Minimal Disturbance Placement and Promotion

Elvira Teran† Yingying Tian‡* Zhe Wang§* Daniel A. Jiménez†
†Texas A&M University ‡Advanced Micro Devices, Inc. §Intel Labs
†{eteran,djimenez}@tamu.edu ‡Yingying.Tian@amd.com §zhe2.wang@intel.com

ABSTRACT

Cache replacement policies often order blocks into distinct positions. A block is placed into a set in some initial position. A re-referenced block is promoted into a higher position while other blocks may move into lower positions. A block in the lowest position is a candidate for replacement. Tree-based PseudoLRU is a well-known space-efficient replacement policy based on representing block positions as distinct paths in a binary tree. We find that a placement or promotion for one block often needlessly disturbs the non-promoted blocks. Guided by the principle of minimal disturbance, *i.e.* that a policy should seek to disturb the order of non-promoted blocks to the smallest extent possible, we develop a simple modification to PseudoLRU resulting in a policy that improves performance over previous techniques while retaining the low cost of PseudoLRU. The result is a minimal disturbance placement and promotion (MDPP) policy.

We first give a static formulation of MDPP and show that it provides superior performance to LRU, PseudoLRU and matches performance for SRRIP for both single-threaded and multi-core workloads. We then give a dynamic formulation that uses dead block prediction for placement and bypass and show that it meets or exceeds state-of-the-art performance with lower overhead. For single-threaded workloads, dynamic MDPP matches the 5.9% speedup over LRU of the state-of-the-art policy SHiP. For multi-core workloads, dynamic MDPP gives a normalized weighted speedup of 14.3% over LRU, compared with SHiP that yields a speedup of 12.3% over LRU and requires double the storage overhead per set. We show that minimal disturbance policies can reduce the frequency of a costly read-modify-write cycle for replacement state, making them potentially suitable for future work in DRAM caches.

1. INTRODUCTION

With the disparity between cache latency and memory latency, a replacement policy that reduces misses can significantly improve performance. Many replacement policies order blocks into distinct positions, choosing to replace the block in a least favorable position when there is a miss to a set. Positions are manipulated by the placement and promotion policies. The placement policy decides what the initial position of a block should be, while the promotion policy decides what the new position of a re-referenced block should be. To accommodate a block in a new position, often other blocks must change their positions. These changes can disturb non-referenced blocks, causing them to move closer to the least favorable position by an amount that may be out of proportion to their merit. Previous work focuses on what happens to the just-referenced block without regard to the effect on the other, non-referenced blocks.

Figure 1 shows the average change in position of non-referenced blocks for two common replacement policies: least-recently-used (LRU) replacement policy as well as tree-based PseudoLRU over a set of 100 multi-core workloads described in Section 5. An evicted block travels one position beyond the maximum value, *i.e.* it moves out of the set. The LRU policy is relatively well-behaved, moving non-referenced blocks no more than one position and often less than one position when a block is promoted or placed. Tree-based PseudoLRU, a widely-used space efficient policy, moves blocks' positions much further than LRU, and the movement varies wildly with the non-referenced block's original position.

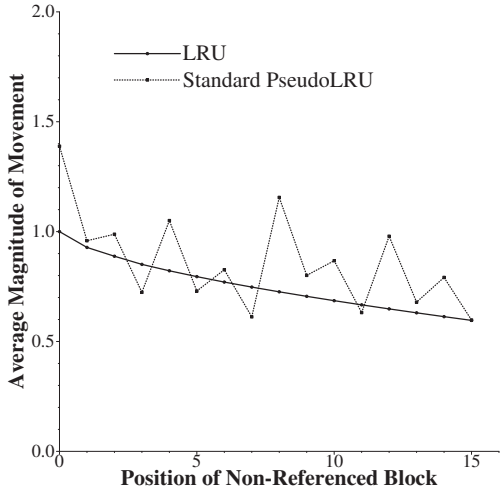

Figure 1: Average change in position caused by a promotion for the non-promoted blocks

In this paper, we develop a replacement policy that balances the

*Yingying Tian and Zhe Wang contributed to this work while they were students at Texas A&M University.

HPCA 2016 March 12–16, 2016, Barcelona, Spain.
978-1-4673-9211-2/16/$31.00 ©2016 IEEE.

risk of a promoted block being evicted with the harm caused to the other blocks by the disturbance caused by the promotion.

Widrow and Lehr introduced the *minimal disturbance principle* applied to neural network learning. The principle states that "during training it is advisable to inject new information into a network in a manner that disturbs stored information to the smallest extent possible" [25]. The internal node values of a PseudoLRU tree are roughly analogous to the weights of a neural network in that they learn over time information about block locality. On a block placement or promotion, the values are updated in a way that records new information about one block while risking disturbance to the information of the other blocks. Guided by the minimal disturbance principle, we propose a technique that promotes referenced blocks in a way that affords them the protection they need from being evicted while minimizing disturbance to the other blocks.

The result is the Minimal Disturbance Placement and Promotion (MDPP) policy. We give a simple static MDPP policy that significantly outperforms true LRU and matches the SRRIP policy with less complexity, and a dynamic MDPP policy that outperforms the state-of-the-art on multi-core workloads. Dynamic MDPP achieves a normalized weighted speedup of 14.3% on multi-core workloads over LRU while SHiP [26] achieves an 12.3% speedup.

2. BACKGROUND AND RELATED WORK

To put this work in the proper context and give background on the technique we build upon, we review related work.

2.1 Placement Policy

The LRU policy places blocks into the MRU position. Qureshi *et al.* observe that changing the placement position to LRU sometimes results in an improvement as blocks that receive no new hits are quickly eliminated from the cache [21]. They propose dynamic insertion policy (DIP) that determines at run-time whether LRU or MRU placement works best for a particular workload. Jaleel *et al.* propose a thread-aware version of DIP, TADIP [8] that improves over DIP as well as previous thread-aware cache management policies. Khan and Jiménez propose decision tree analysis for choosing from a wider range of placement positions [12]. Xie and Loh propose PIPP, a dynamic promotion and placement policy for pseudo-partitioning multi-core caches [27]. The dynamic MDPP policy adaptively changes placement position based on dead block prediction, while both static and dynamic MDPP policies also use an optimized promotion policy.

2.2 Re-Reference Prediction

Much recent work on last-level cache replacement has focused on predicting whether and when a block will be re-referenced. We describe three relevant techniques.

RRIP.

Re-reference Interval Prediction (RRIP) [9] is an efficient implementation of reuse-distance prediction [11]. RRIP categorizes blocks as near-immediate re-reference interval, intermediate re-reference intervals, and distant re-reference interval [9], associating these positions with cache blocks in a way roughly analogous to the positions in other replacement policies. Static RRIP (SRRIP) always places into the same position (*e.g.* distant re-reference interval) while Dynamic RRIP (DRRIP) uses set-dueling [23] to adapt the placement position to the particular workload. RRIP is a simple policy, requiring little overhead and no complex prediction structures, while resulting in significant improvement in performance.

Dead Block Prediction.

Dead block predictors predict whether a block will be used again before it is evicted. There are numerous dead block predictors applied to a variety of applications in the literature [1, 6, 13–17, 24]. We make use of Sampling Dead Block Prediction (SDBP) of Khan *et al.* [13]. In this work, a *sampler* keeps partial tags of sampled sets. Three tables of two-bit saturating counters are accessed using a technique similar to a skewed branch predictor [19]. For each hit in the sampled set, the program counter (PC) of the relevant memory instruction is hashed into the three tables and the corresponding counters are decremented. For each eviction from a sampled set, the counters corresponding to the PC of the last instruction to access the victim block are incremented. For any access to the LLC, the predictor may be consulted by hashing the PC of the memory access instruction into the three tables and taking the sum of the indexed counters. When the sum exceeds some threshold, the accessed block is predicted to be dead. Tags in sampled sets are managed with true LRU and a reduced associativity, but the LLC may be managed by any policy. The paper applies dead block prediction to replacement and bypass, but in this work we use the predictor for determining placement position and bypass.

Signature-Based Hit Prediction.

Wu *et al.* [26] propose a dead block predictor with a structure similar to SDBP. SHiP uses a table of 3-bit saturating counters indexed by a signature of the block, *e.g.* the memory instruction PC that caused the initial fill of the block. SHiP is called a "hit predictor" as the counters count up instead of down on a hit, and down instead of up on an eviction. Thus, above some threshold, the block is predicted to be hit soon and thus not dead. The predictor is used to determine placement position for RRIP. When a block is predicted not to be dead, its placement position is "intermediate re-reference interval," otherwise is it "distant re-reference interval."

2.3 Tree-based PseudoLRU Placement and Promotion

Genetic Insertion and Promotion for PseudoLRU Replacement (GIPPR) uses genetic algorithms to find *insertion and promotion vectors* (IPVs) for tree-based PseudoLRU [10]. The vectors give the placement position for new blocks as well as the position to which a block in a given original position should be promoted. A single vector does not generalize well to different workloads, so the paper proposes to supply the policy with four vectors from which one is chosen at run-time using set-dueling. The approach of that work is antithetical to our work: it begins with a blank slate on which the trace-driven genetic algorithm writes a novel placement and promotion drawn from an enormous search space. The work of this paper is the opposite: beginning with the minimal disturbance principle, MDPP prioritizes the promotion of reused blocks with lower levels of protection while minimizing the disturbance caused to the rest of the blocks in the set. The GIPPR paper only gives results for single-threaded workloads, showing that the technique is competitive with state-of-the-art techniques. As we will see in Sections 6 and 7, the technique performs somewhat poorly for multi-core workloads compared to the other techniques, while MDPP does well on both single-threaded and multi-core workloads. We conjecture that the machine learning approach of GIPPR overfits the policy it discovers to single-threaded workloads.

3. TREE-BASED PSEUDOLRU

PseudoLRU policies are based on providing an effect similar to the least-recently-used (LRU) policy, but with a more efficient representation. LRU orders blocks in a recency stack from most-recently-used (MRU) to least-recently-used, placing new blocks

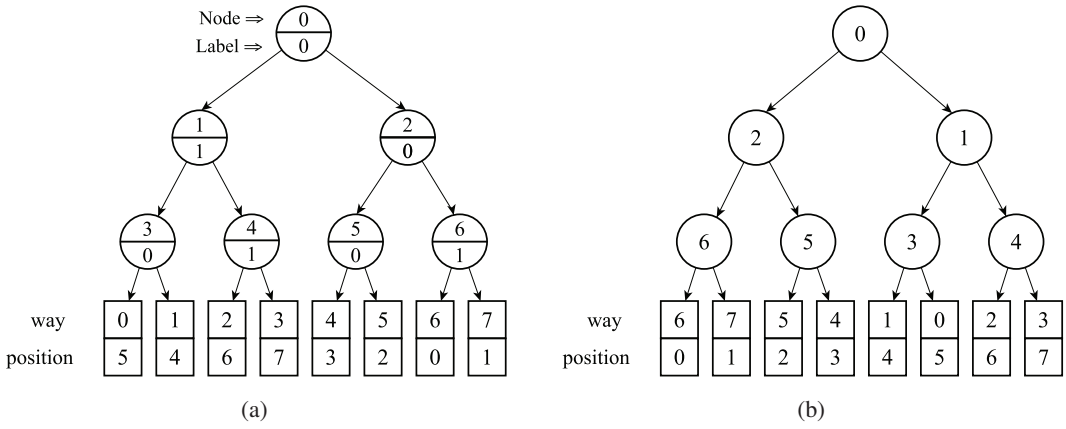

Figure 2: The nodes in an example PseudoLRU tree arranged (a) in ascending order of node number showing the labels guiding replacement policy, and (b) in order of positions.

into the MRU position and evicting blocks from the LRU position. As blocks are reused, they are moved to the MRU position with all blocks previously ahead of the reused block shifted down the recency stack. At any point in the algorithm, each block has a distinct position in the recency stack. For an n-way set-associative cache, the positions are numbered 0 (MRU) through $n-1$ (LRU).

3.1 A Policy Based on Binary Trees

The tree-based PseudoLRU replacement policy uses a binary tree to prioritize cache blocks [4]. The leaves of the tree are the indices of blocks (*i.e.* ways) in a set. The internal nodes are labeled with bits: 0 or 1. The internal nodes are represented as an array of bits with an implicit tree structure. There are $n-1$ internal nodes in a binary tree with n leaves. Thus, tree-based PseudoLRU is space efficient since it requires only $n-1$ bits per set compared with $n \log_2 n$ bits per set for LRU.

PseudoLRU Replacement.

On replacement, the victim block is chosen by tracing a path from the root of the tree to a leaf block. At each internal node, if the label is one, then the next node on the path is the right child of the current node; otherwise, it is the left child.

PseudoLRU Placement and Promotion.

When a block is placed or re-referenced, it is promoted. The bits along the path from the root to the associated leaf are modified such that the replacement algorithm would choose to go in the opposite direction from the leaf. That is, for all the nodes along a path from root to leaf, if the leaf is in the left subtree of the current node, then the bit in that node is changed to 1, otherwise the bit is changed to 0. Thus, among all blocks in the set, the promoted block receives the most protection from being evicted.

Protection.

We can formalize the notion of protection in a PseudoLRU tree: a leaf is *protected* at tree level l if the l^{th} node on a path from root to leaf is labeled 0 when the leaf is in the right subtree of that node, or labeled 1 when the leaf is in the left subtree of that node; otherwise, the leaf is *unprotected*. A promoted block is protected at all levels, while the block chosen as the victim is unprotected at all levels. We also define the *protected subtree* of a node to be the right subtree if the node is labeled 0, or the left subtree if the node is labeled 1.

PseudoLRU Ordering.

The leaves of the PseudoLRU tree may be ordered from 0 to $n-1$ in much the same way as the positions in the LRU recency stack [10]. The most protected leaf occupies position 0, *i.e.* the "PseudoMRU" block. The most unprotected leaf occupies position $n-1$, *i.e.* it is the PseudoLRU block. The other blocks occupy positions in between. For example, the sibling of the PseudoMRU block occupies position 1, and the first cousins of the PseudoMRU block are in positions 2 and 3. Let us think of the position of a leaf as a binary integer whose least significant bit is associated with the leaf. At each level of the tree, the relevant bit in the position is 1 if the leaf is unprotected at that level, or 0 if it is protected at that level.

One way to see the ordering is by considering a simple transformation on the binary tree: for each internal node labeled with 0, swap the left and right subtrees and change the label to 1. This procedure produces a canonical form in which the leaf nodes are ordered from left to right in ascending recency position: the rightmost leaf is the PseudoLRU way and the leftmost leaf is the PseudoMRU way. For example, Figure 2 shows an example 8-way associative PseudoLRU tree representing internal nodes numbers 0 through 6 labeled with randomly chosen binary values. Figure 2(b) shows the tree after the transformation. Nodes 0, 2, 3, and 5 have had their left and right subtrees swapped, placing the leaves into ascending order of position.

Promoting a block has the effect of moving other blocks into less fortunate positions, sometimes dramatically so. For instance, promoting a block such that the bit in the root node is toggled may move the previous PseudoLRU block down by $n/2$ positions. Only one block at a time is in the PseudoLRU position, but the position of each block is roughly proportional to its vulnerability to being moved to the PseudoLRU position in the near future.

3.2 PseudoLRU Disturbs Non-Promoted Blocks

Let us consider the placement/promotion algorithm for tree-based PseudoLRU. At each level in the tree, the algorithm alters a bit to protect the referenced block without regard for the disturbance caused to other blocks. We consider the disturbance to a block to

Old Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
New Position	0	1	2	3	4	5	6	7	0	1	2	3	0	1	0	0

Table 1: Minimal disturbance promotion for 16-way associativity.

be the number of positions it is moved as a result of an placement or promotion to another block.

PseudoLRU changes the positions of non-referenced blocks out of proportion to the analogous changes in standard LRU. Figure 1 shows the average magnitude of the change in position (the y-axis) for non-referenced blocks when a block at a given position (the x-axis) is placed or promoted for LRU and PseudoLRU. The average is for a 16-way set-associative 16MB cache taken over the 100 8-core workloads described in Section 5. For LRU, the maximum change in the position of a non-referenced block is 1 when the formerly MRU block is not referenced, and goes down to 0.6 for the LRU block. For tree-based PseudoLRU, the non-referenced blocks' positions are changed much more. The formerly PseudoMRU block travels on average two positions away, the block in position 12 moves 1.39 positions away, and at most block positions the distance traveled is significantly higher than the corresponding position in true LRU.

Thus, the blocks' PseudoLRU recency positions are inaccurate reflections of their true recency positions in true LRU. That is, blocks that might have locality may needlessly be moved too close to LRU, exposing them to the risk of eviction. Thus, PseudoLRU somewhat fails at imitating LRU. This is not really surprising as we expect to give up some performance when gaining a more space-efficient policy. However, what is surprising is that *this level of disturbance is totally unnecessary*, and removing it results in a replacement policy far superior to LRU.

4. A MINIMAL DISTURBANCE POLICY

We propose a different placement and promotion policy for tree-based PseudoLRU: place and promote blocks to positions that cause the minimal amount of disturbance among all blocks while giving the promoted block the protection it needs. The positions of blocks in the tree can be directly manipulated with simple algorithms [10], but we must still choose what the new positions should be to minimize disturbance.

4.1 Minimal Disturbance Placement

When an incoming block is placed, we must balance the potential disturbance caused to existing blocks with the potential risk of eviction to the placed block if it is left completely unprotected from eviction. What is the choice of placement position to achieve minimal disturbance? We propose that the best choice of placement is the one that affords the most protection to the incoming block while still allowing for a majority of the existing blocks to remain in the same position, while disturbing a minority of blocks.

Thus, for an n-way set-associative cache, a minimal disturbance policy should place incoming blocks in position $3n/4$. Since the previous position of a newly evicted block was $n-1$, placing the block into position $3n/4$ insures that the labels of the root node and the second-level nodes remain the same so there are no wide swings in the positions of the $3/4$ of blocks closest to PseudoMRU. Changing bits any closer to the root risks affecting the positions of a majority of other blocks. The newly placed block remains unprotected at the root level and second level, but fully protected beyond that so it has a good opportunity to be referenced and thus promoted.

4.2 Minimal Disturbance Promotion

Table 1 gives the old and new positions for promotions for 16-way associativity. Let us recursively subdivide the PseudoLRU tree into $1+\log_2 n$ regions of greater and lesser protection, starting with the protected subtree of the root, then the protected subtree of the unprotected subtree of the root, and so on, ending with singleton subtrees at positions $n-2$ and $n-1$. Figure 3 illustrates these regions of protection, with more vulnerable regions shaded darker. When a block *B* is promoted, a minimal number of bits in the tree are changed such that the first block in the smallest unprotected region that contains *B* is moved to the MRU position. As a result, *B* is moved along with the first block to a position relatively close to MRU. This strategy gives needed protection to the promoted block while minimizing movements among the other blocks. Thus, for $n = 16$, *i.e.* 16-way associativity, the blocks in positions 0 through 7 remain in the same positions because they are already in the protected subtree. Position 8 moves to position 0, position 9 moves to position 1, and so forth as in Table 1.

4.3 Three Examples

The key insight into why this policy works is that it provides the protection a block needs given its position, balanced with the desire to minimally disturb the other blocks. Let us consider three examples. Refer to Figure 3.

1. Suppose the block in position 15 is hit. This block is in the most vulnerable region of the tree. According to Table 1, this block will move all the way to position 0, causing significant disturbance. Why is that the right move? That block somehow moved to the PseudoLRU position, but just experienced a hit which is a strong hint that it has locality with a long reuse distance. We should move that block as far away from the PseudoLRU position as possible in case it experiences that long-term reuse again.

2. On the other hand, suppose the block in position 3 is hit.

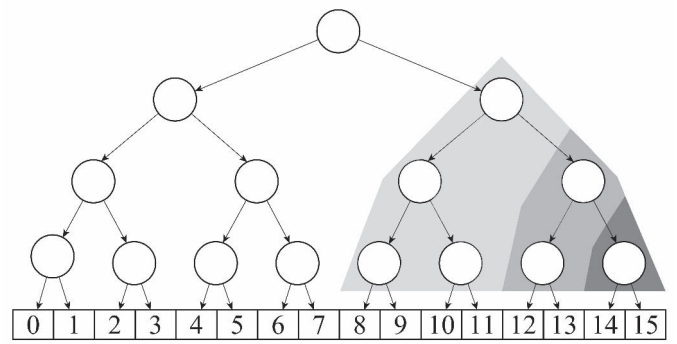

Figure 3: Regions of Protection in the PseudoLRU Tree. Leaf nodes are cache blocks in order of their PseudoLRU positions. Darker shaded regions are more vulnerable to eviction so the blocks in those regions are afforded more protection when they are hit.

According to Table 1, it should remain in position 3. It is in little danger of being evicted soon, and its more short-term locality hints that it will be reused soon.

3. Now suppose the block in position 7 is hit. It should remain in position 7. It is possible that a subsequent hit to the other half of the PseudoLRU tree will move that block to the PseudoLRU position, but 1) those blocks are statistically less likely to be hit, and 2) those blocks are at higher risk of eviction, so the block in position 7 accepts its temporary disturbance to pay for the protection of a more vulnerable block that has demonstrated locality. If the disturbed block has high locality, it will be hit again before becoming a victim.

Note that implementing our modified promotion and placement policies incurs virtually no extra energy or area penalty; the replacement status vector is updated using the same sort of logic as tree-based PseudoLRU, but with a different set of state transitions. Indeed, dynamic energy is reduced because for a great many accesses no update needs to be made to the status vector.

4.4 Effect of Minimal Disturbance on Block Positions

Figure 4 shows the effect of minimal disturbance placement and promotion on the average change of non-referenced blocks. The magnitude of changes in position is far less than the average for PseudoLRU or even true LRU. The block in position 0 moves the most, an average of 0.79 positions. Movements of subsequent blocks steadily decrease until block 8, which moves an average of 0.40 positions. The block in the PseudoLRU position moves an average of 0.27 positions. We can quantify the "total disturbance" of a placement and promotion policy as the area under the curve in Figure 4. Integrating the curves for the three replacement policies, we see that the total disturbance caused by LRU is 12.1, by PseudoLRU is 13.8, and by minimal disturbance PseudoLRU is 5.2.

Quantifying disturbance for RRIP is problematic for two reasons: 1) block positions are not distinct, as in LRU and PLRU; and 2) promotion of a block in RRIP is decoupled from changes in other block positions; blocks are only moved away from MRU on evictions so RRIP is not directly comparable with policies where promotions necessarily disturb some non-promoted blocks.

4.5 Disturbance to the Evicted Block

Figure 4 assumes that the disturbance of moving from one position to the next is constant. However, evicting a block, *i.e.* moving it from LRU to a position out of the cache, incurs far more disturbance to a block than simply moving it to another position. Thus, for example, an absurd policy of inserting only into the LRU position and never promoting any blocks would have a very high miss rate but relatively cause minimal disturbance. A more nuanced measurement of disturbance would assign a larger distance to evicting a block than to simply changing its position. This distance would take into account latency to satisfy a cache miss as well as the probability that the block will be referenced again. There is no general way to know that probability *a priori* but, as we will see in Section 4.6 we can estimate it dynamically with dead block prediction.

4.6 Dynamic MDPP

The MDPP policy as presented so far is a *static* policy. That is, the policy remains the same regardless of the run-time features of the workload. However, a great deal of recent work has focused on *dynamic* policies that change in response to workload characteristics. Thus, we extend the minimal disturbance idea to a dynamic

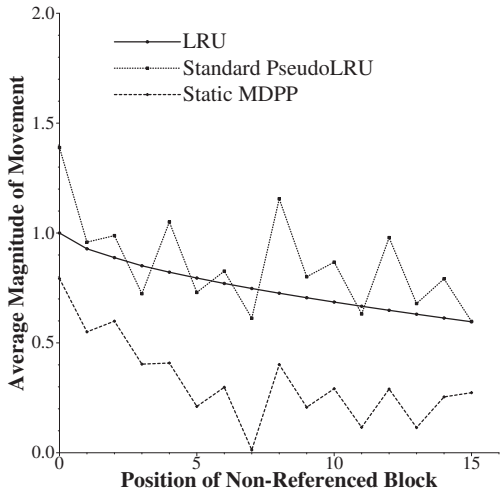

Figure 4: Average change in recency stack position for LRU, PseudoLRU, and Static MDPP

policy. In particular, we allow the placement position to be adapted through dead block prediction.

Some recent dynamic policies adapt the placement position using techniques such as prediction [26] and set-dueling [9,21]. Other policies implement selective *bypass*, *i.e.* when an incoming block is predicted to not be used before it is evicted, it is not placed in the cache but rather bypassed to the core [3, 13]. Both of these ideas are compatible with the minimal disturbance idea of causing as little disturbance as possible. Clearly, selective bypass works along provoking minimal disturbance: a block that will not be re-referenced will only pollute the cache and confuse the replacement policy. Such a block cannot be disturbed by bypass because it is already a dead block. Adapting the placement position can also achieve the minimal disturbance for PseudoLRU caches as long as the placement position is beyond the halfway point in the PseudoLRU recency stack so as not to make wide swings in block positions.

We use the sampling-based dead block prediction algorithm (SDBP) [13] to drive block placement and bypass. A dead block predictor predicts whether a block will be referenced again before it is evicted. The SDBP paper uses the program counter (PC) of a memory instruction to index a skewed predictor that provides a number between 0 and 9. If that number exceeds a threshold, the referenced block is predicted to be dead. If the block is reused, the corresponding counters in the prediction tables are decremented. If the block is evicted before being reused, the counters are incremented. The predictor learns from a *sampler*, a small set of partial tags held separately from the cache and managed with the LRU policy. In the original paper, each cache block has a prediction bit associated with it to drive the replacement policy. Our technique does not need this extra bit since the PseudoLRU block is always evicted. The technique only consults the predictor when a block is placed.

We modify the MDPP policy to incorporate dynamism for a 16-way set-associative cache as follows: on a cache miss, the PC of the offending memory instruction is used to consult the dead block predictor. The predictor returns a confidence value c between 0 and 9. If $c = 0$, the block is placed in position 8, which is the closest placement to PseudoMRU that does not disturb the 8 blocks

in the protected subtree of the root. If $0 < c < 6$ then the block is placed in position 14. This choice of placement position swaps the previous PseudoLRU block with the next-to-PseudoLRU block, having no effect on the positions of any other block in the set. If $c \geq 6$ then the block is bypassed, and the replacement state for the set remains unchanged. The cutoff for bypass of 6 or greater was determined empirically.

5. METHODOLOGY

5.1 Performance Models

We model performance with two simulators: one for single-threaded workloads and one for 8-core multi-programmed workloads. Both simulators use the following memory hierarchy parameters: L1 data cache: 32KB 8-way associative, L2 unified cache: 256KB 8-way, DRAM latency: 200 cycles. The single-thread simulator uses a 4MB L3 cache while the multi-core simulator uses a 16MB L3 cache. The single-thread simulator is a modified version of CMP$im [7]. The version we used was provided with the JILP Cache Replacement Championship [2]. It models an out-of-order 4-wide 8-stage pipeline with a 128-entry instruction window. This infrastructure enables collecting instructions-per-cycle (IPC) figures as well as misses per kilo-instruction and dead block predictor accuracy. For multi-core workloads, we found that CMP$im was quite slow and was unable to complete correctly beyond six cores. Thus, we adapted an in-house simulator to interoperate with CMP$im's cache replacement code to model 8-core workloads.

5.2 Workloads

We use the 29 SPEC CPU2006 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.

Single-Threaded Workloads.

For single-threaded workloads, the results reported per benchmark are the weighted average of the results for the individual simpoints. We use SimPoint [20] to identify up to 6 segments (i.e. *simpoints*) of one billion instructions each characteristic of the different program phases for each workload. 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. For each run, the simpoint is used to warm microarchitectural structures for 500 million instructions, then measures and reports results for the subsequent one billion instructions.

Multi-Core Workloads.

For 8-core multi-programmed workloads, we generated 100 workloads consisting of mixes from the SPEC CPU2006 simpoints described above. We follow the sample-balanced methodology of FIESTA [5]. We begin by selecting regions of equal standalone running time for each simpoint. Each region begins at the start of the simpoint and ends when the number of cycles in a standalone simulation reaches one billion cycles. Each workload is a mix of 8 of these regions chosen uniformly randomly without replacement. For each workload the simulator warms microarchitectural structures until 500 million total instructions have been executed, then measures results until each benchmark has executed for at least one billion additional cycles. When a thread reaches the end of its one billion cycle region, it starts over at the beginning. Thus, all 8 cores are active during the entire measurement period.

5.3 Replacement Policies

We compare our proposed technique against several related techniques: static and dynamic versions of re-reference interval prediction (SRRIP/DRRIP) [9], signature-based hit predictor for RRIP (SHiP) [26], and static and dynamic versions of genetic placement and promotion for tree-based PseudoLRU replacement (GIPPR/DGIPPR) [10]. For both simulators, we directly use code provided by the respective authors either through the World Wide Web or through personal communication (we developed GIPPR in our lab so the effort to duplicate that work was minimal). Our dynamic policy uses SDBP [13]. For that predictor, we used the code that is available from the cache replacement championship website [2].

The SHiP authors graciously provided code for their predictor in a form that was readily adaptable to the RRIP code. We modified this code to implement the sampling-based policy described in their paper and use the program counter (PC) as the signature for the predictor. Thus, what we hereafter refer to as SHiP in this paper is called SHiP-PC-S in that paper's nomenclature [26]. This modification allows us to control the overhead consumed by SHiP's structures. Thus, we may allocate the same amount of storage to both SHiP and SDBP for a fair comparison of the dynamic policies. To be clear, we implement sampling-based SHiP using the PC as the signature and deciding the placement position for a baseline RRIP policy with a maximum re-reference prediction value of three and using thread-aware policy decisions (TA-DRRIP) [8, 26].

The results we present in this paper compare our technique against SRRIP, DRRIP, SHiP, and GIPPR. In preparing this work, we also compared our technique against SDBP-based replacement and bypass [13] and protecting distance based policy [3]. We find that SHiP is superior in performance to these other policies, so we choose not to present those results and are content to compare with SHiP. We compare with SRRIP as an excellent static policy. We compare with DRRIP as a relatively simple dynamic policy with good performance. We compare with GIPPR/DGIPPR as the most closely related previous work.

5.4 Overhead for Predictors

Both SHiP and SDBP require extra state for their prediction structures. SHiP uses a 14-bit signature derived as a hash of the PC indexing a table of 16,384 three-bit saturating counters. It also keeps 14-bit signatures for each block in the sampled sets. SDBP keeps three tables of 8,192 2-bit saturating counters each indexed by a different 13-bit hash of the PC. It also keeps a *sampler*, an array of cache metadata kept separately from the main cache. Each block in the sampler includes a 15-bit partial tag, a valid bit, a prediction bit, a 4-bit LRU stack position and a 15-bit signature. The associativity of the sampler is 12. To keep the same amount of state for each predictor, we use 192 sampled sets for SHiP and 96 sampled sets for SDBP. Both predictors consume approximately 11KB.

5.5 Overhead for Techniques

Each replacement policy we study require some overhead. Table 2 summarizes the overhead of the various polices. Note that DRRIP and DGIPPR require a few extra bits for set dueling counters. We ignore this tiny extra storage overhead. The multi-core MDPP policy saves area over SHiP due to the fact that it requires only one bit of replacement state per block rather than two. This savings translates to a fraction of a percent of the bits allocated to the cache so we do not wish to overstate this advantage. Nevertheless, we note that the replacement state saved is typically implemented using larger 8T register file cells rather than 6T SRAM cells, and the savings in terms of cells per core of dynamic MDPP over SHiP is approximately 8KB, which is roughly equivalent to

Technique	Overhead for 4MB cache	Overhead for 16MB cache
LRU	4 bits $\times$ 2^{16} blocks = 32KB	4 bits $\times$ 2^{18} blocks = 128KB
PseudoLRU	15 bits $\times$ 2^{12} sets = 7.5KB	15 bits $\times$ 2^{14} sets = 30KB
Static MDPP	same as PseudoLRU: 7.5KB	same as PseudoLRU: 30KB
SRRIP/DRRIP	2 bits $\times$ 2^{16} blocks = 16KB	2 bits $\times$ 2^{18} blocks = 64KB
SHiP	16KB for RRIP + 11KB predictor = 27KB	64KB for RRIP + 11KB predictor = 75KB
Dynamic MDPP	7.5KB + 11KB predictor = 18.5KB	30KB + 11KB predictor = 41KB

Table 2: Overhead required by the various techniques

the size of a moderate dual-ported branch direction predictor or branch target buffer.

5.6 Measuring Performance

In Sections 6 and 7 we report performance relative to LRU for the various techniques tested. For single-threaded workloads, we report the speedup over LRU, i.e. the instructions-per-cycle (IPC) of a technique divided by the IPC given by LRU. For the multi-core workloads, we report the weighted speedup normalized to LRU. That is, for each thread i sharing the 16MB cache, we compute IPC_i. Then we find SingleIPC_i as the IPC of the same program running in isolation with a 16MB cache with LRU replacement. Then we compute the weighted IPC as $\sum \text{IPC}_i/\text{SingleIPC}_i$. We then normalize this weighted IPC with the weighted IPC using the LRU replacement policy.

6. RESULTS FOR STATIC POLICIES

This section gives results for static policies. These policies do not adapt to changing workload characteristics, but are relatively simple to implement and verify. For instance, the static MDPP policy could be plugged in easily to an existing PseudoLRU-replaced cache by simply modifying the algorithm or lookup table that implements placement and promotion.

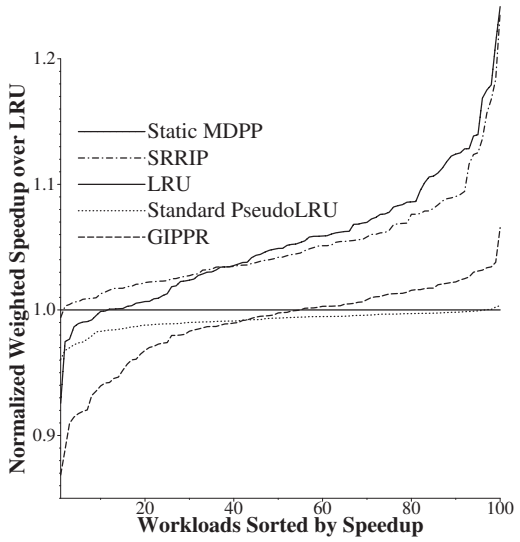

Figure 5: Normalized Weighted Speedup over LRU for Static Policies on 8-core Workloads

6.1 Multi-Core Workloads

6.1.1 Performance

Figure 5 shows weighted speedup normalized to LRU for SRRIP, static MDPP, and standard PseudoLRU with a 16MB last-level cache (see Section 5.6 for our definition of speedup). The figure shows the speedups for each workload in ascending sorted order to yield S-curves. Standard PseudoLRU performs somewhat worse than LRU, giving a geometric mean speedup of 0.99. SRRIP gives a 5.0% improvement over LRU. Static MDPP gives a 5.3% improvement over LRU. The best speedup of static MDPP over SRRIP is 18%, while the best speedup of SRRIP over static MDPP is 15%. Thus, static MDPP exceeds the performance of SRRIP on multi-core workloads.

GIPPR with a single vector performs poorly. Its average speedup over LRU is 0.98, *i.e.* it delivers a 2% slowdown. We have probed deeply into the GIPPR algorithm with the author of the original work to try to determine why it performs poorly on multi-core workloads. GIPPR was designed by a genetic algorithm using single-threaded workloads. We conclude that its parameters work well in that context, but are not suitable for multi-core workloads. Single-core workloads are characterized by a few distinct patterns of behavior that can be captured by a handful of GIPPR vectors, but multi-core workloads combine characteristics resulting in a wide range of behaviors that cannot be distilled down to a small number of vectors.

6.2 Single-Threaded Workloads

6.2.1 Performance

Figure 6 gives the speedup over LRU for several static techniques on single-threaded workloads with a 4MB last-level cache. The benchmarks on the y-axis are sorted based on their performance on the DRRIP policy. We use the same ordering in Section 7 for the bar chart there. Most of the benchmarks are not helped or significantly hurt by most of the policies. The aggregate speedup attributable to the polices are mostly due to the memory intensive benchmarks listed to the right of `401.bzip2`. However, some benchmarks such as `459.GemsFDTD` and `471.omnetpp` experience some slowdown from non-LRU policies.

Standard PseudoLRU gives no speedup or slowdown over LRU on average. SRRIP gives a geometric mean 2.6% speedup over LRU. GIPPR with a single vector optimized for the SPEC CPU2006 benchmarks gives a speedup of 3.5%. Static MDPP gives a speedup of 2.5%. Thus, static MDPP gives comparable performance to SRRIP. While GIPPR is slightly better than the static MDPP Cache on single-threaded workloads, static GIPPR performs poorly on multi-core workloads.

6.2.2 Misses

Figure 7 illustrates misses per 1000 instructions for the single-threaded workloads. The benchmarks on the y-axis are sorted in descending order of misses for DRRIP. GIPPR yields the fewest misses at an average 6.2 MPKI. The arithmetic mean MPKI for standard PseudoLRU is approximately the same as LRU. SRRIP

yields 6.6 MPKI on average while static MDPP gives 6.4 MPKI on average.

7. RESULTS FOR DYNAMIC POLICIES

This section gives results for dynamic policies. These policies adapt to different workload characteristics using set-dueling and/or prediction. Dynamic policies deliver the best performance in the literature. In particular, SHiP represents the current state-of-the-art cache management policy.

7.1 Multi-Core Workloads

Figure 8(a) shows the normalized weighted speedup for the various dynamic techniques. The speedups for the 100 workloads are shown as S-curves, sorted in ascending order. DRRIP yields a geometric mean 10.5% speedup. The 4DGIPPR technique gives a geometric mean 4.3% improvement over LRU. The state-of-the-art SHiP policy gives a 12.3% speedup. Dynamic MDPP achieves a 14.3% speedup. The best speedup of dynamic MDPP over SHiP is 24%, while the best speedup of SHiP over dynamic MDPP is 19%. The figure shows a clear separation between each technique, with dynamic MDPP outperforming all of the other techniques for almost all of the workloads. Thus, MDPP improves performance over previous techniques without increasing complexity.

7.2 Single-Threaded Workloads

7.2.1 Performance

Figure 9 illustrates the speedup over LRU of the various dynamic techniques. The benchmarks on the y-axis are sorted based on their speedup over LRU for the DRRIP policy. DRRIP achieves a speedup of 5.4% over LRU. Dynamic MDPP yields a 5.9% speedup. 4DGIPPR, the 4-vector dynamic version of GIPPR, gives a 5.6% speedup. SHiP gives a 5.9% speedup. Thus, dynamic MDPP matches the performance of the state-of-the-art policy SHiP and exceeds the performance of 4DGIPPR whose 4 vectors are specially designed to work well with these specific single-threaded workloads.

7.2.2 Misses

Figure 10 gives the misses per 1000 instructions for the dynamic policies. The benchmarks on the y-axis are sorted in descending order of their misses on DRRIP. DRRIP and 4DGIPPR both give an arithmetic mean misses of 91% of LRU. SHiP delivers a arithmetic mean normalized misses of 89% of LRU. Dynamic MDPP yields 88% of the misses of LRU.

7.3 Impact of Storage Overhead

In the main results, we model SHiP and dynamic MDPP with the same storage overhead for the predictors, including tables of counters and set sampling overhead. However, SHiP consumes 32 extra bits per set for storing the 16 2-bit re-reference interval prediction values, while MDPP requires only an additional 15 bits per set to store the PseudoLRU tree internal node values.

In another area study (not illustrated for space reasons), we performed three iso-area comparisons of SHiP with dynamic MDPP to study the impact of this additional storage overhead. In the first experiment, we modeled SHiP with a smaller 2.5KB predictor, thus reducing the total overhead of SHiP to roughly the same as the dynamic MDPP Cache for the single-core 4MB cache configuration. This change had negligible impact on SHiP's performance. In the second experiment, we modified SHiP to use one bit for RRIP states rather than two bits, so that SHiP and dynamic MDPP would have equivalent storage requirements for the larger 16MB cache. This change resulted in a reduction of SHiP's speedup from 12.3% to 9.3%. In the third experiment, we allowed the 16MB dynamic MDPP to expand the number of sampled sets such that SHiP and MDPP consume the same number of bits overall, including the per-set bits in the hardware budget. This allows MDPP to use 966 sampled sets instead of 96, resulting in an increase in the speedup from 14.3% to 15.0%. Thus, while the size of the SHiP predictor does not seem to have a large impact on performance, the size of the per-block state, which is the majority of the cache management metadata storage in the multi-core cache, has a significant impact on performance. Moreover, replacing SHiP with dynamic MDPP and keeping total area constant results in an 15.0% speedup.

8. ANALYSIS

Figure 11 quantifies the contributions of the individual placement and promotion policies to improving tree-based PseudoLRU. All combinations of standard PseudoLRU placement and promotion, minimal disturbance promotion, and static and dynamic minimal disturbance placement are considered. The effect of the technique with and without bypass is also illustrated, alongside SHiP with and without bypass. The normalized weighted speedups are aggregated over the 100 multi-core workloads. The bar chart is ordered in ascending geometric mean speedup over true LRU.

8.1 Effects of Minimal Disturbance

Minimal disturbance promotion with standard placement at posi-

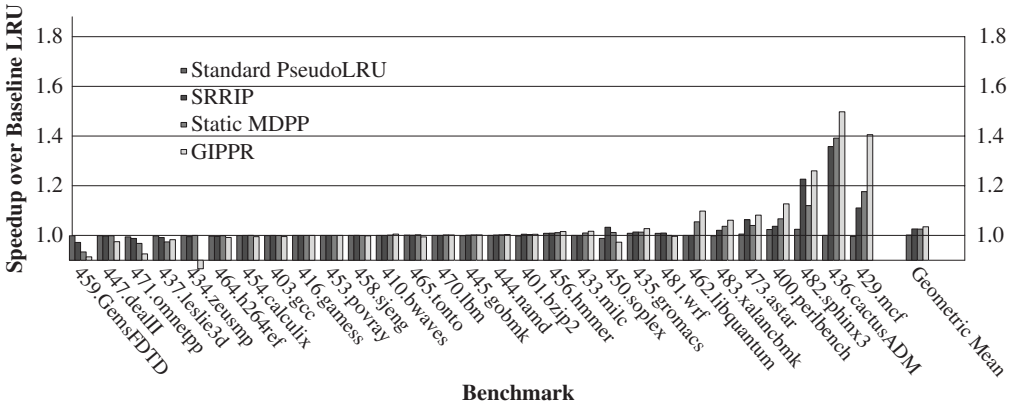

Figure 6: Speedup over LRU for Static Policies on Single-Threaded Workloads

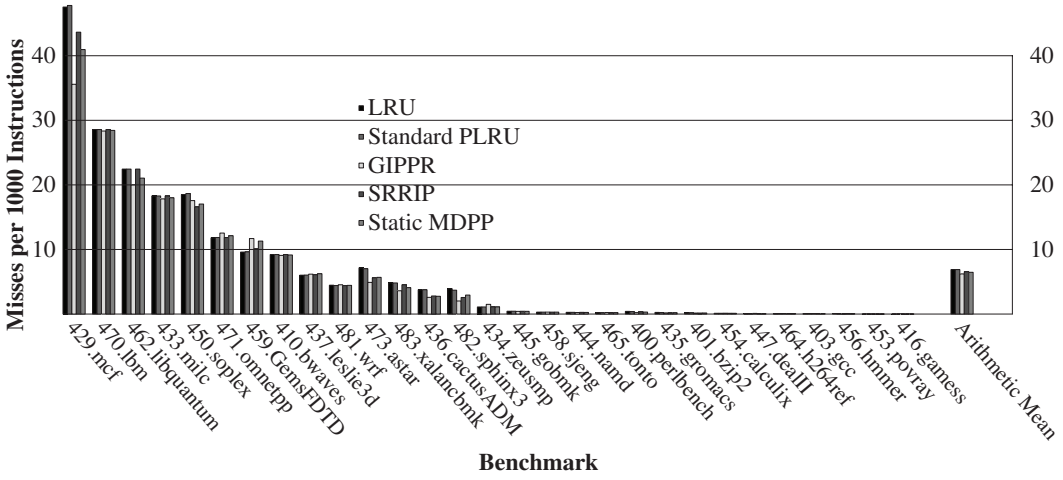

Figure 7: Misses per 1000 Instructions for Static Policies

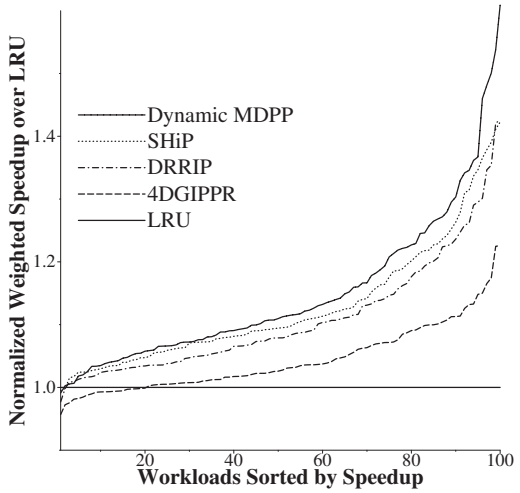

Figure 8: Normalized Weighted Speedup over LRU for Dynamic Policies on 8-core Workloads

tion 0 gives a geometric mean of 98.8% of the performance of LRU, very slightly worse than normal PseudoLRU at a 99.2%. Standard promotion with minimal disturbance placement yields a 4.3% speedup over LRU. Minimal disturbance promotion and static minimal disturbance placement together (*i.e.* static MDPP) give a 5.3% improvement over LRU. At this point, it is clear that the placement position is more important for performance than the promotion policy, but both are needed for the best static policy.

The contribution of minimal disturbance promotion is more apparent for the dynamic policies. Dynamic minimal disturbance placement with standard PseudoLRU promotion yields a geometric mean speedup of 12.3%, which is matches the performance of SHiP (not illustrated). With both minimal disturbance promotion and dynamic minimal disturbance placement (*i.e.* dynamic MDPP), the geometric mean speedup is boosted to 14.3%.

8.2 Effect of Bypass

Dynamic MDPP bypasses blocks that are predicted dead, in keeping with the minimal disturbance principle *i.e.* blocks that do not enter the cache cannot disturb the cache. SHiP was described as a placement policy for RRIP without considering bypass. It uses a threshold on the 3-bit confidence counter read from the prediction table to decide whether to place a block in the distant re-reference interval or the immediate re-reference interval. We modify SHiP to use two thresholds: one to indicate that the probability of a hit is so low that the block should be bypassed rather than placed, and a higher one to decide whether to place in the distant or immediate re-reference interval. Sampled sets are not bypassed so that the predictor can continue to learn from them. We exhaustively search the design space of all pairs of feasible thresholds and report the configuration with the best performance: a threshold of 0 for bypass and 1 for distant re-reference interval placement. Refer again to Figure 11. We find that SHiP with bypass performs about as well as SHiP without bypass: 12.2% speedup vs. 12.3% speedup over LRU. We also model MDPP without bypass, which yields a speedup of 11.1% over LRU, vs. 14.3% with bypass. Dead blocks entering an MDPP-managed cache can cause disturbance harmful to performance in the PseudoLRU tree in the rare case that a promotion to another block may unintentionally move a dead block out of the PseudoLRU position. Dead blocks entering a SHiP-managed cache do not affect the re-reference interval predictions of other blocks since they enter with a distant re-reference interval and are quickly evicted. Thus, bypass significantly helps MDPP while not helping SHiP.

9. FUTURE WORK

In this section, we motivate future work in minimal disturbance policies. In particular, we would like to apply minimal disturbance policies to die-stacked DRAM caches. In a set associative DRAM cache design, placing tags and replacement state in the on-chip SRAM causes a large capacity overhead. The minimal disturbance policy uses less than half of the per-set state of DRRIP, SRRIP, and SHiP, so for smaller DRAM caches it may be possible to keep the replacement state on-chip. Still, for larger caches, it may be necessary to keep it in DRAM.

Recent work proposes storing tags and replacement state in the DRAM rather than SRAM [18], or even doing away with associativity altogether [22]. Changing replacement state in the DRAM cache involves a costly read-modify-write cycle separate from data access. For a miss, this cycle may be unavoidable as tags need to

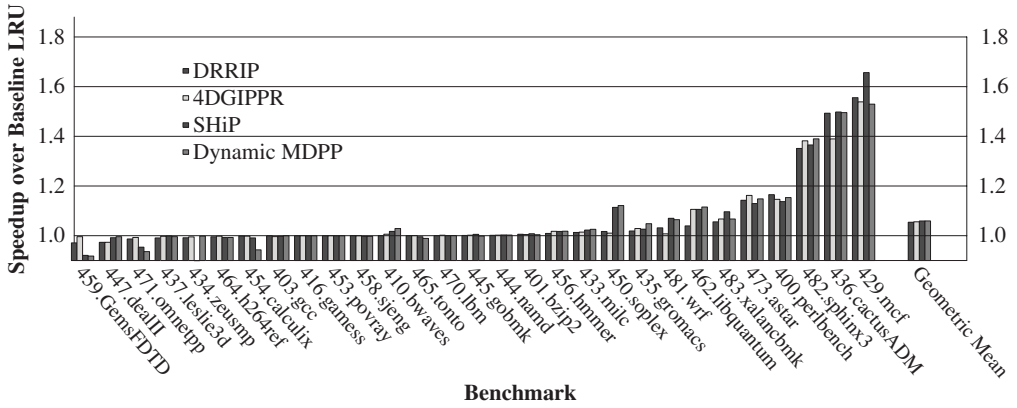

Figure 9: Speedup over LRU for Dynamic Policies on Single-Threaded Workloads

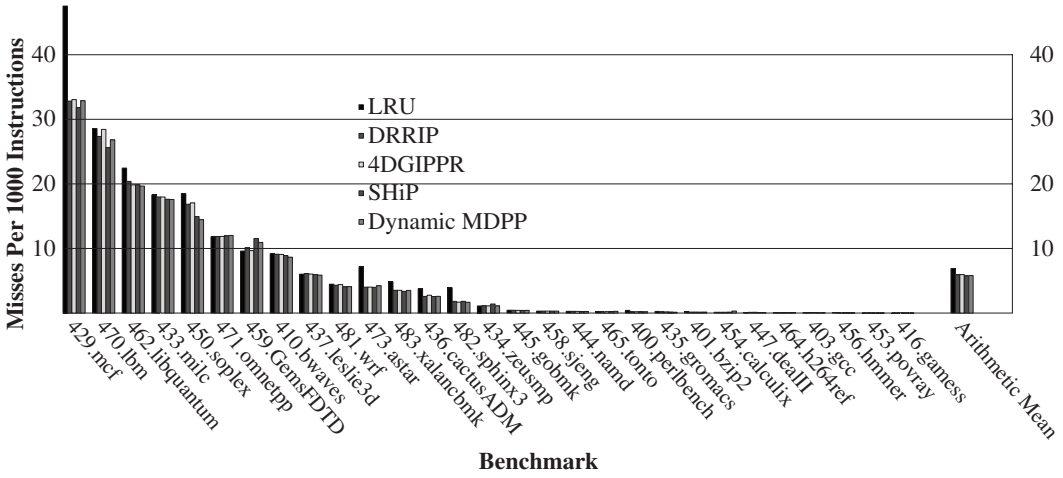

Figure 10: Misses per 1000 Instructions for Dynamic Policies

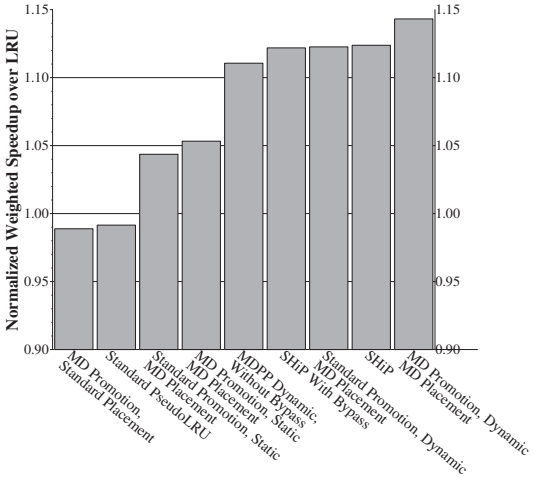

Figure 11: Contributions of aspects of MDPP to performance

be modified as well, but for a hit, if the promotion does not change replacement state, then there is no need to modify it in DRAM. The MDPP policy could avoid this costly operation for the promotions that leave the referenced block in the same position.

We measured the percentage of accesses that are hits whose promotions modify replacement state for the various policies average over the 100 multi-core workloads. LRU, PseudoLRU, GIPPR, DGIPPR, and SRRIP promotions all modify replacement state for more than 20% of all cache accesses. DRRIP promotions modify replacement state on 18% of accesses. SHiP promotions modify state on 17% of accesses. Promotions from dynamic MDPP modify state for 14% of access, and promotions from static MDPP modify state for only 9% of accesses. Thus, a DRAM cache management strategy that optimizes for the case when replacement state need not be changed in DRAM can potentially find significant benefit from using MDPP.

10. CONCLUSIONS

In this paper, we have considered the principle of minimal disturbance as applied to the changes in positions of cache blocks. We have shown that tree-based PseudoLRU is disturbing, and that mending it significantly improves the policy. We have shown that the static minimal disturbance policy as well as a dynamic version using dead block prediction provide performance competitive with state-of-the-art policies at a significantly lower cost in terms of storage overhead. In future work, we will investigate applying favor-

able features of these policies, namely, low overhead, avoidance of many read-write-modify promotions, and low miss rate, to associative DRAM caches.

11. ACKNOWLEDGEMENTS

Elvira Teran and Daniel A. Jiménez are supported by NSF grant CCF-1332598. We thank the anonymous reviewers for their helpful feedback.

12. REFERENCES

[1] J. Abella, A. González, X. Vera, and M. F. P. O'Boy le, "Iatac: a smart predictor to turn-off l2 cache lines," *ACM Trans. Archit. Code Optim.*, vol. 2, no. 1, pp. 55–77, 2005.

[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] 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.

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

[5] A. Hilton, N. Eswaran, and A. Roth, "FIESTA: A sample-balanced multi-program workload methodology," in *Workshop on Modeling, Benchmarking and Simulation (MoBS)*, June 2009.

[6] Z. Hu, S. Kaxiras, and M. Martonosi, "Timekeeping in the memory system: predicting and optimizing memory behavior," *SIGARCH Comput. Archit. News*, vol. 30, no. 2, pp. 209–220, 2002.

[7] 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.

[8] A. Jaleel, W. Hasenplaugh, M. K. Qureshi, J. Sebot, S. S. Jr., and J. Emer, "Adaptive insertion policies for managing shared caches," in *Proceedings of the 2008 International Conference on Parallel Architectures and Compiler Techniques (PACT)*, September 2008.

[9] 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.

[10] D. A. Jiménez, "Insertion and promotion for tree-based pseudolru last-level caches," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46, 2013, pp. 284–296.

[11] 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.

[12] 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.

[13] S. M. Khan, Y. Tian, and D. A. Jiménez, "Sampling dead block prediction for last-level caches," in *MICRO*, December 2010, pp. 175–186.

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

[15] A.-C. Lai and B. Falsafi, "Selective, accurate, and timely self-invalidation using last-touch prediction," in *International Symposium on Computer Architecture*, 2000, pp. 139 – 148.

[16] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block prediction & dead-block correlating prefetchers," *SIGARCH Comput. Archit. News*, vol. 29, no. 2, pp. 144–154, 2001.

[17] 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.

[18] G. H. Loh and M. D. Hill, "Efficiently enabling conventional block sizes for very large die-stacked dram caches," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 454–464.

[19] P. Michaud, A. Seznec, and R. Uhlig, "Trading conflict and capacity aliasing in conditional branch predictors," in *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997, pp. 292–303.

[20] 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.

[21] 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.

[22] M. K. Qureshi and G. H. Loh, "Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45, Washington, DC, USA, 2012, pp. 235–246.

[23] 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.

[24] S. Somogyi, T. F. Wenisch, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi, "Memory coherence activity prediction in commercial workloads," in *WMPI '04: Proceedings of the 3rd workshop on Memory performance issues*. New York, NY, USA: ACM, 2004, pp. 37–45.

[25] B. Widrow and M. Lehr, "30 years of adaptive neural networks: Perceptron, MADALINE, and backpropagation," *Proceedings of IEEE*, vol. 78, no. 9, pp. 1415–1442, September 1990.

[26] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, J. Simon C. Steely, and J. Emer, "SHiP: Signature-based hit predictor for high performance caching," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 430–441.

[27] 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.