A Case for MLP-Aware Cache Replacement

Moinuddin K. Qureshi *Daniel Lynch* *Onur Mutlu* *Yale N. Patt*

High Performance Systems Group
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78712-0240

TR-HPS-2006-003
February 27, 2006

This page is intentionally left blank

A Case for MLP-Aware Cache Replacement

Abstract

Performance loss due to long-latency memory accesses can be reduced by servicing multiple memory accesses concurrently. The notion of generating and servicing long-latency cache misses in parallel is called Memory Level Parallelism (MLP). MLP is not uniform across cache misses – some misses occur in isolation while some occur in parallel with other misses. Isolated misses are more costly on performance than parallel misses. However, traditional cache replacement is not aware of the MLP-dependent cost differential between different misses. Cache replacement, if made MLP-aware, can improve performance by reducing the number of performance-critical isolated misses.

This paper makes two key contributions. First, it proposes a framework for MLP-aware cache replacement by using a run-time technique to compute the MLP-based cost for each cache miss. It then describes a simple cache replacement mechanism that takes both MLP-based cost and recency into account. Second, it proposes a novel, low-hardware overhead mechanism called *Sampling Based Adaptive Replacement (SBAR)*, to dynamically choose between an MLP-aware and a traditional replacement policy, depending on which is more effective in reducing the number of memory related stalls. Evaluations with the SPEC CPU2000 benchmarks show that MLP-aware cache replacement can improve performance by as much as 23%.

1 Introduction

As the imbalance between processor and memory speeds increases, the focus on improving system performance moves to the memory system. Currently, processors are supported by large on-chip caches that try to provide faster access to recently-accessed data. Unfortunately, when there is a miss at the largest on-chip cache, instruction processing stalls after a few cycles [9], and the processing resources remain idle for hundreds of cycles [22]. The inability to process instructions in parallel with long-latency cache misses results in substantial performance loss. One way to reduce this performance loss is to process the cache misses in parallel.[1] Techniques such as non-blocking caches [11], out-of-order execution with large instruction windows, runahead execution [5][14], and prefetching improve performance by parallelizing long-latency memory operations. The notion of generating and servicing multiple outstanding cache misses in parallel is called *Memory Level Parallelism* (MLP) [6].

1.1 Not All Misses are Created Equal

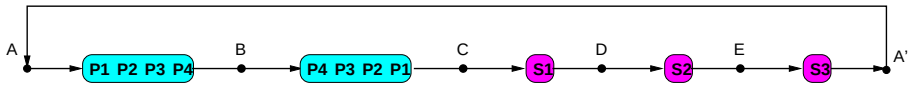
Servicing misses in parallel reduces the number of times the processor has to stall due to a given number of long-latency memory accesses. However, MLP is not uniform across all memory accesses in a program. Some misses occur in isolation (e.g., misses due to pointer-chasing loads), whereas some misses occur in parallel with other misses (e.g., misses due to array accesses). The performance loss resulting from a cache miss is reduced when multiple cache misses are serviced in parallel because the idle cycles waiting for memory get amortized over all the concurrent

[1] Unless stated otherwise, *cache* refers to the largest on-chip cache. *Cache miss* refers to a miss in the largest on-chip cache. Multiple concurrent misses to the same cache block are treated as a single miss. *Parallel misses* refers to a scenario where there is more than one miss outstanding. *Isolated miss* refers to a miss which is not serviced concurrently with any other miss.

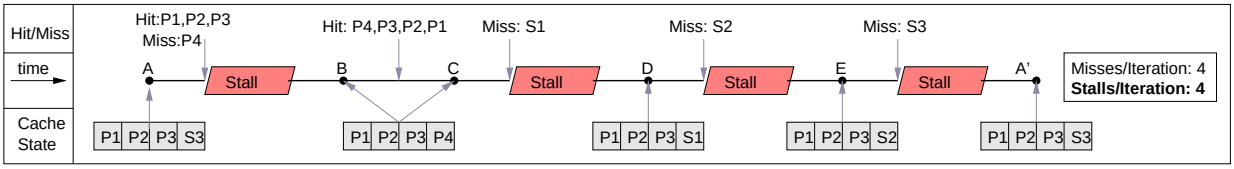
misses. Isolated misses hurt performance the most because the processor is stalled to service just a single miss. The non-uniformity in MLP and the resultant non-uniformity in the performance impact of cache misses open up an opportunity for cache replacement policies that can take advantage of the variation in MLP. Cache replacement, if made MLP-aware, can save isolated (relatively more costly) misses instead of parallel (relatively less costly) misses.

Unfortunately, traditional cache replacement algorithms are not aware of the disparity in performance loss due to misses that results from the variation in MLP. Traditional replacement schemes try to reduce the absolute number of misses with the implicit assumption that reduction in misses correlates with reduction in memory related stall cycles. However, due to the variation in MLP, the number of misses may or may not correlate directly with the memory related stall cycles. We demonstrate how ignoring MLP information in replacement decisions hurts performance with the following example. Figure 1(a) shows a loop containing 11 memory references. There are no other memory access instructions in the loop and the loop iterates many times.

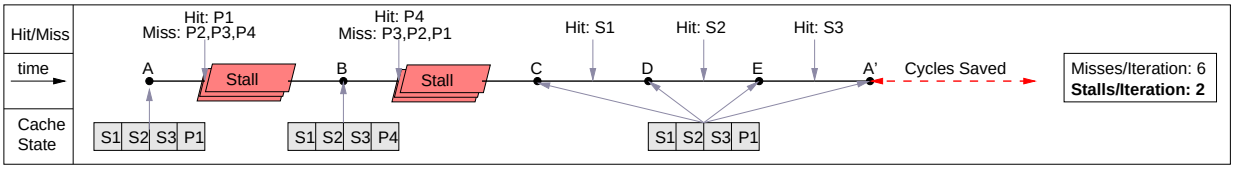
Let K (K $>$ 4) be the size of the instruction window of the processor on which the loop is executed. Points A, B, C, D, and E each represent an interval of at least K instructions. Between point A and point B, accesses to blocks P1, P2, P3, and P4 occur in the instruction window at the same time. If these accesses result in multiple misses then those misses are serviced in parallel, stalling the processor only once for the multiple parallel misses. Similarly, accesses between point B and point C will lead to parallel misses if there is more than one miss, stalling the processor only once for all the multiple parallel misses. Conversely, accesses to block S1, S2, or S3 result in isolated misses and the processor will be stalled once for each such miss. We analyze the behavior of this access stream for a fully-associative cache that has space for four cache blocks, assuming the processor has already executed the first iteration of the loop.



(a) An access pattern. A miss to an S block (S1,S2,S3) results in an Isolated miss. A miss to a P block (P1,P2,P3,P4) can be serviced in parallel with misses to other P blocks.



(b) Execution timeline for one iteration with Belady's OPT replacement



(c) Execution timeline for one iteration with MLP−aware replacement

Figure 1: The drawback of not including MLP information in replacement decisions.

First, consider a replacement scheme which tries to minimize the absolute number of misses, without taking MLP information into account. Belady's OPT [2] provides a theoretical minimum for the number of misses by evicting a block that is accessed furthest in the future. Figure 1(b) shows the behavior of Belady's OPT for the given access stream. At point B, blocks P1, P2, P3, and P4 were accessed in the immediate past and will be accessed again in the immediate future. Therefore, the cache contains blocks P1, P2, P3, and P4 at point B. This results in hits for the next accesses to blocks P4, P3, P2, and P1, and misses for the next accesses to blocks S1, S2, and S3. To guarantee the minimum number of misses, Belady's OPT evicts P4 to store S1, S1 to store S2, and S2 to store S3. Since the misses to S1, S2, and S3 are isolated misses, the processor incurs three long-latency stalls between points C and A. At point A, the cache contains P1, P2, P3, and S3 which results in a miss for P4, stalling the processor one more time. Thus, for each loop iteration, Belady's OPT causes four misses (S1, S2, S3, and P4) and four long-latency stalls.

Second, consider a simple MLP-aware policy, which tries to reduce the number of isolated misses. This policy keeps in cache the blocks that lead to isolated misses (S1, S2, S3) rather than the blocks that lead to parallel misses (P1, P2, P3, P4). Such a policy evicts the least-recent P-block from the cache. However, if there is no P-block in the cache, then it evicts the least recent S-block. Figure 1(c) shows the behavior of such an MLP-aware policy for the given access stream. The cache has space for four blocks and the loop contains only 3 S-blocks (S1, S2, and S3). Therefore, the MLP-aware policy never evicts any S-blocks at any point in the loop. After the first loop iteration, each access to S1, S2, and S3 results in a hit. At point A, the cache contains S1, S2, S3, and P1. From point A to B, the access to P1 hits in the cache, and the accesses to P2, P3, and P4 miss in the cache. However, these misses are serviced in parallel, therefore the processor incurs only one long-latency stall for these three misses. The cache evicts P1 to store P2, P2 to store P3, and P3 to store P4. So, at point B, the cache contains S1, S2, S3, and P4. Between point B and point C, the access to block P4 hits in the cache, while accesses to blocks P3, P2, and P1 miss in the cache. These three misses are again serviced in parallel, which results in one long-latency stall. Thus, for each iteration of the loop, the MLP-aware policy causes six misses ([P2, P3, P4] and [P3, P2, P1]) and only two long-latency stalls.

Note that Belady's OPT uses oracle information, whereas the MLP-aware scheme uses only information that is available to the microarchitecture. Whether a miss is serviced in parallel with other misses can easily be detected in the memory system, and the MLP-aware replacement scheme uses this information to make replacement decisions. For the given example, even with the benefit of an oracle, Belady's OPT incurs twice as many long-latency stalls compared to a simple MLP-aware policy.[2] This simple example demonstrates that it is important to incorporate MLP information into replacement decisions.

[2]We use Belady's OPT in the example only to emphasize that the concept of reducing misses and making the replacement scheme MLP-aware are orthogonal. However, Belady's OPT is impossible to implement because it requires knowledge of the future. Therefore, we will use LRU as the baseline policy for the remainder of this paper. For the LRU policy, each iteration of the loop shown in Figure 1 causes six misses ([P2, P3, P4], S1, S2, S3) and four long-latency stalls.

1.2 Contributions

Based on the observation that the aim of a cache replacement policy is to reduce memory related stalls, rather than to reduce the raw miss counts, we propose MLP-aware cache replacement and make the following contributions:

1. As a first step to enable MLP-aware cache replacement, we propose a run-time algorithm that can compute MLP-based cost for in-flight misses.

2. We show that, for most benchmarks, the MLP-based cost repeats for consecutive misses to individual cache blocks. Thus, the last-time MLP-based cost can be used as a predictor for the next-time MLP-based cost.

3. We propose a simple replacement policy called the Linear (LIN) policy which takes both recency and MLP-based cost into account to implement a practical MLP-aware cache replacement scheme. Evaluation with the SPEC CPU2000 benchmarks shows performance improvement of up to 23% with the LIN policy.

4. The LIN policy does not perform well for benchmarks in which the MLP-based cost differs significantly for consecutive misses to an individual cache block. We propose a mechanism called *Contest Based Selection (CBS)*, which selects the best replacement policy, on a per-set basis, depending on which policy reduces the number of memory related stall cycles.

5. It is expensive to implement the selection mechanism on a per-set basis for all the sets in the cache. Based on the key insight that few sampled sets can be used to decide the replacement policy globally for the cache, we propose a novel, low-hardware-overhead adaptation mechanism called *Sampling Based Adaptive Replacement (SBAR)*. SBAR allows dynamic selection between LIN and LRU while incurring a hardware overhead of 1854B (less than 0.2% area of the baseline 1MB cache).

2 Background

Out-of-order execution engines inherently improve MLP by continuing to execute instructions after a long-latency miss. Instruction processing stops only when the instruction window becomes full. If additional misses are encountered before the window becomes full, then these misses are serviced in parallel with the stalling miss. The analytical model of out-of-order superscalar processors proposed by Kharkhanis and Smith [10] provides fundamental insight into how parallelism in L2 misses can reduce the cycles per instruction incurred due to L2 misses.

The effectiveness of an out-of-order engine's ability to increase MLP is limited by the instruction window size. Several proposals [14][1][21][4][24] have looked at the problem of scaling the instruction window for out-of-order processors. Chou et al. [3] analyzed the effectiveness of different microarchitectural techniques such as out-of-order execution, value prediction [25], and runahead execution on increasing MLP. They concluded that microarchitecture optimizations can have a profound impact on increasing MLP. They also formally defined instantaneous MLP as *the*

average number of useful long-latency off-chip accesses outstanding when there is at least one such access outstanding. MLP can also be improved at the compiler level. Read miss clustering [16] is a compiler technique in which the compiler reorders load instructions with predictable access patterns to improve memory parallelism.

All of the techniques described thus far try to improve MLP by overlapping long-latency memory operations. MLP is not uniform across all memory accesses in a program though. While some of the misses are parallelized, many misses still occur in isolation. It makes sense to make this variation in MLP visible to the cache replacement algorithm. Cache replacement, if made MLP-aware, can increase performance by reducing the number of isolated misses at the expense of parallel misses. To our knowledge no previous research has looked at including MLP information in replacement decisions. Srinivasan et al. [20][19] analyzed the criticality of load misses for out-of-order processors. But, criticality and MLP are two different properties. Criticality, as defined in [20], is determined by how long instruction processing continues after a load miss, whereas, MLP is determined by how many additional misses are encountered while servicing a miss.

Cost-sensitive replacement policies for on-chip caches were investigated by Jeong and Dubois [7][8]. They proposed variations of LRU that take *cost* (any numerical property associated with a cache block) into account. In general, any cost-sensitive replacement scheme, including the ones proposed in [8], can be used for implementing an MLP-aware replacement policy. However, to use any cost-sensitive replacement scheme, we first need to define the *cost* of each cache block based on the MLP with which it was serviced. As the first step to enable MLP-aware cache replacement, we introduce a run-time technique to compute MLP-based cost.

3 Computing MLP-Based Cost: A First Order Model

For current instruction window sizes, instruction processing stalls shortly after a long-latency miss occurs. The number of cycles for which a miss stalls the processor can be approximated by the number of cycles that the miss spends waiting to get serviced. For parallel misses, the stall cycles can be divided equally among all concurrent misses.

3.1 Algorithm

The information about the number of in-flight misses and the number of cycles a miss is waiting to get serviced can easily be tracked by the MSHR (Miss Status Holding Register). Each miss is allocated an MSHR entry before a request to service that miss is sent to memory [11]. To compute the MLP-based cost, we add a field *mlp_cost* to each MSHR entry. The algorithm for calculating the MLP-based cost of a cache miss is shown in Algorithm 1.

When a miss is allocated an MSHR entry, the mlp_cost field associated with that entry is initialized to 0. We count instruction accesses, load accesses, and store accesses that miss in the largest on-chip cache as demand misses. All misses are treated on correct path until they are confirmed to be on the wrong path. Misses on the wrong path are not counted as demand misses. Each cycle, the mlp_cost of all demand misses in the MSHR is incremented by the amount

Algorithm 1 Calculate MLP-Based cost for cache misses

init_mlp_cost(miss): /* gets called when miss enters MSHR */
  $miss.mlp\_cost = 0$

update_mlp_cost(): /* gets called every cycle */
  $N \Leftarrow$ Number of outstanding demand misses in MSHR
  for each demand miss in the MSHR
    $miss.mlp\_cost$ += $(1/N)$

1/(Number of outstanding demand-misses in MSHR).[3,4] When a miss is serviced, the mlp_cost field in the MSHR represents the MLP-based cost of that miss. Henceforth, we will use mlp-cost to denote MLP-based cost.

3.2 Distribution of mlp-cost

Figure 2 shows the distribution of mlp-cost for 14 SPEC benchmarks measured on an eight-wide issue, out-of-order processor with a 128-entry instruction window. An isolated miss takes 444 cycles (400 cycle memory latency + 44 cycle bus delay) to get serviced. The vertical axis represents the percentage of all misses and the horizontal axis corresponds to different values of mlp-cost. The graph is plotted with 60-cycle intervals, with the leftmost bar representing the percentage of misses that had a value of $0 \leq$ mlp-cost < 60 cycles. The rightmost bar represents the percentage of all misses that had an mlp-cost of more than 420 cycles. All isolated misses (and some of the parallel misses that are serialized because of DRAM bank conflicts) are accounted for in the right-most bar.

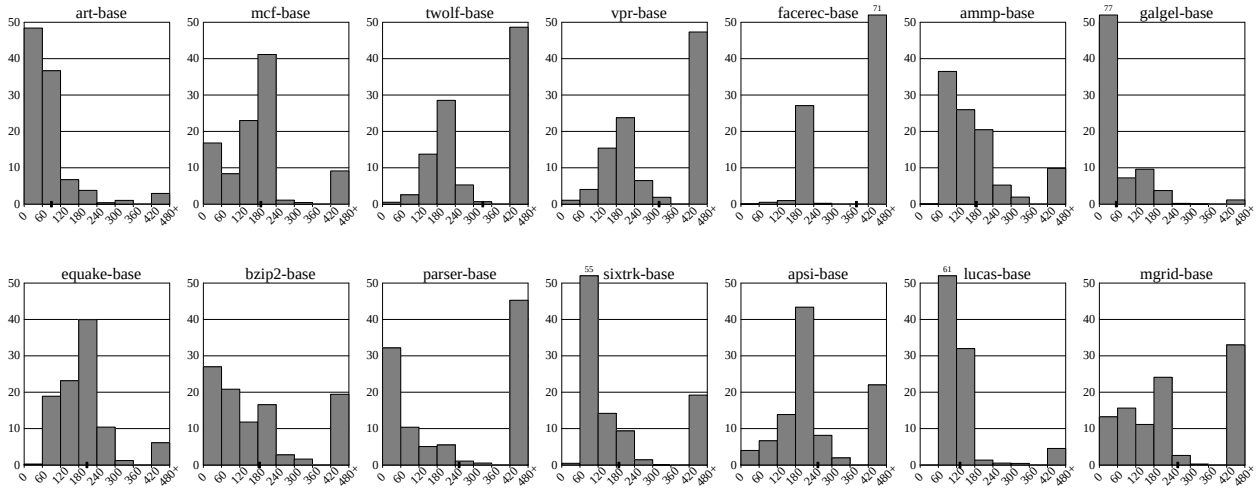


Figure 2: Distribution of mlp-cost. The horizontal axis represents the value of mlp-cost in cycles and the vertical axis represents the percentage of total misses. The dot on the horizontal axis represents the average value of mlp-cost.

[3]The number of adders required for the proposed algorithm is equal to the number of MSHR entries. However, for the baseline machine with 32 MSHR entries, time sharing four adders among the 32 entries has only a negligible effect on the absolute value of the MLP-based cost. For all our experiments, we assume that the MSHR contains only four adders for calculating the MLP-based cost. If more than four MSHR entries are valid, then the adders are time-shared between all the valid entries using a simple round-robin scheme.

[4]We also experimented by increasing the mlp_cost only during cycles when there is a full window stall. However, we did not find any significant difference in the relative value of mlp_cost or the performance improvement provided by our proposed replacement scheme. Therefore, for simplicity, we assume that the function update_mlp_cost is invoked every cycle.

For each benchmark, the average value of mlp-cost is much less than 444 cycles (number of cycles needed to serve an isolated miss). For art, more than 85% of the misses have an mlp-cost of less than 120 cycles indicating a high parallelism in misses. For mcf, about 40% of the misses have an mlp-cost between 180 and 240 cycles, which corresponds to two misses in parallel. Mcf also has about 9% of its misses as isolated misses. Facerec has two distinct peaks, one for the misses that occur in isolation and the other for the misses that occur with a parallelism of two. Twolf, vpr, facerec, and parser have a high percentage of isolated misses and hence the peak for the rightmost bar. The results for all of these benchmarks clearly indicate that there exists non-uniformity in mlp-cost which can be exploited by MLP-aware cache replacement. The objective of MLP-aware cache replacement is to reduce the number of isolated (i.e., relatively more costly) misses without substantially increasing the total number of misses. mlp-cost can serve as a useful metric in designing an MLP-aware replacement scheme. However, for the decision based on mlp-cost to be meaningful, we need a mechanism to predict the future mlp-cost of a miss given the current mlp-cost of a miss. For example, a miss that happens in isolation once can happen in parallel with other misses the next time, leading to significant variation in the mlp-cost for the miss. If mlp-cost is not predictable for a cache block, the information provided by the mlp-cost metric is not useful. The next section examines the predictability of mlp-cost.

3.3 Repeatability/Predictability of the mlp-cost metric

One way to predict the future mlp-cost value of a block is to use the current mlp-cost value of that block. The usefulness of this scheme can be evaluated by measuring the difference between the value of mlp-cost for successive misses to a cache block. We call the absolute difference in the value of mlp-cost for successive misses to a cache block as *delta*. For example, let cache block A have mlp-cost values of $\{444$ cycles, 110 cycles, 220 cycles, 220 cycles$\}$ for the four misses it had in the program. Then, the first delta for block A is 334 ($\|444 - 110\|$) cycles, the second delta for block A is 110 ($\|110 - 220\|$) cycles, and the third delta for block A is 0 ($\|220 - 220\|$) cycles. To measure delta, we do an off-line analysis of all the misses in the program. Table 1 shows the distribution of delta. A small delta value means that mlp-cost does not significantly change between successive misses to a given cache block.

Table 1: The first three rows represent the percentage of deltas that were between 0-59 cycles, 60-119 cycles, and more than 120 cycles respectively. The last row represents the average value of delta.

Benchmark	art	mcf	twolf	vpr	facerec	ammp	galgel	equake	bzip2	parser	apsi	sixtrack	lucas	mgrid
delta $<$ 60	86%	86%	52%	50%	96%	82%	71%	78%	43%	43%	85%	100%	84%	18%
60 $\leq$ delta $<$ 120	7%	7%	12%	14%	0%	10%	9%	12%	15%	5%	5%	0%	6%	16%
delta $\geq$ 120	7%	7%	36%	36%	4%	8%	20%	10%	42%	52%	10%	0%	10%	66%
Average delta	30	21	98	100	9	32	82	40	126	190	28	1	52	187

For all the benchmarks, except bzip2, parser, and mgrid, the majority of the delta values are less than 60 cycles. The average delta value is also fairly low, which means that the next-time mlp-cost for a cache block remains fairly close to the current mlp-cost. Thus, the current mlp-cost can be used as a predictor of the next mlp-cost of the same block in MLP-aware cache replacement. We describe our experimental methodology before discussing the design and implementation of an MLP-aware cache replacement scheme based on these observations.

4 Experimental Methodology

4.1 Configuration

We perform our experiments using an execution-driven simulator that models the Alpha ISA. Table 2 describes the baseline configuration. Our baseline processor is an aggressive eight-wide issue, out-of-order superscalar with a 128-entry instruction window. Because our studies deal with the memory system, we model bank conflicts, queueing delays, and port contention in detail. An isolated miss requires 444 cycles (400 cycle memory access + 44 cycle bus delay) to get serviced. Store instructions that miss the L2 cache do not block the window unless the store buffer is full.

Table 2 : Baseline processor configuration.

Instruction Cache	16KB, 64B line-size, 4-way with LRU replacement; 8-wide fetch with 2 cycle access latency.
Branch Predictor	64K-entry gshare/64K-entry PAs hybrid with 64K-entry selector; 4K-entry, 4-way BTB; minimum branch misprediction penalty is 15 cycles.
Decode/Issue	8-wide; reservation station contains 128 entries and implements oldest-ready scheduling
Execution Units	8 general purpose functional units; All INT instructions, except multiply, take 1 cycle; INT multiply takes 8 cycles. All FP operations, except FP divide, take 4 cycles; FP divide takes 16 cycles.
Data Cache	16KB, 64B line-size, 4-way with LRU replacement, 2-cycle hit latency, 32-entry MSHR.
Unified L2 Cache	1MB, 64B line-size, 16-way with LRU replacement, 15-cycle hit latency, 32-entry MSHR, 128-entry store buffer. Store misses do not block window unless the store buffer is full.
Memory	32 DRAM banks; 400-cycle access latency; bank conflicts modeled; maximum 32 outstanding requests;
Bus	16B-wide split-transaction bus at 4:1 frequency ratio. queueing delays modeled

Table 3 : Benchmark summary. (B = Billion).

Name	Type	FFWD	Num L2 Misses	Compulsory Misses
art	FP	18.25B	9680K	0.5%
mcf	INT	14.75B	23123K	2.2%
twolf	INT	30.75B	859K	2.9%
vpr	INT	60B	541K	4.3%
ammp	FP	4.75B	704K	5.1%
galgel	FP	14B	1333K	5.9%
equake	FP	26.25B	4604K	14.2%
bzip2	INT	2.25B	572K	15.5%
facerec	FP	8.75B	1190K	18.0%
parser	INT	66.25B	382K	20.3%
sixtrack	FP	8.5B	105K	20.6%
apsi	FP	3.25B	74K	22.8%
lucas	FP	2.5B	4041K	41.6%
mgrid	FP	3.5B	1932K	46.6%

4.2 Benchmarks

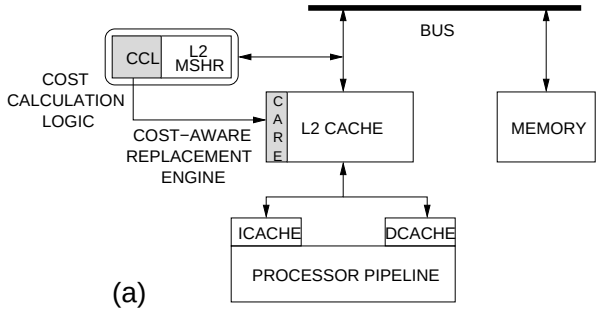
We use SPEC CPU2000 benchmarks compiled for the Alpha ISA with the `-fast` optimizations and profiling feedback enabled. For each benchmark, a representative slice of 250M instructions was obtained with a tool we developed using the Simpoint [18] methodology. For all benchmarks, except apsi, the `reference` input set is used. For apsi, the `train` input set is used.

Because cache replacement cannot reduce compulsory misses, benchmarks that have a high percentage of compulsory misses are unlikely to benefit from improvements in cache replacement algorithms. Therefore, we show results only for benchmarks where less than 50% of the misses are compulsory.[5] Table 3 shows the type, the fast-forward interval, the number of L2 misses, and the percentage of compulsory misses for each benchmark.

[5]For the remaining SPEC CPU2000 benchmarks, the majority of the misses are compulsory misses. Therefore, our proposed scheme does not significantly affect the performance of these benchmarks. With the proposed MLP-aware replacement scheme, the performance improvement ranges from +2.5% to -0.5% for those benchmarks.

5 The Design of an MLP-Aware Cache Replacement Scheme

Figure 3(a) shows the microarchitecture design for MLP-aware cache replacement. The added structures are shaded. The cost calculation logic (CCL) contains the hardware implementation of Algorithm 1. It computes mlp-cost for all demand misses. When a miss gets serviced, the mlp-cost of the miss is stored in the tag-store entry of the corresponding cache block. For replacement, the cache invokes the Cost Aware Replacement Engine (CARE) to find the replacement victim. CARE can consist of any generic cost-sensitive scheme [8][15]. We evaluate MLP-aware cache replacement using both an existing as well as a novel cost-sensitive replacement scheme.



(b)

Computed value of MLP−Based Cost	Quantized value of MLP−Based Cost
0 to 59 cycles	0
60 to 119 cycles	1
120 to 179 cycles	2
180 to 239 cycles	3
240 to 299 cycles	4
300 to 359 cycles	5
360 to 419 cycles	6
420+ cycles	7

Figure 3: (a) Microarchitecture for MLP-aware cache replacement. (b) Quantization of mlp-cost.

Before discussing the details of MLP-aware replacement scheme, it is useful to note that the exact value of mlp-cost is not necessary for replacement decisions. In a real implementation, to limit storage, the value of mlp-cost can be quantized to a few bits and this quantized value would be stored in the tag-store. We consider one such quantization scheme. It converts the value of mlp-cost into a 3-bit quantized value, according to the intervals shown in Figure 3(b). Henceforth, we use cost_q to denote the quantized value of mlp-cost.

5.1 The Linear (LIN) Policy

The baseline replacement policy is LRU. The replacement function of LRU selects the candidate cache block with least recency. Let $Victim_{LRU}$ be the victim selected by LRU and $R(i)$ be the recency value (highest value denotes the MRU and lowest value denotes LRU) of block i. Then, the victim of the LRU policy can be written as:

$$Victim_{LRU} = \arg\min_i \{R(i)\}. \quad (1)$$

We want a policy that takes into account both cost_q and recency. We propose a replacement policy that employs a linear function of recency and cost_q. We call this policy the Linear (LIN) policy. The replacement function of LIN can be summarized as follows: Let $Victim_{LIN}$ be the victim selected by the LIN policy, $R(i)$ be the recency value of block i, and $\text{cost}_q(i)$ be the quantized cost of block i, then the victim of the LIN policy can be written as:

$$Victim_{LIN} = \arg\min_i \{R(i) + \lambda * \text{cost}_q(i)\}. \quad (2)$$

The parameter λ determines the importance of cost_q in choosing the replacement victim. In case of a tie for the minimum value of $\{R + \lambda * \text{cost}_q\}$, the candidate with the smallest recency value is selected. Note that LRU is a

special case of the LIN policy with $\lambda = 0$. With high value of λ, the LIN policy tries to retain recent cache blocks that have high mlp-cost. For our experiments, we used the position in the LRU stack as the recency value (e.g. for a 16-way cache, $R(MRU) = 15$ and $R(LRU) = 0$). Since cost_q is quantized into three bits, its range is from 0 to 7. Unless stated otherwise, we use $\lambda = 4$ in all our experiments with the LIN policy.

5.2 Results for the LIN Policy

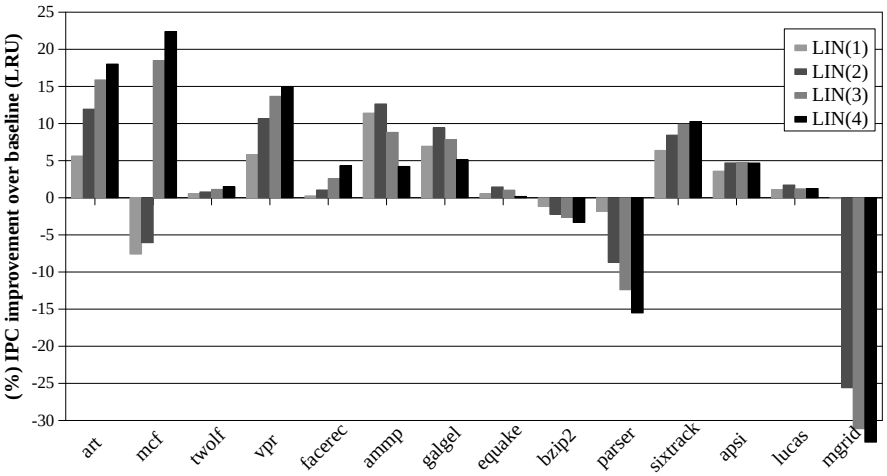


Figure 4: IPC variation with LIN (λ) as λ is varied from 1 to 4.

Figure 4 shows the performance impact of the LIN policy for different values of λ. The effect of the LIN policy is more pronounced as the value of λ is increased from 1 to 4. With λ=4, the LIN policy provides significant IPC improvement for art, mcf, vpr, ammp, galgel, and sixtrack. In contrast, it degrades performance for bzip2, parser, and mgrid. These benchmarks have high average delta values (refer to Table 1), so replacement decisions based on mlp-cost proves detrimental to performance. LIN can improve performance by reducing the number of isolated misses, or by reducing the total number of misses, or both. We analyze the LIN policy further by comparing the mlp-cost distribution of the LIN policy with the mlp-cost distribution of the baseline.

Figure 5 shows the mlp-cost distribution for both the baseline and the LIN policy. The inset contains information about the change in the number of misses and the change in IPC due to LIN. For mcf, almost all the isolated misses are eliminated by LIN. For twolf, although the total number of misses increases by 7%, IPC increases by 1.5%. A similar trend of increase in misses accompanied by increase in IPC is observed for ammp and equake. For these benchmarks, the IPC improvement is coming from reducing the number of misses with high mlp-cost even if this translates into a slightly-increased total number of misses. For all benchmarks, except art and galgel, the distribution of mlp-cost is skewed towards the left (i.e. lower mlp-cost) for the LIN policy when compared to the baseline. This indicates that LIN -successfully- has a bias towards reducing the proportion of high mlp-cost misses.

For art, galgel, and sixtrack, LIN reduces the total number of misses by more than 30%. This happens for applications that have very large data working-sets with low temporal locality, causing LRU to perform poorly [17][23]. The LIN policy automatically provides filtering for access streams with low temporal locality by at least keeping some of

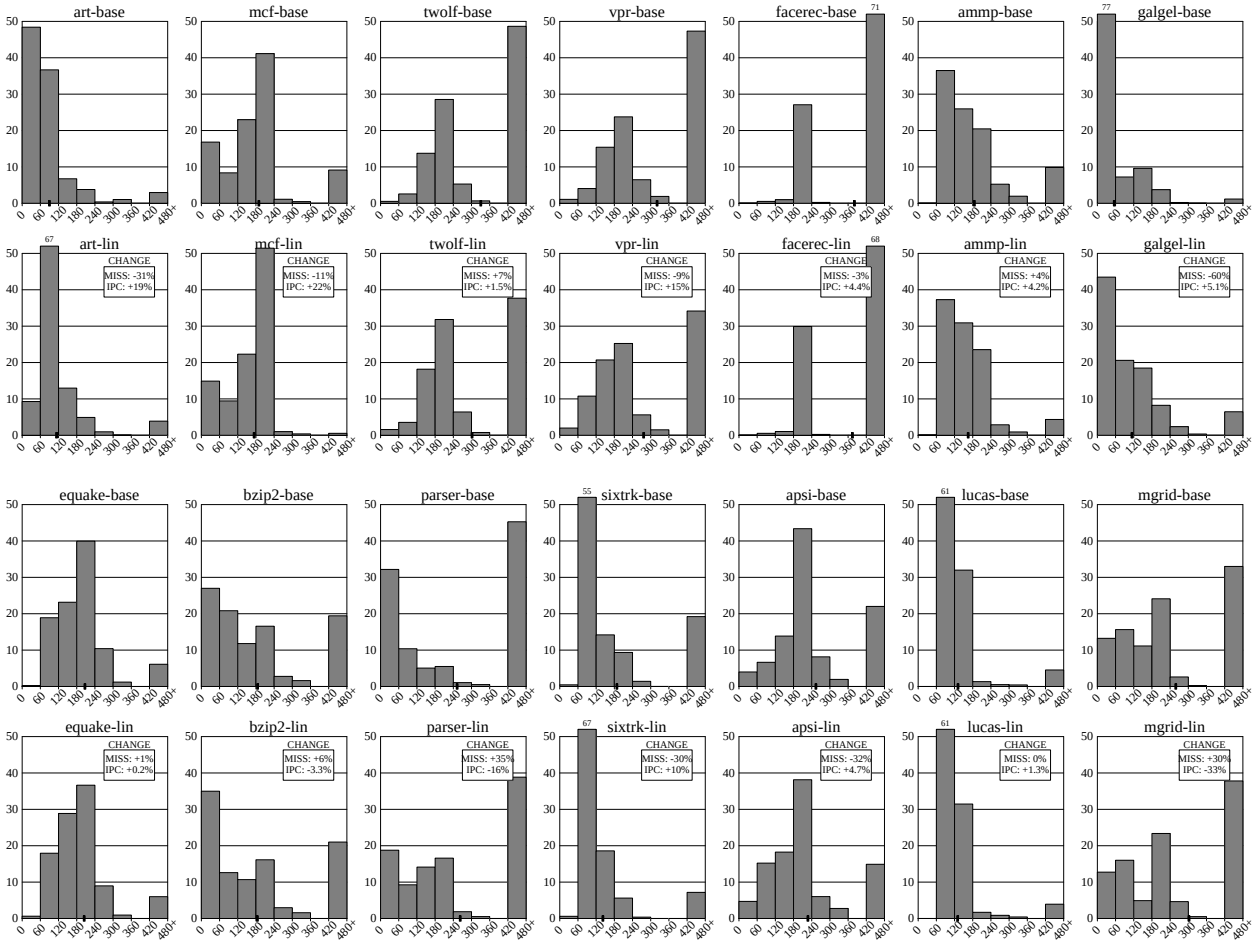


Figure 5: Distribution of mlp-cost for baseline and LIN ($\lambda = 4$). The horizontal axis represents the value of mlp-cost in cycles and the vertical axis represents the percentage of all misses. The dot on the horizontal axis represents the average value of mlp-cost. The insets in the graphs contain information about the change in the number of misses and IPC with the use of the LIN policy.

the high mlp-cost blocks in the cache, when LRU could have potentially caused thrashing. The large reduction in the number of misses for art and galgel reduces the parallelism with which the remaining misses get serviced. Hence, for both art and galgel, the average mlp-cost with the LIN policy is slightly higher than for the baseline.

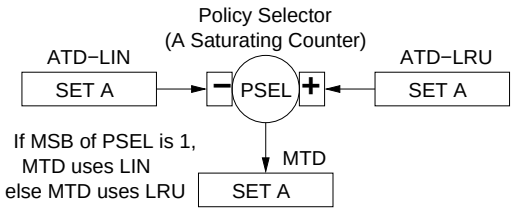
The LIN policy tries to retain recent cache blocks that have high mlp-cost values. The implicit assumption is that the blocks that were high mlp-cost at the time they were brought in the cache will continue to be high mlp-cost the next time they need to be fetched. Therefore, the LIN policy performs poorly for benchmarks in which current mlp-cost is not a good indicator of the next-time mlp-cost. Examples of such benchmarks are bzip2 (average delta = 126 cycles), parser (average delta = 190 cycles), and mgrid (average delta = 187 cycles). For these benchmarks, the number of misses increases significantly with the LIN policy. For the LIN policy to be useful for a wide variety of applications, we need a feedback mechanism that can limit the performance degradation caused by LIN. This can be done by dynamically choosing between the baseline LRU policy and the LIN policy depending on which policy is doing better. The next section presents a novel, low-overhead, adaptation scheme that provides such a capability.

6 A Practical Approach to Hybrid Replacement

LIN performs better on some benchmarks and LRU performs better on some benchmarks. We want a mechanism that can dynamically choose the replacement policy that provides higher performance, or equivalently fewer memory-related stall cycles. A straightforward method of doing this is to implement both LIN and LRU in two additional tag directories (note that data lines are not required to estimate the performance of replacement policies) and to keep track of which of the two policies is doing better. The main tag directory of the cache can select the policy that is giving the lowest number of memory-related stalls. In fact, a similar technique of implementing multiple policies and dynamically choosing the best performing policy is well understood for implementing hybrid branch predictors [12]. However, to our knowledge, previous research has not looked at dynamically selecting between multiple cache replacement schemes by implementing the multiple schemes concurrently. Part of the reason is that the hardware overhead of implementing two or more additional tag directories, each the same size as the directory of the main cache, is expensive. To reduce this hardware overhead, we provide a novel, cost-effective solution that makes hybrid replacement practical. We describe our selection mechanism before describing the final cost-effective solution.

6.1 Contest Based Selection of Replacement Policy

Let MTD be the main tag directory of the cache. For facilitating hybrid replacement, MTD is capable of implementing both LIN and LRU. MTD is appended with two Auxiliary Tag Directories (ATDs): ATD-LIN and ATD-LRU. Both ATD-LIN and ATD-LRU have the same associativity as MTD. ATD-LIN implements only the LIN policy, and ATD-LRU implements only the LRU policy. A saturating counter called the *policy selector (PSEL)* keeps track of which of the two ATDs is doing better. The access stream visible to MTD is also fed to both ATD-LIN and ATD-LRU. Both ATD-LIN and ATD-LRU compete and the output of PSEL is an indicator of which policy is doing better. The replacement policy to be used in MTD is chosen based on the output of PSEL. We call this mechanism Contest Based Selection (CBS). Figure 6 shows the operation of the CBS mechanism for one set in the cache.



ATD−LIN	ATD−LRU	Action
HIT	HIT	PSEL unchanged
MISS	MISS	PSEL unchanged
MISS	HIT	Decrement PSEL by Cost $_q$ of Miss in ATD−LIN
HIT	MISS	Increment PSEL by Cost $_q$ of Miss in ATD−LRU

Figure 6: Contest-Based Selection for a single set.

If a given access hits or misses in both ATD-LIN and ATD-LRU, neither policy is doing better than the other. Thus, PSEL remains unchanged. If an access misses in ATD-LIN but hits in ATD-LRU, LRU is doing better than LIN for that access. In this case, PSEL is decremented by a value equal to the cost_q of the miss (a 3-bit value) incurred by ATD-LIN. Conversely, if an access misses in ATD-LRU but hits in ATD-LIN, LIN is doing better than

LRU. Therefore, PSEL is incremented by a value equal to the cost_q of the miss incurred by ATD-LRU.[6] Unless stated otherwise, we use a 6-bit PSEL counter in our experiments. All PSEL updates are done using saturating arithmetic.

If LIN reduces memory-related stall cycles more than LRU, then PSEL will be saturated towards its maximum value. Similarly, PSEL will be saturated towards zero if the opposite is true. If the most significant bit (MSB) of PSEL is 1, the output of PSEL indicates that LIN is doing better. Otherwise, the output of PSEL indicates that LRU is doing better. Note that PSEL is incremented or decremented by cost_q instead of by 1, which results in selection based on the cumulative value of MLP-based cost of misses (i.e., $\sum \mathsf{cost}_q$), rather than the raw number of misses. This is an important factor in the CBS mechanism that allows it to select the policy that results in the smallest number of stall cycles, rather than the smallest number of misses. If the value of cost_q is constant or random, then the adaptation mechanism automatically degenerates to selecting the policy that gives the smallest number of misses.

6.2 Sampling: The Need, the Insight, and the Concept

A simple, but expensive, way to implement hybrid replacement is to implement the CBS mechanism for every set in the cache. In such an implementation, for each set in MTD, there would be a corresponding set in ATD-LIN and ATD-LRU, and a PSEL counter. MTD can consult the PSEL counter corresponding to its set for choosing between LIN and LRU. We call this implementation CBS-local as it implements CBS locally on a per-set basis. CBS-local requires two ATDs, each sized the same as MTD, which makes it a high-overhead option.

Another method of extending the CBS mechanism for the entire cache is to have both ATD-LIN and ATD-LRU feed a single global PSEL counter. The output of the single PSEL decides the policy for *all* the sets in MTD. We call this mechanism CBS-global. An example of the CBS-global scheme is shown in Figure 7(a) for a cache that has eight sets. Note that CBS-global reduces the number of PSEL counters to only one, but it does not reduce the number of costly ATD entries associated with each set in CBS-local.

The key insight that allows us to reduce the number of ATD entries for CBS-global is the realization that it is not necessary to have all the sets participate in deciding the output of PSEL. If only a few sampled sets are allowed to decide the output of PSEL, then the CBS-global mechanism will still choose the best performing policy with a high probability. The sets that participate in updating PSEL are called *Leader Sets*. Figure 7(b) shows a CBS-global mechanism with *sampling*. Sets B, E, and G are the leader sets. These sets have ATD entries and are the only sets that update the PSEL counter. There are no ATD entries for the remaining sets. For the example in Figure 7(b), sampling reduces the number of ATD entries required for the CBS-global mechanism to 3/8 of its original value. A natural question is: how many leader sets are sufficient to select the best performing replacement policy? We provide both analytical as well as empirical answers to this question.

[6] Only accesses that result in a miss for MTD are serviced by the memory system. If an access results in a hit for MTD but a miss for either ATD-LIN or ATD-LRU, then it is not serviced by the memory system. Instead, the ATD which incurred the miss finds a replacement victim using its replacement policy. The tag-store information associated with the replacement victim of ATD is updated. The value of cost_q associated with the block is obtained from the corresponding tag-store entry in MTD.

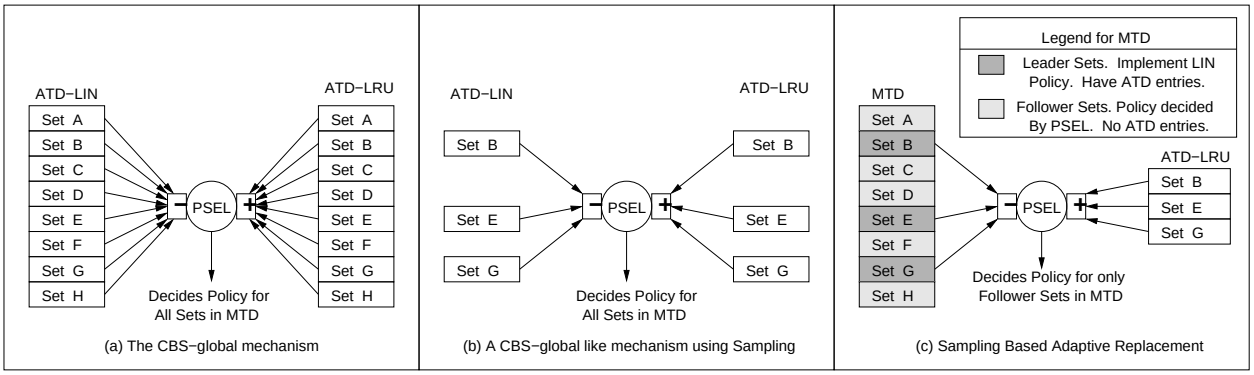


Figure 7: (a) The CBS-global mechanism (b) An approximation to CBS-global mechanism using sampling (c) Sampling Based Adaptive Replacement(SBAR) demonstrated on a cache that has eight sets.

6.3 Analytical Treatment of Sampling

To keep the analysis tractable, we make the simplifying assumption that all sets affect performance equally. Let $P(Best)$ be the probability that the best performing policy is selected by the sampling-based CBS-global mechanism. Let there be N sets in the cache. Let p be the fraction of the sets that favor the best performing policy. Given that we have two policies, LRU and LIN, by definition $p \geq 0.5$.

If only one set is selected at random from the cache as the leader set, then $P(Best) = p$. If three sets (N>>3) are chosen at random from the cache as leader sets, then for the mechanism to correctly select the globally best performing policy, at least two of the three leader sets should favor the globally best performing policy. Thus, for three leader sets, $P(Best)$ is given by:

$$P(Best) = p^3 + 3 * p^2 * (1 - p) \quad (3)$$

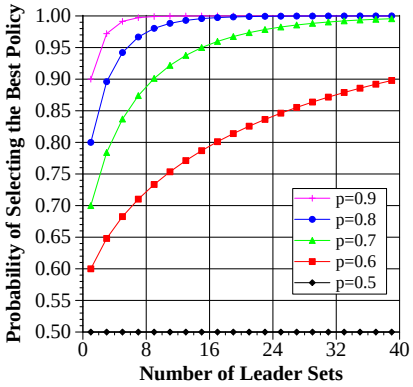


Figure 8: Analytical model for sampling.

In general, if k sets (k << N) are randomly selected from the cache as leader sets, then $P(Best)$ is given by:

$$For\ odd\ k, \quad P(Best) = \sum_{i=0}^{(k+1)/2} ({}^kC_i) * p^{(k-i)} * (1-p)^i \quad (4)$$

$$For\ even\ k, \quad P(Best) = \sum_{i=0}^{(-1+k/2)} ({}^kC_i) * p^{(k-i)} * (1-p)^i + (1/2) * ({}^kC_{k/2}) * p^{k/2} * (1-p)^{k/2} \quad (5)$$

Where $({}^kC_i)$ refers to the operation k-choose-i $(k!/(i! * (k-i)!)$. Figure 8 plots $P(Best)$ for different numbers of leader sets as p is varied. Experimentally, we found that the average value of p for all benchmarks is between 0.74 and 0.99. From Figure 8 we can conclude that a small number of leader sets (16-32) is sufficient to select the globally best-performing policy with a high ($> 95\%$) probability. This is an important result because it means that the baseline cache can have the expensive ATD entries for only 16-32 sets (i.e., about 2% to 3% of all sets) instead of all the 1024 sets in the cache.

6.4 Sampling Based Adaptive Replacement

Sampling makes it possible to choose the best performing policy with high probability even with very few sets in the ATD. Because the number of leader sets is small, the hardware overhead can further be reduced by embedding the functionality of one of the ATDs in MTD. Figure 7(c) shows such a sampling-based hybrid scheme, called Sampling Based Adaptive Replacement (SBAR). The sets in MTD are logically divided into two categories: *Leader Sets* and *Follower Sets*. The leader sets in MTD implement only the LIN policy for replacement and participate in updating the PSEL counter. The follower sets implement both the LIN and the LRU policies for replacement and use the PSEL output to choose their replacement policy. The follower sets do not update the PSEL counter. There is only a single ATD, ATD-LRU. ATD-LRU implements only the LRU policy and only has sets corresponding to the leader sets. Thus, the SBAR mechanism requires a single ATD with entries only corresponding to the leader sets.

We now discuss a method to select leader sets. Let N be the number of sets in the cache and K be the number of leader sets (in our studies we restrict the number of leader sets to be a power of 2). We logically divide the cache into K equally-sized regions each containing N/K sets. We call each such region a *constituency*. One leader set is chosen from each constituency, either statically at design time or dynamically at runtime. A bit associated with each set then identifies whether the set is a leader set. We propose a leader set selection policy that obviates the need for marking the leader set in each constituency on a per-set basis. We call this policy the *simple-static* policy. It selects set 0 from constituency 0, set 1 from constituency 1, set 2 from constituency 2, and so on. For example, if K=32 and N=1024, the simple-static policy selects sets 0, 33, 66, 99,..., and 1023 as leader sets. For the leader sets, bits [9:5] of the cache index are identical to the bits [4:0] of the cache index, which means that the leader sets can easily be identified using a single five-bit comparator without any additional storage. Unless stated otherwise, we use the simple-static policy with 32 leader sets in all our SBAR experiments. We analyze the effect of different leader set selection policies and different number of leader sets in Section 6.6.

6.5 Results for Sampling Based Adaptive Replacement

Figure 9 shows the IPC improvement over the baseline configuration when the SBAR mechanism is used to dynamically choose between LRU and LIN. For comparison, the IPC improvement provided by the LIN policy is also shown. For art, mcf, vpr, facerec, sixtrack, and apsi, SBAR maintains the performance improvement provided by LIN. The most important contribution of SBAR is that it eliminates the performance degradation caused by LIN on bzip2, parser, and mgrid. For these benchmarks, the PSEL in the SBAR mechanism is almost always biased towards LRU. The marginal performance loss in these three benchmarks is because the leader sets in MTD still use only LIN as their replacement policy. For ammp and galgel, the SBAR policy does better than either LIN or LRU alone. This happens because in some phases of the program LIN does better, while in others LRU does better. With SBAR, the cache is able to select the policy better suited for each phase, thereby allowing it to outperform either policy implemented alone. In Section 7.1, we analyze the ability of SBAR to adapt to varying program phases using ammp as a case study.

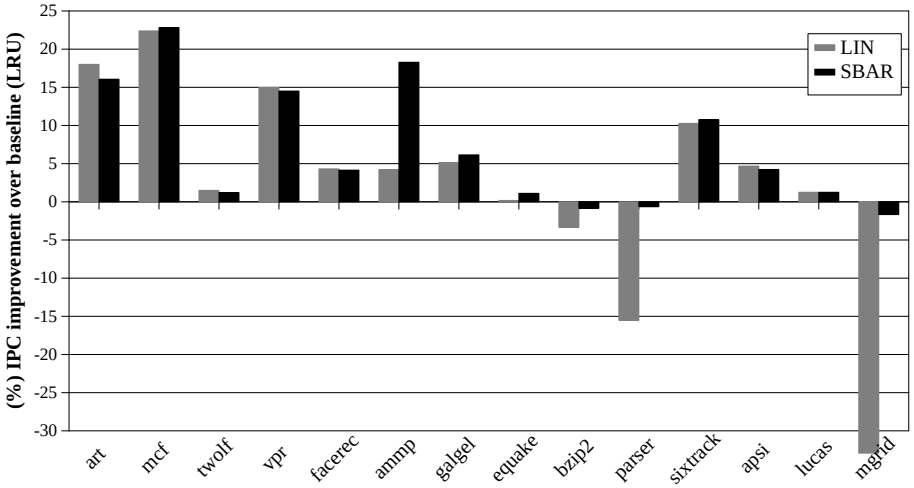


Figure 9: IPC variation with the SBAR mechanism.

6.6 Effect of Leader Set Selection Policies and Different Number of Leader Sets

To analyze the effect of leader set selection policies, we introduce a runtime policy, *rand-dynamic*. Rand-dynamic randomly selects one set from each constituency as the leader set. In our experiments, we invoke rand-dynamic once every 25M instruction and mark the sets chosen by rand-dynamic as leader sets for the next 25M instruction. Figure 10 shows the performance improvement for the SBAR policy with the simple-static policy and the rand-dynamic policy for 8, 16, and 32 leader sets.

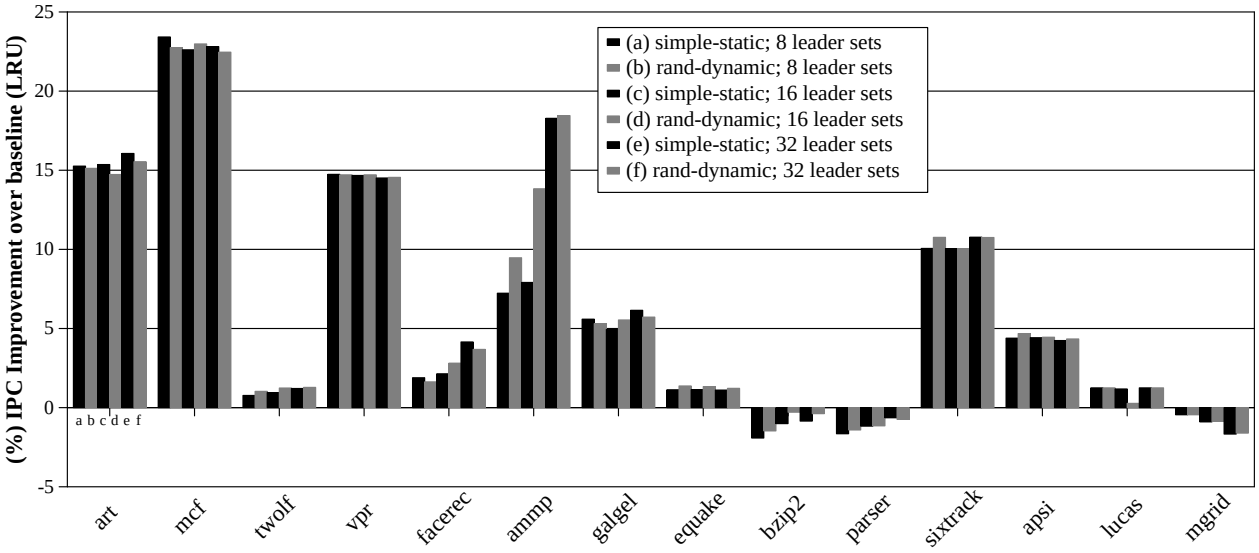


Figure 10: Performance impact of SBAR for different leader set selection policies and different number of leader sets.

For all benchmarks, except ammp, the IPC improvement of SBAR is relatively insensitive to both the leader set selection policy and the number of leader sets. In most benchmarks, one replacement policy does overwhelmingly

better than the other. This causes almost all the sets in the cache to favor one policy. Hence, even as few as eight leader sets are sufficient, and the simple-static policy works well. For ammp, the rand-dynamic policy performs better than the simple-static policy when the number of leader sets is 16 or smaller. This is because ammp has widely-varying demand across different cache sets, which is better handled by the random selection of the rand-dynamic policy than the rigid static selection of the simple-static policy. However, when the number of leader sets increases to 32, the effect of the set selection policy is less pronounced, and there is hardly any performance difference between the two set selection policies. Due to its simplicity, we use the simple-static policy with 32 leader sets as default in all our SBAR experiments.

We also compared SBAR to CBS-global and CBS-local and found that, except for art and ammp, the IPC increase provided by the SBAR policy is within 1% of the best performing CBS-global[7] or CBS-local policies. For ammp, CBS-global improves IPC by 20.3% while SBAR improves IPC by 18.3%. For art, CBS-local improves IPC by 18% whereas SBAR improves IPC by 16%. However, SBAR requires 64 times fewer ATD entries than CBS-local or CBS-global, which makes it a much more practical solution.

7 Analysis

7.1 Ammp: A Case Study

For ammp, SBAR improves IPC by 18.3% over the baseline LRU policy while the LIN policy improves IPC by only 4.2%. This difference in IPC improvement between SBAR and LIN is because ammp has two distinct phases: in one phase LIN performs better than LRU and in the other LRU performs better than LIN. To view this time-varying phase behavior, we collected statistics from the cache every 10M retired instructions during simulation. Figure 11(a) shows the average $\texttt{cost}_q$ per miss, Figure 11(b) shows the misses per 1000 retired instructions, and Figure 11(c) shows the IPC for three different policies: LRU, LIN, and SBAR over time during the simulation runs.

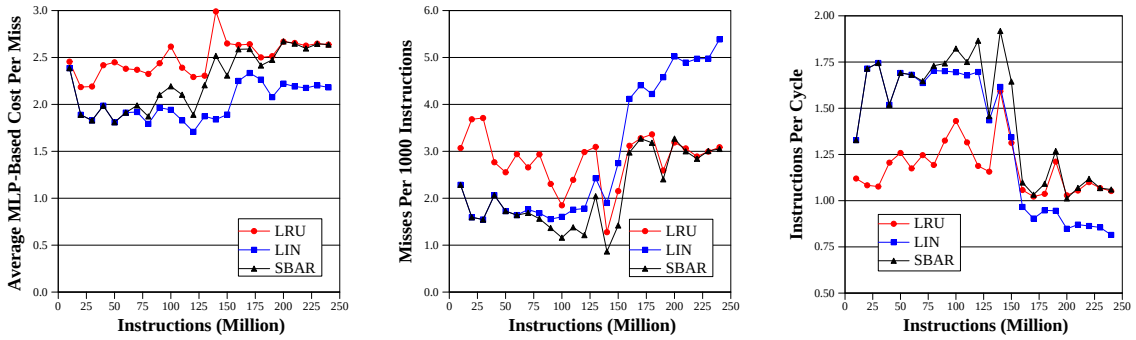


Figure 11: Comparison of LRU, LIN, and SBAR for the ammp benchmark in terms of: (a) the average cost of misses, (b) the number of misses per 1000 instructions, and (c) IPC performance.

[7]The CBS-global policy has 1024 sets updating a single global PSEL counter. We found that, for the CSB-global policy, a 7 bit PSEL counter is more effective than the default 6-bit PSEL counter. We use a 7-bit PSEL counter for implementing the CBS-global policy.

As expected, LIN results in lower cost_q per miss than LRU throughout the whole simulation, indicating that the LIN policy is successful at reducing the cost_q of misses. However, this reduction can come at the expense of significantly increasing the raw number of misses, which may negatively impact the IPC performance. Until 150M instructions, this is not a problem: LIN has both lower cost_q per miss and fewer misses than LRU. Therefore, the IPC with LIN is much better than the IPC with LRU for the first 150M instructions. However, after 150M instructions, LIN has significantly more misses than LRU, which reduces the IPC for the LIN policy compared to LRU. With SBAR, the cache dynamically adapts and uses the policy that is best suited for each phase: LIN until 150M instructions and LRU after 150M instructions. Therefore, SBAR provides higher performance than both LIN and LRU.

7.2 Quantifying the Storage Cost of MLP-Aware Replacement

The performance improvement of MLP-aware replacement comes at a small hardware overhead. For each entry in the MSHR, an additional 14 bits are required to compute the mlp-cost.[8] Also, cost_q is stored in each tag-store entry in the cache, increasing the size of each tag-store entry by three bits. If SBAR is used to adaptively choose between LRU and LIN, then additional storage is required for the ATD entries. Table 4 details the storage overhead of SBAR assuming a 40-bit physical address space and 32 leader sets. SBAR requires a storage overhead of 1856 bytes, which is less than 0.2% of the total area of the baseline L2 cache.

Table 4: Storage overhead of SBAR.

Size of each ATD entry (1 valid bit + 24-bit tag + 4-bit LRU)	29 bits
Total number of ATD entries per leader set	16
ATD overhead per leader set (29 bits/way * 16 ways)	58 B
Total SBAR overhead (32 leader sets * 58 B/set)	1856 B
Area of baseline L2 cache (64kB tags + 1MB data)	1088 kB
Percentage increase in L2 cache area due to SBAR (1856B/1088kB)	**0.166%**

7.3 MLP-Aware Replacement using an Existing Cost-Sensitive Replacement Policy

We proposed the SBAR mechanism to implement a MLP-aware cache replacement policy. However, the central idea of this paper, MLP-aware cache replacement, is not limited in implementation to the proposed SBAR mechanism. Our framework for MLP-aware cache replacement makes even existing cost-sensitive replacement policies applicable to the MLP domain. As an example, we use Adaptive Cost-Sensitive LRU (ACL) [8] to implement a MLP-aware replacement policy. ACL was proposed for cost-sensitive replacement in Non-Uniform Memory Access (NUMA) systems and used the bank access latency as the *cost* parameter. Similarly, MLP information about a cache block can also be used as a *cost* parameter in ACL. Figure 12 shows the performance improvement of an MLP-aware replacement scheme implemented using ACL. For comparison, the results for SBAR is also shown.

[8] We assume that each MSHR entry stores the mlp-cost in a 9.5 fixed point format, where 9 bits are used to encode the integer part and 5 bits are used to encode the fractional part. As the MSHR contains 32 entries, the minimum value used to update mlp-cost is 1/32, hence the 5 bits for the fractional part. Also, if mlp-cost is more than 420 cycles it does not need to be updated any further if the quantization scheme shown in Figure 3(b) is used, hence the 9 bits for the integer part.

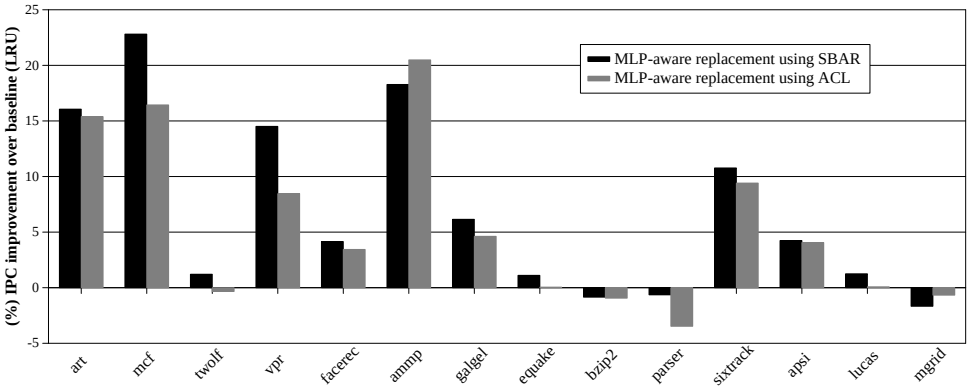


Figure 12: MLP-aware replacement using different cost-sensitive policies.

MLP-aware replacement improves performance for both implementations: ACL and SBAR, indicating that MLP-aware replacement works well with both existing(ACL) and proposed (SBAR) cost-sensitive polices. However, SBAR has higher performance and 33 times lower hardware overhead than ACL[9], which makes SBAR a much more favorable candidate for implementing MLP-aware cache replacement.

8 Conclusion

Memory Level Parallelism (MLP) varies across different misses of an application, causing some misses to be more costly on performance than others. The non-uniformity in the performance impact of cache misses can be exposed to the cache replacement policy so that it can improve performance by reducing the number of costly misses. Based on this observation, we propose MLP-aware cache replacement. We present a run-time technique to compute the MLP-based cost for each cache block. This cost metric is used to drive cost-sensitive cache replacement policies. We also propose Sampling Based Adaptive Replacement (SBAR) to dynamically choose between a MLP-aware replacement policy (LIN) and a traditional (LRU) replacement policy, depending on which is giving better performance. Our experiments reveal that MLP-aware cache replacement can improve performance by up to 23%.

The two key ideas proposed in this paper, MLP-aware cache replacement and Sampling Based Adaptive Replacement can both be extended in several directions. The MLP-aware replacement concept can be extended to take into account prefetching, runahead execution, and other MLP improving techniques. This paper used SBAR for choosing between LIN and LRU. However, SBAR is a general framework that allows dynamic selection between multiple competing replacement policies depending on which one is providing higher performance. The idea of sampling can also be used for other cache related optimizations, such as dynamically tuning the parameters of a given replacement policy, reducing the hardware overhead of an expensive replacement policy (e.g., [13]), or reducing the pollution caused by prefetching mechanisms. Exploring these optimizations is part of our future work.

[9]The cost-sensitive policy employed by ACL requires a shadow directory on a per-set basis. For the baseline 16-way cache, ACL needs a 15-way shadow directory [8]. Assuming a 40-bit physical address space, each entry in the shadow directory needs four bytes of storage (24-bit tag + 1valid bit + 4 LRU bits + 3 cost bits = 4B). Thus, the total overhead of the shadow directory is 60kB (4B/entry * 15 entries/set * 1024 sets = 60 kB). Comparatively, the overhead of SBAR is only 1854B (see Table 4), which is 33 times smaller than the overhead of ACL. Because ACL requires shadow directory information on a per-set basis, it is not straightforward to use sampling to reduce the storage overhead of ACL.

References

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the 36th Annual ACM/IEEE International Symposium on Microarchitecture*, 2003.

[2] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. In *IBM Systems journal*, pages 78–101, 1966.

[3] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proceedings of the 31th Annual International Symposium on Computer Architecture*, 2004.

[4] A. Cristal, O. J. Santana, F. Cazorla, M. Galluzzi, T. Ramirez, M. Pericas, and M. Valero. Kilo-instruction processors: Overcoming the memory wall. *IEEE Micro*, 25(3), May 2005.

[5] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 1997 International Conference on Supercomputing*, pages 68–75, 1997.

[6] A. Glew. MLP yes! ILP no! In *Wild and Crazy Ideas Session. 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, October 1998.

[7] J. Jeong and M. Dubois. Optimal replacements in caches with two miss costs. In *SPAA '99: Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures*, 1999.

[8] J. Jeong and M. Dubois. Cost-sensitive cache replacement algorithms. In *Proceedings of the Ninth IEEE International Symposium on High Performance Computer Architecture*, 2003.

[9] T. Karkhanis and J. E. Smith. A day in the life of a data cache miss. In *Second Annual Workshop on Memory Performance Issues*, 2002.

[10] T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *Proceedings of the 31th Annual International Symposium on Computer Architecture*, 2004.

[11] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, 1981.

[12] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[13] N. Megiddo and D. Modha. ARC: A low overhead self tuning replacement cache. In *USENIX File and Storage Technologies (FAST)*, 2003.

[14] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the Ninth IEEE International Symposium on High Performance Computer Architecture*, 2003.

[15] N. Young. The K-server dual and loose competetiveness for paging. *Algorithmica*, 11(2):525–541, 1994.

[16] V. S. Pai and S. Adve. Code transformations to improve memory parallelism. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, 1999.

[17] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, 1994.

[18] E. Perelman, G. Hamerly, M. V. Biesbrouck, T. Sherwood, and B. Calder. Using simpoint for accurate and efficient simulation. *SIGMETRICS Perform. Eval. Rev.*, 31(1), 2003.

[19] S. T. Srinivasan, R. D. ching Ju, A. R. Lebeck, and C. Wilkerson. Locality vs. criticality. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001.

[20] S. T. Srinivasan and A. R. Lebeck. Load latency tolerance in dynamically scheduled processors. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, 1998.

[21] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, 2004.

[22] M. V. Wilkes. The memory gap and the future of high performance memories. *ACM Computer Architecture News*, 29(1):2–7, Mar. 2001.

[23] W. A. Wong and J.-L. Baer. Modifiied LRU policies for improving second-level cache behavior. In *Proceedings of the Sixth IEEE International Symposium on High Performance Computer Architecture*, 2000.

[24] H. Zhou. Dual-core execution: Building a highly scalable single-thread instruction window. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, 2005.

[25] H. Zhou and T. M. Conte. Enhancing memory level parallelism via recovery-free value prediction. In *Proceedings of the 17th International Conference on Supercomputing*, 2003.