# **Simple Penalty-Sensitive Cache Replacement Policies**

#### Jaeheon Jeong

Intel Corporation Digital Enterprise Group Hillsboro, OR97214, USA

#### Per Stenström

Department of Computer Engineering Chalmers University of Technology SE-412 96 Göteborg, Sweden

#### **Michel Dubois**

Department of Electrical Engineering - Systems University of Southern California Los Angeles, CA90089-2562, USA PERS@CE.CHALMERS.SE

JAEHEON.JEONG@INTEL.COM

DUBOIS@PARIS.USC.EDU

### Abstract

Classic cache replacement policies assume that miss costs are uniform. However, the correlation between miss rate and cache performance is not as straightforward as it used to be. Ultimately, the true performance cost of a miss should be its access penalty, i.e. the actual processing bandwidth lost because of the miss. Contrary to loads, the penalty of stores is mostly hidden in modern processors. To take advantage of this observation, we propose a simple scheme to replace load misses by store misses. We extend LRU (Least Recently Used) to reduce the aggregate miss penalty instead of the miss count. The new policy is called PS-LRU (Penalty-Sensitive LRU) and is deployed throughout most of this paper. PS-LRU systematically replaces first a block predicted to be accessed with a store next. This policy minimizes the number of load misses to the detriment of store misses.

One key issue in this policy is to predict the next access type to a block, so that higher replacement priority is given to blocks that will be accessed next with a store. We introduce and evaluate various prediction schemes based on instructions and broadly inspired from branch predictors. To guide the design we run extensive trace-driven simulations on eight Spec95 benchmarks with a wide range of cache configurations and observe that PS-LRU yield positive load miss improvements over classic LRU across most the benchmarks and cache configurations. In some cases the improvements are very large.

Although the total number of load misses is minimized under our simple policy, the number of store misses and the amount of memory traffic both increase. Moreover store misses are not totally "free". To evaluate this trade-off, we apply DCL and ACL (two previously proposed cost-sensitive LRU policies) to the problem of load/store miss penalty. These algorithms are more competitive than PS-LRU. Both DCL and ACL provide attractive trade-offs in which less load misses are saved than in PS-LRU, but the store miss traffic is reduced.

## 1. Introduction

To sustain the performance growth of modern processors, the hit rate, access time and bandwidth

of caches must be improved. The common approach to improve the cache hit rate is to increase the cache size and associativity. To improve the trade-off between the hit rate and access time, caches are often organized in hierarchies. These hierarchies work well when the first-level cache has a good hit rate. Otherwise the cost of handling misses may offset the gains. Fortunately, the tight integration of the processor with its caches enables sophisticated hardware schemes to better manage the caches. Cache related activities inside the processor are easily monitored and the information gathered can be used to optimize cache behavior. If the added hardware cost and complexity are reasonable, such approaches become feasible and cost-effective.

Cache replacement algorithms widely used in modern processors are LRU (Least Recently Used), Partial LRU (PLRU), and Random [19, 20]. These algorithms aim to reduce the aggregate miss count, and their cost model is that all misses have the same performance effect. However the correlation between miss rate and cache performance in modern processors is not as straightforward as it used to be. A precise measure of the cost of a miss is the *penalty* paid when some processor resource is stalled waiting for the completion of a memory access. More effective cache replacement policies should be sensitive to the penalty of a miss because misses are different, in the sense that their performance impact can vary widely. One problem is that the miss penalty is very dynamic and variable due to non-blocking caches and dynamic instruction scheduling. Thus the accurate measurement and prediction of memory access penalties are both very hard problems [1, 17, 21].

One simple approach is to take advantage of the penalty difference between a store miss and a load miss. In a processor with a properly designed and enabled store buffer, the penalty of stores is mostly hidden because a store retires as soon as it reaches the top of the active list, whereas, in the case of loads, the processor needs the value back before the load can retire. It is therefore advantageous in a system to replace load misses with store misses, by giving higher replacement priority to cache blocks that are accessed next by a store instruction. In the process the total miss rate may be increased, but we expect that the aggregate penalty of all misses will be less. We call such replacement policies Penalty-Sensitive replacement policies. Although any replacement policy such as Random and Pseudo LRU can be sensitized to penalty, we focus in this paper on an extension to LRU called PS-LRU (Penalty-Sensitive LRU). In this policy, blocks that will be accessed next by a store are replaced first, regardless of their position in the LRU stack. This is equivalent to assuming a cost model in which the penalty of stores is zero and the penalty of loads is one, and to minimizing the aggregate penalty of all misses.

To reap the benefits of this idea, we need to effectively predict the next access type (load or store) for every block in the set at the time of replacement. This can be done statically through profiling at compile-time, or dynamically by maintaining Access Type Predictors (ATPs). ATPs have other applications besides penalty-sensitive cache replacement, but, in this paper, we concentrate on cache replacements. We introduce and evaluate various prediction schemes based on instructions and broadly inspired from branch predictors. To guide the design we run extensive tracedriven simulations on eight Spec95 benchmarks with a wide range of cache configurations and observe that PS-LRU yield positive load miss improvements over classic LRU across most the benchmarks and cache configurations. In some cases the improvements are very large.

Although the total number of load misses is minimized under our simple policy, the number of store misses and the amount of memory traffic both increase. Moreover store misses are not totally "free". To evaluate this trade-off, we apply DCL and ACL (two previously proposed cost-sensitive LRU policies) to the problem of load/store miss penalty. These algorithms are more competitive than PS-LRU. Both DCL and ACL provide attractive trade-offs in which less load misses are

saved, but the store miss traffic is reduced.

The rest of the paper is organized as follows. In Section 2, we introduce our simple penaltysensitive extension to LRU called PS-LRU and also review the DCL and the ACL policies. Section 3 describes our architecture assumptions and the evaluation methodology. In Section 4 we evaluate penalty-sensitive replacement algorithms when the penalty is 100% accurately predicted. In Sections 5, 6, and 7 we develop and evaluate more realistic policies. Section 9 compares PS-LRU and DCL/ACL. Finally we overview previous work in Section 9 and conclude in Section 10.

### 2. Penalty-sensitive Cache Replacement Policies

Cost-sensitive replacement policies were designed to take into account both locality and miss cost in the replacement decision [7, 9]. There are many possible approaches to account for non-uniform miss costs in the management of caches. Here, we overview the various policies exploited in this paper.

#### 2.1. Basic Cost-sensitive Policy

One simple approach consists in always victimizing the lower-cost block with the least temporal locality (lowest in the LRU stack). The simplest case is when we have two types of blocks so that misses of one type has cost zero and misses of the other type have a non-zero constant cost, and the cost ratio is infinite. In this case, blocks with zero cost are always replaced first (since they don't add to cost) until the only blocks left in the set have nonzero cost.

The policy deployed throughout most of this paper follows this approach. Each store miss is assigned a cost of zero and each load miss is assigned a cost of one. We divide the blocks in each cache set into two subsets: *store subset* and *load subset*. The store subset contains the blocks that will be accessed next by a store whereas the load subset contains the blocks that will be accessed next by a load. If the store subset is empty, the victim is selected by LRU, otherwise LRU is used to select a victim among the blocks in the store subset. This new policy is called PS-LRU (Penalty-Sensitive LRU)<sup>1</sup>.

Although very simple, this approach uses cost as the dominant factor for replacement and tends to keep nonzero cost blocks in the set for inordinate amount of time, which may increase the miss rate on zero cost blocks and thus contribute to added memory traffic.

#### 2.2. ACL and DCL

Towards the end of the paper we will evaluate the effectiveness of this basic strategy by comparing it with previous strategies in which both locality and cost are constantly in competition for each replacement decision [8]. More specifically, we use two replacement strategies previously proposed [8, 9] and called DCL (Dynamic Cost-sensitive LRU) and ACL (Adaptive Cost sensitive LRU), and we apply them to the case of load/store miss penalty. In these strategies the cost ratio is

<sup>1.</sup> Similar penalty sensitive policies could be defined with Random and PLRU. We did not run any experiments with PS-Random and Random, but we ran all experiments reported in this paper with PS-PLRU and PLRU. There was very little difference in all the observations. Thus, the fact that the miss rate of PLRU is, in general, worse than the miss rate of LRU does not translate into significant penalty increases for penalty-sensitive policies.

finite and we assign a finite cost to both load misses and store misses. Of course the cost of a load miss is always larger than the cost of a store miss. As the cost ratio between load and store misses goes down, more and more blocks in the load subset are replaced even if there are blocks in the store subset (although there is always a bias in favor of replacing blocks in the store subset). Here we briefly explain the idea behind DCL and ACL. More details can be found in [9].

In DCL, we keep track of the position of each block in an LRU stack as well as of the dynamic cost of each block. When a block is loaded in the set its predicted next miss cost is tagged on to it as its dynamic cost. We always victimize the LRU block with the lowest dynamic cost and, every time we victimize a block that is not in the LRU position, we say that the block in the LRU position is *reserved*. If it turns out that the victimized block is later referenced before the reserved block then we reduce the dynamic cost of the reserved block by the dynamic cost of the victimized block. With this competitive algorithm the dynamic cost of a reserved block goes down every time a non-LRU block is wrongly victimized in its place. At one point the reserved block becomes a candidate for replacement.

DCL pursues reservations of LRU blocks greedily, whenever a high-cost block is in a low locality position. Although reservations are terminated quickly if they do not bear fruit, the wasted cost of these attempted reservations accrue to the final cost of the algorithm. It turns out that, in some applications, reservations yielding cost savings are often clustered in time, and reservations often go through long streaks of failure. We also have observed that this pattern of alternating streaks of success and failure varies significantly among cache sets.

Based on these observation, the adaptive cost-sensitive LRU algorithm (ACL) derived from DCL implements an adaptive reservation activation scheme exploiting the history of cost savings in each set. To take advantage of the clustering in time of reservation successes and failures, we associate a two-bit saturating counter in each cache set to enable and disable reservations. The counter increments or decrements when a reservation succeeds or fails, respectively. When the counter value is greater than zero, reservations are enabled, otherwise the policy reverts to LRU.

Implementations of DCL and ACL are quite simple and details can be found in [9]. Extensive simulations reported in [8] and [9] show that, although ACL does not always yield the lowest cost, its behavior is more stable across benchmarks.

## 3. Baseline Architecture and Evaluation Methodology

### 3.1. Baseline Architecture

Figure 1 shows the baseline architecture. It consists of a processor with on-chip split instruction/ data caches and a store buffer in parallel with the data cache, and a main memory connected through a system bus. The instruction cache is infinite and the data cache is non-blocking and write-back.

The role of the store buffer is to eliminate the penalty of stores [11][18]. Among possible write policies [11] we use write-allocate and fetch-on-write. When a store misses in the data cache, the store immediately writes the data into the store buffer if an entry is available and the store instruction can retire. With this policy, the only penalty incurred by stores is when the store buffer is full. However this penalty can be minimized with a proper retirement policy and a large store buffer [18].

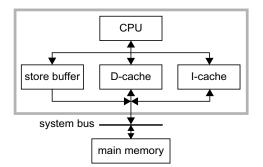


Figure 1: Block diagram of the baseline architecture.

### 3.2. Evaluation Methodology

We use trace-driven simulations to efficiently evaluate various replacement policies and approaches to prediction. The traces are generated from eight Spec95 benchmarks [22] whose main characteristics are listed in Table 1. For the benchmarks marked with an asterisk, the inputs have been modified to yield reasonable trace sizes and simulation time.

	Benchmark	Input	Inst. count $(10^6)$	Loads (%)	Stores (%)	Load PCs	Store PCs
	compress	compress train		21.7	13.1	762	569
	gcc	protoize.i	501.9	26.8	14.7	32417	14392
Int	go	train	546.7	21.1	7.6	11352	5385
mit	ijpeg	test	537.6	20.0	8.7	3966	2783
	li	train	183.3	25.9	16.6	1443	1141
	vortex	train	742.7	30.5	22.3	14719	13157
FP	apsi	train	662.6	24.0	13.9	6257	4377
r'F	mgrid	train	585.9	33.3	3.0	1903	1297

### Table 1: Characteristics of the benchmarks.

The traces are generated with the Simple scalar tool set [2]. This simulator models an ILP processor with an ISA similar to the MIPS ISA. Fortran programs are translated into C programs using an f2c translator from AT&T. The benchmarks are compiled by gcc2.6.3 with O3 and loop unrolling optimization options. The traces are gathered for the entire execution to avoid the problem of selecting a representative execution window. The traces include all user data references but no instructions. They also include the PCs of every memory references to simulate access type predictions. With these traces we are able to derive a complete set of results such as the ones in Figure 2 in less than 12 hours on our workstation. The speed is critical to this research since the search space of solutions is huge.

The cache is write-back and blocks are allocated in the cache upon misses. Cache block size is set to 32 bytes for all evaluations and cache associativity varies from 2 to 8. Cache size selection is a difficult problem to fairly evaluate replacement algorithms. If the working set is too small or too large, replacement algorithms do not affect much the performance of the cache. In practice the gains by replacement algorithms may vary significantly depending on the working set size profile of each application. Thus, we vary the cache size from 16 KBytes to 1 Mbytes rather than simply arbitrarily picking a few cache sizes.

Our primary metric is the reduction of the number of replacement load misses over the basic replacement policy. We only count replacement misses since cold misses are not affected by replacement algorithms. A miss is counted only if the missing block was previously replaced from the cache. By doing this, caches are automatically and precisely warmed up without blindly skipping a certain number of references, and the performance comparisons are fair and noise-free with respect to replacement algorithms. We also measure the memory traffic to estimate the impact of the replacement policies on memory bandwidth.

### 4. Perfect Access Type Prediction

One key implementation issue is to predict whether the next access to each block in a set will be a load or a store. When the predictions are 100% accurate, we maximize the number of replacement store misses displacing load misses and we minimize the increase in the number of replacement load misses due to bad predictions. Thus, under perfect access type prediction, we can expect a maximum reduction of the number of replacement load misses, across all feasible prediction schemes.

Figure 2 shows three graphs per benchmark. To get these numbers, we first run the trace and mark each memory access to a block with the next access type to the same block; then we use this augmented trace to simulate the system with 100% access type prediction accuracy. The first graph shows the load miss rate (i.e., the total number of replacement load misses by LRU divided by the total number of loads) as a function of the cache size. The second graph shows the load miss savings (i.e., the total number of replacement load misses saved by PS-LRU over LRU) as a function of the cache size. The third graph shows the load miss improvement by PS-LRU over LRU (this is the load miss savings divided by the number of replacement load misses) as a function of the cache size.

First of all we observe that the load miss improvement is always positive and can be huge in some cases. For example, *compress* and *ijpeg* reach more than 80% improvement, and *gcc* and *apsi* more than 50%. At first these improvements were surprising to us, because the fraction of store misses that can displace load misses is not that much. However, the improvements are related to working set behavior rather than relative number of load and store misses.

By correlating the load miss rate and the load miss savings, we observe that the benchmarks can be classified into two groups. In the first group which includes *gcc*, *go*, *vortex* and *apsi*, the savings decreases almost monotonically with the load miss rate. This means that the savings opportunities are uniform across all caches sizes. In the second group which includes *compress*, *ijpeg*, *li* and *mgrid*, the load miss savings curves have one or two peaks. The peaks occur for 16 KBytes caches in *ijpeg* and for 64 KBytes cache in *compress* and *li*. *Mgrid* shows two peaks, one for 32 KBytes and one for 512 KBytes caches. These savings peaks can be explained by the data working set behavior of these applications. At a working set size transition (i.e. when the cache becomes large enough to contain the next working set), the penalty sensitive policy effectively increases the cache size for loads, by providing more cache space for data next accessed by loads. Even if this effective cache size increase is relatively small the rapid fall of the miss rate at the working set size transition is directly translated into a rapid rise of the load miss savings. In a range of cache sizes with no working set size transition, the savings are rather moderate and rather insensitive to the cache configuration. For very large cache sizes, the data of the application fits in the cache and so the load miss savings reaches zero as replacements become very scarce.

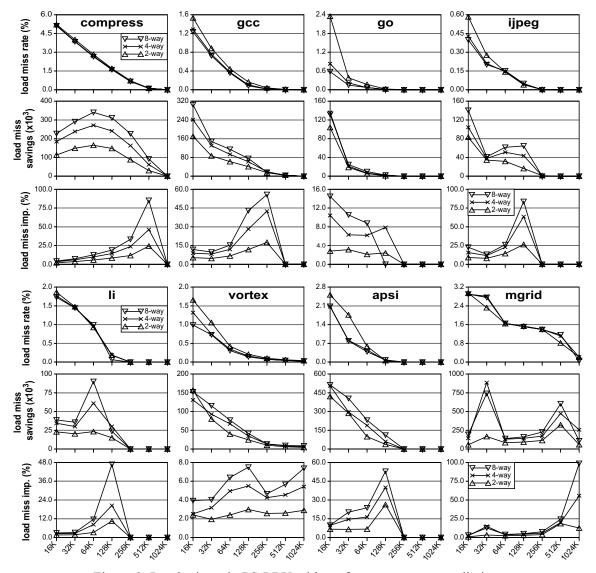


Figure 2: Load misses in PS-LRU with perfect access type prediction.

In most benchmarks, the load miss improvement slowly increases as the cache size increases and shows a peak before it quickly approaches zero. With smaller cache sizes, the load miss savings are high, but the load miss rate is also high, thus resulting in moderate load miss improvement. With bigger caches, such that the miss rate is lower, the improvement may show large peaks, especially near working set size transitions. When the cache is extremely large the improvement usually vanishes with the miss rate.

The savings and the improvement increase with cache associativity, because the replacement policy has more effect on caches with larger associativity. Table 2 shows the weighted arithmetic average of the load miss improvement across all the benchmarks. This table shows that, on the average, the rate of improvement increases with the cache size and associativity. Therefore the cost effectiveness of the hardware needed to implement penalty sensitive policies improves with the complexity of the cache.

	average load miss improvement (%)												
	16KB	32KB	64KB	128KB	256KB	512KB	1MB						
2-way	3.36	3.91	4.40	5.76	5.46	17.16	10.27						
4-way	5.33	9.49	8.21	10.04	9.20	20.30	45.05						
8-way	6.58	10.12	10.39	13.37	12.25	25.63	51.77						

Table 2: Average of load miss improvement rate by PS-LRU.

## 5. Access Type Prediction

A 100% accurate prediction is impossible to achieve. The next access type must be predicted either statically, dynamically or both (hybrid). Similar to other prediction schemes [3, 5, 12, 13, 26, 27], we attach next access type predictions to memory access instructions.

Figure 3 shows the motivation for attaching the next access type predictions to load and store instructions in the code. In the simple access sequence in Figure 3(a) a load and a store access the same block alternatively. If this pattern is very regular and common to many cache blocks, we can safely predict the next access type solely based on the PC of the memory access instruction. In this example, we just need to keep track of which type of memory access instruction follows the load or the store. If the PC of the current access to any block is 100 then the next access type to the same block is predicted as store, and if the current PC is 120 then the next access type is predicted as load. Note that the predictions for this sequence can be done easily at compile time, by profiling the execution for example.

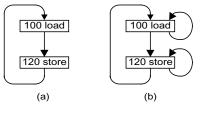


Figure 3: Examples of access type sequences.

In the second sequence in Figure 3(b) each instruction is repeated several times before moving to the other. This case is more complex to predict than the previous sequence and compiler-based prediction may fail here. (In general, sequences may be much more unpredictable; in some cases the next memory access to the same block may have many different PCs in a data dependent manner.)

Even for a very regular access pattern such as in Figure 3(b) a dynamic prediction scheme is needed and access history must be kept. For instance, suppose each instruction of Figure 3(b) executes twice per each iteration of the outer loop. After several iterations, the next access type history of PC 100 will be "010101", where '0' indicates a load and '1' indicates a store. From the history pattern we can predict that the next access by PC 100 will be a load. Note that in practice predictions must be made on cache blocks instead of individual words, and this can be hard for compilers to do.

In the following we develop simple static and dynamic access prediction schemes. In the dynamic case we present various design alternatives with cost consideration.

## 6. Static Prediction

We use a simple profiling scheme. We first run the program with a given data set and profile it. During the profiling stage, we count the number of times that the next access to the same block following each memory access instruction is either a load or a store. The static prediction for each memory access instruction is derived from the following two values: l the fraction of loads to the same block following the instruction, and T the threshold of acceptance. If l is greater than T, the prediction is load and the PC for that prediction is deemed *static*. If 1-l is greater than T, the prediction is store and the PC for that prediction is deemed *static*. Otherwise the instruction is not predictable and the PC is deemed *dynamic*.

Table 3 shows the static prediction accuracy of this simple approach in the case of an 8-way 128 KByte cache. To analyze the performance of the static prediction we use the following four metrics: (i) the fraction of memory instructions which are deemed static (column "pc"), (ii) the fraction of data references accessed by static memory instructions (column "ld/st"), (iii) the fraction of stores covered by static prediction (column "st"), and (iv) the fraction of stores excluding MRU hits which are accurately predicted (column "st-n"). The rationale for these latter numbers is that predictions are never needed after MRU hits and thus predictions following MRU hits are use-less.

program		T = 0	).999			T =	0.99		T = 0.95				
program	pc	ld/st	st	st-n	pc	ld/st	st	st-n	pc	ld/st	st	st-n	
comp.	93.0	78.0	75.9	6.9	93.5	78.7	75.9	6.9	94.7	88.7	90.3	18.1	
gcc	77.4	61.9	53.1	17.7	79.5	67.3	57.1	24.3	82.9	73.3	65.2	49.7	
go	75.5	53.1	53.2	20.8	78.2	60.6	55.3	21.4	83.3	70.5	58.4	23.2	
ijpeg	88.5	68.9	67.1	27.2	90.0	80.2	68.2	27.6	92.2	86.7	76.7	42.3	
li	80.6	63.6	61.2	26.7	82.5	72.0	65.6	57.3	83.9	76.9	69.3	62.0	
vortex	84.8	70.1	71.6	31.9	85.9	75.1	76.5	37.2	88.1	78.5	79.7	70.5	
apsi	90.1	73.6	73.7	33.2	90.9	76.0	74.1	33.2	92.4	79.5	75.1	34.1	
mgrid	88.9	76.0	69.9	7.72	91.2	87.8	69.9	7.7	93.6	94.1	70.8	7.8	
Avg.	81.3	68.7	66.9	18.0	83.0	72.3	68.6	23.5	86.1	74.4	70.1	32.5	

Table 3: Coverage by static prediction (8-way 128 KB cache).

To see the effect of the threshold *T*, we vary *T* from 0.999 to 0.95. We see that when *T* is 0.999 which allows less than 0.1% of errors, the weighted average across all the benchmarks shows that 81% of PCs are static and accurately cover 69% of memory references and 67% of stores. As we increase the error rate to around 5%, the static PCs cover up to 74% of memory references on average. Overall the coverage of stores is lower than the coverage of loads and the gap increases as the error rate increases.

A reverse interpretation of the prediction results in the table is that 19% of PCs show dynamic behavior and they cover 31% of memory references and 34% of stores when T is 0.999. Thus the number of dynamic PCs is far less than the number of static PCs but each dynamic PC covers more references and even more stores.

Furthermore, when considering the stores excluding MRU hits, the coverage by static prediction is extremely low especially in *compress* and *mgrid*. On average it ranges from 18% to 33%. This means that the static behavior is concentrated on MRU blocks. Accesses to non-MRU blocks and accesses with wide inter-reference gaps are more dynamic. Also this static behavior can vary widely with input data. This strongly suggests that dynamic prediction must be used to improve the coverage especially for stores excluding MRU hits although, in general, static prediction can be quite accurate.

We now apply the static predictions with 1% error rate to LRU. For dynamic PCs, the access type is predicted as load. This results in near perfect predictions of load access type and almost eliminates the cases that loads are mispredicted as stores. Figure 4 shows the load miss improvement by PS-LRU over LRU with this static prediction. To judge the overall performance of the static prediction, we compare Figure 2 (perfect prediction) with Figure 4 (static prediction) in the light of the coverage of stores that are not MRU hits shown in Table 3.

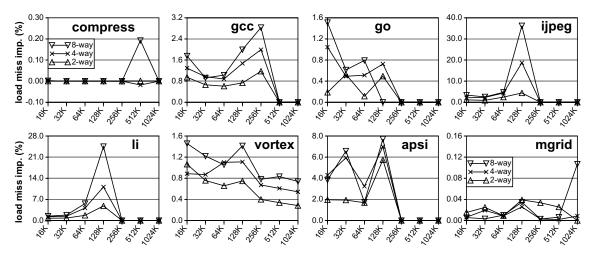


Figure 4: Load miss improvement by static prediction (T = 0.99).

*Compress* and *mgrid* show almost no improvement due to their low coverages. In general, curves for the load miss improvement with static prediction have the same shape as the curves for perfect predictions but the improvements are much lower. Notable exceptions are *li* and *vortex* in which the static prediction reaches 50% of the gains obtained with perfect prediction. We also observe that the improvements of static predictions are relatively better with small caches in *gcc*, *vortex* and *apsi*. This is because more stores are covered as the number of MRU blocks is less in smaller caches.

### 7. Dynamic Prediction

To predict the next access type dynamically we explore dynamic ATP (Access Type Predictor) structures based on two levels of access type history. Figure 5 shows the general structure of these dynamic ATPs. The hardware consists at most of three tables, PC table (PCT), access type history table (ATHT) and pattern history table (PHT). These structures are not very different from one-level or two-level branch predictors [27] except for the addition of the PCT at the frontend. PCT is indexed with the data block address and yields the PC used to last access the block and called the

Previous PC or PPC. The remaining tables maintain up to two levels of access type history. ATHT keeps track of the next access type history for each memory instruction and is indexed with the PPC, using a simple hashing function. Every time the next access is a "load" a zero is shifted in the ATHT entry. Every time the next access is a "store", a one is shifted in the ATHT entry. Each entry in PHT contains a saturating two-bit up-down counter, incremented every time the access is a store and decremented every time it is a load. The prediction outcome is store if the counter is greater than one.

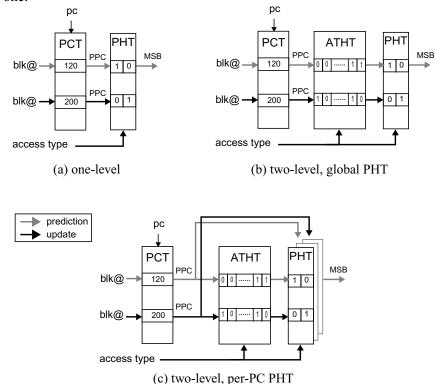


Figure 5: General structures of dynamic ATPs.

Four schemes are evaluated in this paper. The first dynamic ATP (Figure 5(a)) utilizes one level history with no ATHT. The second ATP (Figure 5(b)) maintains two levels of access type history by analogy with the structure of a PAg branch predictor. In this case, the size of PHT depends on the size of each ATHT entry. Aliasing can occur in PHT when the values of several entries in ATHT are the same. To reduce this aliasing PHT must be structured into multiple tables and these tables must be accessed with individual PC or set of PCs [27], leading to the third ATP (Figure 5(c)), which is identical to the second ATP but different PHT tables are used for different groups of PC. Thus the size of PHT tables may be large. The last ATP ("hybrid") is a hybrid scheme involving the static scheme and the per-PC PHT scheme of Figure 5, such that static predictions with 0.1% error rate override the predictions by the per-PC PHT ATP.

There are several possible implementations for PCT. It could be integrated into the data cache or could be a separate table. In the first case, the PPC field is simply added to the blockframe tag.

### 7.1. Prediction Updates and History Updates

The prediction update is a simple table lookup in PCT, ATHT (except for one-level), and PHT. The entry in PHT is then used to change the store subset in the cache. The PCT is updated on every cache access by storing the current PC into the PCT in a location indexed by the block address. To update the history, PCT is indexed by the current block address, and the PPC is read from the entry and indexes ATHT and/or PHT. The entry in ATHT and/or PHT is then updated.

Since replacements are invoked only upon cache misses, the latest time the store subset must be identified is right after a miss occurs. Predictions made right after an MRU hit are useless, because no replacement takes place then. MRU is particularly critical because most cache hits are on MRU blocks and these hits happen in streaks. So we only change the store subset on MRU changes.

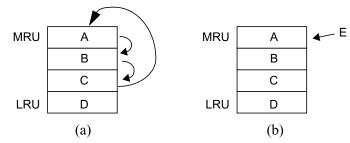


Figure 6: Timing of store group transitions.

MRU changes [20] happen when non-MRU blocks are hit or right after a cache miss. Figure 6 illustrates the situations. Upon a cache hit to a non-MRU block C as shown in Figure 6(a), MRU block A and second MRU block B are moved down while block C takes MRU position. Figure 6(b) shows the case that block E misses in the cache and takes MRU position. In both cases, the next access type of block A is predicted and the change of store subset is limited to block A which was previously the MRU block. In the case of Figure 6 (b) the victim is then selected based on the new store subset.

The second operation is to update the history in ATHT and PHT. Because predictions are made at the time of an MRU change, history modification on every cache access (including MRU hits) may introduce noise in the prediction. As we will see it is better to update the history for blocks that move into the MRU position at the time of an MRU change, such as blocks C and E in Figure 6.

### 7.2. Prediction Accuracy (Infinite Hardware)

In this section we assume that an infinite amount of hardware is available for each possible ATP structure so that no aliasing occurs. To select a particular prediction scheme, we measure the prediction accuracy of each access type (load or store) since they each have a different impact. Although both predictions are closely related to each other, the accuracy of store predictions affects the number of load miss savings opportunities whereas the accuracy on load predictions affects the number of accesses that are wrongly predicted as stores and hence increase the number of load misses.

### 7.2.1. Dynamic-ALL ATPs

We first consider ATP structures which update the history on every reference. We call these ATPs *dynamic-ALL ATPs*. Table 4 shows the misprediction rate per access type for every reference and

program		Dynamic-ALL with MRU hits									Dynamic-ALL without MRU hits							
	one-level		global PHT		per-PC PHT		hybrid		one-level		global PHT		per-PC PHT		hybrid			
	ld%	st%	ld%	st%	ld%	st%	ld%	st%	ld%	st%	ld%	st%	ld%	st%	ld%	st%		
compress	1.60	4.46	1.55	3.73	1.52	3.42	1.52	3.41	11.08	51.26	13.34	53.52	13.44	49.82	13.44	49.81		
gcc	3.60	7.35	3.36	6.67	2.35	6.53	2.35	6.40	4.07	27.46	4.35	27.57	3.40	30.20	3.40	30.00		
go	4.09	15.36	4.20	16.41	3.45	14.01	3.45	13.93	5.58	41.79	5.76	45.40	5.45	42.85	5.45	42.78		
ijpeg	1.55	3.20	1.32	3.17	1.05	2.97	1.05	2.91	12.60	41.40	12.82	41.21	12.66	40.84	12.66	40.77		
li	4.73	6.69	4.19	6.77	2.99	4.85	2.99	4.83	7.65	28.83	6.25	28.09	5.37	28.11	5.37	28.06		
vortex	3.37	6.46	1.79	3.24	0.93	2.12	0.93	2.07	1.05	16.39	1.02	15.55	0.84	15.92	0.84	15.40		
apsi	5.24	9.62	4.17	7.96	3.16	6.18	3.16	6.15	7.90	40.46	6.68	37.24	6.05	35.95	6.05	35.92		
mgrid	0.59	8.44	0.59	8.19	0.38	8.20	0.58	8.15	11.15	52.99	11.18	52.88	11.18	52.92	11.18	52.90		
Average	3.01	7.53	2.43	6.09	1.80	5.03	1.80	4.97	6.45	40.46	6.52	40.03	6.22	39.31	6.22	39.22		

for the references excluding MRU hits in 128 KBytes eight-way cache and assuming infinite hardware.

Table 4: Misprediction rate by dynamic-ALL ATPs (128KByte 8-way cache; infinite hardware).

First we focus on the misprediction rate including MRU hits (left part of Table 4). Overall, the prediction accuracy is excellent for all prediction schemes. As expected the ATPs using a two-level history outperform the APT with one-level history. From the comparison between the second (global PHT) and the third (per-PC PHT) schemes, we observe that there is sizable room for improvement if the aliasing on PHT is eliminated. The predictions by the hybrid scheme are slightly better than the per-PC PHT scheme. Overall the misprediction rates of stores are higher than the misprediction rates of loads.

If we focus now on the prediction on non-MRU hits only (right part of Table 4), the misprediction rate is much higher across all the benchmarks, especially for stores. This is because stores have higher hit rate on MRU blocks than loads especially when compilers with high optimization levels (such as the one we use) attempt to cluster stores. As a result, the more accurate predictions for MRU hits are removed. Unfortunately, the numbers without MRU hits in Table 4 are more indicative of the performance of the predictor for the problem addressed in this paper.

### 7.2.2. Dynamic-MRU ATPs

In this section we focus on dynamic ATPs which update history and predict access type on MRU changes only. We call these *dynamic-MRU ATPs*.

Table 5 shows their misprediction rates in 128 KBytes 8-way cache. This data shows that, on average, the predictions for non-MRU accesses are noticeably improved by dynamic-MRU ATPs. One-level history outperforms two-level history with global PHT and is also almost as good as two-level per-PC PHT. The predictions by the hybrid scheme are slightly better than per-PC PHT, as in dynamic-ALL.

	Dynamic-MRU											
program	one-level		globa	PHT	per-PC	C PHT	hybrid					
	ld%	st%	ld%	st%	1d%	st%	ld%	st%				
compress	8.12	48.60	8.50	55.32	8.50	53.19	8.48	53.47				
gcc	3.61	19.62	4.62	20.96	2.20	30.14	2.20	29.11				
go	3.13	33.29	3.07	37.31	2.19	41.24	2.19	40.96				
ijpeg	7.40	25.08	9.05	25.04	6.67	25.05	6.67	24.70				
li	5.62	22.37	4.74	23.84	2.00	24.45	1.99	24.11				
vortex	0.71	11.09	0.68	11.57	0.55	15.78	0.55	14.58				
apsi	3.14	14.48	3.52	13.40	2.19	11.02	2.19	10.96				
mgrid	10.75	34.23	10.39	34.74	9.72	34.88	9.72	34.88				
Average	4.64	25.67	4.78	26.92	3.92	27.83	3.92	27.56				

Table 5: Misprediction rate by dynamic-MRU ATPs (128 KByte 8-way cache; infinite hardware).

The number of MRU blocks in the cache has an impact on history updates in the case of dynamic-MRU. Figure 7 shows the misprediction rate by two dynamic-MRU ATPs as the number of MRU blocks (or cache sets) varies from 64 to 16 K by changing the size of an 8-way cache. The misprediction rates mostly increase with the number of MRU blocks since more references are filtered out by MRU hits such that remaining references are more sparse and difficult to predict. The data also indicates that the two-level scheme is not very effective as compared to the one-level scheme.

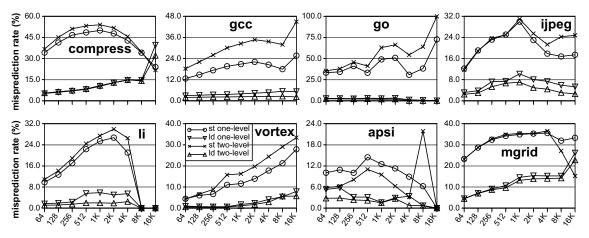


Figure 7: Misprediction rate by dynamic-MRU ATPs by set count.

Figure 8 shows the load miss improvement by PS-LRU with two dynamic-MRU ATPs. Overall the improvement rates are very close, with a few exceptions. In *apsi*, the two-level ATP performs better due to better predictions for both loads and stores as shown in Figure 7. The data shows very low or even negative improvements at several design points especially with eight-way caches. These results are very different from the results with perfect predictions in which eightway caches yield large improvements. Negative improvements in *ijpeg* and *apsi* are mostly due to cache conflicts by several blocks whose access type is wrongly and consistently predicted as store and accessed alternatively many times. If this happens, the caches with larger associativity suffer more from the conflicts by leaving other blockframe in a set under-utilized. As the cache size increases these conflicts are reduced and PS-LRU improves.

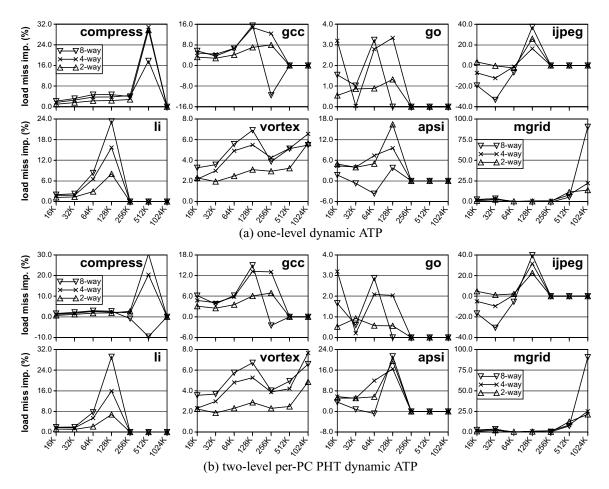


Figure 8: Load miss improvement rate in PS-LRU with dynamic-MRU ATPs (infinite hardware).

In summary, we observe that one-level does as well as two-level per-PC and outperforms twolevel global. This advocates the use of the one-level scheme since its implementation cost is significantly lower than the cost of the two-level scheme.

### 7.3. One-level Dynamic-MRU ATPs with Finite Hardware

With finite hardware, integrating PCT with the cache in the one-level scheme is simpler and ensures that we at least have the PPCs for all blocks present in the cache, which are the most likely blocks to be accessed next. When the PCT is separate it may not contain the PPCs of all the blocks in the cache, unless its organization is such that it includes the cache.

When the PCT is integrated into the cache, its number of entries is the number of cache blocks and the PPC is not available for blocks which are not currently in cache. Thus we cannot update the history upon cache misses, we can only update it on non-MRU hits, and the prediction accuracy may be degraded. Moreover in the case of an access to a block just replaced from a near-MRU position due to a store prediction the history cannot be updated because we do not have the PPC and the prediction remains unchanged. Of course, the smaller the cache, the worse these effects are.

To solve this problem, we need to allocate more memory for PCT. Keeping with the idea that it is advantageous to match the content of the PCT with the cache content, we add PCT entries in each cache set for blocks that have just been replaced. We call this approach *extended PPC directory* (EPD), which is similar to the shadow directory idea proposed in [4] and [24]. The shadow directory can be to help smart prefetching. In this paper, the role of the EPD is to support penalty-sensitive replacement policies. The EPD is physically implemented as a part of cache but each entry consists of only a PPC field and the block address tag, which is part of the directory.

It is interesting to note that PCT now serves a dual purpose as it can also be used to prefetch. We have not explored the use of the PCT to prefetch selectively based on access type prediction. These open possibilities show the advantage of merging PCT with the cache.

When a block is replaced, the PPC value is moved into an entry of EPD. EPD is maintained by LRU. If an entry for a block missing in the cache set is found in EPD, the history is updated with the matching PPC. Otherwise the history update is simply skipped. In our evaluation, the number of entries in EPD is identical to the number of blocks in a cache set. Thus the hardware overhead of EPD goes down with the block size. To find out how many bits should be in the PPC field, we evaluate the effect of the number of entries in PHT. Figure 9 shows the weighted average of the misprediction rates across all the benchmarks by one-level dynamic-MRU ATPs with and without EPD for various number of PHT entries vary. The prediction with EPD is always better for both load and store predictions.

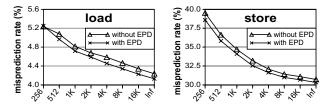


Figure 9: Misprediction rate by one-level dynamic-MRU ATP.

Figure 10 shows the improvement rate on load misses in PS-LRU with a PHT of 8K entries. Therefore PPC is 13 bit long and holds in two bytes. Overall the system with EPD outperforms the system without EPD as expected. Also the performance of dynamic prediction with EDP is very close to the performance of the dynamic-MRU ATP with infinite hardware shown in Figure 8. Exceptions are *gcc* and *go* in which the number of distinct PCs are significantly larger than others. *Li* and *vortex* perform well in both cases.

#### 7.4. Memory Traffic

PS-LRU affects the traffic between the data cache and lower memory. The traffic is incurred by the following three components: load misses, store misses and write-backs from the cache.

Figure 11 shows the traffic in a four-way cache. The first bar shows the decomposed traffic for LRU. The other bars show the traffic for PS-LRU normalized to the traffic for LRU with different

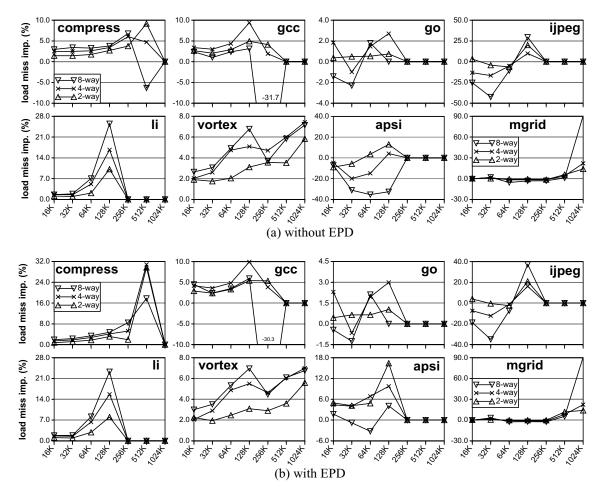


Figure 10: Load misses in PS-LRU by one-level dynamic ATP with 8K-entry PHT.

predictors. "Perfect" is for perfect prediction, "static" is for static prediction, "L2-inf" is the 2level per-PC PHT scheme with infinite hardware, "L1-inf" is the one-level history scheme with infinite hardware, and "L1-8K" is the one-level history scheme with EPD and 8K PHT entries.

The number of store misses is of course increased in order to save load misses. In most benchmarks, the increased rate of store misses is higher than the reduction rate of load misses. In some benchmarks it takes three to ten store misses to save one load miss. This is mainly because stores have more temporal locality in our benchmarks especially since we applied a high compiler optimization level. Thus the total traffic by PS-LRU is increased in most situations as compared to the traffic by LRU. However, this is not always the case. In *li* and *mgrid*, the total traffic by PS-LRU is even decreased in 128KByte cache indicating that the replacements of blocks in store subset are very effective at saving load misses. The write-back traffic also increases with the store miss traffic because, when a block is replaced from the cache due to a store prediction, the chance is high that the block is already dirty. Ideally, the total traffic by PS-LRU should not exceed the traffic by PS-LRU with perfect prediction. If it does, it is due to mispredictions or to conflicts incurred by several blocks whose access type predictions are stores. This situation occurs in *ijpeg* and *apsi*. Overall the traffic increase is below 20% except for a few cases.

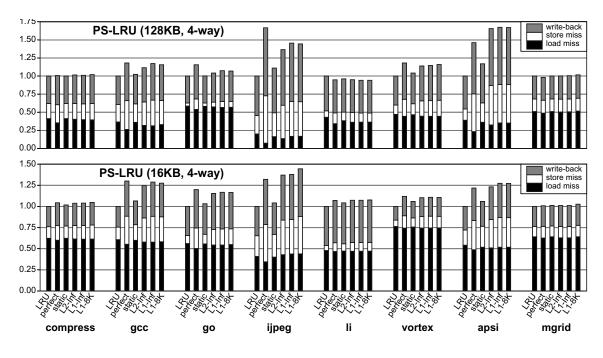


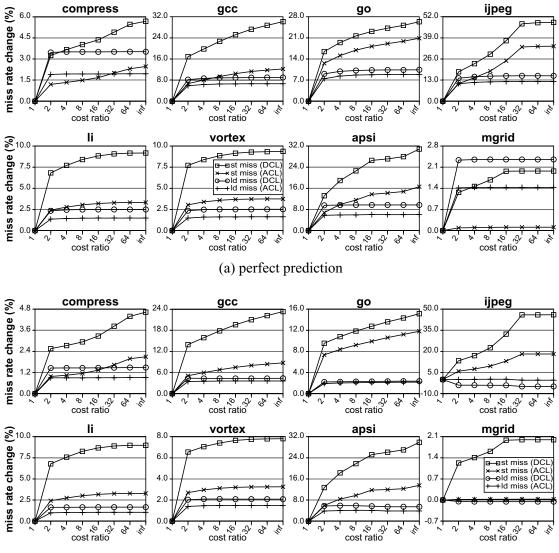
Figure 11: Normalized traffic in PS-LRU.

## 8. Injecting Finite Cost Ratios with DCL and ACL

So far, we have designed various access type predictors and evaluated them in PS-LRU. From the results of prediction accuracy and the load miss improvements, we conclude that the one-level dynamic-MRU ATP with EPD is the most cost-effective scheme among the various ATP schemes we have considered. PS-LRU reduces the number of load misses by replacing them with store misses whenever possible. This strategy reduces the total load/store miss penalty to a minimum if the penalty of store can be kept negligible. However the number of store misses increases and, with it, the memory traffic. To evaluate this trade-off further, we now apply DCL and ACL to the problem of load/store miss penalties. In DCL and ACL, the cost ratio between a load miss and a store miss is finite and replacement of block in load and store subset is competitive. As the load/ store miss penalty ratio goes down, we tend to replace more and more block in the load subset, based on the temporal locality of all blocks.

In this section, we vary the cost ratio *r* between load and store miss penalties from 2 to infinite to understand how DCL and ACL suppress the increase of store misses while reducing the number of load misses (note that DCL with *r* infinite is equivalent to PS-LRU.) We apply two ATPs (the perfect predictor and the one-level dynamic-MRU predictor with EPD and 8K-entry PHT) to DCL and ACL and compare their performance. We focus on a 16-KByte 4-way cache and avoid larger cache sizes where huge load miss improvements are obtained with very low miss rates.

Figure 12 shows the load miss improvements and the relative increase of store misses by DCL and ACL over LRU as a function of r the cost ratio. The relative increase in the number of store misses is the ratio between the increase of the number of store misses (over LRU) and the number of load misses in LRU. In this way, the reduction of load misses and the increase of store misses are on the same scale and they can be compared directly.



(b) one-level Dynamic-MRU with 8K-entry PHT

Figure 12: Relative miss rate changes by DCL and ACL (16-KByte 4-way cache).

We first focus on the case of the perfect predictor in Figure 12(a). The graphs show that the load miss improvements by DCL quickly saturate after r = 2 in all benchmarks. On the other hand, the relative increase in store misses steadily grows with the cost ratio, except for a few benchmarks. With r infinite, the relative increase in store misses reaches almost up to three times the load miss improvements except for *compress* and *mgrid*. In contrast, when r is small, DCL effectively suppresses the increase in store misses in all benchmarks while achieving load miss improvements comparable to the improvements with infinite cost ratio. This result strongly indicates that stores are more clustered than loads such that additional load miss savings becomes much more difficult as the cost ratio increases. Overall, the results indicate that DCL can effectively reduce the number of load misses with a minimal increase in store misses by injecting a small cost ratio between loads and stores in the replacement policy.

The graphs also show that ACL is very effective in suppressing the increase in store misses

while the load miss improvement is marginally reduced. In *mgrid*, the increase in store misses is extremely small as compared to the reduction of load misses.

In the case of the one-level ATP, the miss rate changes are scaled down but the shape of the graphs follows the shape of the perfect prediction, except for *ijpeg* and *mgrid*. In *ijpeg* and *mgrid*, ACL has almost the same performance as LRU, while avoiding the negative improvements observed in DCL.

Table 6 shows the reservation (RV) rate and the reservation success (RVS) rate in DCL and ACL with r = 2. The RV rate is the fraction of replacements in which the LRU block is reserved, and the RV success rate is the fraction of reservations that are eventually successful. Overall the RV rate and the RV success rate are very low mainly due to a large fraction of loads in all applications. These low rates are directly translated into low load miss improvements. With the perfect ATP, ACL as compared to DCL effectively cuts fruitless reservations by yielding lower RV rates but higher RV success rates across all benchmarks especially in *mgrid*. With the one-level ATP, both the RV rate and RV success rate are reduced due to the misprediction of access type. In *mgrid*, the performance of ACL is exceptional in suppressing fruitless reservations.

Overall, we observe that injecting small cost ratio instead of infinite cost ratio can be very advantageous. ACL yields very reliable performance across all benchmarks by suppressing fruitless reservations.

			comp	gcc	go	ijpeg	li	vortex	apsi	mgrid
	RV rate	DCL	15.3	22.3	20.5	20.1	13.4	10.3	23.8	13.0
perfect	KV Tate	ACL	6.5	11.5	16.4	13.8	6.0	4.6	11.7	2.0
ATP	RVS rate	DCL	8.2	16.6	26.8	25.7	8.7	9.9	13.9	9.0
		ACL	11.9	25.5	28.3	29.4	11.9	15.0	16.3	39.2
	RV rate	DCL	10.8	22.0	17.2	24.3	11.4	10.7	24.8	14.0
one-level	KV rate	ACL	5.1	11.1	13.8	13.3	5.3	4.8	11.0	0.4
ATP	RVS rate	DCL	6.8	15.4	24.7	15.1	8.2	10.3	9.7	0.3
	KV 5 Tale	ACL	8.6	23.3	25.9	20.6	11.1	16.4	12.9	6.9

Table 6: RV rate and RVS rate in DCL and ACL (r = 2).

## 9. Related Work

There is a vast body of literature on caches. We briefly overview here the work that is most relevant. Previous papers to improve basic cache replacement mainly focus on reducing the miss count. One common approach is to identify the type of locality. Gonzalez et al. [6] proposed the use of independent caches based on the locality type. Wong and Baer [26] proposed instruction-based prediction schemes to predict the locality per cache block. PCs are physically associated with cache blocks and used to index the locality history table. Blocks which do not show locality are considered first for replacements over others, however MRU blocks are never replaced. Tyson et al. [25] proposed cache bypassing. In their scheme, memory operations which generate many misses are first identified. Then the cache blocks accessed by those memory operations bypass the caches so that blocks with high locality stay in the cache longer. Overall the savings opportunities sought by these schemes are different than the ones sought by penalty-sensitive algorithms and hence these schemes can be combined together with our algorithms to further reduce load misses.

Mounes-Toussi and Lilja [14] evaluated state-based cache replacement algorithms under the MESI protocol. They found that a certain static replacement priority based on cache coherence states with Random policy shows marginal improvement over Random policy. In the context of non-uniform costs in caches, Jeong and Dubois [7] first proposed optimal cost-sensitive replacement algorithms and evaluated optimal miss cost savings in multiprocessor systems where miss costs differ by physical memory mapping. Later they proposed several realistic on-line algorithms based on LRU [8][9].

The prediction of next access type is also addressed in other papers for different purposes. Mowry [15] proposed exclusive mode prefetches in multiprocessors to save separate ownership requests if the prefetched blocks will be written next. These perfetches are determined statically through compiler analysis. The prediction of migratory sharing [12][23] is closely related to the prediction of store access type. Many prediction schemes [3][5][12][13][25][26] rely on instructions to capture program behavior. Johnson and Hwu [10] proposed the use of macroblocks based on the addresses of memory references to determine fetch size. Mukherjee and Hill [16] applied two-level branch predictors to the predictions of next coherence messages. So and Rechtschaffen [20] looked at the effects of accesses to MRU blocks. They observed that the working set changes whenever the MRU block changes and accesses to MRU blocks dominates accesses to non-MRU blocks in various kinds of programs.

## **10.** Conclusion

This paper introduces and evaluates simple penalty-sensitive cache replacement algorithms, motivated by the observation that the penalty of stores is mostly hidden in modern processors. Thus the idea is to replace some load misses with store misses. Various common replacement policies such as LRU, PLRU and Random can be easily sensitized to access penalties using this simple idea. In this paper we focus on PS-LRU (Penalty-Sensitive LRU). The new policies rely on predicting the next access type to each block. We have explored perfect prediction, static prediction and instruction-based dynamic access type predictors (ATPs).

The results for perfect prediction show the promise of considerable load miss improvements using our simple policies. Moreover, the prediction of the next access type can be very accurate in general. Unfortunately most of the accurate predictions are done on MRU hits and these predictions are useless for replacements. Nevertheless, we have explored three types of dynamic ATPs, mostly inspired from branch predictors and found the best configurations to reap the maximum average benefits across eight benchmarks. We were able to obtain spectacular results in some configurations, and low-to-moderate results in many cache configurations. But unfortunately, a few configurations show negative results. Of course this is somewhat expected in replacement policies and we should look at the average picture. However, there is room for improvement. One way is to understand the reason for the bad observations and tune the policy to avoid them. Another direction of research is to explore new, different ATPs to avoid the negative results observed in some configurations.

One concern of course is the increased memory traffic. In some cases the traffic increase is high. To solve this problem one could exploit the write cache for example. If the write cache is large enough then blocks in the write cache can be reclaimed quickly in the main cache, so that the store miss rate to memory and the traffic to memory are both cut. In this paper, we have also considered finite cost ratios between store misses and load misses to utilize cost-sensitive replacement algorithms and achieved reasonable increase of store misses while the reduction of load misses are similar as for PS-LRU. To reduce the amount of store miss traffic, we have also applied previously proposed replacement algorithms called DCL and ACL. These algorithms provide a trade-off between complexity and memory traffic.

We have focused on a write-back data cache. If write-through caches are considered, the penalty model needs to be changed since a no-allocate policy on write misses is commonly used. Our policies could also be extended to lower-level caches in systems with cache hierarchies. The prediction scheme can be common to all levels of caches. The major overhead is to pass the PC to the lower level caches. With the trend of integrating meta data for several levels of caches inside the processor, PCs are visible to the caches. Since an L2 cache is usually combined and instructions are read only, it is possible that instructions stay longer in L2 cache. In this case the policies should be further extended to consider instructions separately from data as an "instruction subset" for instance and treat them in different way.

In our study, dynamic ATPs are dedicated to improving cache replacements by treating stores and loads differently. In this context, the role of ATPs can easily be extended to capture locality information on block accesses especially when PHT is incorporated with the cache. In this case, the locality information is exploited on top of our algorithms and used to override LRU in selecting a victim in the load subset to further reduce load misses or to prefetch selectively based on next access type and locality.

Additionally, ATPs can be used for other purposes especially in the context of multiprocessors. With the knowledge of the next access type, prefetches could be issued to reduce the cost of ownership transfer. Similarly, if each memory request is tagged with its next access type, home nodes or remote nodes serving the request can optimize future actions in advance not only by knowing the next access type following the external request but also by predicting its own next access type to the same block. By doing this, migratory sharing and invalidation actions can be better optimized.

Finally, to optimize cache behavior and implement better penalty-sensitive cache policies, the actual penalty of each memory access instruction should be monitored and predicted so that cache replacement and prefetching can be tuned accordingly. This is a hard problem, much harder than the one we have addressed here. However we feel that the work presented here and the work by others [1][17] are initial steps in that direction.

### References

- [1] S. Abraham, R. Sugumar, D. Windheiser, B. Rau, and R. Gupta, "Predictability of Load/Store Instruction Latencies," in *Proceedings of the 26th International Symposium on Microarchitecture*, pp. 139-152, December 1993.
- [2] D. Burger and T. Austin, "The SimpleScalar Tool Set, Version 2.0," *Computer Sciences Dept. Tech. Report #1342*, Univ. of Wisconsin-Madison, June 1997.
- [3] T. Chen, "Data Prefetching for High-Performance Processors," Ph.D. Dissertation, University of Washington, July 1993.
- [4] T. Collins and D. Tullsen, "Hardware Identification of Cache Conflict Misses," in Proceedings of the 32nd International Symposium on Microarchitecture, pp. 126-135, Nov. 1999.

- [5] J. Fu, J. Patel, and B. Janssens, "Stride Directed Prefetching in Scalar Processors," In Proceedings of the 25th International Symposium on Microarchitecture, pp. 102-110, Dec. 1992.
- [6] A. Gonzalez, C. Aliagas, and M. Valero, "A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality," In *Proceeding of International Conference on Supercomputing*, pp. 338-347, July 1995.
- [7] J. Jeong and M. Dubois, "Optimal Replacements in Caches with Two Miss Costs," in Proceedings of the 11th ACM Symposium on Parallel Algorithms and Architectures, pp. 155-164, June 1999.
- [8] J. Jeong, "*Cost-Sensitive Cache Replacement Algorithms*," Ph.D. Dissertation, University of Southern California, May 2002.
- [9] J. Jeong and M. Dubois, "Cache Replacement Algorithms with Nonuniform Miss Costs," *IEEE Transactions on Computers*, vol. 55, no. 4, pp. 353-365, April 2006.
- [10] T. Johnson, M. Merten, and W. Hwu, "Run-time Spatial Locality Detection and Optimization," in *Proceedings of the 30th International Symposium on Microarchitecture*, Dec. 1997.
- [11] N. Jouppi, "Cache Write Policies and Performance," in *Proceedings of the 27th International Symposium on Computer Architecture*, pp. 191-201, May 1993.
- [12] S. Kaxiras and J. Goodman, "Improving CC-NUMA Performance Using Instruction-Based Prediction," in *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, pp. 161-170, Jan. 1999.
- [13] A. Lai and B. Falsafi, "Selective, Accurate, and Timely Self-Invalidation Using Last-Touch Prediction," in *Proceedings of the 27th International Symposium on Computer Architecture*, pp.139-148, June 2000.
- [14] 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, Aug. 1998.
- [15] T. Mowry, "Tolerating Latency Through Software-Controlled Data Prefetching," Ph.D. Dissertation, Stanford University, Mar. 1994.
- [16] S. Mukherjee and M. Hill, "Using Prediction to Accelerate Coherence Protocols," in Proceedings of the 25th International Symposium on Computer Architecture, pp. 179-190, June 1998.
- [17] A. Seznec and F. Lloansi, "About Effective Cache Miss Penalty on Out-of-Order Superscalar Processors," *IRISA Report #970*, Nov. 1995.
- [18] K. Skadron and D. Clark, "Design Issues and Tradeoffs for Write Buffers," in Proceedings of the 3rd International Symposium on High-Performance Computer Architecture, pp. 144-155, Feb. 1997.
- [19] A.J. Smith, "Cache Memories," ACM Computing Surveys, vol. 3, pp. 473-530, Sept. 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. Srivivasan, R. Ju, A. Lebeck and C. Wilkerson, "Locality vs. Criticality," in *Proceedings* of the 28th International Symposium on Computer Architecture, pp. 132-143, July 2001.

- [22] Standard Performance Evaluation Corporation, http://www.specbench.org.
- [23] P. Stenström, M. Brorsson, and L. Sandberg, "An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing," in *Proceedings of the 20th International Symposium on Computer Architecture*, pp. 109-118, May 1993.
- [24] H. Stone, *High-Performance Computer Architecture*, 2nd Edition, *Addison-Wesley Publishing Company*, Nov. 1990.
- [25] G.S. Tyson, M. Farrens, J. Matthews, and A. Pleszkun, "A Modified Approach to Data Cache Management," in *Proceedings of the 28th International Symposium on Microarchitecture*, pp. 93-103, Dec. 1995.
- [26] 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, Jan. 2000.
- [27] T. Yeh and Y. Patt, "Alternative Implementation of Two-Level Adaptive Branch Prediction," in *Proceedings of the 19th International Symposium on Computer Architecture*, pp. 124-134, May 1992.