

Self-Correcting LRU Replacement Policies

Martin Kampe, Per Stenstrom, and Michel Dubois*

Department of Computer Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
{mkampe,pers}@ce.chalmers.se

*Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-2562, USA
dubois@paris.usc.edu

Abstract

With wider associativity the replacement algorithm becomes critical. Although LRU makes many good replacement decisions, the wide performance gap between OPT, the optimum off-line algorithm, and LRU suggests that LRU still makes too many mistakes.

Self-correcting LRU is based on LRU augmented with a feedback loop to constantly monitor and correct the mistakes done. It relies on several mechanisms to detect, predict, and correct bad replacement decisions. We identify three types of mistakes and associate them with memory-access instructions. Our findings show that an instruction causing a mistake is prone to cause the same type of mistake at its next execution. Our goal is to prevent a mistake to occur more than once.

Our approach combines three basic mechanisms. The first mechanism is a shadow directory, which is instrumental in detecting the three classes of mistakes. A second idea is to associate replacement mistakes with the memory access instructions causing them as opposed to the data blocks. These instructions are logged in a Mistake History Table, from which predictions for future mistakes are retrieved. Finally, an MRU victim cache is added to offset the effects of misguided mistake predictions when the replaced block is in the MRU position.

Based on evaluations using a set of seven SPEC95 benchmarks, we show that our approach achieves significant miss rate improvement for 2-way and 4-way caches and can do this at a low implementation cost.

Keywords: Cache management, least recently used replacement algorithm, mistake prediction, shadow directories.

1. Introduction

Innovations in processor architecture in recent years have led to an increasing gap between processor and memory performance. A predominant approach to reduce this gap is to add a cache hierarchy between the processor and the main memory. At the top of the hierarchy is a first-level cache, accessed directly by the processor. As the processor/memory gap widens, it becomes critical to improve first-level cache behavior, as it matches closely the speed of the processor. This can be done by increasing its associativity to cut down on the number of conflict misses. As its associativity increases, however, the replacement algorithm becomes more important [17]. For a 2-way cache, the miss rate obtained by a Least Recently Used (LRU) algorithm is fairly close to that achieved by an omniscient optimal (OPT) replacement algorithm. OPT can perfectly identify the block in a set that will be accessed farthest into the future [17]. This is the optimal decision since, when we keep a block in the cache we hope to keep it until the next

access to it to avoid a miss, and this can be done most effectively for block references that are closer in the trace. With higher associativities, there are more victims to consider. For a 4-way cache, we have observed in our benchmarks that a miss rate improvement of up to 76% is possible with the OPT algorithm. While OPT is impossible to implement in practice, it is certainly possible to use it as a guide to correct mistakes made by the LRU algorithm.

A number of methods have been proposed to address this issue. The first attempts were more focused on reducing the impact of the mistakes done by LRU rather than improving the algorithm itself. One approach is to split the cache into two caches, one for blocks classified as temporal and another for spatial blocks [5][12], in order to reduce cache pollution. However, the replacement algorithm is still the same. Another class of methods addresses the weakness of the algorithm itself and the idea is to detect dead blocks, i.e. blocks only used once by the processor, and either never insert them in the cache [16] or downgrade them faster [11][17].

The novelty of our replacement policy, called *self-correcting LRU*, comes from a feedback loop which constantly detects its own mistakes and corrects them on the fly, as the program executes. We detect some of the most egregious mistakes made by LRU selection and avoid repeating the same mistakes in the future. Intuitively, LRU selection can make two major types of mistakes: keeping in the cache a block that will not be accessed again until it leaves the cache or replacing a block that will be referenced in the immediate future. Many blocks are only accessed in the most-recently used (MRU) position of the cache set and then are not accessed again before leaving the cache. Among these blocks several are accessed a single time in the cache so that they never hit; such blocks should have bypassed the cache and we call them *Bypass Blocks* (or *B-blocks*). Other such blocks are referenced more than once in the MRU position and are called *Dead Blocks* (or *D-blocks*); they should be removed as soon as they leave the MRU position in the cache. Another mistake LRU selection can make is to pick a *Live Block* (or *L-Block*) as a victim. This occurs when a block in the LRU position of the set is accessed right after the next miss. Of course, LRU makes other bad decisions, but these three mistakes involving B-, D-, and L-blocks can be easily detected and are significant enough to merit special treatment. So we have targeted them in this work to verify the validity of the whole approach.

Our approach uses a shadow directory (SD) structure [14] and a victim cache. The original SD proposal was to add a small storage space to each cache set, which keeps the tags of the last evicted blocks. If a cache miss hits in the shadow

directory, the data block is flagged. The flag gives a hint to the replacement algorithm to downgrade the block in the set at a slower pace. A glaring problem with this idea is that the flag will be lost as soon as the block leaves the cache and the SD. We address this problem by associating such blocks with the path of up to 16 memory access instructions leading up to the instruction causing the misguided replacement in a *Mistake History Table (MHT)*. We also use the SD and MHT to help keeping track of Bypass and Dead blocks. With this setup, we are able to keep track of the path of memory access instructions leading to the replacement mistakes and prevent some of them in the future.

While previous work [17] has shown that it can be disastrous to choose the MRU block for eviction, the potential gains obtained by evicting dead MRU blocks are very high. To eliminate the downside of evicting MRU blocks and still benefit from the high performance potential, we move blocks replaced in the MRU position to an *MRU victim cache* before retiring them to memory. By doing this, the cost of prediction errors is greatly mitigated. For some of the seven applications in the SPEC95 benchmarks we evaluate, we are able to reduce the miss rate by 24% with our method.

The outline of the rest of the paper is as follows. In the next section we compare the performance of LRU and OPT. In particular, we examine some of the mistakes LRU makes. In Section 3 we describe our self-correcting LRU policy in details, after which we present our experimental findings in Section 4. A proposed hardware implementation of self-correcting LRU is evaluated in Section 5. In Section 6 we put our work in context of related work. Finally, we conclude our paper and present future research in the last section of the paper.

2. Performance of the LRU Replacement Algorithm

2.1 Experimental Methodology

To analyze the miss rates of different replacement algorithms, we used the “cache” version in the SimpleScalar package [2]. In order to obtain the miss rate for the OPT algorithm, we used the “cheetha” package included in SimpleScalar. SimpleScalar is a MIPS ISA based execution-driven simulator. The simulator is driven by SPEC95 [13] benchmarks (see Table 1) selected from both the integer and floating point sets. The benchmarks are simulated with the reference set of data input.

Due to limitations in “cheetha” [2] we only simulate the first 2^{32} data references, but before that we let the benchmark execute 100 million references. However, the data cache is started cold after the first 100 million references. Two of the benchmarks are fully executed, gcc and m88ksim, since the total number of references is less than 2^{32} . By default, we model a 4K-byte with 4-way associativity and a block size of 32 bytes.

OPT knows exactly which block in the set will be referenced farthest away in the future and chooses this block as the victim. A weakness of OPT in its current form is that it loads all blocks into the cache whether or not they will be re-accessed during their residency in the cache. Thus we can improve LRU by correcting some of the mistakes it makes as compared to OPT and by bypassing the cache for blocks that

are referenced only once.

Benchmark	Int/Float	Indata	Data Ref
fpppp	float	ref	$>2^{32}$
gcc	integer	jump.i	32M
m88ksim	integer	dhry.bi	231M
mgrid	float	ref	$>2^{32}$
su2cor	float	ref	$>2^{32}$
swim	float	ref	$>2^{32}$
tomcatv	float	ref	$>2^{32}$

Table 1: Benchmarks

2.2 Comparison of LRU and OPT

In Figure 1, we compare the miss rate for each benchmark using LRU and OPT. Figure 2 shows the corresponding improvement ratio for OPT over LRU. Clearly, there is a great potential for improvement and LRU makes a large number of bad decisions, especially for the Tomcatv benchmark.

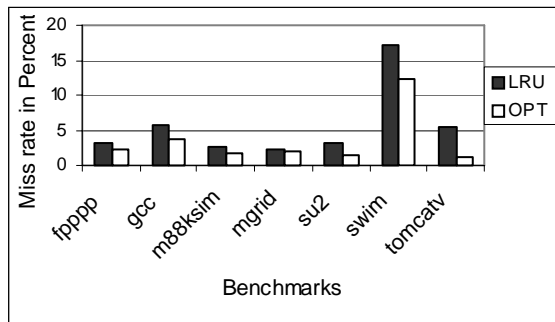


Figure 1 Miss rates for LRU and OPT in a 4k 4-way cache.

2.3 Replacement Mistakes in LRU

The first mistake LRU can make is to keep a block in the set while it is not going to be referenced before it leaves the cache. By keeping this block we occupy space that could be used by more useful blocks instead. There are two versions of this scenario. First, some blocks, labelled *Bypass (B) blocks* are never referenced while they are in the cache set; hence they could bypass the cache. Second, some blocks are accessed multiple times in the MRU position and then never accessed again. An example is a block with word spatial locality, where several words in the block are accessed consecutively with no intervening access to another block in the set and then the block is not accessed again. We label such blocks as *Dead (D) blocks*. Dead blocks should be replaced as soon as they leave the MRU position in the set.

The second mistake LRU can make is to victimize a block that is going to be referenced in the near future. This will occur when a block in the LRU position in the cache set is the next block to be referenced after a miss. We will label this block as

a *Live (L) block*. Blocks other than B, D, or L-blocks are called *Untagged (U) blocks*. Untagged blocks do not receive special treatment and their management is dictated by LRU.

In order to improve the algorithm we need to detect the LRU mistakes as well as store and use the mistake information to predict and prevent future mistakes.

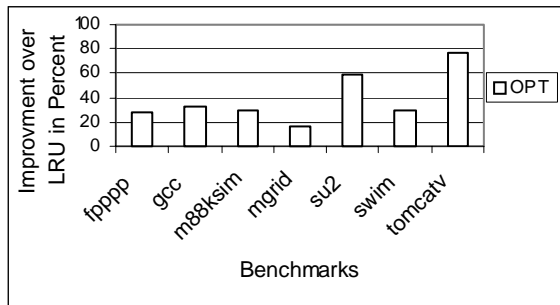


Figure 2 Improvement of OPT over LRU in a 4k, 4-way cache.

3. Self-Correcting LRU

Self-correcting LRU is based on LRU augmented with a feedback loop to constantly improve on the mistakes it made in the past. In this section, we first present the mechanisms needed to support self-correcting LRU and then we present the algorithm to predict and avoid mistakes made by LRU selection.

3.1 Mechanisms used in the Cache Structure

We detect replacement mistakes by using a shadow directory (SD). In its original proposed form [14] the shadow directory is made of a set of tags associated with every set in the cache. When a block is evicted from the set, its tag is inserted into the SD. If the tag of a missing block is in the SD, the block was evicted prematurely. Hence the block is inserted into the cache with a flag that informs the replacement algorithm to downgrade the block at a slower pace. SD has been extended to support prefetching, instead of replacement, as presented in a paper by Collins and Tullsen [4] and in a paper by Charney and Puzak [3]. Prefetching is beyond the scope of this paper.

Unfortunately, when the block is evicted from the cache and the SD, there is no information left about the past mistake [14]. To keep track of such mistakes we use an instruction-centric approach as in [10]. In this approach, we associate the block access behavior with the *accessing instruction (AI)* and save the AI, with a mistake tag in a separate storage unit, called *Mistake History Table (MHT)*. It is important to point out that only AIs that caused a mistake in the data cache are stored in the MHT. The AIs are accessed in the MHT using a simple hash function of the last bits of their program counter. The hash function we use is the same as the one proposed by Johnson and Hwu [7]. An instruction-centric approach saves space in the MHT, since a program tends to execute far less distinct memory instructions than the number of data they touch and, thus, for a given MHT size, it is possible to keep this information for longer periods of time than with a data centric approach. In addition, a block can be accessed by different instructions with various access patterns and this effect is exploited by the instruction-centric method.

Blocks that bypass the cache are directly inserted into the MRU victim cache (MRU VC for short). Because of the severe penalties associated with mispredictions [17], we also move blocks victimized in the MRU position into the MRU VC to recover from mispredictions involving the MRU block with low penalty.

Figure 3 shows our proposed cache architecture with the SD to detect and predict mistakes and the MRU VC to recover from inaccurate predictions. The baseline data cache is organized in the same way as an ordinary 4-way cache with the added SD. Each entry in the cache set also has an instruction cache tag for the AI associated with it, so that upon a detected mistake the AI is known and can be stored in the MHT. Please observe that the AI field in the data cache can be altered during the lifetime in the data cache whenever a data block is accessed through a different AI.

The third entry associated with each block in the cache set in Figure 3 is only needed when we investigate the benefits of using a path of up to 16 memory instructions feeding into the AI. We draw it in dotted lines, as it is optional. An AI with a path can separate cases when an AI has different behavior depending on the context it is reached from. The optional AI path requires more storage space in the cache as well as in the MHT. The increased storage space is proportional to the number of memory instructions in the path, since each memory instruction in the path needs the same amount of storage space as the AI itself. When the path is added, its index in the MHT is the concatenation of the last bits of every memory instruction in the path and in the AI. From now on, unless otherwise specified, we assume no path, i.e. a tag in MHT is characterized by its AI only.

3.2 Self-correcting LRU Algorithm

Self-correcting LRU is based on the LRU algorithm. At the start all AIs are untagged (U-blocks) and the corresponding data blocks are brought in the cache in the MRU position. As long as the cache hits, the management of U-blocks follows the LRU algorithm. U-blocks are replaced in the LRU position on a miss. As the algorithm progresses, we detect mistakes and start to tag blocks according to the replacement mistakes. Tagged blocks are treated differently than in the pure LRU algorithm.

3.2.1 Block Tagging

The first type of replacement mistake an LRU-based algorithm can make is to keep a D-block longer than necessary. We detect a D-block when a U-block hits multiple times in the MRU position and then is progressively downgraded to the SD-position and is finally evicted from SD without ever being accessed. The associated AI is then stored in the MHT with a D-tag.

The second type of replacement mistake is to cache a block accessed only once. This block should bypass the cache. A Bypass-block is detected when a block is not accessed in the MRU position and then is progressively downgraded to the SD-position and is evicted from SD without being accessed in any position. In other words, the block does not hit during its lifetime in the set. The AI is then stored with a B-tag in the MHT.

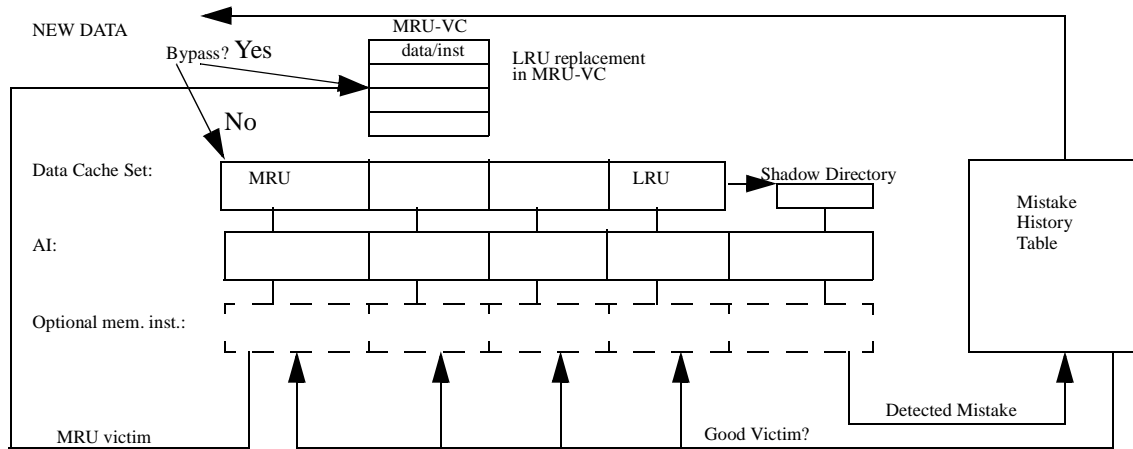


Figure 3 Organization of the cache with self-correcting LRU.

The third type of mistake is the L-block mistake. A U-block that is going to be accessed in the near future is in the LRU position and is chosen as a victim. As the block is evicted from the cache set, the cache tag of that block is placed in the SD. If a cache access hits in the SD and if there exists a block in the same cache set that was not accessed in the time window between when the victim was put in SD and the time it is accessed again, the block currently accessed is tagged as Live in MHT. The reason for this rule is that a victim accessed right after eviction can still be the correct victim if all the other blocks in the set were accessed since the victim was evicted.

Please observe that we only insert in the MHT the AIs that cause replacement mistakes in the cache rather than all AIs. Hence, we save space in the MHT. Moreover, new tags are inserted in MHT at the time of a miss only. Later on we will show that inserting new mistake tags in MHT on both hits and misses is not productive.

The AI that brought the data into the cache must be stored in the data cache because it will be needed if the AI must be logged in MHT (i.e. if a bad replacement is detected), so that mistakes are predicted at the time when a block is brought in cache. When a miss occurs in the cache, the new AI is compared to B-tagged AIs in the MHT. If there is a match, the block bypasses the cache and is placed directly in the MRU VC. Hence the content of the cache set is not affected.

3.2.2 Victim Selection

When a victim is to be chosen in the cache set (i.e. at the time of a miss), we will first look for a D-block. Each AI in the data cache set has to be looked-up in MHT to get the mistake tag, from MRU to LRU (i.e. from left to right in Figure 3) in order to evict these blocks as early as possible. The reason that the MHT must be looked up (instead of storing the tag in the cache) is that a data block can change its mistake tag if it is accessed by different AIs during its lifetime in the cache. A D-block victim is always placed in the SD in order to recover from wrong predictions. If there is a hit on the tag in SD for a D-block, the mistake tag (D) is removed by removing the AI entry from the MHT.

If the lookup in the MHT did not return any D-tags, L- and U-blocks are examined. The first U-block from LRU to MRU (i.e. from right to left in Figure 3) is evicted since U-blocks are less likely to be used in the near future than L-blocks. In case all blocks have an L-tag, the block in the LRU position is chosen as a victim. At this time, the L-block moves to SD. If the block had no hits from MRU to SD the AI is then tagged as Dead or if the block did not hit at all during its lifetime in the cache its AI is tagged as Bypass. With neither scenario being true, the L-tag is removed by removing its AI entry from the MHT.

The MRU VC is accessed in parallel with the L1 cache and if it hits, the B or D-tagged AI entry in the MHT is removed. If this occurs, a victim is chosen from the cache set and swapped with the B/D-block in the MRU VC.

4. Experimental Results

This section presents our experimental results. In Section 4.1 we show the results using a shadow directory as originally suggested by Stone[14]. Sections 4.2 and 4.3 provide an analysis for our self-correcting LRU policy for 4-way and 2-way caches.

4.1 Shadow Directory

A simple approach for self-correcting LRU is the Shadow Directory (SD) [14]. The shadow directory consists of tags added to each cache set. Upon removal from the cache set, the tag of the victim is put into SD. The SD-algorithm classifies misses into *transient misses*, in which the tag of the missing block is not in the SD and *shadow misses*, in which the tag of the missing block is in the SD. Blocks loaded in cache after a shadow miss are tagged. When a victim for replacement is chosen, the tagged blocks have lower replacement priority.

The miss rate improvement of LRU with one SD tag per set over pure LRU is shown in Figure 4. Two implementations are included. "SD 2" shows the improvement when replacements are restricted to the two blocks at the bottom of the LRU stack. In "SD 3" any block but the MRU block can be evicted (in a 4-way cache). Some of our benchmarks show a negative improvement. Obviously the simple feedback enabled by the shadow directory is neither sufficient nor reliable.

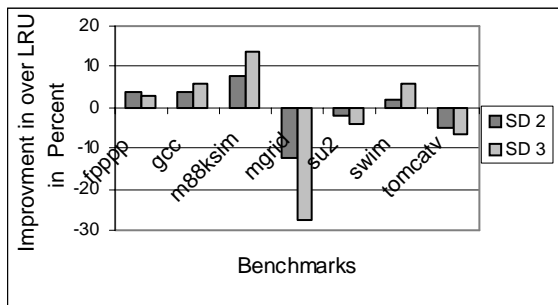


Figure 4 Improvement of LRU with SD over pure LRU.

4.2 Performance Impact of Various Design Alternatives (4-way cache)

In this section we evaluate design alternatives for the feedback loop in order to improve the correction of LRU mistakes. We first look at the effect of the size of MHT in an ideal case. Then we show that the quality of the dynamic classification of blocks into L-, D-, B-, and U-blocks is high and that having an instruction path with the AI or using predictors with two bits do not provide any improvement over the basic algorithm with the AI and 1-bit predictors.

4.2.1 Size of MHT

By taking an instruction-centric approach over a data-centric approach to mistake prediction, we expect to gain two things. First, we will have to tag fewer AIs. Secondly, the behavior associated with instructions should be more stable than the behavior associated with data.

Off-line, it is possible to tag each missing block in the cache with the correct mistake tag, by looking ahead in the trace and analyze the future accesses to the blocks in the set. With this capability we can observe the access pattern to the set after the miss, tag the block with a correct tag and select a victim with perfect accuracy. We call this ideal policy *LRU with perfect tags*. Figure 5 shows the number of different AIs logged in the MHT under LRU with perfect tags. AIs causing mistakes in LRU with perfect tags are counted to estimate the size of MHT. The same AI causing different mistakes is counted as one. As can be observed in Figure 5, 1500 entries are needed in the MHT to save all AIs causing mistakes.

4.2.2 Quality of the Block Classification

An important issue is the number of instructions before the AI. It is expected that, by adding instructions in the AI path, the accuracy of mistake predictions will be improved. We first establish the distribution of mistake tags under LRU with perfect tags, which is displayed in Figure 6.

LB, UB, BB, and DB stand for Live, Untagged, Bypass, and Dead blocks, respectively. The number attached to each benchmark indicates the number of memory instructions in the path before the AI (1 means no instruction in the path and 2 means 1 instruction in the path). As seen in Figure 6, the largest fraction of blocks are tagged as Dead. Hence we need to be most accurate at predicting D-blocks. However, there is more

to gain with Bypass blocks and the fraction of Bypass blocks is significant for four benchmarks. Our conclusion is that D and B-blocks need to be accurately predicted, whereas U and L-blocks do not need the same accuracy, since they are only a small fraction of all blocks.

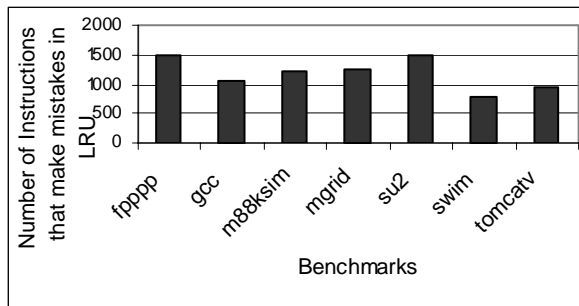


Figure 5 Number of AIs that causes mistakes in LRU with perfect tags.

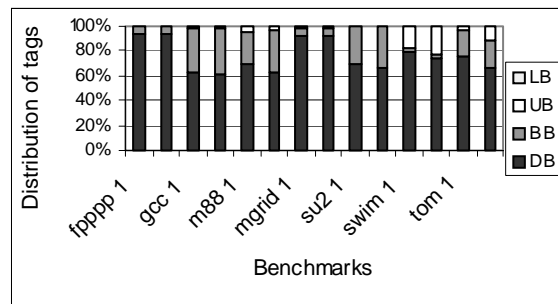


Figure 6 Block distribution with paths of zero or one before the AI in LRU with perfect tags.

Figure 7 shows the prediction accuracy for each tag. Accuracy is measured by comparing LRU with perfect tags, as described above. Hence the black bar for each type of block shows the fraction of correct mistake predictions our prediction algorithm makes. Here we only show the accuracy for empty path before the AI. We simulated paths with up to 15 instructions. Unfortunately, no noticeable difference was discovered. As observed in Figure 7, our method has a very high accuracy for D-blocks and a good accuracy for B-blocks. In other words prediction accuracy is high where it is needed, and we do not need to store a path either in the cache or in MHT.

It has been suggested in previous studies [10] to use a 2-bit predictor similar to the 2-bit branch predictor in order to improve on prediction accuracy. The 2-bit predictor prevents occasional changes of the prediction tag. Let say that an AI is live, but has a dead behavior once. If we use a 2-bit predictor, we only mispredict once. However with a 1-bit predictor, we make two mispredictions.

In Figure 8 we show the miss reduction over LRU of five variations of our prediction algorithm. The first four bars denoted LD correspond to the self-correcting LRU policy in which we correct mistakes due to L- and D-blocks only and never replace from MRU. The “3” in LD3 indicates that we select victims from the 3 blocks at the bottom of the LRU

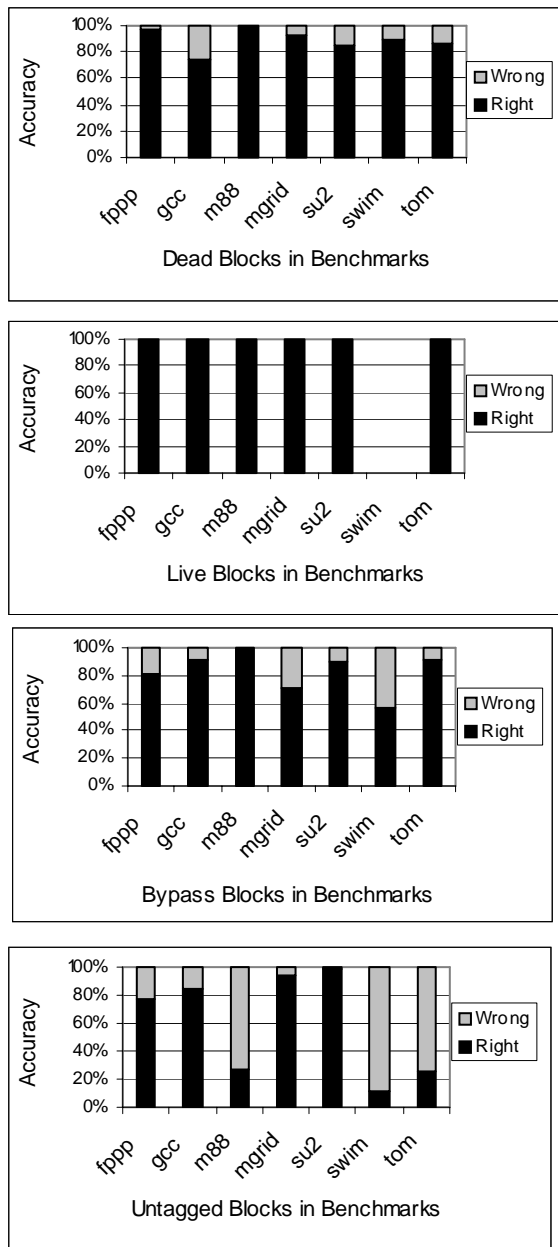


Figure 7 Accuracy of mistake tag prediction.

stack. These policies use neither cache bypass nor the MRU VC and the MRU block is left alone when choosing a victim. The policies also use 1 (only the AI), or 2 (AI plus 1) instructions, and the 1 or 2 bit predictor. As seen in Figure 8, there is no clear advantage in adding a path before the AI to characterize mistakes or in using the 2-bit predictor.

4.2.3 Bypass Effectiveness

The fifth bar of Figure 8, LDB3, uses the same algorithm as the fourth bar with the addition of correcting mistakes due to Bypass-blocks, but without the MRU-VC. B-blocks bypass the cache and move directly into the SD, in order to give them a chance to be re-tagged if necessary. We see that LDB3 is

sometimes worse than LD3 with no path before the AI and one bit prediction. This is due to the high penalty we pay if we wrongly eject an MRU block. It is clear that bypass is ineffective without a mechanism to recover from such incorrect predictions.

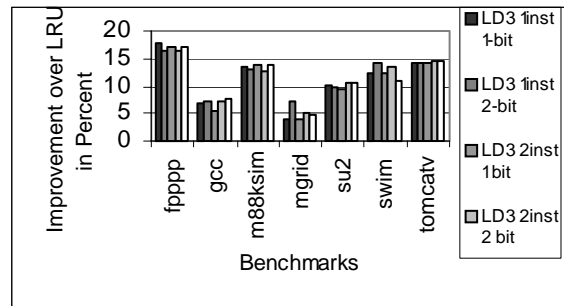


Figure 8 Variation in setup for mistake prediction.

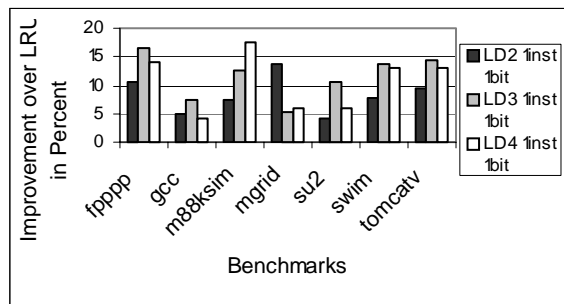


Figure 9 Impact on LD of the number of candidates for eviction.

4.2.4 Effectiveness of the MRU Victim Cache

In Figure 9 we show the improvement of LD with various numbers of victim candidates (2, 3, or 4) as compared to pure LRU. For m88ksim we can reap large gains by choosing a victim from any block in the set. However, all other benchmarks lose performance when we include the MRU block as a possible replacement candidate.

Our solution is to use a Victim Cache (VC) for the blocks that are victimized from the MRU position as well as for the blocks that bypass the cache. With the MRU VC we can still recover the data from the MRU VC in case the MRU block is wrongly victimized. In Figure 10, LD3 and LDB4 with MRU VC --our best self-correcting LRU policy-- are compared to OPT. The absolute mis rates are shown in Figure 11.

LDB4 with MRU VC is close to optimal in the cases of m88ksim and swim. For Fpppp it is better not to use the MRU VC. In the case of fpppp, there are so many dead blocks that they literally flood the MRU VC and give no chance to B-blocks. On average, only 1 block in 28 in the MRU VC needs to be recovered. The 4-entry MRU-VC is too small. With a 16-entry MRU VC we reach the same performance level as LD3 and with a 64-entry MRU VC the miss rate is practically the same as OPT's.

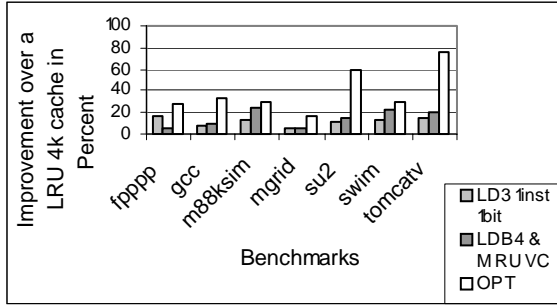


Figure 10 Comparison of LD3, LDB4 with MRU VC and OPT.

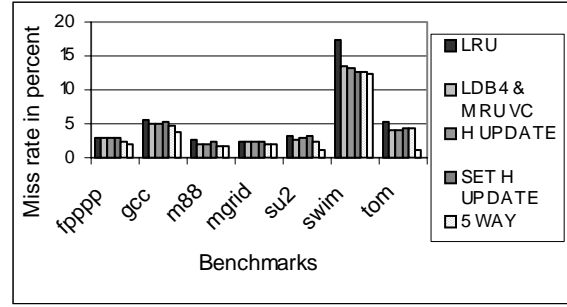


Figure 12 Effect of updating the tags on hit and of per set mistake histories compared to LRU, LDB4 with MRU VC and OPT.

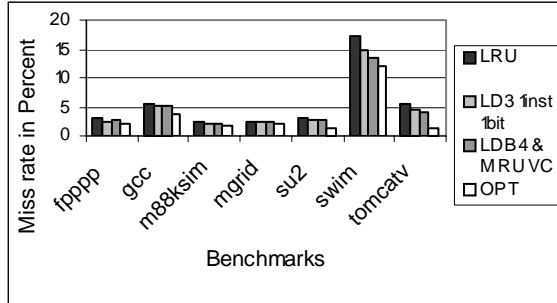


Figure 11 Miss rates for LRU, LD3, LDB4 with MRU VC, and OPT.

4.2.5 Other Options

Up until now, we update the tags in MHT on cache misses only so that accesses to MHT are off the critical path. The prediction accuracy might improve if we update MHT on cache hits as well, although it would put a strain on the feedback hardware. The reason for the better accuracy can be illustrated with a simple example. Let us assume that a block X is accessed by AI P. Assume further that AI P is tagged as D, hence X will be evicted from the cache as soon as possible. However, let us say that an access to X hits, before the next miss. Thus X is a live block, but we would end up evicting it unless we update MHT at the time of the hit on X. In Figure 12 we compare updating the tags on miss only to updating the tags on both misses and hits (H UPDATE).

Another concern with our approach up until now is that the mistake history in MHT is global for all sets. This could have an impact if an AI has different behaviors depending on the cache set the accessed block maps to. This approach is labeled SET H UPDATE in Figure 12. Clearly there is no clear advantage in either H UPDATE or SET H UPDATE.

4.3 Case of a 2-way Cache

Even though there is less room for improvement over the LRU algorithm in a 2-way than in a 4-way cache, our self-correcting LRU policy with MRU-VC still yields significant improvements over LRU, as shown in Figure 13. However the improvement is not as spectacular as for the 4-way cache.

4.2.6 Summary

In this section we have investigated a number of design decisions for self-correcting LRU policies. We find that it is preferable to use cache bypass and to select the victim from any position in the set. However to do so we need a victim cache for MRU block and Bypass blocks in order to recover from incorrect predictions. A path of memory instructions before the AI is not necessary nor is a 2-bit predictor in MHT. It is also sufficient to update MHT on cache misses only and to keep a single MHT for the entire cache.

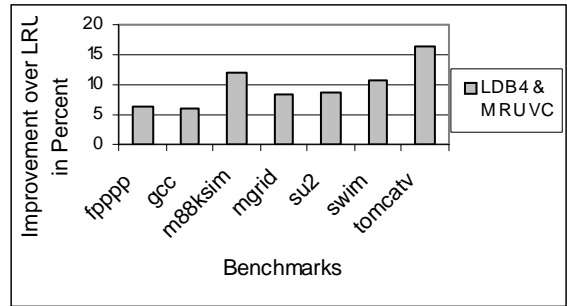


Figure 13 Improvement over LRU with LD4 & MRU VC in a 2-way data cache.

5. Hardware Implementation

Our suggested hardware implementation is built on the model described in Section 3. However there are a few additions.

First each data block has two bits associated with it. The first bit is a “Hit on MRU” bit (HMRU) and the second bit is “Hit after Last Eviction” (HLE). If a block access hits in the MRU position, its HMRU-bit is set but, if a subsequent access to the block hits again on its way towards the LRU-position, the HMRU bit is unset. Its function is to determine whether the block is a Bypass-block.

In order to determine if a block is an L-block, we add the HLE-bit. A block could be viewed as an L-block if it hits in SD (as was done in the original proposal [14]). However, there may not have been any option to choose another block. Each HLE bit in a set is set when a block is loaded in SD and is reset when its block is accessed. If the tag in the SD position hits on

a cache miss and at least one of the HLE bits in the set is set, the retired block is tagged as Live.

Regarding the D-blocks, all blocks evicted from the LRU position are considered D-blocks if they did not experience any hit in the cache after the MRU position. Therefore another bit in the SD indicates whether the victim comes from the LRU position or not, and is called the LRUV-bit.

The overall hardware overhead is small. The last 16 bits of the AI (resulting in negligible aliasing) are stored with each data block and SD. Space is also needed for the LRUV-tag in SD, and the HLE-bit and HMRU-bit for each data block entry. The additional hardware needed for each set is displayed in Figure 14. We need a total of 117 bits of storage per set. Additionally we also need a 4 blocks MRU-VC.

As the cache will be slightly larger, we compare it to a 5-way LRU cache in Figure 15. Please observe that a 5-way cache with 32 byte blocks is larger than our implementation by 12%. In the diagram we can observe that the miss rate for a 5-way cache is lower for all benchmarks except Tomcatv. However MRU VC is close to 5-way and it uses less hardware.

The MHT is also part of the cache and has 1500 entries used in conjunction with a simple hash function. Each MHT entry contains a 16-bit key, the AI, and a 2-bit field to store one of three different tags (L-/D-/B-). MHT is not on the critical path, since MHT lookups are only made on a miss in the L1 cache. Hence the MHT can be part of the L2 cache as long as it is possible to lookup five AIs before the cache controller needs to decide which of the four data blocks to evict or whether the new data is to be inserted at all.

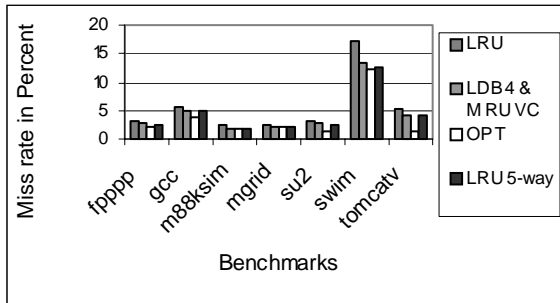


Figure 15 LRU for a 4-way cache compared to LDB4 & MRU VC, OPT, and a 5-way LRU cache.

6. Comparison with Related Work

In a paper by Wong and Bear [17] dead blocks are tagged with the accessing instruction in the context of level two caches. Their approach comes in two versions: static and dynamic. The static version, called PRL, profiles a program and detects AIs accessing data with temporal behavior. Data blocks accessed by an AI with temporal behavior are dynamically tagged in the cache (similar to our live blocks), and the

replacement algorithm evicts data blocks that are not temporal (similar to our dead block) first. A dynamic approach (ORL) is also proposed. The dynamic version works in the same way as the static, but the temporal property of AIs is determined by accessing a lookup table similar to our MHT. The previous AI to a block is inserted in the table when an access to the block hits in the cache. AIs are also stored in the same manner as we do in the data cache. Because the prediction is not always correct, both approaches simply avoid evicting MRU blocks. As a result, the penalty for an incorrect prediction is not as high. However higher gains obtained by evicting MRU or bypassing the cache are squandered.

In a paper by Lai *et al.* [11], the prediction for live/dead blocks is similar to ours. However, it is applied to prefetching and not to cache replacement. As soon as a block is considered dead, it is replaced by a prefetched live block. Lai *et al.* made the same observation that it does not pay off to increase the length of the path before the AI. It takes more than 16 memory instructions in the AI path to see any significant miss rate improvement as compared to none or one instruction in the path.

The PCS [15] model is one of the most recently proposed methods using bypassing. PCS is instruction centric like our approach and has evolved from the C/NA model, a data-centric method developed by Tyson *et al.* [16]. The data cache in the PCS approach is divided into two parts: A large ordinary cache and a smaller cache. On a miss, a Detection Unit (DU) is accessed. This unit is similar to our MHT. If the AI is present in DU and is non-temporal (what we call Dead), the data is placed in the smaller cache to exploit the spatial locality. The AI is tagged as non-temporal if the data only experiences spatial behavior while being in the cache. All data are originally placed in the larger cache.

The PCS model was developed for direct-mapped caches and, in this context, it reduces the miss rate significantly. In Figure 16 we show the miss rate reduction for PCS, using one instruction of history (PCS1) and two instructions of history (PCS2) for a 4-Kbyte 4-way cache and a DU with 1500 entries. Also the small cache contains 4 blocks so that the amount of hardware is comparable to our proposal. As observed, the miss rate reduction is far from perfect. Because the small cache has only 4 entries, there are only 3 global misses to detect that a block was incorrectly classified as non-temporal. By contrast, our approach uses a shadow tag for each cache set, as well as a 4 entry MRU VC; hence, in the best case, there are a total of (#cache sets + 4) global misses to detect incorrect predictions. We have shown the importance of recovering from inaccurate predictions. Since we compare a 4K LRU cache to a PCS cache our cache has the same total size, including the shadow tag, as the ordinary larger cache in PCS.

$$\begin{array}{r}
 \text{Data Set:} \\
 \begin{array}{|c|c|c|c|} \hline \text{Data} & \text{AI} & \text{HLE} & \text{HMRU} \\ \hline 256 & 16 & 1 & 1 \\ \hline \end{array} \\
 \times 4 \\
 \hline
 \begin{array}{|c|c|c|} \hline \text{SD} & \text{AI} & \text{LRUV} \\ \hline 28 & 16 & 1 \\ \hline \end{array} \\
 = 1141 \text{ bits}
 \end{array}$$

Figure 14 The number of bits required in each data set, excluding the data tag, in LDB4 & MRU-VC cache.

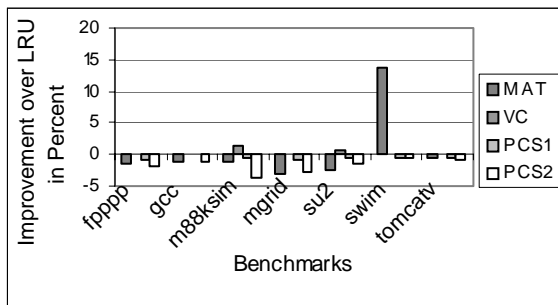


Figure 16 Improvement over LRU with MAT, VC, PCS1, and PCS2.

The Memory Accessing Table (MAT) is described in a paper by Johnson and Hwu [7] and in Johnson *et al.* [6]. Sections of memory, called macroblocks in [6], are tagged as live or dead. Each macroblock has a counter associated with it and upon a hit in the cache to any block in the macroblock the counter is incremented. The counters are stored in the Memory Access Table, which is similar to our MHT. When a miss occurs, the hit counter entry in MAT for the macroblock containing the victim is decremented and compared to the new block. If the victim has a higher MAT value the new block is placed in a smaller bypass buffer. This bypass buffer is similar to our MRU VC. For fair comparison with our self correcting LRU we simulate a 4-block bypass buffer and a MAT with 1500 entries. We use a macroblock of 1024 Kbytes, as suggested in [6].

Figure 16 presents the improvement with MAT over LRU. As with PCS the MAT approach was originally evaluated for direct-mapped caches, where it is very effective, and not for set-associative caches, where it is much less effective. (This low effectiveness of the MAT method with higher associativity was also observed in other studies [9].) The reason for the bad performance is that the bypass buffer is too small, as in the PCS model. Only three global misses can occur before the incorrectly predicted block is hit again to be able to recover from inaccurate predictions. In our benchmarks (with the exception of Swim), the MAT blocks frequently change behavior, hence inaccurate predictions occur frequently. If inaccurate predictions are common a more efficient method for recovery is needed.

A major difference between both MAT and PCS, and our proposal, is that the detection unit in PCS and the MAT are updated on every access, while our MHT is only updated on misses. Hence, in our case, the history table is not in the critical cache access path.

We have also implemented the MAT algorithm according to [6][7], using an instruction-centric approach. Instead of associating the counter value with a section of the memory we associate the counters with an AI. The rationale for this, as previously discussed, is that AIs tend to have more stable behavior than data. As shown in Figure 17, the idea of associating the counters to AIs is not very good. This poor performance is mostly due to the fact that, in the instruction-centric MAT approach, practically all AIs must be inserted in the MAT. By contrast, we only insert AIs causing mistakes in

MHT. In other words, the MAT instruction-centric method cannot save information long enough, since the MAT is flooded with AIs.

The goal of both PCS and MAT is to reduce conflict misses in direct-mapped caches. A 4-way cache eliminates most conflict misses as is demonstrated by including a victim cache [8]. In Figure 16 we see that a victim cache with four entries does not improve the miss rate much. This is one of the reasons why PCS and MAT do not work well in a 4-way cache.

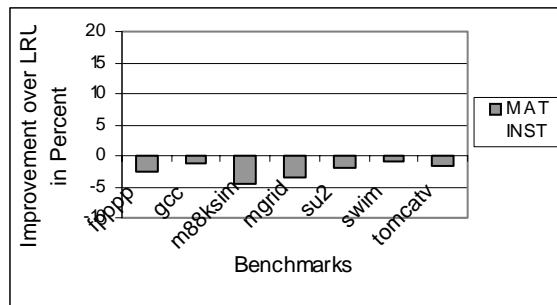


Figure 17 Improvement over LRU with MAT INST.

Another approach to improving the miss rate in a cache is to split the cache in two parts, a temporal- and a spatial- part. In this setup, the temporal blocks may live longer and hopefully experience more hits. The idea was originally presented by Milutinovic *et al.*[12] and later by Gonzales *et al.* [5]. Instead of partitioning the cache statically, a dynamic partitioning scheme was presented in a paper by Kampe and Dahlgren [10], in which data blocks are loaded in different ways depending on whether the data exhibits temporal or spatial behavior.

A good survey of all the approaches presented in this section can be found in a paper by Tam *et al.* [15].

Most approaches to reduce the miss rate in a cache either target the replacement algorithm, as in our case, or use prefetching in one form or another. A different but interesting approach is presented in a paper by Karlsson and Hagersten [9]. Their method reduces the miss rate by carefully selecting the blocks to place in L1. In order to make this decision all data blocks are first placed in a smaller cache parallel to L1 and called K1. The behavior of a block in K1 decides whether the block is placed in L1 upon eviction from K1. However, the authors observe the need for a way to recover from a poor decision on where to place an evicted K1 block. Upon eviction from K1 all data are timestamped and, if the time span between the eviction of and the next access to the same block, is shorter than a threshold the block is placed in L1, even if the behavior in K1 indicated it should not. Thereby it is possible to recover from faulty predictions. The problem is to define the threshold. Counting the number of references between eviction and the next access is not sufficient, as shown in the paper, since different applications have different threshold. Most likely the threshold is also data dependent. This problem is avoided by defining a new time unit, the *Cache Allocation Tick*, which is the number of blocks placed in L1. This unit seems to have very stable behavior according to the data presented in the paper. Instead of storing the information regard-

ing the behavior of the blocks in a special unit, such as our MHT, it is stored with the data block in L2. Of course, the effect of placing behavior data in L2 is yet to be determined, since that space in L2 could also be used for actual data.

7. Conclusion and Future Work

The main contribution of this paper is to show that it is possible to improve the LRU-replacement algorithm in a data cache by detecting the mistakes it makes. Our approach, called self-correcting LRU, works by detecting and correcting dynamically two types of mistakes made by LRU. The first mistake is the dead block mistake and it comes in two variants. The first variant is the Bypass-block. The Bypass-block is only referenced once and then is never referenced again for a long time. There is no use to put a Bypass-block in the cache and by doing so the space is saved for more useful blocks. The second variant is the Dead (D) block, where the block only hits in the MRU position due to spatial locality. D-blocks must be evicted as soon as possible after leaving MRU to save the space for more useful blocks. The second mistake the LRU algorithm can make is to evict a block that is going to be accessed in the near future. The cache should keep these Live (L) blocks for a longer period. By using a shadow directory to detect the mistakes in each set of the data cache we are able to improve the miss rate by up to 24% in a 4-way cache. The effectiveness of self-correcting LRU is by no means limited to 4-way caches as investigated here. As shown in Figure 13, it improves the miss rate of a 2-way cache as well. With wider associativities, we believe that there is even greater room for improvement.

The possibilities of our self-correcting LRU algorithm are far from fully investigated. It can for instance be used for prefetching as suggested in other studies [11]. The Mistake History Table (MHT) can also be improved, since all of the detected AIs in Figure 5 are not necessary for good performance. In our future studies, we will focus on improving the hardware implementation as well as looking into using the prediction scheme for prefetching.

Acknowledgments

The invaluable stay for Martin Kampe at University of Southern California has been supported by grants from The Swedish Foundation for International Cooperation in Research and Higher Education (STINT) under the MECCA project. Michel Dubois is funded by NSF Grant No. CCR-0105761 and by an IBM Faculty Partnership award. Martin Kampe is indebted to Jianwei Chen for valuable discussions on this topic.

References

[1] L. A. Barroso, K. Gharachorloo and E. Bugnion. Memory System Characterization of Commercial Workloads. In *Proceedings of the 25th International Symposium on Computer Architecture*, pp. 3-14, June 1998.

[2] D. Burger, Doug and T. M. Austin., *The SimpleScalar Tool Set, Version 2*. University of Wisconsin-Madison Computer Science Department, TN-1342, 1997.

[3] M.J. Charney and T.R. Puzak. Prefetching and Memory System Behavior of the SPEC95 Benchmark Suite. *IBM Journal of Research and Development*, 41(3), May 1997.

[4] J. Collins and D. Tullsen. Hardware Identification of Cache Conflict Misses. In *Proceedings of MICRO-32*, pp. 126-135, November 1999.

[5] Antonio González, Carlos Aliagas and Mateo Valero. A Data Cache with Multiple Caching Strategies tuned to Different Types of Locality. In *Proceedings of the 9th ACM international conference on Supercomputing*, pp. 338-347, July 1995.

[6] T. L. Johnson, D. A. Connors, M. C. Merten, and W. W. Hwu. Run-Time Cache Bypassing. In *IEEE Transactions on Computers*, Vol. 48, No. 12, pp. 1338-1354, December 1999.

[7] Teresa L. Johnson and Wen-mei W. Hwu. Run-time Adaptive Cache Hierarchy Management via Reference Analysis. In *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 315-326, June 1997.

[8] N.P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 364-373, May 1990.

[9] M. Karlsson and E. Hagersten. Timestamp-Based Selective Cache Allocation. In *Proceedings of the Workshop on Memory Performance Issues*, held in conjunction with the *28th International Symposium on Computer Architecture*, Goteborg, Sweden, June 2001.

[10] M. Kämpe and F. Dahlgren. Exploration of the Spatial Locality on Emerging Applications and the Consequences for Cache Performance. In *Proceedings of the 14th International Parallel and Distributed Computing Symposium*, pp. 163-170, May 2000.

[11] A-C Lai, C. Fide, and B. Falsafi. Dead-block Prediction and Dead-block Correlating Prefetchers. In *Proceedings of the 28th International Symposium on Computer Architecture*, pp. 144-154, July 2001.

[12] V. Milutinovic, B. Markovic, M. Tomasevic, and M. Tremblay. The Split Temporal/Spatial Cache: Initial Performance analysis. In *Proceedings of SC'96*, pp. 63-70, March 1996.

[13] Standard Performance Evaluation Corporation. <http://www.spec.org>. *SPEC 95*. August 1995.

[14] H. S. Stone. *High Performance Computer Architecture*. Addison-Wesley 1993 (3rd ed.). ISBN: 0-201-52688-3.

[15] E. S. Tam, J. A. Rivers, V. Srinivasan, G. S. Tyson, E. S. Davidson. Active Management of Data Caches by Exploiting Reuse Information. In *IEEE Transactions on Computers*, Vol. 48, No. 11, pp. 1244-1259, November 1999.

[16] Gary Tyson, Matthew Farrens, John Matthews, Andrew R Pleszkun. A Modified Approach to Data Cache Management. In *Proceedings of MICRO-28*, pp. 93-103, November 1995.

[17] Wayne A. Wong and Jean-Loup Baer. Modified LRU Policies for Improving Second-level Cache Behavior. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, pp. 49-60, January 2000.