Improving Cache Performance by Combining Cost-Sensitivity and Locality Principles in Cache Replacement Algorithms [1]

Rami Sheikh
North Carolina State University
rmalshei@ncsu.edu

Mazen Kharbutli
Jordan Univ. of Science and Technology
kharbutli@just.edu.jo

***Abstract*—Due to the ever increasing performance gap between the processor and the main memory, it becomes crucial to bridge that gap by designing an efficient memory hierarchy capable of reducing the average memory access time. The cache replacement algorithm plays a central role in designing an efficient memory hierarchy. Many of the recent studies in cache replacement algorithms have focused on improving L2 cache replacement algorithms by minimizing the miss count. However, depending on the dependency chain, cache miss bursts, and other factors, a processor's ability to partially hide the cost of an L2 cache miss varies; that is, cache miss costs are not uniform. Therefore, a better solution would account also for the aggregate miss cost in designing cache replacement algorithms. Our proposed solution combines the two principles of locality and cost-sensitivity into one which we call: LACS: Locality-Aware Cost-Sensitive cache replacement algorithm. LACS estimates a cache block's cost from the number of instructions the processor manages to issue during a cache miss on that block and then victimizes cache blocks with low cost and poor locality in order to maximize the overall cache performance. When LACS is evaluated using a uniprocessor architecture model, it speeds up 10 L2 cache performance-constrained SPEC CPU2000 benchmarks by up to 85% and 15% on average while not slowing down any of the 20 SPEC CPU2000 benchmarks evaluated. When evaluated using a dual-core CMP architecture model, LACS speeds up 6 SPEC CPU2000 benchmark pairs by up to 44% and 11% on average.**

I. INTRODUCTION

With the performance gap between the processor and main memory continuously widening, the design of an effective memory hierarchy becomes even more crucial in order to reduce the average memory access times seen by the processor. The design of an effective L2 cache has been the center of numerous research papers for several reasons: First, although a processor may be able to hide an L1 cache miss followed by an L2 cache hit by exploiting ILP, out-of-order execution, and non-blocking caches, it is almost impossible to fully hide the long L2 cache miss penalty. Second, with the L1 cache optimized for short hit times, the L2 cache's hit time becomes less critical allowing expensive optimizations such as smarter replacement algorithms [1],[2].

A crucial design issue for L2 caches is the replacement algorithm used. Recently, we have seen many papers proposing improved L2 cache replacement algorithms such as dead block predictors [3]-[9], OPT emulators [10], and CMP cache partitioning algorithms [11], among others. Unfortunately, most of these algorithms only target and attempt to reduce the cache's miss rate while only a few attempt to reduce the aggregate miss cost or penalty.

In modern superscalar processors, the processor attempts to hide cache misses by exploiting ILP through issuing and executing independent instructions. Unfortunately, even with the most aggressive superscalar processors, it is quite impossible to hide the large L2 cache miss penalty. During this long miss penalty, the reorder buffer and the other processor queues fill up. The processor may also run out of free physical registers and other resources. All this eventually stalls the whole processor waiting on the L2 cache miss. Yet, depending on the dependency chain, miss bursts and other factors, the processor's ability to partially hide the L2 cache miss penalty differs widely from one miss to another [12].

Fig. 1 illustrates the above point and shows the histogram of the number of issued instructions during the service of an L2 cache miss for the benchmark *mcf* [13]. The number of issued instructions is counted from the time the instruction experiencing the L2 cache miss is issued until the requested data is received. The x-axis shows the number of issued instructions while the y-axis shows the occurrence. For example, looking at the first bar, for about 10,000,000 of the L2 cache misses, the processor managed to issue only 0-10 instructions per miss. We will refer to such a graph as an *Issued Instructions per Miss Histogram*. The figure clearly shows that the number of issued instructions during a miss is not uniform and varies widely. The more instructions the processor manages to issue during the miss, the better it is capable of hiding the miss penalty and the lower the cost of that miss. We define cache blocks where the processor manages to issue a small/large number of instructions during a miss on that block as high/low-cost blocks, respectively. Even if the miss rate is not reduced, substituting high-cost misses by low-cost misses would reduce the aggregate miss penalty and thus enhance performance.

[1] Most of this work is based on Sheikh's M.S. thesis at Jordan University of Science and Technology.

978-1-4244-8935-0/10/$26.00 ©2010 IEEE

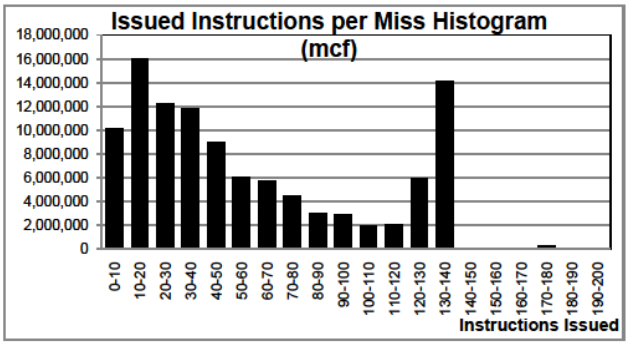

Fig. 1: Issued Instructions per L2 Cache Miss Histogram for *mcf*. Simulation parameters follow those in Table 1.

In this paper, we propose a novel cache replacement algorithm that we call LACS: Locality-Aware Cost-Sensitive Cache Replacement Algorithm. LACS estimates the cost of a cache block based on the processor's ability to (partially) hide its miss penalty by counting the number of instructions issued during the miss. It then classifies blocks as low-cost or high-cost blocks. On a cache miss, when a victim block needs to be found, LACS chooses a low-cost block keeping high-cost blocks in the cache. This is called block reservation [14]-[17]. However, since a block with a high cost cannot be reserved forever, a mechanism must exist to relinquish the reservation after some time. To achieve this, LACS relies on an underlying locality-based algorithm such as LRU or pseudo-LRU to estimate a cache block's locality. As a result, LACS attempts to reserve high-cost blocks in the cache, but only while their locality is still high (i.e. they have been accessed recently). The underlying locality-based algorithm can also be a dead block predictor. We attempted to integrate LACS with several dead-block predictors [3],[6],[7], but found that, although dead block predictors outperform LRU as cache replacement algorithms, LRU and dead block predictors almost perform equally when their goal is limited to providing locality hints to LACS.

This paper has two main contributions:

- First, a novel, effective, but simple, cost estimation method is presented. The cost is estimated based on the number of instructions the processor manages to issue during the miss, which reflects how well the processor is capable of hiding the miss penalty. The effectiveness of this cost estimation method is demonstrated in both uniprocessor and CMP architectures alike.
- Second, this cost estimation method is integrated with a locality-based algorithm resulting in an effective cache replacement algorithm (LACS) that is both cost-sensitive and locality-aware.

LACS is evaluated using a detailed simulation environment. When evaluated using a uniprocessor architecture model, LACS speeds up 10 L2 cache performance-constrained SPEC CPU2000 benchmarks by up to 85% and 15% on average while not slowing down any of the 20 SPEC CPU2000 benchmarks used in the study by more than 1%. When evaluated using a dual-core CMP architecture model, LACS speeds up 6 SPEC CPU2000 benchmark pairs by up to 44% and 11% on average. These performance improvements are achieved using a simple cost estimation method and a 48 KB prediction table. The prediction table is only accessed during an L2 cache miss allowing it to be placed off-chip and its access time to be fully-overlapped with the L2 cache miss latency. In addition, LACS's effectiveness is demonstrated over a wide range of L2 cache configurations achieving performance improvement for different L2 cache sizes and associativities.

The rest of the paper is organized as follows. Section II presents the related work and compares LACS to other replacement algorithms. Section III discusses our proposed algorithm in detail. Section IV describes the evaluation environment and Section V discusses the experimental evaluation. Finally, Section VI concludes the paper.

II. RELATED WORK

Cache replacement algorithms were originally developed with the aim of reducing the aggregate miss count and thus assumed that misses are uniform. Belady's optimal (OPT) replacement algorithm [18] victimizes the block in the set with the largest future usage distance. It guarantees a minimal miss count but requires future knowledge and thus remains theoretical and can only be emulated in real systems [10]. The LRU replacement algorithm and its approximations rely on the principle of temporal locality and victimize the least recently used block in the set. However, studies showed that the performance gap between LRU and OPT was wide for high-associativity L2 caches [8]. To bridge the gap between OPT and LRU, dead block predictors [3]-[9] were proposed and aimed to predict dead blocks (blocks that will no longer be used during their current generation times) in the cache and evict them early while preserving live blocks. Cache replacement algorithms in shared CMP caches have been studied in the context of shared cache partitioning among the concurrently-running threads [11].

Replacement algorithms such as OPT, LRU, dead block predictors, and others only distinguish between blocks in terms of liveness and do not distinguish between blocks in terms of miss costs. However, in modern systems, cache misses are not uniform and possess different costs [12],[19]. Thus, it is wiser to take into consideration the miss costs in addition to the access locality in the replacement policy in order to improve the cache's overall performance, which is exactly what LACS is designed to achieve.

Srinivasan and Lebeck [12] explore latency tolerance in dynamically scheduled processors and show that load instructions are not equal in terms of processor tolerance for load latencies. They also show that load latency tolerance is a function of the number and type of dependent instructions especially mispredicted branches. Moreover, Puzak et al [19] also show that misses have variable costs and present a new simulation-based technique for calculating the cost of a miss

for different cache levels.

The observation of the non-uniform nature of cache misses led to a new class of replacement algorithms called Cost-Sensitive Cache Replacement Algorithms. These algorithms assign different costs to cache blocks according to well-defined criteria, and rely on these costs to select which block to evict on a cache miss (least cost block gets evicted first). The miss cost may be latency, penalty, power consumption, bandwidth consumption, or any other property attached to a miss [14]-[17],[20],[21].

One of the earliest implementations of cost-sensitive cache replacement algorithms were proposed by Jeong and Dubois [14]-[16] in the context of CC-NUMA multiprocessors in which the cost of a miss mapping to a remote memory, as opposed to local memory, is higher in terms of latency, bandwidth, and power consumption. A cost-sensitive optimal replacement algorithm (CSOPT) for CC-NUMA multiprocessors with two static miss costs is evaluated and found to outperform a traditional OPT algorithm in terms of overall miss costs although the miss count increases. In addition, several realizable algorithms are evaluated with a cost based on the miss latency. In comparison, LACS estimates a block's cost based on the processor's ability to tolerate and hide the miss not on the miss latency itself.

Jeong et al [17] also proposed a cost-sensitive cache replacement algorithm in the context of uniprocessors that assigns cost based on whether a block's next access is predicted to be a load (high-cost) or store (low-cost) since processors can better tolerate store misses over load misses. In their implementation, all loads are equal and are considered high-cost. In comparison, LACS does not treat load misses equally but distinguishes between load misses in terms of cost based on the processor's ability to tolerate and hide the load miss. Our study and others' [12],[19] show that load miss costs are not uniform and thus should not be treated equally. Moreover, some store misses may be critical and can stall the processor. This, for example, can happen if the L2 cache MSHR or write buffers get full after a long sequence of consecutive store misses such as when initializing or copying an array.

Srinivasan et al [21] proposed a hardware scheme in which critical blocks are preserved in a special critical cache or used to initiate prefetching. Criticality is estimated by keeping track of a load's dependence chain and the processor's ability to execute independent instructions following the load. Although they demonstrate the effectiveness of their approach when all critical loads are guaranteed to hit in the cache, no significant improvement is achieved under a realistic configuration due to the large working set of critical loads and the inefficient way of identifying critical loads. In comparison, LACS does not need to keep track of a load's dependence chain, instead it uses a simpler and more effective approach for cost estimation. Moreover, LACS also achieves considerable performance improvement under a realistic configuration because high-cost blocks are reserved in the L2 cache itself instead of a smaller critical cache, and because LACS includes a mechanism to relinquish high-cost blocks that may no longer be needed by the processor making room for other useful blocks in the cache.

Qureshi et al [20] proposed a cost-sensitive cache replacement algorithm based on Memory Level Parallelism (MLP). The MLP-aware cache replacement algorithm relies on the parallelism of miss occurrences since some cache misses occur in isolation (classified as high-cost and are thus reserved in the cache) while others occur and get served concurrently (classified as low-cost). Because of its significant performance degradation in pathological cases, it is used in conjunction with a tournament-predictor-like Sampling Based Adaptive Replacement (SBAR) algorithm to choose between the MLP-aware algorithm and the traditional LRU depending on which provides better performance. In Section V, LACS is compared against and is shown to outperform the MLP-aware with SBAR algorithm.

III. LACS: LOCALITY-AWARE COST-SENSITIVE CACHE REPLACEMENT ALGORITHM

A. L2 Cache Miss Costs and Effects on Dynamically-Scheduled Superscalar Processors

Modern dynamically-scheduled superscalar processors improve performance by issuing and executing independent instructions in parallel and out-of-order. Multiple instructions are fetched, issued, and executed every clock cycle. After an instruction completes execution, it writes back its result and waits to be retired (by updating the architectural state) in program order. Although instructions get issued and executed out-of-order, program order is preserved at retirement using the ROB [1]. Dynamically-scheduled superscalar processors can tolerate the high latency of some instructions (e.g. loads that suffer L1 cache misses) by issuing and executing independent instructions. However, there is a limit to the delay a processor can tolerate and may eventually stall. This happens in particular when a load instruction suffers an L2 cache miss and has to be serviced by the long latency L3 cache or the main memory.

The main reason why a processor may stall after an L2 cache load miss is that the ROB would fill up and dependences would clog it [22]. Even if there are not so many dependent instructions, the load instruction would reach the head of the ROB and prevent the retirement of completed instructions following it, again filling up the ROB and preventing the dispatch of new instructions because no free ROB entries are available. The clogging of the ROB has a domino effect on other pipeline queues such as the instruction queue and the issue queues preventing the fetch and dispatch of new instructions. Moreover, even store instructions can stall the processor. This can happen after a long sequence of L2 cache store misses that fill up the cache's MSHR or write buffer preventing new instructions from being added and thus stalling the processor. Such a scenario could happen when a large array is being initialized

or copied. This is why LACS treats store misses similar to load misses and assigns costs to them.

Yet, the processor's ability to tolerate the miss latency differs from one instruction to another [12],[19]. Consider the following two extreme scenarios: In the first, a load instruction is followed by a long chain of dependent instructions that directly or indirectly depend on the load instruction. If the load instruction suffers an L2 cache miss, the processor will stall immediately since none of the instructions following it can issue. After the load instruction completes, the dependent instructions need to be issued and executed. Their execution times will be added to the miss latency. In the second scenario, a load instruction is followed by independent instructions. If the load instruction suffers an L2 cache miss, the processor can still keep busy by issuing and executing the independent instructions. It will only stall once the ROB is clogged by the load instruction and the completed (but not retired) independent instructions. However, once the load instruction retires, all the following completed instructions can retire relatively quickly. The execution times of these instructions will be overlapped with the miss latency and thus saved. Most load instructions exhibit scenarios between these two extremes. Fig. 1 earlier demonstrated this observation and showed that the number of instructions that a processor manages to issue during a cache miss differs widely from one miss to another. Therefore, the cost of the miss can be effectively estimated from the number of instructions issued during the miss. Cache misses in which the processor fails to issue many instructions are considered high-cost while cache misses in which the processor manages to keep busy and issue many instructions are considered low-cost since they can be highly tolerable by the processor.

LACS uses the above heuristic in estimating the cost of a cache block based on whether the miss on the block is a low- or a high-cost miss. If the number of issued instructions during the miss is larger than a threshold value, the block is considered a low-cost block. Otherwise, it is considered a high-cost block. LACS attempts to reserve high-cost blocks in the cache at the expense of low-cost blocks. When a victim block needs to be found, a low-cost block is chosen.

B. LACS Implementation Details

In order for LACS to calculate the number of issued instructions during a miss, a performance counter that tracks the number of issued instructions in the pipeline is needed. This performance counter is incremented for every issued instruction. Such a performance counter (or similar ones) is available in most modern processors. We will call this counter the IIC (Issued Instructions Counter). In addition, every L2 cache MSHR entry is augmented with a field: IIR (Issued Instructions Register) that stores the value of the IIC when the load/store instruction is added to the MSHR. Moreover, for each block (whether in the cache or not), two values are maintained in a prediction table: The block's cost and a confidence value. Confidence bits are used to improve the accuracy because the behavior of the cache block might change when the application is transitioning from one execution phase to another. The cost value is used only if the behavior has stabilized.

Fig. 2 shows the implementation details of LACS. On an L2 cache miss on block B in set S: First, the IIC value is copied into the IIR corresponding to block B's MSHR entry (Step 1). Second, and while the miss is being serviced, a victim block in set S is found. Low-cost blocks in the set are identified and a victim is chosen randomly from among them. If no low-cost block is found, the LRU block is chosen as the victim. A block is considered a low-cost block if its cost is larger than a threshold value and its confidence is nonzero (Steps 2,3). After the miss is serviced, block B is placed in the cache and its new cost value is calculated as the difference between the current IIC and block B's IIR values (Step 4). This difference is equal to the number of issued instructions during the miss. Finally, block B's prediction table information is updated. If there is an entry for block B in the table, then the stored cost value is compared to the new cost value and if they are (approximately) equal, the confidence counter is set. Otherwise, the confidence counter is reset. In addition, the new cost value is stored. If no entry for block B is found in the table, an entry is reserved and initialized with the new cost value and a confidence value of zero (Step 5). LACS's implementation can be performed off the L2 cache's critical path. First, the prediction table is only accessed on a cache miss and therefore its access can be overlapped with the miss latency. Second, block B's table information can be updated offline after the requested data has been forwarded to the processor.

For the choice of the threshold value, we experimented with a static approach versus a dynamic approach, and found that a dynamic approach clearly outperforms a static approach. This is because a dynamic approach can adapt to different application behaviors and different execution phases. In the dynamic approach, which we adopted, the threshold value is one fourth the average of previous cost values. Two registers are used: the first is a Cost Accumulator that accumulates new cost values as they are found. The second is a Cost Counter that is incremented on every cost value found.

```
On an L2 cache miss on block B in set S:
Step 1: MSHR[B].IIR = IIC
Step 2: Identify all low cost blocks in set S.
        A block X is a low cost block if
        (X.cost > threshold) and (X.conf == 1)
Step 3: If there is at least one low cost block in the set, the victim
        is chosen randomly from among them. Otherwise, the
        LRU block is chosen.
Step 4: When the miss returns, calculate block B's cost:
        newCost = IIC – MSHR[B].IIR
Step 5: Update block B's information in the table.
        If(B.cost ≈ newCost) B.conf=1, else B.conf=0
        B.cost = newCost
```

Fig. 2: LACS implementation details.

The threshold value is calculated as:

Threshold = (CostAccumulator / CostCounter) >> 2

The threshold value was selected and fine tuned after studying the Issued Instructions per Miss Histograms of SPEC CPU2000 benchmarks. We found that because these histograms are not uniform, choosing the threshold value as one quarter the average divides the cache blocks almost in half: Half the blocks are high-cost and the other half are low-cost blocks. A higher value would reduce the number of low-cost blocks available for replacement and increase the number of high-cost blocks requiring preservation, and vice versa for a lower threshold value. To account for different execution phases of the program, the Cost Accumulator and Cost Counter are reset every 1 million L2 cache misses.

C. Storage Organization

The storage organization for LACS consists of the following: The IIC is a 32-bit counter that is incremented on every instruction issued in the pipeline. Each IIR is also a 32-bit register that stores the value of the IIC when the miss instruction is added to the MSHR. The prediction table is organized with 8K sets and 4-ways for a total of 32K entries. Each prediction table entry consists of a 6-bit hashed tag, a 5-bit cost value and a 1-bit confidence. Thus, the total prediction table size is 32K entries x 1.5 bytes/entry = 48KB. Assuming a 32 entry MSHR, the total storage overhead for LACS is 48K + (32+1)x4 ≈ 48⅛ KB. This is a reasonable overhead for typical L2 cache sizes. It is equivalent to 9.4% of a 512KB cache or 4.7% of a 1MB cache.

In order to achieve such a small storage overhead, some approximations were made in the stored cost value. First, our studies found that the cost value which is equal to the number of issued instructions during the miss latency is almost never larger than 512. Therefore, a 9-bit cost value is sufficient. If the cost value happens to be larger than 511, it is stored as 511. Moreover, the exact cost value is not really needed since LACS only identifies a low-cost block if its cost value is larger than a threshold. LACS does not classify different levels of low- and high-cost. Therefore, the cost value can be approximated by keeping the 5 most significant bits of the 9-bits described above. As a result, LACS identifies cost value ranges in multiples of 2^4=16. For example, a cost value of 0 indicates that only 0-15 instructions were issued during the miss. Finally, the prediction table is indexed with the lower 13-bits of the block address. The remaining block address bits are hashed into a 6-bit tag value using XORing. The effects of negative aliasing in the table are reduced using the confidence bit.

D. Implementation of LACS in a CMP/SMT Environment

Both the cost estimation heuristic proposed in this paper and LACS can be used in uniprocessor and multiprocessor/multithreading architectures alike. In Section V, we demonstrate the effectiveness of LACS in both uniprocessor and CMP architecture models. However, several issues arise when using LACS for shared L2 caches: First, should the threshold value and prediction table be shared among the competing threads or private for each thread? Second, should the victim block be chosen from among the blocks of the thread experiencing the cache miss or can it be any block in the set? The second point relates to the problem of cache partitioning [11]. LACS is independent of, and can be built over, any cache partitioning algorithm. The first point, however, requires a detailed analysis and is out of the scope of this paper. It is left as future work.

E. Other Implementation Issues

In highly-associative caches (e.g. L2 caches), maintaining the LRU stack information becomes difficult and expensive. Therefore, LRU is approximated using pseudo-LRU algorithms. LACS only uses a locality-based algorithm to provide it with block locality information used for block relinquishing. Thus, LACS can be built over any locality-based algorithm such as LRU, pseudo-LRU, or dead-block predictors. We tested these different options and found that the performance of LACS in all cases was almost identical.

Prefetching is commonly used nowadays and can be easily integrated with LACS. LACS can even help the prefetcher by directing it to prioritize prefetching of high-cost blocks.

Although not evaluated explicitly, we believe the impact of the proposed algorithm on power consumption to be modest: First, the total storage required is small (48KB) compared to typical L2 cache sizes, thus only slightly increasing static power consumption. Second, the prediction table is only accessed on an L2 cache miss (infrequent), thus only slightly increasing dynamic power consumption.

The LACS implementation described earlier assumes fixed memory latency. However, this fixed memory latency used with LACS is more of an approximation than a necessary assumption. LACS estimates a block's cost based on the processor's achieved IPC during the miss. For a fixed miss latency, this is equivalent to the number of issued instructions during the miss (which is what is proposed). For non-fixed miss latency, LACS would have to estimate the block's cost by calculating the processor's achieved IPC during the miss, which is equal to the number of issued instructions during the miss divided by the miss latency cycles. This only requires the addition of a division operation, which can be approximated using simple shifting.

IV. EVALUATION ENVIRONMENT

Our proposed algorithm is evaluated using SESC [23]: a detailed execution-driven cycle-by-cycle simulator environment. Table 1 lists the different architectural parameters used in the evaluation. In addition, 20 of the 26 SPEC CPU2000 benchmarks [13] are used in the evaluation. The missing 6 benchmarks are excluded because our cross-

compiler does not support them. The benchmarks were compiled with gcc using –O3 optimization level. The reference input sets were used for all benchmarks. Each benchmark was simulated for two billion instructions after fast forwarding the first two billion instructions.

Table 2 lists the 20 benchmarks used in the evaluation along with their L2 cache miss rates and the fraction of their execution times stalled on L2 cache misses. This fraction was calculated from the execution times of a regular L2 cache versus a perfect always-hit L2 cache. The benchmarks are divided into two groups based on whether their performance is constrained by the L2 cache performance. A benchmark is considered L2 cache performance-constrained if over 50% of its execution time is spent on L2 cache miss stalls. Those benchmarks are grouped into Group A in the table and are the focus of our evaluation because, since LACS attempts to reduce the aggregate miss cost, its impact will only be evident in benchmarks where L2 cache miss stalls dominate the execution time. The remaining 10 benchmarks are not L2 cache performance-constrained and are placed in Group B.

Table 1: The simulated architecture parameters. AT: Access Time

PROCESSOR			
Clock Rate	5 GHz	Functional Units	8 INT, 8 FP, 2 LD, 2 ST
Fetch/Issue/Retire Width	4		
ROB Entries	256	Physical Registers	156 INT 156 FP
Max Pending LD/ST	48/48		
MEMORY HIERARCHY			
L1 Data Cache	16KB, 2-way, 64B line, WB, LRU, 2-cycle AT		
L1 Inst. Cache	16KB, 2-way, 64B line, WB, LRU, 2-cycle AT		
L2 Unified Cache	512KB, 8-way, 64B line, WB, 10-cycle AT, 32-entry MSHR		
Main Memory	infinite size, 75ns latency, 10 GB/s		
EXTRA STORAGE FOR LACS			
48KB, 4-cycle AT prediction table (32+1 x 4) Bytes for IIC and IIRs			

Table 2: The benchmarks used, their L2 cache miss rates, and the fraction of their execution times stalled on L2 cache misses.

Group A			**Group B**		
Benchmark	Miss Rate	Stalls Fraction	Benchmark	Miss Rate	Stalls Fraction
ammp	80%	91%	apsi	29%	20%
applu	81%	56%	bzip2	31%	24%
art	100%	72%	crafty	1%	4%
equake	83%	68%	gap	95%	11%
gcc	70%	89%	gzip	1%	4%
mcf	79%	83%	mesa	12%	18%
mgrid	78%	65%	parser	18%	39%
swim	64%	52%	perlbmk	9%	13%
twolf	35%	58%	vortex	7%	18%
vpr	24%	51%	wupwise	83%	44%

V. EVALUATION

A. Performance of LACS

Fig. 3 shows the percentage of IPC improvement of different algorithms over the base LRU replacement. LACS randomly chooses a victim block from the low-cost blocks in a set, so it is compared against a random replacement algorithm [RND] to demonstrate that LACS's performance improvement is not due to the randomness in selecting a low-cost block as a victim. Moreover, the total storage of LACS is about 48 KB, therefore, we compare LACS against a larger (512 KB + 64 KB =) 576 KB 9-way LRU cache [576KB]. We also compare LACS against a state-of-the-art cost-sensitive cache replacement algorithm: MLP-aware replacement using SBAR [MLP] [20]. AVG-GrpA and AVG-All show the average performance improvement achieved for Group A benchmarks versus all benchmarks (Groups A+B), respectively. Fig. 3 shows that LACS achieves an average speedup of about 15% on the L2 cache performance-constrained Group A benchmarks and up to 85% in the case of the benchmark gcc. LACS speeds up 6 of the 10 benchmarks by more than 4% (ammp, art, gcc, mgrid, swim, and vpr). Furthermore, LACS does not slow down any benchmark by more than 1%. The average performance improvement on the Group A benchmarks due to increasing the cache size or using random replacement is comparably insignificant, 4% and 5%, respectively, and is less than one third the performance improvement of LACS. Moreover, the random replacement algorithm suffers from pathological behavior for some Group B benchmarks reducing its overall performance improvement for all benchmarks to about 1%.

The average performance improvement from using MLP is about half that achieved using LACS. While LACS does not slow down any application (from Groups A and B) by more than 1%, MLP slows down a couple of Group B benchmarks by up to 20% in the case of the benchmark parser, significantly lowering its overall average performance improvement (AVG-All).

Next, we studied the effect of LACS on the L2 cache miss rates. Fig. 4 shows the L2 cache miss rates for the LRU and LACS replacement algorithms. LACS reduces the miss rates of 4 benchmarks (ammp, art, gcc, and mgrid) significantly. Looking back at Fig. 3, we find that those 4 benchmarks have large performance improvements. Generally, cost-sensitive cache replacement algorithms were thought to reduce the aggregate miss cost at the expense of the miss count. Yet, LACS also reduces the miss count relative to LRU. This result is counter-intuitive and deserves further investigation (below). However, looking more closely at the benchmarks swim and vpr, we find that their miss rates slightly increase (Fig. 4), and yet, their performance is improved by LACS (Fig. 3), which demonstrates LACS's ability to reduce the aggregate miss cost without reducing the miss count for some benchmarks.

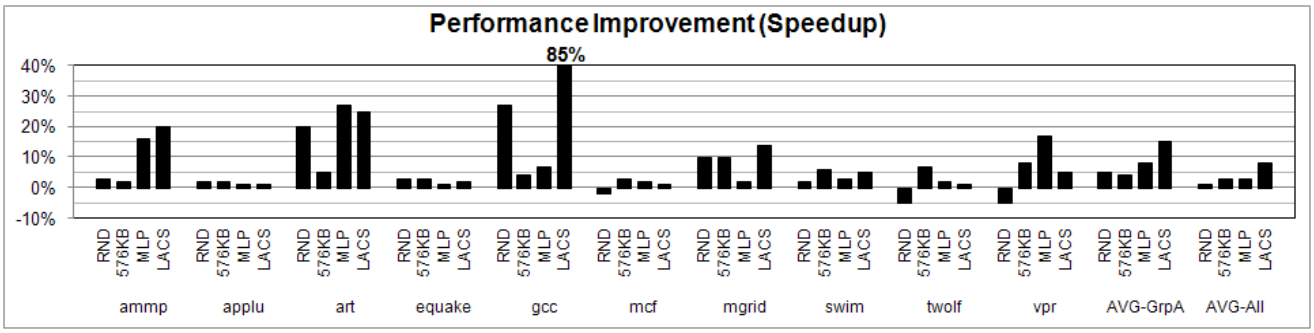

Fig. 3: Performance improvement (Speedup)

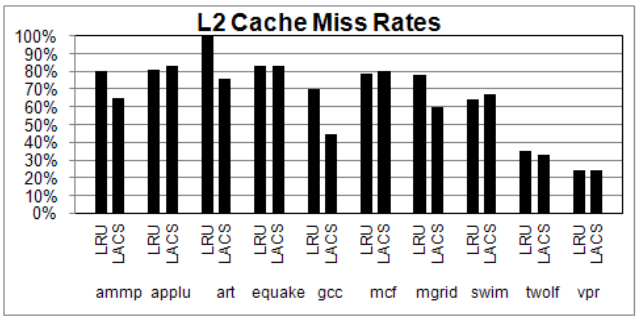

Fig. 4: L2 cache miss rates

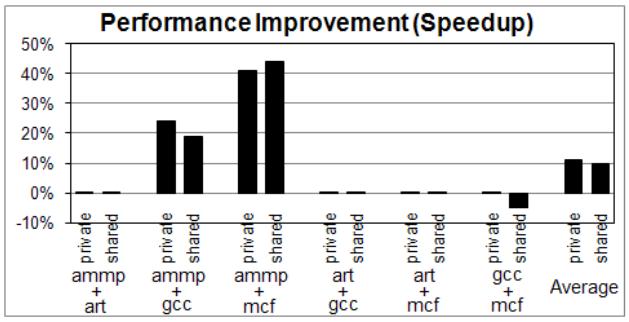

Fig. 5: Performance improvement (Speedup) for a CMP arch.

To investigate why LACS reduces the miss rates of some benchmarks, Table 3 shows the fraction of LRU blocks reserved by LACS that get re-used transforming a miss under LRU to a hit under LACS and thus reducing the overall miss rate. The table shows that the fraction of such blocks in ammp, art, gcc, and mgrid is high clarifying the source of the reduction in their miss rates.

To further understand the above observation, we studied the correlation between blocks evicted by OPT replacement algorithm [18] and their cost as estimated by LACS. We ran simulations with OPT as the replacement algorithm while keeping track of each block's cost as estimated by LACS without using this cost for replacement decisions. We found that, although low-cost blocks accounted for less than 20% of the blocks in the cache, 40% to 98% of OPT evicted blocks were low-cost blocks. This interesting observation is a clear indication of a strong correlation between low-cost blocks and blocks chosen by OPT for eviction.

We also evaluated the effectiveness of LACS in a shared L2 cache dual-core CMP architecture. In the evaluation, the same processor and L1 cache parameters found in Table 1 were used. The L1 caches were assumed to be private while a 2MB 8-way shared L2 cache was used. The prediction table was also shared. We investigated the use of a shared threshold value for both threads and a private threshold value for each. Six benchmark pairs from the Group A set of benchmarks in Table 2 were used.

Table 3: Fraction of LRU blocks reserved under LACS that get re-used

ammp	applu	art	equake	gcc	mcf	mgrid	swim	twolf	vpr
94%	22%	51%	15%	89%	1%	33%	11%	21%	22%

Fig. 5 demonstrates the effectiveness of LACS when used in a CMP architecture. LACS speeds up the benchmark pairs by up to 44% and 11% on average compared to using LRU replacement. The results show no clear insight on whether it is better to use shared or private threshold values. A more comprehensive evaluation of LACS in a multi-threaded environment is left for future work.

B. Sensitivity to Cache Parameters

Table 4 summarizes the percentage of IPC improvement of LACS over LRU replacement for different L2 cache configurations in a uniprocessor using Group A benchmarks. The table lists the minimum, average, and maximum improvement values obtained for each configuration. The table demonstrates LACS's robustness for different cache configurations as LACS achieves considerable speedups in all configurations. The maximum improvement is achieved for the benchmark gcc in all configurations.

Table 4: Performance improvement for different cache conf.

Configuration	Minimum	Average	Maximum
256 KB, 8-way	0%	3%	9%
512 KB, 8-way	0%	15%	85%
1 MB, 8-way	-3%	8%	47%
2 MB, 8-way	-3%	19%	195%
512 KB, 4-way	0%	12%	69%
512 KB, 16-way	-1%	17%	101%

VI. CONCLUSION

We have presented a novel locality-aware cost-sensitive (LACS) cache replacement algorithm. LACS estimates a cache block's cost based on the number of instructions the processor manages to issue during the miss latency, which reflects how well the processor manages to hide this miss latency. If the number of issued instructions during the miss is large, the processor is capable of tolerating the miss and the block is considered a low-cost block. Otherwise, the block is considered a high-cost block. LACS attempts to reserve high-cost blocks in the cache at the expense of low-cost blocks with the goal of reducing the aggregate miss cost. On a cache miss, a victim is randomly chosen from among the low-cost blocks in the set. LACS uses the blocks' locality information in order to relinquish high-cost blocks that are no longer needed.

We evaluated LACS using SPEC CPU2000 benchmarks. In a uniprocessor architecture, it sped up 10 L2 cache performance-constrained benchmarks by up to 85% and 15% on average without slowing down any benchmark. In a shared L2 cache dual-core CMP architecture, it sped up 6 benchmark pairs by up to 44% and 11% on average. LACS's robust performance was also demonstrated over a wide range of cache configurations. In addition, LACS was compared against and found to outperform another state-of-the-art cost sensitive cache replacement algorithm: MLP-aware replacement using SBAR. This performance is achieved using a modest 48 KB prediction table and a simple algorithm, making LACS a very attractive cache replacement algorithm.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the constructive reviewers' comments and the insightful feedback by the committee members of Sheikh's M.S. thesis, namely, Ali Shatnawi and Gheith Abandah.

REFERENCES

[1] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*. 4th edition, Morgan Kaufmann Publishers, 2006.

[2] M. Wilkes, "The Memory Gap and the Future of High Performance Memories," *ACM Computer Architecture News*, 29(1):2–7, 2001.

[3] M. Kharbutli and Y. Solihin, "Counter-Based Cache Replacement and Bypassing Algorithms," *IEEE Transactions on Computers*, 57(4):433-447, April 2008.

[4] J. Abella, A. Gonzalez, X. Vera, and M. O'Boyle, "IATAC: A Smart Predictor to Turn-Off L2 Cache Lines," *ACM Transactions on Architecture and Code Optimization*, 2(1):55-77, 2005.

[5] Z. Hu, S. Kaxiras, and M. Martonosi, "Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior," in *Proc. 29th Annual International Symposium on Computer Architecture*, 2002.

[6] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-Block Prediction and Dead-Block Correlating Prefetchers," in *Proc. 28th Annual International Symposium on Computer Architecture*, 2001.

[7] H. Liu, M. Ferdman, J. Huh, and D. Burger, "Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency," in *Proc. 41st Annual International Symposium on Microarchitecture*, 2008.

[8] W. Wong and J.-L. Baer, "Modified LRU Policies for Improving Second-Level Cache Behavior," in *Proc. 6th Annual International Symposium on High-Performance Computer Architecture*, 2000.

[9] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power," in *Proc. 28th Annual Int. Symposium on Computer Architecture*, 2001.

[10] K. Rajan and G. Ramaswamy, "Emulating Optimal Replacement with a Shepherd Cache," in *Proc. 40th Annual International Symposium on Microarchitecture*, 2007.

[11] S. Kim, D. Chandra, and Y. Solihin, "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," in *Proc. 13th Int. Conf. on Parallel Architectures and Compilation Techniques*, 2004.

[12] S. Srinivasan and A. Lebeck, "Load Latency Tolerance in Dynamically Scheduled Processors," in *Proc. 31st Annual Int. Symposium on Microarchitecture*, 1998.

[13] Standard Performance Evaluation Corporation. *SPEC CPU2000 Benchmarks*. http://www.spec.org/osg/cpu2000.

[14] J. Jeong and M. Dubois, "Optimal Replacements in Caches with Two Miss Costs," in *Proc. 11th Annual Symposium on Parallel Algorithms and Architectures*, 1999.

[15] J. Jeong and M. Dubois, "Cost-Sensitive Cache Replacement Algorithms," in *Proc. 9th International Symposium on High-Performance Computer Architecture*, 2003.

[16] J. Jeong and M. Dubois, "Cache Replacement Algorithms with Non-uniform Miss Costs," *IEEE Transactions on Computers*, 55(4):353-365, 2006.

[17] J. Jeong, P. Stenstrom, and M. Dubois, "Simple Penalty-Sensitive Cache Replacement Policies," *Journal of Instruction-Level Parallelism*, vol. 10, June 2008.

[18] L. Belady, "A Study of Replacement Algorithms for a Virtual Storage Computer," *IBM Systems Journal*, 5(2):78-101, 1966.

[19] T. Puzak, A. Hartstein, P.G. Emma, V. Srinivasan, and A. Nadus, "Analyzing the Cost of a Cache Miss Using Pipeline Spectroscopy," *Journal of Instruction-Level Parallelism*, Vol. 10, June 2008.

[20] M. Qureshi, D. Lynch, O. Mutlu, and Y. Patt, "A Case for MLP-Aware Cache Replacement," in *Proc. 33rd Annual International Symposium on Computer Architecture*, 2006.

[21] S. Srinivasan, R. Dz-ching Ju, A. Lebeck, and C. Wilkerson, "Locality vs. Criticality," in *Proc. 28th Annual International Symposium on Computer Architecture*, 2001.

[22] L. Ceze, K. Strauss, J. Tuck, J. Renau, and J. Torrellas, "CAVA: Using Checkpoint-Assisted Value Prediction to Hide L2 Misses," *ACM Transactions on Architecture and Code Optimization*, 3(2):182-208, 2006.

[23] J. Renau. *SESC Simulator*, http://sesc.sourceforge.net.