COMPARING ADAPTIVE INSERTION AND RE-REFERENCE INTERVAL PREDICTION CACHE REPLACEMENT POLICIES

Implementation of DIP by *Abhishek Nayak*
Implementation of DRRIP by *Deepayan Dasgupta*
(https://github.com/DeepDasg/gem5_cache_replacement)

***Abstract* – Hybrid policies like DIP and DRRIP recognize that different workloads or phases within the same workload benefit from different replacement strategies. For example, if a program alternates between small and large working sets, it will benefit from a recency-friendly policy when it is small and from a thrash-resistant policy when the working set is large. Thus, hybrid policies assess the requirements of the application's current working set and dynamically choose from among multiple competing policies.**

In this report, we analyze two such replacement policies, (1) Dynamic Insertion policy (DIP) and (2) Dynamic Re-Reference Interval Prediction (DRRIP). The DIP chooses between traditional LRU policy and Bimodal Insertion policy (BIP), depending on which policy incurs fewer misses. On the other hand, the DRRIP chooses dynamically between SRRIP (which is suitable for being scan-resistant) and BRRIP (which is thrashing-resistant). Both the policies use a set-dueling mechanism to dynamically switch between two static insertion policies and settle on which would perform the best out of the two for the given workload. The set-dueling mechanism significantly saves on extra hardware overhead and is relatively robust.

Our DIP and DRRIP replacement policies implementations give us a throughput improvement of up to 4-8% for DIP and 4-9.5% for DRRIP over LRU replacement. For overall LLC cache misses, we see an improvement of up to an average of ~14% for DIP and ~28% for DRRIP over LRU replacement for specific benchmarks.

1. INTRODUCTION

Caches act as a high-speed buffer between CPU and Main Memory. It improves performance by making the data needed in the pipeline available within a few cycles, thereby preventing stalling in the pipeline. Although caches are much faster than main memory, they are expensive from a hardware budgeting standpoint, resulting in processors having small cache sizes. Since the data in a cache is updated frequently, a data replacement policy is necessary. Most cache replacement policies can be broadly split into two parts: eviction policy and insertion policy.

In an ideal scenario, the replacement policy should evict the cache block, which would be referenced furthest in the future. Thus, an adequate replacement policy must be able to predict the re-reference interval of the block and then replace the block, which is referenced furthest in the future, making the prediction accuracy a critical factor in determining the performance of the last level cache (LLC). The performance of the LLC is vital to the performance of the processor since any miss in LLC requires blocks to be fetched from the main memory, incurring a penalty of several cycles.

The Least Recently Used (LRU) Policy is a commonly used replacement policy in modern processors. LRU creates a queue to represent the recency of the cache blocks. The Most Recently Used (MRU) block is at the head of the queue, and the Least Recently Used (LRU) block is at the tail of the queue. Whenever a new cache block is bought in, LRU places that block at the MRU position and evicts the LRU block. When the MRU block is predicted to be re-referenced sometime soon, it is said to have a near-immediate re-reference interval. In contrast, when the LRU block is predicted to be re-referenced far into the future, it is said to have a distant re-reference interval. Thus, LRU predicts that every newly added cache block has a near-immediate re-reference interval and places the new block at the MRU position. When a hit happens in the cache and a block is re-referenced, LRU policy moves that block to the MRU position, which means that LRU again predicts that the re-referenced block has a near-immediate re-reference interval.

Another aspect that needs to be taken care of is the insertion policy. For example, we might mark the incoming line as least/most recently used depending on the workload. Eviction and insertion policy together comprise the cache replacement policy. The traditional LRU replacement policy is made of LRU eviction policy and MRU insertion policy.

Since LRU assigns every newly added or re-referenced cache block to have a near-immediate re-reference interval, it works well for workloads that have a high amount of data locality. However, this prediction fails for workloads where blocks are re-referenced farther into the future, and the LRU replacement policy has many misses. This happens in workloads where the working set is much larger than the cache size or when workloads have a burst of non-temporal data which replaces the active working set in the cache. In both cases, the new cache block has no temporal locality, i.e., it has a distant re-reference interval.

In cases where the set size exceeds the cache size, the MRU insertion policy of LRU replacement would get zero cache hits. Such workloads are LRU-averse. However, if we can use an insertion policy that dynamically determines if a workload is LRU-averse or LRU-friendly, we can solve the problem of cache thrashing.

Table 1 lists common access patterns that are encountered in workloads. The address of the cache line is a_i and $(a_1, a_2, \ldots a_{k-1}, a_k)$ represents the temporal sequence of k unique addresses that are referenced in the workload. A temporal sequence of k unique addresses that is repeated N times is represented by $(a_1, a_2, \ldots, a_{k-1}, a_k)^N$. $P_\varepsilon (a_1, a_2, \ldots, a_{k+1}, a_m)$ means that the temporal access sequence of addresses from a_1 to a_m can occur with a probability of ε.

TABLE 1

COMMON CACHE ACCESS PATTERNS

(a)	**Recency-friendly Access Pattern (for any k)**
	$(a_1, a_2, \ldots, a_{k-1}, a_k, a_k, a_{k-1}, \ldots, a_2, a_1)^N$
(b)	**Thrashing Access Pattern (*k* > cache size)**
	$(a_1, a_2, \ldots, a_{k-1}, a_k)^N$
(c)	**Streaming Access Pattern (*k* = ∞)**
	$(a_1, a_2, \ldots, a_{k-1}, a_k)$
(d)	**Mixed Access Pattern (*k* < cache size AND *m* > cache size, 0 < ε < 1)**
	$[(a_1, \ldots, a_k, a_k, \ldots, a_2, a_1)^A P_\varepsilon (a_1, a_2, \ldots, a_{k+1}, a_m)]^N$
	$[(a_1, \ldots, a_k)^A P_\varepsilon (b_1, b_2, \ldots, b_m)]^N$

Common access patterns are listed below:

1) Recency-friendly access pattern: Table I(a) shows a recency-friendly access pattern being repeated N times. It usually involves a temporal sequence of k unique addresses followed by the series of references being repeated in the reverse order. These cache accesses have a near-immediate re-reference interval, ensuring LRU works well for these accesses, while non-LRU policies will degrade performances for the same accesses.

2) Thrashing access pattern: Table I(b) shows a cyclic access pattern to k unique addresses repeated N times. For k less than or equal to the cache size, the entire set fits into the cache, and we have no misses. However, for k larger than the cache size, LRU would evict the whole block to bring in a new cache block. A non-LRU replacement policy would allow us to preserve some of the working sets in the cache.

3) Streaming access pattern: Table I(c) shows a streaming access pattern, where at any location k, the block has an infinite re-reference interval, i.e., there is no locality in the access pattern. Here, both LRU and non-LRU access patterns would give the same performance.

4) Mixed access pattern: Table I(d) shows a mixed access pattern, where some references have a near-immediate re-reference interval while others have a distant re-reference interval. Both examples include an access pattern of length k that repeats A times followed by a reference to a sequence of length m with probability ε, with both repeating N times. When m + k is less than the cache size, the entire working set fits into the cache, and LRU works well. However, when m + k is greater than the cache size, LRU discards the frequently referenced working set (scan) from the cache. A scan is defined as a burst of references to data whose re-reference interval is in the distant future. In comparison, accesses that do not belong to the scan have a near-immediate re-reference interval. Here the prediction of near-immediate re-reference interval does not hold, and LRU results in cache miss when the frequently referenced working set is re-referenced after the scan. Thus, in a case where a mix of re-reference accesses occurs, LRU alone cannot give us optimal performance.

In conclusion, LRU requires a high cache locality to perform optimally. In Last Level Caches, most of the locality has been filtered out by L1 and L2, and L3 serves the L1/L2 cache misses. The reduced locality makes LRU a sub-optimal solution for LLCs. Cache utilization is lowered even further when the distance between the consecutive references to the cache block is greater than the cache size, causing LRU to evict the cache block before it is re-referenced. Thus, we need a policy that performs well for both LRU-friendly and LRU-averse mixed workloads, while requiring negligible hardware overhead and changes.

2. RELATED WORK

There has been a plethora of works from industry and academia looking into cache replacement policies to minimize the encountered misses and improve cache performance. In this section, we list down and discuss some of the replacement policies and show their effectiveness to the different access patterns discussed earlier.

I. LRU Insertion Policy

LRU Insertion Policy (LIP) places all incoming lines in the LRU position. These lines are promoted from the LRU position to the MRU position only if they are referenced in the LRU position.

LIP policy effectively achieves high cache performance with a scan access pattern where the working set is more significant than the cache size. It is essential to retain a part of the active set of frequently accessed temporal data in such access patterns. When the LRU replacement policy is followed, the often-referenced working set ends up getting evicted from the cache when a burst of non-temporal references comes. As a result, access to data with high temporal locality after the scan always results in a miss. However, with LIP, all incoming blocks are inserted at the LRU position. The re-referenced data is moved to the MRU position, and as a result, the data that is not referenced anytime soon will remain at the end of the recency queue and get evicted more shortly than it would when the blocks are placed at the MRU position. This allows the LIP replacement policy to retain a part of the working set of the scan that is frequently accessed, and that data continues getting hit even after the burst of non-temporal data.

II. Bimodal Insertion Policy

Bimodal Insertion Policy (BIP) is like LIP, except that BIP infrequently (with a low probability) places the incoming line in the MRU position. BIP adapts to changes in the working set while retaining the thrashing protection of LIP.

It uses a bimodal throttle parameter (ε) to determine the percentage of incoming lines to insert at the MRU position. Both LRU and LIP can be thought of as a particular BIP case. LRU is the same as BIP with $\varepsilon = 100$ such that all incoming lines are inserted at the MRU position, while LIP is the same as BIP with $\varepsilon = 0$ such that all incoming lines are inserted at LRU. BIP was proposed to tackle the lack of aging problem with LIP. The effectiveness of BIP over LIP is discussed below.

Consider an access pattern of the form $(a_1, a_2 \ldots a_T)^N$ & $(b_1, b_2 \ldots b_T)^N$, the access to addresses a_i is repeated N times followed by accesses to addresses b_i which is again repeated N times. Suppose the cache size is K and $K < T$. Table II lists down the number of hits received by LRU, LIP, BIP, and OPT replacement policies. OPT is the optimum replacement policy, and it's the upper ceiling in the number of hits that can be received by any practical replacement policy. As discussed before as well, for this kind of access pattern, LRU would cause thrashing, and consequently, it receives no hits. An optimum replacement policy would retain $K-1$ out of the T lines during a cyclic reference pattern. Hence, once the cache warm-up period is over, OPT replacement policy keeps those blocks within the cache and achieves a hit rate of $(K-1)/T$. For the first access pattern, after cache warm-up, even LIP can retain $K-1$ out of the T blocks that are referenced in the cyclic pattern. For the first access pattern LIP achieves the same hit rate at the optimum replacement policy. However, the problem arises when the access moves on to the second pattern. Since LIP always ends up placing blocks at LRU position, for the second reference pattern, the blocks are never able to move to a non-LRU position in the recency queue. As a result, LIP achieves no hits for the second access pattern. BIP on the other hand inserts approximately $\varepsilon(T-K)$ at the MRU location. This leads to a hit rate of $(K-1-\varepsilon\,(T-K))/T$. Since ε is a small parameter (typically chosen to be as low as 3%) the hit rate is approximately equal to $(K-1)/T$. So, for the first access pattern, BIP performs as good as the optimum, same as LIP. When the access pattern changes, after K/ε misses, all the cache lines in the cache would belong to the second access patterns and after those number of misses, at steady state, BIP again would achieve a hit rate of $(K-1-\varepsilon.(T-K))/T$. Hence unlike LIP where the second access pattern was stuck at the LRU position, BIP sets some of the blocks be inserted to MRU and can perform as good as the optimum replacement policy for this access pattern.

III. Dynamic Insertion Policy

The Dynamic Insertion Policy (DIP) combines the benefit of recency-friendly policies and thrash-resistant policies by dynamically modulating the insertion positions of incoming lines. DIP alternates between the recency friendly LRU and the thrash resistant Bimodal Insertion Policy.

A straightforward method of implementing DIP is to implement both LRU and BIP in two extra tag directories (data lines are not necessary for estimating the misses incurred by an insertion policy) and keep track of the number of misses incurred by the two policies. The cache's main tag directory can then use the policy that incurs the fewest misses. However, this implementation incurs prohibitive area and power overheads. For the proposed mechanism to be useful, it must incur very low overhead and minimal changes to the existing cache design. Thus, we next describe a cost-effective runtime mechanism to implement DIP.

To choose between the two policies, DIP uses Set Dueling to dynamically track the hit rate of each policy. Figure 1 shows that DIP dedicates a few sets to LRU (sets 0, 5, 10 and 15 in Figure 1) and a few sets to BIP (sets 3, 6, 9 and 12 in Figure 1). These dedicated sets are called Set Dueling Monitors (SDMs), while the remaining sets are called the follower sets. The policy selection (PSEL) saturating counter determines the winning policy by identifying the SDM that receives more cache hits. In particular, the PSEL is incremented when the SDM dedicated to LRU receives a hit, and it is decremented when the SDM dedicated to BIP receives a hit (a k-bit PSEL counter is initialized to 2^{k-1}). The winning policy is identified by the MSB of the PSEL. If the MSB of PSEL is 0, the follower sets use the LRU policy; otherwise, the follower sets use BIP. Thus, Set Dueling does not require any separate storage structure other than the PSEL counter.

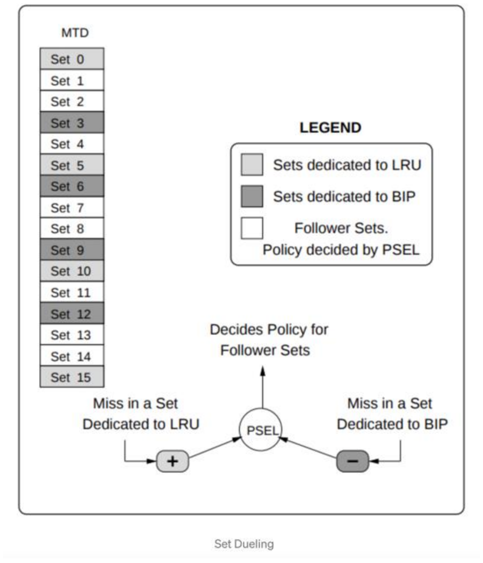

Figure 1

IV. Least Frequently Used

Least Frequently Used (LFU) replacement policy tackles the issue of handling dead block brought in due to scan access pattern. As the name suggests, LFU replacement policy replaces the block which has been least frequently used since it was brought in. This ensures that the block which does not receives any further hits is the one to be evicted first. LRU does not follow a recency queue like LRU but instead it uses access frequency to predict the re-reference interval of the cache blocks. LFU assumes that block that has been frequently referenced would have a near immediate re-reference interval while the block which is in frequently referenced would have a distant re-reference interval. To determine the access frequency of the cache blocks, LFU policy requires a saturating counter with every cache block. When the block is brought into the cache, the counter is set to the minimum value. Upon a cache hit, the counter associated with the block is incremented. The block with the least value of the counter is evicted whenever a new block is brought in on a cache miss. LFU significantly improves the performance of workloads with frequent scans. However, it fails for those workloads where recency is the preferred choice of action.

V. Not Recently Used

Not Recently Used Replacement Policy (NRU) approximates LRU. Instead of using counter for each cache block like LFU, NRU adds an extra bit of information with each cache block. The nru-bit is used to predict the re-reference interval of the cache blocks. If the nru-bit is ’0’ then it means that the block has a near immediate re-reference interval. If the nru-bit is ’1’ then it means that the block has a distant re-reference interval. Like LRU, NRU assumes that the incoming block has a near immediate re-reference interval and the nru-bit is set to 0. Also, on a cache hit, it is again assumed for the block to have a near immediate re-reference interval. Victim selection for NRU is a little different compared to LRU. The block which is assumed to have a distant re-reference interval is the one which is evicted. The replacement policy starts the search from one end of the cache set and evicts the first block it encounters with nru bit as 1. When there is no block with nru bit as 1, this means that all the blocks are assumed to have a near-reference interval. In such a case, the nru-bits for all the cache blocks in the set is set to 1 and then the policy resumes its search for victim. Flipping the nru bit to 1 from 0 for all the blocks allows the policy to make forward progress and remove stale blocks from the cache. NRU either predicts the cache blocks to either have a near immediate re-reference or a distant re-reference. Always predicting near immediate re-reference on all cache accesses does not work for scan access patterns because it causes scan blocks to unnecessarily occupy cache blocks. While always predicting a distant re-reference interval fails for workloads which have predominantly near immediate re-reference interval. NRU either predicts either near re-reference interval or distant re-reference interval based on the nru-bit and without any external information, NRU cannot preserve non scan blocks (which are frequently re-referenced) in the cache and as a result lead to unnecessary misses.

VI. Static Re-reference Interval Prediction

Static Re-reference Interval Prediction (SRRIP) replacement policy tries to address the limitations with NRU policy. It uses multiple bits of information to predict re-reference interval of each cache block. These multiple bits of Re Reference Prediction Values (RRPV) are used to predict re-reference interval of cache blocks. With M bits, each cache block is allowed to have 2^M RRPV values. RRPV of 2^{M-1} implies the block would receive hits in the distant future while RRPV value of 0 implies that the block is predicted to have a near immediate re-reference interval. Any RRPV value between 0 and 2^{M-1} implies that the block is predicted to have an intermediate re-reference interval. Blocks with lower value of RRPV are predicted to be re-referenced earlier than blocks with higher RRPV. SRRIP is identical to NRU when $M = 1$ and the prediction is made to be either near immediate or distant. For $M > 1$, intermediate values of re-reference intervals are possible.

SRRIP tries to prevent scan blocks with distant re-reference interval from polluting the cache. Since, always predicting near immediate or a distant re-reference interval is not robust to scan access patterns, SRRIP predicts that every incoming cache line to have a long re-reference interval. Long re-reference interval is an intermediate re-reference that is skewed towards distant re-reference interval. Usually setting long re-reference interval as $2^M - 2$ works fine. The idea behind choosing all incoming cache lines to be predicted to have long re-reference interval is that it doesn’t let blocks with low temporal locality addressed during scan pollute the cache. When the workload demands near immediate re-reference interval prediction SRRIP can update its RRPV value to be smaller than before. Consequently, SRRIP can learn the re-reference prediction demanded by the workload and achieve improved performance. Upon receiving a cache hit, the RRPV is decremented. While selecting a victim to be evicted from cache, SRRIP searches for the block with RRPV equivalent to distant re-reference interval prediction. If no such block exists, RRPV of each cache block is increased, and the victim selection process is repeated. While SRRIP can effectively deal with scan access patterns, if the re-reference interval of all the blocks is larger than the cache size, then SRRIP can cause thrashing. This is a consequence of RRPV of all new cache blocks being initialized close to distant re-reference intervals, as was seen in the LIP replacement policy.

VII. Bimodal Re-Reference Interval Prediction

Bimodal Re-reference Interval Prediction (BRRIP) tries to address the inability of SRRIP in managing workloads where the working set is larger than the cache size (thrashing workloads). To avoid thrashing, BRRIP assigns most of the incoming cache blocks with distant re-reference intervals and infrequently assigns some of the blocks with long re-reference intervals. This is like BIP replacement policy, but instead of infrequently inserting blocks with near-immediate re-reference interval, BRRIP inserts them with a long re-reference interval. Analogous to BIP, a bimodal throttling parameter decided the percentage of blocks to be inserted with long re-reference interval prediction. However, always using BRRIP can lead to lower performance on workloads is a non-thrashing access pattern.

VIII. Dynamic Re-Reference Interval Policy

DRRIP builds on DIP to add scan resistance. DRRIP uses set dueling to create a hybrid of SRRIP, which is the scan-resistant version of LRU, and BRRIP, which is the scan-resistant version of BIP.

Set Dueling mechanism dedicates a few sets in the cache to follow SRRIP and a few sets to follow BRRIP. The performance of the two policies on the dedicated sets is monitored, and the remaining sets (termed as follower sets) follow the policy, which incurs the least misses. DRRIP keeps track of the misses incurred in the dedicated sets by using a counter termed as the Set Dueling Counter (SDC). When a miss is incurred in a dedicated set following SRRIP, SDC is incremented. If a miss is incurred in a dedicated set following BRRIP, SDC is decremented. When the program starts execution, the SDC is initialized to a middle value within the dynamic range of the counter. If the value of SDC is more significant than its initialized value, this means that the SRRIP policy has incurred more misses than the follower sets that follow BRRIP. Figure 2 shows the working of set dueling on a cache containing sixteen sets.

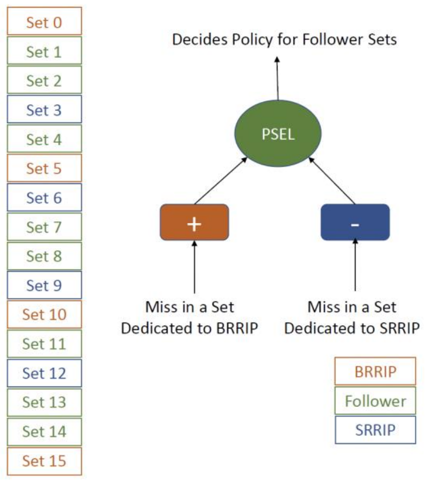

Figure 2

Follower sets follow SRRIP when BRRIP has incurred more misses (i.e., the SDC value is lower than its initialized value). A number of bits dedicated to the saturating counter determines its dynamic range. If the dynamic range of the counter is too low, then SDC would frequently switch between its max and min value and consequently, the follower sets will keep switching between SRRIP and BRRIP replacement policy. Another challenge to be addressed is the choice of dedicated sets. They can be chosen dynamically or statically during design time. One crucial design choice to keep in mind is that the selected dedicated sets must be spatially spread out over the cache. Spatial locality within the workload can cause incorrect decisions if nearby cache sets are dedicated to following one of the two policies. Assuming the cache to have N sets and K sets out of them are dedicated to following each policy. The cache is first logically divided into equally sized N/K sections. Each section is called *constituency*. One set is selected from each constituency to follow SRRIP and BRRIP. The set index consists of $log_2(N)$ bits. Out of these, the most significant $log_2(K)$ bits represent the constituency, and the remaining $log_2(N/K)$ bits determine the first set in the constituency. Sets indices where the constituency is equal to the offset are dedicated to following the SRRIP replacement policy, while all the sets where the complement of the offset equals constituency are devoted to following BRRIP.

3. IMPLEMENTATION

This section describes the significant details required to implement the adaptive insertion and RRIP policies in gem5. We start by looking at how to set the configuration for the experiments. And subsequently, understand the code hierarchy of gem5 for the classic memory system, which is divided and organized in the *mem* directory. We implement the two policies in separate files, which can be chosen by providing the required options while simulating the code.

I. System Configuration

We evaluate DIP and DRRIP using the out-of-order O3 CPU provided in gem5. The simulations are done on a single-core CPU with three levels of caches. The L1 cache is a 4-way set associative 32kB memory with one cycle load-use or data latency. L2 cache is 8-way with 256kB size and ten times higher latency penalty than L1. In contrast, the last level cache is 8-way associative 2MB memory which is inclusive, unlike L1 and L2. The configurations varied for the DIP paper, which did not implement the L3 cache. Moreover, the L1 cache is 2-way set associative 16kB memory and L2 (LLC) is a 16-way set associative with 1MB memory.

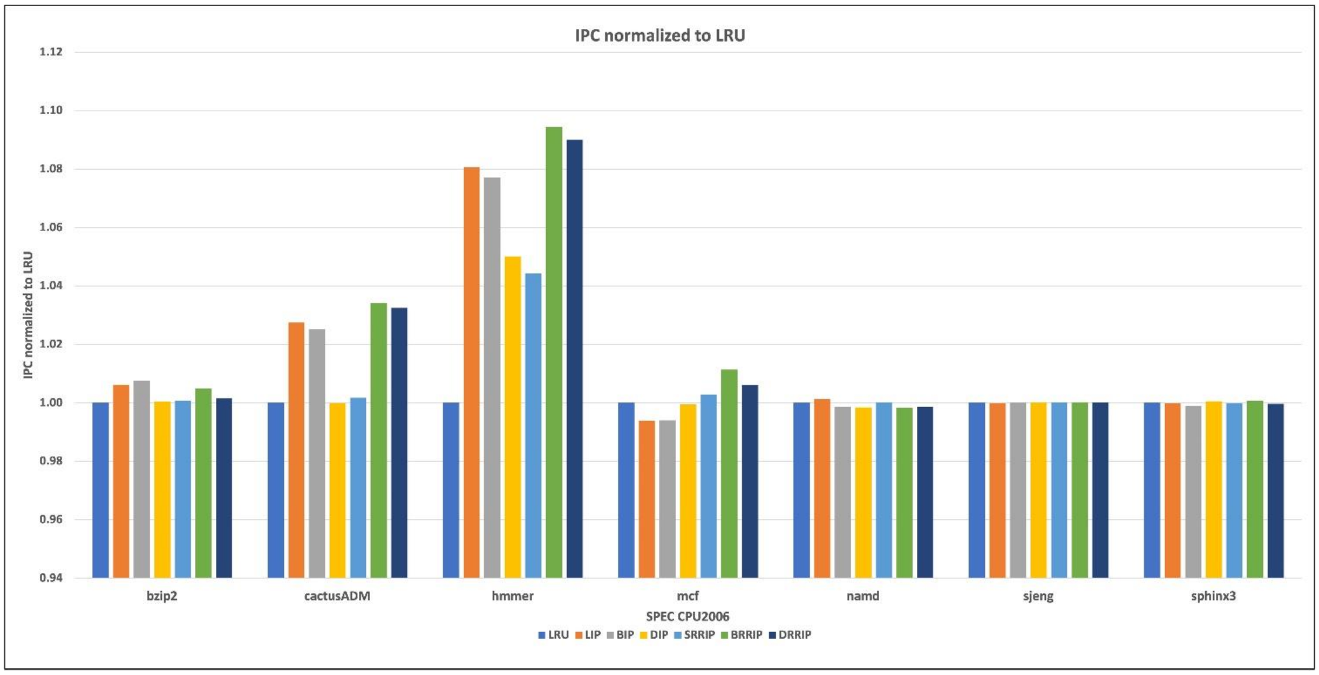

Figure 3

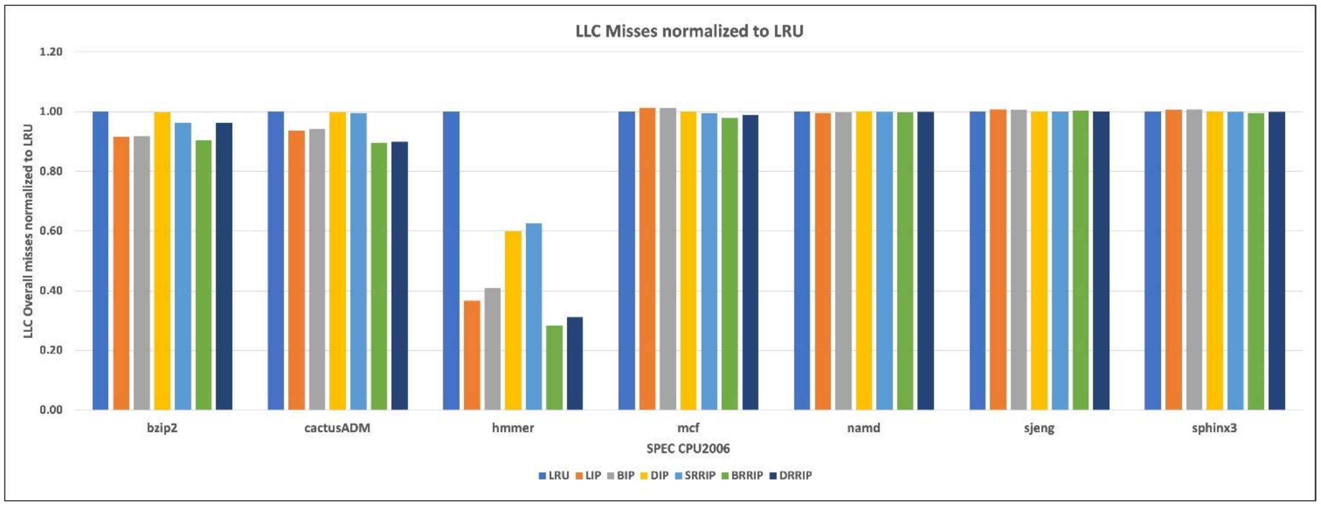

Figure 4

Since gem5 provides only a two-level cache hierarchy by default in the python configuration file, we declared a new class named L3Cache to implement the LLC and can be included with the flag –l3cache, like other cache options in gem5. The L2 bus going to the main memory earlier is connected to the CPU side of LLC (L3XBar), and the memory side of LLC is connected to the main memory. The memory hierarchy is also listed in Table II.

II. Simulation methodology

Since the project involves cache analysis, it is important to warm–up the memory well before analyzing the performance and benefits of the replacement policies. For this project, we used the *AtomicSimpleCPU* from gem5 to fast forward the workload by a billion instructions to skip the initial instructions, which might be just declaring the variables or instantiating the objects. We hope to reach the critical part of the benchmark by doing fast-forwarding. *AtomicSimpleCPU* is used for forwarding because it uses atomic memory accesses, making it more quickly than others. Next, we do the warm-up with 50 million instructions using the *TimingSimpleCPU*, which also accounts for the timing of the memory accesses, making it a comparatively accurate choice than *AtomicSimpleCPU* but still faster than *DerivO3CPU*. We reset the statistics after the warm-up is complete in our python configuration file and finally run a billion instructions to measure the performance benefits of all the replacement policies. We analyze the overall misses from the *DerivO3CPU* for the LLC and instructions per cycle (IPC) to measure how DIP and DRRIP are performing compared to LRU. We have chosen the SPEC 2006 benchmarks mentioned in the DIP and DRRIP papers, along with a few more relatively fast running benchmarks.

TABLE 1I
SYSTEM CONFIGURATION

L1 inst. cache	4-way 32kB LRU replacement policy
L1 data cache	4-way 32kB LRU replacement policy
L2 unified cache	8-way 256kB LRU replacement policy
L3 unified cache	8-way 2MB

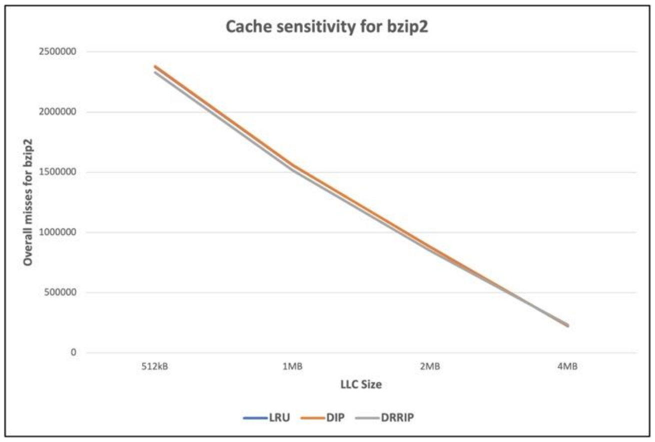

Figure 7

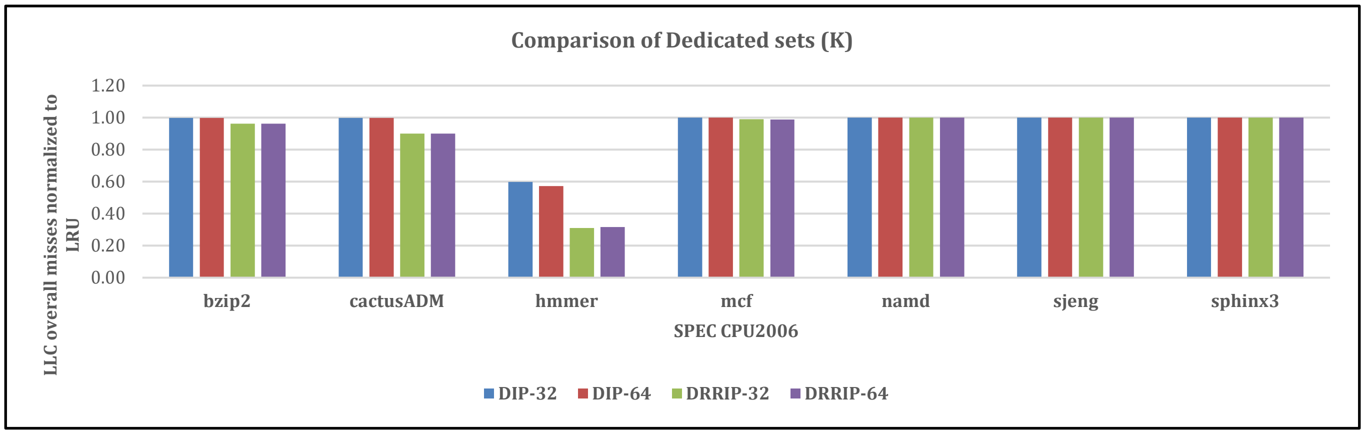

Figure 5

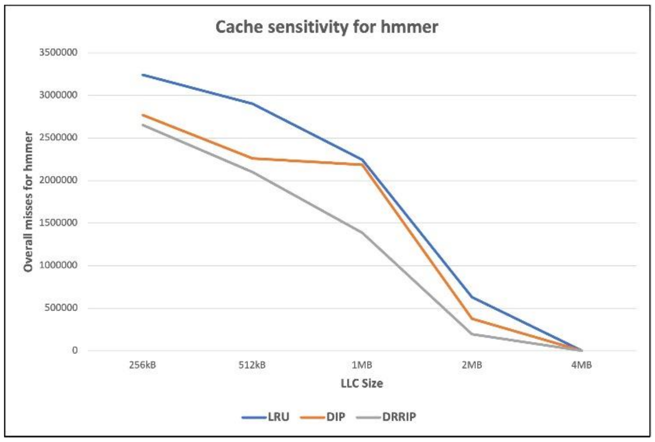

Figure 6

4. RESULTS

Comparing the benchmark results for DRRIP and DIP, in Figures 3 and 4 through the LLC demand misses and IPC obtained on the seven benchmarks, we see noticeable improvements in performance and a significant reduction in LLC misses for the *hmmer* benchmark (~40% for DIP and ~68% for DRRIP with respect to traditional LRU), followed by *cactusADM* and *bzip2*. For the other four benchmarks, *namd*, *mcf*, *sphinx3* and *sjeng*, there are very marginal benefits in LLC misses and IPC compared to LRU. The variation in performance over different benchmarks can be attributed to the type of instructions and working set requirement.

As an example, we have analyzed the performance of the *bzip2* benchmark. The *bzip2* benchmark performs poorly with SRRIP (with 0.07% IPC improvement wrt LRU), but BRRIP performs better (with 1% IPC improvement). This could be attributed to the thrashing access pattern of the bzip2 benchmark. Note that the DRRIP performs better than SRRIP for *bzip2*.

For benchmarks where we see similar performance between DIP, DRRIP, and LRU, one possible explanation could be that most of the working set for those benchmarks create a streaming pattern of accesses, severely reducing the locality of data for the replacement policies.

I. Effect on the increase of cache size

We also checked the cache size sensitivity of *hmmer* and *bzip2* in Figures 6 and 7, respectively. For the *hmmer* workload, it is observed that with the increase in cache size, the cache miss rates decrease and reach a limit in terms of cache performance at 4MB cache size. From 256kB to 4MB, the cache miss rate reduces by a percentage of ~99%. The drastic decrease in the number of overall misses drops for *hmmer* at 4MB cache size implies that the working set is smaller than 4MB.

On the other hand, the cache sensitivity for *bzip2* varies linearly, which could be attributed to the thrashing access pattern of the benchmark.

II. Effect on the dedicated set count

In Figure 5, we have analyzed the change in LLC miss count with the difference in the number of dedicated sets (K = 32, 64) of the DIP and DRRIP that uses the set-dueling mechanism. Increasing the number of dedicated sets from 32 to 64 gives us a marginal reduction in cache misses since the probability of selecting the best policy increases.

5. CONCLUSION

We have demonstrated the performance of the two replacement policies, DIP and DRRIP, along with other replacement policies of LIP, BIP, SRRIP, and BRRIP on SPEC CPU2006 benchmarks by comparing demand misses and IPC on GEM5. We observed that there is significant performance improvement with DRRIP compared with DIP.

We need to recall how DIP works to understand why DRRIP performs better than DIP in the benchmarks. DIP dynamically uses LIP for workloads whose working set is larger than the available cache and relies on LRU for all other workloads, addressing the cache thrashing problem by preserving some of the working set in the cache. Unfortunately, DIP makes the exact predictions for all references of a workload. As a result, DIP limits the performance of workloads where frequent scans discard the active working from the cache. Consequently, when the workload re-reference pattern is mixed, i.e., both near-immediate and distant re-references occur, neither LRU nor DIP can make accurate predictions. This is where DRRIP can improve performance by dynamically choosing SRRIP or BRRIP based on the type of workload. As mentioned earlier, SRRIP can achieve high performance on scan workloads while BRRIP can achieve high performance on thrashing workloads. However, SRRIP fails when the workload has a thrashing access pattern, and BRRIP fails when the workload has a near-immediate re-reference pattern. Since DRRIP can choose between the two policies through set dueling, it is able to achieve scan resistance like SRRIP and thrashing resistance like BRRIP, thus delivering the observed improved result.

REFERENCES

[1] M. Qureshi, A. Jaleel, Y. Patt, S. Steely, J. Emer. "Adaptive Insertion Policies for High Performance Caching." In ISCA-34, 2007.

[2] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," ACMSIGARCH Computer Architecture News, vol. 38, no. 3, 2010.