A Modified Approach to Data 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. In this paper, a detailed characterization of data cache behavior for individual load instructions is given. We show that by selectively applying cache line allocation according the characteristics of individual load instructions, overall performance can be improved for both the data cache and the memory system. This approach can improve some aspects of memory performance by as much as 60 percent on existing executables.

1. Introduction

The *average data access time* is a measure of the time it takes to read a data item from memory. Since most programs need to access data, minimizing this term is crucial to achieving high performance. Unfortunately, access time to off-chip memory (measured in processor clock cycles) has increased dramatically as the disparity between main memory access times and processor clock speeds widen. Since there is no indication that dynamic memory access times will decrease significantly in the near future, this situation is unlikely to change for a while.

In order to minimize the impact of slow main memory access times, several strategies are available. Most machines now include a first-level cache, which is designed to reduce the average data access time by capturing the most frequently used data items. If necessary, one can also 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.

Another option is to interleave main memory, so that each word of a cache line does not have to experience the full latency of the main memory. Widening the bus between the primary cache and the main memory (or second-level cache) is also an option. Both of these approaches 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.

As multiple-issue processors continue to increase the number of instructions issued on each cycle, there has been a 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 warranted. In this paper, we examine the potential of reducing the average data access time 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

In a system with a cache, the average access time for a memory reference is a function of the hit rate in the cache, the corresponding miss rate and the the miss penalty. 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.

This work was supported by National Science Foundation Grants CCR-94-03651, CCR-92-13627, MIP-92-57259, and grants from the SUN Microsystems and Tektronix corporations.

2.1. Reducing Miss Rate

The miss rate is best reduced by increasing the size and/or the associativity of the cache. Unfortunately, in the design of todays high-performance processors, it is difficult to substantially increase either of these terms 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 relationship between cache access time, 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, for example, 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 primary cache size much beyond 16K bytes.

2.2. Reducing Miss Penalty

A number of studies have proposed techniques (either compiler-based or hardware-based) that reduce the miss penalty of the cache by performing some type of data prefetch [CaPo, ChBa95, KlLe91]. 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 the associated miss penalty will not be experienced. 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. IBM uses a similar hardware approach [EKPP93] in which they associate previous miss behavior with a load instruction and use that information to do prefetching.

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.

3. Deciding What to Cache

Instead of concentrating on miss penalty, cache size or cache associativity, we decided to look at the source of cache misses. One could potentially reduce the miss rate of the data cache by simply not caching those data references that lead to a poor miss rate. As Abraham, et el. [ASWR93] point out, a large percentage of the data misses are caused by a very small number of instructions. Instead of using this information to make prefetching decisions, 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 data cache. Such an approach has the potential to more effectively utilize the cache because instructions that generate a large number of cache misses are removing more heavily utilized data items from the cache. In addition, if we do not cache the data associated with high-miss-rate instructions, memory bandwidth requirements could be reduced since these references would only request a single word from memory, instead of an entire cache line.

Since the study by Abraham, et el. [ASWR93] did not look at an extensive set of benchmark programs, we began by performing experiments similar to theirs in which we measured the miss rate associated with individual load and store instructions for a more extensive set of programs. Using the ATOM program trace facilities [SrWa94] and the SPEC92 suite of benchmarks, such statistics were relatively straight-forward to gather. Each program in the SPEC 92 suite was instrumented 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 [Joup90].

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.

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 data references and data cache misses caused 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 8 columns which are labeled with a percentage of total load references and total data cache misses each have two sub-columns indicating the total number of instructions that caused that percentage and the percentage of the total number of instructions that represents.

If we look at the **compress** benchmark, for example, we see that 34 instructions are responsible for 75% of all load references and those 34 instruction account for 1.11% of all load instructions in the benchmark (e.g. there are 3058 load instructions in compress and 34/3058 = 1.11%). Continuing across the table, we see that 54 (1.77%) of the load/store instructions account for 95% of the data references. The remaining 98.23% of load/store instructions

Table 1: Data Cache Hit Rate for SPEC Benchmarks

SPEC Benchmark	Input File	# of Load Refs	8K Cache Hit Rate (%)				16K Cache Hit Rate (%)			
			D	2	4	V	D	2	4	V
compress	in	24434068	79.73	84.77	85.35	84.38	81.33	85.94	86.3	85.41
eqntott	int_pri_3.eqn	231129466	94.14	95.4	95.49	95.47	94.72	95.62	95.67	95.65
espresso.cps	cps.in	110649363	92.82	94.49	95.04	94.45	96.07	96.84	96.97	97.08
espresso.tail	tail.in	218814920	92.32	95.6	96.26	95.47	95.58	98	98.68	97.44
espresso.ti	ti.in	123094748	92.74	94.37	95.12	94.69	95.62	96.57	96.99	96.84
gcc.insn	insn.i	42239266	91.19	94.06	95.02	93.87	94.5	96.53	97.11	96.09
gcc.integrate	integrate.i	18050705	90.68	93.97	95.09	93.62	94.27	96.58	97.27	96.11
gcc.stmt	stmt.i	34338681	90.86	94.13	95.3	93.86	94.64	96.8	97.47	96.41
gcc.tree	tree.i	15088536	90.81	94.04	95.22	93.79	94.86	97.07	97.74	96.61
li	li_input.lsp	1923073359	89.75	94.58	95.68	94.81	93.4	96.59	97.32	96.85
sc.loada1	loada1	325317586	84.43	86.38	87.17	86.55	86.37	87.64	87.77	87.51
sc.loada2	loada2	365851437	88.44	90.44	91.14	90.87	90.18	91.76	91.95	91.63
sc.loada3	loada3	104774876	92.73	94.31	94.96	94.7	94.77	95.62	95.83	95.66
Int Ave			90.05	92.81	93.60	92.81	92.79	94.74	95.16	94.56
doduc	doducin	337197266	88.54	92.96	96.36	94.01	91.31	96.74	97.29	95.78
ear	ref.m22	3833127750	96.73	97.62	97.52	97.72	98.69	99.53	99.93	99.28
fpppp	natoms	1529995204	94.59	97.64	98.05	96.81	96.44	99.12	99.7	98.12
hydro2d	hydro2d.in	1376293089	81.5	83.69	84.79	83.36	81.98	84.68	84.91	83.64
mdljdp2	input.file	414815687	85.51	88.44	91.56	89.23	93.00	93.81	94.5	94.04
mdljsp2	input.file	753307193	95.33	96.97	97.67	97.58	97.35	98.52	98.6	98.58
nasa	NASA7.CHK	1768897327	56.61	59.75	58.28	62.32	64.7	69.66	69.61	68.89
ora	params	1343836643	97.15	100.00	100.00	100.00	97.15	100.00	100.00	100.00
spice2g6	greycode.in	5265522592	69.75	72.99	74.26	72.93	78.23	81.21	83.29	80.08
su2cor	su2cor.in	1069475885	48.61	49.71	64.53	84.42	62.97	65.79	65.5	85.59
swm256	swm256.in	2851234080	75.72	68.66	66.54	92.87	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		758316846	89.08	91.58	91.48	93.82	94.76	95.84	95.81	95.71
FP Ave			80.22	81.56	83.62	88.55	86.53	89.52	89.93	90.85

generate only 5% of the data references. 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. This is also shown in 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 Table 2 makes 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 through an array with a long 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 the ones with the lowest cache hit rate. These 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.

Table 2: Cumulative Load Instruction Reference Counts

Bench-mark	Load Inst	Percent of Total Data References								Percent of Total Data Cache Misses							
		75%		90%		95%		99%		75%		90%		95%		99%	
		# of Insts	% of Total	# of Insts	% of Total	# 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	3058	34	1.11	49	1.60	54	1.77	60	1.96	6	0.20	9	0.29	15	0.49	29	0.95
eqntott	4656	8	0.17	59	1.27	96	2.06	157	3.37	6	0.13	19	0.41	33	0.71	96	2.06
espresso.cps	16647	137	0.82	299	1.80	588	3.53	1388	8.34	72	0.43	237	1.42	423	2.54	998	5.99
espresso.tail	16647	150	0.90	366	2.20	599	3.60	1265	7.60	84	0.50	235	1.41	385	2.31	876	5.26
espresso.ti	16647	159	0.96	419	2.52	690	4.14	1408	8.46	98	0.59	302	1.81	520	3.12	1105	6.64
gcc.insn	51555	1322	2.56	2874	5.57	4232	8.21	6931	13.44	735	1.43	1562	3.03	2205	4.28	3862	7.49
gcc.integrate	51555	1750	3.39	3691	7.16	5382	10.44	9003	17.46	880	1.71	2017	3.91	3026	5.87	5814	11.28
gcc.stmt	51555	1765	3.42	3682	7.14	5409	10.49	9215	17.87	972	1.89	2176	4.22	3301	6.40	5890	11.42
gcc.tree	51555	1929	3.74	4042	7.84	5993	11.62	10817	20.98	1033	2.00	2439	4.73	3553	6.89	6114	11.86
li	8083	103	1.27	165	2.04	224	2.77	345	4.27	50	0.62	108	1.34	152	1.88	236	2.92
sc.loada1	15968	101	0.63	252	1.58	421	2.64	1054	6.60	14	0.09	71	0.44	141	0.88	396	2.48
sc.loada2	15968	133	0.83	360	2.25	552	3.46	1207	7.56	47	0.29	144	0.90	252	1.58	585	3.66
sc.loada3	15968	119	0.75	336	2.1	519	3.25	1206	7.55	82	0.51	227	1.42	328	2.05	552	3.46
doduc	21313	1485	6.97	2167	10.17	2404	11.28	3069	14.4	301	1.41	563	2.64	787	3.69	1069	5.02
ear	6079	15	0.25	27	0.44	39	0.64	153	2.52	14	0.23	39	0.64	49	0.81	57	0.94
fpppp	19012	2106	11.08	2951	15.52	3233	17.00	3586	18.86	147	0.77	273	1.44	357	1.88	499	2.62
hydro2d	17595	253	1.44	360	2.05	448	2.55	635	3.61	121	0.69	218	1.24	285	1.62	438	2.49
mdljdp2	17493	13	0.07	61	0.35	98	0.56	165	0.94	19	0.11	54	0.31	66	0.38	85	0.49
mdljsp2	17560	24	0.14	82	0.47	121	0.69	215	1.22	28	0.16	68	0.39	84	0.48	113	0.64
nasa	17634	192	1.09	330	1.87	432	2.45	636	3.61	99	0.56	178	1.01	234	1.33	383	2.17
ora	14526	31	0.21	41	0.28	47	0.32	84	0.58	3	0.02	4	0.03	5	0.03	6	0.04
spice2g6	35185	100	0.28	356	1.01	530	1.51	716	2.03	11	0.03	60	0.17	105	0.3	234	0.67
su2cor	20636	189	0.92	394	1.91	620	3.00	1301	6.30	137	0.66	362	1.75	547	2.65	1227	5.95
swm256	15141	28	0.18	50	0.33	57	0.38	63	0.42	27	0.18	37	0.24	40	0.26	50	0.33
tomcatv	13422	63	0.47	92	0.69	101	0.75	109	0.81	29	0.22	69	0.51	83	0.62	94	0.70
wave5	23087	276	1.20	471	2.04	608	2.63	1076	4.66	78	0.34	187	0.81	296	1.28	514	2.23

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. Hit Rate and Memory Bandwidth Utilization

Table 3 shows the change in cache hit rate and required memory bandwidth after the poorest performing instructions were marked C/NA. Column one contains the name of the benchmark program and the second shows the range of instructions that were made C/NA (since the count of instructions varied depending on the cache configuration). Columns 3-7 show the change in hit rates (compared to the entries in table 1) for caches that are direct mapped and 2-way set associative. As can be seen in the table, there was a uniform slight decrease in the hit rate across all configurations.

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

Table 3: Change in Data Cache Hit Rate and Memory Bandwidth After Removal of Target Instructions (Static)

Bench-mark	# of C/NA Instructions	% Change in Cache Hit Rate				% Change in Memory Bandwidth Requirements			
		8K-byte Cache		16K-byte Cache		8K-byte Cache		16K-byte Cache	
		Direct	2-way	Direct	2-way	Direct	2-way	Direct	2-way
compress	5-10	-2.34	-1.76	-3.08	-2.38	-56.18	-62.62	-55.36	-61.62
eqntott	25-54	-1.47	-1.58	-1.54	-1.37	-17.51	-14.09	-17.58	-13.84
espresso.cps	94-216	-2.44	-1.92	-0.72	-0.54	-18.36	-20.93	-17.50	-24.13
espresso.tail	93-265	-2.81	-1.35	-0.25	-0.24	-25.08	-21.36	-18.28	-6.69
espresso.ti	108-265	-2.72	-2.47	-1.31	-1.05	-22.50	-27.83	-17.85	-22.01
gcc.insn	259-725	-3.17	-2.27	-2.07	-1.54	-24.29	-19.95	-16.68	-7.59
gcc.integrate	131-406	-2.95	-2.30	-1.65	-1.27	-20.12	-14.51	-12.41	-2.47
gcc.stmt	182-614	-2.92	-2.04	-1.36	-1.00	-20.76	-15.89	-12.89	-5.96
gcc.tree	63-342	-2.38	-1.89	-0.83	-0.49	-21.74	-19.32	-16.93	-8.22
li	59-135	-3.57	-3.52	-2.24	-2.87	-17.48	3.98	-19.26	14.23
sc.loada1	181-295	-16.56	-14.78	-13.71	-13.32	-53.23	-49.57	-51.18	-52.62
sc.loada2	233-381	-9.98	-10.07	-9.40	-9.05	-47.16	-40.44	-42.57	-42.40
sc.loada3	153-274	-3.78	-2.48	-2.11	-0.61	-51.88	-57.91	-52.21	-66.11
Int Ave		-4.39	-3.73	-3.10	-2.75	-30.48	-27.73	-26.98	-23.03
doduc	358-707	-2.69	-3.93	-2.25	-2.11	-41.02	-27.52	-36.88	-14.84
ear	7-49	-0.49	-0.35	-0.25	0.03	-4.20	-2.51	-18.34	-19.14
fpppp	51-263	-1.87	-0.39	-1.71	-0.11	-26.27	-28.48	-13.03	-25.56
hydro2d	390-491	-21.76	-19.53	-20.99	-17.94	10.67	14.49	10.54	14.80
mdljdp2	117-143	-6.80	-6.16	-5.57	-5.29	-2.55	-8.01	-4.95	-6.07
mdljsp2	47-74	-0.44	-0.33	-0.14	-0.12	1.95	2.22	0.93	1.59
nasa	448-635	-15.27	-15.85	-17.43	-16.05	-43.49	-41.49	-36.53	-33.50
ora	0-3	-0.81	0.00	-0.81	0.00	-67.86	-0.38	-67.86	0.15
spice2g6	320-446	-23.95	-24.63	-26.87	-27.24	-49.39	-45.53	-34.30	-27.79
su2cor	889-1650	-3.40	-3.77	-12.65	-13.55	-65.06	-64.71	-49.90	-45.63
swm256	61-129	-4.19	3.45	-0.21	-0.28	-43.84	-68.49	-2.65	-3.24
tomcatv	46-78	-11.74	-9.35	-8.99	-11.48	-42.05	-45.27	-34.00	5.59
wave5	127-240	-2.55	-2.79	-1.57	-1.03	-24.25	-13.38	3.18	2.91
FP Ave		-7.38	-6.43	-7.65	-7.32	-30.57	-25.31	-21.83	-11.59

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 allows us to calculate the percentage change in the bus bandwidth needed by the Static approach. The results of these calculations are shown in the last 4 columns of table 4. So, for example, after the C/NA transformations the compress program run on a 16K-byte 2-way set associative data cache requires 61.62% less bandwidth than that required by the same program run on the same hardware without the C/NA transformations.

The table shows a significant overall decrease in the required memory bandwidth. In particular, it shows that the static scheme used in conjunction with an 8K direct mapped cache results in an average decrease in bus activity of approximately 30% for both the integer and the floating point programs.

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 4. 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

Table 4: Analysis of Average Memory Reference Activity (Static)

Instruction Classification	Number of Instructions	% of Refs	Memory Refs Pre-Transformation	Memory Refs Post-Transformation		Change
				C/NA	Non-C/NA	
CNA	351	19%	117,282,922	149,700,459	-	27.64%
Increased Miss Rate	443	12%	10,070,793	-	45,447,421	351.28%
No Change	3,882	35%	14,297,639	-	14,297,639	0.00%
Decreased Miss Rate	753	33%	43,831,661	-	31,260,346	-28.68%
Not Referenced	16,054	-	-	-	-	-
Total	21,483	100%	185,483,016	149,700,459	91,005,406	29.77%

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.

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 4 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. The fifth and sixth columns show the number of memory references after the C/NA transformations were performed and which instructions were responsible for the references.

In this table we see that the total number of memory references has increased by over 29%. This is due in large part to the 351% increase in the number of cache misses experience by 443 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.

4.2. Improved Static Method

In order to improve the performance of the simple static technique, the number of instructions marked C/NA had to be reduced. This was accomplished 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, this modification to the static approach was implemented 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. Hit Rate and Memory Bandwidth Utilization

As shown in Table 5, the Improved Static approach provides hit rates very close to those presented in Table 1. Cache performance was only slightly worse for the both the integer and floating point benchmarks (on average).

Table 5 also shows how the improved Static scheme affects the bus bandwidth. The table shows that the Improved Static scheme consistently reduced the memory bandwidth requirements 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 5 shows that the improved static scheme used in conjunction with an 8K cache results in an average decrease in bus activity by as approximately as 30%, and by more than 50% for 5 of the programs.

4.2.2. Memory Activity

An examination of the memory activity shown in Table 6 reveals several interesting observations. For example, the number of instructions in the C/NA class dropped from 351 to 187, indicating that there are a lot of instructions with high miss rates that are actually performing useful work (prefetching). As one might expect, the increase in memory activity due to the C/NA instructions dropped as well. However, the most dramatic change is in the number of instructions that have their miss rate increase - this drops from 443 to 307, resulting in a reduction in memory activity from 351% to 62%.

Table 5: Change in Data Cache Hit Rate and Memory Bandwidth After Removal of Instructions (Improved Static)

Bench-mark	# of C/NA Instructions	% Change in Cache Hit Rate				% Change in Memory Bandwidth Requirements			
		8K-byte Cache		16K-byte Cache		8K-byte Cache		16K-byte Cache	
		Direct	2-way	Direct	2-way	Direct	2-way	Direct	2-way
compress	5-7	-0.28	-1.76	-1.02	-2.38	-58.71	-62.62	-58.12	-61.62
eqntott	11-30	-1.19	-1.45	-1.28	-1.28	-18.43	-14.55	-18.60	-14.20
espresso.cps	50-120	-0.36	-0.46	-0.17	-0.10	-22.12	-24.03	-17.50	-24.78
espresso.tail	38-129	0.18	-0.04	0.18	-0.09	-14.09	-7.68	-13.17	-6.60
espresso.ti	59-145	-0.35	-0.36	-0.24	0.01	-25.79	-31.43	-23.24	-28.83
gcc.insn	89-341	-0.97	-0.55	-0.37	-0.22	-26.25	-23.94	-18.99	-13.32
gcc.integrate	30-185	-1.06	-0.75	-0.44	-0.21	-22.30	-17.81	-15.18	-6.89
gcc.stmt	62-290	-1.08	-0.59	-0.35	-0.20	-23.10	-19.20	-15.24	-9.50
gcc.tree	30-176	-1.13	-1.08	-0.42	-0.19	-23.53	-21.27	-17.88	-9.74
li	21-49	0.24	0.07	0.63	0.00	-22.28	-5.86	-26.74	-2.69
sc.loada1	77-123	-5.40	-5.87	-4.81	-4.31	-55.96	-53.69	-54.97	-57.45
sc.loada2	91-160	-3.86	-3.56	-3.09	-2.64	-48.87	-43.87	-44.61	-45.64
sc.loada3	48-103	-1.23	-0.18	-0.69	0.61	-52.32	-58.66	-53.67	-68.19
Int Ave		-1.27	-1.28	-0.93	-0.85	-31.83	-29.59	-29.07	-26.88
doduc	83-254	0.83	-0.04	0.60	-0.22	-43.10	-26.68	-39.75	-15.68
ear	2-17	0.13	0.10	0.19	0.08	-8.94	-7.31	-26.67	-21.74
fpppp	28-110	0.00	0.07	-0.23	0.03	-31.10	-29.77	-19.74	-27.41
hydro2d	70-115	-0.84	-0.02	-0.78	-0.02	-2.03	-0.57	-1.92	-0.22
mdljdp2	33-49	-0.50	-0.29	-0.18	-0.08	-2.87	-9.57	-3.42	-5.23
mdljsp2	10-18	0.00	0.01	0.00	0.00	-0.44	-0.57	-0.38	-0.17
nasa	237-379	-3.86	-3.18	-4.34	-4.59	-49.17	-47.01	-42.61	-39.44
ora	0-1	-0.20	0.00	-0.20	0.00	-41.07	-0.38	-41.07	0.15
spice2g6	161-246	-22.34	-23.44	-25.52	-25.99	-49.81	-46.30	-35.16	-29.04
su2cor	323-1522	0.73	1.24	-5.00	-5.34	-67.06	-67.16	-54.26	-50.52
swm256	39-99	-0.01	4.44	0.00	0.09	-45.73	-69.26	-3.30	-4.42
tomcatv	3-31	4.96	4.33	3.29	-0.25	-43.71	-40.95	-40.31	-3.85
wave5	38-88	0.06	-0.36	-0.12	0.02	-29.23	-18.63	-3.27	-3.34
FP Ave		-1.62	-1.32	-2.48	-2.79	-31.87	-28.01	-23.99	-15.45

Table 6: Analysis of Average Memory Reference Activity (Improved Static)

Instruction Classification	Number of Instructions	% of Refs	Memory Refs Pre-Transformation	Memory Refs Post-Transformation		Change
				C/NA	Non-C/NA	
CNA	187	11%	71,394,580	87,222,835	-	22.17%
Increased Miss Rate	307	7%	12,050,735	-	19,620,224	62.81%
No Change	4,152	47%	37,264,111	-	37,264,111	0.00%
Decreased Miss Rate	782	35%	73,301,176	-	51,759,319	-29.39%
Not Referenced	16,054	-	-	-	-	-
Total	21,483	100%	194,010,602	87,222,835	108,643,654	0.96%

The most significant number in Table 6 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 while simultaneously decreasing the amount of traffic to memory.

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 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 instruction. A table, similar to a branch prediction table, can be maintained which tracks whether the data referenced by a load 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 instruction. A miss prediction counter is initially set to zero and it is incremented each time a load causes a cache miss. If the load 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, can change during 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.

Experiments were performed using 2-bit counter miss prediction schemes. Initially the size of the miss prediction table was unlimited in order to evaluate the ability of the 2-bit scheme to track a hit/miss history. In later runs the size of the miss prediction table was fixed.

Table 7 summarizes the average memory reference activity when using 2-bit counters for miss prediction on the SPEC benchmarks. As in Tables 4 and 6 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 7, 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 4 and 6 with the first three lines of Table 7. 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 92.15%.

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 92% 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

Table 7: Analysis of Average Memory Reference Activity - Dynamic 2-bit Counter Scheme

Instruction Classification	Number of Instructions	% of Refs	Memory Refs Pre-Transformation	Memory Refs Post-Transformation		Change
				C/NA	Non-C/NA	
<5 CNA	762	33%	66,335,966	44,468,025	35,402,802	20.40%
5-95 CNA	359	6%	21,741,685	74,689,601	4,450,122	264.00%
> 95 CNA	509	15%	61,618,437	174,892,281	219,579	184.19%
Increased Miss Rate	292	5%	1,264,062	-	1,833,090	45.02%
No Change	2,999	21%	8,624,533	-	8,624,533	0.00%
Decreased Miss Rate	507	20%	24,824,073	-	9,764,111	-60.67%
Not Referenced	16,054	-	-	-	-	-
Total	21,483	100%	184,408,757	294,049,907	60,294,238	92.15%

Table 9: Change in Cache Hit Rate and Memory Bandwidth After Removal of Instructions (Improved Dynamic)

Bench-mark	# of C/NA Instructions	% Change in Cache Hit Rate				% Change in Memory Bandwidth Requirements			
		8K-byte Cache		16K-byte Cache		8K-byte Cache		16K-byte Cache	
		Direct	2-way	Direct	2-way	Direct	2-way	Direct	2-way
compress	7-7	-0.36	-0.57	-0.59	-0.75	-50.31	-63.69	-45.45	-58.86
eqntott	52-73	-0.05	-0.03	-0.11	-0.11	-20.23	-24.79	-20.60	-24.16
espresso.cps	120-247	-0.25	-0.30	-0.13	0.13	-7.54	-8.54	-6.87	-8.72
espresso.tail	75-204	-0.01	-0.16	-0.03	-0.57	-1.04	-0.80	-0.23	-0.43
espresso.ti	180-315	-0.02	0.31	0.02	0.29	-14.43	-18.17	-13.59	-17.96
gcc.insn	291-551	-0.29	-0.40	-0.27	-0.64	-8.30	-8.50	-6.32	-6.75
gcc.integrate	310-595	-0.09	-0.44	-0.10	-0.58	-8.78	-8.48	-6.66	-5.91
gcc.stmt	338-676	-0.02	-0.35	-0.06	-0.46	-7.72	-7.19	-5.95	-5.40
gcc.tree	200-481	0.06	-0.23	-0.01	-0.49	-6.95	-6.99	-5.82	-5.13
li	47-91	-0.05	0.22	0.03	0.25	-7.36	-1.04	-10.14	-0.38
sc.loada1	66-137	-7.22	-5.46	-6.30	-5.32	-54.99	-52.64	-54.17	-52.94
sc.loada2	85-155	-6.06	-3.85	-5.85	-4.09	-51.02	-43.57	-51.46	-43.09
sc.loada3	38-91	-1.71	0.13	-0.93	0.24	-46.88	-58.07	-50.29	-62.59
Int Ave		-1.24	-0.86	-1.10	-0.93	-21.97	-23.27	-21.35	-22.49
doduc	8-24	0.32	1.09	0.31	-0.94	-5.25	-1.32	-6.45	-1.24
ear	1-8	0.02	0.05	0.09	-0.23	-0.50	2.12	-7.39	-2.35
fpppp	84-97	-0.08	-0.78	-0.19	-0.98	-1.15	-4.37	3.10	-3.82
hydro2d	82-135	0.13	-0.34	0.17	-1.05	-3.21	-0.08	-3.08	-0.05
mdljdp2	17-21	-0.04	0.77	-0.01	0.22	-0.82	-0.92	-0.58	-0.51
mdljsp2	12-14	0.00	0.61	0.00	0.06	-0.15	-0.13	-0.12	0.01
nasa	317-354	-0.15	-0.02	-2.88	-4.73	-51.68	-50.01	-47.34	-44.59
ora	3-3	0.00	0.00	0.00	0.00	0.00	-0.09	0.00	0.18
spice2g6	236-301	-4.09	-3.56	-4.17	-4.56	-27.39	-25.86	-17.34	-16.68
su2cor	318-1267	2.62	26.09	1.08	15.73	-24.86	0.72	-16.75	0.33
swm256	46-50	0.00	24.21	0.00	0.59	-0.46	-1.53	-1.49	-1.60
tomcatv	1-39	-4.86	25.83	-1.18	1.82	-13.69	0.00	-3.21	0.00
wave5	96-125	-1.46	1.21	-1.31	-0.97	-5.07	-7.49	-0.97	-0.10
FP Ave		-0.58	5.78	-0.62	0.38	-10.33	-6.84	-7.82	-5.42

6. Conclusions and Future Work

In this work, we have investigated the potential for improving average data access time 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.

Based on this observation, we analyzed the impact on cache and memory system performance if certain data items were not cached. In the first part of our simulation studies, we determined whether an instruction's data item should be cached by performing a static analysis of program behavior. The results of these studies indicate that the amount of memory activity, the required memory bandwidth, could be substantially reduced by not caching all data items.

Since this static analysis requires executing the entire program and marking which instructions should have their data cached, we then looked at dynamic schemes that could dynamically detect which data items should be cached. The dynamic schemes we investigated are based on 2-bit branch prediction schemes. Instead of a branch prediction table, we have a miss prediction table that holds a 2-bit counter associated with load and store instructions. We investigated two 2-bit miss prediction strategies. Both of these strategies offered a reduction in a program's memory bandwidth requirements. However, neither dynamic scheme performed as well as the our improved static scheme.

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 21st Annual International 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.

[EKPP93] P. G. Emma, J. W. Knight, J. H. Pomerene, T. R. Puzak and R. N. Rechtschaffen, ''Cache Miss Facility with Stored Sequences for Data Fetching'', *U.S. Patent 5,233,702*(Issued: August 3, 1993).

[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.

[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 customized 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.