam Majumder, Mohsen Mahmoudi and Aaron Skouby and the students of CSE 614 on student discussion forums. Several youtube videos.

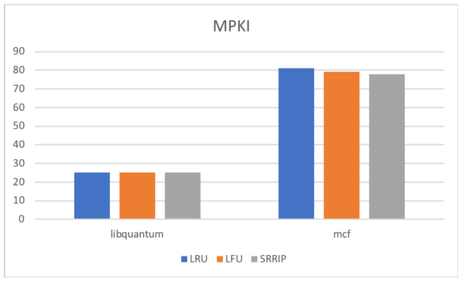

Fig. 6. SPEC integer benchmark.

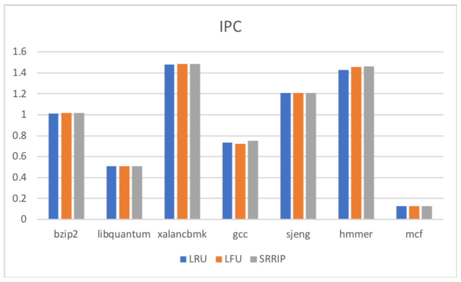

Fig. 7. SPEC integer benchmark.

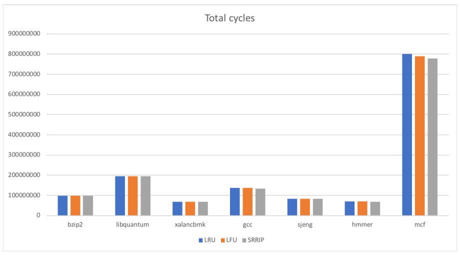

Fig. 8. SPEC integer benchmark.

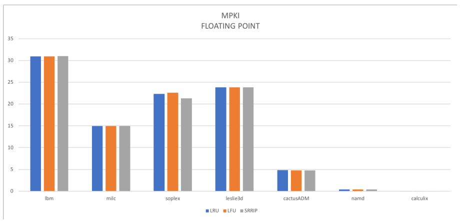

Fig. 9. SPEC floating point benchmark.

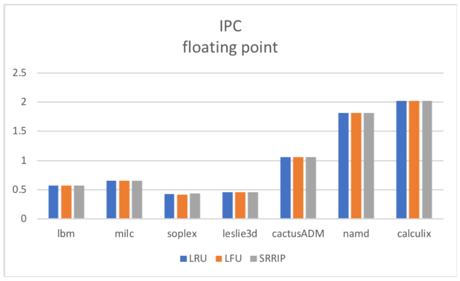

Fig. 10. SPEC floating point benchmark.

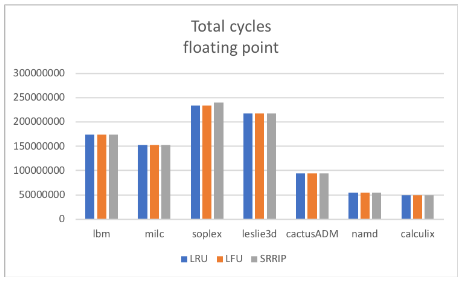

Fig. 11. SPEC floating point benchmark.

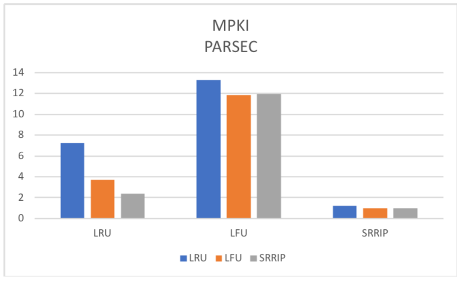

Fig. 12. SPEC floating point benchmark.

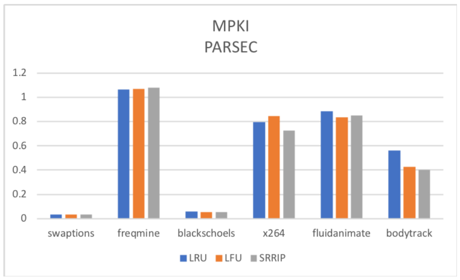

Fig. 13. PARSEC benchmark.

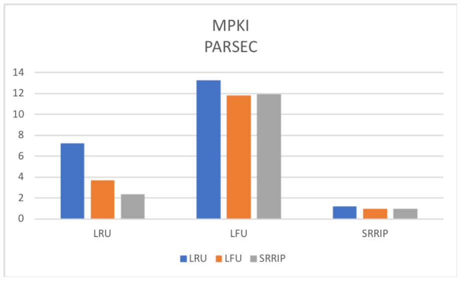

Fig. 14. PARSEC benchmark.

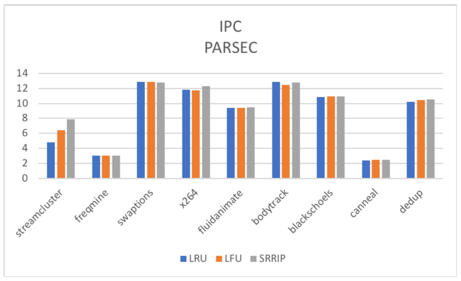

Fig. 15. PARSEC benchmark.

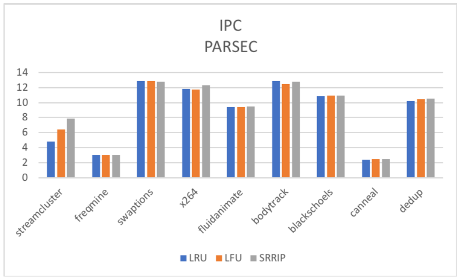

Fig. 16. PARSEC benchmark.

Conclusions

In our simulations, SRRIP performed better than both LFU and LRU and among LFU and LRU, the first one performed better than LRU, and we see that any cache block that receives a hit will have a near-immediate re-reference and thus should be retained in the cache for an extended period of time. The advantage of SRRIP is that it is more vigorous to new instructions inserting into the cache while the LRU and a little bit LFU are more sensitive.

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
[4] thousand-core systems. In Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA). 475–486 T. G. Rogers, M. O’Connor, and T. M. Aamodt. Cache-conscious Wavefront Scheduling. In MICRO, 2012
[5] http://zsim.csail.mit.edu/tutorial/