Using Dead Blocks as a Virtual Victim Cache

Samira Khan† Daniel A. Jiménez† Doug Burger‡ Babak Falsafi§

† Dept. of Computer Science
University of Texas at San Antonio
{skhan, dj}@cs.utsa.edu

‡Microsoft Research
dburger@microsoft.com

§Parallel Systems Architecture Lab
Ecole Polytechnique Fédérale de Lausanne
babak.falsafi@epfl.ch

Abstract

Caches mitigate the long memory latency that limits the performance of modern processors. However, caches can be quite inefficient. On average, a cache block in a 2MB L2 cache is *dead* 59% of the time, i.e., it will not be referenced again before it is evicted. Increasing cache efficiency can improve performance by reducing miss rate, or alternately, improve power and energy by allowing a smaller cache with the same miss rate.

This paper proposes using predicted dead blocks to hold blocks evicted from other sets. When these evicted blocks are referenced again, the access can be satisfied from the other set, avoiding a costly access to main memory. The pool of predicted dead blocks can be thought of as a *virtual victim cache*. A virtual victim cache in a 16-way set associative 2MB L2 cache reduces misses by 11.7%, yields an average speedup of 12.5% and improves cache efficiency by 15% on average, where cache efficiency is defined as the average time during which cache blocks contain live information. This virtual victim cache yields a lower average miss rate than a fully-associative LRU cache of the same capacity. Using an adaptive insertion policy, the virtual victim cache gives an average speedup of 17.3% over the baseline 2MB cache.

The virtual victim cache significantly reduces cache misses in multi-threaded workloads. For a 2MB cache accessed simultaneously by four threads, the virtual victim cache reduces misses by 12.9% and increases cache efficiency by 16% on average

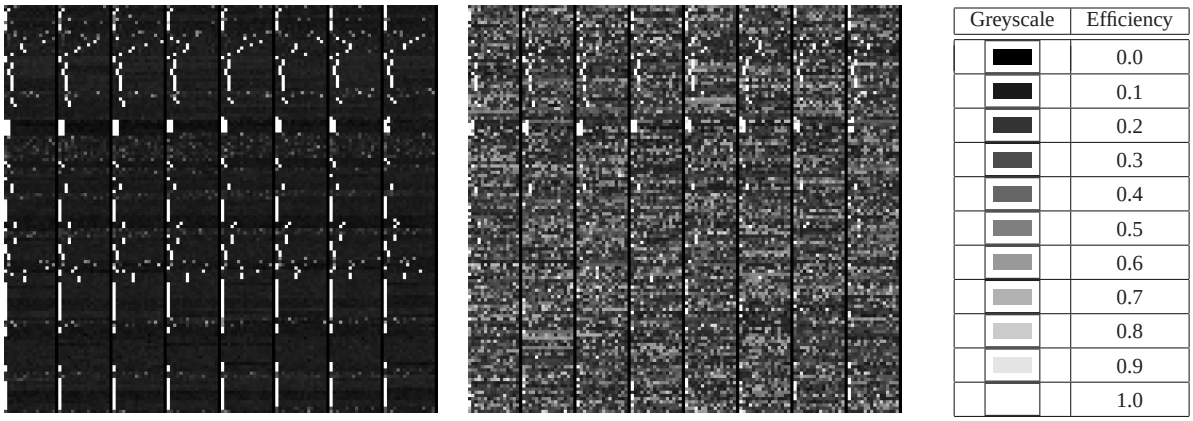
Alternately, a 1.7MB virtual victim cache achieves about the same performance as a larger 2MB L2 cache, reducing the number of SRAM cells required by 16%, thus maintaining performance while reducing power and area.

1 Introduction

The performance gap between modern processors and memory is a primary concern for computer architecture. Processors have large on chip caches and can access a block in just a few cycles, but a miss that goes all the way to memory incurs hundreds of cycles of delay. Thus, reducing cache misses can significantly improve performance.

One way to reduce the miss rate is to increase the number of *live* blocks in the cache. A cache block is live if it will be referenced again before its eviction. From the last reference until the block is evicted the block is *dead* [10]. Studies show that cache blocks are dead most of the time; for the benchmarks and 2MB L2 cache used for this study, cache blocks are dead on average 59% of the time. Dead blocks lead to poor cache efficiency [12, 3] because after the last access to a block, it resides a long time in the cache before it is evicted. In the least-recently-used (LRU) replacement policy, after the last access, every block has to move down from the MRU position to the LRU position and then it is evicted. Cache efficiency can be improved by replacing dead blocks with live blocks as soon as possible after a block becomes dead, rather than waiting for it to be evicted. Having more live blocks in the same size of cache improves the system performance by reducing miss rate; more live blocks means more cache hits. Alternately, a technique that increases the number of live blocks may allow reducing the size of the cache, resulting in a system with the same performance but reduced power and energy needs.

This paper describes a technique to improve cache performance by using predicted dead blocks to hold victims from cache evictions in other sets. The pool of predicted dead blocks can be thought of as a *virtual victim cache* (VVC). Figure 1 graphically depicts the efficiency of a 1MB 16-way set associative L2 cache with LRU replacement for the SPEC CPU 2006 benchmark `456.hmmer`. The amount of time each cache block is live is shown a as a greyscale intensity. Figure 1(a) shows the unoptimized cache. The darkness shows that many blocks remain dead for large stretches of time. Figure 1(b) shows the same cache optimized with the VVC idea. Now many blocks have more live time so the cache is more efficient.



(a) (b)

Figure 1: Virtual victim cache increases cache efficiency. Block efficiency (i.e., fraction of time block is live) shown as greyscale intensities for `456.hmmer` for (a) a baseline 1MB cache and (b) a VVC-enhanced cache; darker blocks are dead longer.

The VVC idea uses a dead block predictor, i.e., a microarchitectural structure that uses past history to predict whether a given block is likely to be dead at a given time. This study uses a trace based dead bock predictor [10].

This idea has some similarity to the victim cache [6], but victims are stored in the same cache from which they were evicted, simply moving from one set to another. When a victim block is referenced again, the access can be satisfied from the other set. Another way to view the idea is as an enhanced combination of block insertion (i.e. placement) policy, search strategy, and replacement policy. Blocks are initially placed in one set and migrated to less a active set when they become least-recently-used. A more active block is found with one access to the tag array, and a less active block may be found with an additional search.

1.1 Contributions

This paper explores the idea of placing victim blocks into dead blocks in other sets. This strategy reduces the number of cache misses per thousand instructions (MPKI) by 11.7% on average with a 2MB L2 cache, yields an average speedup of 12.5% over the baseline and improves cache efficiency by 15% on average. The VVC outperforms a fully associative cache with the same capacity; thus, VVC does not simply improve performance by increasing associativity. The VVC also outperforms a real victim cache that uses the same additional hardware budget as the VVC structures, e.g. the predictor tables. It also provides an improvement for multi-threaded workloads for a variety of cache sizes.

This paper introduces a new dead block predictor organization inspired by branch predictors. This organization reduces harmful false positive predictions by over 10% on average, significantly improving the performance of the VVC with the potential to improve other optimizations that rely on dead block prediction.

The VVC idea includes a block insertion policy driven by cache evictions and dead block predictions. Using an adaptive insertion policy, the VVC gives an average speedup of 17.3% over the baseline 2MB cache, and 4% over the baseline cache enhanced with a dynamic insertion policy (DIP) from previous work.

2 Related Work

In this section we discuss related work. Previous work introduced several dead block predictors and applied them to problems such as prefetching and block replacement [10, 12, 8, 5, 1], but did not explore coupling dead block prediction with alternative block placement strategies.

2.1 Dead Block Predictors

The VVC depends on an accurate means of determining which blocks are dead and thus candidates to replace with victims from other sets. We discuss related work in dead block prediction.

2.1.1 Trace Based Predictor

The concept of a Dead Block Predictor (DBP) was introduced by Lai *et al.* [10]. The main idea is that, if a given sequence of accesses to a given cache block leads to the death (i.e. last access) of the block, then that same sequence of accesses to a different block is likely to lead to the death of that block. An access to a cache block is represented by the program counters (PC) of the instructions making the access. The sequence, or *trace* of PCs of the instructions accessing a block are encoded as the fixed-length truncated sum of hashes of these PCs. This trace encoding is called a *signature*. For a given cache block, the trace for the sequence of PCs begins when the block is refilled and ends when the block is evicted. The predictor learns from the trace encoding of the evicted blocks. A table of saturating counters is indexed by block signatures. When a block is replaced, the counter associated with it is incremented. When a block is accessed, the corresponding counter is decremented. A block is predicted dead when the counter corresponding to its trace exceeds a threshold.

This dead block predictor is used to prefetch data into predicted dead blocks in the L1 data cache, enabling lookahead prefetching and eliminating the necessity of prefetch buffers. That work also proposes a dead block correlating prefetcher that uses address correlation to determine which block to prefetch in the dead blocks.

A trace based predictor is also used to optimize a cache coherence protocol [9, 20]. Dynamic self-invalidation involves another kind of block "death" due to coherence events [11]. PC traces are used to detect the last touch and invalidate the shared cache blocks to reduce cache coherence overhead.

2.1.2 Counting Based Predictor

Dead blocks can also be predicted depending on how many times a block has been accessed. Kharbutli and Solihin propose a counting based predictor for an L2 cache where a counter for each block records how many times the block has been referenced [8]. When a block is evicted the history table stores the reference count and the first PC that brought that block into the cache. When a block is brought into cache again by the same PC the dead block predictors predicts it to be dead after the number of references reaches the threshold value stored in the history table. Kharbutli and Solihin use this counter based dead block predictor to improve the LRU replacement policy [8]. This improved LRU policy replaces a dead block if available, the LRU block if not. Our technique also replaces predicted dead blocks with other blocks, but the other blocks are victims from other sets, effectively extending associativity in the same way a victim cache does.

2.1.3 Time Based Predictor

Another approach of dead block prediction is to predict a block dead when it is not accessed for a certain number of cycles. Hu *et al.* proposed a time based dead block predictor [5]. It learns the number of cycles a block is live and predicts the block dead if it is not accessed more than twice the number of cycles that it had been live. This predictor is used to prefetch data into the L1 cache. This work also proposed using dead times to filter blocks in the victim cache. Blocks with shorter dead times are likely to be reused before getting evicted from the victim cache, so time base victim cache stores only blocks that are likely to be reused.

Abella *et al.* propose another time based predictor [1]. It also predicts a block dead if it has not been accessed for a certain number of cycles. But here the number of cycles is calculated from the number of accesses of that block. Abella *et al.* reduce cache leakage for L2 cache by dynamically turning off cache blocks whose content is not likely to be reused without hurting the performance.

2.1.4 Cache Burst Predictor

Cache bursts [12] can be used with trace based, counting based and time based dead block predictors. A cache burst consists of all the contiguous accesses that a block receives while in the most-recently-used (MRU) position. Instead of each individual references, cache burst based predictor updates the predictor only on each bursts. It also improves prediction accuracy by making prediction only when a block moves out of the MRU position. The

dead block predictor needs to store trace or reference count information for each burst only rather than for each reference. But since prediction is made only after a block becomes non MRU, some of the dead time is lost compared to non burst predictors. Cache burst predictors improve prefetching, bypassing and enhancing LRU replacement policy both for L1 Data cache and L2 cache.

2.1.5 Other Dead Block Predictors

Another kind of dead block prediction involves predicting in software [22, 18]. In this approach the compiler collects dead block information and provides hints to the microarchitecture to make cache decisions. If a cache block is likely to be reused again it hints to keep the block in the cache; otherwise, it hints to evict the block.

2.2 Cache Placement and Replacement Policy

Adaptive insertion policy [15] adaptively inserts the incoming lines in the MRU position when the working size becomes larger than the cache size; we explore this idea in more detail in Section 7. Keramidas *et al*. [7] proposed a cache replacement policy that uses reuse distance prediction. This policy tries to evict cache blocks that will be reused furthest in the future. A memory-level parallelism aware cache replacement policy relies on the fact that isolated misses are more costly on performance than parallel misses [16].

3 Using Dead Blocks as a Virtual Victim Cache

Victim caches [6] work well because they effectively extend the associativity of any hot set in the cache, reducing localized conflict misses. However, victim caches must be small because of their high associativity, and are flushed quickly if multiple hot sets are competing for space. Thus, victim caches do not reduce capacity misses appreciably, nor conflict misses where the reference patterns do not produce a new reference to the victim quickly, but they provide excellent miss reduction for a small additional amount of state and complexity. Larger victim caches have not come into wide use because any additional miss reduction benefits are outweighed by the overheads of the larger structures.

Large caches already contain significant quantities of unused state, however, which in theory can be used for optimizations similar to victim caches if the unused state can be identified and harvested with sufficiently low overhead. Since the majority of the blocks in a cache are dead at any point in time, and since dead-block predictors have been shown to be accurate in many cases, the opportunity exists to replace these dead blocks with victim blocks, moving them back into their set when they are accessed. This *virtual victim cache* approach has the potential to reduce both capacity misses and additional conflict misses: Capacity misses can be reduced because dead blocks are evicted before potentially live blocks that have been accessed less recently (avoiding misses that would occur with full associativity), and conflict misses can be further reduced if hot set overflows can spill into other dead regions of the cache, no matter how many hot sets are active at any one time.

An important question is how the dead blocks outside of a set are found and managed without adding prohibitive overhead. By coupling small numbers of sets, and moving blocks overflowing from one set into the predicted dead blocks (which we call *receiver blocks*) of a ”partner set,” a virtual victim cache can be established with little additional overhead. While this approach effectively creates a higher-associativity cache, the power overheads are kept low because only the original set is searched the majority of the time, with the partner sets only searched upon a miss in the original set. The overheads include more tag bits (the log of the number of partner sets) and more energy and latency incurred on a cache miss, since the partner sets are searched to no avail.

3.1 Description of the VVC

3.1.1 Identifying Potential Receiver Blocks

A trace based dead block predictor keeps a trace encoding for each cache block. The trace is updated on each use of the block. When a block is evicted from the cache, a saturating counter associated with that block’s trace is incremented. When a block is used, the counter is decremented.

Ideally, any victim block could replace any receiver block in the entire cache, resulting in the highest possible usage of the dead blocks as a virtual victim cache. However, this idea would increase the dead block hit latency

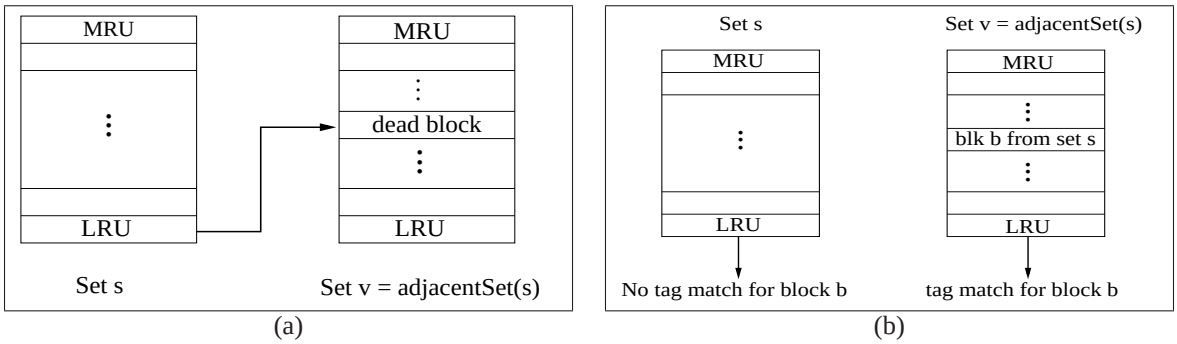


Figure 2: (a) Placing evicted block into an adjacent partner set, and (b) hitting in the virtual victim cache

and energy as every set in the cache would have to be searched for a hit. Thus, there is a trade-off between the number of sets that can store victim blocks from a particular set and the time and energy needed for a hit. We have determined that, for each set, considering only one other partner set to identify a receiver block yields a reasonable balance. Sets are paired into *adjacent sets* that differ in their set indices by one bit.

3.1.2 Placing Victim Blocks into the Adjacent Set

When a victim block is evicted from a set, the adjacent set is searched for invalid or predicted dead receiver blocks. If no such block is found, then the LRU block of the adjacent set is used. Once a receiver block is identified, the victim block replaces it. The victim block is placed into the most-recently-used (MRU) position in the adjacent set. This MRU insertion policy is not always best; Section 7 explores an adaptive insertion policy that chooses between the LRU and MRU position.

3.1.3 Block Identification in the VVC

If a previously evicted block is referenced again, the tag match will fail in the original set, the adjacent set will be searched, and if the receiver block has not yet been evicted then the block will be found there. The block will then be refilled in the original set from the adjacent set, and the block in the adjacent set will be marked as invalid. A small penalty for the additional tag match and fill will accrue to this access, but this access is considered a hit in the L2 for purposes of counting hits and misses (analogously, an access to a virtually-addressed cache following a TLB miss may still be considered a hit, albeit with an extra delay).

To distinguish receiver blocks from other blocks, we keep an extra bit with each block that is true if the block is a receiver block, false otherwise. When a set's tags are searched for a normal cache access, receiver blocks from the adjacent set are prevented from matching to maintain correctness. Note that keeping this extra bit is equivalent to keeping an extra tag bit in a higher associativity cache.

Figure 2(a) shows what happens when a LRU block is evicted from a set s. If the adjacent set v has any predicted dead or invalid block in it, the victim block replaces that block, otherwise the LRU block of set v is used. Similarly, Figure 2(b) depicts a VVC hit. If the access results in a miss in the original set s, that block can be found in a receiver block of the adjacent set v. Algorithm 1 shows the complete algorithm for the VVC.

3.1.4 Caching Predicted Dead Blocks

Note that blocks that are predicted dead and evicted from a set may be cached in the VVC. Although it might seem counterintuitive to replace one dead block with another dead block, this policy does give an advantage over simply discarding predicted dead blocks because the predictor might be wrong, i.e. one or both blocks might not be dead. We favor the block from the hotter set, likely to be the set just accessed.

Algorithm 1 Virtual Victim Cache with Trace based Predictor

```
On an access to set s with address a, PC pc
if the access is a hit in block blk then
   blk.trace ← updateTrace(blk.trace, pc)
   isDead ← lookupPredictor(blk.trace)
   if isDead then
      mark blk as dead
      return
   end if
else /* search adjacent set for a dead block hit */
   v = adjacentSet(s)
   access set v with address a
   if the access is a hit in a dead block dblk then
      bring dblk back into set s
      return
   end if
else /* this access is a miss */
   repblk ← block chosen by LRU policy
   updatePredictor(repblk.trace)
   v = adjacentSet(s)
   place repblk in an invalid/dead/LRU block in set v
   place block for address a into repblk
   repblk.trace ← updateTrace(pc)
   return
end if
```

3.2 Implementation Issues

Adjacent sets differ in one bit, bit k. The set adjacent to set index s is s exclusive-ORed with 2^k. A value of $k = 3$ provides good performance, although performance is largely insensitive to the choice of k. Victims replace receiver blocks in the MRU position of the adjacent set and are allowed to be evicted just as any other block in the set. Evicted receiver blocks are not allowed to return to their original sets, i.e., evicted blocks may not "ping-pong" back and forth between adjacent sets.

Each cache block keeps the following additional information: whether or not it is a receiver block (1 bit), whether or not the block is predicted dead (1 bit), and the truncated sum representing the trace for this block (14 bits). The dead block predictor additionally keeps two tables of two-bit saturating counters indexed by traces. The predictor tables consume an additional 2^{14} entries $\times 2$ bit counters $\times 2$ tables = 64 kilobits, or 8 kilobytes.

4 Skewed Dead Block Predictor

In this section we discuss a new dead block predictor based on the reference trace predictor of Lai *et al.* [10] as well as skewed table organizations [19, 13].

4.1 Reference Trace Dead Block Predictor

The reference trace predictor collects a trace of the instructions used to access a particular block. The theory is that, if a sequence of memory instructions to a block leads to the last access of that block, then the same sequence of instructions should lead to the last access of other blocks. The reference trace predictor encodes the path of memory access instructions leading to a memory reference as the truncated sum of the instructions' addresses. This truncated sum is called a *signature*. Each cache block is associated with a signature that is cleared when that cache block is filled and updated when that block is accessed. The signature is used to access a table of two-bit saturating counters. When a block is accessed, the corresponding counter is decremented and then the signature is

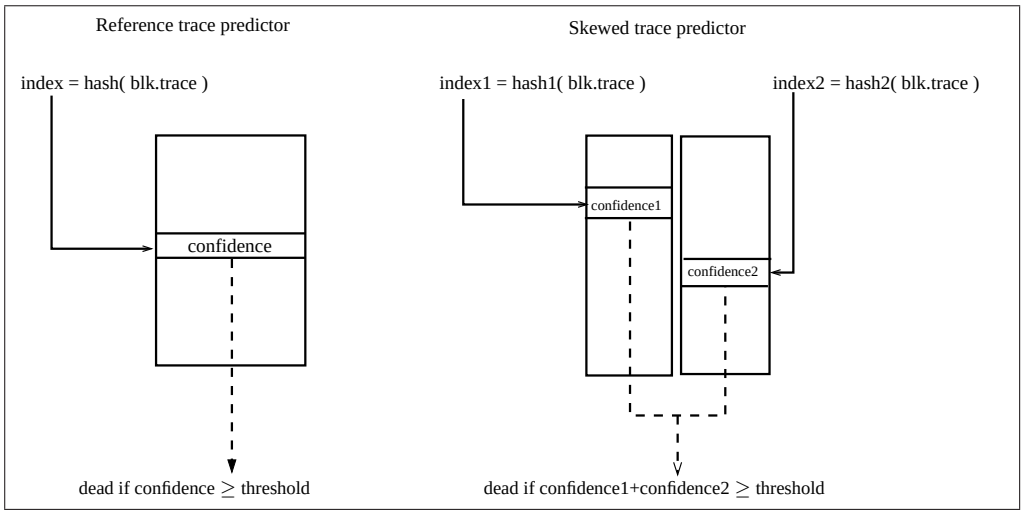


Figure 3: Block diagrams for dead block predictors

updated. When a block is evicted, the counter is incremented. Thus, a counter is only incremented by a signature resulting from the last access to a block.

When a block is accessed and then the signature is updated, the table of counters is consulted. If the counter exceeds a threshold (e.g. 2), then the block is predicted dead. Each cache block stores a single bit prediction. For comparison, we use a 15-bit signature indexing a 32K-entry table of counters. The predictor exclusive-ORs the first 15 bits of each PC with the next 15 bits and adds this quantity to a running 15-bit trace.

The original reference trace predictor of Lai *et al.* uses data addresses as well as instruction addresses, requiring a large table because of the high number of signatures. Subsequent work found that using only instruction addresses was sufficient and allows smaller tables [12]; thus, we use only instruction addresses for all of the predictors in this paper.

4.2 A Skewed Organization

In the original trace-based dead block predictor, a single table is indexed with the signature. For this study, we explore an organization that uses the idea of a skewed organization [19, 13] to reduce the impact of conflicts in the table. The predictor keeps two 16K-entry tables of 2-bit counters, each indexed by a different 14-bit hash of the 15-bit block signature. Each access to the predictor yields two counter values. The sum of these values is used as a confidence that is compared with a threshold; if the threshold is met, then the corresponding block is predicted dead. This organization offers an improvement over the original reference trace predictor because two distinct traces might conflict in one table, but are less likely to conflict in both tables, so the effect of destructive conflicts is reduced.

Figure 3 shows the difference in the design of the original and skewed reference trace predictors.

5 Experimental Methodology

This section outlines the experimental methodology used in this study.

5.1 Simulation Environment

We use SPEC CPU 2000 and SPEC CPU 2006 benchmarks. We simulate the VVC by modifying SimpleScalar [4]. This infrastructure enables collecting instructions-per-cycle figures as well as misses per kilo-instruction, dead block predictor accuracy, and cache efficiency.

Parameter	Configuration
Issue width	4
Reorder Buffer	128 entry
Load/Store Queue	32 entry
L1 I-Cache	64KB, 2 way LRU, 64B blocks, 1 cycle hit
L1 D-Cache	64KB, 2 way LRU, 64B blocks, 3 cycle hit
L2 Cache	2 MB, 16 way LRU, 64B blocks, 12 cycle hit
Virtual addresses	64 bits
Main Memory	270 cycle

(a)

Parameter	Configuration
Trace encoding	15 bit
Predictor table entries	32, 768
Predictor entry	2 bit
Predictor overhead	8KB
Cache overhead	64KB
Total overhead	76 KB

(b)

Table 1: (a) Microarchitectural simulator parameters, and (b) dead block predictor parameters.

Table 1(a) shows the configuration of the simulated machine. Each benchmark is compiled for the Alpha EV6 instruction set. For SPEC CPU 2000, we use the Alpha executables that were at one time available from `simplescalar.com` compiled with DEC C V5.9008, Compaq C++ V6.2-024, and Compaq FORTRAN V5.3-915. For SPEC CPU 2006, we use binaries compiled with the GCC 4.11 compilers for C, C++, and FORTRAN. For most experiments, we model a 16-way set-associative cache to remain consistent with other previous work [9, 12, 15, 16, 17], but Section 6.5 shows that our technique maintains significant improvement at lower associativities.

SimPoint [14] identifies a single two billion instruction characteristic region (i.e. *simpoint*) of each benchmark. The infrasturcture simulates two billion instructions, using the first 500 million to warm microarchitectural structures and reporting the results on the next 1.5 billion instructions.

We choose a memory-intensive subset of the benchmarks based on the following criteria: a benchmark is used if it (1) does not cause an abnormal termination in the baseline sim-outorder simulator for the chosen simpoint, and (2) if increasing the size of the L2 cache from 1MB to 2MB results in at least a 5% speedup. Benchmarks that experience negligible improvement from a higher capacity cache are unlikely to be affected positively or negatively by our optimization.

5.2 Dead Block Predictor Details

5.2.1 Accounting for State in the Predictor

The dead block predictor keeps two 16K-entry tables of 2-bit counters. Each cache block includes additional VVC metadata: 1 bit that is true if the block is predicted dead, 1 bit that is true if the block is a receiver block, and 15 bits for the current trace encoding for that block. The overhead of the predictor and VVC metadata is 76KB which is 3.4% of the total 2MB cache space (including both the data and tag arrays). An accounting of the predictor overhead is given in Table 1(b).

5.3 Estimating Dead Block Hit latency

An L2 cache access takes 12 cycles in the simulated environment. CACTI 5.1 [21] estimates that an additional tag match in the adjacent set, made once the initial tag match in the original set has failed, consumes an extra 2 cycles[1].

An additional sequential tag match latency is simulated for VVC hits. Experiments show that IPC is insensitive to additional L2 hit latency as long as that latency is a small fraction of the miss latency. For instance, negligible change results when pessimistically assuming that VVC hits take double the normal hit latency because a) most accesses hit in the normal L2 cache, and b) a VVC hit at twice the L2 hit latency avoids a much larger L2 miss latency.

[1]We considered doing the two tag matches in parallel, but decided against it because of power and complexity concerns; in essence, this would be no better in terms of power than doubling the associativity of the cache.

5.4 Measuring Cache Efficiency

Cache efficiency is a statistic defined by Burger *et al.* [3] to quantify the average amount of time blocks in the cache contain live data. Cache efficiency is computed as:

$$E = \frac{\sum_{i=0}^{A \times S - 1} U_i}{N \times A \times S}$$

where N is the total number of cycles executed, A is the number of blocks per set, S is the number of sets in the cache, and U_i is the total number of cycles for which cache block i contains live data, i.e., data that will be referenced again before it is evicted. Thus, cache efficiency is the average of the live cycles for each cache block. The performance simulation infrastructure collects block live times and produces cache efficiency as an output.

5.5 Simulating Multiple Threads

We use a trace-based simulator to measure the performance of the virtual victim cache in the presence of multiple threads. The address traces come from the same simpoints described above. We then synthesize new address traces representing four of the benchmarks running simultaneously, using instruction sequence numbers to determine the order of the memory accesses. This approach allows for very fine-grained interleaving of the memory accesses as one would expect from four cores.

A program chooses four benchmarks at random to combine into a single trace. We run this program 10 times to synthesize 10 randomly chosen combinations. We simulate the baseline cache as well as the virtual victim cache using the same basic cache simulation infrastructure as the other experiments. In this scenario, the L2 cache is shared among all threads. The trace-based approach cannot measure speedup, but can collect misses per kilo-instruction, predictor accuracy, and cache efficiency statistics.

6 Experimental Results

In this section we discuss results of our experiments.

6.1 Reduction in L2 Misses

We investigate the virtual victim cache in the context of a baseline 2MB 16-way set associative cache as well as a 2MB fully associative cache and a 2MB cache enhanced with a 64KB victim cache. Note that both the fully associative cache and the 64KB victim cache are infeasible in hardware, requiring 32K entry and 1K entry associative memories, respectively. We choose a 64KB victim cache because it requires approximately the same amount of SRAM, including the tag array, as the extra structures of the VVC.

Figure 4 shows the impact of the VVC on L2 misses per thousand instructions (MPKI). Figure 4(a) shows the raw MPKI values for each benchmark and cache configuration. The average MPKIs for the baseline and fully associative LRU caches are approximately 9.7. The real victim cache yields 8.9 MPKI. The VVC provides an average MPKI of 7.6, an improvement over the baseline of 22% and over the real victim cache of 15%. The VVC provides its best reduction in raw MPKI for `181.mcf`, reducing misses by approximately 26 MPKI. As a lower limit on MPKI, we include results for Belady's MIN optimal replacement policy [2] which results in an average MPKI of 4.9. The VVC moves average MPKI 44% closer to optimal than the baseline cache.

Figure 4(b) shows the percent improvement in MPKI. The VVC, with an 11.7% improvement in MPKI over the baseline, outperforms the fully associative LRU cache whose improvement is 8.4%, and outperforms the real victim cache whose improvement is 2.7%. The best percentage reduction over the baseline is 79% for `187.facerec`. A fully associative cache with optimal replacement improves MPKI by an average of 45%, so clearly there is room for improvement.

6.2 IPC Improvement

Reducing cache misses translates into improved performance. Figure 5 shows the instructions-per-cycle rates given by the VVC as well as other techniques. The VVC achieves a harmonic mean IPC of 0.791, an improvement

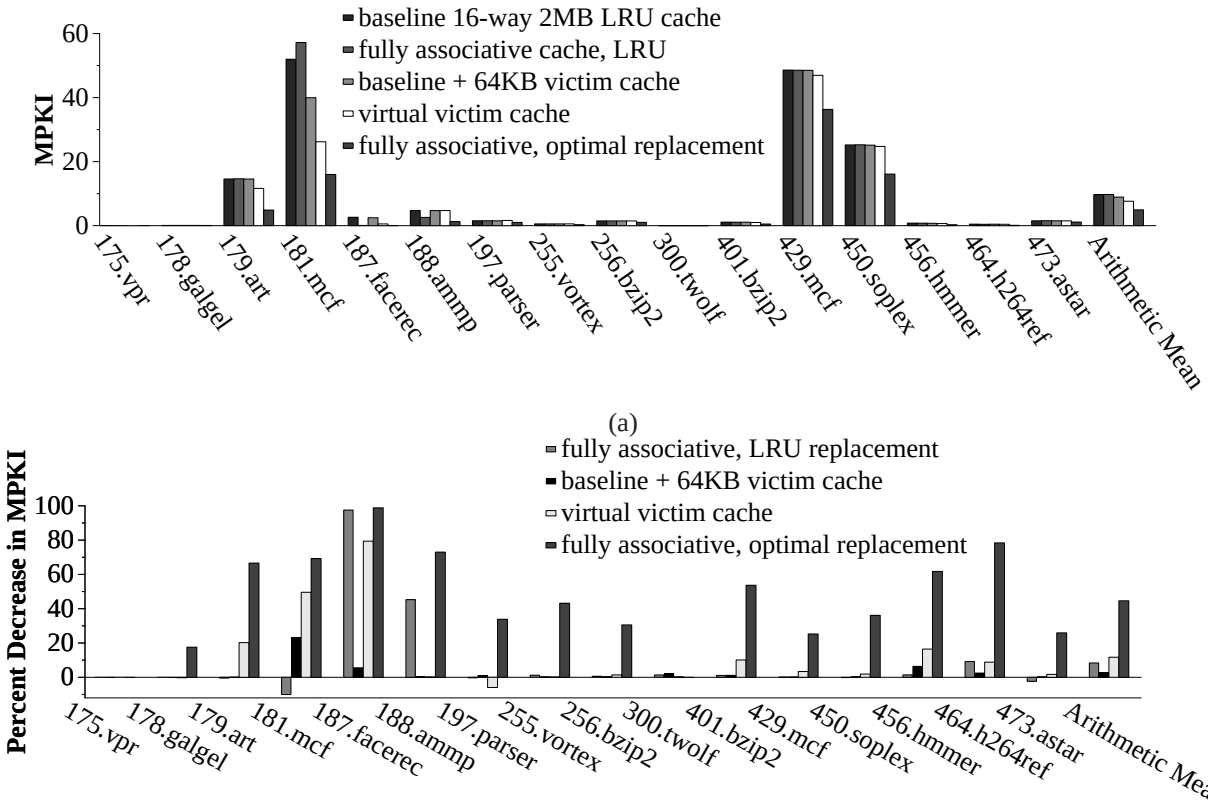


Figure 4: L2 cache misses per thousand instructions (a), and percent improvement in MPKI (b)

of 23% over the baseline, 23% over the fully associative cache, and 12% over the real victim cache.

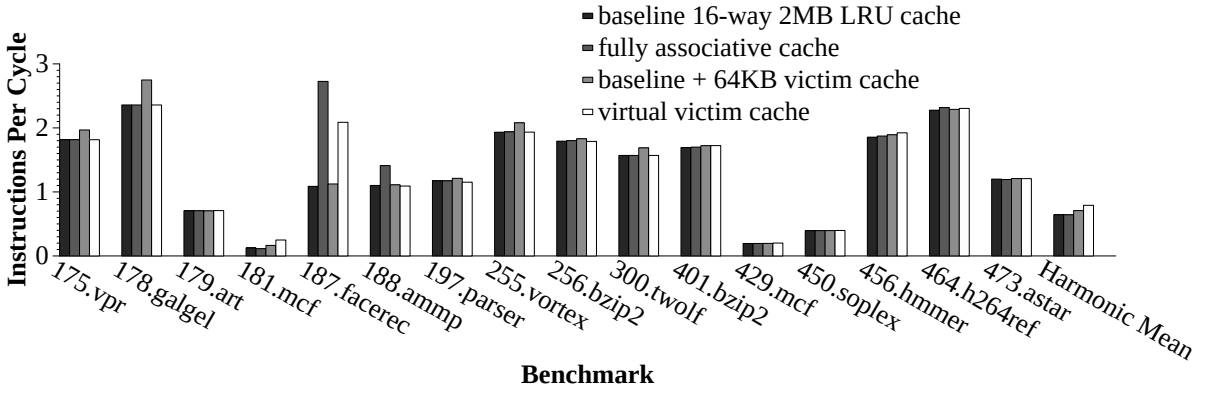


Figure 5: IPC improvement

Figure 6 shows the speedup computed by dividing the improved IPC by the baseline IPC for each benchmark, as well as the arithmetic mean speedup. The arithmetic mean speedup for the VVC is 12.5%, compared with 5.5% for the real victim cache and 10.8% for the fully associative cache. Two benchmarks in particular, `181.mcf` and `187.facerec`, yield remarkable speedups of 98% and 92%, respectively, while the VVC while other benchmarks show more modest improvements. No benchmark is significantly slowed down by the VVC; `197.parser` is the worst case in terms of slowdown, with a speedup of -1.8%.

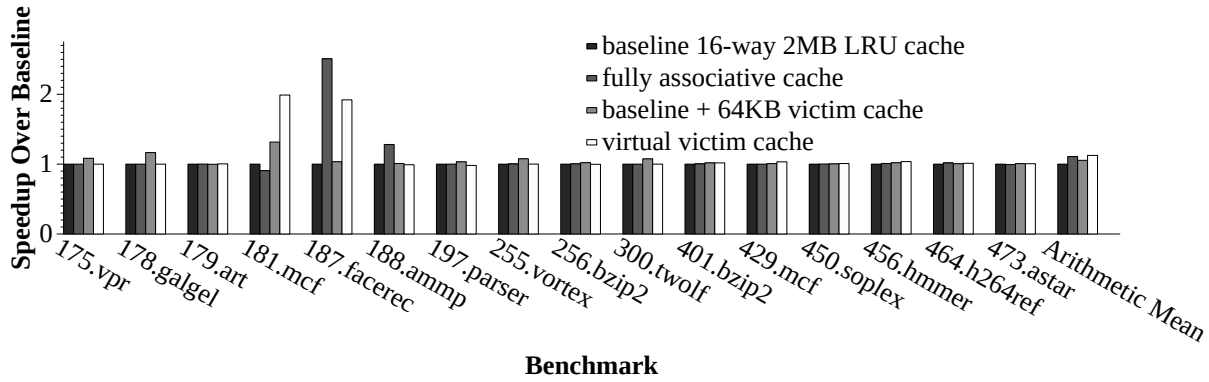


Figure 6: Speedup

6.3 Prediction Accuracy and Coverage

In this section we quantify the improvement given by our new dead block predictor using a skewed organization derived from a branch predictor design. Mispredictions come in two varieties: false positives and false negatives. False positives are more harmful for most applications of dead block prediction because they falsely allow an optimization to use a live block for some other purpose. False negatives, while not harmful to performance, limit the potential applicability of the optimization. Thus, we would like to reduce both kinds of errors.

The coverage of a dead block predictor is the number of positive predictions divided by the total number of predictions. If a dead block predictor is consulted on every cache access, then the coverage represents the fraction of cache accesses when the optimization may be applied. Higher coverage means more opportunity for the optimization.

Figure 7 shows the coverage and false positive rates of the old and new predictors. On average, our skewed predictor covers 17.2% of accesses, compared with 18.0% for the Lai *et. al* predictor. However, *all* of the extra coverage attributable to the Lai *et al.* predictor is due to false positive mispredictions. On average, our predictor has a false positive rate of 3.5%, compare with 4.3% for the Lai *et al.* predictor; thus, harmful false positives are significantly reduced.

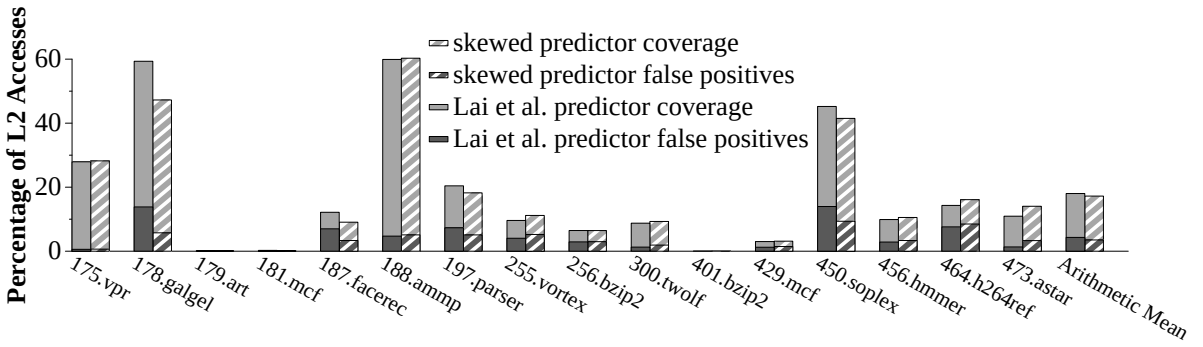


Figure 7: Predictor coverage and false positive rate

Figure 8 shows the false positive misprediction rate for each predictor as the hardware budget for the prediction tables varies from 128 bytes to 128KB. We choose the 8KB hardware budget as our predictor because it represents a good trade-off between area and performance, but clearly there is potential to improve accuracy with a larger hardware budget. As mentioned above, at the 8KB hardware budget, the skewed predictor has a false positive misprediction rate of 3.5% compared with 4.3% for the Lai *et al.* predictor. At a 128KB hardware budget, the new predictor has a false positive misprediction rate of 3.4% compared with 4.2% for the Lai *et al.* predictor. The reduction in mispredictions at the 8KB budget translates into a speedup of 1% in average speedup (not illustrated)

using the VVC.

With a 8KB budget, the original Lai *et al.* predictor allowed the VVC to achieve a 5.4% average speedup over all the benchmarks. The skewed predictor improves this average speedup to 12.5% (not graphed for space reasons).

We did investigate other dead block predictors such as the reference counting predictors [8] and cache bursts, however neither of these predictors provided any accuracy or performance improvements in our study. The skewed organization resulted in significant improvements in accuracy and performance; we believe that future improvements to dead block predictors will result in improved performance for the VVC as well as other optimizations.

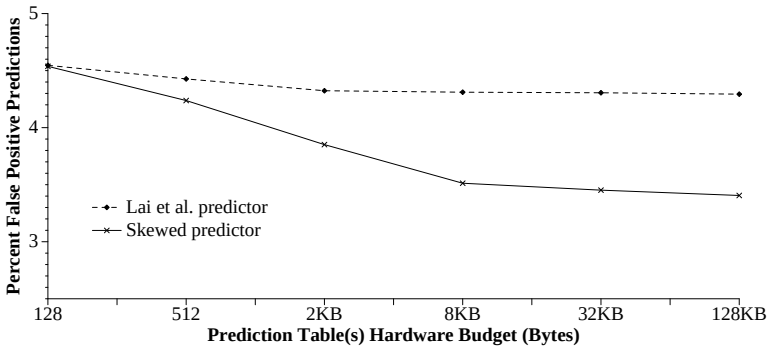


Figure 8: Arithmetic mean predictor false positive mispredictions for a range of hardware budgets

6.4 Improvement in Cache Efficiency

As stated in the introduction, the VVC improves cache efficiency by making sure more cache blocks are live. Figure 9(a) quantifies this improvement for each benchmark. The average cache efficiency is 0.412 for the baseline, compared with 0.442 for the VVC.

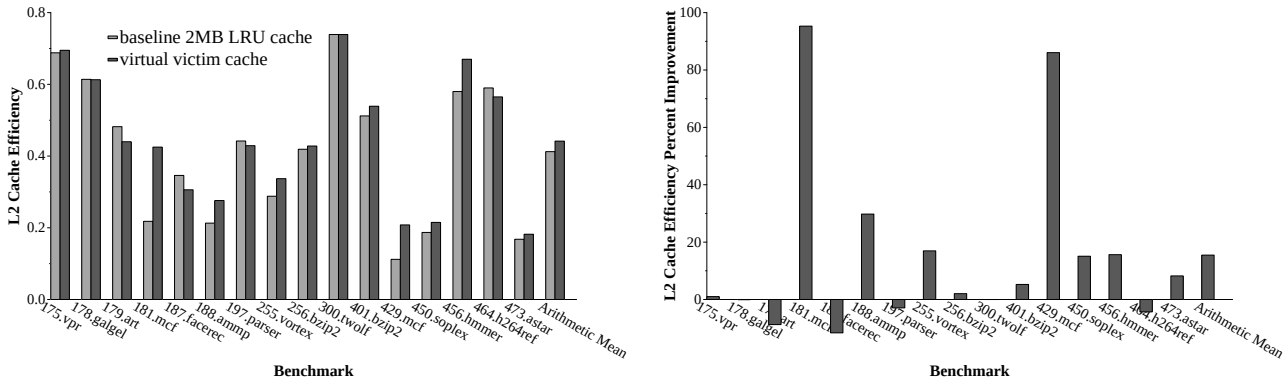


Figure 9: Cache efficiency

Figure 9(b) shows the percent improvement in cache efficiency given by the VVC. The maximum improvement is 95% for `181.mcf`, and the average improvement is 15.5%.

6.5 Reduction in Cache Area

We have shown that the VVC can deliver improved performance by reducing the number of L2 cache misses. Alternately, the VVC can reduce the area consumed by the L2 cache, leading to equivalent performance with reduced power and area requirements. Figure 10 illustrates the ability of the VVC to deliver equivalent performance with a reduced capacity cache. Figure 10(a) shows the average MPKI for the baseline cache and the VVC for a

variety of cache sizes obtained by increasing the associativity of the cache from 8 through 16. At a capacity of 1.7MB representing an associativity of 13, the VVC achieves an average MPKI of 9.9, just above the MPKI of the 2MB baseline cache at 9.7. At a capacity of 1.8MB representing an associativity of 14, the VVC outperforms the baseline with an MPKI of 9.1. Figure 10(b) shows how these MPKIs translate into performance. The baseline harmonic mean IPC for a 2MB cache is 0.64, compared with 0.63 for a 1.7MB VVC and 0.68 for a 1.8MB VVC.

At the 1.7MB capacity, the VVC reduces the number of SRAM cells required by 16% including the SRAM cells for data, tags, predictor structures and metadata. At the 1.8MB capacity, the VVC reduces the SRAM cells needed by 9.5%.

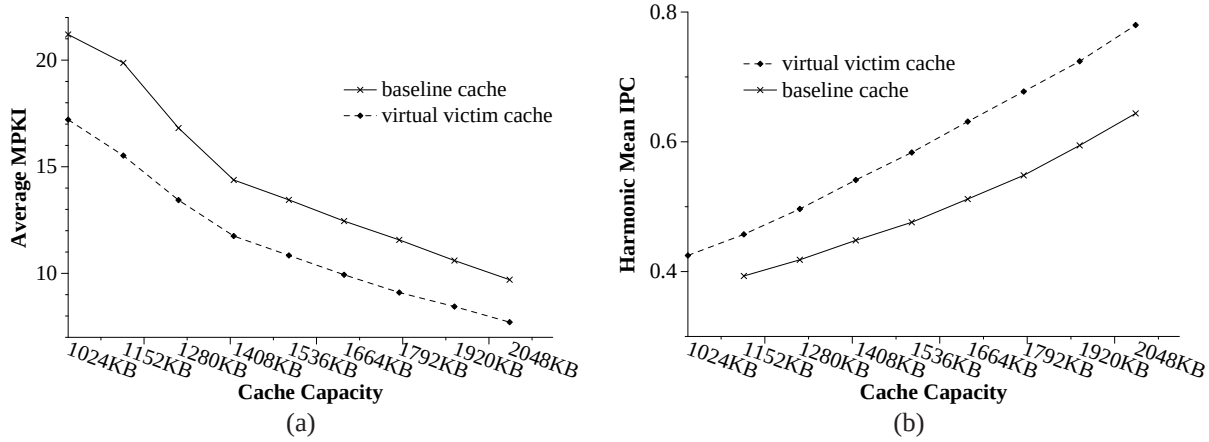


Figure 10: Reduction in cache area

6.6 Increase in Tag Array Access

The improved performance of the VVC comes at the expense of extra reads to the tag array. Most accesses hit in the cache; however, every time a tag match fails in a set, the adjacent set must also be searched. Figure 11 shows the number of reads to the L2 tag array for the baseline 2MB cache as well the VVC. In two billion executed instructions, the L2 tag array is read on average 77 million times in the baseline cache and 97 million times in the VVC, an increase of 26%. To put it another way, the number of tag array reads in the baseline cache is 3.9% of the total number of instructions executed, versus 4.9% for the VVC.

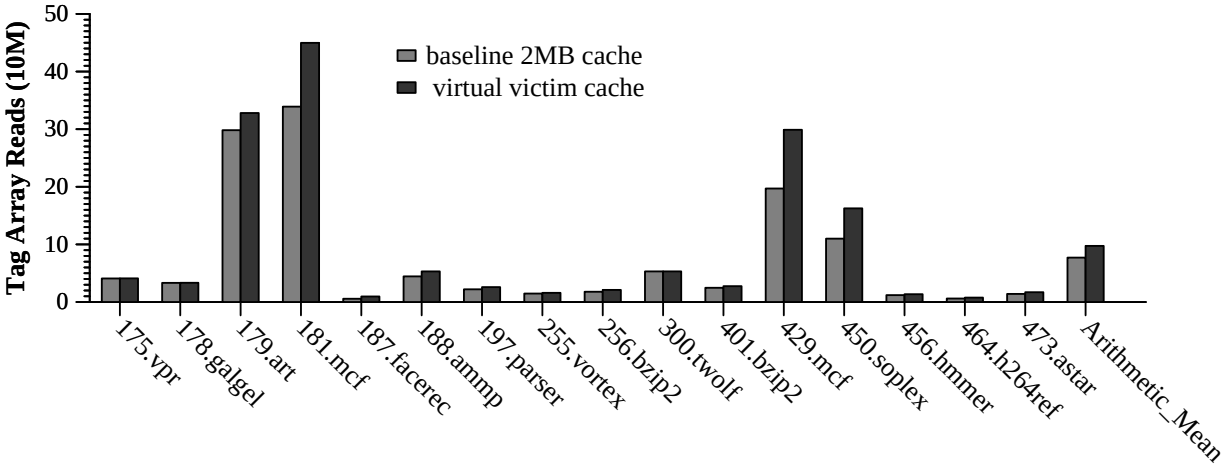


Figure 11: Increase in tag array reads due to VVC

6.7 Multiple Threads

We simulate the baseline and VVC configurations in the presence of multiple threads. The methodology outlined in Section 5.5 is used to simulate the behavior of the cache in a multi-core environment. We simulate a shared cache, i.e., the L2 cache is not partitioned.

Figure 12 illustrates the MPKI for 10 combinations of four benchmarks (benchmark numbers are listed on the x-axis, separated by plus symbols) for a 2MB cache. The baseline cache yields an average 39.9 MPKI, while the VVC delivers 33.8 MPKI, an improvement of 15%.

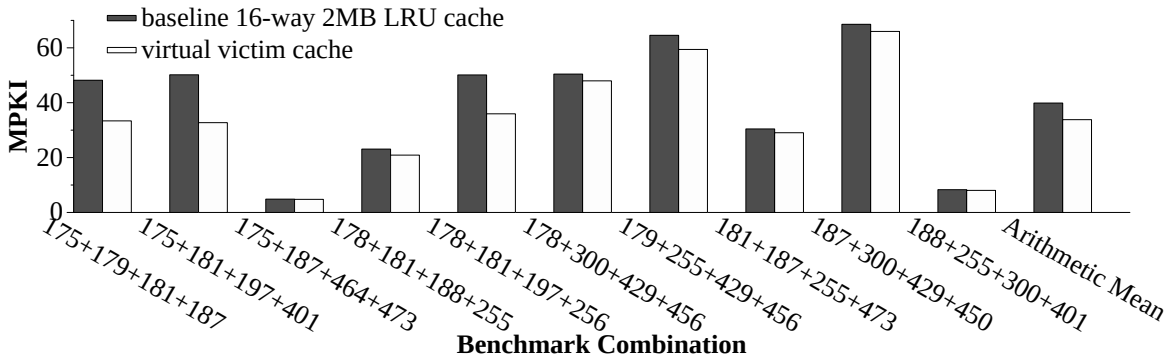


Figure 12: MPKI for multiple threads and a 2MB cache

Figure 13 shows the improvement in MPKI delivered by the VVC for a variety of L2 cache sizes for the multi-threaded workloads. For a 1MB cache, the VVC lowers MPKI by 13.8%. For a 2MB cache, the VVC lowers MPKI by 12.9%. The improvements for other cache sizes are more modest, e.g. for an 8192MB cache the improvement is 5.5%. However, all cache sizes are improved by the VVC.

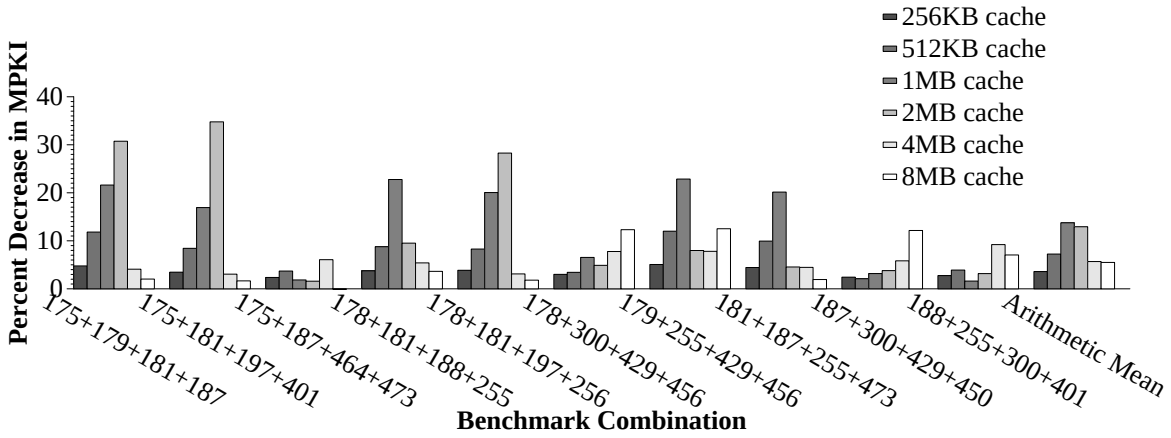


Figure 13: MPKI improvement for multiple threads for various cache sizes

On average, the cache efficiency (not illustrated) for the baseline cache is 0.198, while the cache efficiency for the VVC is 0.230, an improvement of 16%. Note that the efficiency of the cache is far lower for both the baseline and VVC than it is for the single-threaded workloads due to decreased live times caused by increased numbers of evictions from the cache.

7 Adaptive Insertion Policy

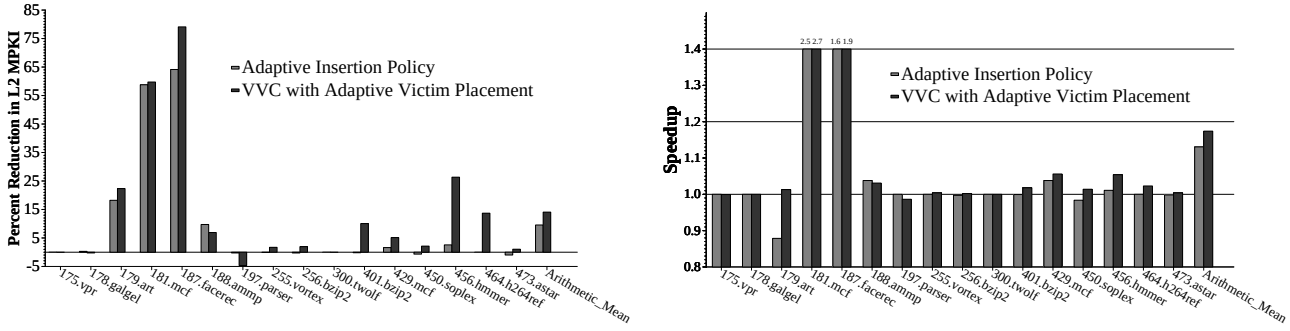
In this section we explore an adaptive insertion policy for the VVC and compare it with a similar policy for caches from the literature.

7.1 Set Dueling

Qureshi *et al.* [15] proposed Dynamic Insertion Policy (DIP) to improve cache performance by preventing thrashing in large workloads. The technique proposed in improved block insertion (i.e. placement) policy. The victim selection policy selects the least recently used (LRU) block. The insertion policy decides what position in the LRU stack the incoming block is placed. Two distinguished positions are consider: the LRU and MRU positions. For most workloads, the standard MRU insertion policy is sufficient. However, for workloads whose working sets exceed the capacity of the cache, some fraction of the working set can be retained in the cache by inserting the incoming block in the LRU position. This LRU line moves to the MRU position only if it is referenced again, otherwise it is evicted. This ensures that blocks that are never used between insertion and eviction do not occupy cache space. This insertion policy can hurt performance of LRU friendly workloads, so a technique called *set dueling* is used to dynamically determine the best policy [17]. Set dueling dedicates a small number of sets in the cache to LRU replacement policy and a another small set of sets to LRU Insertion policy. The policy resulting in fewer misses in the dedicated sets wins and rest of the sets in the cache follows that policy.

7.2 Adaptive Insertion in the VVC

The VVC suffers from a similar insertion policy dilemma: some workloads benefit from placing the receiver of a victim block into the MRU position, while others benefit from placement in the LRU position depending on the behavior of the program. For example, benchmarks such as `187.facerec` and `401.bzip2` perform well when evicted blocks are placed in MRU position, but `179.art`, `188.ammp`. For some workloads, the MRU insertion policy might be better because a victim block is not be accessed immediately, but it will be accessed soon so it will still be in the adjacent set making its way down the LRU stack. For other workloads, the LRU insertion policy might be better because 1) when a block will be accessed again, it will be accessed soon, or 2) many victim blocks placed in adjacent sets will not used for a long time and might as well be evicted instead of more useful data; after all, this is the point of the LRU replacement policy.



(a) (b)

Figure 14: Virtual victim cache with dynamic insertion policy vs. set dueling

Set dueling for the VVC insertion policy outperforms set dueling for the baseline cache insertion policy. The average MPKI for the baseline 2MB cache enhanced with set dueling is 7.5, compared with 7.2 for the VVC with set dueling. Recall that the VVC without set dueling achieves an MPKI of 7.6; thus, set dueling improves VVC MPKI by 5.2% over VVC without set dueling, and by 4% over set dueling without VVC.

Figure 14 shows the improvements in MPKI and IPC using set dueling with and without the VVC. Set dueling for the baseline cache results in a 9.6% average improvement in MPKI and an average speedup of 13%, while set dueling for the VVC improves MPKI by 14.1% over the baseline, with a 17.4% speedup. **Note:** for the VVC, we use set dueling to guide the block insertion policy for placing victim blocks into adjacent sets; we do not use adaptive insertion for block placement within the cache itself.

8 Conclusion and Future Work

This paper has explored a cache management strategy that is a combination of block placement, search, and block replacement driven migrating LRU blocks into predicted dead blocks in other sets. The virtual victim cache significantly reduces misses and improves performance for several benchmarks, provides modest improvement to most benchmarks, and does not significantly slow down any benchmarks. It significantly reduces misses in multi-threaded workloads. It does this with a relatively small amount of extra state and a modest increase in the number of accesses to the tag array. Alternately, the virtual victim cache allows reducing the size of the L2 cache while maintaining performance. Using an adaptive insertion policy, more benchmarks are improved.

We see several future directions for this work. Adapting a skewed organization inspired by branch prediction research has improved dead block predictor accuracy. It might be possible to adapt other predictor organizations to improve dead block predictor accuracy and coverage. Reducing the number of accesses to the tag array through a more intelligent search strategy could improve the power behavior of the cache. The VVC allows reducing the associativity and size of the cache while maintaining performance, but the potential for reducing the number of sets has not been explored. An adaptive insertion policy improves the performance of the VVC, but perhaps other policies would provide additional improvement. So far, the VVC does not distinguish between victims that are likely to be used again and those that are not. A more discriminating technique could further improve performance by filtering out cold data.

References

[1] Jaume Abella, Antonio González, Xavier Vera, and Michael F. P. O'Boyle. Iatac: a smart predictor to turn-off l2 cache lines. *ACM Trans. Archit. Code Optim.*, 2(1):55–77, 2005.

[2] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[3] D. Burger, J. R. Goodman, and A. Kagi. The declining effectiveness of dynamic caching for general-purpose microprocessors. *Technical Report 1261*, 1995.

[4] Doug Burger and Todd M. Austin. The SimpleScalar tool set version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin, June 1997.

[5] Zhigang Hu, Stefanos Kaxiras, and Margaret Martonosi. Timekeeping in the memory system: predicting and optimizing memory behavior. *SIGARCH Comput. Archit. News*, 30(2):209–220, 2002.

[6] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *SIGARCH Comput. Archit. News*, 18(3a):364–373, 1990.

[7] Georgios Keramidas, Pavlos Petoumenos, and Stefanos Kaxiras. Cache replacement based on reuse-distance prediction. In *ICCD*, pages 245–250, 2007.

[8] Mazen Kharbutli and Yan Solihin. Counter-based cache replacement and bypassing algorithms. *IEEE Transactions on Computers*, 57(4):433–447, 2008.

[9] An-Chow Lai and Babak Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *International Symposium on Computer Architecture*, pages 139 – 148, 2000.

[10] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-block prediction & dead-block correlating prefetchers. *SIGARCH Comput. Archit. News*, 29(2):144–154, 2001.

[11] Alvin R. Lebeck and David A. Wood. Dynamic self-invalidation: reducing coherence overhead in shared-memory multiprocessors. *SIGARCH Comput. Archit. News*, 23(2):48–59, 1995.

[12] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pages 222–233, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[13] Pierre Michaud, André Seznec, and Richard Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 292–303, June 1997.

[14] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using simpoint for accurate and efficient simulation. *SIGMETRICS Perform. Eval. Rev.*, 31(1):318–319, 2003.

[15] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely Jr., and Joel S. Emer. Adaptive insertion policies for high performance caching. In *34th International Symposium on Computer Architecture (ISCA 2007), June 9-13, 2007, San Diego, California, USA*. ACM, 2007.

[16] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. A case for mlp-aware cache replacement. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 167–178, Washington, DC, USA, 2006. IEEE Computer Society.

[17] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. A case for mlp-aware cache replacement. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 167–178, Washington, DC, USA, 2006. IEEE Computer Society.

[18] Jennifer B. Sartor, Subramaniam Venkiteswaran, Kathryn S. McKinley, and Zhenlin Wang. Cooperative caching with keep-me and evict-me. *Annual Workshop on Interaction between Compilers and Computer Architecture*, 0:46–57, 2005.

[19] André Seznec. A case for two-way skewed-associative caches. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 169–178, New York, NY, USA, 1993.

[20] Stephen Somogyi, Thomas F. Wenisch, Nikolaos Hardavellas, Jangwoo Kim, Anastassia Ailamaki, and Babak Falsafi. Memory coherence activity prediction in commercial workloads. In *WMPI '04: Proceedings of the 3rd workshop on Memory performance issues*, pages 37–45, New York, NY, USA, 2004. ACM.

[21] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P. Jouppi. Cacti 5.1. *Technical report HPL-2008-20*, 2008.

[22] Zhenlin Wang, Kathryn S. McKinley, Arnold L. Rosenberg, and Charles C. Weems. Using the compiler to improve cache replacement decisions. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, page 199, Los Alamitos, CA, USA, 2002. IEEE Computer Society.