Modified LRU Policies for Improving Second-level Cache Behavior

Wayne A. Wong and Jean-Loup Baer
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195
{waynew,baer}@cs.washington.edu

Abstract

Main memory accesses continue to be a significant bottleneck for applications whose working sets do not fit in second-level caches. With the trend of greater associativity in second-level caches, implementing effective replacement algorithms might become more important than reducing conflict misses. After showing that an opportunity exists to close part of the gap between the OPT and the LRU algorithms, we present a replacement algorithm based on the detection of temporal locality in lines residing in the L2 cache. Rather than always replacing the LRU line, the victim is chosen by considering both its priority in the LRU stack and whether it exhibits temporal locality or not.

We consider two strategies which use this replacement algorithm: a profile-based scheme where temporal locality is detected by processing a trace from a training set of the application and an on-line scheme where temporal locality is detected with the assist of a small locality table. Both schemes improve on the second-level cache miss rate over a pure LRU algorithm, by as much as 12% in the profiling case and 20% in the dynamic case.

1 Introduction

Until recently, caches were automatically managed by the hardware. With the growing gap between processor speed and memory latency, techniques to hide or tolerate memory latency have been developed. Some of them are purely hardware-based, such as lock-up free caches, and are quite successful for hiding the latency between the on-chip cache (L1) and a closely integrated second-level cache (L2). Others are software-oriented, such as code and data reorganization to increase cache efficiency, and require the cache to be exposed to the user. Finally, some latency hiding techniques can use both hardware and software support: this is the case for prefetching.

Most research to date has concentrated on the L1-L2 interface. Techniques for reducing the cost of accessing main memory, i.e., the L2 to main memory interface, have received much less attention. While this may be of lesser importance for applications that fit in relatively small L2 caches, such as those similar to SPEC95int or desktop applications [Lee 98], it is an important issue for commercial and database applications [Perl & Sites 96, Barroso *et al.* 98]. In their study of the performance of the Pentium Pro, Bhandarkar and Ding specifically point out that “during L2 misses, the CPU can run out of other machine resources causing back pressure on earlier pipe stages” [Bhandarkar & Ding 97].

A simplistic but telling way of illustrating the importance of L2 misses is as follows. Assume a processor and associated L1 cache with an IPC (number of instructions executed per cycle) of 4 (CPI=0.25) when there are no L2 cache misses. This goal could be reasonably achieved by an 8-way out-of-order processor, 32 KB L1 I-cache and 32 KB lock-up free L1 D-cache with a L1-L2 latency of 6 cycles. Assume now that we have a L2 global cache miss rate of one miss every 100 instructions (0.01 MPI). Then, if the memory latency is 100 cycles, the IPC decreases to 0.8. If we were able to reduce the L2 miss rate by 10% (0.009 vs. 0.010 MPI), the IPC would increase from 0.8 to 0.87, a 9% increase. As the relative memory latency continues to increase, improvements to L2 cache performance result in corresponding improvements in overall system performance. Thus, spending some effort in reducing the L2 miss rate is clearly important considering the current trends.

In this paper, we propose a L2 replacement policy that increases the L2’s effectiveness. The basic philosophy is to modify the standard LRU (least recently used) replacement algorithm so that lines that exhibit temporal locality, i.e., that have a high probability of being reused in the near future, are not replaced as readily as those that do not appear to exhibit temporal locality. We demonstrate two techniques which classify instructions’ temporal locality and use this replacement policy. The first one statically classifies instructions exhibiting temporal

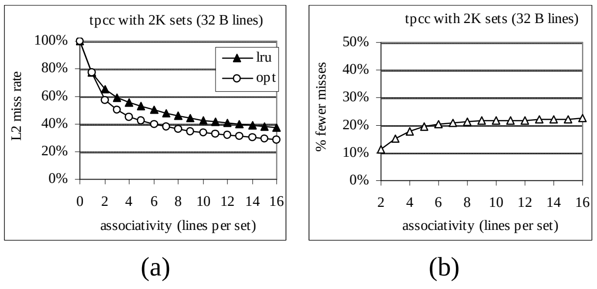

Figure 1. Comparison of LRU and OPT replacement policies. Graph (a) shows the L2 data miss rates (L2 misses/L2 references) for both LRU and OPT replacement policies with various cache sizes and a mapping of 2 K sets. Graph (b) shows the percentage fewer misses that OPT exhibited than LRU. Each curve in both graphs has constant cache line size with increasing number of lines per set. The trace has been previously filtered with a 16 KB, 4-way set associative L1 cache with 32 byte lines.

locality by using profiling. The second strategy uses dynamic detection of temporal locality via hardware assist structures at run-time.

The rest of this paper is as follows. In the remainder of this section we further motivate our study by showing that a margin of improvement exists between the current least recently used (LRU) based replacement algorithms and the optimal replacement algorithm. We also review previous work in the area of optimal algorithms and profiling studies. Section 2 describes the new replacement algorithm and the two techniques to classify temporal locality. Section 3 presents trace driven simulation results showing the improvements in L2 cache miss rates that can be achieved by using these two strategies. Section 4 summarizes and suggests areas of further study.

1.1 Motivation

A current trend in the design of second-level caches is an increase in capacity and associativity. With non-blocking loads and high miss latencies, L2 cache lookup is not as time critical, thus, enabling larger cache associativity. With L2 caches becoming 4-way or 8-way set associative (and maybe even more associative in the future), strategies that emphasize the reuse of lines already present in the cache might be more important than those that target conflict misses. With associative caches and more corresponding victim choices, it might be time to revisit the conventional wisdom that cache replacement algorithms are not much of a performance factor.

The most popular replacement algorithm is LRU and it generally performs well. LRU and approximations to it are practical for reasonable set sizes [Smith 82]. However,

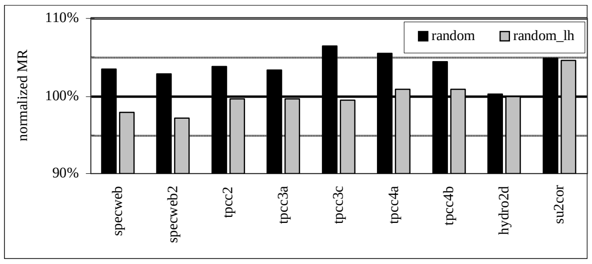

Figure 2. Comparing LRU and random replacement policies. This graph shows the performance of two random policies, random and random_lh, compared to LRU. Each bar represents the number of L2 misses normalized to LRU. Above 100% means the policy had more misses than LRU. The L2 cache is 256 KB, 8-way set associative with 32 byte lines.

some recent studies [Sugumar & Abraham 93] have shown that in the case of larger associativities there is noticeable room for improvement between LRU and optimality as in the off-line MIN [Belady 66] or the equivalent OPT algorithms [Mattson *et al*. 70]. Increasing the number of lines per set can increase the probability that the LRU line is not the optimal victim.

To illustrate this latter point, we plot in Figure 1 (a) the L2 data cache miss rates for the *tpcc* application (see Section 3 for the description of our benchmarks) with the OPT and LRU replacement algorithms. All data reported in this paper assumes the references have been filtered by a L1 data cache. We consider a fixed number of sets (2K) with increasing set size. We vary the associativity from direct mapped to 16-way set associative. Figure 1 (b) shows the percentage of fewer misses incurred when OPT is used. For example, with a line size of 32 B and a set-associativity of 4, a 256 KB cache (2K * 4 * 32 = 256K) has a miss rate of approximately 56% under LRU and 46% under OPT (Figure 1a) resulting in 18% fewer misses with OPT (Figure 1b). Looking at Figure 1 (b), we see that for this application the potential margin of improvement over LRU is in the range of 11 to 23%. The margin tends to increase with increased associativity and lower absolute number of misses.

LRU approximates OPT closely at low associativity because there are few victims to choose from. With more flexibility in victim choices permitted by larger set sizes, LRU does not take advantage of the flexibility to select better victims. To further illustrate this behavior, we compared two random replacement policies to LRU. The first random policy, *random*, chooses its victim randomly from all lines in the set. The second random policy, *random_lh*, chooses its victim randomly from the lower LRU half of the set. For example, with an 8-way set associative cache, *random_lh* chooses its victim randomly from the four least recently used lines. As shown in Figure 2, LRU performs better than the pure *random* policy. With some of the applications though, *random_lh* actually does better than LRU. This implies that below some LRU depth, the LRU depth is not a good predictor of the likelihood that a cache line will be reused.

Clearly, opportunity to improve over LRU exists. Our goal is to enhance LRU to approximate OPT more closely by modifying the choice of the victim line based on static or dynamic detection of temporal locality.

1.2 Related work

Optimal replacement algorithms date back to the introduction of paging systems. Belady's MIN algorithm [Belady 66] is an off-line algorithm that gives the minimum number of page faults for a given program. Until recently, efficient implementations of MIN required two passes over the input string. A good example is the OPT algorithm [Mattson 70] which, like LRU, has the "stack" property. By using limited look-ahead windows and correcting the contents of the stack when necessary, a "one-pass" algorithm can be devised for fully-associative and set-associative caches [Sugumar & Abraham 93]. Sugumar and Abraham also show that there is a gap between the cache hit rates under OPT and LRU and that the miss rates for the former can be lower by as much as 32% for 2-way set-associative caches and 70% for fully-associative caches. LRU works well in general but besides the fact that there is a gap between LRU and OPT there also exist situations where LRU is not the algorithm of choice [Smith & Goodman 85].

OPT has the great advantage of relying on the knowledge of future references while LRU is restricted to the recent past reference history. Profiling can be used to approximate this future knowledge. There are many examples of effective use of profiling to make static changes to a program in order to optimize a program's reference behavior. Profiling with MIN analysis has been used to annotate instructions for prefetching [Abraham *et al.* 93]. Other techniques restructure the layout of references, both data [Lebeck & Wood 94, Calder *et al.* 98] and instructions [Pettis & Hansen 90, Hashemi *et al.* 97]. However, the majority of the restructuring techniques have the goal of eliminating conflict misses in direct-mapped caches while our interest is in caches with larger set-associativity.

Variations on LRU [Smith 82] have been extensively studied and recently there has been renewed interest in bypassing, a mechanism which loads directly data in registers (or an assist buffer) without modifying the contents of the primary cache. Several techniques for bypassing, based on the reuse of cache lines and associated hardware assists, have been proposed. Lines with low probability of reuse are subject to bypassing, thus increasing the cache residency of those lines with higher temporal locality. The method for classifying data reuse can be based upon instruction or data address. CNA is an instruction-based reuse classification scheme which can be used with [Rivers *et al.* 98] or without an assist cache [Tyson *et al.* 95]. Data address classification strategies such as MAT [Johnson & Hwu 97] or NTS [Rivers *et al.* 98] can also be effective. Here also, most of these strategies were targeted at first-level caches or removing conflict misses in caches with limited associativity. With increased cache associativity, conflict misses are less significant and, thus, bypassing becomes less effective. Also, statically segregating the cache can lead to less efficient use of the cache area when compared to a single cache structure.

2 Enabling better victimization

In the previous section we showed that there is a noticeable opportunity to improve over LRU. The replacement algorithm that we propose is based upon LRU and enables more intelligent victim selection by deviating from the strict LRU victimization policy. We partition the cache lines into two categories: those that we expect to have sustained temporal locality and those which we expect won't. When it is time to victimize a line, the replacement algorithm, called *Reference Locality Replacement* (RLR) algorithm, will attempt to keep lines with sustained temporal locality in the cache, even if one of them happens to be the LRU line. We begin by describing the hardware mechanism and replacement policy required to implement RLR. Afterwards, we describe two techniques to classify temporal locality as needed to utilize the RLR replacement policy.

tag	temporal bit	LRU stack
A	0	MRU
B	1	
C	0	
D	1	LRU

Figure 3. Temporal bit addition to the cache. To implement the RLR policy, we add a temporal bit which indicates that the cache line exhibits temporal locality and should not be victimized if possible. The LRU stack field prioritizes the lines based upon recency of last reference. For the shown cache set with set-associativity of 4, on a miss to line E, the RLR victim is C, while the LRU victim is D.

time		1	2	3	4	5	6	7	8	9
PC		**z**	**y**	x	w	v	u	t	s	r
data address		**A**	**B**	c	d	e	f	g	**A**	**B**
TNR		8	9	1000	900	1500	2000	3000	15	17
stack depth	1	A	B	c	d	e	f	g	A	B
	2		A	A	A	A	A	**A**	g	A
	3			B	B	B	B	B	**B**	g
	4				c	d	d	d	d	d
	5					c	e	f	f	f
	6						c	e	e	e
	7							c	c	c

Figure 4. Classifying temporal loads.This figure illustrates how PRL uses the OPT algorithm to classify instructions as exhibiting temporal locality. The state of the OPT stack is shown after the current reference is processed. The time of next reference (TNR) is the address' priority and is used during stack updates. If the referenced address is found within a specified window, in this case depth 2 through 6, the address exhibited temporal locality and the previous referencing instruction is counted as having made a temporal reference. In this example, the temporal reference counters associated with PCs z and y, at times 1 and 2, respectively, are incremented because the second references to A and B were later found, at times 8 and 9, within the temporal stack window.

2.1 Line victimization

In order to enable making a replacement decision different from LRU, the RLR cache line replacement policy adds a temporal bit to the cache line metadata (see Figure 3). This temporal bit is a hint that the cache line exhibits temporal locality and should be kept in the cache, even if it is the LRU element. We call cache lines with the temporal bit set *temporal lines*. Those lines that do not have this bit set, *nontemporal lines*, are the preferred victims and should be chosen for replacement over the temporal lines. When a replacement is necessary, RLR hardware victimizes the LRU nontemporal line. The most recently used (MRU) line is never chosen as the victim. In the cases where all non-MRU lines in the set have the same temporal bit value, RLR behaves like LRU and chooses the same victim as LRU.

The simple RLR strategy is designed to keep temporal designated cache lines resident. This is the desired effect when temporal lines are rereferenced. However, if the temporal line is no longer referenced (i.e., it is a dead line) and not enough temporal references are made to other data that map to the set, the unreferenced temporal line will remain in the set which reduces the set's effective size. To address this concern, we modified RLR to limit an unreferenced temporal line's life in the set.

The victim selection policy remains the same as before. Except for the MRU line, we victimize non-temporal lines over temporal lines. However, when a victim is selected that is not the LRU line, the LRU stack is reordered and the LRU line, which is necessarily a temporal line, has its temporal bit cleared. The LRU stack reordering is as follows. The original MRU line is unaffected and its state remains the same. Temporal lines are moved to the top of the LRU stack and non-temporal lines to the bottom.

We expect that the addition of the temporal bit will not have a large impact on the cache's implementation. First, the bit's representation overhead is small. Second, although RLR does make the replacement algorithm more complex, this replacement is at the L2 level where there is ample time to choose a victim and update the set's priority stack while the missing line is being fetched from memory.

2.2 Identifying temporal lines

Temporal lines are classified by two different strategies: the first is a profile-based approach that statically classifies instructions, while the second is a strictly hardware online approach that dynamically detects instruction temporality.

2.2.1 Profile Reference Locality

Profile Reference Locality (PRL) assumes the existence of temporal load instructions. These load instructions are similar to normal load instructions, but, in addition, they set the temporal bit for the line containing the referenced data, both on cache line allocation and hits during L2 accesses. These cache management instructions are similar to other proposed instructions that have the same user visible semantics as normal instructions but inform the memory hierarchy to behave differently [Tyson *et al.* 95, Intel 99]. Normal non-temporal memory instructions that access L2 (on a L1 cache miss) clear the temporal bit.

We now describe the profiling phase where PRL determines which loads to statically convert to temporal loads. Profiling is performed on an address trace, and since we are profiling for L2 use, the address trace is first filtered by a L1 cache. We then process the trace by simulating the OPT algorithm's replacement stack [Mattson *et al.* 70] for each cache set. The OPT algorithm's stack prioritizes the recent references based upon the time they are next referenced with the references closest to the top of the stack being the ones that will be rereferenced the soonest. Hence, data at the top of the stack are data we would prefer to keep in the cache. We define data, at the cache line granularity, as exhibiting temporal locality if it is rereferenced within a window at the top of the stack. The previous load that referenced the temporal line is classified as a temporal reference and is a candidate temporal load (see Figure 4 for an example). The depth of the window in the OPT stack that we use is dependent upon the targeted cache's set size. For example, to target an 8-way associative cache, a window of stack depth 2 through 6 (MRU has depth 1) is used. We do not include the MRU element in our locality window as references to the MRU are cache hits after the line is allocated in the cache and do not incur a replacement decision.

For each load, we record the number of references and the number of temporal references. At the end of profiling each application, a list of static loads having above a specified percentage of dynamic temporal references is generated. We will refer to this percentage as the *temporal reference threshold* (TRT). The loads that belong to the list of those above the TRT are converted to temporal loads. Note that a high TRT may result in too few loads being classified as temporal loads. Likewise, a low TRT can result in too many of the static loads being classified as temporal loads. At either extreme of the spectrum of TRT values, too many or too few references classified as temporal loads will cause PRL to make the same decisions as LRU.

2.2.2 Online Reference Locality

The PRL algorithm consists of two phases: statically classifying the temporal instructions via profiling and setting the temporal bit values in the cache at run-time. Online Reference Locality (ORL) follows the same philosophy, but now the two phases are necessarily closely interleaved. Profiling is approximated by keeping a table, created and updated at run-time, of whether an instruction (static PC) exhibits temporal locality or not. An entry in this table, called the *locality table*, consists of a single bit representing a locality boolean value. Allocation and updates of entries are performed according to the instruction's reference locality which is based upon the LRU stack depth of the instruction's previous reference. In our implementation, we consider that any hit to a non-MRU line indicates the presence of sustained temporal locality. As in the PRL case, the temporal instruction is the one that previously referenced the line with a PC of value *prev_PC*. The setting of temporal bits in the cache is performed as in PRL, except that temporality is determined by a table look-up in the locality table rather than via temporal or non-temporal instructions.

In order to implement ORL, not only do we need the locality table but also the PC of the instruction that previously referenced the line in the cache (*prev_PC*) must be recorded with the tag, a field that we call *PCtag*, for that line in the cache. The *PCtag* value will be used to update values in the locality table. On a L2 access, the current PC (*curr_PC*) must be sent along with the requested data address. The locality table is similar to a branch history table and the implementation should be similar.

We now describe what happens on a L2 reference from an instruction at *curr_PC*:

- •The reference is to the MRU line. No change in either the cache or the locality table states.
- •The reference is a hit to a non-MRU line. The locality bit of the *prev_PC* (found in the *PCtag* field of the cache) entry in the locality table is set to true. The cache line becomes MRU and its temporal bit is set to the value of the locality bit in the *curr_PC* entry of the locality table. The *PCtag* field for the cache line is set to *curr_PC*.
- •The reference is a miss. The victim is chosen as in Section 2.1. The missing line is brought in, becomes MRU, and its temporal bit is set to the value of the locality bit in the *curr_PC* entry of the locality table. The *PCtag* field for that line is set to *curr_PC*. The locality table entry with the victim's *PCtag* is reset to false.

3 Experiments

3.1 Methodology

To test the effectiveness of our approach, we used trace-driven simulations of applications that exercise second-level caches. The traces were filtered by a L1 cache as mentioned earlier. The benchmarks are briefly described in Table 1. Table 2 shows the trace execution characteristics and miss rates for a 256 KB, 8-way set associative L2 cache for various lines sizes with standard

application	description
hydro2d	SPEC95; astrophysics; solving hydrodynamical equations
su2cor	SPEC95; quantum physics; particle mass computations
specweb	SPEC web server benchmark
tpcc	TPC C database benchmark

Table 1. Benchmark descriptions.

application	use	references (M)		L2 data refs (M) (L1 misses)	L2 misses per instruction (x100)		
		inst	data		32 B	64 B	128 B
hydro2d	train	200.00	96.96	6.77	3.00	1.50	0.75
hydro2d-t	test	200.00	96.96	6.77	3.00	1.50	0.75
su2cor	train	200.00	95.89	3.96	1.24	0.62	0.31
su2cor-t	test	200.00	102.65	4.53	1.34	0.68	0.35
specweb	train	69.14	43.93	2.93	1.61	1.67	2.03
specweb2	test	30.00	19.04	2.67	2.67	2.81	2.83
specweb3	test	30.00	19.12	2.72	2.74	2.89	2.84
specweb4	test	30.00	19.01	2.61	2.70	2.77	2.67
tpcc2	train	148.80	83.28	2.44	0.91	0.59	0.42
tpcc3a	test	49.58	17.60	0.50	0.92	0.58	0.39
tpcc3c	test	102.69	63.24	2.23	0.89	0.63	0.47
tpcc4a	test	170.70	92.08	2.66	0.95	0.59	0.41
tpcc4b	test	217.86	113.47	2.80	0.91	0.55	0.37

Table 2. Trace characteristics. This table shows the data cache miss characteristics of our applications at both the first and second-level caches. The first level (L1) cache is 16 KB and 4-way associative. The second-level (L2) cache is 256 KB and 8-way associative. Both caches have 32 B lines and used LRU replacement policies. For the L2 cache miss rate, we show the L2 misses per instruction (x 100) for three different line sizes.

LRU replacement. Traces were obtained with two different methods. The SPEC95 traces contain user references and were generated through the SimpleScalar toolset [Burger & Austin 97] with a simple in-order processor model. The traces for these applications were generated by 200 million instructions from the middle of program execution. The *tpcc* and *specweb* traces were obtained with a hardware trace gatherer and contain both user and system references. Those traces were obtained from systems running the Microsoft NT 4.0 operating system. We used only the data references in our studies and treated all data accesses as reads.

The PRL algorithm evaluation is performed in two phases. Phase 1, done once per application, is the profiling itself performed on a training data set (*hydro2d*, *su2cor*, *specweb*, and *tpcc2* in Table 2). The output of this phase is a list of static PC's where temporal loads are generated as explained in Section 2.2.1. Phase 2 performs a subsequent trace-driven cache simulation with the test data set traces using the output of the first phase to drive the PRL replacement policy.

3.2 PRL evaluation

The goal of the first PRL experiments, whose results are presented now, is to select the PRL profiling parameters that will be used to generate the profiling data (phase 1) for later PRL cache simulations (phase 2). We present results where the profiled data was generated with parameters that matched the target cache. In general, we would like to select the profiling parameters that offer the best universal performance and stability.

3.2.1 PRL parameters

In these experiments, the same application input data was used during profiling and subsequent PRL simulations. Our goal is to determine what *temporal reference threshold* (TRT) value to use and using the same data for profiling and PRL simulation should give us a rough upper bound on the gains possible with PRL. Recall that the TRT value determines how many and which loads are statically converted to temporal loads (Section 2.2.1). Table 3 shows the number of static PCs converted and the percentage of dynamic temporal loads executed for different TRT values (percentage dynamic temporal loads is the ratio of L2 references caused by temporal loads to total L2 references). For the applications *su2cor* and *hydro2d*, few of the static PCs are responsible for a large percentage of the dynamic PCs that resulted in L2 accesses. *Hydro2d* suffers from the additional circumstance that its L2 references do not exhibit temporal locality. Hence, for *hydro2d* we expect PRL to have little impact.

In Figure 5 we show the resulting performance of PRL with different TRT values. PRL can noticeably reduce the number of misses for a range of TRT values, up to 17% fewer misses than LRU (*specweb* with 64 byte lines and TRT=60%). Using a TRT value of 30 to 60% generally performs the best and improves performance, except in one instance (*su2cor* with 128 B lines) where these TRT values resulted in up to 2% more misses. For *hydro2d*, as expected, the TRT value tended to not affect performance. For the simulations, on the testing data sets, we used the TRT values shown in Table 4. Although choosing a fixed TRT value per application may not give us the best performance for each cache configuration, it eliminates having separate profiles for each cache configuration.

3.2.2 PRL with different application data

In this section, we present the results of the PRL on test data different from the training data sets used for profiling. The performance of the PRL algorithm is shown in Figure 6. The number of L2 misses normalized to LRU misses is

application	TRT(%)=10		20		30		40		50		60		70		80		90	
	static	dyn	static	dyn	static	dyn	static	dyn	static	dyn	static	dyn	static	dyn	static	dyn	static	dyn
lsize 32, assoc 8																		
specweb	7367	89.63	7118	87.82	6781	86.47	6255	79.46	5886	75.95	4839	70.57	3910	62.72	3149	39.90	2561	2.122
tpcc2	8165	75.33	7650	68.32	6963	61.23	6137	51.80	5450	41.96	4188	32.78	3144	18.42	2429	8.13	1889	2.55
su2cor	168	77.30	152	71.35	137	64.26	119	47.62	101	37.33	87	32.48	76	28.41	68	24.00	60	22.11
hydro2d	234	35.05	218	25.88	187	13.31	154	5.15	142	1.60	121	1.53	114	1.51	94	0.38	86	0.15
lsize 64, assoc 8																		
specweb	6066	86.70	5608	81.35	5031	77.24	4323	55.82	3865	48.37	2949	41.92	2342	24.34	1993	4.16	1678	1.42
tpcc2	7018	64.87	6359	53.83	5501	44.61	4701	37.31	4185	29.18	3259	21.20	2575	15.25	2028	7.38	1626	2.58
su2cor	139	72.21	117	51.63	93	32.23	83	24.06	63	7.46	46	1.92	39	0.62	33	0.47	27	0.07
hydro2d	232	25.90	184	5.24	168	1.71	152	0.55	121	0.23	94	0.14	86	0.11	68	0.06	64	0.04
lsize 128, assoc 8																		
specweb	5269	81.94	4665	62.15	4003	54.62	3373	48.64	3003	12.82	2282	9.57	1836	5.62	1588	3.34	1380	1.51
tpcc2	6016	53.47	5212	42.25	4444	33.21	3815	27.02	3416	22.30	2698	18.48	2226	13.84	1823	8.19	1457	3.04
su2cor	178	97.74	167	93.96	160	87.60	158	87.58	151	86.93	122	70.82	103	56.35	70	32.52	45	8.03
hydro2d	181	5.23	162	0.72	135	0.31	130	0.28	121	0.23	100	0.15	85	0.08	65	0.05	64	0.05

Table 3. Instructions converted to temporal instructions. This table shows the number of static loads (column static) and percentage of dynamic loads (column dyn) that were converted to temporal loads. The percentage is the ratio of L2 temporal loads executed to total L2 loads executed. The profile set and line size parameters targeted a 256 KB cache with 8-way associativity. A locality window of stack depth 2 through 6 was used.

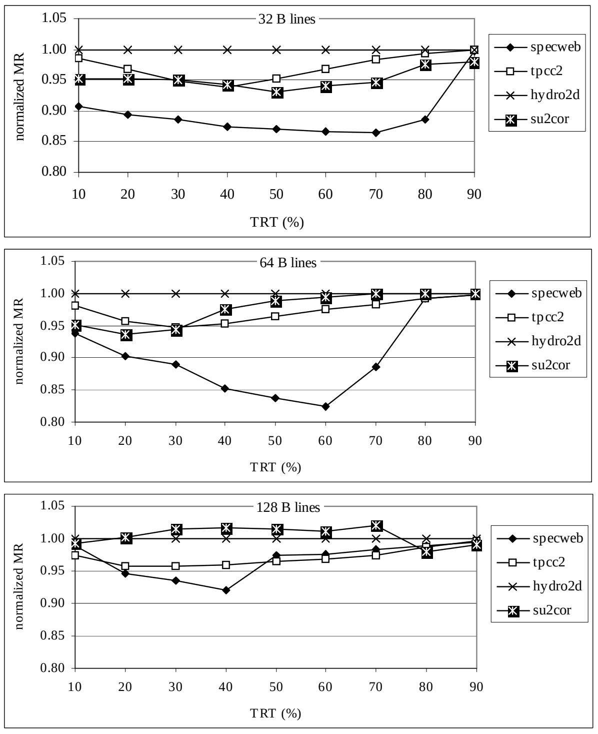

Figure 5. PRL with various TRT values. These graphs show the L2 cache misses incurred by PRL as a percentage of the LRU misses for various TRT values. The results were obtained with three different line sizes (32, 64, and 128 B lines). The L2 cache is 8-way associative and 256 KB.

application	TRT (%)
specweb	60
tpcc	30
hydro	40
su2cor	30

Table 4. Chosen TRT values.

shown for both PRL and the optimal (OPT) replacement policies. From the graphs, we see that PRL can noticeably reduce the number of misses (up to 12% on *specweb* and 10% on *tpcc*). In all but one case, *su2cor* with 128 byte lines, PRL generated fewer misses than LRU. For the case where PRL performed worse than LRU, the additional number of misses generated is less than 1%.

PRL's performance on test data compared to its performance on the training set is dependent upon the application. PRL's performance is generally slightly lower and noticeably lower with 64 byte lines when *specweb*'s data is changed. On *tpcc*, the performance of PRL changed, sometimes better and sometimes worse. However, the relative data-dependent performance changes seem to be due to the changes in opportunity, as seen by the performance of OPT, rather than a change in *tpcc*'s data. PRL's performance is unchanged for both *hydro2d* and *su2cor.*

For some cache configurations PRL performs the same as traditional LRU. Recall from Section 3.2 that a general trend is that with longer set lines, the percentage of temporal loads is much lower (except for *su2cor*) which decreases the opportunities for PRL to make a decision different from LRU (see Table 5). This combined with the

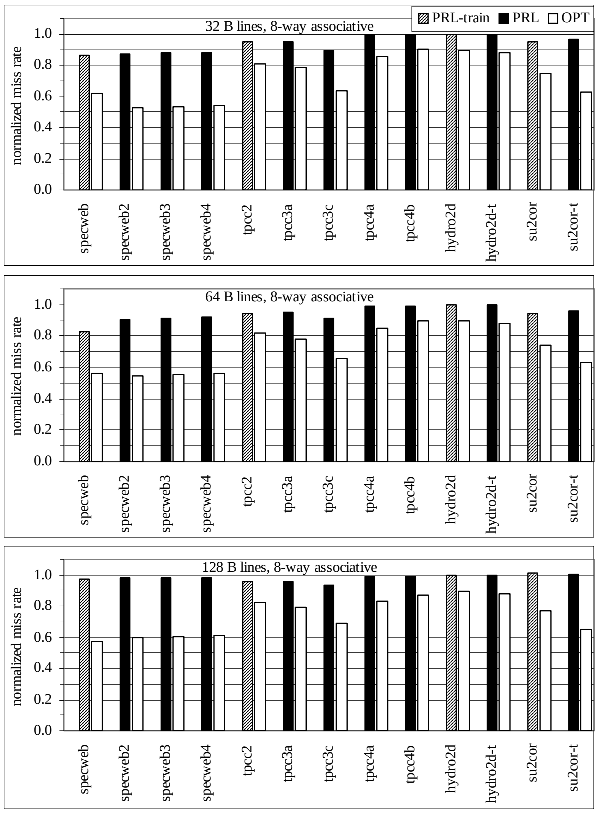

Figure 6. PRL Performance with different application input. These graphs show the performance of PRL when different application data are used. PRL-train are the cases where the profiling phase and subsequent PRL simulation used the same application data. The performance is measured as the number of L2 misses generated relative to LRU.

reduced opportunity, because of the narrowed gap between OPT and LRU, results in limited, if any, PRL gains. Still, PRL is able to obtain moderate gains (4-6%) on *tpcc* with longer 128 bytes cache lines.

From our simulations we have demonstrated that PRL can improve the L2 cache performance of some applications. While the improvements are relatively moderate reductions, these reductions are on applications that have high L2 miss rates and significant miss frequencies. Thus, the L2 cache improvements should result in noticeable end-to-end improvements. However, static classification can be ineffective in some cases, exemplified by *hydro2d* and *su2cor*, where relatively few instructions are responsible for the majority of L2 accesses. In the next section, we propose a hardware-based alternative to PRL to address these limitations.

application	32 B lines	64 B lines	128 B lines
specweb2	48.25	28.05	8.81
specweb3	45.98	26.57	8.24
specweb4	46.34	26.49	8.82
tpcc3a	57.17	40.54	30.71
tpcc3c	74.65	54.43	40.96
tpcc4a	19.97	15.75	11.72
tpcc4b	13.95	11.16	8.38
hydro2d-t	5.15	0.55	0.28
su2cor-t	49.93	23.08	72.10

Table 5. Instructions converted to temporal instructions with different traces. This table shows the percentage of dynamic L2 references that were treated as temporal instructions. The static counts are not shown and are the same as in Table 3.

3.3 Hardware based approach

The purely hardware based online algorithm, Online Reference Locality (ORL), that we present now does not statically classify an instruction; instead, the hardware sets the temporal bit in the cache based upon the instruction's previous reference history. A disadvantage of ORL is that the hardware has limited reference history information available. We expect some loss in performance, compared with PRL, due to the limited history and lack of future reference information that was approximated with profiling. However, we also expect an increase in performance due to dynamic instruction classification.

3.3.1 ORL evaluation

To evaluate ORL, we present the results of two sets of experiments. In the first set, we assume an unbounded locality table size (i.e., with one entry for each possible PC value). Using an unlimited locality table allows us to determine the extent to which ORL can improve performance. In the subsequent experiments, we evaluate ORL with practical locality table geometries.

3.3.2 Unbounded locality table

In Figure 7 we compare the performance of the ORL and PRL strategies. On *specweb,* ORL was able to consistently generate fewer misses than PRL. On *tpcc*, the results were mixed; ORL's performance with *tpcc2* and *tpcc3* were slightly worse. However, on the more difficult to improve *tpcc4*, ORL achieved fewer misses. ORL's performance on *hydro2d* and *su2cor* was similar to that of PRL. Surprisingly, ORL was able to improve performance with longer 128 byte lines where PRL was generally not able to. From these experiments, we conclude that the ORL strategy has the potential to obtain performance similar to or better than that of PRL.

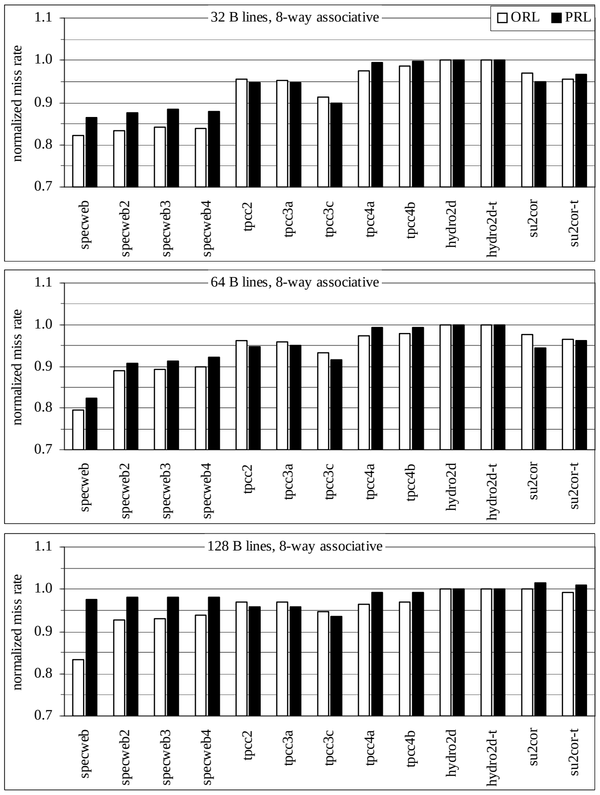

Figure 7. ORL performance. These graphs show the performance of ORL. The number of L2 misses normalized to LRU misses are shown both for ORL and PRL strategies. The performance of ORL is with an unbounded locality table.

3.3.3 Fixed locality table

In this section, we investigate the performance of ORL with practical locality table geometries, namely 8 K, 16 K and 32 K entries (recall that an entry is a single bit). For all applications, except *specweb*, the performance of ORL, measured in relative L2 misses, with limited table sizes was within 1% of the performance of ORL with unbounded tables. For *specweb* (Figure 8), 32 K entries were required in order to limit the increase of L2 misses to 2% with 32 B lines. Although the number of misses increased with a fixed locality table, the total misses are still lower than with PRL (see Figure 9). Generally, there was not a significant increase in the number of L2 misses with practical locality table geometries when compared with an unlimited locality table.

An entry in the locality table is small: a single bit and, thus, a 32 K entry table requires a real estate of 4 KB. The *PCtag* field in the data cache requires less than 2 bytes to address the 32 K entry locality table. For a 256 KB, 8-way set-associative cache, the *PCtag* costs would be 16 KB, 8 KB, and 4 KB for 32 B, 64 B, and 128 B L2 line sizes,

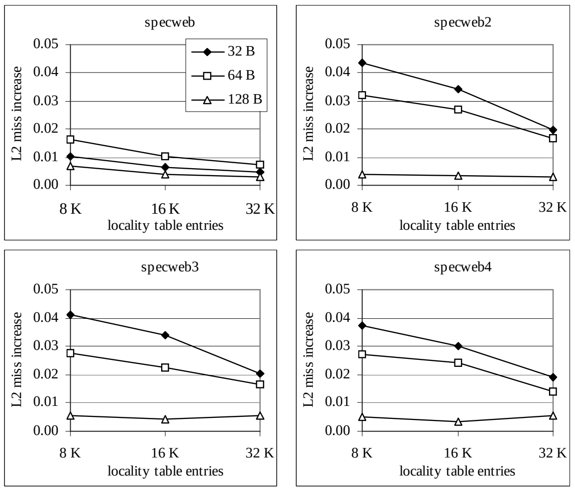

Figure 8. Performance with a fixed locality table. These graphs show the miss rate increases on specweb for ORL with using a fixed bounded locality table compared with an unbounded locality table. The number of locality table entries were 8 K, 16 K, and 32 K. The increase in L2 misses, relative to an unbounded locality table, are shown. Each line is the performance with a specific L2 line size. A 16 byte PC granularity was used to index into the locality table.

respectively. The estimated total cost of ORL would be 8-20 KB for our targeted 256 KB L2 geometries.

In summary, ORL performs as well, and noticeably better in some cases, than PRL. It can adapt better to dynamic changes and does not require a profiling phase or changes to the ISA. On the other hand, it requires additional hardware both in the L2 tag array and in the form of an additional table as well as the constraint of transmitting the PC value to the L2 cache on a L1 miss.

Table 6 show the IPC speedups obtained by ORL when compared to LRU. The execution cycles were calculated with a simple inorder, single-issue pipeline. The L2 latency is 10 cycles and the memory latency is 100 cycles. Understandably, the IPC speedups are lower than the L2 miss rate reductions. Still, the speedups are noticeable on the *specweb* application with all cache line sizes.

3.3.4 Comparison with other cache models

We compared the ORL approach with two other cache assist models. The first model follows a victim cache (VC) strategy [Jouppi 90]. The second model follows the CNA strategy which uses instruction-based classification for data reuse [Rivers *et al.* 98]. For both models, we used a 256 KB primary cache and added a 16 KB, 32-way set-associative assist cache (see Figure 10). The size of the assist cache (data and tags) is comparable to the size (8-20

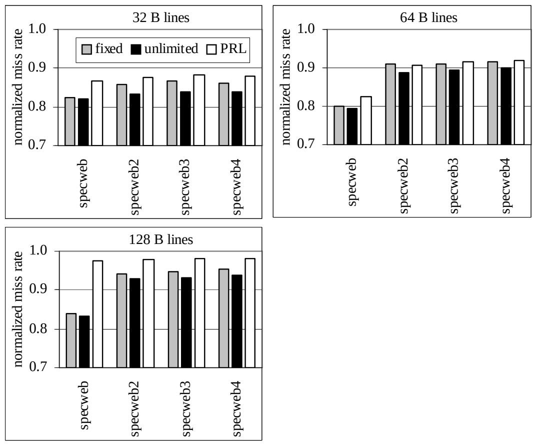

Figure 9. Comparing ORL with a fixed locality table and PRL. These graphs show the L2 miss rate (normalized to LRU) on specweb for the fixed and unlimited ORL strategy compared with PRL. The locality table for the fixed ORL approach has 32 K entries with a 16 byte granularity.

application	32 B lines	64 B lines	128 B lines
specweb	1.1042	1.1233	1.1080
specweb2	1.1091	1.0689	1.0488
specweb3	1.1042	1.0666	1.0451
specweb4	1.1073	1.0624	1.0397
tpcc2	1.0197	1.0123	1.0075
tpcc3a	1.0218	1.0128	1.0078
tpcc3c	1.0369	1.0224	1.0140
tpcc4a	1.0110	1.0085	1.0088
tpcc4b	1.0062	1.0063	1.0069
su2cor	1.0166	1.0108	1.0007
su2cor-t	1.0216	1.0130	1.0024

Table 6. IPC Speedups. This table shows the IPC speedups obtained by ORL over LRU.

KB) of the additional structures required by ORL. Note that the CNA strategy also requires a structure similar to our locality table, called a detection unit (DU) in their paper. The CNA's DU consists of 4 K entries with 4-way set-associativity.

In Figure 11 we show the performance for the three cache models, ORL, VC, and CNA. Our ORL approach tends to perform similarly or better than VC, except with *su2cor*. On *su2cor*, VC is effective, where ORL and CNA are not. CNA, on occasion, is able to noticeably outperform the other models (*specweb* with long cache lines and *tpcc3b*). However, it's behavior is more erratic and can lead to noticeable degradation (e.g., *specweb4* with short cache lines). With the *specweb* and *tpcc*

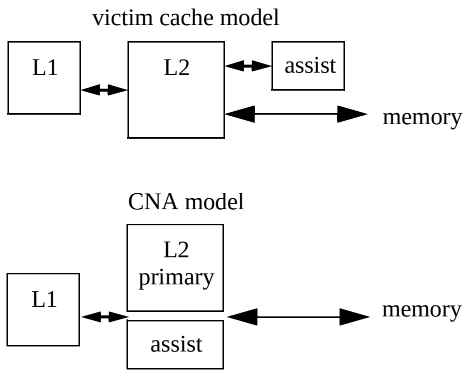

Figure 10. Other cache models. This figure illustrates the organization of the other cache models in our comparisons. The victim cache model uses the assist cache to cache victims from L2. The CNA model selectively allocates lines to either the primary or higher set-associative assist cache.

applications, despite using less amount of cache, ORL provides the best overall performance and has the advantage of a single cache structure.

4 Conclusions

The current trends of decreasing memory latency not keeping pace with increases in processor speed suggests that eliminating cache misses at the cache-memory interface will continue to be important for speeding up large applications. With caches becoming more set-associative, strategies that eliminate conflict misses will exhibit diminishing returns in performance with future cache geometries. Instead, strategies that target making more intelligent replacement decisions will continue to show performance improvement.

Consistent with this trend, we demonstrate the effectiveness of a new replacement algorithm for large set-associative L2 caches that is a modification of the standard LRU replacement algorithm. The modifications take into account the potential temporal locality of lines residing in the L2 cache by favoring the victimization of non-temporal lines. We have investigated two possible strategies for detecting temporal locality.

The first strategy is a profile-based approach (PRL) that statically determines *temporal* and *non-temporal* instructions. We show that PRL can yield up to 12% fewer misses than LRU. With compiler analysis or programmer hints, it should be possible to further increase the effectiveness of this static classification approach.

The second strategy is a purely hardware-based online

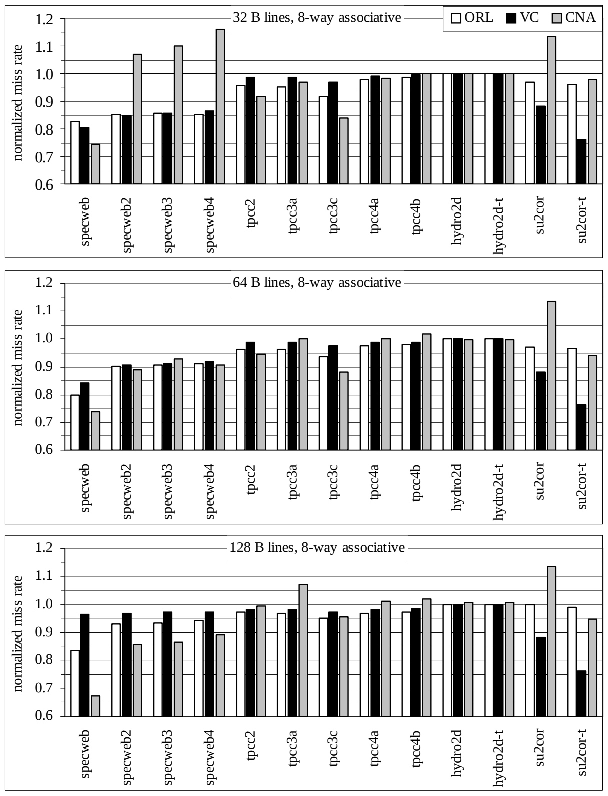

Figure 11. Comparing with other cache models. These graphs show the performance of the ORL strategy compared with two other cache models, victim caching (VC) and CNA. The misses are normalized to LRU's. The primary L2 cache size is 256 KB. The secondary cache for VC and CNA is 16 KB, 32-way set associative. A hit in either the primary or secondary cache is considered a hit for the cache model.

approach (ORL). By dynamically classifying the locality behavior of instructions at run-time, ORL has similar performance as PRL and is even capable of achieving up to 20% fewer misses than LRU. We show that the hardware requirements of ORL are not large and can be implemented with the equivalent space of a relatively small cache.

We are planning research using the temporal line concept to further improve the behavior of L2 caches. In this paper, we have looked at references to each set in isolation and we have not attempted to detect locality in adjacent sets. Doing so and coupling replacement algorithms with prefetching might yield further improvements. The fact that our current on-line scheme requires the value of the PC that generated the L1 miss to be transmitted to the L2 along with the missing L1 line address should be advantageous in this respect.

5 Acknowledgments

We would like to thank Chris Wilkerson (Intel MRL) and Nick Wade (Intel MRL) for providing the traces and for their insights, observations, and suggestions. We also thank Todd Austin for his review and suggestions for strengthening the paper. This work was supported in part by NSF Grant MIP-9700970 and by a gift from Intel Corporation.

References

[Abraham *et al.* 93] Abraham, S. G., Sugumar, R. A., Windheiser, D., Rau, B. R., Gupta, R. Predictability of load/store instruction latencies. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 139-152, Austin, Texas, December 1-3, 1993.

[Barroso *et al.* 98] Barroso, L., Gharachorloo, K., and Bugnion, E. Memory System Characterization of Commercial Workloads. In *The Twenty-fifth Annual International Symposium on Computer Architecture*, pages 3-14, June 1998.

[Belady 66] Belady, L. A. study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78-101, 1966.

[Bhandarkar & Ding 97] Bhandarkar, D. and Ding, J. Performance characterization of the Pentium Pro processor. In *3rd Annual International Symposium on High-Performance Computer Architecture*, pages 288-297, February, 1997.

[Burger & Austin 97] Burger, D. C. and Austin, T. M. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[Calder *et al.* 98] Calder, B., Krintz, C., John, S., and Austin, T. Cache-Conscious Data Placement. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 139-149, October 1998.

[Hashemi *et al.* 97] Hashemi, A., Kaeli, D., and Calder, B. Efficient Procedure Mapping Using Cache Line Coloring. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 171-182, June 1997.

[Johnson & Hwu 97] Johnson, T. L. and Hwu, W-M. Run-time Adaptive Cache Hierarchy Management via Reference Analysis. In *The Twenty-fourth Annual International Symposium on Computer Architecture*, pages 315-326, June 1997.

[Intel 99] Intel Corporation. *IA-64 Application Developer's Architecture Guide*, 1999. Order Number: 245188-001.

[Jouppi 90] Jouppi, N.P. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *The Seventeenth Annual International Symposium on Computer Architecture*, pages 364-373, May 1990.

[Lebeck & Wood 94] Lebeck, A. and Wood, D. Cache profiling and the SPEC benchmarks: A case study. *Computer*, 27(10), 15-26, 1994.

[Lee *et al.* 98] Lee, D., Crowley, P., Baer, J-L., Anderson, T., and Bershad, B. Execution Characteristics of Desktop Applications on Windows NT. In *The Twenty-fifth Annual International Symposium on Computer Architecture*, pages 27-38, June 1998.

[Mattson *et. al.* 70] Mattson, R. L., Gecsei, J., Slutz, D. R., and Traiger, I. L. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78-117, 1970.

[Perl & Sites 96] Perl, S. and Sites, R. Studies of Windows NT performance using dynamic execution traces. In *Second USENIX Symposium on Operating Systems Design and Implementation*, pages 169-193, 1996.

[Pettis & Hensen 90] Pettis, K. and Hansen, R. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 16-27, June 1990.

[Rivers *et al.* 98] Rivers, J. A., Tam, E. S., Tyson, G. S., Davidson, E. S., and Farrens, M. Utilizing Reuse Information in Data Cache Management. In *Proceedings of the 12th ACM International Conference on Supercomputing*, July 1998.

[Smith 82] Smith, A. J. Cache Memories. ACM Computing Surveys, 14(3):473:530, 1982.

[Smith & Goodman 85] Smith, J. and Goodman, J. Instruction Cache Replacement Policies and Organizations. *IEEE Transactions on Computers*, C-34(3), 234-241, 1985.

[Sugumar & Abraham 93] Sugumar, R. A. and Abraham, S. G. Efficient Simulation of Caches under Optimal Replacement with Applications to Miss Characterization. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 24-35, May 1993.

[Tyson *et al.* 95] Tyson, G., Farrens, M., Matthews, J., and Pleszkun, A. A Modified Approach to Data Cache Management. In *The Twenty-eighth Annual International Symposium on Microarchitecture*, pages 93-103, December 1995.