Insertion and Promotion for Tree-Based PseudoLRU Last-Level Caches

Daniel A. Jiménez
Department of Computer Science and Engineering
Texas A&M University

ABSTRACT

Last-level caches mitigate the high latency of main memory. A good cache replacement policy enables high performance for memory intensive programs. To be useful to industry, a cache replacement policy must deliver high performance without high complexity or cost. For instance, the costly least-recently-used (LRU) replacement policy is not used in highly associative caches; rather, inexpensive policies with similar performance such as PseudoLRU are used.

We propose a novel last-level cache replacement algorithm with approximately the same complexity and storage requirements as tree-based PseudoLRU, but with performance matching state of the art techniques such as dynamic re-reference interval prediction (DRRIP) and protecting distance policy (PDP). The algorithm is based on PseudoLRU, but uses set-dueling to dynamically adapt its insertion and promotion policy. It has slightly less than one bit of overhead per cache block, compared with two or more bits per cache block for competing policies.

In this paper, we give the motivation behind the algorithm in the context of LRU with improved placement and promotion, then develop this motivation into a PseudoLRU-based algorithm, and finally give a version using set-dueling to allow adaptivity to changing program behavior. We show that, with a 16-way set-associative 4MB last-level cache, our adaptive PseudoLRU insertion and promotion algorithm yields a geometric mean speedup of 5.6% over true LRU over all the SPEC CPU 2006 benchmarks using far less overhead than LRU or other algorithms. On a memory-intensive subset of SPEC, the technique gives a geometric mean speedup of 15.6%. We show that the performance is comparable to state-of-the-art replacement policies that consume more than twice the area of our technique.

1. INTRODUCTION

The last-level cache (LLC) mitigates long latencies from main memory. A good replacement policy in the LLC allows the cache to deliver good performance by replacing blocks that are least likely to be referenced again in the near future. The least-recently-used (LRU) replacement policy is motivated by the observation that, if a block has not been used recently, then it is unlikely to be used again in the near future. This intuition is reasonable, but leaves a considerable amount of room for improvement.

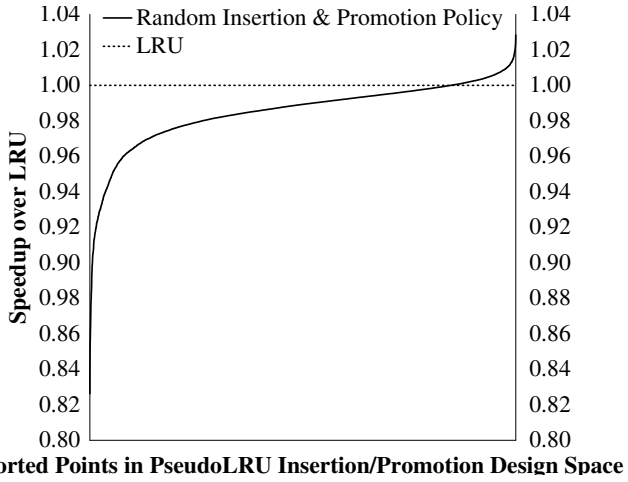

Figure 1: Random Design Space Exploration for PseudoLRU Insertion and Promotion

There has been much previous work on last-level cache replacement. From this work, we have ample evidence that we can improve significantly over LRU, but at a cost. Even the LRU policy itself is expensive: in a 16-way set associative cache, LRU requires four bits per cache block. Other proposed algorithms improve on this cost.

This paper describes a last-level cache replacement algorithm with very low overhead that delivers high performance. The algorithm uses less than one bit per cache block; on a 16-way set associative cache, it uses 15 bits per set or less than 0.94 bits per block. It extends tree-based PseudoLRU [11], a policy with performance similar to LRU. We enhance it to deliver performance comparable with state-of-the-art replacement algorithms. In particular, we compare our technique with dynamic re-reference interval prediction (DRRIP) [13] and Protecting Distance Policy (PDP) [6] and find that it yields similar performance with much lower overhead. Our technique gives a geometric mean 5.61% speedup over LRU over all SPEC CPU 2006 benchmarks while DRRIP gives a comparable 5.41% speedup and PDP achieves a similar 5.69% speedup. On a memory-intensive subset

MICRO-46, December 07 - 11 2013, Davis, CA, USA

http://dx.doi.org/10.1145/2540708.2540733 .

of SPEC, our technique gives a 15.6% speedup compared with 15.6% for DRRIP and 16.4% for PDP. DRRIP requires twice as much storage overhead as our technique and PDP requires even more additional storage as well as a specialized microcontroller for implementing its policy.

PseudoLRU, as its intellectual parent LRU, has an inherent insertion and promotion policy: blocks are inserted into the most recently used (MRU) position, and when they are accessed again they are moved to the MRU position. However, there are many degrees of freedom in the choice of insertion and promotion that are not exploited by LRU or PseudoLRU. The basic idea of our technique is to explore this space of possible insertion and promotion policies to choose a policy that maximizes performance over a set of training workloads. Figure 1 shows the speedups obtained over LRU by a uniformly random sampling of the design space for our technique, ranging from significant slowdowns to speedups around 2.8% over LRU (see Section 4.1 for more on this figure). Clearly most of the points in this random sample are inferior to LRU, but there are some areas of improvement over LRU. Random search leaves significant potential undiscovered, so we use genetic algorithms to further explore the enormous search space.

The paper is organized as follows: in Section 2 we describe the basic idea of our technique within the context of the LRU replacement policy. In Section 3 we give background into tree-based PseudoLRU and describe how our technique extends it. In Section 4 we discuss our experimental methodology, including details of our genetic algorithms, our workload neutral evaluation, and our simulation infrastructures. Section 5 gives the results of experiments showing that our technique performs well in terms of improving performance and reducing misses. We discuss related work in Section 6. Finally, in Section 7, we give directions for future research and conclude.

2. MOTIVATION

This section describes the motivation behind the proposed algorithm.

2.1 LRU Replacement Policy

To provide context, we give a brief overview of LRU and its implementation.

2.1.1 The Idea

The idea of the LRU policy is simple: when a cache set is full and a new block must be inserted into this set, we evict the block that was used least recently. The intuition behind this algorithm is that the recency of use for a given block is a good predictor for how long it will be before it is used again.

2.1.2 The Implementation

The LRU policy may be implemented by maintaining the cache blocks in a set as a *recency stack*. Consider a k-way set associative cache. Each block has a distinct position in the recency stack. The top of the stack at position 0 is the most-recently-used (MRU) block, the next block at position 1 is the next-most-recently-used, etc., and the last block in the stack at position $k-1$ is the LRU block. When a block in position i is accessed, blocks from positions 0 through $i-1$ are pushed down by one while the block in position i is moved to position 0. That is, a block accessed from within the recency stack is *promoted* to the MRU position. When a victim is required, it is chosen as the last item in the stack in position $k-1$, i.e. from the LRU position. An incoming block replaces the victim in position $k-1$, then moved to position 0. That is, an incoming block is *inserted* into the MRU position.

One way to implement the recency stack would be to order blocks according to their position in the recency stack, i.e. the block in position i in the recency stack would be kept in way i in the set. This implementation requires no extra space overhead, but incurs a frighteningly high cost in energy and delay because of the constant movement of large chunks of data between cache ways as the stack is updated.

A better implementation is to associate an integer from 0 through $k-1$ with each block in a set. That integer gives the block's position in the recency stack. When blocks are moved in the stack, only the values of the integers must be changed. Maintaining the integers is conceptually simple, but requires an extra $\log_2 k$ bits per block, or $k \log_2 k$ additional bits per set to store the positions. For a 16-way cache, this works out to 4 extra bits per block or 64 extra bits per set.

2.2 LRU Needs Improving

Although the intuition behind LRU seems good, memory access behavior at the last-level cache is often more nuanced. For instance, many blocks in the last-level cache are "zero-reuse blocks" [20]. These blocks are brought into the cache and used once, never to be reused again. Other blocks are similarly used a few times and then not reused. These dead blocks [4] must proceed from the MRU position down to the LRU position before they are evicted, wasting the capacity that might otherwise be allocated to more useful blocks. Sometimes it makes sense to, for instance, place an incoming block in the LRU position rather than the MRU position, so that if it is not referenced again it will be quickly evicted.

2.3 Improving Placement and Promotion in LRU

Previous work has observed that LRU-like algorithms can be improved by changing the insertion and promotion policies. The original LRU policy promotes accessed blocks to the MRU position and inserts incoming blocks into the MRU position. Qureshi *et al.* observed that sometimes placing an incoming block into the LRU position provides a benefit [25]. Loh *et al.* as well as others have observed that promoting a block up the stack but not all the way to MRU can also yield a benefit [30, 13]. We generalize this notion into the concept of a *insertion/promotion vector*, or IPV. For a k-way set-associative cache, an IPV is a $k+1$-entry vector of integers ranging from $0..k-1$ that determines what new position a block in the recency stack should occupy when it is re-referenced, and what position an incoming block should occupy. Let $V[0..k]$ be an IPV. Then $V[i]$ contains the new position to which a block in position i should be promoted when it is accessed, or if $i = k$ the position where a new incoming block should be inserted. If $V[i] < i$, then blocks between positions $V[i]$ and $i-1$ are shifted down to make room; otherwise blocks between positions $i+1$ and $V[i]$ are shifted up to make room.

2.4 Example IPVs

The normal LRU insertion and promotion policy is given by $V_{\text{LRU}} = [0, 0, ..., 0]$. LRU insertion as in Qureshi *et*

al. [25] is given by $V_{\text{LRUinsertion}} = [0, 0, ..., 0, k-1]$. If we let $V = [0, 0, ..., 0, k/2, k-1]$ then an incoming block is inserted into the LRU position, then if it is referenced again it is promoted into the middle of the stack, and then a third reference promotes it to MRU. Figure 2 graphically illustrates the transition graph representing LRU for $k = 16$. The vertices of the graph are the recency stack positions. The solid edges represent change in position when a source node's block is accessed. The dashed edges represent the new position to which a block is shifted to accommodate a block moving into its position.

2.5 Finding A Good Placement and Promotion Vector

As a proof of concept, we propose to explore the space of IPVs to find one that works best for a given set of benchmarks. (In Section 4 we refine this approach with a statistically valid methodology but for now we are simply demonstrating the idea.) Ideally, we would find the optimal IPV giving maximum benefit. We evaluate each IPVs by measuring the speedup it yields over normal LRU on the SPEC CPU 2006 benchmark suite [29] using a simulator.

Within this framework there are many possible IPVs. For a k-way set associative cache there are k^{k+1} possible insertion and promotion policies. For example, for $k = 16$, there are 2.95×10^{20} different IPVs. An exhaustive search for the vector providing the best speedup would require simulating the 29 SPEC benchmarks 295 quintillion times. Barring improvements to our 200-CPU cluster, we estimate that this parallel search would require over 300 billion years of wall clock time by which time the Sun will no longer be shining and cosmic inflation will have long since moved all galaxies outside the Local Group beyond the boundaries of the observable universe [21][1]. Thus, we turn to a heuristic search technique: we use a genetic algorithm to evolve a good IPV. We take traces from multiple simpoints [28] of the SPEC CPU 2006 benchmarks collected using the version of CMP$im distributed for the 2010 cache replacement championship [2]. We simulate a 4MB 16-way set associative last-level cache (other configuration details of the simulator are given in Section 4). The genetic algorithm evolves an IPV that results in the a good arithmetic mean speedup over all the benchmarks. The results are shown in Figure 3. The best insertion/promotion vector found by the genetic algorithm is [0 0 1 0 3 0 1 2 1 0 5 1 0 0 1 11 13]. An incoming block is inserted into position 13. A block referenced in the LRU position is moved to position 11. A block referenced in position 2 is moved to position 1, etc. Figure 3 illustrates the transition graph for this vector.

The vector specifies some counterintuitive actions. For instance a block referenced in position 5 is moved to MRU, while a block referenced in position 4 is moved to position 3. Note that blocks can also move to new positions as a consequence of being shifted to accommodate other blocks; e.g., a block in position 3 may move to MRU by being first being shifted to position 4 to accommodate the promotion of another block, then moved to position 0 from 4. Thus, some aspects of the vector go against intuition. However, recall that it is intuition that led to the LRU policy, which is often no better than random (see Figure 4) so for this work we will suspend our intuition in favor of that of the machine.

[1]We can significantly improve this time by eliminating degenerate IPVs for which it is impossible for a block to be promoted to MRU, i.e. if the graph induced by adding edges for all possible changes of position does not include a path from insertion to MRU. However, given resources available to us the resulting exhaustive search would still take billions of years.

2.6 Performance Improvement

Figure 4 shows the speedup over LRU given by the technique described above: Genetic Insertion and Promotion for LRU Replacement (GIPLR) (please see Section 4 for details of the simulation methodology). The figure also shows the speedup of random replacement. GIPLR yields a 3.1% geometric mean speedup over LRU. Random replacement performs better on some workloads and worse on others, giving a geometric mean performance of 99.9% of LRU. Tree-based PseudoLRU performs on average about as well as true LRU.

The vector is the result of an incomplete search of a huge design space. Thus, some of the elements of the vector are not optimal; for instance, replacing the first 12 elements of the vector with 0s yields a slight improvment in the speedup from 3.1% to 3.12%. We may further refine the vector using a hill-climbing approach. However, we will defer to future research the problem of understanding exactly how or why one vector might be better than another.

The figure shows the potential of an algorithm like GIPLR to improve LRU. However, in the following section we show that an algorithm requiring far less state is able to achieve similar performance and, with dynamic adaptivity, yield performance equivalent to state-of-the-art replacement policies.

3. GENETIC INSERTION AND PROMOTION FOR PSEUDOLRU REPLACEMENT

This section describes the main contribution of our research. We apply the technique introduced in Section 2 to tree-based PseudoLRU replacement, and make it adaptive using set-dueling.

3.1 Tree-based PseudoLRU Replacement

A cheaper alternative to the LRU replacement policy is tree-based PseudoLRU [11]. The idea is to maintain a complete binary tree whose leaf nodes are the blocks in a set. The internal nodes each contain one bit; let us call this bit the *plru bit*. To find a victim, we start at the root and go left when we find a 0 plru bit or right when we find a 1 plru bit. When we reach a leaf, we have found our victim: the PLRU block. When a block is accessed, we promote the block by setting the values of the plru bits on the path from the block to the root such that they all lead away from this block. For example, if the block is a left child then its parent's plru bit is set to 1, and if the parent is a right child then the grandparent's plru bit is set to 0, and so on up to the root. We say that such a newly promoted block is in the PMRU position. Figures 5 and 6 give pseudocode for the algorithms to find the PseudoLRU (PLRU) block and promote a block to the pseudo MRU (PMRU) position, respectively. Although the PLRU block is not always the LRU block, it is guaranteed not to be the MRU block and is very likely not to have been used for quite a while because in order to become the PLRU block there would have to have been several intervening promotions to other blocks. In practice, PLRU provides performance almost equivalent to full LRU.

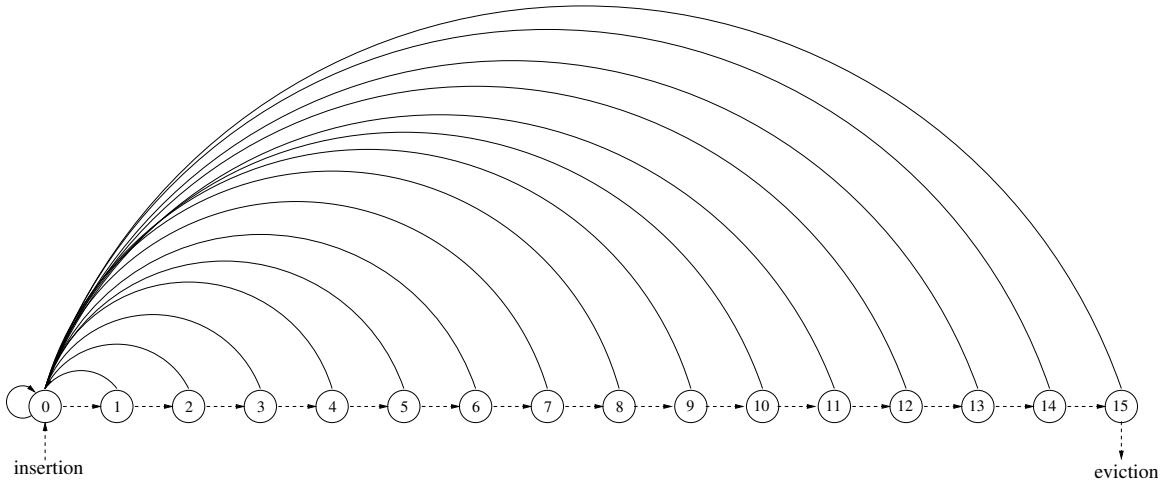

Figure 2: Transition Graph for LRU. A solid edge points to the new position for an accessed or inserted block. A dashed edge shows where a block is shifted when another block is moved to its position.

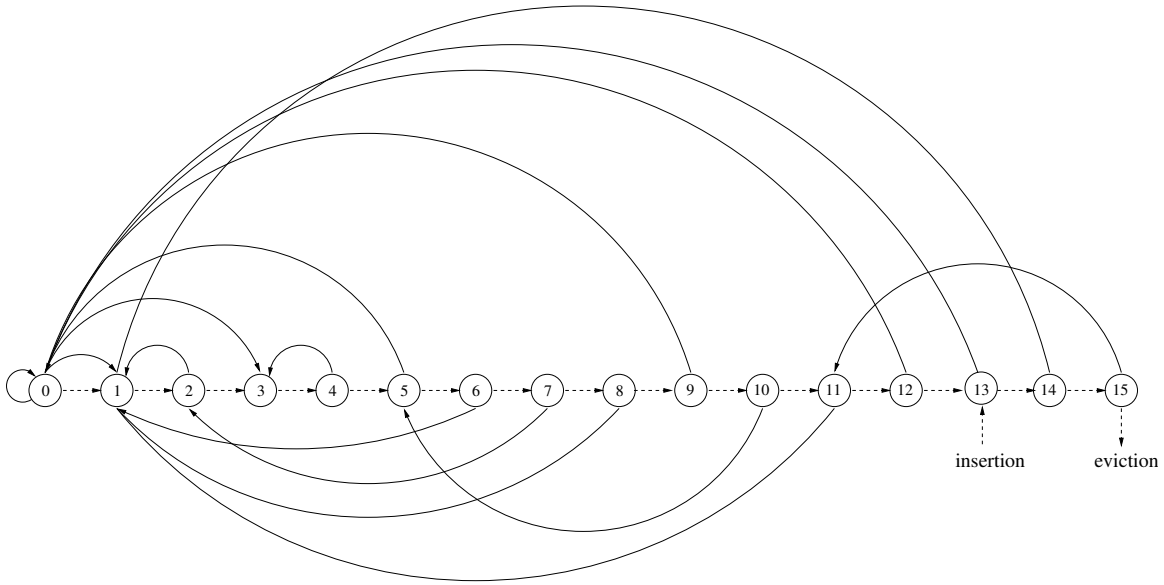

Figure 3: Transition Graph for Vector [0 0 1 0 3 0 1 2 1 0 5 1 0 0 1 11 13]

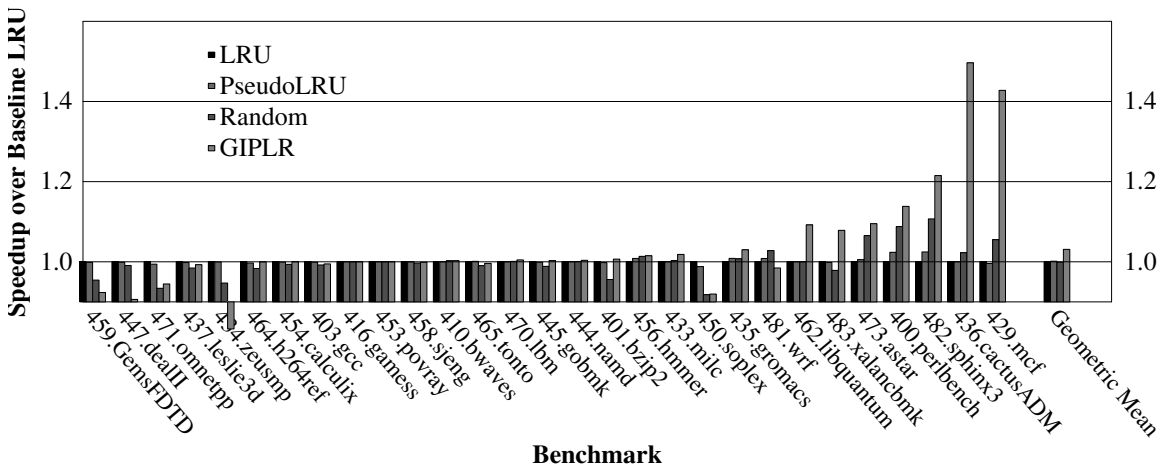

Figure 4: Speedup for the Vector [0 0 1 0 3 0 1 2 1 0 5 1 0 0 1 11 13]

```
find_plru ( r ) r is the root node of the binary tree
        p = r
        while ( p is not a leaf )
                if ( p's plru bit == 1 )
                        p = right child of p
                else
                        p = left child of p
                end if
        end while
        return the block associated with p
```

Figure 5: Algorithm to find Pseudo LRU block

```
promote ( p ) p is the leaf node associated with a block to be promoted
        while ( p is not the root )
                if ( p is a left child )
                        p's plru bit = 1
                else
                        p's plru bit = 0
                end if
                p = parent of p
        end while
```

Figure 6: Algorithm to promote a block to be PMRU

PseudoLRU is cheaper than LRU. Consider a k-way set associative cache, and assume k is a power of 2. A complete binary tree with k leaf nodes has $\sum_{d=0}^{\lceil \log_2 k \rceil - 1} 2^d = k - 1$ internal nodes. PseudoLRU requires storing one plru bit for each of the internal nodes. The structure of the complete binary tree is implicit in the indexing of this array of plru bits, so the only storage needed to represent the tree is the plru bits themselves. For a 16-way set associative cache, this is 15 bits per set as opposed to 64 bits per set for full LRU, a savings of 77%. In general, PseudoLRU reduces the number of bits required by a factor of $\log_2 k$ over LRU. PseudoLRU is also less complex to implement: an insertion or promotion can change only up to $\log_2 k$ plru bits on the path from the root to the leaf, as opposed to full LRU which can change all $k \log_2 k$ bits when a block is inserted or promoted from LRU to MRU. Thus, PseudoLRU is used in practice.

3.2 PseudoLRU Recency Stack

Blocks in a PseudoLRU set can be thought of as occupying a recency stack with each block holding a distinct position, just as in LRU. Blocks will be ordered from PMRU position 0 to PLRU position of all-bits-1 (e.g. 15 for 16-way). We would like to know the value of the position in the recency stack of a given block. Let us order the nodes visited from leaf (i.e. a block) to root starting with 0. If the i^{th} node is a right child, then the i^{th} bit in the position is the plru bit of that node's parent; in the eviction process, a 1 bit leads to this right child and a 0 bit leads away from it. If the i^{th} node is a left child, then the i^{th} bit in the position is the complement of the plru bit of that node's parent; in the eviction process, a 0 bit leads to this left child and a 1 bit leads away from it. Figure 7 gives pseudocode for the algorithm for determining the PLRU position of a given block[2]. Intuitively, more 1 plru bits in the position puts a block in increasing jeopardy of being evicted. For instance, if the root plru bit is 1, then every block in the right-hand side of the tree will have a 1 as the most-significant-bit of its position, meaning that those blocks are in the bottom half of the PseudoLRU recency stack. Figure 8 illustrates an example PseudoLRU tree with internal nodes giving their plru bit values and leaf nodes giving the corresponding blocks' positions in the PseudoLRU recency stack.

```
find_index ( p ) p is the leaf node associated with a block
        x = 0
        i = 0
        q = p
        while ( q is not the root )
                if ( q is a right child )
                        if ( q's parent's plru bit is 1 ) set bit i of x to 1
                else
                        if ( q's parent's plru bit is 0 ) set bit i of x to 1
                end if
                q = the parent of q
                i = i + 1
        end while
        return x
```

Figure 7: Algorithm to find the position in the PseudoLRU recency stack of a block p

3.3 Setting the PseudoLRU Position

Figure 9 gives the algorithm to set the PLRU position of a

[2]The inner loop of the algorithm setting bit i in the position can be more efficiently implemented as " set bit i of x to (q is a right child) == (q's parent's plru bit)" but we present the "if" version for clarity.

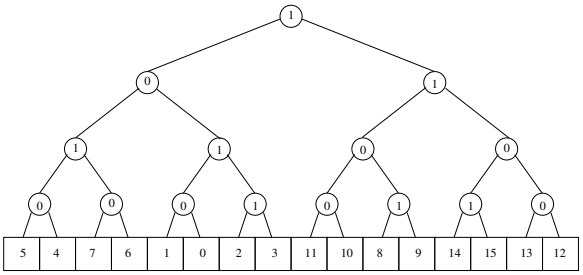

Figure 8: Example PseudoLRU Tree. Blocks are labeled with their positions in the PseudoLRU recency stack derived from internal node plru bit values.

```
set_index ( p, x )      p is the leaf node associated with a block,
    i = 0                       x is the new position to assign to p
    q = p
    while ( q is not the root )
        if q is a right child then
            q's parent's plru bit = bit i of x
        else
            q's parent's plru bit = not (bit i of x)
        endif
        q = the parent of q
        i = i + 1
    end while
    return x
```

Figure 9: Algorithm to set the position in the PseudoLRU recency stack of a block p to x

given block. We use this algorithm to implement improved insertion and promotion for PseudoLRU. The algorithm has a straightforward implementation in digital logic with little additional complexity over standard PseudoLRU [3]. The number of bits modified on an insertion or promotion is bound from above by $\lceil \log_2 k \rceil$ where k is the associativity, just as in standard PseudoLRU, and as opposed to $k\lceil \log_2 k \rceil$ modified bits for standard LRU.

3.4 Improved Insertion and Promotion for Tree-Based PseudoLRU

We apply the same technique to PseudoLRU as we did before to LRU. We evolve an insertion and promotion vector (IPV) giving the positions to insert incoming blocks and to promote accessed blocks. The insertions and promotions now change the positions in the PseudoLRU recency stack by setting bits along the path from the block to the root such that the new path represents the desired position as in Figure 9. This technique will have the side effect of changing other blocks' positions in a more drastic way than LRU, so we must evolve a new IPV that takes into account this different nature of PseudoLRU insertion and promotion. We call this new algorithm Genetic Insertion and Promotion for PseudoLRU Replacement (GIPPR).

3.5 Dynamic Adaptivity for GIPPR

There is no single IPV that is the best choice for all workloads. For instance, as previous work has observed, some workloads perform better with LRU insertion and some with MRU insertion. Thus, it makes sense to dynamically adapt the IPV to the currently running workload. One idea is to run the genetic algorithm on-line to continually adapt the IPV to the workload; although our preliminary experiments in this area have shown promise, the overhead in terms of area and power is prohibitive. Instead, we evolve several IPVs off-line and dynamically select between them at run-time using set-dueling.

Set-dueling was introduced by Qureshi *et al.* [26]. To choose between two cache management policies A and B, set-dueling allocates a small fraction of cache sets to implement policy A and another equally sized portion of sets to implement policy B. These sets are called "leader sets." An up/down counter is kept that counts up when policy A causes a miss in a leader set and down when policy B causes a miss in a leader set. The rest of the sets in the cache (the "follower sets") follow policy A when the counter is below 0, or B when the counter is at least 0. This way, most of the cache follows the "winning" policy. Since the behavior of a few sampled sets often generalizes to the rest of the cache, set-dueling usually finds the best policy. Other work has generalized set-dueling to multiple policies, e.g. Khan *et al.* introduced decision-tree analysis [15] to choose between a number of possible placement positions.

We develop DGIPPR, a dynamic version of GIPPR. We use set-dueling and decision-tree analysis to dynamically choose between two (2-DGIPPR) or four (4-DGIPPR) evolved IPVs. In the 2-vector case, we use set-dueling as described by Qureshi *et al.* [25], i.e. a single counter counts up for misses from leader sets for one IPV and down for leader sets for the other IPV while the cache follows the "winning" policy. In the 4-vector case, we use multi-set-dueling as described by Loh [22], i.e. two counters keep track of misses for leader sets from two pairs of IPVs, and those two counters duel using a meta-counter with the winning element of the winning pair used for the rest of the cache. We find that extending beyond four vectors yields diminishing returns, so in this research we limit the number of evolved vectors to four. We keep only one set of PseudoLRU bits per cache set even while changing the IPVs, rather than keeping different PseudoLRU bits per IPV which would have a costlier hardware overhead. In Section 4 we describe our methodology for evolving the vectors. To eliminate the impact of using the same training and testing data, we give *workload neutral* results showing the speedup of DGIPPR using cross-validation where a workload is not part of the training set used to evolve its own IPV. To show the potential of DGIPPR, we also evaluate the technique in an environment where all workloads use the same vectors; we find very little difference between the two methodological scenarios.

3.6 Overhead of GIPPR/DGIPPR

GIPPR/DGIPPR consumes space equivalent to that of PseudoLRU. For 16-way set associativity, GIPPR/DGIPPR consumes 15 bits per set (i.e. less than 0.94 bits per block), or 7KB for a 4MB cache. As with PseudoLRU, only the bits along the path from leaf to root are touched during an insertion, promotion, or eviction leading to few replacement-related switching events per cache access. The dynamic versions use counters for set-dueling. 2-DGIPPR uses a single 11-bit counter while 4-DGIPPR uses three 11-bit counters. That is, there are three 11-bit counters *per last-level cache*, not per block or set; only 33 bits are added to the entire microprocessor. Thus, these counters add negligible overhead. For comparison, LRU in a 16-way cache uses 4 bits

[3]Again, the *if* statement in the inner loop can be replaced with the more brief but less clear statement "q's parent's plru bit = (q is a right child) == (bit i of x)."

per block, or 32KB for a 4MB cache and must touch potentially all 16 recency positions during an insertion and promotion. DRRIP uses 2 bits per block, or 16KB and PDP used 3 or 4 bits per block, i.e. 24KB or 32KB, plus a number of bits and transistors related to the implementation of a microcontroller to compute protecting distances.

4. METHODOLOGY

In this section, we detail the experimental methodology. We discuss the techniques used to generate the insertion and promotion vectors (IPVs) as well as the simulation infrastructure.

4.1 Searching A Large Design Space

It is not immediately clear how to build a good IPV algorithmically. One idea is to simulate the effect of different IPVs over a set of workload traces, retaining the IPV that maximizes performance. With 16-way associativity, there are $16^{17} = 2.95 \times 10^{20}$ IPVs of 17 entries (16 promotion + 1 insertion) with 16 possible values per entry. One simple approach is to randomly search the design space. We allowed each value in the vector take a uniformly pseudo-random integer value from 0 through 15. We evaluated each combination using the fitness function described below, leading to Figure 1 showing the speedup of each of 15,000 IPVs sorted in ascending order of speedup. However, this very small sample of the design space took three days of simulation time and left significant potential undiscovered. Exhaustively searching the space of IPVs is impractical, so we turn to genetic algorithms to find good IPVs.

4.2 Using A Genetic Algorithm to Generate IPVs

Genetic algorithms are a well-known general optimization technique [9]. The basic idea is that a population of potential candidates are mated and mutated with the best individuals being propagated to the next generation. Each individual is evaluated with respect to a fitness function. After many generations, the algorithm tends to converge on a good solution. In the case of IPVs, the population consists of many randomly-generated IPVs. Individual IPVs are mated with crossover, i.e., elements $0..k$ of one vector and $k+1..16$ of another vector are put into corresponding positions of a new vector, where k is chosen randomly. For mutation, for each new IPV, with a 5% probability, a randomly chosen element of the vector is replaced with a random integer between 0 and 15. To generate a vector, we start with a population size of 20,000 in the first generation and drop the size to 4,000 for subsequent generations. We generate many such vectors through many runs in parallel. These vectors are used for the single-vector results.

We then use these vectors to seed another genetic algorithm implemented in pgapack [19], an open-source MPI-based approach to parallizing the genetic algorithm. We use a population size of 256 and allow the algorithm to run on 96 processors. The runs take about one day each to converge. We manually check for the appearance of convergence every few hours rather than use an automated stopping condition. For the workload neutral results, no vectors generated using data from a particular benchmark are used for that benchmark to avoid artificially giving that benchmark an unfair advantage. Using this methodology, we evolve IPVs for 2-DGIPPR and 4-DGIPPR.

4.3 Fitness Function

The fitness function for our genetic algorithm is the average speedup obtained on a simplified cache model. For each workload under consideration, we use a modified version of Valgrind [23] to collect traces representing each last-level cache access for each simpoint of 1.5 billion instructions. We run these traces through a last-level cache simulator that uses a replacement policy driven by the IPV under consideration. We use the records from the first 500 million instructions to warm the cache, and the next billion instructions to measure the number of misses. We estimate the resulting cycles-per-instruction (CPI) as a linear function of the number of misses. We then compute the fitness function as the speedup of this CPI over the CPI estimated when LRU is used as the replacement policy.

This fitness function can be evaluated much more quickly than performing a full simulation. On our machines, the fitness function takes about 5 minutes to complete for all 92 simpoints of the 29 SPEC CPU 2006 benchmarks, as opposed to hours with a performance simulator. However, the fitness function cannot take into account the effects of memory-level parallelism or other effects of out-of-order execution.

4.4 Workload Neutral Vectors

If we use a given workload as part of the fitness function for developing an IPV, then the performance on that workload might be artificially higher than we would expect for an arbitrary workload encountered in practice. Thus, in order to provide a fair evaluation of GIPPR/DGIPPR, we develop *workload neutral* IPVs for each of the 29 SPEC CPU 2006 benchmarks. A common practice in machine learning is to use *cross-validation*, where part of a set of data is held back and not used for training but rather for validation. Our *workload neutral k* (WNk) methodology uses the same idea. In general, with n workloads, a WNk methodology would hold out k workloads, using the other $n-k$ workloads to generate IPVs, then use the IPVs to evaluate GIPPR/DGIPPR on the first k workloads. This methodology allows us to eliminate any bias in the IPVs for a particular workload that would unfairly cause our technique to yield more performance than it would in practice.

There is a trade-off between the number of workloads held out (k) and the quality of the IPVs generated. With $k = 1$ (i.e. WN1), we find the best quality IPVs taking into account all workloads except for one, then use that IPV to evaluate the performance on that one.

For completeness, we also report results with IPVs obtained using all 29 workloads (i.e. workload inclusive or WI). The resulting performance is slightly better than the WN1 results.

4.5 Performance Simulation

We use a modified version of CMP$im, a memory-system simulator that is accurate to within 4% of a detailed cycle-accurate simulator [12]. The version we used was provided with the JILP Cache Replacement Championship [2]. It models an out-of-order 4-wide 8-stage pipeline with a 128-entry instruction window. This infrastructure enables collecting instructions-per-cycle figures as well as misses per kilo-instruction and dead block predictor accuracy. We use the following memory hierarchy parameters: L1 data cache: 32KB 8-way associative, L2 unified cache: 256KB 8-way L3:

4MB, 16-way, DRAM latency: 200 cycles.

4.6 Workloads

We use the SPEC CPU 2006 benchmarks. Each benchmark is compiled for the 64-bit X86 instruction set. The programs are compiled with the GCC 4.1.0 compilers for C, C++, and FORTRAN. We use SimPoint [24] to identify up to 6 segments (i.e. *simpoints*) of one billion instructions each characteristic of the different program phases for each workload. The results reported per benchmark are the weighted average of the results for the individual simpoints. The weights are generated by the SimPoint tool and represent the portion of all executed instructions for which a given simpoint is responsible. Each program is run with the first `ref` input provided by the `runspec` command.

4.7 Techniques Simulated

We report results for workload neutral and workload inclusive IPVs for GIPPR, 2-DGIPPR, and 4-DGIPPR. We compare these results with LRU as well as DRRIP [13] and Protecting Distance based Policy (PDP) [6]. We use versions of DRRIP and PDP with C++ files graciously provided by the respective authors that work in the CMP$im infrastructure. DRRIP is another space-efficient replacement policy that uses set-dueling and improves significantly over LRU. PDP computes protecting distances stored to cache lines to determine how long a block should be protected from being evicted. We configure PDP to use 4 bits per block and to not bypass the cache. PDP requires an additional storage overhead related to computing protecting distances as well as an additional microcontroller used to run the algorithm that computes protecting distances.

For measuring the number of misses, we also use an in-house trace-based last-level cache simulator that computes the optimal number of misses using Belady's MIN algorithm [3]. MIN chooses to evict the block referenced farthest in the future. We do not implement MIN in the performance simulator and indeed believe that it is not well-defined in a system that allows out-of-order issue of loads and stores, since the block referenced farthest in the future may depend on the latencies of the accesses to previous blocks.

5. EXPERIMENTAL RESULTS

This section gives the results of our experiments with GIPPR/DGIPPR. We show some of the IPVs generated and show the speedup and misses per one thousand instructions (MPKI) for the various techniques. In the bar charts, the benchmarks are sorted in ascending order of the statistic being measured for DRRIP.

5.1 Cache Misses

Figure 10 shows the misses per 1000 instructions (MPKI) normalized to the baseline LRU misses for 1, 2, and 4-vector workload neutral versions of GIPPR/DGIPPR. Also shown are misses given by the optimal replacement policy. The single-vector WN1-GIPPR technique incurs a geometric mean 95.2% of the misses of LRU. The two-vector WN1-2-DGIPPR yields 96.5% of the misses of LRU. The four-vector WN1-4-DGIPPR technique gives a 91.0% of the misses of LRU. The optimal replacement policy, i.e. the result using Belady's MIN algorithm [3], yields only 67.5% of the misses of LRU, showing that there is significant room for improvement in future work on replacement policy.

This graph shows that the 4-vector version of DGIPPR is probably the best configuration for actual implementation. As we can see, the benchmark `456.hmmer` is case where WN1-2-DGIPPR performs significantly worse than the other techniques, but WN1-4-DGIPPR is close to optimal. Thus, we recommend that PseudoLRU insertion and promotion be deployed using at least four IPVs as in WN1-4-DGIPPR.

Figure 11 shows the normalized MPKI for WN1-4-DGIPPR compared with DRRIP and PDP. The four-vector WN1-4-DGIPPR technique gives a 91.0% of the misses of LRU. DRRIP gives a comparable 91.5% of the misses of LRU while consuming more than twice the number of bits for replacement information; DRRIP uses two bits per block, or 16KB for a 4MB cache, while GIPPR/DGIPPR uses 15 bits per set, or 7KB for a 4MB cache. PDP achieves a slightly better 90.2% of the misses of LRU while consuming four times the number of bits as well as area devoted to a specialized microcontroller.

Misses are not increased over LRU for all the techniques for most workloads. `447.dealII` is a notable exception: its misses are increased greatly over LRU for each of DRRIP, PDP, and WN1-4-DGIPPR, with PDP faring better than the others. If such workloads are commonplace in a targeted environment, designers may consider implementing full LRU and dueling between the more high-performance policies and LRU.

For several of the SPEC CPU 2006 benchmarks the optimal replacement policy performs no better than LRU. For example, for `416.gamess` and `453.povray`, MIN, LRU, and all of the other replacement policies deliver about the same number of misses. However, for most workloads, the optimal policy is significantly better than LRU whether or not the other policies are affected.

5.2 Speedup

5.2.1 Workload Neutral versus Workload Inclusive

Figure 12 shows the difference between the workload neutral and workload inclusive versions of GIPPR/DGIPPR. Workload neutral results use IPVs for a given workload that were trained using only other workloads to provide a fair means of estimating the performance of the technique in practice. Workload inclusive results use all of the workloads for training all of the IPVs. The geometric mean different between the two kinds of results is small. The single-vector WN1-GIPPR technique achieves a geometric mean 3.47% speedup over LRU while the workload inclusive version yields a slightly better 3.68% speedup. The two-vector WN1-2-DGIPPR yields a 4.96% speedup over LRU while the workload inclusive version has a 5.12% speedup. The workload neutral 4-vector technique (WN1-4-DGIPPR) yields a speedup of 5.61% over LRU versus 5.66% for the workload inclusive technique (WI-4-DGIPPR). The most significant difference is in `436.cactusADM` where the workload neutral technique yields a 39% speedup but the workload inclusive technique yields a 49% speedup. Some workloads such as `447.dealII` and `483.xalancbmk` experience very slight reductions in performance using the workload inclusive technique. We believe this seemingly paradoxical situation arises because the genetic algorithm cannot actually give the optimal vector for a given set of traces. Moreover, the fitness function currently used cannot take into account the effects of memory-level parallelism which can have a sig-

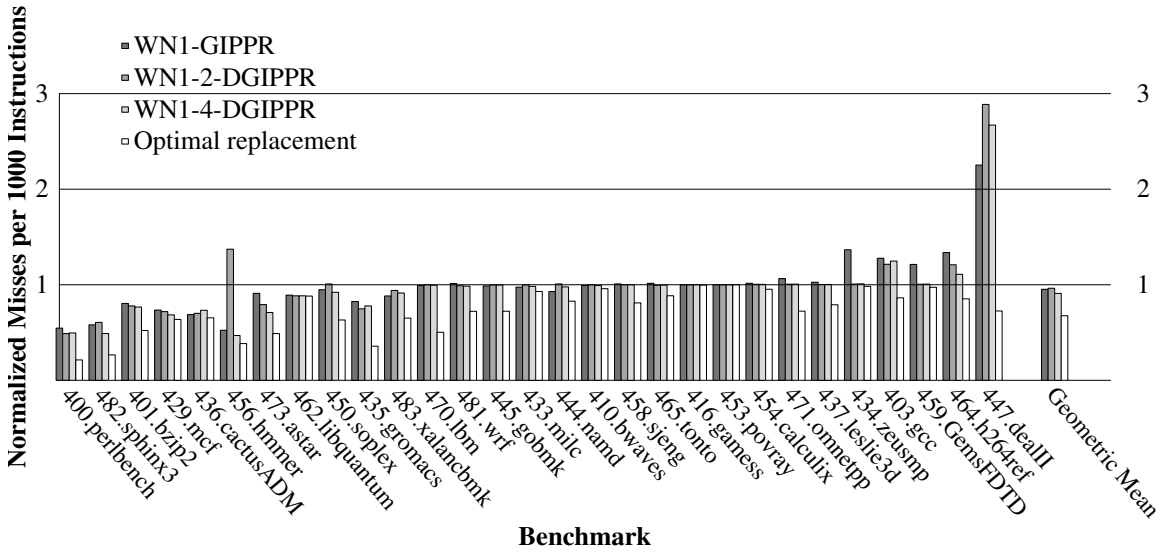

Figure 10: Misses per 1000 Instructions Normalized to LRU Misses

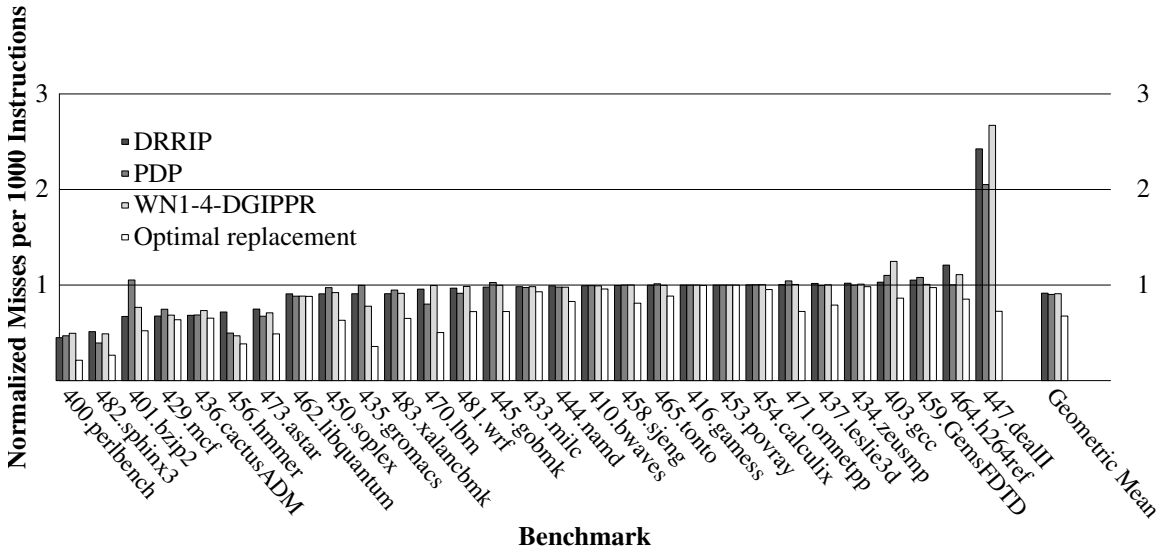

Figure 11: Misses per 1000 Instructions Normalized to LRU Misses

nificant impact on the best replacement decision for some workloads [26].

5.2.2 DGIPPR versus Other Techniques

We compare the 4-vector workload neutral technique (WN1-4-DGIPPR) with other techniques. Figure 13 illustrates the speedup of the various techniques over the baseline LRU policy. The workload neutral versions of 4-DGIPPR is used to provide a fair comparison to DRRIP. The four-vector WN1-4-DGIPPR technique gives a 5.61% speedup over LRU. DRRIP gives a comparable 5.41% speedup over LRU. PDP achieves a similar 5.69% speedup over LRU. We cannot report the results of optimal replacement because the MIN algorithm is not well-defined in a system that allows out-of-order issue of loads and stores.

Our technique delivers more consistent speedup over LRU than the other two. For `447.dealII`, WN1-4-DGIPPR achieves only 97.3% of the performance of LRU. For all other workloads, the performance is more than 99% of LRU. DRRIP achieves less than 99% of the performance of LRU on three workloads: `459.GemsFDTD`, `447.dealII`, and `471.omnetpp`. PDP yields less than 96% of the performance of LRU for `459.GemsFDTD` and `471.omnetpp`.

We consider a memory-intensive subset of SPEC defined as workloads where the speedup of DRRIP over LRU exceeds 1%. These are the workloads on the x-axis of Figure 13 from `433.milc` through `429.mcf`. For this subset, WN1-4-DGIPPR achieves a speedup of 15.6%, compared with the same 15.6% for DRRIP and a slightly higher 16.4% for PDP. In other words, our technique achieves the same performance as DRRIP with half the storage overhead, and 95% of the same performance as PDP with a small fraction of the com-

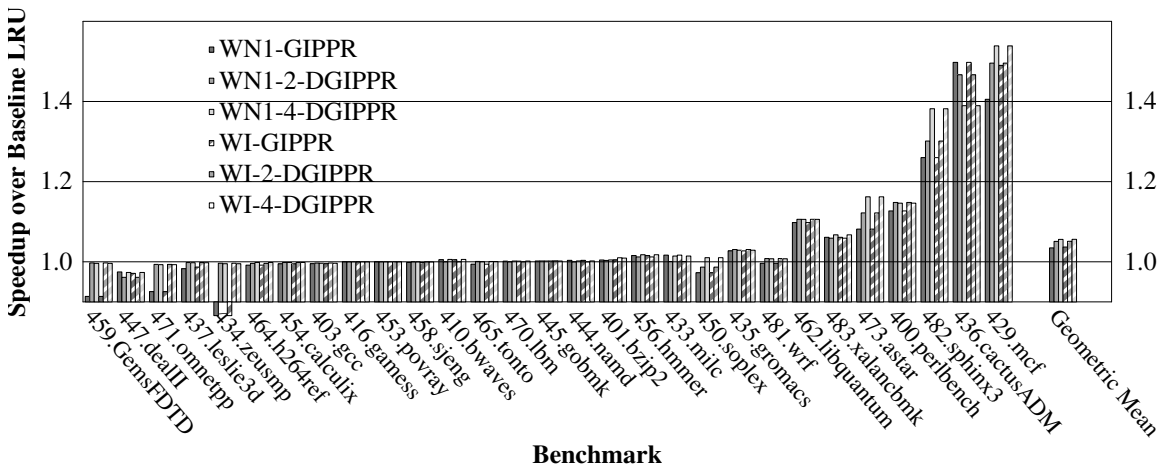

Figure 12: Workload neutral versus workload inclusive speedup

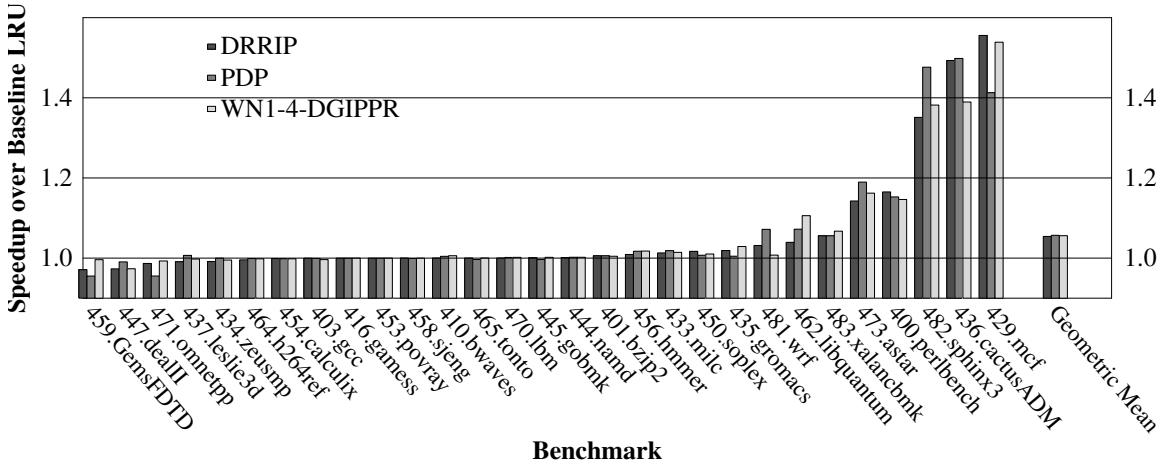

Figure 13: Speedup

plexity.

5.3 Vectors Learned

The workload inclusive IPV learned for GIPPR is [0 0 2 8 4 1 4 1 8 0 14 8 12 13 14 9 5]. Space prohibits us from showing all of the workload neutral vectors learned by the genetic algorithms, but as an example the best single workload neutral vector for `400.perlbench` is [12 8 14 1 4 4 2 1 8 12 6 4 0 0 10 12 11]

The set of two vectors for WI-2-DGIPPR is:

```
[ 8 0 2 8 12 4 6 3 0 8 10 8 4 12 14 3 15 ]
[ 0 0 0 0  0 0 0 0 8 8 8  8 0  0  0 0  0 ]
```

The set of four vectors for WI-4-DGIPPR is:

```
[ 14  5  6  1 10 6  8 8 15 8  8 14 12 4 12  9  8 ]
[  4 12  2  8 10 0  6 8  0 8  8  0  2 4 14 11 15 ]
[  0  0  2  1  4 4  6 5  8 8 10  1 12 8  2  1  3 ]
[ 11 12 10  0  5 0 10 4  9 8 10  0  4 4 12  0  0 ]
```

5.3.1 About Intuition

We are happy to provide all of the vectors used for this study to any interested party. At a first glance they do not seem to yield a deep understanding of why the GIPPR technique works as well as it does; they are jumbles of numbers. The vectors can be viewed as a highly compressed representation of clusters of program behaviors leading to good replacement decisions, and as such their detailed analysis is beyond the scope of this paper.

Much previous work places a high value on the intuition behind a given technique, and provide the results of studies performed to determine whether the intuition is valid. We applaud this sort of work, but also wonder whether other avenues that are counter-intuitive or non-intuitive may be of equal or perhaps greater value. After all, let us recall that the LRU policy itself is quite intuitive: a block that was referenced least recently should be least likely to be referenced again soon. However, LRU is so bad at keeping the right data in the LLC that there is not much to recommend it over random replacement. We believe that going beyond human intuition and letting the machine take over the design of cache management and perhaps other areas has great potential to improve processor design.

5.3.2 Interpreting the Vectors

Notwithstanding our viewpoint on intuition, let us attempt a preliminary interpretation of these insertion/placement vectors. We can observe some patterns in the vectors. For instance, the WI-2-DGIPPR IPVs clearly duel be-

tween PLRU and PMRU insertion, just as DIP [25] would do. However, the first vector for WI-2-DGIPPR seems to prefer a very pessimistic promotion policy, moving most referenced blocks closer to the PLRU position. On the other hand, the second vector is very close to PLRU by itself, with the exception of a block beginning at position 8 where referenced blocks in positions 8 through 12 go back to position 8. The WI-4-DGIPPR IPVs switch between PLRU, PMRU, close to PMRU, and "middle" insertion. The promotion strategies are quite varied as well.

6. RELATED WORK

Cache replacement has a long history in the literature. The optimal replacement policy in terms of minimizing cache misses is Belady's MIN [3]. The MIN algorithm replaces the block that will be used farthest in the future. This algorithm is impractical to implement due to its dependence on perfect knowledge of the future. Thus, many algorithms have been proposed based on intuitions that attempt to approximate MIN. The classic random, first-in-first-out (FIFO), and least-recently-used (LRU) replacement policies have been known for over 45 years [5]. There has been voluminous subsequent work on improving cache replacement algorithms; a complete survey could fill several MICRO papers worth of pages. Thus, we focus on the most relevant recently introduced policies for last-level cache replacement. LRU is described in more detail in Section 2.1.2.

6.1 Insertion Policy

Recent work has focused on LRU-like policies that modify the handling of the recency stack through insertion and promotion.

Qureshi *et al.* propose Dynamic Insertion Policy (DIP) [25]. That begins with the observation that some workloads perform better when incoming blocks are inserted in the LRU position. In some workloads, many incoming blocks are not used again. If a block is not referenced again before another victim is needed, it can be quickly eliminated from the cache by being chosen as the LRU victim. Since this strategy works poorly for other workloads that have less streaming behavior, the policy uses set-dueling (described in Section 2.3) to choose between LRU placement and the traditional MRU placement. DIP focuses only on insertion policy; promotion is the same as in traditional LRU. DIP relies on the costly LRU replacement policy which consumes an extra 4 bits per block in a 16-way set associative cache; by contrast, DGIPPR uses less than one bit per block as it is based on tree PseudoLRU.

Jaleel *et al.* propose re-reference interval prediction (RRIP) [13]. The paper introduces several techniques, but the main contribution of the work is Dynamic RRIP (DRRIP). Each block has associated with it a 2-bit re-reference prediction value (RRPV). An RRPV is roughly analogous to a coarse-grained position in an LRU recency stack, although more than one block can have the same RRPV. When a block is accessed, its RRPV is updated to 0. On a miss, a victim with an RRPV of 3 is chosen. If there is no such block, all blocks' RRPVs are incremented until an RRPV of 3 is found. DRRIP uses set-dueling to choose between two techniques: SRRIP and BRRIP. SRRIP inserts blocks with an initial RRPV of 2. BRRIP usually inserts with an RRPV of 3, but with low probability will insert with an RRPV of 1. To our knowledge, DRRIP is the most efficient replacement policy in the literature that uses no additional information other than the stream of block addresses accessing cache sets. DRRIP requires 2 extra bits per cache block, making it the most efficient of the published high-performance cache replacement schemes.

6.2 Promotion Policy

Kron *et al.* extend the idea of DIP in DoubleDip [18]. DIP is combined with an adaptive promotion policy that promotes blocks an increment up the recency stack proportional to the strength of set-dueling confidence counters. Xie and Loh further extend this line of thought with promotion/insertion pseudo-partitioning (PIPP) targeted at very large capacity highly associative caches such as those constructed with die-stacked DRAM [30].

6.3 Other Replacement Work

Kaxiras *et al.* propose directly predicting the re-use distance of a given block and using that prediction to guide cache replacement [14]. This work uses a sampling methodology based on the PC of memory access instructions, inspiring sampling-based dead block prediction [16]. Dead block prediction [4, 20, 16] also uses the address of memory access instructions to predict whether a block will be used again before it is evicted. Dead block prediction can be used to drive replacement policy by evicting predicted dead blocks [17, 16], but the implementation is costly in terms of state and/or the requirement that the address of memory instructions be passed to the LLC.

Wu *et al.* propose a signature-based hit predictor (SHiP) as a significant improvement to DRRIP. The idea is to use the address of the memory access instruction to give a prediction of the re-reference behavior of the data accessed by that instruction. Based on this prediction, a more intelligent insertion decision can be made resulting in significantly improved performance. The resulting cache management policy uses 5 extra bits per cache block. This work also depends on the address of memory access instructions being available to the last-level cache, which would require an extra communication channel from the cores to the LLC.

Duong *et al.* propose using dynamic reuse distances instead of recency stack positions for cache replacement [6]. Their proposal results in the Protecting Distanced based Policy (PDP) that keeps a cache line from being evicted until after a certain number of accesses to the set. The policy makes use of *bypass*, i.e., if an analysis shows that a particular cache line is unlikely to be reused, it bypasses the last-level cache. This work surpasses DRRIP's performance but uses significantly more state than DRRIP. When used with bypass, PDP is unsuitable for use with an inclusive cache because bypass necessarily violates inclusion; however, PDP can be used without bypass in which case it still provides an improvement over DRRIP. This work requires 3 or 4 bits (depending on the configuration) of extra state per block. The technique also requires the incorporation of a specialized microcontroller consuming 10K NAND gates to compute protecting distances.

6.4 Genetic Algorithms in Architecture

Emer and Gloy use genetic algorithms to design accurate branch predictors [8]. They develop a language of capable of describing branch predictors, then use a genetic algorithm to search through the space of sentences in that language

to find a branch predictor with very good accuracy. Gomez *et al.* use neuroevolution, i.e. genetic algorithms to evolve neural networks, to control resource allocation on a chip-multiprocessor [10]. Ebrahimi *et al.* use a genetic algorithm to optimize parameters for a prefetching algorithm [7].

7. CONCLUSION AND FUTURE WORK

We have presented GIPPR and DGIPPR, last-level cache replacement policies based on changing insertion and promotion for PseudoLRU. We have demonstrated that DGIPPR is competitive in terms of performance with DRRIP and PDP while consuming far less overhead. We seek to extend the work in a number of ways:

1. The low overhead of GIPPR/DGIPPR may allow it to be combined with other policies through set-dueling with an acceptable overhead. For instance, we are investigating combining DGIPPR with a predictor that decides whether a block should bypass the cache.

2. We are currently working on ways to take MLP into account in the fitness function.

3. We use a genetic algorithm to develop the vectors, but we are investigating ways to find these vectors more systematically.

4. We have demonstrated the technique on single-threads workloads, but we are actively researching extending it to multi-core.

5. Although we have focused on PseudoLRU insertion and promotion, the full LRU version of the technique also deserves further study, and it may be adapted to other LRU-like algorithms such as RRIP.

6. We would like to explore the performance of our technique at high levels of associativity; a high-performance replacement policy should complement techniques such as zCache [27] that provide high effective associativity with low overhead.

7. Finally, we are investigating generalizations of insertion and promotion policies beyond IPVs.

For years the disparity between processor and memory speeds, i.e. the "memory wall," has stymied progress in improving the performance of microprocessors. Our hope is that techniques such as DGIPPR will tear down this wall.

8. ACKNOWLEDGEMENTS

This research was supported by Texas NHARP grant 010115-0079-2009 and National Science Foundation grants CCF-1216604 and CCF-1012127. I thank Brian Grayson, Gabe Loh, and Moin Qureshi for their very helpful feedback on preliminary versions of this research. I would also like to thank Yale Patt who, in a happy coincidence, happened to show up to a somewhat randomly scheduled talk I gave based on this research in Spain. His boisterous participation in the discussion has inspired my future research in this area.

I would like to acknowledge George Gipp for inspiring the silly name for my technique. Gipp was the quarterback for Notre Dame's Fighting Irish and is reported to have said to Coach Knute Rockne, "Some time, Rock, when the team is up against it, when things are wrong and the breaks are beating the boys, ask them to go in there with all they've got and win just one for the Gipper." [1] Later, Ronald Reagan played the role of Gipp in a movie, and much later adopted the moniker "The Gipper" in his political life. Reagan immortalized Gipp's words as he implored presidential candidate George H.W. Bush to "win one for the Gipper" (WN1-4DGIPPR).

9. REFERENCES

[1] *The Homiletic Review*. Funk & Wagnalls, 1931, no. v. 102.

[2] A. R. Alameldeen, A. Jaleel, M. Qureshi, and J. Emer, "1st JILP workshop on computer architecture competitions (JWAC-1) cache replacement championship," http://www.jilp.org/jwac-1/.

[3] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[4] A. chow Lai, C. Fide, and B. Falsafi, "Dead-block prediction and dead-block correlating prefetchers," in *In Proceedings of the 28th International Symposium on Computer Architecture*, 2001, pp. 144–154.

[5] P. J. Denning, "The working set model for program behavior," *Commun. ACM*, vol. 11, no. 5, pp. 323–333, May 1968.

[6] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum, "Improving cache management policies using dynamic reuse distances," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 389–400.

[7] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated control of multiple prefetchers in multi-core systems," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 316–326.

[8] J. Emer and N. Gloy, "A language for describing predictors and its application to automatic synthesis," in *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997, pp. 304–314.

[9] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, 1st ed. Addison-Wesley Professional, January 1989.

[10] F. Gomez, D. Burger, and R. Miikkulainen, "A neuroevolution method for dynamic resource allocation on a chip multiprocessor," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN-01)*, July 2001.

[11] J. Handy, *The Cache Memory Book*. Academic Press, 1993.

[12] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob, "CMP$im: A pin-based on-the-fly single/multi-core cache simulator," in *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS 2008)*, June 2008.

[13] A. Jaleel, K. Theobald, S. S. Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," in *Proceedings of the 37th*

Annual International Symposium on Computer Architecture (ISCA-37), June 2010.

[14] G. Keramidas, P. Petoumenos, and S. Kaxiras, "Cache replacement based on reuse-distance prediction," in *In Proceedings of the 25th International Conference on Computer Design (ICCD-2007)*, 2007, pp. 245–250.

[15] S. M. Khan and D. A. Jiménez, "Insertion policy selection using decision tree analysis," in *In Proceedings of the 28th IEEE International Conference on Computer Design (ICCD-2010)*, October 2010.

[16] S. M. Khan, Y. Tian, and D. A. Jimenez, "Sampling dead block prediction for last-level caches," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 175–186.

[17] M. Kharbutli and Y. Solihin, "Counter-based cache replacement and bypassing algorithms," *IEEE Transactions on Computers*, vol. 57, no. 4, pp. 433–447, 2008.

[18] J. D. Kron, B. Prumo, and G. H. Loh, "Double-dip: Augmenting dip with adaptive promotion policies to manage shared l2 caches," in *Proceedings of the 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects (CMP-MSI)*, 2008.

[19] D. Levine, "Users guide to the pgapack parallel genetic algorithm library," Argonne National Laboratory Report ANL-95/18, Tech. Rep., January 1996.

[20] H. Liu, M. Ferdman, J. Huh, and D. Burger, "Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. Los Alamitos, CA, USA: IEEE Computer Society, 2008, pp. 222–233.

[21] A. Loeb, "Cosmology with Hypervelocity Stars," *JCAP*, vol. 1104, p. 023, 2011.

[22] G. H. Loh, "Extending the effectiveness of 3d-stacked dram caches with an adaptive multi-queue policy," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 201–212.

[23] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 89–100.

[24] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 1, pp. 318–319, 2003.

[25] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. S. Jr., and J. 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.

[26] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A case for mlp-aware cache replacement," in *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 167–178.

[27] D. Sanchez and C. Kozyrakis, "The zcache: Decoupling ways and associativity," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 187–198.

[28] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[29] *SPEC CPU 2006, http://www.spec.org/cpu2006*. Standard Performance Evaluation Corporation, August 2006.

[30] Y. Xie and G. H. Loh, "Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches," in *International Symposium on Computer Architecture*, 2009, pp. 174–183.