A New Approach to Cache Management

**Gary Tyson, Matthew Farrens,
John Matthews and Andrew Pleszkun**

Technical Report No. CSE-95-6
Computer Science Department
University of California, Davis, CA 95616
tel: (916) 752-7002, fax: (916) 752-4767
{tyson,farrens,matthewj}@cs.ucdavis.edu
arp@tosca.colorado.edu

August 8, 1995

Abstract

As processor performance continues to improve, more emphasis must be placed on the performance of the memory system. The caches employed in current processor designs are very similar to those described in early cache studies. In this paper, a detailed characterization of cache behavior for individual load instructions is given. A novel approach to modifying cache behavior to improve overall performance of the cache and the memory system is also described. This approach can improve aspects of memory performance by as much as 60 percent on existing executables.

This work was supported by National Science Foundation Grants CCR-8706722 and CCR-90-11535.

A New Approach to Cache Management

Gary Tyson

Department of Computer Science
University of California, Riverside
Riverside, CA
(tyson@cs.ucr.edu)

Matthew Farrens
John Matthews

Computer Science Department
University of California, Davis
Davis, CA 95616
(farrens@cs.ucdavis.edu)
(matthewj@cs.ucdavis.edu)

Andrew R. Pleszkun

Department of Electrical
and Computer Engineering
University of Colorado-Boulder
Boulder, CO 80309-0425
(arp@tosca.colorado.edu)

ABSTRACT

As processor performance continues to improve, more emphasis must be placed on the performance of the memory system. The caches employed in current processor designs are very similar to those described in early cache studies. In this paper, a detailed characterization of cache behavior for individual load instructions is given. A novel approach to modifying cache behavior to improve overall performance of the cache and the memory system is also described. This approach can improve aspects of memory performance by as much as 60 percent on existing executables.

1. Introduction

Because data requests occur much less frequently than instruction references, the handling of data references by the data cache has kept pace with improvements in processor performance by more or less traditional means. Data cache cycle times have been been able to keep pace with the clock cycle of the machine, lock-up free caches designs have been used, more pipelined cache designs are beginning to appear, and a number of schemes to deal with writes so that data writes do not slow down the normal operations of the cache have been developed. All these approaches help decrease the average access time to the cache, but they do not fundamentally change the way in which the cache operates. The placement and replacement strategies are essentially the same as when caches first appeared. And as multiple-issue processors continue to increase the the number of instructions they can issued each cycle, there will been a

This work was supported by National Science Foundation Grants CCR-8706722 and CCR-90-11535.

corresponding increase in the demands placed on the bandwidth to the data memory - the data cache in particular will be hard pressed to service more than one data reference per cycle.

Since it is not clear that traditional methods of reducing the data cache miss rate and miss penalty will be sufficient, we believe that a somewhat different approach is needed. In this paper we examine the potential of reducing the data cache miss rate by dynamically deciding whether to cache a particular data item based on the address of the load instruction generating the request. The techniques we will be describing are largely orthogonal to standard miss rate/miss penalty reduction techniques, and should work well in conjunction with improvements made on other fronts.

2. Background

Caches are a very well-studied and well-understood tool used to reduce the average memory access time. In this section we will briefly summarize the aspects of cache behavior relevant to this study.

The average access time for a memory reference is determined by the hit rate in the cache, the corresponding miss rate and the miss penalty. From [HePa90]:

Memory stall clock cycles	=	*Reads x Read miss rate x Read Miss Penalty*
	+	*Writes x Write miss rate x Write Miss Penalty*

This equation shows that in order to minimize the average access time, the hit rate should be maximized (thereby minimizing the miss rate) while simultaneously minimizing the miss penalty.

2.1. Cache Misses (Miss Rate)

Cache misses can be categorized into three types of misses: *Compulsory, Capacity and Conflict* [HePa90].[1] *Compulsory* misses are those misses that are initially experienced when a cache is being filled (often called cold start misses), and are very difficult to eliminate. A *capacity* miss occurs in a cache when more active data items exist than the cache can encompass. These misses can be reduced in number by either increasing the number of lines in the cache or by increasing the size of each line in the cache (or both). A *conflict* miss may occur in cache designs that restrict the placement of data items in the cache, when more items map to the same cache set than the associativity of the cache supports. In order to reduce the number of conflict

[1]The cache design will determine the relative weight of each type of miss on the overall cache miss rate.

misses, the design should relax the restrictions on the placement of a line in the cache. Conflict misses do not occur in a fully associative cache, but become the dominate miss category in the large direct mapped caches commonly used. Generally, increasing the associativity of the cache reduces the frequency of conflict misses.

Unfortunately, in the design of todays high-performance processors, it is difficult to increase either the size or the associativity of a cache because the access time of the data cache must first and foremost match the clock cycle time of the processor. Several studies have investigated the effect of cache access time with respect to cache size and cache associativity [MuQF91, WaRP92]. These studies carefully parameterized a hardware model of the components of the cache (such as data array, tag array, compare logic, bus delays, etc.) and found that going from a direct mapped to 2-way associative cache substantially increases the access time to the cache. A similar conclusion can be made when increasing the cache size beyond 8K or 16K bytes.

2.2. Miss Penalty

The miss penalty accounts for the time it takes to read a cache line from (or write a cache line to) the next level in the memory hierarchy. This penalty has been climbing as the disparity between main memory access times and processor clock speeds widen. Especially in processor designs with on-chip caches, access time to off-chip memory (measured in processor clock cycles) has increased dramatically. Since there is no indication that dynamic memory access times will significantly decrease in the near future, this situation is unlikely to change for some time.

In order to minimize the impact of slow main memory access times, several strategies are often used. First, one can interleave the main memory, so that each word of a cache line does not have to experience the full latency of the main memory. Second, one can place a second-level cache into the system - since the second-level cache will presumably be smaller than the main memory, it can be built using faster and more expensive logic. A third option is to widen the bus between the primary cache and the main memory (or second-level cache).

These options all have the effect of increasing the bandwidth of the data flowing across the chip boundary. The effectiveness of these strategies depends on how easy it is for designers to increase the number of pins on a chip and/or increase the rate at which these pins are driven.

2.3. Other Approaches to Reducing Cache Miss Penalty and Miss Rate

There have been a number of studies investigating alternate methods for reducing the overall miss penalty of a data reference. Most of these studies have focused on reducing the

miss penalty of the cache by performing some type of data prefetch. If data items are prefetched during idle data cache cycles, references to a prefetched item will find it already in the cache and thus will not cause a miss and will not experience the associated miss penalty. A number of studies have investigated various forms of prefetching [CaPo, ChBa95, KlLe91], using either compiler-based or hardware-based techniques to determine which data items to prefetch. An example of hardware-based prefetching is the work by Chen and Baer [ChBa95]. In this paper the authors propose keeping a history of the strides of data references, and using that information to make predictions as to what should be prefetched.

A somewhat different hardware approach to reducing the miss penalty is put forth by [Joup90]. The author makes the observation that a cache or a degree of associativity that is too small will lead to a substantial number of conflict (or capacity) misses, and that there is a good chance that the line that is selected for replacement will be needed again soon. Therefore, he proposes the use of a small, fully associative buffer sitting at the back end of the cache (called a *victim cache*) to buffer these replaced lines. By keeping these recently replaced lines in close proximity to the data cache, subsequent references to them will not experience the full main memory miss penalty.

Among the most intriguing software approaches to reducing the miss penalty is a study by Abraham, et. al. [ASWR93] in which they observe that a very small number of load instructions are responsible for causing a disproportionate percentage of cache misses. By using profiling techniques similar to those used to schedule code for VLIW machines, the compiler can accurately identify those data reference instructions which cause the highest data cache miss rates. By recognizing these data references and using special instructions to control the cache, the software can effectively prefetch those instructions and reduce the miss penalty.

The work described above focuses primarily on reducing the miss *penalty* of a data cache. Instead of concentrating on miss penalty, one could potentially reduce the miss rate of the data cache by simply not caching those data references that would lead to a poor miss rate.

3. Deciding What to Cache

As Abraham, et el. [ASWR93] point out, a large percentage of the data misses are caused by a very small number of instructions. We decided to look at the impact on the data cache miss rate if the data cache is smarter about what it decides to cache and does not allow these troublesome instructions to allocate space in the cache. Since they did not look at an extensive set of benchmark programs, we began by performing some experiments similar to theirs (although more extensive) in which we measured the miss rate associated with individual load and store instructions.

Using the ATOM program trace facilities [SrWa94] and the SPEC92 suite of benchmarks, such statistics were relatively straight-forward to gather. We instrumented each program in the SPEC 92 suite in order to track the data cache hit rate associated with each unique data address. We simulated a 32-byte line size, and both 8K-byte and 16K-byte caches, which were direct mapped, 2-way set associative, 4-way set associative, and direct-mapped with a victim cache.

Table 1 presents a detailed breakdown of each benchmark analyzed, the input that was used, the total number of load data references and the hit rates for each of the cache configurations simulated. Our results are not surprising, and match those from many other studies - as expected, direct mapped performs the worst in general while increasing the associativity improves the accuracy.

It is worth noting using a 16 line victim cache in conjunction with a direct mapped cache provides cache performance that generally lies somewhere between the 2-way and 4-way

Table 1: Data Cache Hit Rate for SPEC Benchmarks

SPEC	Input	# of	8K Cache Hit Rate (%)				16K Cache Hit Rate (%)			
Benchmark	File	Load Refs	D	2	4	V	D	2	4	V
compress	in	171038476	84.75	90.54	90.92	90.72	85.90	91.24	91.46	91.26
eqntott	int_pri_3.eqn	231129466	94.14	95.40	95.49	95.47	94.72	95.62	95.67	95.65
espresso	bca.in	93384214	89.70	90.70	91.15	90.83	91.39	92.46	92.73	92.05
espresso	cps.in	110649363	92.82	94.49	95.04	94.45	96.07	96.84	96.97	97.08
espresso	tail.in	218814920	92.32	95.60	96.26	95.47	95.58	98.00	98.68	97.44
espresso	ti.in	123094748	92.74	94.37	95.12	94.69	95.62	96.57	96.99	96.84
gcc	cexp.i	5516709	90.71	94.33	95.60	94.30	94.87	96.98	97.73	96.74
gcc	jump.i	12342182	90.69	94.14	95.34	93.94	94.51	96.72	97.45	96.43
gcc	stmt.i	34338681	90.86	94.13	95.30	93.86	94.64	96.80	97.47	96.41
li	li_input.lsp	1923073359	89.75	94.58	95.68	94.81	93.40	96.59	97.32	96.85
sc	loada1	325317588	84.43	86.38	87.17	86.55	86.37	87.64	87.77	87.51
sc	loada2	365851439	88.44	90.44	91.14	90.87	90.18	91.76	91.95	91.63
sc	loada3	104774878	92.73	94.31	94.96	94.70	94.77	95.62	95.83	95.66
Int Ave			90.31	93.03	93.78	93.13	92.92	94.83	95.23	94.73
doduc	doducin	337197266	88.54	92.96	96.36	94.01	91.31	96.74	97.29	95.78
fpppp	natoms	1529995204	94.59	97.64	98.05	96.81	96.44	99.12	99.70	98.12
mdljdp2	input.file	414815788	85.51	88.44	91.56	89.23	93.00	93.81	94.50	94.04
mdljsp2	input.file	753307294	95.33	96.97	97.67	97.58	97.35	98.52	98.60	98.58
spice2g6	greycode.in	5265522592	69.75	72.99	74.26	75.31	78.23	81.21	83.29	80.08
su2cor	su2cor.in	1069475885	48.61	49.71	64.53	84.42	62.97	65.79	65.50	85.59
swm256	swm256.in	2851234080	75.72	68.66	66.54	92.99	92.88	92.64	91.95	93.23
tomcatv		247280519	63.78	60.25	66.03	86.08	75.44	86.24	88.06	88.06
wave5		758316974	89.08	91.58	91.48	93.82	94.76	95.84	95.81	95.71
FP Ave			78.99	79.91	82.94	90.03	86.93	89.99	90.52	92.13

associative configurations. In fact, in several benchmarks a considerable improvement over 4-way associativity is demonstrated (e.g. a jump from 64% to 84% in su2cor). This indicates that the number of conflicting items in the direct mapped cache is small but can be clustered in a single cache line. By allowing the associativity of the victim cache to be directed at those contention spots, performance can be improved with less hardware resources.

In order to better understand what is causing the cache misses we looked at the reference pattern of each program in greater detail. Table 2 presents the cumulative percentage of references cause by the most heavily executed load instructions in an 8K-byte direct mapped cache. Each row of the table contains the information gleaned from a run of the given SPEC benchmark, and the 6 columns which are labeled with a percentage of total load references each have two sub-columns indicating the total number of instructions that caused that percentage and the

Table 2: Cumulative Load Instruction Reference Counts

Bench-mark	Percent of Total Data References											
	50%		75%		90%		95%		99%		100%	
	# of Insts	% of Total	# of Insts	% of Total	# of Insts	% of Total	# of Insts	% of Total	# of Insts	% of Total	# of Insts	% of Total
compress	18	0.59	38	1.24	50	1.64	54	1.77	58	1.90	3058	100
eqntott	2	0.04	8	0.17	59	1.27	96	2.06	157	3.37	4656	100
espresso.bca	28	0.17	70	0.43	157	0.95	280	1.70	933	5.67	16447	100
espresso.cps	55	0.33	137	0.82	299	1.80	588	3.53	1388	8.34	16647	100
espresso.tail	59	0.35	150	0.90	366	2.20	599	3.60	1265	7.60	16647	100
espresso.ti	54	0.32	159	0.96	419	2.52	690	4.14	1408	8.46	16647	100
gcc.cexp	593	1.15	1704	3.31	3563	6.91	5413	10.50	9799	19.01	51555	100
gcc.jump	543	1.05	1729	3.35	3619	7.02	5281	10.24	9264	17.97	51555	100
gcc.stmt	567	1.10	1765	3.42	3682	7.14	5409	10.49	9215	17.87	51555	100
li	54	0.67	103	1.27	165	2.04	224	2.77	345	4.27	8083	100
sc.loada1	29	0.18	101	0.63	252	1.58	421	2.64	1054	6.60	15968	100
sc.loada2	44	0.28	133	0.83	360	2.25	552	3.46	1207	7.56	15968	100
sc.loada3	37	0.23	119	0.75	336	2.10	519	3.25	1206	7.55	15968	100
doduc	544	2.55	1485	6.97	2167	10.17	2404	11.28	3069	14.40	21313	100
fpppp	769	4.04	2106	11.08	2951	15.52	3233	17.01	3586	18.86	19012	100
mdljdp2	8	0.05	13	0.07	61	0.35	98	0.56	165	0.94	17493	100
mdljsp2	14	0.08	24	0.14	82	0.47	121	0.69	215	1.22	17560	100
ora	17	0.12	31	0.21	41	0.28	47	0.32	84	0.58	14526	100
spice2g6	18	0.06	100	0.31	356	1.11	530	1.65	716	2.22	32185	100
su2cor	72	0.35	189	0.92	394	1.91	620	3.00	1301	6.30	20636	100
swm256	19	0.13	28	0.18	50	0.33	57	0.38	63	0.42	15141	100
tomcatv	20	0.15	63	0.47	92	0.69	101	0.75	109	0.81	13422	100
wave5	101	0.44	276	1.20	471	2.04	608	2.63	1076	4.66	23087	100

percentage of the total number of instructions that represents.

If we look at the **doduc** benchmark using the doducin input, for example, we see that 544 instructions are responsible for 50% of all load references and those 544 instruction account for 2.55% of all load instructions in the benchmark (e.g. there are 21313 load instructions in doduc and 544/21313 = 2.55%). Continuing across the table, we see that 2167 (10.17%) of the load/store instructions account for 90% of the data references. The remaining 89.83% of load/store instructions generate only 10% of the data references.[2]

Table 3: Cumulative Load Instruction Cache Miss Counts

Bench-mark	Percent of Total Data Cache Misses											
	50%		75%		90%		95%		99%		100%	
	# of Insts	% of Total	# of Insts	% of Total	# of Insts	% of Total	# of Insts	% of Total	# of Insts	% of Total	# of Insts	% of Total
compress	4	0.13	7	0.23	9	0.29	14	0.46	28	0.92	3058	100
eqntott	3	0.06	6	0.13	19	0.41	33	0.71	96	2.06	4656	100
espresso.bca	5	0.03	12	0.07	67	0.41	187	1.14	605	3.68	16447	100
espresso.cps	18	0.11	72	0.43	237	1.42	423	2.54	998	6.00	16647	100
espresso.tail	26	0.16	84	0.50	235	1.41	385	2.31	876	5.26	16647	100
espresso.ti	26	0.16	98	0.59	302	1.81	520	3.12	1105	6.64	16647	100
gcc.cexp	249	0.48	845	1.64	1987	3.85	3065	5.95	5732	11.12	51555	100
gcc.jump	265	0.51	879	1.70	2032	3.94	3097	6.01	5538	10.74	51555	100
gcc.stmt	296	0.57	972	1.89	2176	4.22	3301	6.40	5890	11.42	51555	100
li	13	0.16	50	0.62	108	1.34	152	1.88	236	2.92	8083	100
sc.loada1	5	0.03	14	0.09	71	0.44	141	0.88	396	2.48	15968	100
sc.loada2	8	0.05	47	0.29	144	0.90	252	1.58	585	3.66	15968	100
sc.loada3	18	0.11	82	0.51	227	1.42	328	2.05	552	3.46	15968	100
doduc	123	0.58	301	1.41	563	2.64	787	3.69	1069	5.02	21313	100
fpppp	68	0.36	147	0.77	273	1.44	357	1.88	499	2.62	19012	100
mdljdp2	6	0.03	19	0.11	54	0.31	66	0.38	85	0.49	17493	100
mdljsp2	9	0.05	28	0.16	68	0.39	84	0.48	113	0.64	17560	100
ora	2	0.01	3	0.02	4	0.03	5	0.03	6	0.04	14526	100
spice2g6	3	0.01	11	0.03	60	0.19	105	0.33	234	0.73	32185	100
su2cor	41	0.20	137	0.66	362	1.75	547	2.65	1227	5.95	20636	100
swm256	14	0.09	27	0.18	37	0.24	40	0.26	50	0.33	15141	100
tomcatv	12	0.09	29	0.22	69	0.51	83	0.62	94	0.70	13422	100
wave5	24	0.10	78	0.34	187	0.81	296	1.28	514	2.23	23087	100

[2]This demonstrates a well known principle of program execution, that a small portion of the program is responsible for much of the execution effort. This is an effect of the 90/10 locality rule which states that a program spends approximately 90% of execution time in only 10% of the code.

Given that a small number of instructions are responsible for the majority of data references, it is reasonable to expect that this same effect would be reflected in the distribution of cache misses. Table 3 shows the frequency of cache misses for those load instructions that have the most misses; the layout of Table 3 is the same as that of Table 2. Overall, we find that not only does the 90/10 law still hold, but the miss pattern is even more clustered than the overall reference pattern. For almost all benchmarks, less than 5% of the total load instructions are responsible for causing over 99% of all cache misses.

The data in Tables 2 and 3 make it clear that in general a small number of load/store instruction have a disproportionately large effect on the cache miss rate when compared to the number of total data references they generate. This is not all that surprising if one considers program behavior. References to global variables and to local variables (even if they reference the procedure call stack) can account for the data references with a large hit rate, especially if one considers the looping behavior of programs. Examples of references that generate low hit rates would include references to items in a linked-list, or to some array with an odd stride.

4. Analysis of Caching Potential

Given that a small number of load instructions are responsible for generating the majority of data cache misses, we decided to measure the cache hit rate and the corresponding memory bandwidth required if these troublesome load instructions were prohibited from allocating space in the data cache. In order to accomplish this, we examined the cache behavior of each load instruction and identified those loads with the lowest cache hit rate. These instructions were marked **C/NA** (Cacheable/Non-Allocatable), which means that the data references generated by these load instructions will not invoke the allocation policy of the hardware cache management algorithm. It does **not** mean that the data reference will not be in the cache - the data item might be in the cache if a different instruction that **will** allocate on miss references that address.

It is important to stress that we are deciding whether or not to allocate based on the *instruction* address, not the effective address of the data reference. Thus, a cache lookup for an item is unaffected by whether it is marked C/NA or not - only the *allocation* on a miss is affected. We looked at both static (similar to [ASWR93]) and dynamic approaches to identifying and marking these C/NA instructions.

4.1. Static Method

We began by modeling a simple strategy in which all load instructions that do not meet a threshold for cache hit rate are marked C/NA. We looked at several threshold values, balancing the desire to remove poorly performing loads with the conflicting desire to utilize the cache for

as many references as possible. We finally settled on a threshold value of 75%, for a number of reasons. A lower value was too aggressive in removing load references from using the cache, and a higher value did not remove a sufficient number of load instructions to help performance. Furthermore, the 75% threshold also relates to the memory bandwidth requirements for a cache line replacement (32 bytes) and a 64-bit load reference (8 bytes), and is the same value settled on by [ASWR93].

4.1.1. Cache Hit Rate

Table 4 shows the cache hit rate after the poorest performing instructions were marked C/NA. The first column contains the name of the benchmark program and the second shows the number of instructions that were made C/NA (the numbers are expressed as a range since the count of instructions varied depending on the cache configuration). The remainder of the table shows the new hit rates for caches that are direct mapped, 2-way set associative, 4-way set

Table 4: Data Cache Hit Rate After Removal of Target Instructions (Static)

Bench-mark	# of C/NA Instructions	8K-byte Cache				16K-byte Cache			
		D	2	4	V	D	2	4	V
compress	5-9	77.39	83.07	83.20	89.18	78.25	83.56	83.69	88.95
eqntott	0-0	94.14	95.40	95.49	95.65	94.72	95.62	95.67	95.47
espresso.bca	84-217	84.01	85.10	85.65	87.16	86.44	87.48	87.73	85.24
espresso.cps	85-216	90.40	92.59	93.11	96.59	95.35	96.30	96.68	92.70
espresso.tail	94-265	89.53	94.27	94.94	97.10	95.34	97.75	98.60	93.76
espresso.ti	106-265	90.00	91.90	92.73	95.76	94.31	95.53	96.55	92.37
gcc.cexp	33-131	88.82	93.49	94.71	95.99	93.94	96.55	97.39	93.26
gcc.jump	101-337	88.04	92.21	93.49	95.36	93.11	95.74	96.59	91.94
gcc.stmt	223-614	87.87	92.07	93.26	95.20	93.26	95.80	96.63	91.75
li	54-135	86.26	90.99	92.28	93.96	91.15	93.73	94.79	91.20
sc.loada1	160-296	77.03	79.52	80.61	82.07	80.22	81.26	85.36	80.41
sc.loada2	245-386	83.47	85.19	85.68	86.57	85.14	86.45	86.87	85.71
sc.loada3	127-274	90.63	92.22	92.91	94.91	93.92	95.04	95.36	93.30
Int Ave		86.74	89.85	90.62	92.73	90.40	92.37	93.22	90.47
doduc	400-707	85.51	88.97	93.23	94.14	88.73	94.59	95.15	91.28
fpppp	113-263	92.92	97.15	97.70	97.52	94.92	98.99	99.66	95.78
mdljdp2	118-143	78.71	82.28	85.40	88.92	87.43	88.52	89.17	83.43
mdljsp2	51-74	94.88	96.63	97.39	98.44	97.20	98.40	98.49	97.33
spice2g6	376-446	45.80	48.42	51.50	52.49	51.36	53.97	73.60	48.13
su2cor	929-1650	45.50	45.95	49.28	68.95	50.61	52.25	52.13	62.63
swm256	69-119	71.53	72.11	65.61	93.20	92.68	92.36	91.41	92.83
tomcatv	39-78	52.04	50.90	47.07	76.30	66.45	74.76	77.89	72.09
wave5	133-240	86.54	88.74	89.19	94.43	93.18	94.77	94.78	91.29
FP Ave		72.60	74.57	75.15	84.93	80.28	83.18	85.81	81.64

associative and direct mapped with a 16 line victim cache. Comparing the cache hit rates from the unmodified case (found in Table 1) to those in Table 4 reveals a slight decrease in the average cache hit rate across all cache configurations.

4.1.2. Memory Activity

Another important measure of the effectiveness of this technique is the amount of memory traffic that ensues. This information is shown in Table 5. There are 4 classifications of load instructions shown in this table:

Cacheable/Non-Allocatable - those load instructions that have been identified as C/NA.

Increased - load instructions that are cached and have a higher miss rate because of the transformation.

No Change - load instructions that are cached and maintain their original cache hit activity.

Decreased - load instructions that are cached and have a lower miss rate because of the transformation.

In order to reduce the tremendous amount of data generated, we show information that has been averaged over all benchmarks for an 8K-byte direct mapped cache.

In order to better understand what is happening, imagine a situation where items A and B both map to the same cache line and are repeatedly accessed. In this case each reference will experience a high miss rate. However, by prohibiting one of these items (A, for example) from allocating the cache line on a miss, the remaining item (B) will experience a much lower miss rate due to the elimination of contention. This effect is shown in the **Decreased** field of the table.

Table 5: Analysis of Average Memory Reference Activity (Static)

Instruction Classification	Number of Instructions	Percent of Refs	Memory Refs Pre-Transformation	Memory Refs Post-Transformation	Change
C/NA	315	16.06	50047859	54961970	9.82
Increase	270	9.92	2236595	12855381	474.77
No Change	4898	46.17	4951049	4951049	0.00
Decrease	811	27.85	7909096	6376529	-19.38
Not Referenced	14824	0.00	0	0	0.00
Total	21118	100.00	65144599	79144929	21.49

On the other hand, some items with a high miss rate actually perform a useful function by bringing a line into the cache that will be later referenced by other load instructions. By eliminating the cache line allocation of these instructions, the cache hit performance of these other loads is decreased - this is reflected in the **Increased** field.

The first column of Table 5 shows the load instruction classification. The second and third columns show the average number of load instructions and the percentage of the total load references these instructions perform, respectively. The fourth column contains the average number of references to memory (the number of cache misses) that occurred before any loads were marked C/NA, and the fifth column shows the number of memory references after the C/NA transformations were performed.

In this table we see that the total number of memory references has increased by over 21%. This is due in large part to the 475% increase in the number of cache misses experience by 270 of the non-C/NA load instructions. This approach is apparently being too aggressive in marking loads C/NA - by blindly removing those loads with poor performance, we are often simply shifting a miss from that instruction to the next instruction referencing that location. Clearly, a more refined approach to marking certain high miss load instructions C/NA is called for.

4.1.3. Memory Bandwidth Utilization

A potentially more meaningful measure of the demands made on the memory system is to determine the total amount of data (in bytes) that must be fetched from the memory system. Since we used a system configuration in our simulations similar to that of the Alpha (32 byte cache lines and single references being 8 bytes), we were able to determine the total number of bytes that the memory system must process and the impact of these C/NA transformations on the bus activity.

We calculated the total bus utilization for the Static case by multiplying the number of allocatable misses by 4 (32/8), and adding the number of references to instructions marked C/NA. Dividing this number by the base case bus utilization gives us the percentage of the original bus bandwidth needed by the Static approach.

The results of these calculations are shown in Table 6. As in Table 4, the first column contains the name of the benchmark program, and the second column contains the number of instructions that were marked C/NA. The remaining columns show what percent of the original bandwidth for that cache configuration was required after the transformation of the C/NA loads. So, for example, after the C/NA transformations the compress program run on a 16K-byte 2-way set associative data cache requires only 45.39% of the bandwidth required by the same program run on the same hardware without the C/NA transformations.

Table 6: Relative Memory Bandwidth Requirements After Removal of Target Instructions (Static)

Bench-mark	C/NA Instr	8K-byte Cache				16K-byte Cache			
		D	2	4	V	D	2	4	V
compress	5 - 9	52.25	43.73	44.38	43.39	53.31	45.39	45.69	44.91
eqntott	0 - 0	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
espresso.bca	84 - 217	73.47	72.03	71.22	71.89	70.09	67.24	66.37	68.53
espresso.cps	85 - 216	81.29	78.84	79.04	77.37	81.95	75.67	71.33	80.60
espresso.tail	94 - 265	74.54	78.36	75.67	78.98	81.16	93.30	97.80	95.45
espresso.ti	106 - 265	77.54	72.09	71.34	71.41	82.00	77.85	73.23	78.59
gcc.cexp	33 - 131	83.00	89.97	94.66	90.99	88.50	97.25	101.54	95.88
gcc.jump	101 - 337	81.07	85.70	89.81	84.18	86.31	95.54	98.37	93.91
gcc.stmt	223 - 614	79.81	84.28	86.86	80.97	87.40	93.92	98.12	91.20
li	54 - 135	81.29	104.95	109.67	106.73	80.67	114.15	118.95	116.79
sc.loada1	160 - 296	61.12	65.86	65.60	64.07	64.19	65.74	62.89	65.34
sc.loada2	245 - 386	71.00	79.12	81.62	79.85	77.70	84.08	86.11	81.56
sc.loada3	127 - 274	68.01	69.91	72.37	77.82	70.46	67.15	65.10	70.93
Int Ave		75.72	78.83	80.17	79.05	78.75	82.87	83.50	83.36
doduc	400 - 707	59.23	72.24	89.24	69.94	63.23	85.81	90.69	72.97
fpppp	113 - 263	70.08	70.24	69.86	75.30	81.43	74.37	78.77	79.68
mdljdp2	118 - 143	97.45	91.99	99.69	97.46	95.05	93.93	100.16	94.71
mdljsp2	51 - 74	102.01	102.32	102.87	101.88	101.05	101.60	101.57	102.17
spice2g6	376 - 446	50.61	54.24	54.69	19.92	65.70	72.21	69.81	69.36
su2cor	929 - 1650	34.50	35.28	54.21	106.47	49.48	54.35	53.98	109.01
swm256	69 - 119	56.15	31.51	32.71	33.42	97.29	96.76	94.61	98.45
tomcatv	39 - 78	57.95	54.73	84.28	117.78	66.00	105.59	112.68	116.11
wave5	133 - 240	75.64	86.90	81.32	100.84	103.27	103.53	102.25	103.82
FP Ave		67.07	66.61	74.32	80.33	80.28	87.57	89.39	94.03

The table shows a significant overall decrease in the required memory bandwidth. In particular, Table 6 shows that the static scheme used in conjunction with a victim cache results in an average decrease in bus activity of nearly 20%.

4.2. Improved Static Method

In order to improve the performance of the simple static technique, we needed to reduce the number of instructions marked C/NA. We accomplished this by associating with each cache line the address of the instruction that was responsible for bringing that line into the cache. This information allowed us to distinguish between misses that bring data into the cache that is later referenced (performed a useful prefetch) and those misses that are not referenced before the data is returned to memory due to the cache replacement strategy. Only instructions that do not perform a useful prefetch are marked C/NA. We refer to this as the *Improved* Static Method.

In our simulations, we implemented this modification to the static approach in the following manner: We used the same 75% hit rate threshold to identify potential C/NA instructions. Once these were identified, they were analyzed to determine if they were performing a useful prefetch. If at least 3/4 of the misses prove to be prefetches, then the instruction was removed from the C/NA list, resulting in a less aggressive application of C/NA.

4.2.1. Cache Hit Rate

As shown in Table 7, the Improved Static approach provides hit rates very close to those presented in Table 1. Cache performance was only slightly worse for the integer benchmarks (on average) and actually improved for the floating point benchmarks.

Table 7: Data Cache Hit Rate After Removal of Target Instructions (Improved Static)

Bench-mark	# of C/NA Instructions	8K-byte Cache				16K-byte Cache			
		D	2	4	V	D	2	4	V
compress	5-6	79.45	83.07	83.20	88.95	80.31	83.56	83.69	89.18
eqntott	0-16	92.95	93.95	94.03	94.00	93.44	94.34	94.37	94.16
espresso.bca	43-112	87.74	88.80	89.29	89.04	90.26	91.31	91.58	90.95
espresso.cps	39-120	92.45	94.04	94.50	94.10	95.92	96.74	97.08	96.92
espresso.tail	34-129	92.50	95.57	96.19	95.46	95.78	97.91	98.68	97.43
espresso.ti	56-145	92.39	94.01	94.69	94.35	95.39	96.58	97.05	96.71
gcc.cexp	9-64	89.94	94.08	95.26	93.92	94.42	96.89	97.71	96.41
gcc.jump	20-144	89.79	93.50	94.67	93.38	94.07	96.56	97.28	96.23
gcc.stmt	67-290	89.75	93.53	94.56	93.36	94.27	96.60	97.39	96.17
li	14-49	90.08	94.60	95.70	94.85	94.03	96.59	97.31	96.87
sc.loada1	69-123	84.55	85.69	86.77	81.67	86.27	87.43	87.60	82.91
sc.loada2	90-106	86.46	88.63	89.46	90.33	88.80	90.26	90.25	91.11
sc.loada3	39-103	93.03	94.49	94.98	94.71	95.03	95.93	96.24	96.14
Int Ave		89.31	91.84	92.56	92.16	92.15	93.90	94.33	93.94
doduc	78-254	89.44	92.97	95.96	94.19	91.99	96.51	97.05	96.15
fpppp	30-110	94.97	97.80	98.19	96.97	96.57	99.16	99.71	98.17
mdljdp2	32-49	85.01	88.15	91.18	89.16	92.82	93.73	94.50	93.89
mdljsp2	11-18	95.33	96.98	97.67	97.59	97.35	98.52	98.60	98.58
spice2g6	163-202	-	-	-	53.72	-	-	-	49.54
su2cor	165-1522	49.53	50.96	58.96	80.25	58.16	60.46	60.23	68.82
swm256	30-38	75.71	73.10	66.06	93.23	92.88	92.73	92.12	92.87
tomcatv	0-30	68.74	64.58	67.18	85.82	78.73	85.99	88.06	88.06
wave5	40-88	89.17	91.20	91.37	93.56	94.64	95.84	95.81	95.65
FP Ave		80.99	81.97	83.32	91.35	87.89	90.37	90.76	91.52

4.2.2. Memory Activity

An examination of the resultant memory activity (Table 8) reveals several interesting observations. For example, the number of instructions in the C/NA class dropped from 331 to 177, indicating that there are a lot of instructions with high miss rates that are actually performing useful work (prefetching). As one would expect, the increase in memory activity due to the C/NA instructions dropped as well (from 9.9% to 1.3%). However, the most dramatic change is in the number of instructions that have their miss rate increase - this drops from 270 to 117, resulting in a reduction in memory activity from 475% to 34%.

The most significant number in Table 8 is the total change in memory activity. This shows that by applying the improved Static method to a program the cache hit rates can be maintained (and slightly improved) while simultaneously decreasing the amount of traffic to memory.

4.2.3. Analyzing Memory Bandwidth Utilization

Table 9 shows how the improved Static scheme affects the bus bandwidth. The format of the table is identical to table 6.

The table shows that the Improved Static scheme has improved the memory bandwidth utilization over the original Static scheme. This was achieved by reducing the memory requirements for more than 1/2 of the cache misses, those that did not allocate a new cache line. In particular, Table 9 shows that the improved static scheme used in conjunction with a victim cache results in an average decrease in bus activity by as much as 60% for some benchmarks.

5. Dynamic Cache Model

It is clear that the use of the improved static approach will improve data cache performance. However, the static approach requires training runs of the program, and the introduction of new instructions in order to specify the alternate cache operation. Both of these factors

Table 8: Analysis of Average Memory Reference Activity (Improved Static)

Instruction Classification	Number of Instructions	Percent of Refs	Memory Refs Pre-Transformation	Memory Refs Post-Transformation	Change
C/NA	177	9.92	36344490	36812026	1.29
Increase	117	3.41	2362602	3173837	34.34
No Change	5195	54.21	10898840	10898840	0.00
Decrease	805	32.46	15538667	12596981	-18.93
Not Referenced	14824	0.00	0	0	0.00
Total	21118	100.00	65144599	63481684	-2.55

Table 9: Relative Memory Bandwidth Requirements After Target Instruction Removal (Improved Static)

Bench-mark	C/NA Instr	8K-byte Cache				16K-byte Cache			
		D	2	4	V	D	2	4	V
compress	5 - 6	48.13	43.73	44.38	43.39	48.85	45.39	45.69	44.91
eqntott	0 - 16	100.00	100.00	100.00	85.51	100.00	100.00	100.00	85.98
espresso.bca	43 - 112	70.55	68.36	69.34	69.79	66.18	62.27	61.12	64.18
espresso.cps	39 - 120	77.82	75.76	76.12	74.72	81.88	75.03	69.38	80.32
espresso.tail	34 - 129	85.71	92.06	97.90	93.39	86.20	93.36	97.48	97.86
espresso.ti	56 - 145	74.23	68.48	67.61	68.99	76.55	71.02	69.86	72.38
gcc.cexp	9 - 64	81.40	89.21	93.02	90.92	87.91	95.49	99.10	94.78
gcc.jump	20 - 144	78.48	82.38	85.67	84.14	84.00	91.52	94.32	91.29
gcc.stmt	67 - 290	77.13	80.93	82.88	81.27	85.08	90.44	93.98	89.56
li	14 - 49	76.52	95.16	95.30	95.65	73.18	97.23	98.85	98.09
sc.loada1	69 - 123	58.81	61.93	61.85	44.73	60.82	60.54	60.05	43.78
sc.loada2	90 - 106	59.49	66.68	68.96	77.82	64.95	67.48	70.32	79.58
sc.loada3	39 - 103	69.93	71.43	74.18	39.73	67.32	64.43	62.00	34.49
Int Ave		73.71	76.62	78.25	73.08	75.61	78.02	78.63	75.17
doduc	78 - 254	55.91	72.62	87.42	74.98	58.76	84.56	90.04	75.77
fpppp	30 - 110	63.73	66.06	69.02	72.17	72.16	71.46	76.79	81.02
mdljdp2	32 - 49	97.13	90.44	96.80	96.42	96.58	94.77	99.80	95.52
mdljsp2	11 - 18	99.63	99.54	99.91	99.21	99.73	99.80	99.77	99.80
spice2g6	--	--	--	--	--	--	--	--	--
su2cor	165 - 1522	32.55	32.83	50.63	100.33	45.19	49.46	49.20	100.90
swm256	30 - 38	54.25	30.74	32.38	33.39	96.64	95.58	94.89	98.42
tomcatv	0 - 30	56.29	59.05	78.89	100.47	59.69	96.15	100.00	100.00
wave5	40 - 88	70.56	81.55	76.49	92.53	96.78	97.13	97.33	97.26
FP Ave		58.89	59.20	65.73	76.57	69.50	76.55	78.65	90.77

markedly decrease the applicability of this approach. Our goal is to develop a scheme that will provide the same performance enhancement transparently.

In order to select which items should be marked C/NA, we turn to the body of work on branch prediction strategies. There has been a great amount written about branch prediction strategies recently [CaGr94, FiFr92, PaS92, Smit81, YeP91, YeP92, YeP93]. Briefly, dynamic branch prediction strategies collect run-time information about branch behavior to predict whether a branch will be taken in the future. Typically, these strategies associate several bits of information with a branch instruction. This information is updated each time the branch instruction is executed and is used to make a prediction about the branch instruction's behavior.

In a similar way, several bits can be associated with a load or store instruction. A table, similar to a branch prediction table, can be maintained which tracks whether the data referenced by a load or store instruction caused a miss in the data cache. This information can then be used

to decide whether an instruction should be marked C/NA.

In our study, we simulated *miss prediction tables* using a 2-bit counter associated with each load or store instruction. A miss prediction counter is initially set to zero and it is incremented each time a load or store causes a cache miss. If the load or store instruction causes a cache hit, the counter is decremented. When the counter enters its highest state ("11"), the instruction is marked C/NA.

It is worth stressing again that the counters simply inform the cache allocation hardware whether the data should be placed in the cache on a miss. Regardless of the state of the counters, a data cache lookup is performed on every data reference, since the data may have been brought into the cache by some other instruction. Thus, there is the possibility that in one phase of program execution an instruction will be prevented from caching its data, but in a different phase of the program it will be allowed to do so. This differs from the static methods presented in the previous section because the load instruction's status, whether it is C/NA or not, changes throughout the execution of the program. Since the C/NA marking is maintained as part of the miss prediction table, it does not require new types of instructions to be added to the architecture as would be the case with a static scheme.

Using 2-bit counter miss prediction schemes, we performed several sets of experiments. Initially the size of the miss prediction table was unlimited in order to evaluate how well the a 2-bit scheme can track a hit/miss history. In later experiments, we fixed the size of the miss prediction table.

Table 10 summarizes the average memory reference activity when using 2-bit counters for miss prediction on the SPEC benchmarks. As in Tables 5 and 8 for the static schemes, the results are averaged across all the benchmarks for an 8K byte direct mapped cache configuration. Unlike the results for the static schemes, the C/NA instruction classification is broken down into 3 categories. This is necessary because with a dynamic scheme an instruction can be in the C/NA state only part of the time. Thus, we decided on the three categories: (1) *<5 C/NA*, the instruction was in the C/NA state for less than 5% of its references, but for at least one reference, (2) *5-95 C/NA*, the instruction was in the C/NA state for between 5% and 95% of its references, and (3) *> 95 C/NA*, the instruction was in the C/NA state for 95% or more of its references. Another difference in these tables is the separation of the post-transformation misses into two types, those misses that do not cause a cache line replacement (because the load instruction is in the C/NA state), and those misses that do cause a line replacement.

Looking at the results shown in Table 10, we first note that the number of instructions that spend some amount of time in the C/NA state is much larger than for either of the static

methods. This is seen by comparing the first line (C/NA) of Tables 5 and 8 with the first three lines of Table 10. Clearly the dynamic behavior of the program has a significant impact on whether the data item for a particular load instruction will be found in the cache. Further comparisons between the static results and dynamic results indicate that, as one might expect, the 2-bit dynamic scheme is moving instructions from the Increase, No Change and Decrease categories into one of the C/NA categories. Overall, this shift increased the average number of memory references by 63.91%.

As with the first static scheme, the 2-bit miss prediction scheme is too aggressive in classifying instructions as C/NA. Too quickly marking an instruction as C/NA results in the large 63.91% increase in memory references. As a next step, we modified the 2-bit scheme to mimic the Improved Static scheme. In the Improved Dynamic scheme, each line of the cache has associated with it the address of the load instruction that brought that line into the cache. On a cache hit, the 2-bit counter associated the instruction that caused the hit is decremented and in addition, the 2-bit counter associated with the instruction that brought the cache line into the cache is also decremented. Thus, those instructions that do useful prefetching of data for other instructions are not marked as C/NA. Results of simulations using the Improved Dynamic 2-bit miss prediction table are shown in Table 11.

As can be seen in Table 11, the number of instructions that are placed in the C/NA category is much smaller when compared with those in Table 10. This results in reducing the number of memory references such that there is only a 0.28% increase compared to a conventional cache. The small change in the number of memory references and the small number of instructions in the C/NA categories indicates that perhaps this improved strategy is too conservative in marking instructions whose data should not be cached.

Table 10: Analysis of Average Memory Reference Activity - Dynamic 2-bit Counter Scheme

Instruction Classification	Number of Instructions	Percent of Refs	Memory Refs. Pre-Transformation	Post-Trans. Refs.		Change
				C/NA	Non-C/NA	
< 5 C/NA	480	29.00	10928660	702483	5335239	-44.75
5-95 C/NA	1021	14.37	15866569	22358298	6036362	78.96
> 95 C/NA	449	15.69	31398171	67933394	30426	116.46
Increase	335	3.67	216738	0	281825	30.03
No Change	3438	22.68	2025516	0	2025516	0.00
Decrease	571	14.59	4708945	0	2073066	-55.98
Not Referenced	14824	0.00	0	0	0	0.00
Total	21118	100.00	65144599	90994175	15782434	63.91

Table 11: Analysis of Average Memory Reference Activity - Improved Dynamic 2-bit Counter Scheme

Instruction Classification	Number of Instructions	Percent of Refs	Memory Refs. Pre-Transformation	Post-Trans. Refs. C/NA	Post-Trans. Refs. non-C/NA	Change
< 5 C/NA	94	13.20	9089400	182664	7811943	-12.04
5-95 C/NA	83	1.63	3042361	1696542	1698745	11.60
> 95 C/NA	49	2.96	10266889	12771268	28415	24.67
Increase	66	2.41	1000689	0	1149173	14.84
No Change	5417	51.39	28625580	0	28625580	0.00
Decrease	585	28.42	13119680	0	11360887	-13.41
Not Referenced	14824	0.00	0	0	0	0.00
Total	21118	100.00	65144599	14650474	50674743	0.28

As was the case for the static schemes, an analysis can be made of the memory bandwidth requirements of the dynamic schemes. This analysis accounts for transferring an entire cache line from memory on a cache misses and also referencing data items that will not be cached. Table 12 provides a summary of the results of this analysis for each of the SPEC benchmarks. The data in the table is a computation of the percentage of memory bandwidth required compared to a conventional cache scheme that does not use a miss prediction table. The columns of the table show the average memory bandwidth required for 8K-byte and 16-byte direct mapped caches, with and without victim buffers, using the 2-bit dynamic and the 2-bit improved dynamic strategies.

The results in Table 12 indicate that the bandwidth requirements of the dynamic schemes are not reduced as substantially as with the static schemes. This makes sense since with the static schemes, we have more information when marking which instructions should be C/NA. Nonetheless, for most programs the bandwidth requirements are reduced, and in several cases the reductions are substantial. Furthermore, the data in Table 10-12 indicate the trade-offs between caching data items and the resultant bandwidth requirements. With the more aggressive dynamic strategy, where more instructions are marked C/NA, there is more memory activity. However, the memory activity is for a single data item instead of an entire cache line. Thus, there is a reduction in the required memory bandwidth. On the other hand, with the Improved Dynamic strategy, there is less memory activity, but the required memory bandwidth is higher since the memory activity involves more fetches of entire cache lines.

The final set of experiments that we performed involved fixing the size of the miss prediction table. For this set of experiments we looked at a direct mapped cache of 8K-bytes using the first dynamic prediction strategy. The miss prediction table was fixed 4-way set associative, while the table size was varied. The results of these experiments are summarized in Figure 1.

Table 12: Relative Memory Bandwidth Requirements After Dynamic Instruction Removal

Bench-Mark	8K-byte Cache				16K-byte Cache			
	2-bit		2-bit improved		2-bit		2-bit improved	
	Direct	Victim	Direct	Victim	Direct	Victim	Direct	Victim
compress	41.08	30.99	49.60	36.25	33.21	33.40	54.59	41.16
eqntott	171.20	98.01	79.74	75.21	127.32	99.72	79.41	75.85
espresso.bca	75.87	66.17	99.77	99.85	70.18	60.47	99.88	99.91
espresso.cps	85.88	73.77	92.60	91.34	84.67	78.60	92.90	91.09
espresso.tail	95.32	97.67	99.05	98.93	102.20	105.52	99.49	99.38
espresso.ti	79.35	66.86	85.73	81.83	88.79	71.86	86.55	81.96
gcc.cexp	67.48	76.52	91.82	94.58	76.96	85.70	95.77	96.88
gcc.jump	70.15	74.23	92.85	93.83	77.91	84.03	93.85	95.93
gcc.stmt	72.09	73.67	92.86	92.92	81.90	84.02	94.39	95.03
li	100.16	96.63	91.70	99.32	109.01	96.71	89.82	99.77
sc.loada1	65.49	60.04	62.32	61.35	68.70	60.86	62.24	62.84
sc.loada2	74.68	76.49	73.76	77.80	78.04	77.99	75.60	80.23
sc.loada3	68.76	71.31	79.91	75.06	70.83	72.02	74.20	71.44
Int Ave	82.12	74.03	83.98	82.94	82.29	77.76	84.51	83.96
doduc	50.40	70.40	96.03	98.14	54.03	75.71	95.30	97.56
fpppp	55.13	54.78	96.29	94.77	64.56	64.14	96.86	94.78
mdljdp2	90.43	93.69	99.18	99.08	96.41	98.59	99.43	99.49
mdljsp2	108.74	102.32	99.91	99.91	129.80	99.80	100.00	100.00
spice2g6	53.82				65.27			
su2cor	30.45	96.31	74.75	100.72	39.33	96.94	82.66	100.33
swm256	38.89	0.00	99.53	33.39	98.42	0.00	98.45	98.40
tomcatv	52.72	100.00	86.31	100.00	56.64	100.00	96.79	100.00
wave5	78.57	95.22	94.82	92.47	113.24	104.01	99.07	99.88
FP Ave	63.17	76.59	93.35	89.81	81.55	79.90	96.07	98.80

In Figure 1, we have plotted the table size on horizontal axis, while the hit rate in the table and the resultant bandwidth requirements are plotted on the vertical axis. As can be seen in the center of the figure, a miss table of 256 entries reduces the average memory bandwidth requirements to a value very close to what one would get with an infinitely large miss prediction table.

6. Conclusions and Future Work

In this work, we have investigated the potential of improving data cache performance by being more selective in what data items are cached. This work was motivated by the apparent limitations in the size, organization and speed of first level data caches. To make the data cache smarter with what items it caches, we first examined and analyzed which instructions generated data cache misses. In this analysis, we confirmed and expanded on the results of other work that indicates a very small number of instructions are responsible for a very large percentage of data cache misses.

We have performed a preliminary study of the feasibility of incorporating a hardware based speculative prefetch unit to extend this work. Caches work well in exploiting the spatial and temporal locality of certain data references, but fail when locality is missing. Prefetch works well when there is regularity in the access pattern regardless of locality. By incorporating a hardware prefetch unit for C/NA items, it may be possible to hide the latency of even those loads that have little locality.

Another possible application of a dynamic scheme similar to the one described in this paper involves dynamically configuring a cache coherence protocol to fit the requirements for each load instruction; instructions that are likely to share data could use a different protocol from those that access local data.

We believe that using a method of dynamic configuration of cache operations like the one described in this paper can have broad applicability. Similar schemes can not only improve the performance of the cache, but can allow for other hardware based memory enhancements to be selectively applied.

7. References

[ASWR93] S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau and R. Gupta, ‘‘Predictability of Load/Store Instruction Latencies’’, *Proceedings of the 26th Annual International Symposium on Microarchitecture*, Austin, Texas (December 1-3, 1993), pp. 139-152.

[CaGr94] B. Calder and D. Grunwald, ‘‘Fast and Accurate Instruction Fetch and Branch Prediction’’, *Proceedings of the 21th Annual Symposium on Computer Architecture*, Chicago, Illinois (April 18-21, 1994), pp. 2-11.

[CaPo] D. Callahan and A. Porterfield, ‘‘Data Cache Performance and Supercomputer Applications’’, *Proceedings of Supercomputing ’90*, pp. 564-572.

[ChBa95] T. Chen and J. Baer, ‘‘Effective Hardware Based Data Prefetching for High-Performance Processors’’, *IEEE Transactions on Computers,* vol. 44, no. 5 (May 1995), pp. 609-623.

[FiFr92] J. A. Fisher and S. M. Freudenberger, ‘‘Predicting Conditional Branch Directions from Previous Runs of a Program’’, *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA (October 12-15, 1992), pp. 85-95.

[HePa90] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufman, San Mateo, California, (1990).

[Joup90] N. Jouppi, ‘‘Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers’’, *Proceedings of the Seventeenth Annual International Symposium on Computer Architecture,* vol. 18, no. 2 (May 1990), pp. 364-373.

[KlLe91] A. C. Klaiber and H. M. Levy, ‘‘An Architecture for Software-Controlled Data Prefetching’’, *Proceedings of the Eighteenth Annual International Symposium on Computer Architecture*, Toronto, Canada (May 27-30, 1991), pp. 43-53.

[MuQF91] J. M. Mulder, N. T. Quach and M. J. Flynn, ‘‘An Area Model for On-Chip Memories and its Application’’, *IEEE Journal of Solid-State Circuits,* vol. 26, no. 2 (February 1991), pp. 98-105.

[PaS92] S. Pan, K. So and J. T. Rahmeh, ‘‘Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation’’, *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA (October 12-15, 1992), pp. 76-84.

[Smit81] J. E. Smith, ‘‘A Study of Branch Prediction Strategies’’, *Proceedings of the Eighth Annual International Symposium on Computer Architecture,* Minneapolis, Minnesota (May 1981), pp. 135-148.

[SrWa94] A. Srivastava and D. W. Wall, ‘‘Atom: A system for building custonized program analysis tools’’, *Proceedings of the ACM SIGPLAN Notices 1994 Conference on Programming Languages and Implementations*(June 1994), pp. 196-205.

[WaRP92] T. Wada, S. Rajan and S. A. Przybylski, ‘‘An Analytical Access Time Model for On-Chip Cache Memories’’, *IEEE Journal of Solid-State Circuits,* vol. 27, no. 8 (August 1992), pp. 1147-1156.

[YeP91] T. Yeh and Y. Patt, ‘‘Two-Level Adaptive Training Branch Prediction’’, *Proceedings of the 24th Annual International Symposium on Microarchitecture*, Albuquerque, New Mexico (November 18-20, 1991), pp. 51-61.

[YeP92] T. Yeh and Y. Patt, ‘‘Alternative Implementations of Two-Level Adaptive Training Branch Prediction’’, *Proceedings of the Nineteenth Annual International Symposium on Computer Architecture*, Queensland, Australia (May 19-21, 1992), pp. 124-134.

[YeP93] T. Yeh and Y. Patt, ‘‘A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History’’, *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, San Diego, CA (May 16-19, 1993), pp. 257-266.