Reuse-Aware Re-Reference Interval Prediction with Bypassing Techniques for Last-Level Cache Replacement Policies

Zachary Murtishi

University of Rhode Island, ELE 548

***Abstract*—The cache memory is an important mechanism for improving system performance in the face of high-latency access times associated with main memory accesses. While much of the performance improvements associated with the cache are due to the inherent speed associated with its proximity to the processor and low-latency memory technology, the cache replacement policy managing it has an important role in decreasing memory access times. This work proposes a simple, yet intelligent cache replacement policy called "Reuse-Aware Re-Reference Interval Prediction" (RA-RRIP) that adds a re-use awareness mechanism to the base re-reference interval prediction policy in order to optimize the insertion position of blocks based on their reuse frequency. Several other improvements are proposed to the base RA-RRIP policy, including a hybrid variant that uses set-dueling to select between RA-RRIP and LRU-like RRIP and others that make use of various cache bypassing schemes to improve performance. These policies were evaluated using the SPEC2006 CPU benchmarks for both single and multi-core configurations and the Cloudsuite benchmarks for multi-core only. Single core simulation results show that RA-RRIP variants perform up to 2.18% better than LRU for the SPEC2006 benchmarks. Multi-core results show up to 3.45% speedup over LRU for SPEC2006 benchmarks and up to 3.74% speedup over LRU for Cloudsuite benchmarks.**

***Index Terms*— least-recently-used, re-reference interval prediction, confidence counter, reuse bit, low-confidence hit ratio, intelligent bypassing.**

I. Introduction

Modern computer applications tend to have high memory requirements and require frequent, costly accesses to the main system memory. Due to the high latency associated with a main memory access, a multi-level cache serves as an intermediary between the processor cores and the main memory controller. The cache stores objects recently used by the processor but with significantly reduced access times and capacity compared to dynamic random-access memory. The cache provides the last opportunity for the processor to find an object without the high-latency cost associated with performing a main memory access. The cache is typically accessed sequentially, with the first-level cache being the smallest and fastest and the last-level cache being the largest and slowest to access.

This work was intended for submission as a final project report for ELE 548 Computer Architecture at the University of Rhode Island

As the cache has a significantly reduced capacity compared to main memory, algorithms determining how data blocks are placed into the cache and removed from it is a well-studied topic in computer architecture. These algorithms are referred to as cache replacement policies and are a relatively low-cost technique to considerably improve processor performance, as no changes to the underlying system architecture or processor design are required to implement them. Cache replacement policies are implemented at all levels of the cache and are usually different due to the memory capacity of each level.

Intelligent and effective last-level cache replacement policies are important for several reasons. The first and most obvious advantage of an effective last-level cache replacement policy are the latency savings gained from preventing unnecessary main memory accesses that may otherwise occur without a cache replacement policy that keeps frequently used blocks in the cache and discards seldom used ones. In modern times, most computer systems feature multiple processor cores on one chip. Multicore systems often designate the last-level cache as a shared cache that is common to each of the system's processor cores. This requires a policy that is not only effective for single core workloads but can respond effectively to the memory patterns of each of the system's threads that may have different access patterns for the workloads running on them.

Commonly used policies for cache replacement typically include those based on the Least-Recently Used (LRU) and Re-Reference Interval Prediction (RRIP) models. The LRU policy removes the block that sees the least recent use in each set in the cache. It is commonly used as a cache replacement policy for first-level caches due to its ease of implementation and low hardware budget required for caches with few sets. RRIP is a similar policy used as a base for many modern cache replacement policies. It closely approximates LRU,

though with a lesser hardware budget and slightly different logic used for eviction. This work will describe performance in terms of speedup over LRU for all discussions on performance improvements gained by improvements or changes to cache replacement policies due to its relatively simple implementation and the fact that it is considered a baseline for many other cache replacement policies.

This work seeks to explore cache replacement policies using intelligent mechanisms to detect reuse of cache blocks and successfully adapt to changing memory access patterns. Another goal of this work is to explore an intelligent mechanism capable of using cache bypassing to improve system performance to prevent blocks with little reuse potential from being placed in the last-level cache; however, this mechanism must be mindful of which blocks are to be bypassed so that unnecessary accesses to main system memory are avoided to reduce latency.

II. Previous Work

Several works have been published discussing the development and evaluation of adaptive cache replacement policies. These policies tend to have basic eviction policies that are based on existing policies such as LRU or RRIP, but with a modified insertion policy capable of using trends or information to place a block in an ideal insertion position.

The LRU Insertion Policy (LIP) is an example of a policy that attempts to solve the shortcomings of an existing policy with a novel, but simple change to the base policy [1]. In this case, it is the baseline LRU policy that is to be improved by the LRU Insertion Policy. LRU is a policy that performs well for cyclic memory access patterns; that is, a pattern that tends to re-reference the same set of blocks periodically. A cyclic memory access pattern can be the result of a loop-based program that tends to execute the same set of instructions continuously. However, the utility of LRU is largely restricted to cyclic access patterns, as frequent references to data objects without future reuse (called “scans”) can result in these objects taking up space in the cache without timely eviction to free up space for blocks that will see frequent reuse.

The LRU Insertion Policy assumes that most blocks in the last-level cache will not be re-used and that many blocks that will see reuse will see it immediately. This is partly due to lower-level caches filtering accesses to the last-level cache and preventing the last-level cache from making perfect use of temporal locality. To this end, LIP places all incoming blocks at the LRU position so that they will be immediately evicted. If a block sees immediate re-use before it is evicted, it will be promoted to the MRU position. In theory, LIP provides a form of scan resistance that is missing from the baseline LRU policy.

LIP derivative policies were proposed to add more improvements to the LRU policy [1]. The first of these derivative policies is known as Bimodal Insertion Policy (BIP). Bimodal Insertion Policy is a modification of LIP that places most blocks at the MRU position but will place a block at the LRU position with a low probability. The intent of BIP is to add a thrash resistant element to LRU. “Thrash” can be described as a phenomenon where a workload with cyclic memory access patterns and a very large working set greater than the capacity of the cache will see no blocks in the cache reused. This is because a large working set cycling in and out of the cache will inevitably lead to useful blocks being evicted before they are reused. BIP seeks to add some thrash resistance by randomly placing blocks deep in the LRU chain so that most blocks are quickly evicted, but some may persist long enough to be reused by a thrashing workload.

The last LIP variant is Dynamic Insertion Policy, a hybrid policy that attempts to combine the best aspects of LRU with BIP to achieve an LRU policy with intrinsic scan and thrash resistance for high performance during mixed access patterns [1]. Set dueling is used to determine which policy to be run for a given workload; that is, a small number of leader sets are allocated to run LRU, and an equal number are allocated to run BIP. A miss by either collection of sets will result in a global counter (PSEL, or policy selection) being incremented or decremented. The value of this PSEL counter will determine the global policy to be run for all follower sets.

One downside of LRU and its derivative policies is the high hardware budget required for keeping track of each block’s position on the LRU chain. For example, a 32-way set-associative cache requires five bits per way to implement an LRU chain, or 160 bits per set. In order to reduce hardware overhead and improve performance, the LRU chain can be approximated to be a form of hit predictor; that is, blocks lower on the chain closer to the MRU position can be assumed to be more likely to see future reuse and those closer to the LRU position are less likely. This is the reasoning behind the Re-Reference Interval Prediction (RRIP) model, which approximates the LRU chain to a smaller RRIP chain containing a Re-Reference Prediction Value (RRPV) for each block [2]. This chain is shortened, as a two or three-bit length is sufficient in most cases. In fact, an advantage of RRIP is its ability to scale well with highly associative caches, requiring only two or three bits for each block to implement the RRIP chain rather than a budget that scales with associativity like LRU. In this approximation, an RRPV of 0 is analogous to the position MRU and an RRPV of 2^n- 1 for an n-bit RRPV is analogous to the LRU position in LRU in that blocks with that value are eligible for eviction.

Similar improvements were proposed to the RRIP policy, analogous to LRU Insertion Policy and its derivatives that sought to add scan and thrash resistance to the baseline LRU policy [2]. It can be noted that LIP and its derivatives can be easily modified to adapt to an RRIP chain and seem to be better suited for one. In this case, a Static RRIP policy was considered the baseline policy in which all blocks were inserted at the $RRPV_{max} - 1$ position. Bimodal RRIP (BRRIP) and Dynamic RRIP (RRIP) were proposed as variants of RRIP and they are analogous to the BIP and DIP policies discussed.

Many modern cache replacement policies are based on RRIP chains and use more sophisticated insertion policies to adapt to changing memory accesses. One method of profiling the potential reuse or lack thereof from a block involves using the program counter (PC) of the instruction bringing each block in the cache as the index to a table that logs a value called "confidence". Several existing policies make use of a PC-based classifier mechanism to determine the insertion position of a cache block on an RRIP chain. [3][4][5][6]. This is an effective way of classifying block reuse that associates reuse with an instruction.

It is the goal of this work to offer a suite of improvements to the adaptive policies discussed and introduce a cache replacement policy capable of outperforming them. Combining a hybrid policy with an intelligent PC-aware reuse prediction mechanism to achieve a high-performance cache replacement policy capable of outperforming both Dynamic RRIP and LRU is the objective that this work seeks to achieve.

III. Methodology

In order to quantify performance of cache replacement policies, simulations of a CPU and cache architecture are required. To this end, the ChampSim trace-based CPU simulator is used to model the performance of cache replacement policies. The ChampSim simulator is a model of an out-of-order processor and is based on the Intel Core i7 architecture.

The Cache Replacement Championship 2 version of ChampSim is the version used. This version of ChampSim does not allow any modifications to any part of the simulator except for its L2 (in this case, the last-level cache) cache replacement policy. Four configurations are allowed for cache simulations; two configurations (1 and 3) allow for a single-core and quad-core simulation with two more configurations enabled the use of hardware prefetchers (2 and 4). For this work, simulations will be restricted to configurations without prefetchers. The architecture is fixed to a 16-way set-associative cache for all configuration modes. Configuration 1 assumes the use of a 2MB L2 cache and Configuration 3 assumes the use of an 8MB L2 cache. Figure 1 shows the simulation configurations used for this work.

ChampSim CRC2 Cache Configurations			
Configuration	Prefetchers?	# of cores	Cache size
Configuration 1	No	1	2MB L2 cache
Configuration 3	No	4	8MB shared L2 cache

Figure 1: ChampSim simulator configurations

For evaluating the performance of a cache replacement policy, traces based on SPEC2006 CPU benchmarks are used in concert with the ChampSim simulator. These benchmarks represent a set of common workloads used to measure and quantify CPU performance. For the purposes of this work, the performance of SPEC2006 CPU traces using LRU will serve as the baseline and performance will be quantified by using speedup over LRU.

IV. Reuse-Aware Re-Reference Interval Prediction

The many advantages of RRIP over LRU make it an ideal candidate for further improvements and modification over LRU. The reduced chain length both simplifies the logic and reduces the hardware budget required to implement a new policy, allowing some savings that permit the use of additional hardware structures that may not otherwise be possible due to hardware limitations. For this new policy, RRIP will be the base policy to be improved upon.

Successfully identifying reuse trends for cache blocks is an important objective of any intelligent cache replacement policy. Generally, there are few tools for a designer to use when attempting to identify a block and its reuse patterns when it is to be inserted into the cache. Using the PC value of the instruction as a is generally a good way to identify incoming blocks as it is linked to the instruction bringing the block into cache memory. Many existing policies use a PC signature to identify a specific instruction with a cache entry [3][4][5][6] Linking this PC value to a counter allows a satisfactory action (such as a cache hit) to increment a counter located in a table indexed by the PC. Conversely, a counter can be decremented if a negative action (such as a cache miss or eviction) occurs.

Reuse-Aware Re-Reference Interval Prediction (RA-RRIP) uses a PC-indexed to determine where a block should be placed in the RRIP chain. To reduce the amount of hardware required for an implementation, the confidence table is not indexed by the whole PC, but only by the lower 13 bits of the PC. Each entry in the confidence table contains a 4-bit confidence counter (CC) that keeps track of the degree of reuse experienced by each PC signature in the table. Saturating increment and decrement logic is used to increment and increment each CC entry.

Unlike policies such as SHiP, updates to the confidence table for RA-RRIP are not restricted to a select few sampling sets and all cache accesses may alter it. The confidence counter is incremented on a cache hit, which is indicative of a reuse. When a block is inserted into the cache, its associated CC is decremented. While decrementing on a cache insertion may seem counter-productive, note that a cache miss is equivalent to a cache miss and can be interpreted as a block seeing no reuse. While it may seem wiser to decrement this CC counter on a cache eviction rather than a cache fill, note that the hardware overhead required to store 13 bits of the PC associated with the block for eviction purposes would be high. Decrementing on a cache fill can be viewed as simply decrementing in anticipation of an eviction before it takes place, and while not equivalent, provides similar feedback.

For the RA-RRIP policy, the CC is initialized at a value of

7 (roughly half of the maximum value of a 4-bit CC). At the high end (CC>=10), a block can be considered to have high reuse potential and is inserted with high priority to the $RRPV_{max}$ -2 position, similarly to Bimodal RRIP's irregular insertion position. At the low end (CC<=4), a block can be considered to have low reuse potential and is inserted with reduced priority to the $RRPV_{max}$ position. Blocks with CC values in between 4 and 10 are inserted with regular priority to the $RRPV_{max}$ -1 position, or the same position used by Static RRIP for all blocks. Note that on a hit, the CC value of a block is incremented and its RRPV of the block is set to zero. The full policy is tabulated in Figure 2.

Reuse-Aware RRIP	
Condition	**Action**
Cache fill that is a writeback access	RRPV = $RRPV_{max}$, CC--
Cache fill with CC <= 4	RRPV = $RRPV_{max}$, CC--
Cache fill with CC >= 10	RRPV = $RRPV_{max}$ -2, CC--
All other cache fills	RRPV = $RRPV_{max}$ -1, CC--
A block receives a hit	RRPV = 0, CC++

Figure 2: A description of the RA-RRIP policy

Writebacks are cache blocks that are written to the last-level cache upon their eviction from lower-level caches. As they are not direct memory accesses from the CPU, they can be treated differently from regular cache accesses. The SHiP++ policy describes a writeback-specific bypassing mechanism in its insertion policy: SHiP++ considers writebacks to be the end of a context and it is described that should be bypassed entirely; however, they should be placed with maximum eviction priority in certain scenarios where writebacks cannot be bypassed [4]. RA-RRIP uses this same logic and designates writebacks as a separate access type from regular demand accesses and exempts them from CC-determined insertion priority. Instead, they are placed at the $RRPV_{max}$ position despite the value of their CC.

The RA-RRIP policy is intended to combine the scan-resistant features of Static RRIP with a mechanism capable of identifying a block's optimal insertion policy. The RA-RRIP policy is intended to have a high degree of scan resistance, but no specific functions add thrash resistance to this policy. As RRIP may be easily thrashed by a large working set, specific mechanisms must be included to keep a section of the working set in the cache to mitigate any concerns of thrashing.

V. Dynamic Reuse-Aware Re-Reference Interval Prediction

While RA-RRIP does provide adaptations to be made to the insertion position of blocks, it is inherently unfriendly to workloads with cyclic memory accesses. This is due to its default insertion policy placing new blocks to the end of the RRIP chain and these blocks seeing little reuse as a result. Unfortunately, this may lead to a condition where the lack of reuse is combined with negative feedback that forces these blocks to be inserted further to the end of the RRIP chain as a result of their lack of reuse. This results in a condition that closely mirrors the issue with the LRU Insertion Policy: memory accesses from cyclic workloads will not result in blocks persisting in the cache and they will be quickly evicted before reuse is possible.

A solution to this issue is a hybrid policy combining RA-RRIP with a policy that performs well for cyclic memory accesses. Several policies are designed as hybrid policies combining a novel policy with LRU to improve benchmarks with cyclic memory access patterns [1][2][7]. Due to RA-RRIP using an RRIP chain in lieu of an LRU-like chain, LRU cannot be used in conjunction with RA-RRIP; however, it is possible to approximate LRU with a policy known as LRU-like RRIP. This policy is rather simple and approximates LRU by inserting all incoming blocks with an RRPV of 0. In this scenario, LRU-like RRIP provides a mechanism for dealing with cyclic memory accesses that may not perform well with the base RA-RRIP. A relatively high number of sampling sets may increase the thrash resistance of DRA-RRIP, as this preserves part of the working set in the cache by running LRU-like RRIP permanently for these sets and allows it to be accessible despite a thrashing access pattern.

Dynamic RA-RRIP (DRA-RRIP) is a hybrid policy that combines the adaptive insertion policy of RA-RRIP with LRU-like RRIP to improve performance for workloads that see performance increases when LRU is used. This is akin to the Dynamic Insertion Policy and Dynamic RRIP implementations discussed. Set dueling is used in order to allow global policy selection from the performance of a select few leader sets. These leader sets will always be dedicated to running a single policy, in this case RA-RRIP or LRU-like RRIP. Upon a miss, a global PSEL counter is modified by the leader sets if a select few conditions are met. The DRA-RRIP set dueling policy is described in Figure 3.

Dynamic RA-RRIP, set dueling policy	
Condition	**Action**
A non-writeback miss with PC associated with reuse occurs at a leader set running LRU-like RRIP	Increment the PSEL counter
A non-writeback miss with PC associated with reuse occurs at a leader set running RA-RRIP occurs	Decrement the PSEL counter
A miss does not have the reuse bit set	Do not alter the PSEL counter
A miss is a writeback	Do not alter the PSEL counter
PSEL is greater than 512	Follower sets use RA-RRIP
PSEL is less than or equal to 512	Follower sets use LRU-like RRIP

Figure 3: A description of the DRA-RRIP set-dueling policy

Unlike the previously discussed set dueling implementations, some changes have been made to the set dueling mechanism used by Dynamic RA-RRIP. While all misses result in a PSEL update for Dynamic Insertion Policy and Dynamic RRIP, Dynamic RA-RRIP does not update the PSEL counter for two scenarios: 1) for a block associated with

a PC that has not experienced reuse and 2) for writebacks.

Blocks without reuse do not affect the PSEL counter as they are assumed to be scans. Scans do not see reuse and a miss on a scan is not considered to be a meaningful metric for cache performance. Therefore, a miss to a block associated with a reuse bit value of 0 does not update the PSEL counter.

A reuse table is kept and indexed by the same lower 13-bits of the PC used to index the confidence table. Each entry of the reuse table contains a 1-bit counter value called a reuse bit. When a block receives a hit, its corresponding entry in the reuse table is incremented. This indicates that blocks associated with an instruction at that PC have received reuse and that any block brought into the cache by that PC is unlikely to be a scan.

Writebacks do not modify the PSEL counter because writeback misses are to be expected as they signify the end of an access pattern. A writeback miss is considered insignificant and does not convey any importance information for cache performance.

VI. Bypassing Techniques for Cache Performance

If it can be determined with certainty that a block will not see future reuse, the ideal decision would be to not cache that block at all. In this case, the block is not placed in the cache at all, and no victim is selected. There is a significant performance advantage associated with performing a bypass: a block that may see reuse is given a spot in the cache that may otherwise be occupied by a block that will see no reuse. This may reduce the number of main memory accesses significantly if performed many times over the course of a single workload.

However, it is impossible to know for sure whether a block will see reuse in the cache. For that reason, bypassing must be performed conservatively. A poorly executed bypass can be dangerous as it may increase the probability of an unnecessary main memory access and the introduce significant latency to a workload. Assumptions about a block's future reuse must not be based on short-term performance but must instead be based on trends observed over an extended period. Several different mechanisms will be discussed to determine the conditions to perform a bypass.

VII. Simple Bypassing

The confidence counter (CC) gives a short-term history of the reuse patterns for a given block. As the CC uses only a 4-bit saturating counter, only short-term history can be assumed from its value. Predictable access patterns with many bursts of non-temporal data may benefit from this algorithm, as short-term history can predict future trends of the workload. A "0" or "1' CC value can indicate that a block has seen little to no reuse in the short term and can therefore be bypassed. For this type of bypassing, the CC is read for a given block upon victim selection. If the CC value is "0" for a given block, the block is bypassed. If the CC value is "1", then there exists a 70% chance that this block will be bypassed. When paired with the DRA-RRIP policy, bypassing cannot be performed during victim selection for a leader set, allowing the CC to be updated once again in the event it reaches zero for a given block. This allows learning of memory access patterns despite the fact that these low-confidence blocks are normally bypassed in a follower set. The simple bypassing policy is described in Figure 4.

Simple bypassing algorithm	
Condition	**Action**
CC = 0	Bypass 100% of the time
CC = 1	Bypass 70% of the time
CC = 0 or 1 but block is in a leader set	Do nothing

Figure 4: The simple bypassing algorithm

VIII. Ideal Bypassing

In an ideal scenario, a block that does not see reuse will always be bypassed and result in a performance gain. For a cache replacement policy, performance is typically expressed in terms of Instructions Per Cycle (IPC), or the number of instructions executed per clock cycle by the CPU for a given workload. If bypassing is working and a good fit for the workload at hand, then the IPC of a CPU using bypassing should in theory be greater than the IPC of a CPU not using bypassing.

This algorithm can be expressed by the simple finite state machine in Figure 5. It is centered around an algorithm that collects and compares the CPU's current instruction and cycle count to determine an IPC value. The algorithm initially runs for 10,000 cycles in a no bypassing state before transitioning to a bypassing state for another 10,000 cycles. The mode (bypassing or no bypassing) with the higher IPC value after these 20,000 cycles is run for an extended period. The IPC of the mode run during this extended period is then compared to the mode not selected. If the IPC is less than the value calculated during the initial 10,000 access period, then the mode is switched and run for 30,000 cycles; otherwise, the mode is not switched and run for another 30,000 cycles.

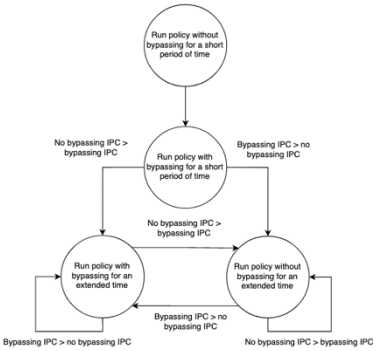

Figure 5: Transition between states for the ideal bypassing algorithm

Note that this type of cache replacement policy assumes that the cache has access to the CPU's internal state and can

perform near instantaneous floating-point division due to the differing access periods. It is a purely ideal method to determine the effectiveness of bypassing and should be used as a baseline to compare other bypassing algorithms to.

IX. Intelligent Bypassing

While ideal bypassing uses performance metrics derived from CPU state information normally unavailable to the cache to determine the effectiveness of bypassing, intelligent bypassing seeks to decide if bypassing should be enabled using only data that can be captured from cache accesses and metadata. Existing intelligent bypassing policies are conservative with their decision to bypass and tend to only bypass blocks without a potential for reuse [6][8]. To determine the suitability of bypassing, long-term data is required to assume that a block will likely see little reuse. In order to determine a workload's suitability for bypassing, a metric that will be referred to as "low confidence hit ratio" will be used.

Low confidence hit ratio refers to blocks with a confidence counter entry of 0 that do experience reuse once they reach this state. By incrementing a counter for every low-confidence access (that is, incrementing a counter every time a block with a confidence counter of 0 is inserted into the cache) and incrementing another counter for every low-confidence access that results in a hit, a hit ratio can be calculated. This hit ratio is simply the ratio of low-confidence hits to low-confidence misses for a given period of low-confidence accesses.

The intelligent bypassing algorithm will track these low-confidence accesses and low-confidence hits. After 10,000 low-confidence accesses, the number of hits will be compared to a threshold. If this number of hits exceeds a threshold, bypassing will not be performed, and another 10,000 low-confidence accesses will occur at which point the suitability for bypassing will be reassessed. If this hit count does not exceed a threshold, bypassing will be performed for 100,000 accesses, after which the suitability for bypassing will be reassessed once again. This algorithm is described with the finite state machine in Figure 6.

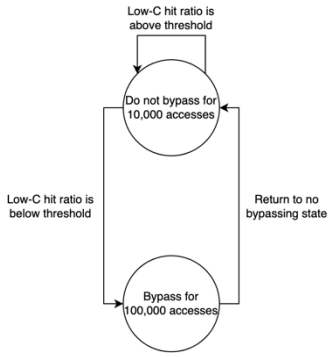

Figure 6: The intelligent bypassing algorithm as a simple finite state machine

Ideally, the threshold selected should be on the order of 1% of the access period. While this policy does not prevent all blocks with future reuse from being bypassed, it does attempt to limit the number of blocks with future reuse that are bypassed. This threshold was determined by simple analysis of two SPEC2006 CPU benchmarks: zeusmp, which performs poorly when bypassing is used, and libquantum, which performs well when bypassing is used. The low-confidence reuse statistics for these benchmarks is tabulated in Figure 7.

Benchmark	Low-confidence reuse (%) for DRA-RRIP	Speedup with simple bypassing (%) for DRA-RRIP
zeusmp	1.37	-8.17
libquantum	0.0151	10.3

Figure 7: Reuse and performance statistics for the zeusmp and libquantum benchmarks when run with DRA-RRIP

X. Multicore Policies for Shared Caches

As most computer systems now make use of multicore processors and shared last-level caches, it is important that cache replacement policies can adapt to these features in order to deliver high performance. Policies intended for last-level caches should be able to take advantage by noting the actions of multiple processor cores so that they are better able to manage the space in a shared cache.

Many policies have optimizations such as the ability to perform set dueling on a per-thread basis to better manage a shared cache [2][9]. An example of a multicore-optimized policy is Thread-Aware Dynamic Re-Reference Interval Prediction (TA-DRRIP), which is simply a variant of DRRIP that features set dueling on a per hardware thread basis rather than globally [2]. This is implemented by simply allocating a PSEL counter to each core so that each core may select a different policy for cache insertion, as each core may be running a different set of workloads that perform better under BRRIP or SRRIP. This way, thread-aware set dueling provides an advantage over global set-dueling that may select a single policy for several varying workloads.

Similarly, DRA-RRIP was developed to have a thread-aware variant: Thread-Aware DRA-RRIP (TA-DRA-RRIP). Like TA-DRRIP, set dueling occurs on a per thread basis so that each core may use either RA-RRIP or LRU-like RRIP depending on their workload. However, other aspects of the policy are also adapted for thread-awareness: the PC-indexed confidence and reuse tables are kept on a per-thread basis such that each processor core can update these tables independently, as each core keeps its own program counter and avoiding collisions on these tables will improve performance.

Bypassing is also performed on a per-thread basis for the ideal and intelligent cases. Due to its simple implementation, simple bypassing is by default a global policy that cannot be made thread aware. The counters used to keep the current bypassing state and access counters are both accessed on a per-thread basis for both cases so that bypassing can be performed by individual processor cores rather than performed globally.

XI. Analysis and Results

A. Determination of DRA-RRIP Leader Set Count

A series of simulations of the Dynamic RA-RRIP policy running with varying leader set counts were performed to collect empirical data to determine the optimal number of leader sets. The metric used to determine this optimal number was the resulting arithmetic mean IPC for a set of SPEC2006 CPU benchmarks. The ideal number of sets was determined to be a value between 64 and 128 leader sets, as they resulted in nearly identical performance. The results are graphed in Figure 8.

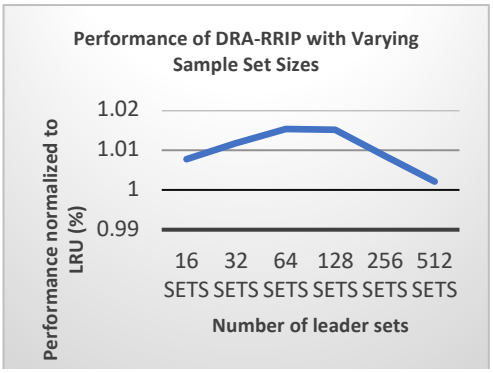

Figure 8: Average performance of DRA-RRIP with a varying leader set count

It was determined that 128 leader sets would be used for all simulations (512 for multi-core as cache size is scaled by four), as this count of leader sets resulted in better performance among benchmarks favoring LRU-like RRIP despite the nominally higher arithmetic mean IPC for simulations running with 64 leader sets as seen in Figure 9.

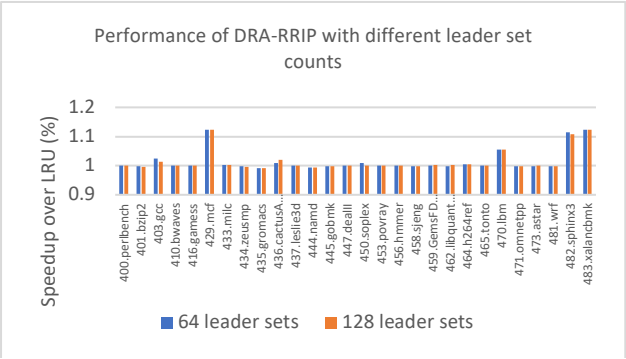

Figure 9: Performance of DRA-RRIP with 64 leader sets compared to DRA-RRIP with 128 leader sets for all SPEC2006 CP

B. Evaluation of LRU Insertion Policy and RRIP Variants

An objective of this work is to establish a baseline for the performance of the LRU Insertion Policy and the RRIP variants discussed by simulating them in the parameters set by the Cache Replacement Championship 2. To this end, these policies were simulated using ChampSim and the set of SPEC2006 CPU benchmark traces. These simulations were performed using the single-core Configuration 1, with a 2MB, 16-way set-associative L2 cache. In order to simplify the comparison between policies, the arithmetic mean of each policy's performance for all benchmarks as a function of speedup over LRU will be used to compare each policy's performance to one another. A probability of 31/32 was used for placing blocks at the LRU position for BIP (and conversely, p=1/32 for blocks to be placed at the MRU position). DIP was evaluated using two numbers of leader sets: 32 and 64. Similarly, Bimodal RRIP was modeled using the same probabilities as BIP and Dynamic RRIP was modeled with an identical count of leader sets. A 2-bit RRPV was used for all RRIP variants.

Figure 10 shows the average performance of LIP and its derivatives as modeled in the CRC2 framework. LIP and BIP show poor performance relative to LRU, with performance suffering by up to -1.96% for LIP and -1.62% for BIP. However, there is a very minor performance improvement associated with DIP, showing a 0.602% improvement over LRU for a 32-leader set implementation and a 0.572% improvement for a 64 leader set implementation.

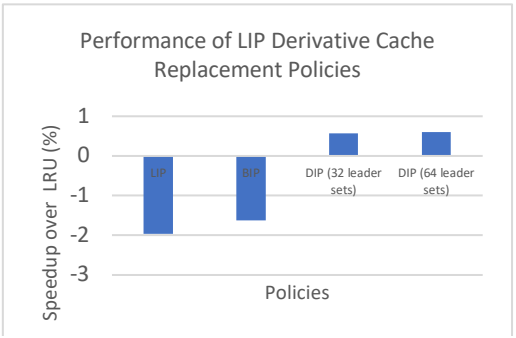

Figure 10: Performance of LIP Policies

Figure 11 shows the average performance of SRRIP, BRRIP, and DRRIP as modeled in the CRC2 framework. Despite the similarities of the RRIP policies to their LIP analogues, performance is much more consistent with the RRIP policies, with all policies performing roughly the same as LRU.

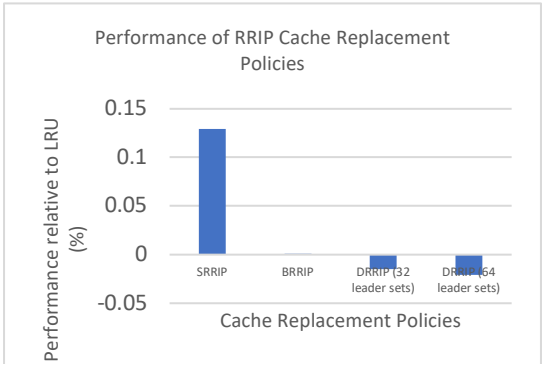

Figure 11: Performance of RRIP Policies

The shortened length of the RRIP chain over LRU and more forgiving insertion position of SRRIP over LIP may be to explain for RRIP's relative stability compared to LIP's volatility. Interestingly, it can be observed that DRRIP does not perform better than SRRIP or BRRIP in this simulation environment and performs slightly worse than LRU. As it is an adaptive policy, it will be used for comparisons against

RA-RRIP despite its relatively poor performance relative to SRRIP and BRRIP.

C. *Results for RA-RRIP and DRA-RRIP*

Single-core simulations were performed to quantify the performance of the base policy implementations for both RA-RRIP and DRA-RRIP without bypassing. These simulation results were compared to Dynamic Insertion Policy and Dynamic Re-Reference Interval Prediction as a measure of their performance. Figure 12 shows a comparison between DIP, DRRIP, RA-RRIP, and DRA-RRIP for all SPEC2006 CPU benchmarks.

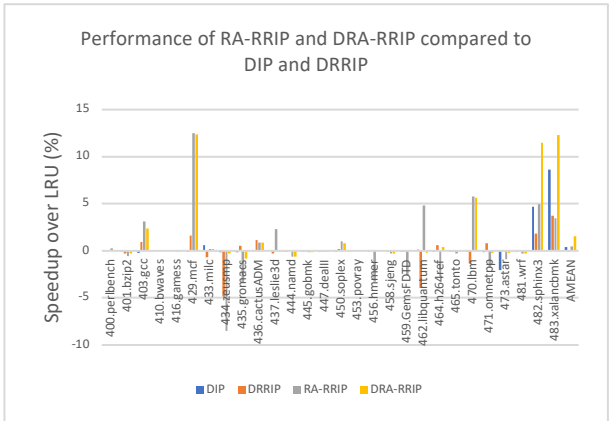

Figure 12: Comparison of DIP, DRRIP, RA-RRIP, and DRA-RRIP for all SPEC2006 CPU benchmarks

It can be observed that RA-RRIP offers minor performance improvements over LRU and DRRIP and performs very slightly better than DIP. However, DRA-RRIP offers a substantial performance increase over RA-RRIP, performing significantly better than all policies evaluated with a speedup of 1.53% over LRU. The DRA-RRIP policy manages to improve performance for several benchmarks that took a performance hit from the use of RA-RRIP by improving performance to near the LRU baseline. This is a result of the LRU-like RRIP component of DRA-RRIP, which approximates LRU for these benchmarks that tend to favor it. A comparison between RA-RRIP and DRA-RRIP for all benchmarks is shown in Figure 13.

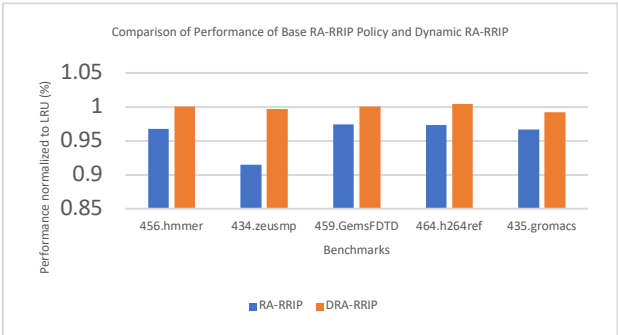

Figure 13: Performance of benchmarks that favor LRU cache replacement for both RA-RRIP and DRA-RRIP

As can be seen in Figure 12, RA-RRIP tends to have a lack of consistent performance when compared to LRU. DRA-RRIP seems to "fix" RA-RRIP by increasing performance to a level where all benchmarks perform at either near-LRU levels or exceed it. This improvement results in an overall increase in performance across the board for DRA-RRIP and improved average performance.

D. *Results for RA-RRIP with Bypassing Techniques*

Simulations were performed for both RA-RRIP and DRARRIP with bypassing enabled. The three different variants of bypassing discussed were implemented and run alongside a base policy of RA-RRIP or DRA-RRIP. The results of these simulations are graphed in Figure 14.

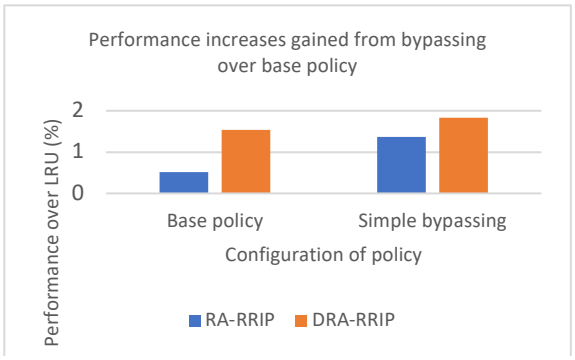

Figure 14: Comparison of performance between RA-RRIP and DRA-RRIP without bypassing and those policies with bypassing

Simple bypassing results in performance increases for both RA-RRIP and DRA-RRIP. The best performing configuration of the simple bypassing algorithm was the DRA-RRIP policy with simple bypassing, featuring a 1.825% speedup over LRU.

The ideal bypassing algorithm was evaluated using simulations to test three different access periods for an extended access period: 30,000, 50,000, and 70,000. The ideal bypassing algorithm was added to both the RA-RRIP and DRA-RRIP policies for this simulation. The results are graphed below in Figure 15.

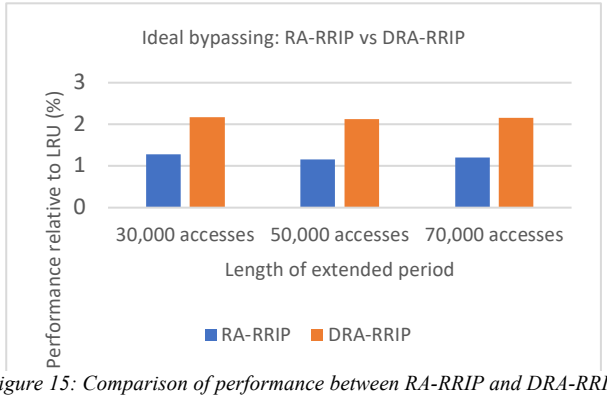

Figure 15: Comparison of performance between RA-RRIP and DRA-RRIP using varying extended access periods for an ideal bypassing algorithm.

The 30,000 extended access period performed the best of the three evaluated access periods for both RA-RRIP and DRA-RRIP. The best performing configuration was an ideal bypassing DRA-RRIP policy with a 30,000-access extended access period with a 2.18% speedup over LRU.

Simulations were performed for the intelligent bypassing algorithm testing three different thresholds for enabling the bypassing component of the algorithm: 0.5%, 1%, and 2%. These simulations were performed for variants of the intelligent bypassing algorithm using both RA-RRIP and DRA-RRIP as a base policy. The results of these simulations are graphed in Figure 16.

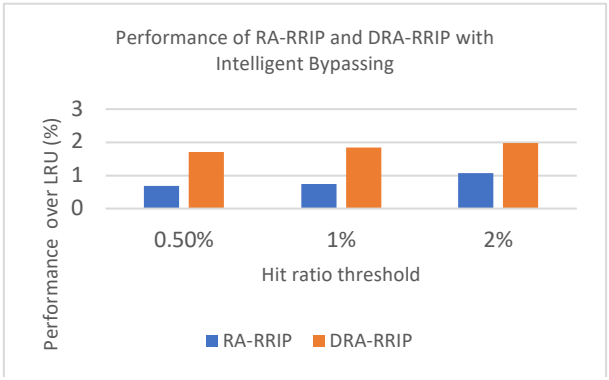

Figure 16: Comparison of intelligent bypassing algorithm implementations for both RA-RRIP and DRA-RRIP with varying bypass thresholds

The 2% threshold was determined to be the most effective for both RA-RRIP and DRA-RRIP. The most effective configuration tested was the 2% threshold with the DRA-RRIP base policy, performing 1.98% better than LRU.

The average performance of DRA-RRIP for its base case, the 30,000-access ideal bypassing case, and the 2% threshold intelligent bypassing case are compared to the Hawkeye and SHiP++ policies in Figure 17. Hawkeye and SHiP++ are state of the art, high-performance cache replacement policies in use today.

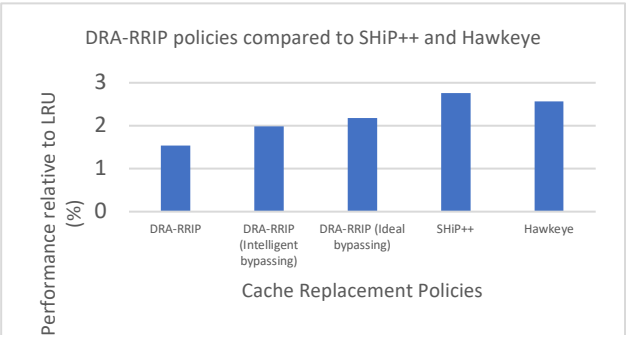

Figure 17: Graphed average performance for DRA-RRIP cache replacement policies compared to Hawkeye and SHiP++

It can be observed that Hawkeye and SHiP++ outperform all variants of DRA-RRIP, with the best DRA-RRIP variant (ideal bypassing) performing worse than SHiP++ and Hawkeye in terms of speedup over LRU. While DRA-RRIP with ideal bypassing performed 2.18% faster than LRU, SHiP++ performed 2.76% faster than LRU and Hawkeye performed 2.56% faster than LRU.

E. Results for Multicore Simulations

Multicore simulations were performed for Thread-Aware DRA-RRIP and its variants and compared to several other policies for two sets of benchmarks: a set of multicore Cloudsuite traces that represent server workloads and groups of SPEC2006 CPU benchmarks. Figure 18 shows the performance of these benchmarks for the Cloudsuite benchmarks. Note that the best-performing configuration for each bypassing algorithm for single-core simulations was carried over for multicore simulations.

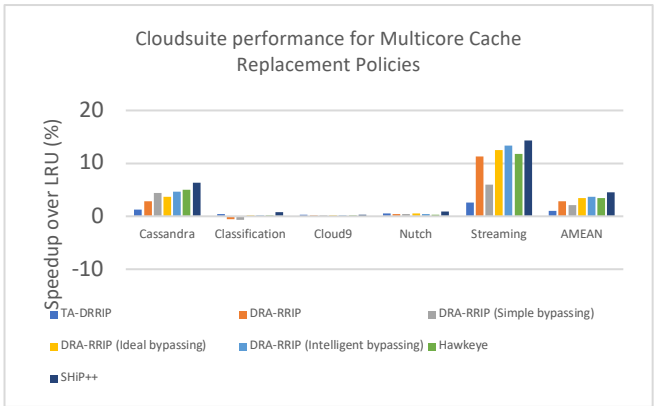

Figure 18: Performance of several cache replacement policies for Cloudsuite benchmarks

It can be observed from this graph that all TA-DRA-RRIP variants outperform TA-DRRIP (which performed 1.03% faster than LRU), with the intelligent bypassing variant of TA-DRA-RRIP being the best-performing one with a 3.74% speedup over LRU and a 2.71% speedup over TA-DRRIP. The intelligent bypassing variant of DRA-RRIP outperforms Hawkeye's speedup of 3.47% over LRU but is in turn outperformed by SHiP++, which features a speedup of 4.52% over LRU.

A set of simulations using multicore SPEC2006 CPU benchmarks was also used for evaluating multicore performance. These simulations were crafted such that each core would run a unique SPEC2006 CPU benchmark for a total of four benchmarks per simulation. The average performance of TA-DRA-RRIP and its variants was compared to the average performance of TA-DRRIP, SHiP++, and Hawkeye in Figure 19. Note that TA-DRRIP with 128 leader sets was used.

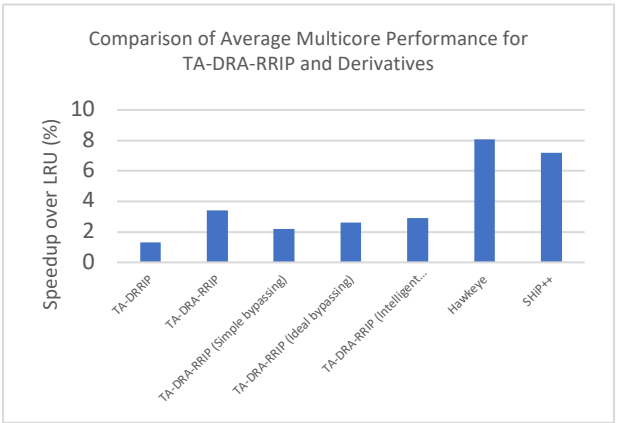

Figure 19: Average performance of several cache replacement policies, including DRA-RRIP and derivatives for multicore SPEC2006 CPU benchmarks.

It can be observed that the base TA-DRA-RRIP policy without bypassing outperforms TA-DRRIP (which performed 1.32% faster than LRU)and all other TA-DRA-RRIP variants at 3.45% speedup over LRU. TA-DRA-RRIP performs better than TA-DRRIP and LRU, but worse than Hawkeye and SHiP++ with speedups of 7.17% and 8.07% respectively.

From these simulations, it can be observed that TA-DRA-RRIP and its derivatives offer improvements over LRU and TA-DRRIP but perform poorly compared to today's state of the art cache replacement policies for SPEC2006 CPU benchmarks. However, despite the relatively poor performance for those benchmarks, superior performance over Hawkeye and slightly inferior performance to SHiP++ was achieved for the Cloudsuite workloads.

XII. Hardware Considerations

The hardware budget of DRA-RRIP and its variants can be calculated by summing the budgets of each hardware component required for implementation. A single core implementation of DRA-RRIP uses a single 32K-block RRIP chain, a single reuse table with 16K entries, a single confidence table with 16K entries, and a 10-bit PSEL counter. Each entry of the RRIP chain requires two bits for a total of 64K bits. The reuse table contains one bit per entry and the confidence table requires four bits entry for a total hardware requirement of 16K bits and 64K bits respectively. This requires a total of 18KB for implementation, along with two additional bytes to store the PSEL counter. DRA-RRIP's hardware requirement scales by four for the multi-core 8MB cache implementation. Note that use of the simple bypassing algorithm requires no additional hardware over base DRA-RRIP.

The intelligent bypassing algorithm requires relatively little hardware overhead for implementation compared to the base DRA-RRIP policy. Only two counters are required for implementation of this policy: a 14-bit counter for counting low-confidence misses and a 17-bit counter for keeping track of the current number of low-confidence accesses to the cache. In total, intelligent bypassing requires an additional four bytes for implementation, or 16 for multicore. The total hardware budget for each policy and configuration is tabulated in Figure 20.

Policy	Single core 2MB	Multicore 8MB
LRU	16KB	64KB
DRRIP	8KB + 2B	32KB + 2B
DRA-RRIP	18KB + 2B	73.728KB + 5B
DRA-RRIP (Intelligent Bypassing)	18KB + 6B	73.728KB + 21B
Hawkeye	31.8KB	90.2KB
SHiP++	16KB	76.25KB

Figure 20: Hardware budgets required for implementation of several cache replacement policies for a 16-way set-associative cache of two sizes

XIII. Future Work

The single core performance of RA-RRIP leaves much to be desired and work is required if it is to be developed as a standalone policy without set dueling. Its similarities to the SHiP policy suggest that this type of improvement is within reach by using a similar set sampling method for updating the confidence table, though any further SHiP-like additions to the policy would result in RA-RRIP becoming a derivative of SHiP. However, the intelligent bypassing algorithm may serve as a useful addition for SHiP++ and should be explored as a method to improve other cache replacement policies. Further improvement on its algorithm should be explored and sandboxing can be used to further improve its utility.

Although the RA-RRIP and DRA-RRIP policies offered performance increases over the baseline cache replacement policies and performed only slightly worse than state of the art cache replacement policies for single-core simulations, there was an extreme disparity in performance between DRA-RRIP and state of the art policies for the multicore simulations using SPEC2006 CPU benchmarks. Improvements on DRA-RRIP's multicore performance must be made for it to be a useful policy for modern processors, as virtually all last-level caches are shared between multiple cores on these systems.

XIV. Conclusions

RA-RRIP and DRA-RRIP offer a significant performance improvement over LRU and DRRIP for both single and multicore simulations. A reuse-aware insertion policy as implemented seems to perform well for some benchmarks, but not as well for benchmarks favoring LRU-like cache replacement, though a hybrid policy with LRU-like RRIP restores LRU-level performance for these benchmarks. However, they fail to outperform state of the art cache replacement policies in most benchmarks and more work is required to develop these techniques.

Bypassing techniques have the potential to improve performance further at little hardware overhead cost. Although not all benchmarks benefit from bypassing, high performance gains can be attained from performing bypasses when appropriate. Algorithms using ideal performance-based bypassing and intelligent reuse-aware bypassing may detect the appropriate time to perform a bypass from cache activity and improve performance significantly.

XV. References

1. Qureshi, Moinuddin & Jaleel, Aamer & Patt, Yale & Jr, Simon & Emer, Joel. (2007). Adaptive insertion policies for high performance caching. ACM Sigarch Computer Architecture News. 35. 381-391. 10.1145/1273440.1250709. A. Jaleel, K. B. Theobald, S. C. Steely Jr. and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)", *Proc. 37th Annu. Int. Symp. Comput. Archit.*, pp. 60-71, 2010.

2. A. Jaleel, K. B. Theobald, S. C. Steely Jr. and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)", *Proc. 37th Annu. Int. Symp. Comput. Archit.*, pp. 60-71, 2010.

3. C. -J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely and J. Emer, "SHiP: Signature-based Hit Predictor for high performance caching," 2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Porto Alegre, Brazil, 2011, pp. 430-441.

4. V. Young, C.-C. Chou, A. Jaleel, and M. Qureshi, "Ship++: Enhancing signature-based hit predictor for improved cache performance," In: International Workshop on Cache Replacement Championship, co-located with ISCA. CRC2. http ://crc2.ece.tamu.edu. June 2017

5. Jain, Akanksha & Lin, Calvin. (2016). "Hawkeye: Leveraging Belady's Algorithm for Improved Cache Replacement." . In: International Workshop on Cache Replacement Championship, co-located with ISCA. CRC2. http ://crc2.ece.tamu.edu. June 2017

6. J. Díaz, P. Ibáñez, T. Monreal, V. Viñals, and J. Llabería. "ReD: A Policy Based on Reuse Detection for a Demanding Block Selection in Last-Level Caches" In: International Workshop on Cache Replacement Championship, co-located with ISCA. CRC2. http ://crc2.ece.tamu.edu. June 2017

7. M. K. Qureshi, D. N. Lynch, O. Mutlu and Y. N. Patt, "A Case for MLP-Aware Cache Replacement," 33rd International Symposium on Computer Architecture (ISCA'06), Boston, MA, USA, 2006, pp. 167-178, doi: 10.1109/ISCA.2006.5.

8. M. Kharbutli and Y. Solihin, "Counter-Based Cache Replacement and Bypassing Algorithms," in IEEE Transactions on Computers, vol. 57, no. 4, pp. 433-447, April 2008, doi: 10.1109/TC.2007.70816.

9. Jaleel, Aamer & Hasenplaugh, William & Qureshi, Moinuddin & Sébot, Julien & Jr, Simon & Emer, Joel. (2008). Adaptive insertion policies for managing shared caches. Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT. 208-219. 10.1145/1454115.1454145.