Cost-Sensitive Cache Replacement Algorithms

Jaeheon Jeong and Michel Dubois

IBM
Research Triangle Park, NC 27709
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 aim to reduce the 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 numerical property attached to a miss. In many practical situations, it is desirable to inject the cost of a miss into the replacement policy.

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 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 are possible in many practical cases.

As an example of their effectiveness, we apply the algorithms to the second-level cache of a multiprocessor with superscalar processors, using the miss latency as the cost function. By applying our simple replacement policies sensitive to the latency of misses we can improve the execution time of some parallel applications by up to 18%.

1. Introduction

Cache replacement algorithms widely used in modern systems aim to reduce the aggregate miss count and thus assume that miss costs are uniform [19][20]. 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, 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 [21] or a store miss are not as taxing on performance as a critical load miss in superscalar processors [18]. 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 situations, it is desirable to inject the actual cost of a miss into the replacement policy. Replacement algorithms which aim to reduce the aggregate miss cost in the face of multiple miss costs have been called *cost-sensitive replacement algorithms* [8].

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*. (The idea of block reservation is inspired from an efficient algorithm we developed earlier to implement optimum cost-sensitive replacement algorithms [8].) To release the reservation we depreciate the cost of the reserved block over time, according to various algorithms.

We first consider GreedyDual (GD), a well-known algorithm for Web caching [3][25]. Then, we introduce three new algorithms which extend LRU by block reservations. 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), depreciates the cost of a reserved block with better accuracy. The third algorithm, called ACL (Adaptive Cost-sensitive LRU), is an adaptive extension of DCL which dynamically switches back to LRU in each cache set whenever its cost may become worse than LRU.

We evaluate and compare these four algorithms in two sets of experiments. The first set is a set of controlled experiments to isolate the effect of the replacement policy on cost and to facilitate the comparison between policies. In these experiments, we use trace-driven simulations and evaluate the aggregate cost of the four replacement policies in the simple case of two static costs. In this framework, we can vary the costs and the distribution of the cost

among blocks at will.

In the second set of experiments, we apply the four replacement algorithms to multiprocessors systems with superscalar processors, in which the cost function is the latency of misses. In this case there are many different costs and the cost associated with each miss of a given block varies with time and must be predicted. These experiments demonstrate that our simple latency-sensitive replacement algorithms can reliably improve execution times by significant amounts in a realistic situation.

The rest of this paper is organized as follows. Section 2 describes the four replacement 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 superscalar processors. Section 5 evaluates the hardware complexity of the schemes. Section 6 overviews related work. Finally we conclude in Section 7.

2. Cost-sensitive Replacement Algorithms

2.1. Locality

According to Belady's MIN algorithm [2][13], the block with the largest *forward distance* in the string of future references made by the processor should be replaced at the time of a miss. Practical replacement algorithms replace the block with the largest *estimated* forward distance. In this paper, we use the term *locality* to mean an estimate of the probability that a block will be referenced in the near future. Typically, a cache replacement algorithm ranks the blocks in a cache set according to their locality and replaces the block with lowest locality at the time of a miss. Replacement algorithms based only on locality try to minimize the number of misses.

In the Least Recently Used (LRU) algorithm, blocks in each cache set are ranked according to the recency of the last access to them. The Least Recently Used (LRU) block has lowest locality and is always selected for replacement.

2.2. Cost

Belady's MIN algorithm minimizes the number of misses. However, it has been recently argued [8] that the total cost of misses --not just the number of misses-- should be the target of an optimum replacement policy.

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.

The simplest cost function maps each block statically to one of two costs. 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.

2.3. Integrating Locality and Cost

Given a method for measuring locality and a cost function, there are many possible approaches to integrate both. 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. Thus a mechanism is needed to depreciate the cost of high-cost blocks.

2.3.1. GreedyDual (GD)

GreedyDual was originally developed in the context of disk paging [25] and, later on, was adapted to Web caching [3]. GD can be easily adapted to processor caches. In GD, each block in a cache set is associated with its miss cost. When a block is replaced, the costs of all blocks remaining in the set are reduced by the current cost of the victim. Whenever a block is accessed, its original cost is restored. The block with lowest cost is always replaced. Unfortunately, GD does not work well when the cost differentials between blocks are small, because it mostly uses cost information to make decisions.

2.3.2. LRU-based Cost-sensitive Replacement Algorithms

To reduce the aggregate miss cost effectively, cost-sensitive replacement algorithms must factor in locality.

In the context of LRU, we must replace the LRU block if the cost of the next miss to it is no greater than the cost of the next miss to any other block in the same set. Otherwise, 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 never reserved, since there is no block in the set with better locality.

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.

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 replace in the set, but their locality is higher. We terminate reservations by depreciating the miss costs of reserved blocks.

2.3.3. Basic Cost-sensitive LRU Algorithm (BCL)

In the basic cost-sensitive LRU algorithm (BCL), we depreciate the cost of a reserved LRU block whenever a (higher locality) block is replaced in its place. Multiple concurrent reservations are handled in a similar way. While a primary reservation for the LRU block is in progress, a secondary reservation can be invoked, if $c[s] \leq c[s\text{-}1]$ 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. The maximum number of blocks that can be reserved is s-1. When multiple reservations are active, BCL only depreciates the cost of the reserved block in the LRU position.

```
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. BCL algorithm

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. 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.

We depreciate *Acost* by twice the amount of the miss cost of the replaced block, thus interrupting fruitless reservations faster [9]. When *Acost* reaches zero the reserved LRU block becomes the prime candidate for 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 concurrent reservations as dictated by BCL.

2.3.4. Dynamic Cost-sensitive LRU Algorithm (DCL)

BCL's weakness is that it assumes that LRU provides a perfect estimate of relative forward distances. To correct for this weakness, the cost of a reserved LRU block in DCL is depreciated only when the non-LRU blocks victimized in its place are actually accessed before the reserved LRU block is. To do this, DCL records every replaced non-LRU blocks in a directory called the *Extended Tag Directory* (ETD) similar to the shadow directory [22]. 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.

Thus, s-1 ETD entries are attached to each set. Each ETD entry consists of the tag of the block, its miss cost and a valid bit. Initially, all ETD entries 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. Entries in ETD are replaced according to LRU. ETD is checked in parallel with each cache access. If an access misses in the cache but hits in ETD, then the cost of the reserved LRU block in the cache is reduced as in BCL and the matching ETD entry is invalidated. When an access hits on the LRU block, all ETD entries are invalidated.

2.3.5. Illustration

To illustrate BCL and DCL, consider a cache of size $s = 3$ and a trace $X = x_1, x_2, \cdots, x_9$, as shown in Figure 2. The block addresses are A, B, C and D. For simplicity, we consider the case of two static miss costs, with $c(C) = 4$, and $c(A) = c(B) = c(D) = 1$. The first two rows show the reference string from the trace. Then we show the cache contents in their stack order for LRU, BCL and DCL along with $c(x_t)$, *Acost* and ETD. The cache contents, *Acost* and ETD are given just before reference x_t is made at time t.

Right before block D is accessed at $t = 1$, the cache

contains blocks *A*, *B* and *C*. At *t* = 1, block *D* misses in all algorithms. BCL reserves high-cost block *C* and replaces block *B* instead; *Acost* is depreciated by 2. At *t* = 2, BCL continues to reserve block *C* by replacing block *A* since *Acost* is still greater than *c*(*A*); *Acost* is adjusted to 0. At *t* = 3, BCL hits on block *C* whereas LRU misses on block *C*.

Time (*t*)		1	2	3	4	5	6	7	8	9	
x_t		*D*	*B*	***C***	*B*	*D*	*A*	*B*	*D*	***C***	
LRU	LRU stack	*A*	*D*	*B*	***C***	*B*	*D*	*A*	*B*	*D*	***C***
		B	*A*	*D*	*B*	***C***	*B*	*D*	*A*	*B*	*D*
		C	*B*	*A*	*D*	*D*	***C***	*B*	*D*	*A*	*B*
	$c(x_t)$	1	0	4	0	0	1	0	0	4	
BCL	BCL stack	*A*	*D*	*B*	***C***	*B*	*D*	*A*	*B*	*D*	***C***
		B	*A*	*D*	*B*	***C***	*B*	*D*	*A*	*B*	*D*
		C	***C***	***C***	*D*	*D*	***C***	***C***	***C***	*A*	*B*
	$c(x_t)$	1	1	0	0	0	1	1	1	4	
	Acost	4	2	0	1	1	4	2	0	1	1
DCL	DCL stack	*A*	*D*	*B*	***C***	*B*	*D*	*A*	*B*	*D*	***C***
		B	*A*	*D*	*B*	***C***	*B*	*D*	*A*	*B*	*D*
		C	***C***	***C***	*D*	*D*	***C***	***C***	***C***	***C***	*B*
	$c(x_t)$	1	1	0	0	0	1	1	1	0	
	Acost	4	4	2	1	1	4	4	2	0	1
	ETD	-	*B*	*A*	-	-	-	*B*	*D*	*A*	-
		-	-	-	-	-	-	-	-		-

Figure 2. Illustration of LRU, BCL and DCL

The difference between BCL and DCL is apparent from *t* = 6 to *t* = 9. At *t* = 6, both algorithms reserve block *C* and replace block *B*. BCL depreciates *Acost* by 2 whereas ACL puts block *B* in ETD leaving *Acost* at 4. At *t* = 7, both algorithms continue to reserve block *C* by replacing block *D*. *Acost* in BCL is set to 0 while *Acost* in DCL is reduced by 2 since block *B* hits in ETD. At *t* = 8, BCL replaces block *C* since *Acost* is 0 whereas DCL continues to reserve block *C*. At *t* = 9, DCL finally hits on block *D*. Overall, the total costs by LRU, BCL and DCL are 10, 9 and 5, respectively.

2.3.6. 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 quickly if they do not bear fruit, the wasted cost of these attempted reservations accrues to the final cost of the algorithm.

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. These observations form the rationale for ACL.

The adaptive cost-sensitive LRU algorithm (ACL) 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 3 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.

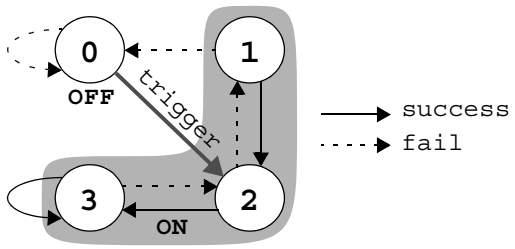


Figure 3. ACL automaton in each set

To trigger reservations from the disabled state, we use a simple scheme that utilizes the ETD differently. When reservations are disabled, an LRU block enters the ETD upon replacement if another block in the set has lower cost. 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.

3. Static Case with Two Costs

In this section we present the results from trace-driven simulation experiments that led us to the replacement algorithms of Section 2.

The baseline architecture is a CC-NUMA multiprocessor system in which cache coherence is maintained by invalidations [5]. 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 have run a set of experiment in which blocks are allocated in memory according to the *first-touch policy*, i.e., a block is allocated to the memory of the processor that first accesses it.

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 [4]. The trace of the selected processor is gathered in the parallel section of the benchmarks. To correctly account for cache invalidations, writes from all processors are included in the trace. The trace excludes private data and instruction accesses.

The main features of four benchmarks selected from the SPLASH-2 suite [24] are in Table 1[1]. Table 1 also shows the fraction of remote accesses measured in one processor using a per-block first-touch placement policy.

Table 1. Characteristics of the benchmarks

Bench-mark	Problem size	Mem. usage (MB)	Reference count	Remote access fraction (first touch)
Barnes	64K	11.3	34.2M	44.8%
LU	512 x 512	2.0	12.7M	19.1%
Ocean	258 x 258	15.0	15.6M	7.4%
Raytrace	car	32.0	14.0M	29.6%

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 ratios. We also consider an infinite cost ratio by setting the low cost to 0 and the high cost to 1. In the case of *r* infinite, the replacements of low-cost blocks are free, and a cost-sensitive replacement algorithm systematically replaces low-cost blocks whenever low-cost blocks are in the cache. A practical example of infinite cost ratio is bandwidth consumption in the interconnection network connecting the processors. The infinite cost ratio experiments also give the maximum possible cost savings for all cost ratios above 32. The cache block size is 64 bytes throughout our evaluations.

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. We selected a 16-Kbyte 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.

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

3.2. Random Cost Mapping

Although random cost mapping is not truly realistic, it allows us to easily vary the high-cost access fraction (HAF) in a trace. Figure 4 shows the relative cost savings gained by the four cost-sensitive algorithms over LRU. The relative cost savings is the ratio between the cost savings obtained by the replacement algorithm as compared to LRU, and the aggregate cost of LRU. The table attached to Figure 4 displays actual numbers. We vary the cost ratio *r* from 2 to infinite 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.

As HAF varies from 0 to 1, the relative cost savings quickly increases and consistently peaks between HAF = 0.1 and 0.3; then it slowly decreases as HAF reaches 1. Clearly it is easier to benefit from a cost-sensitive replacement algorithm when HAF < 0.5. When HAF > 0.5, there are just not enough low-cost blocks in cache to victimize.

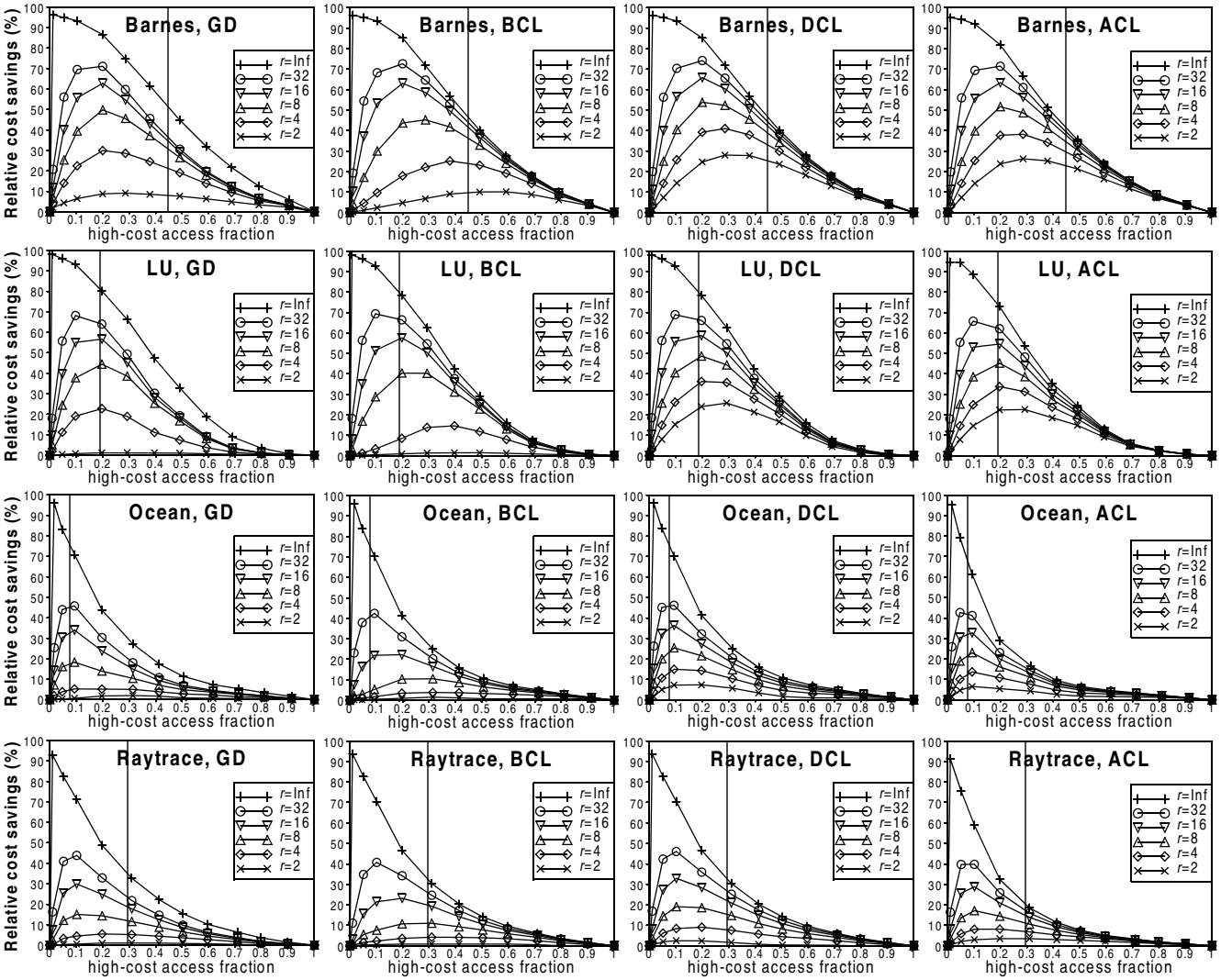
The relative cost savings increases with *r*, but for large values of *r*, it tapers off. The cost savings increases linearly with *r* in absolute terms, but not in relative terms, as the aggregate cost of LRU also increases with *r*. The savings of ACL is slightly lower than the savings of DCL in practically all situations.

Overall the results show that the relative cost savings by DCL is significant and is 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

Under the random cost distribution across block addresses, low-cost and high-cost blocks are homogeneously spread in time and across cache sets. However, in a realistic situation, cost assignments may be highly correlated. For example, 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.

Therefore, the cost savings are not as impressive in actual situations as Figure 4 would let us believe. Table 2 shows the relative cost savings as *r* varies from 2 to 32 under first-touch mapping. The high-cost access fraction



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

(HAF) in each benchmark is shown in Table 1.

Overall, we observe that the differences in the cost savings achieved under the random cost mapping and the first-touch cost mapping (corresponding to the vertical lines in Figure 4) are moderate except for LU. In LU, the savings under the first-touch policy is very poor. It even turns 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.

LU takes advantage of ACL, and even shows small positive savings in ACL. 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.

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

		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

4. Application to CC-NUMA Multiprocessors

In this section, we show that a cost sensitive replacement policy in the second-level caches can improve the performance of a CC-NUMA multiprocessors.

4.1. Miss Cost Prediction

In this application, the cost is the miss latency, which is dynamic and depends on the type of access and the global state of the block. 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 prediction of the future miss cost cannot be verified.

To approach this problem, we compare the average absolute difference in latencies between two consecutive misses in our four SPLASH-2 benchmarks for a MESI protocol without replacement hints [10] and LRU replacement, shown in Table 3. 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. The attributes are the request type (*read* or *read-exclusive*) and the memory block state (*Uncached*, *Shared*, or *Exclusive*).

The table shows that a read miss followed by another read miss to the same block by the same processor while the block is in memory state *Shared* makes up about 54% of all misses. Overall the table shows that 93% of misses are such that their latency is the same as the latency of the preceding miss by the same processor to the same block. In the remaining 7%, the average difference between past and present latencies varies widely, but remains much smaller than the local latency (60 cycles).

In our simulations, we augment each miss request message with a timestamp. When the reply is returned to the requester, the miss 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

RSIM [16] 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 in RSIM. Table 4 summarizes our target system configuration consisting of 16 processors. Data is placed in main memory using the first-touch policy. We consider 500MHz and 1GHz processors.

Due to the slow simulation speed of RSIM, the problem size of the four benchmarks in Table 1 are further reduced. We execute Barnes with 4K particles, LU with

Table 3. Latency variation in protocol without replacement hints

			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

256x256 matrix, Ocean with 130x130 array and Raytrace with teapot scene.

Table 4. Baseline system configuration

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

4.3. Execution Times

Table 5 shows the reduction of the execution time (relative to LRU) for the four cost-sensitive replacement algorithms with processors clocked at 500MHz and 1GHz. Explaining the difference in execution times is much more difficult than for trace-driven simulations, as other effects such as superscalar execution and memory conflicts may play a significant role. Nevertheless, we observe that the execution time results are broadly consistent with the observations made previously.

Table 5. Reduction of execution time over LRU (%)

	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

The results for GD, as compared to BCL, are rather mixed. 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. Overall BCL yields more reliable improvements than GD for both processors. However, the differences between BCL and GD are quite small, as compared to the differences between BCL and DCL/ACL. DCL yields reliable and significant improvements of execution times in every situation. The improvements by DCL over BCL are large in Barnes and Raytrace.

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. 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 processors, 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. Table 5 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 fraction of false match upon cache misses due to aliasing are 45%, 43%, 30% and 27% for Barnes, LU, Ocean and Raytrace, respectively. 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 due to ETD tag aliasing is very marginal.

Overall the improvements on the execution time by DCL is significant. The performance of ACL is often slightly lower than DCL, but ACL gives more reliable performance across various applications, and protects against misbehaving applications.

Table 6. Reduction of execution time over LRU with different L1/L2 cache sizes (%)

	4KB/32KB		8KB/64KB		16KB/128KB	
	DCL	**ACL**	**DCL**	**ACL**	**DCL**	**ACL**
Barnes	16.17	17.34	3.28	3.39	-0.67	-0.20
LU	11.87	13.15	0.31	0.11	0.99	0.57
Ocean	3.40	3.10	-0.46	0.29	0.25	0.31
Raytrace	4.93	6.67	5.95	6.80	-2.21	4.34

Table 6 shows the reduction of the execution time with 500MHz processors as we vary the sizes of L1 and L2 caches. The reduction of the execution time by DCL and ACL decreases with bigger caches. With the selected

problem sizes, the miss rates in L1 and L2 caches are too low in the bigger caches, and the replacement algorithm does not affect performance much. Moreover, as cache sizes increase, the local miss rate in L1 tends to improve faster than the remote miss rate due to higher locality on local accesses, and thus the HAF to L2 cache increases.

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 of the effect on cycle time.

In all four algorithms, tag and cost fields are needed. There are two types of cost fields: fixed cost fields, which store the predicted 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[2] 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 hardware 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 tag aliasing 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 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, which means a memory overhead of 0.5%, 0.9%, 1.5%, and 1.6%.

Although timing is not a critical issue for second-level caches, our algorithms affect the cache hit time minimally, if any. The additional operations on a hit are restoring the miss cost of the MRU block in GD, setting *Acost* for the LRU block in BCL, DCL and ACL, and looking up and invalidating ETD entries for DCL and ACL. These operations are trivial, given the number of bits involved. ETD lookup is not required to detect a cache hit. 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 an approximation to LRU, are widely adopted in caches [19][20]. Lately several cache replacement algorithms (e.g., [11][17][23]) have been proposed to further reduce the miss count in LRU. These proposals are motivated by the performance gap between LRU and OPT [2], and often require large amounts of hardware to keep track of long access history. Mounes-Toussi and Lilja [15] evaluated state-based replacement policies, in which the random replacement policy is modified by several static replacement priority schemes based on cache coherence states. Under the MESI protocol they observed marginal miss rate improvements over the random policy.

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. Greedy-Dual was first proposed by Young [25] later refined by Cao and Irani [3] in the context of Web caching to reduce the miss cost. They found that size considerations play a

2. 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.

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 Greedy-Dual is far less efficient than other algorithms, especially when the cost ratio is small.

The original idea behind the shadow directory [22] was to keep 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. [21] addressed the performance issues caused by critical loads in superscalar processors. Critical loads are loads that have a large miss penalty in superscalar 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.

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.

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 superscalar processors show significant reduction of execution time when cache replacements vie to minimize miss latency instead of miss count.

Our algorithms are readily applicable to the management of various kinds of storage where various kinds of non-uniform cost functions are involved. By contrast to approaches partitioning the cache or adding special-purpose buffers to treat blocks in different ways [21][7][6], cost-sensitive replacement algorithms with properly defined cost functions can maximize cache utilization.

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 [5]. 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 superscalar 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 [21]. 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 superscalar processors and multiprocessor environment provides richer optimization opportunities for cost-sensitive replacements.

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.

It would also be interesting to evaluate the interaction between our replacement algorithms and prefetching. Prefetching may change the HAF and the dynamic distribution of costs, which would affect the multiprocessor simulation results. Prefetching also reduces the effect of memory penalties, and so it may reduce the benefits of cost-sensitive algorithms relative to LRU. However, prefetching only helps execution time if it is done carefully. It is costly in terms of bandwidth and power consumption. In some sense cost-sensitive algorithms tend to have a similar effect as prefetching, but, instead of replacing high-cost blocks and then prefetching them before their next access, we keep them in cache, thus saving the power and bandwidth to prefetch them.

Finally another approach to cut down on the performance effects of remote capacity misses in multiprocessor systems is to add a large DRAM or a smaller SRAM remote data cache [14] in each processor node. For example, in NUMA-Q, each node has a 32Mbyte DRAM remote data cache [5] so that, when the remote data cache hits, the latency of a remote capacity miss is similar to the latency of a local miss. Of course the cost of such caches is much higher than the cost of applying cost-sensitive replacement policies to the second-level caches. It would seem useful to compare the cost-effectiveness of these various solutions.

Acknowledgments

Michel Dubois and Jaeheon Jeong were funded by NSF Grant No. MIP-9223812. NSF Grant No. CCR-0105761, and 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] L. Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer," *IBM Systems Journal*, v. 5, no. 2, pp. 78-101, 1966.

[3] 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.

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

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

[6] B. Fornay, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Storage-Aware Caching: Revisiting Caching for Heterogeneous Storage Systems," In *Proceedings of the 1st USENIX Conference on File and Storage Technologies, January 2002*.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] A. Moga and M. Dubois, "The Effectiveness of SRAM Network Caches in Clustered DSMs," In *Proceeding of Fourth International Symposium on High Performance Computer Architecture*, pp. 103-112, Feb. 1998.

[15] F. Mounes-Toussi and D. Lilja, "The Effect of Using State-Based Priority Information in a Shared-Memory Multiprocessor Cache Replacement Policy," In *Proceedings of International Conference on Parallel Processing*, pp. 217-224, August 1998.

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

[17] 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.

[18] J.-P. Shen and M. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill, 2003.

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

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

[21] 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.

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

[23] 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.

[24] 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.

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