Cost-Sensitive Cache Replacement Algorithms

Jaeheon Jeong* and Michel Dubois

*IBM
Beaverton, OR
jjeong@us.ibm.com

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA90089-2562
dubois@paris.usc.edu

Abstract

Cache replacement algorithms originally developed in the context of simple uniprocessor systems are aimed at reducing the aggregate miss count. However, in modern systems, cache misses have different costs. The cost may be latency, penalty, power consumption, bandwidth consumption, or any other ad-hoc property attached to a miss. Thus, in many practical cases, it is desirable to inject the cost of a miss into the replacement policy in addition to access locality.

In this paper, we propose several extensions of LRU which account for non-uniform miss costs. These LRU extensions have simple implementations, yet they are very effective in various situations. We evaluate our extended algorithms using trace-driven and execution-driven simulations. We first explore the simple case of two static miss costs using trace-driven simulations to understand when cost-sensitive replacements are effective. We show that very large improvements of the cost function over LRU are possible in many practical cases. Then, as an example of their effectiveness, we apply the algorithms to the second level cache of a multiprocessor with ILP processors, using the miss latency as the cost function. We show that by applying our simple replacement policies sensitive to the latency of misses we can improve the execution time of parallel applications by up to 18% using execution-driven simulations with RSIM.

1. Introduction

Cache replacement algorithms widely used in modern systems aim at reducing the aggregate miss count and thus assume that miss costs are uniform [15][16]. However, as the memory hierarchies of modern systems become more complex, and, as other factors besides performance become critical, this uniform cost assumption has lost its validity in recent years, especially in the context of multiprocessors. For instance, the cost of a miss mapping to a remote memory is generally higher in terms of latency, bandwidth consumption and power consumption than the cost of a miss mapping to a local memory in a multiprocessor system. Similarly, a non-critical load miss or a store miss are not as taxing on performance as a critical load miss in ILP processors [14]. Since average memory access latency and penalty are more directly related to execution time, we can expect better memory performance by minimizing these metrics instead of the miss count. Thus, in many cases, it is desirable to inject the actual cost of a miss into the replacement policy, and reliable algorithms to integrate cost and locality information to make better cache replacement decisions are needed. Replacement algorithms which aim to reduce the aggregate miss cost in the face of multiple miss costs have been called *cost-sensitive replacement algorithms* [6].

In this paper, we extend the LRU replacement algorithm to include cost in the replacement decision. The basic idea is to explore the option of keeping the block victimized by LRU in cache until the next reference to it, if its miss cost is greater than the miss cost of other cached blocks. We call this option a (block or blockframe) *reservation*. Since a blockframe for a block with a high-cost miss cannot be reserved forever, a mechanism must exist to relinquish the reservation after some time. This is done by depreciating the cost of the reserved block whenever a lower-cost block[1] is sacrificed in its place.

In this paper, we explore several replacement algorithms to integrate cost with locality. The first one is Greedy-Dual (GD), a well-known algorithm [20], which is cost-centric, in the sense that the cost function dominates the replacement decision. Three new algorithms, which extend LRU by block reservations and are more locality-centric, are then introduced. The first algorithm called BCL (Basic Cost-sensitive LRU) uses a crude method to depreciate the cost of reserved blocks. The second algorithm called DCL (Dynamic Cost-sensitive LRU) can detect actual situations when the reservation of a high-cost block caused other blocks to miss and thus can depreciate the cost of a reserved block with better accuracy. The third algorithm called ACL (Adaptive Cost-sensitive LRU) is an extension of DCL that can turn itself on and off across time and cache sets. We evaluate and compare these four algorithms in the simplest case, the case of two static costs, using a wide range of cost values. Then we apply these algorithms to the special case of multiprocessors systems, in which the cost function is the latency of misses. In this case the latencies can take many different values and the cost associated with each miss of a given block varies with time and must be predicted. We show, in this special but important case, that a very simple miss latency prediction scheme is successful and that, coupled with our simple latency-sensitive replacement algorithms, it can reliably improve execution times by significant amounts.

The rest of this paper is organized as follows. Section 2 covers the design issues of LRU-based cost-sensitive replacement algorithms and describes four algorithms. Section 3 explores these algorithms in the simple case of two static costs using trace-driven simulations. In Section 4 we apply our algorithms to multiprocessor systems with ILP processors to reduce the aggregate miss latency based on a simple latency prediction scheme. Section 5 evaluates the hardware complexity of the schemes. Section 6 overviews related work. Finally we conclude in Section 7.

1. Since the cost is always associated with misses, we refer to "blocks with low (high)-cost miss" simply as "low (high)-cost blocks" throughout the rest of this paper.

2. Cost-sensitive Replacement Algorithms

For a given trace of memory references resulting from an execution, $X = x_1, x_2, \cdots, x_L$, let $c(x_t)$ be the cost incurred by the memory reference with block address x_t at time t. Note that $c(x_t)$ and $c(x_{t'})$ may be different even if $x_t = x_{t'}$ because memory management schemes such as dynamic page migration can dynamically alter the memory mapping of a block at different times, or because the cost of a static memory mapping may vary with time. With no loss in generality, if x_t hits in the cache, then $c(x_t) = 0$. Otherwise, $c(x_t)$ is the cost for the miss, which can be any non-negative number. Then, the problem is to find a cache replacement algorithm such that the aggregate cost of the trace, $C(X) = \sum_{t=1}^{L} c(x_t)$, is minimized.

In general, misses may incur many different costs and these costs may be dynamic. The simplest case is the case of two static costs. In this case, low-cost misses are assigned a cost of 1 and high-cost misses are assigned a cost r. Then the *cost ratio* r is the only parameter related to miss costs throughout the execution. If x_t hits in the cache, $c(x_t) = 0$. Otherwise, $c(x_t) = 1$ or r.

Given the cost function, there are many possible approaches to integrating cost and locality. A brute-force approach is to keep a block in cache until all other blocks have equal or higher cost. This approach may result in high-cost blocks staying in cache for inordinate amounts of time, possibly forever. Clearly, a mechanism is needed to depreciate the cost of high-cost blocks. The problem is to find a good algorithm to depreciate the cost. One algorithm integrating locality and cost is GreedyDual (GD).

2.1. Greedy Dual (GD)

GreedyDual was originally developed in the context of disk paging [20] and, later on, was adapted to web caching [2]. In the web caching version, its goal is to reduce the aggregate latency where size and location of data can vary widely. GD can be easily adapted to processor caches, although it has not been promoted as such. In GD, each block in cache is associated with its miss cost. GD replaces the block with the least cost, regardless of its locality. However, when a block is victimized, the costs of all blocks remaining in the set are reduced by its cost. Whenever a block is accessed, its original cost is restored. Thus the only effect of locality on GD is that high-cost MRU blocks are protected from replacement and that their lifetime in cache is raised. GD is a cost-centric algorithm and works well when cost differentials are wide. GD is theoretically optimum in the sense that its cost cannot be more than s times the optimum cost, where s is the cache set size [20]. Unfortunately, it does not work well when the cost differentials between blocks are small, as is the case, for example, for memory latencies in modern multiprocessors.

Our goal is to explore replacement algorithms that are more locality-centric, that is, algorithms that give priority to locality over cost. Whereas we may lose some gains, we expect that locality-centric algorithms will exploit a wider range of cost differences, including small cost differences. Since LRU --or an approximation to it-- is adopted in modern systems, our algorithms rely on the locality estimate of cached blocks predicted by LRU.

2.2. LRU-based Cost-sensitive Replacement Algorithms

To reduce the aggregate miss cost, cost-sensitive replacement algorithms must consider both the access locality and the miss cost of every cached block. The position of a block in the LRU stack [11] represents an estimate of its relative access locality among all blocks cached in the same set. Cost-sensitive replacement algorithms must therefore replace the LRU block if its next miss cost is no greater than the next miss cost of any other block in the same set. However, if the next miss cost of the LRU block is greater than the next miss cost of one of the non-LRU blocks in the same set, we may save some cost by keeping the LRU block until the next reference to it while replacing non-LRU blocks with lower miss costs. While we keep a high-cost block in the LRU position, we say that the block or blockframe is *reserved*. We do not limit reservations to the LRU block. While the blockframe for the LRU block is reserved in the cache, more reservations for other blocks in other locality positions are possible except for the MRU (Most Recently Used) block, which is not subject to reservation.

If a reserved block is never accessed again nor invalidated after a long period of time, a blockframe reservation may become counterproductive, and the resulting cost savings may become negative. A compromise must be struck between pursuing as many reservations as possible, while avoiding fruitless pursuits of misguided reservations. In the design of LRU-based cost-sensitive algorithms, the following questions must be answered: (i) when to invoke a blockframe reservation, (ii) which low-cost block among multiple low-cost blocks to victimize in the presence of reserved blockframes for high-cost blocks, (iii) when and how to terminate fruitless blockframe reservations to avoid negative cost savings, and (iv) how to handle multiple blockframe reservations.

The replacement algorithms we propose in this paper are locality-centric. The locality property for a block, as predicted by LRU, plays a dominant role in the replacement decision. Let $c(i)$ be the miss cost of the block which occupies the i-th position from the top of the LRU stack in a set of size s. Thus, $c(1)$ is the miss cost of the MRU block, and $c(s)$ is the miss cost of the LRU block in an s-way associative cache. Whenever a reservation is active, we selects the first block in the LRU stack order whose cost is lower than the cost of the reserved block. Thus there might be lower cost blocks to victimize in the set, but their locality is higher. We terminate reservations by depreciating the miss costs of the reserved blocks. Ideally we should depreciate the cost of the reserved LRU block whenever another block in the set is victimized in its place and is accessed again before the next access to the reserved LRU block. However, detecting this condition exactly is not easy in general. The BCL and DCL algorithms try to approximate this condition.

2.3. Basic Cost-sensitive LRU Algorithm (BCL)

In our basic cost-sensitive LRU algorithm (BCL), we decrease the cost of a reserved LRU block whenever a block is replaced in its place, regardless of whether the replaced block is referenced again.

BCL handles multiple reservations in a similar way. While a primary reservation for the LRU block is in progress, a secondary reservation can be invoked, if $c(s) \le c(s\text{-}1)$ and there exists a block $i < s\text{-}1$ whose cost is lower than $c(s\text{-}1)$. More reservations are possible at following positions in the LRU stack while the primary and the secondary reservations are in progress. The maximum number of blocks that can be reserved is s-1. The MRU block can never sat-

isfy the condition for reservation as there is no block in the set with lower locality. When multiple reservations are active, BCL only depreciates the cost of the reserved LRU block.

Figure 1 shows the BCL algorithm in an *s*-way set-associative cache. Each blockframe is associated with a miss cost *c*(*i*) which is loaded at the time of miss. The block with *i* = 1 refers to the MRU position. As blocks change their position in the LRU stack, their associated miss costs follow them. The blockframe in the LRU position has one extra field called *Acost*. Whenever a block takes the LRU position, *Acost* is loaded with *c*(*s*), which is the miss cost of the new LRU block. Later *Acost* is depreciated upon reservations by the algorithm. To select a victim, BCL searches for the block position *i* in the LRU stack such that *c*(*i*) < *Acost* and *i* is closest to the LRU position. If BCL finds one, BCL reserves the LRU block in the cache by replacing the block in the *i*-th position. Otherwise, the LRU block is replaced. In order to depreciate the miss cost of the reserved LRU block, *Acost* is reduced by twice the amount of the miss cost of the block being replaced. Using twice the cost instead of once the cost is safer because it accelerates the depreciation of the high cost. It is a way to hedge against the bet that the high-cost block in the LRU position will be referenced again [7]. When *Acost* reaches zero the reserved LRU block becomes the prime replacement candidate for the next replacement.

The algorithm in Figure 1 is extremely simple. Yet, in all its simplicity, it incorporates the cost depreciation of reserved blocks and the handling of one or multiple simultaneous reservations as dictated by BCL.

```
find_victim()
   for (i = s-1 to 1)// from second-LRU toward MRU
      if (c[i] < Acost)
         Acost ← Acost - c[i]*2
         return i
   return LRU

upon_entering_LRU_position()
   Acost ← c[s]    // assign the cost of new LRU block
```

Figure 1. Algorithm for BCL

2.4. Dynamic Cost-sensitive LRU Algorithm (DCL)

The beauty of BCL is its simplicity. However it has some drawbacks. In BCL, the cost of a reserved LRU block is depreciated whenever a valid non-LRU block is victimized in its place. This approach is based on the pessimistic assumption that the replaced non-LRU blocks will always be accessed before the reserved LRU block so that their misses will accrue to the total cost. However, this assumption can be quite wrong. If the victimized non-LRU block is not referenced again, then the cost of replacing it instead of the reserved block is zero. In this case the replacement algorithm made a correct choice and thus it should not be handicapped by depreciating the cost of the reserved block. By depreciating its cost too rapidly BCL will evict the reserved LRU block earlier than it should. Thus BCL may squander cost savings opportunities by being too conservative. This is especially the case when the cost differential between blocks is relatively small.

The dynamic cost-sensitive LRU algorithm (DCL) is slightly more complex, but it overcomes the shortcomings of BCL. In DCL, the cost of the reserved LRU block is depreciated only when the non-LRU blocks victimized in its place are actually accessed before the LRU block. To do this, DCL keeps a record for each replaced non-LRU blocks in a directory called the *Extended Tag Directory* (ETD) similar to the shadow directory proposed in [17]. On a hit in ETD, the cost of the reserved LRU block is depreciated. For an *s*-way associative cache, we only need to keep ETD records for the *s*-1 most recently replaced blocks in each set because accesses to blocks that were replaced before these *s*-1 most recently replaced blocks would miss in the cache if the replacement was LRU. To show this, consider the *s* most recently replaced blocks since a reservation started and assume that they are not in cache. This means that they have not been referenced since they were replaced and that *s* misses have occurred since the oldest block was replaced. If the replacement was pure LRU just before the oldest block was replaced, then it would be impossible for the oldest block to still be in cache currently since *s* misses for different blocks have occurred since then and there are only *s* blockframes per set.

Thus, *s*-1 ETD entries are attached to each set in an *s*-way set-associative cache. Each ETD entry consists of the tag of the block, its miss cost and a valid bit. Initially, all entries in ETD are invalid. When a non-LRU block is replaced instead of the LRU block, an ETD entry is allocated, the tag and the miss cost of the replaced block are stored in the entry, and its valid bit is set. To allocate an entry in ETD, LRU replacement policy is used, but invalid entries are allocated first. ETD is checked upon every cache access. The tags in ETD and in the regular cache directory are mutually exclusive. If an access misses in the cache but hits in ETD, then the cost of the reserved LRU block in the cache is reduced accordingly (as in BCL) and the matching ETD entry is invalidated. If an access hits on the LRU block in the cache, then all ETD entries are invalidated. Of course, when an invalidation is received for a block present in the ETD (as may happen in multiprocessors, for example), the ETD entry is invalidated.

One may worry about the size of ETD. However, the miss cost field size can be equal to the logarithm base 2 of the number of different costs. Moreover, we do not need to store the entire tag, just a few bits of the tag. Of course this creates aliasing, but this aliasing only affects performance, not correctness. We will explore the performance effects of tag aliasing in ETD in Section 4 and will examine the hardware complexity of DCL more closely in Section 5.

2.5. Adaptive Cost-sensitive LRU Algorithm

Both BCL and DCL pursue reservations of LRU blocks greedily, whenever a high-cost block is in a low locality position. Although reservations in these algorithms are terminated rather quickly if they do not bear fruit, the wasted cost of these attempted reservations accrue to the final cost of the algorithm. If these aborted reservation trials can be filtered out by focusing on reservations with high chances of cost savings, the performance of the replacement algorithm can be further improved and made more reliable.

We have observed that, in some applications, the success of reservations varies greatly with time and also from set to set. Reservations yielding cost savings are often clustered in time, and reservations often go through long streaks of failure. We also observed that this pattern of alternating streaks of success and fail-

ure varies significantly among cache sets.

The adaptive cost-sensitive LRU algorithm (ACL) derived from DCL implements an adaptive reservation activation scheme exploiting the history of cost savings in each set. To take advantage of the clustering in time of reservation successes and failures, we associate a counter in each cache set to enable and disable reservations. Figure 2 shows the automaton implemented in each set using a two-bit counter. The counter increments or decrements whenever a reservation succeeds or fails, respectively. When the counter value is greater than zero, reservations are enabled. Initially the counter is set to zero, disabling all reservations.

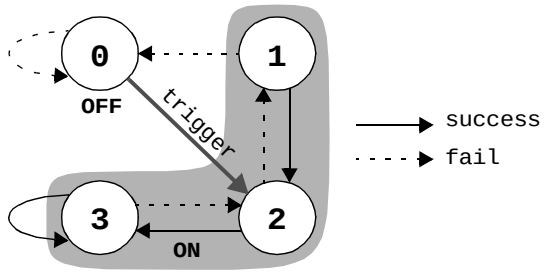


Figure 2. ACL automaton in each set

To trigger reservations from the disabled state, we use a simple scheme that utilizes the ETD differently while reservations are disabled. When reservations are disabled, an LRU block enters the ETD upon replacements if another block in the set has lower cost. ETD is checked upon every cache access, as before. An access hit in the ETD strongly indicates that we might have saved some amount of cost if the block had been reserved in the cache. Thus upon a hit in ETD, all ETD entries are invalidated, and reservations are enabled by setting the counter value to two, with the hope that a streak of reservation successes has just started.

Now that we have introduced the replacement algorithms, we evaluate their effectiveness, first by adopting a simple model with two static costs, and second by evaluating their effect in a complex situation when costs are multiple, variable, dynamic, and must be predicted.

3. Static Case with Two Costs

In general there are multiple costs associated with blocks, and the costs are dynamic and vary with time. This makes it very complex to analyze and visualize the effects of various parameters on the behavior of a particular algorithm. In this section we present the set of experiments that we have run to lead us to the replacement algorithms of Section 2. These experiments are based on a system with two static costs, the simplest case possible. Much can be learned from this simple case.

The baseline architecture is a CC-NUMA multiprocessor system in which cache coherence is maintained by invalidations [4]. The memory hierarchy in each processor node is made of a direct-mapped L1 cache, an L2 cache to which we apply cost-sensitive replacement algorithms, and a share of the distributed memory. The cost of a block is purely determined by the physical address mapping in main memory and this mapping does not change during the entire execution. Based on the address of the block in memory, misses to the block are assigned a low cost of 1 or a high cost of *r*. In a first set of experiments we assign costs to blocks randomly, based on each block address. This approach gives us maximum flexibility since the fraction of high-cost blocks can be modified at will. In practical situations however, costs are not assigned randomly and so costs are not distributed uniformly among sets and in time. To evaluate this effect we then ran a set of experiment in which blocks are allocated in memory according to the first touch policy and we assigned a low cost of 1 to locally mapped blocks and a high-cost of *r* to remotely mapped blocks.

3.1. Methodology

Traces are gathered from an execution-driven simulation assuming an ideal memory system with no cache. We pick one trace among the traces of all processors [3]. The trace of one selected slave process is gathered in the parallel section of the benchmarks. To correctly account for cache invalidations, all writes are included in the trace. Thus our traces contain all the shared data accesses of one processor plus all the shared data writes from other processors. The trace excludes private data and instruction accesses because the number of private data accesses are either very small or are concentrated on a few cache sets yielding an extremely low miss rate, and because our study focuses on data caches.

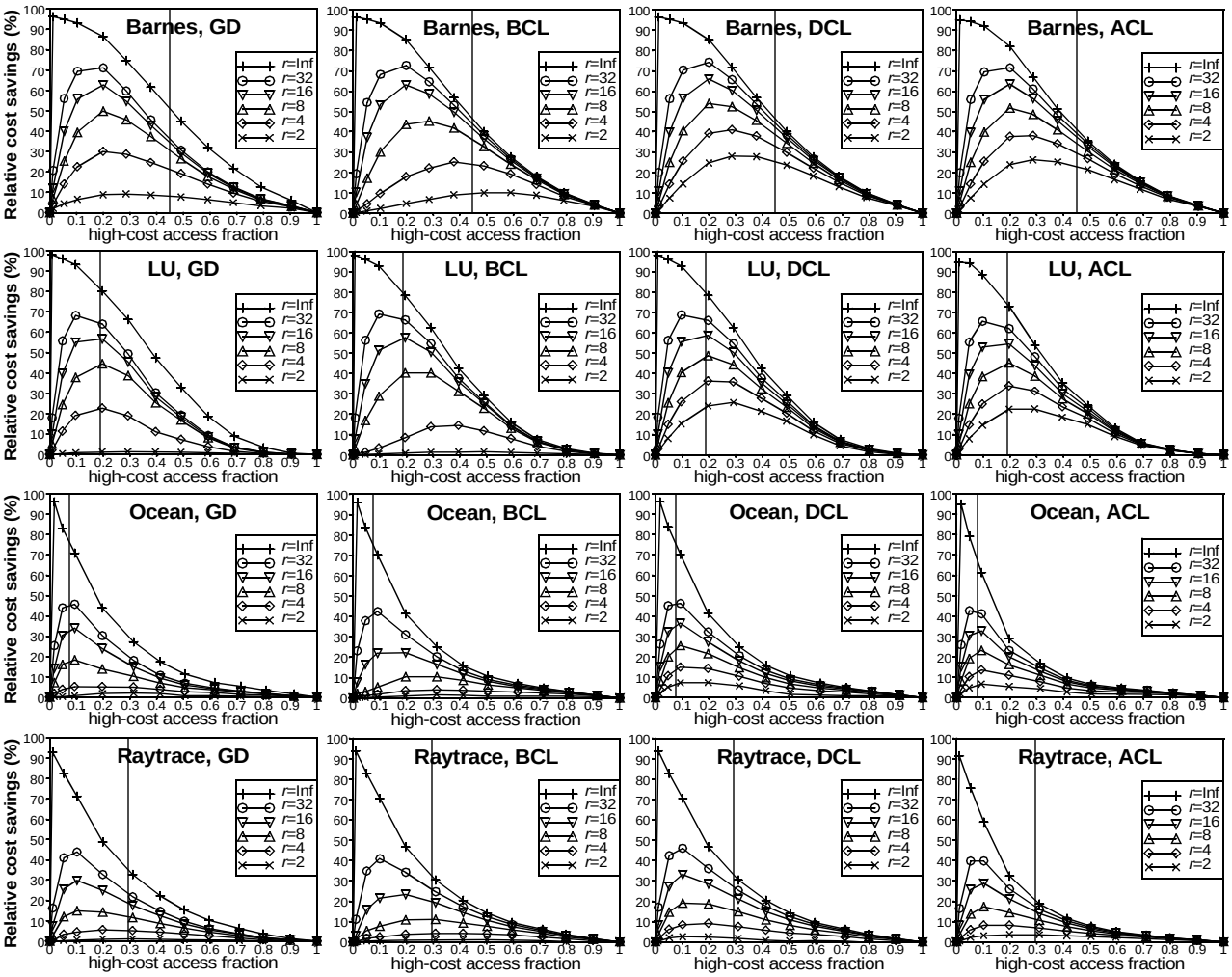
Four benchmarks from the SPLASH-2 suite [19] are used and their main features are listed in Table 1. They were picked because of the variety in their behavior[2]. They are compiled for a SPARC V7 architecture with optimization level O2. Table 1 also shows the fraction of remote accesses measured on one processor using a per-block first-touch placement policy. In Barnes and Raytrace, memory accesses are data-dependent and irregular. Thus the remote access fraction varies among processors and we picked the trace of a processor whose remote access fraction is most representative.

Benchmark	Problem size	# of proc.	Mem. usage (MB)	References by sample processor	Remote access fraction
Barnes	64K	8	11.3	34.2M	44.8%
LU	512 x 512	8	2.0	12.7M	19.1%
Ocean	258 x 258	16	15.0	15.6M	7.4%
Raytrace	car	8	32.0	14.0M	29.6%

Table 1. The characteristics of the benchmarks

In our evaluations, the most important parameters are the cost ratio *r*, the cache associativity *s* and the cache size. We vary *r* from 1 to 32 to cover a wide range of cost ratio. We also consider an infinite cost ratio by setting the low cost to 0 and the high cost to 1. A practical example of infinite cost ratio is bandwidth consumption in the interconnection network connecting the processors. The infinite cost ratio experiments also give us the maximum possible savings for all cost ratios above 32. We vary the cache associativities from 2 to 8, and 64-byte blocks are used throughout our evaluations.

2. Other SPLASH benchmarks including Water, MP3D, FFT and Radix were run as well but yielded no additional insight.



	High-cost Access Fraction = 0.2								High-cost Access Fraction = 0.6							
	r = 2				*r* = 32				*r* = 2				*r* = 32			
	GD	BCL	DCL	ACL	GD	BCL	DCL	ACL	GD	BCL	DCL	ACL	GD	BCL	DCL	ACL
Barnes	8.82	4.69	24.64	23.75	71.10	72.63	74.18	71.43	6.36	10.09	18.16	16.36	19.94	26.65	26.70	23.30
LU	1.17	0.88	24.02	22.36	64.00	66.37	66.07	62.02	0.72	1.20	9.78	9.09	9.24	14.48	14.49	12.38
Ocean	1.84	0.77	7.32	5.39	30.32	31.00	32.24	23.06	0.70	0.87	0.96	1.55	4.55	6.06	6.11	4.18
Raytrace	1.03	0.84	2.38	3.48	32.87	34.22	36.00	25.85	0.68	0.82	0.43	2.15	6.28	8.49	8.50	4.72

Figure 3. Relative cost savings with random cost mapping in 16KB, 4-way L2 cache (%)

To scale the cache size, we first looked at the miss rates for cache sizes from 2 Kbytes to 512 Kbytes. To avoid unrealistic situations while at the same time having enough cache replacements, we first investigated cache sizes such that the primary working sets start to fit in the cache. Overall, this occurs when the cache is 8 Kbytes in Barnes and LU. We also examined a cache such that the secondary working sets fit in the cache. Overall, the knee is at 64 Kbytes.

At the end we selected a 16Kbyte L2 cache, 4-way set-associative with 64-byte blocks. The L1 cache is 4Kbytes and direct-mapped. For Ocean and Raytrace, in which the miss rates are inversely proportional to the cache size, the same sizes are used.

3.2. Random Cost Mapping

In this section, we present the results with random cost mapping based on the block address. This approach, although not truly realistic, allows a systematic performance analysis and characterization of cost-sensitive replacement algorithms. With this approach, we can easily vary the high-cost access fraction (HAF) in a trace.

The relative cost savings of a replacement algorithm is calcu-

lated as the ratio between the cost savings afforded by the replacement algorithm as compared to LRU, and the aggregate cost of LRU. Figure 3 shows the relative cost savings gained by four cost-sensitive algorithms over LRU in our basic cache. A table is also added in Figure 3 to display precise numbers. We vary the cost ratio *r* from 2 to an infinite value and HAF from 0 to 1 with a step of 0.1. We add two more fractions at 0.01 and 0.05 to see the detailed behavior between HAF = 0 and HAF = 0.1.

In all algorithms and benchmarks, as HAF varies from 0 to 1, the relative cost savings quickly increases, consistently showing a peak between HAF = 0.1 and HAF = 0.3; then it slowly decreases after the peak as HAF reaches 1. Clearly it is easier to benefit from a cost-sensitive replacement algorithm when HAF < 0.5. The main reason for this observation is that, when HAF > 0.5, there are not enough low-cost blocks in cache to victimize.

The relative cost savings increases with *r*, as expected. With *r* infinite, the graphs show the theoretical upper-bound of cost savings. In this case, the cost-sensitive replacement algorithms systematically replace low-cost blocks instead of high-cost blocks whenever low-cost blocks exist in the cache, since the cost depreciations of reserved blocks have no effect. The relative cost savings does not increase proportionally to *r*, as *r* increases from 2 to 32. The relative savings quickly increases with small *r*, but for larger *r*, the relative savings tapers off. The cost savings increases linearly with *r* in absolute terms, but not in relative term, as the aggregate cost of LRU also increases with *r*. The savings of ACL is slightly lower than the savings of DCL in practically all situations.

Overall the results show that the relative cost savings by DCL over LRU is significant and is very consistent across all benchmarks. We observe a "sweet spot" for the relative cost savings with respect to HAF and the cost ratio.

3.3. First-Touch Cost Mapping

So far, the cost assignment to blocks has been done randomly. With the random cost assumption, low-cost and high-cost blocks are homogeneously spread in time and across cache sets. However, in a realistic situation, the assignment of cost is not random and cost assignments may be highly correlated. For example, if an application has an HAF of 0.5, we would expect some improvements of the cost function in DCL, according to the evaluations in the preceding section. However, an HAF of 0.5 over the entire execution could result from an HAF of 0 for half of the execution and of 1 for the other half. Or it could be that HAF is 0 for half of the sets and 1 for the other half. In both cases, the gains from DCL are expected to be dismal, if not negative.

Because of this correlation, the cost savings is not as impressive in actual situations as Figure 3 would let us believe. So we have explored a simple, practical situation to investigate the effect of cost correlations among blocks. In this section, we modify the policy to assign costs to blocks as follows. We allocate blocks to memory according to the first touch policy. Remote blocks are assigned a high-cost and local blocks are assigned a low cost. Table 2 shows the relative cost savings by cost-sensitive algorithms over LRU in our basic cache as *r* varies from 2 to 32. The high-cost access fraction in each benchmark is shown in the last column of Table 1.

To understand the performance of the replacement algorithms under the first-touch policy to assign costs, we compare their savings to the savings under the random cost mapping at the same HAF (corresponding to the vertical lines in Figure 3.) Overall, we observe that the differences in the cost savings achieved under the random cost mapping and the first-touch cost mapping are moderate except for LU. In LU, the savings under the first-touch policy is very poor. It even becomes negative in BCL and DCL although the high-cost fraction falls in the "sweet spot". Accesses in LU have high locality and their behavior varies significantly across cache sets. In some cache sets, no reservations succeed.

		r=2	*r*=4	*r*=8	*r*=16	*r*=32
Barnes	GD	7.99	20.62	29.14	32.31	33.94
	BCL	9.98	24.61	36.40	41.11	43.17
	DCL	25.86	33.10	38.18	41.42	43.31
	ACL	24.59	31.48	36.28	39.29	41.02
LU	GD	-0.02	0.30	0.27	0.19	0.47
	BCL	0.04	-0.03	-0.32	-0.65	-0.76
	DCL	-0.37	-0.58	-0.87	-1.19	-1.24
	ACL	0.24	0.42	0.67	0.97	1.48
Ocean	GD	-1.51	2.86	14.99	26.08	35.13
	BCL	-1.08	-0.94	0.99	12.98	35.32
	DCL	6.24	12.40	20.88	29.23	36.03
	ACL	6.21	12.43	20.65	28.49	34.81
Raytrace	GD	0.57	3.83	8.91	13.82	17.25
	BCL	0.16	2.78	7.86	14.59	20.00
	DCL	2.35	7.15	12.68	17.52	20.94
	ACL	3.11	6.67	10.80	14.53	17.17

Table 2. Relative cost savings with first-touch cost mapping(%)

This observation prompted us to explore adaptive algorithms across sets and time and to come up with the design of ACL. LU takes advantage of ACL, and even shows small positive savings in ACL. The cost savings by GD is the same as for the random cost assignments. In Barnes where HAF is high, the cost savings by GD is much lower than the cost savings by BCL. In other benchmarks, GD outperforms BCL with small *r*. Although ACL does not always reap the best cost savings, its cost savings is always very close to the best one among the four algorithms. Moreover, ACL is more reliable across the board as its cost is never worse than LRU's.

4. Improving the Memory Performance of CC-NUMA Multiprocessors

There are many possible applications for cost-sensitive replacement algorithms. One of them is improving the memory behavior of multiprocessors by injecting the performance cost of memory accesses into the replacement algorithm. In this section, we apply our cost-sensitive replacement algorithms to improve the memory performance of multiprocessor systems with non-uniform memory access latencies. We use the miss latency as a measure of miss cost. In a CC-NUMA multiprocessor, the miss latency is dynamic and depends on the type of access and the global state of the block at

			current miss																	
			read									rd-excl								
			occurrence (%)			mismatch (%)			avg. lat. error			occurrence (%)			mismatch (%)			avg. lat. error		
			U	S	E	U	S	E	U	S	E	U	S	E	U	S	E	U	S	E
last miss	read	U	22.1	1.5	0.1	0	0	83	0.0	0.0	25.5	2.2	0.1	1.9	0	59	67	0.0	27.6	70.3
		S	0.2	53.8	0.1	0	0	83	0.0	0.0	17.8	0.0	0.3	0.0	0	68	58	0.0	31.2	26.3
		E	0.0	1.2	0.2	67	100	12	19.8	21.1	28.8	0.0	0.1	0.0	42	67	10	33.3	15.6	28.0
	rd-excl	U	4.6	0.1	0.1	0	0	67	0.0	0.0	38.1	8.9	0.0	0.0	0	57	58	0.0	33.8	40.5
		S	0.2	0.0	0.1	68	70	67	27.3	43.0	17.3	0.1	0.0	0.0	44	34	58	33.1	26.0	18.3
		E	1.9	0.0	0.0	75	67	21	57.4	38.0	33.0	0.3	0.0	0.0	50	57	14	5.2	27.4	30.8

Table 3. Latency variation in protocol without replacement hints

the time of the access. When the replacement decision must be made, the latency of the future miss must be predicted.

4.1. Miss Cost Prediction in CC-NUMA Multiprocessors

To select a victim in a cost-sensitive replacement algorithm, the future miss costs of all cached blocks must be known in advance at the time of replacement. The *future miss cost* of a cached block refers to the miss cost at the next reference to it *if the block is victimized*. In general, the future miss cost is affected by the replacement decision. Such a prediction is hard or even impossible to verify, in general, unless costs are fixed and static.

One way to approach the prediction of miss latencies is to look at the correlation between consecutive unloaded miss latencies to the same block in the normal execution with LRU replacement. Table 3 is a two-dimensional matrix indexed by the attributes of the last miss and the current miss to the same block by the same processor. It yields the average absolute difference in latencies between two consecutive misses across all four SPLASH-2 benchmarks for a MESI protocol without replacement hints [8]. The attributes are the request type (*read* or *read-exclusive*) and the memory block state (*Uncached*, *Shared*, or *Exclusive*). For instance, the table shows that read misses followed by another read miss to the same block by the same processor while the block is in memory state *Shared* make up about 54% of all misses. In such cases the unloaded miss latency does not change from one miss to the next. Overall the table shows (in shaded area) that 93% of misses are such that their latencies are the same as those of the prior misses by the same processor to the same block. For the remaining 7%, the average difference between past and present latencies varies widely. However, the average latency difference is mostly small, and is much smaller than the local latency (60 cycles). Similar results are obtained in the protocol with replacement hints [10].

In the following we simply use the last measured miss latency to predict the future miss latency to the same block by the same processor. The latency is measured by augmenting each message with a timestamp. When the request is sent, the message is stamped with the current time. When the reply is returned to the requester, the latency is measured as the difference between the current time and the timestamp. In the case that a request receives many replies, the latency is measured at the time when the requested block becomes available. If a nacked request is reissued, the original timestamp is re-used.

4.2. Evaluation Approach and Setup

To measure the performance in multiprocessors with modern ILP processors, we use RSIM [12] which models processor, memory system and interconnection network in detail. We have implemented the four cost-sensitive replacement algorithms as well as LRU in the second-level caches. Table 4 summarizes our target system configuration consisting of 16 processors. Data is placed in main memory using the first-touch policy for each individual memory block. The minimum unloaded remote-to-local latency ratio to clean copies is around 3. To reflect modern processors, we consider 500MHz and 1GHz clock speeds. In our evaluations, the sequential memory consistency model is assumed.

Processor Architecture	
Clock	500 MHz or 1 GHz
Active List	64 entries
Functional Units	2 integer units, 2 FP units, 2 address generation units, 32-entry address queue
Memory Hierarchy and Interconnection Network	
L1 Cache	4 Kbytes, direct-mapped, write-back, 2 ports, 8 MSHRs, 64-byte block, 1clock access
L2 Cache	16 Kbytes, 4-way associative, write-back, 8 MSHRs, 64-byte block, 6 clocks access
Main Memory	4-way interleaved, 60 ns access time
Unloaded Minimum Latency	Local clean: 120ns, Remote clean: 380ns, Remote dirty: 480ns
Cache Coherence Protocol	MESI protocol with replacement hints
Interconnection Network	4x4 mesh, 64-bit link, 6ns flit delay, 64-flit switch buffer

Table 4. Baseline system configuration

The four benchmarks in Table 1 are used. However, due to the slow simulation speed of RSIM, the problem sizes are reduced further. We execute Barnes with 4K particles, LU with 256x256 matrix, Ocean with 130x130 ocean and Raytrace with teapot scene. The benchmarks are compiled for Sparc V8 with Sun C compiler with optimization level xO4.

4.3. Execution Times

Table 5 shows the reduction of the execution time (relative to LRU) for each of the four cost-sensitive replacement algorithms with processors clocked at 500MHz and 1GHz. GD, as compared to BCL, shows a mixed behavior. GD slightly outperforms BCL in Ocean and Raytrace whereas BCL outperforms GD in Barnes and LU. The execution times in LU by GD and BCL are slightly increased. This behavior conforms to the results by trace-driven simulations in Section 3.3. Overall BCL yields more reliable improvements than GD in both processors. However, the differences between BCL and GD are quite small, as compared to the differences between BCL and DCL/ACL.

	500MHz Processor					
	GD	**BCL**	**DCL**	**ACL**	**DCL aliasing**	**ACL aliasing**
Barnes	4.94	7.36	16.92	16.15	15.90	15.14
LU	-0.62	-0.40	3.50	3.93	4.46	5.07
Ocean	6.28	5.99	8.29	7.35	7.65	6.84
Raytrace	3.50	2.75	7.19	13.44	5.61	14.56
	1GHz Processor					
	GD	**BCL**	**DCL**	**ACL**	**DCL aliasing**	**ACL aliasing**
Barnes	6.88	8.51	18.12	17.37	18.41	17.20
LU	-0.44	-0.29	3.59	4.20	4.75	5.38
Ocean	6.45	6.18	8.46	7.94	8.00	7.12
Raytrace	3.59	2.30	7.82	7.55	6.70	5.68

Table 5. Reduction of execution time by cost-sensitive replacement algorithms over LRU (%)

DCL yields reliable and significant improvements of execution times in every situation. The improvements by DCL over BCL are large in Barnes and Raytrace whose data access patterns are rather irregular. Thus it is advantageous to utilize ETD for the accurate depreciation of the miss costs in these benchmarks.

As compared to DCL, the execution times in ACL are slightly longer except in a few cases. This indicates that ACL is rather slow in adapting to the rapid changes of the savings pattern. Thus ACL filters some chances of cost savings as well as unnecessary reservations. LU shows consistent but marginal improvements. In LU, the streak of reservation failures is extremely long in some cache sets and ACL effectively filters these unnecessary reservations. In Raytrace with 500MHz processor, the large improvement by ACL over DCL mainly comes from the reduction of synchronization overhead and load imbalance.

To reduce the size of ETD, we have the option to store a few bits of the tag instead of the whole tag, as explained in Section 2.4. The table shows the results with tag aliasing in ETD. We reduce the tag sizes to 4 bits. This tag aliasing practically saves 40% to 60% of the tag storage in ETD depending on the data address space in each benchmark. The ratios of false match upon cache misses due to the aliasing are 45%, 43%, 30% and 27% for Barnes, LU, Ocean and Raytrace, respectively. The false matches result in a more aggressive depreciation of the cost of a reserved block, which seems to benefit LU. The results show that the effect on the execution time by ETD tag aliasing is very marginal.

Overall we believe that the improvements on the execution time by DCL is significant. The performance by ACL is often slightly lower than DCL, but ACL gives more reliable performance across various applications.

5. Implementation Considerations

In this section we evaluate the hardware overhead required by the four cost-sensitive algorithms over LRU, in terms of hardware complexity and its effect on cycle time. Tag fields and cost fields are needed. There are two types of cost fields: fixed cost fields which store the fixed cost of the next miss, and computed (depreciated) cost fields which store the cost of a block while it is depreciated.

We first consider the hardware storage needed for each cache set. In an s-way associative cache, BCL requires s+1 cost fields (one fixed cost for each block in the set and one computed cost for *Acost*). GD requires 2s cost fields (one fixed cost and one computed cost for each block in the set). DCL requires 2s cost fields (s fixed costs and 1 computed cost in cache and s-1 fixed cost in ETD) and s-1 tag fields, and ACL adds a two-bit counter plus one bit field[3] to DCL. All these additional fields can be part of the directory entries which are fetched in the indexing phase of the cache access, with little access overhead.

In a four-way associative cache with 25-bit tags, 8-bit cost fields and 64-byte blocks, the added hardware costs over LRU algorithm are around 1.9%, 2.7%, 6.6% and 6.7% for BCL, GD, DCL and ACL, respectively. If the target cost function is static and associated with the address, a simple table lookup can be used to find the miss cost. In this case, the algorithms do not require the fixed cost fields and the added costs are 0.4%, 1.5%, 4.0% and 4.1%, respectively.

The hardware requirement of DCL and ACL can be further reduced if we allow aliasing for tags in ETD resulting in more aggressive cost depreciation due to false tag matches in ETD.

Even if the costs are dynamic, it is possible to cut down on the number of bits required by the fixed cost fields. For example, instead of measuring accurate latencies as we have done in Section 4, we can use the approximate, unloaded latencies given in Table 4, which can be looked up in a table. In general the number of bits required by the fixed cost fields is then equal to the logarithm base 2 of the number of costs. In the example of Table 4 we would need 2 bits for fixed miss cost fields.

On the other hand, the computed cost fields must have enough bits to represents latencies after they have been depreciated. Let's assume that the greatest common divisor (GCD) of all possible

3. This bit is associated with the LRU blockframe and indicates whether or not the block is currently reserved so that the counter of successful/failed reservations can be updated.

miss costs is G. Then G can be the unit of cost. Let's assume that the largest possible cost is KxG. Then we need $\log_2 K$ bits for the computed (depreciated) cost field. For example, from Table 4, we can use G=60nsec and K=8 (the only problem is the 380nsec latency which would be encoded as 360nsec, a minor discrepancy). Thus 3 bits will be sufficient for the computed cost fields. In this case, with 5 bits for the tags and the valid bit in each ETD entry, the hardware overhead per set over LRU is 11 bits in BCL, 20 bits in GD, 32 bits in DCL and 35 bits in ACL.

The algorithms affect the cache cycle time minimally, if any. The only operations on a hit are restoring the miss cost of the MRU block in GD, the setting of *Acost* for the LRU block in BCL, DCL and ACL, and the lookup of ETD for DCL and ACL. Given the number of bits involved, these are trivial operations. The major work is done at miss time when blocks are victimized and the amount of work is very marginal, compared to the complexity of operation of a lockup-free cache.

6. Related Work

Replacement algorithms to minimize the miss count in finite-size storage systems have been extensively studied in the past. A variety of replacement algorithms have been proposed and a few of them, such as LRU or one of its approximation with lower implementation overhead, are widely adopted in caches [15][16]. Lately several cache replacement algorithms (e.g., [9][13][18]) have been proposed to further reduce the miss count in LRU. These proposals are motivated by the performance gap between LRU and OPT [11], and often require large amounts of hardware to keep track of long access history.

Recently, the problem of replacement algorithms has been revisited in the context of emerging applications and systems with variable miss costs. Albers et al. [1] classified general caching problems into four models in which the size and/or the latency of data can vary. They proposed several approximate solutions to these problems. In the context of disk paging, Young proposed GreedyDual [20]. GreedyDual was later refined by Cao and Irani [2] in the context of web caching to reduce the miss cost. They found that size considerations play a more important role than locality in reducing miss cost. However when applied to processor caches with small, constant data transfer sizes, our results show that GreedyDual is far less cost-efficient than other, more locality-centric algorithms, especially when the cost ratio is small.

In the context of processor caches with multiple miss costs, Jeong and Dubois [6] proposed an off-line replacement algorithm that finds an optimal replacement schedule with the optimal aggregate miss cost from a memory reference trace. They introduced the concept of reservation and found that the victim selection cannot be made at the time of replacement even with full knowledge of the future. Our algorithms exploit the same idea of high-cost block reservation. We have introduced in this paper a heuristic based on cost depreciation of reserved blocks, a key idea which makes the algorithm implementable on-line, in real systems. Moreover we have fine-tuned several simple on-line algorithms and have demonstrated their performance through trace-driven and execution-driven simulations.

The idea behind the shadow directory [17] is to gather extended locality information for blocks already replaced from cache. This information is then used for smart prefetching or replacement decisions. In our algorithms, the extended tag directory used to depreciate the cost of reserved blocks is similar to the shadow directory.

Srinivasan et al. [14] addressed the performance issues caused by critical loads in ILP processors. Critical loads are loads that have a large miss penalty in ILP processors. They proposed schemes to identify blocks accessed by critical loads. Once detected such critical blocks are stored in a special cache upon cache replacement or their stay in caches is extended. They found that the modification of the replacement policy to extend the lifetime in cache of critical blocks does not help much due to the large working set of critical blocks. Whether the algorithms proposed in this paper would fare better in the context of that study is unclear. In this paper, our focus has been on multiprocessor systems.

Finally, we must acknowledge the work of Karlin et al. [7] who introduced competitive snooping algorithms to optimize the snooping overhead in multiprocessors. Cached blocks which incur snooping overhead due to accesses from remote processors are removed from cache based on a dynamic cost adjustment similar to our cost depreciation scheme.

7. Conclusion

In this paper we have introduced new on-line cost-sensitive cache replacement algorithms extended from LRU whose goal is to minimize the aggregate miss cost rather than the aggregate miss count. The algorithms integrate locality and cost and are based on two key ideas, blockframe reservation and cost depreciation. These algorithms have been thoroughly evaluated in this paper.

From trace-driven simulations we observe that our cost-sensitive algorithms yield large cost savings over LRU across various cost ranges and cache configurations. Execution-driven simulations of a multiprocessor with ILP processors show significant reduction of execution time when cache replacements vie to minimize miss latency instead of miss cost.

The major strength of our algorithms comes from their simplicity and careful designs. Their hardware cost is very marginal and their effect on cache access time is negligible. The tight integration of the cache hierarchy inside modern processor chips facilitates further the implementation of our algorithms.

We believe that the application domain of our algorithms is very broad. They are readily applicable to the management of various kinds of storage where various kinds of non-uniform cost functions are involved. Moreover, in contrast to the approaches that divide caches into several regions or add special-purpose buffers to treat blocks in different ways [14][5], cost-sensitive replacement algorithms with properly defined cost functions can maximize cache utilization, which is always a problem in schemes relying on cache partitioning.

There are many open questions left to research. In the arena of multiprocessor memory systems, we can imagine more dynamic situations than the ones evaluated here. First the memory mapping of blocks may vary with time, adapting dynamically to the reference patterns of processes in the application, such as is the case in page migration and COMAs [4]. Second, we could imagine that node bottlenecks and hot spots in multiprocessors could be adaptively avoided by dynamically assigning very high costs to blocks accessible in congested nodes. Other areas of application are power optimization in embedded systems or bus bandwidth optimization in bus-based systems. The memory performance of CC-NUMA

multiprocessors may be further enhanced if we can measure memory access penalty instead of latency and use the penalty as the target cost function.

Single ILP processor systems may also benefit from cost-sensitive replacements. It is well-known that stores can be easily buffered whereas loads are more critical to performance. Even among loads, some loads are more critical than others [14]. Thus if we could predict the nature of the next access to a cached block, we could assign a high cost to critical load misses and low cost to store misses and non-critical load misses, based on a measure of their penalty. Of course, the combination of ILP processors and multiprocessor environment provides richer optimization opportunities for cost-sensitive replacements.

Finally, although our evaluations have focused on second-level caches, latency and penalty sensitive algorithms may be useful at every level of the memory hierarchy, including the first-level cache, both in multiprocessor and uniprocessors. The general approach of pursuing high-cost block reservation and of depreciating their cost to take care of locality effects could also be applied to other replacement algorithms besides LRU.

Acknowledgments

Michel Dubois and Jaeheon Jeong were funded by NSF Grant No.MIP-9223812, NSF Grant CCR-0105761, and by an IBM Faculty Partnership award.

8. References

[1] S. Albers, S. Arora and S. Khanna, "Page Replacement for General Caching Problems," In *Proceedings of Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1999.

[2] P. Cao and S. Irani, "Cost-Aware WWW Proxy Caching Algorithms," In *Proceedings of the 1997 USENIX Symposium on Internet Technology and Systems*, pp. 193-206, December 1997.

[3] J. Chame and M. Dubois, "Cache Inclusion and Processor Sampling in Multiprocessor Simulations," In *Proceedings of ACM Sigmetrics*, pp. 36-47, May 1993.

[4] D. Culler, J. P. Singh and A. Gupta, "Parallel Computer Architecture," *Morgan Kaufmann Publishers Inc.*, 1999.

[5] A. Gonzalez, C. Aliagas and M. Valero, "A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality," In *Proceedings of ACM International Conference on Supercomputing*, July 1995.

[6] J. Jeong and M. Dubois, "Optimal Replacements in Caches with Two Miss Costs," In *Proceedings of 11th ACM Symposium on Parallel Algorithms and Architectures*, pp. 155-164, June 1999.

[7] A. Karlin, M. Manasse, L. Rudolph and D. Sleator, "Competitive Snoopy Caching," In *Proceedings of 27th Annual IEEE Symposium on Foundations of Computer Science*, 1986.

[8] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server," In *Proceeding of 24th International Symposium on Computer Architecture*, pp. 241-251, June 1997.

[9] D. Lee et al., "On the Existence of a Spectrum of Policies that Subsumes the LRU and LFU policies," In *Proceedings of the 1999 ACM Sigmetrics Conference*, pp. 134-143, May 1999.

[10] D. Lenoski et al., "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor," In *Proceeding of 17th International Symposium on Computer Architecture*, pp. 148-159, May 1990.

[11] R. L. Mattson, J. Gecsei, D. R. Slutz and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Systems Journal*, vol. 9, pp. 77- 117, 1970.

[12] V. Pai, P. Ranganathan and S. Adve, "RSIM Reference Manual," *Technical Report* 9705, Department of Electrical and Computer Engineering, Rice University, August 1997.

[13] V. Phalke and B. Gopinath, "Compression-Based Program Characterization for Improving Cache Memory Performance," *IEEE Transactions on Computers*, v. 46, no. 11, pp. 1174-1186, November 1997.

[14] S.T. Srivivasan, R.D. Ju, A.R.Lebeck, and C. Wilkerson, "Locality vs. Criticality," In *Proceedings of the 28th International Symposium on Computer Architecture*, pp. 132-143, July 2001.

[15] A. J. Smith, "Cache Memories," *ACM Computing Surveys*, vol. 3, pp. 473-530, September 1982.

[16] K. So and R. Rechtschaffen, "Cache Operations by MRU Change," *IEEE Transactions on Computers*, v. 37, no. 6, pp. 700-709, June 1988.

[17] H. Stone, "High-Performance Computer Architectures," 3rd Edition, *Addison-Wesley Publishing Company*, 1993.

[18] W. Wong and J. Baer, "Modified LRU Policies for Improving Second-Level Cache Behavior," In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, pp. 49-60, January 2000.

[19] S. Woo et al., "The SPLASH-2 Programs: Characterization and Methodological Considerations," In *Proceedings of 22nd International Symposium on Computer Architecture*, pp. 24-36, June 1995.

[20] N. Young, "The k-server Dual and Loose Competitiveness for Paging," *Algorithmica*, vol. 11, no. 6, pp. 525-541, June 1994.