DATA RELOCATION AND PREFETCHING FOR PROGRAMS WITH LARGE DATA SETS

BY

YOJI YAMADA

B.Engr., Tokyo Institute of Technology, 1980
M.Engr., Tokyo Institute of Technology, 1982

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1995

Urbana, Illinois

DATA RELOCATION AND PREFETCHING FOR PROGRAMS WITH LARGE DATA SETS

YOJI YAMADA, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1995
Wen-mei W. Hwu, Advisor

Numerical applications frequently contain nested loop structures that process large arrays of data. The execution of these loop structures often produces memory preference patterns that poorly utilize data caches. Limited associativity and cache capacity result in cache conflict misses. Also, non-unit stride access patterns can cause low utilization of cache lines. Data copying has been proposed and investigated in order to reduce the cache conflict misses [1][2], but this technique has a high execution overhead since it does the copy operations entirely in software.

I propose a combined hardware and software technique called data relocation and prefetching which eliminates much of the overhead of data copying through the use of special hardware. Furthermore, by relocating the data while performing software prefetching, the overhead of copying the data can be reduced further. Experimental results for data relocation and prefetching show a large improvement in cache performance.

ACKNOWLEDGMENTS

First and foremost, I would like to acknowledge my advisor, Professor Wen-mei W. Hwu, for his kind guidance, his intellectual support, and his patience. I am very glad to have worked with him, and I will be pround of that forever.

Next, I would like to acknowledge the support of my Ph.D. committee members for their useful suggestions with regard to my thesis research.

This research would not have been possible without the support of the IMPACT research group. I would like to thank John Gyllenhaal and Grant Haab for the simulation support and useful discussion regarding my research. Many thanks to Sadun Anik, Roger Bringmann, William Chen, Dave Gallagher, Rick Hank, Sabrina Hwu, Dan Lavery, Scott Mahlke, Krishna Subramanian, and Nancy Warter.

I would like to acknowledge the financial support of Mazda Motor Corporation for my graduate study.

Lastly, I would like to acknowledge the support of my family. I would like to acknowledge my parents for their love. I would like to thank my son, Shinji, and daughter, Emi, for their smiles, which really relaxed my fatigue due to this thesis work. I would like to thank my wife, Hideko, for her cooking, love and patience.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Overview

Numerical applications frequently contain nested loop structures that process large arrays. The execution of these loop structures has been shown to produce memory reference patterns that poorly utilize data caches [3][4]. At least three problems have been identified as the cause of poor cache utilization. The first problem involves an insufficient capacity of the cache: The data accessed by each loop may exceed the cache size, resulting in cache misses. Limited associativity of the cache leads to a second problem: accesses to different arrays, or even to different elements of a single array, may conflict. The third problem involves non-unit stride access patterns that can cause low utilization of cache lines and wasted bus and memory cycles [5].

Potentially, one could use a larger cache size and higher cache associativity to eliminate cache capacity misses and cache conflict misses. This brute force approach, however, does not scale well with the rapidly increasing amount of data used in sophisticated numerical applications. Moreover, it would result in significant hardware cost and increased cache access latency, both of which could be avoided via the more cost-effective approach proposed in this research.

The use of loop blocking transformations could reduce the working-set size of data accessed in loop nests [4] [1] [6]. By reordering the execution of iterations, loop blocking transformations reduce the amount of data referenced between two references to the same datum. Once the data accessed between two references to the same datum is reduced to an amount smaller than the cache size, capacity misses are eliminated. In practice, however, loop blocking transformations may not reduce cache misses because of cache mapping conflicts. Additionally, blocking alone does not reduce the working-set size of data accessed in single loops since the data accesses are not reordered.

Data prefetching has also been proposed to reduce cache misses by fetching data into the cache before it is referenced [7] [8]. When used in conjunction with small cache-block sizes, one can potentially eliminate the problem of low utilization of cache blocks and wasted bus cycles [5]. However, data prefetching may increase the size of the working set, introducing capacity misses. Also, prefetched data may conflict with the current working set in the cache, introducing more conflict misses [9] [10]. In order for data prefetching to improve performance in a reliable manner, one must ensure that both current and future working sets can fit into the cache. The proposed approach achieves this goal by compressing the current and future working sets into a localized region in the virtual address space such that no cache mapping conflicts exist among locations in the region.

This thesis research introduces an approach to solve all three cache performance problems for array-based applications, rather than a solving these problems singly. The first phase in this technique consists of strip-mining inner loops to reduce the number of array accesses in the working set. In the second phase, the insertion of special hardware

instructions compress the working set into a localized region in the virtual address space and prefetch the compressed working set into the cache. The compiler also modifies the working set accesses so that all references will be made to the compressed data in the cache. Since array data are compressed in the localized region, most conflict misses are eliminated. Also, if the original data access pattern is of non-unit stride, unused data are not brought into the cache during compression and prefetch, resulting in improved cache-line utilization. After the computation is completed, additional instructions decompress the modified data and relocate it back to the original program arrays.

In order to minimize the overhead of compressing and decompressing data, compression is performed as the data is prefetched from the memory into the cache. Also, through the use of compiler transformations, compression and prefetching of the next working set is overlapped with the computation for the current working set in order to hide the latency of the relocation.

Using a prototype compiler, an emulation tool, and a simulation tool, I show that this extension to the cache and processor architecture along with the requisite compiler support greatly improves the data cache performance for array-based applications.

1.2 Cache Memory

A cache memory is a small, fast memory located to close to the CPU that holds the most recently accessed code or data. Accessing a cache takes much less time than accessing main memory due to the property that *smaller is faster*, which means that the

cache is smaller and can be made faster than the large main memory. The cache can hold data that will be accessed in the near future due to the property of *locality of reference*.

There are several objectives in the design of a cache. The most important objective is generally the high hit ratio, which is a measure of the likelihood that the cache contains the requested data in it. Another important objective is to achieve fast access time, which is determined by the hardware complexity. Also, it should be as small as possible while maintaining the desired hit ratio. The degree to which these objectives are achieved depends on not only the cache structure but also the characteristics of the targeted program. For example, the cache performance of an array-oriented program is likely different from that of a scalar-oriented program. Supercomputer applications, especially, have many loops that access large arrays. These array accesses are classified into two categories: sequential accesses and no-sequential accesses. Non-sequential accesses occur:

- when elements of an array are accessed by column,

  ```
  Example: A[0][0], A[1][0], A[2][0], ...
  ```

- when the stride of the access is larger than one,

  ```
  Example: A[0], A[2], A[4], A[6], ...
  ```

- or when different arrays are accessed in turn.

  ```
  Example: A[0], B[0], C[0], A[1], B[1], C[1], ...
  ```

These characteristics of numerical programs introduce some inefficiency cache as shown in Section 1.2.1 - 1.2.3.

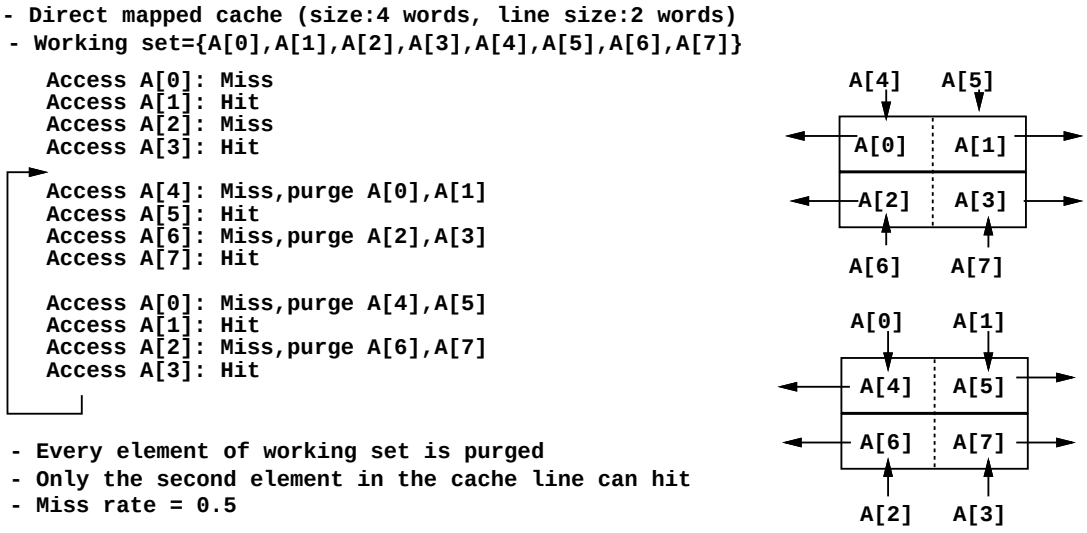

Figure 1.1 Capacity Misses Due to Large Working Set

1.2.1 Capacity Misses

If the cache cannot hold all data needed during the execution of a program, capacity misses will occur due to data being purged and later referenced. In general, numerical programs use large working sets, resulting in frequent capacity misses. This concept is illustrated in Figure 1.1. In this case, the capacity misses comes from an actual lack of cache capacity.

Non-sequential access, another characteristic of numerical programs, may also introduce capacity misses, as shown in Figure 1.2. These capacity misses do not result wholly from a lack of cache capacity, but rather the underutilization of the cache.

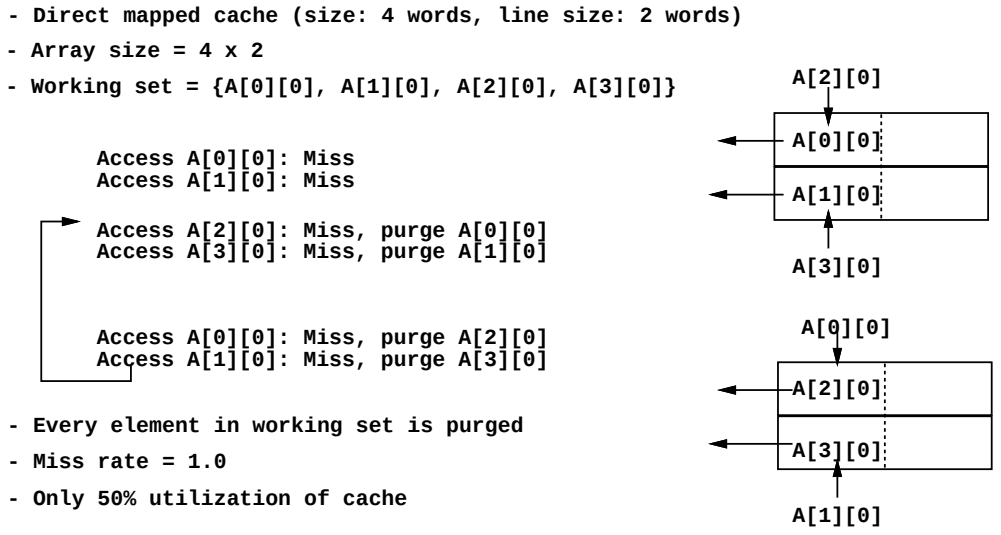

Figure 1.2 Capacity Misses Due to Non-Sequential Accesses

1.2.2 Conflict Misses

If too many cache lines are mapped to a set in the cache, conflict misses will occur due to lines being purged and later referenced. Non-unit stride accesses in an array introduce conflict misses as shown in Figure 1.3. Unlike capacity misses, conflict misses will occur even though unused cache lines exist in the cache. Conflict misses also occur when different arrays are accessed in turn and the elements map to the same set (Figure 1.4).

1.2.3 Line Fetch Overhead

Another problem due to non-sequential accesses is line fetch overhead. In general, the cache line size is larger than one word. When an element of an array is brought into the cache, consecutive elements are also fetched into the same cache line. If the array accesses have non-unit stride, the consecutive elements that were fetched in cache are

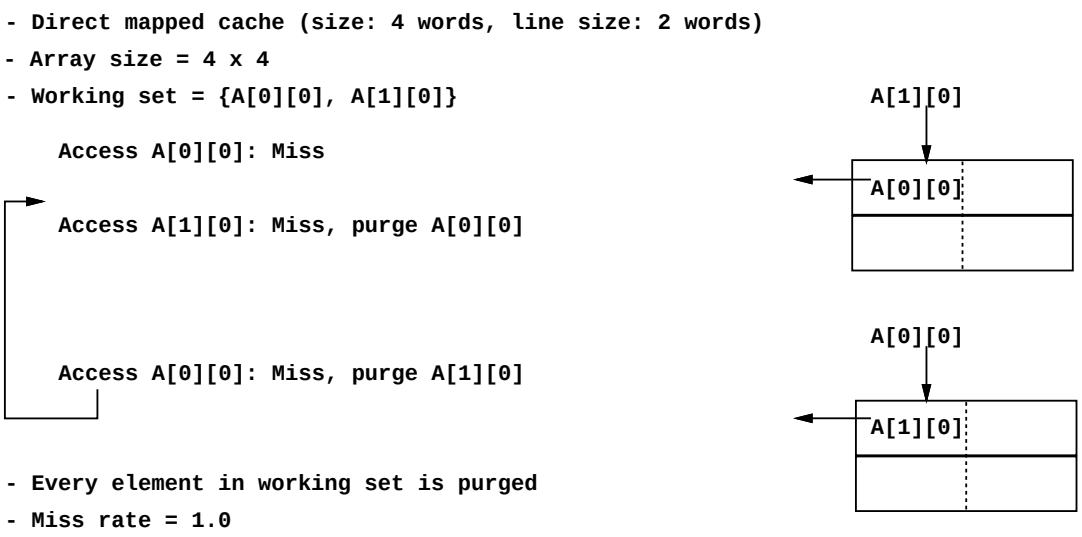

Figure 1.3 Intra-array Conflict Misses

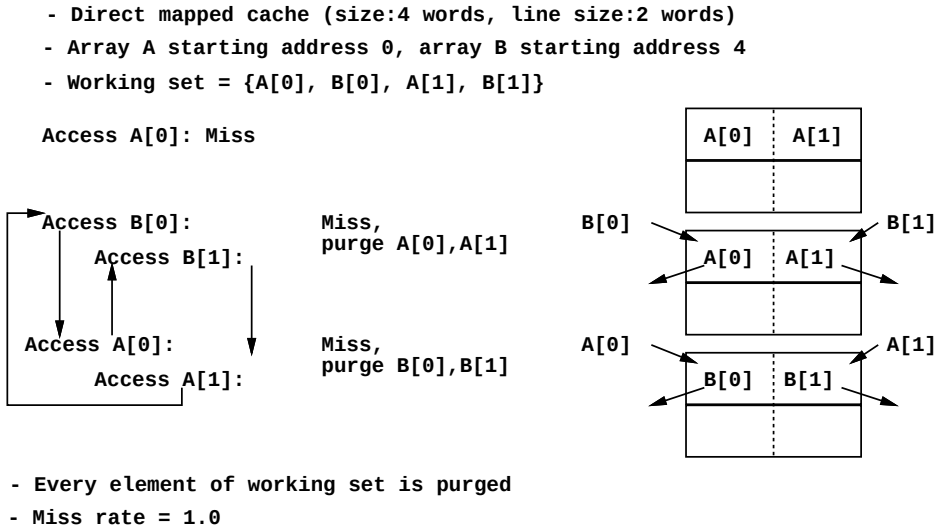

Figure 1.4 Inter-array Conflict Misses

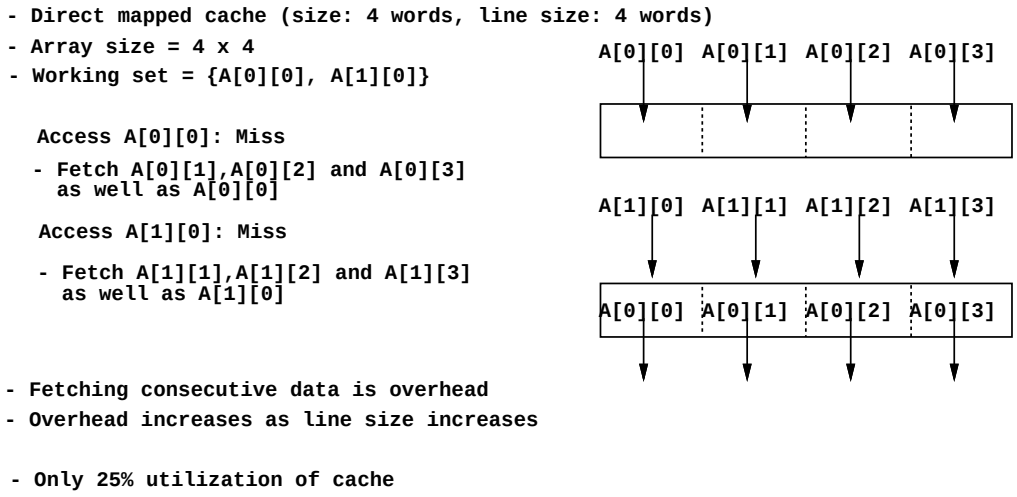

Figure 1.5 Cache Line Fetch Overhead Due to Non-Sequential Accesses

unlikely to be used in near future. Thus, fetching the consecutive elements is considered as overhead, not useful work (Figure 1.5).

1.3 Previous Work

A technique called data copying has been proposed and investigated in order to reduce the cache conflict misses [1][2]. Data copying, however is beneficial only if the performance improvement outweighes the overhead of copying by reusing the data many times. The overhead of copying data from array to array is significant in general. Our proposed method, data relocation and prefetching, has the same benefits as data copying while reducing the overhead of copying. Furthermore, unlike data copying, data relocation and prefetching relocates data into the cache during prefetching, which minimizes the overhead of the relocation.

Gather and *Scatter* operations for vector machines are used in the Cray-1 [11] in order to handle non-unit stride data effectively. In the Cray-1, the array elements are "gathered" from memory into the vector registers before performing vector operations, and "scattered" back to memory after the vector operations are complete. However, the hardware necessary to support data relocation and prefetching would be much easier to add to an existing processor than the hardware to support vectorization.

CHAPTER 2

HIGH LEVEL PROFILING AND SIMULATION

In general, profiling information is very useful for code transformation and optimization [12][13]. This chapter describes a high-level profiling tool for loop iteration analysis and array reference analysis. The statistics of the profiled results for some benchmarks are also described. Finally, high-level cache simulation is described with some results to motivate data relocation and prefetching.

2.1 IMPACT Compiler

The high-level profiling and simulation have been implemented in IMPACT compiler [17], which is a research-oriented proto-type C compiler.

2.1.1 Overview

Figure 2.1 is an overview of the IMPACT C compiler. FORTRAN programs are first translated into C by *f2c* before being compiled by IMPACT. The C program is then parsed, semantically analyzed, and translated into Pcode. Dependence analysis and high-level transformations are done in Pcode. After all transformations and optimizations are performed at the high level, Pcode is translated into Lcode, where low-level optimizations are performed. The code generator converts the Lcode into the assembly language of the

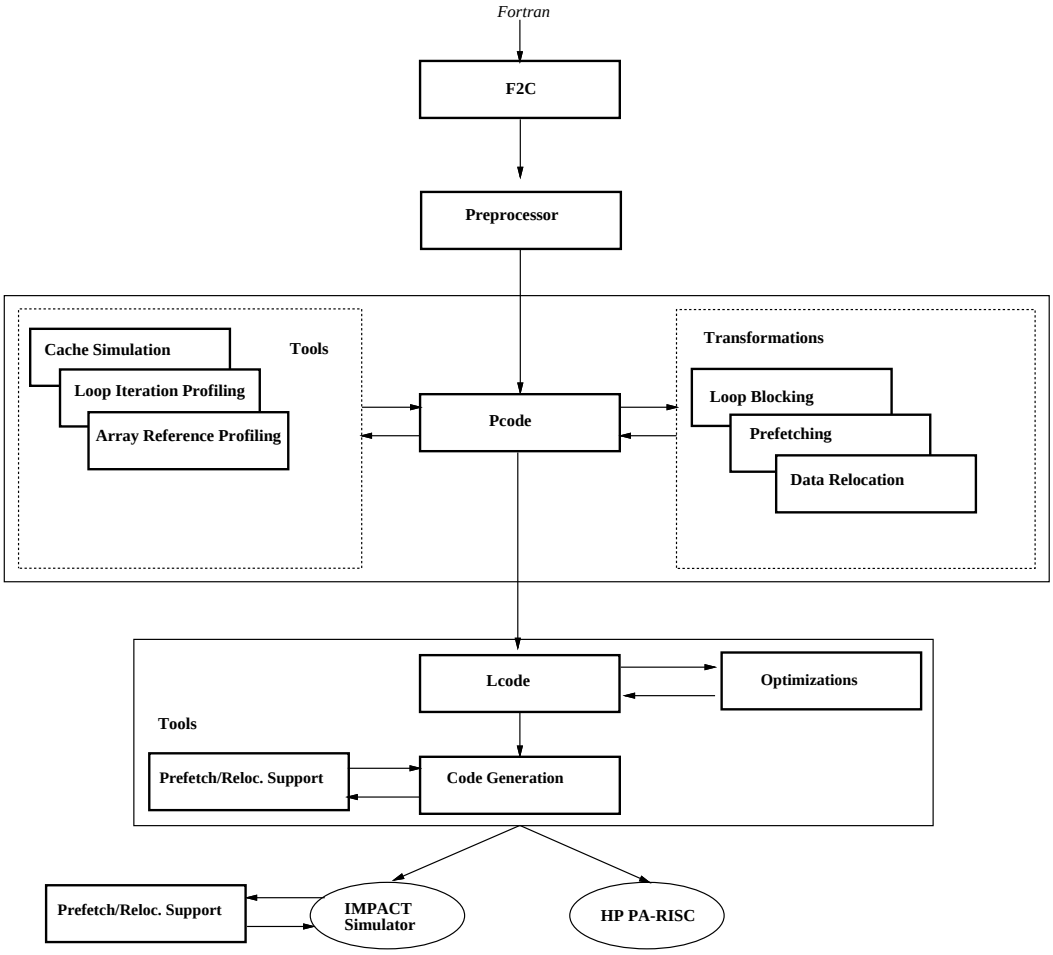

Figure 2.1 An Overview of the IMPACT C Compiler

target machine using a machine specification. The IMPACT C compiler generates code for several existing processor architectures: Sun SPARC, HP PA-RISC, AMD29k and Intel X86.

2.1.2 Pcode Overview

Pcode is the intermediate format which represents C programs. High-level transformations and optimizations are performed at the Pcode level. The restructuring process at the Pcode level consists of the following steps:

(1) Read the source code into the Pcode intermediate representation. The *for, while* and *do* loops in the source are read into the *Ser_loop* statement structure.

(2) Dependence analysis and transformations can be performed on *for* loops that meet the following criteria:

- Loops with simple header expressions,
- Loops with scalar iteration variables,
- Loops nested up to six levels deep,
- Loops that have non-negative integer constant increment expressions,
- Loops that are natural loops (have only one entry point).

These loops are identified and their representation is changed from a *Ser_loop* to a *Par_loop* data structure.

(3) Loop summary information is calculated for each loop nest. Since dependence analysis is performed only on *Par_loops*, the loop summary information is used to check if all the loops for which dependence information is needed to perform a given transformation are *Par_loops*. Transforming a loop nest without this test could be semantically incorrect since the dependence information for such a loop nest is incomplete.

(4) Dependence analysis for scalar and array references is performed on all the loop nests in each function and the information is maintained in an auxiliary data structure. Currently, the dependences across loop nests are not calculated since this

information is not needed by the transformations. The dependence analyzer uses the Omega test [14][15] to compute dependence direction and distance vectors for array references, which is an algorithm based on an extension of Fourier-Motzkin variable elimination to integer programming. An extended data flow algorithm is used to compute loop-carried dependence information for scalars.

(5) Each transformation uses the dependence information to determine validity. The transformations are performed in the order specified by the user of the IMPACT compiler.

Table 2.1 shows some static loop information. The benchmarks are selected from the *PERFECT* [16], *SPEC'89* and *SPEC'92* benchmark suits. In these benchmarks, most *for* loops are transformed into *Par_loops*. A few *for* loops are not transformed into *Par_loops* since the header expression are complex. In the Table 2.1, the *Par_loops* are classified into two types: *Outer* and *Innermost*. An *Outer Par_loop* is one that has one or more loops nested within it. An *Innermost Par_loop* is one that has no loops nested within it.

2.2 Pcode Profiling

Some profile information such as loop iteration counts is very important in order to make the transformations effective. For example, performance degradation can occur if a loop nest is blocked and the iteration counts is smaller than the transformed blocking size. Also, loop invocation counts can be used to screen the unimportant loops so that the possible code expansion due to the transformation can be minimized. In the IMPACT

Benchmark Name	# of Ser_loops	# of Par_loops					
		Loop Type	Nesting Depth				Total
			1	2	3	4	5	
ADM	1	Outer	75	35	2	2	0	114
		Innermost	79	67	40	0	2	188
SPICE	5	Outer	27	7	0	0	0	34
		Innermost	250	49	11	0	0	310
MGD	2	Outer	12	4	1	0	0	17
		Innermost	13	10	11	1	0	35
TRACK	0	Outer	18	8	2	0	0	28
		Innermost	34	23	7	5	0	69
BDNA	0	Outer	50	6	2	0	0	58
		Innermost	129	74	7	2	0	212
OCEAN	9	Outer	42	11	0	0	0	53
		Innermost	28	39	12	0	0	79
DYFESM	2	Outer	70	23	6	0	0	99
		Innermost	68	65	25	7	0	165
ARC2D	1	Outer	63	20	0	0	0	83
		Innermost	71	59	20	0	0	150
FLO52	3	Outer	48	22	0	0	0	70
		Innermost	45	49	24	0	0	118
TRFD	1	Outer	11	13	16	6	0	46
		Innermost	3	3	6	14	6	32
SPEC77	0	Outer	98	26	4	0	0	128
		Innermost	84	150	41	4	0	279
MATRIX300	0	Outer	2	0	0	0	0	2
		Innermost	13	2	0	0	0	15
NASA7	5	Outer	33	17	6	0	0	56
		Innermost	24	30	13	6	0	73
TOMCATV	0	Outer	6	0	0	0	0	6
		Innermost	6	6	0	0	0	12

Table 2.1 Static Loop Information for PERFECT and SPEC

compiler, profiling tools for Lcode have already been implemented. Since a profiling tool for Pcode was unavailable, one has been implemented for my thesis research.

Profiling is the process of selecting a set of inputs for a program, executing the program with these inputs and recording the run-time behavior of the program. The following information is collected by the Pcode profiler.

(1) The invocation count of each loop nest.

(2) The initial value, final value and increment value of each loop nest

(3) The iteration count of each loop nest.

Pcode profiling is supported by three tools: a probe insertion program, an execution monitor and a program to summarize profile information. There are three steps to generate execution profiles:

(1) The probe insertion program assigns a unique identifier or line number to each loop nest and inserts a probe just before the loop nest in Pcode. Whenever the prove is activated, it increments the invocation counter for the associated loop nest. It also produces the initial value, final value and increment value of the loop iteration variable. The probe insertion program also inserts a probe just before the first line in the loop nest to increment the iteration count every time that the body of the loop nest is executed.

(2) The probed insertion program is reverse-translated into C code. The C code is compiled by a standard C compiler, and linked with the execution monitor program to generate the probed executable program.

(3) The executable program is executed for a set of inputs, and generated profile information is collected.

(4) Finally, the average profile data is generated by the program using the profile information and merged back into the Pcode intermediate representation.

Table 2.2 - 2.4 show some dynamic loop information that is summarized from the profile information. Table 2.2 shows how many times each loop was invoked on average at run time for a input. **ADM** and **SPICE** have many loops that are never invoked at run time by the input used. **SPICE**, **BDNA**, **MATRIX300** and **TOMCATV** have very small invocation counts for *Outer* loops compared with those for *Innermost* loops.

Table 2.3 shows how many iterations each loop had on average at run time for a input. This information is very important for some transformations such as loop blocking since the blocking size should be smaller than the iteration counts. The last column of the table shows the average iteration counts per invocation. **ADM** and **SPICE** have very small iteration counts for both *Outer* and *Innermost* loops. In most benchmarks, the iteration counts for *Outer* loops are equal to or less than that for *Innermost* loops. However, **MGD** and **TRACK** have larger iteration counts for *Outer* loops than for *Innermost* loops.

Bench Name	Loop Type	# of Loops	Number of Loops with Average Invocation Count							Average Invocation Per Loop
			0	10	100	1000	10K	100K	More	
ADM	Outer	114	71	12	2	20	0	6	3	19,579
	Innermost	188	102	20	7	7	2	37	13	58,784
SPICE	Outer	34	20	9	5	0	0	0	0	17
	Innermost	310	240	31	11	22	3	0	3	10,971
MGD	Outer	17	5	3	1	6	1	1	0	3,256
	Innermost	35	5	8	1	5	0	3	13	1,602,565
TRACK	Outer	28	5	5	6	4	2	6	0	7,670
	Innermost	69	12	5	13	8	7	23	1	17,391
BDNA	Outer	58	14	43	1	0	0	0	0	2
	Innermost	212	61	113	14	3	21	0	0	414
OCEAN	Outer	53	4	16	5	23	4	1	0	862
	Innermost	79	7	15	5	19	1	25	7	41,763
DYFESM	Outer	99	20	42	8	3	17	9	0	5,486
	Innermost	165	32	65	26	6	12	15	9	53,996
ARC2D	Outer	83	11	18	0	54	0	0	0	144
	Innermost	150	43	10	9	42	19	22	5	14,533
FLO52	Outer	70	9	8	0	23	28	2	0	2,016
	Innermost	118	11	30	6	9	35	23	4	20,370
TRFD	Outer	46	28	5	0	2	7	4	0	16,598
	Innermost	32	17	2	0	1	1	7	4	478,393
SPEC77	Outer	128	11	39	26	6	23	23	0	6,972
	Innermost	279	20	37	32	49	38	57	46	80,700
MATRIX300	Outer	2	0	2	0	0	0	0	0	1
	Innermost	15	1	9	0	2	2	0	1	51,815
NASA7	Outer	56	0	19	9	14	4	10	0	6,142
	Innermost	73	0	13	11	14	10	19	6	28,927
TOMCATV	Outer	6	0	1	0	5	0	0	0	84
	Innermost	12	1	3	0	3	0	5	0	11,614

Table 2.2 Loop Invocation Counts of PERFECT and SPEC

Bench Name	Loop Type	# of Loops Invoked	Number of Loops with Average Iteration Count						Average Iter. Per Loop	Average Iter. Per Invocation
			0	10	100	1000	10K	More		
ADM	Outer	43	1	27	15	0	0	0	16	3
	Innermost	86	0	18	66	2	0	0	44	12
SPICE	Outer	14	0	7	6	1	0	0	37	4
	Innermost	70	0	43	19	8	0	0	31	3
MGD	Outer	12	0	3	0	7	2	0	687	223
	Innermost	30	0	16	6	4	3	1	1,141	12
TRACK	Outer	23	0	12	7	4	0	0	95	81
	Innermost	57	0	31	15	10	1	0	11	9
BDNA	Outer	44	0	35	5	1	3	0	126	176
	Innermost	151	1	65	19	11	54	1	869	285
OCEAN	Outer	49	0	8	29	12	0	0	72	40
	Innermost	72	0	0	28	42	2	0	182	169
DYFESM	Outer	79	0	68	11	0	0	0	8	12
	Innermost	133	6	97	28	2	0	0	11	20
ARC2D	Outer	72	0	11	44	17	0	0	122	150
	Innermost	107	1	0	41	65	0	0	204	154
FLO52	Outer	61	0	26	34	1	0	0	15	16
	Innermost	107	0	4	77	26	0	0	75	58
TRFD	Outer	18	0	0	16	2	0	0	63	24
	Innermost	15	0	1	13	1	0	0	37	21
SPEC77	Outer	117	0	31	81	5	0	0	59	13
	Innermost	259	0	57	167	34	1	0	153	15
MATRIX300	Outer	2	0	0	0	2	0	0	300	300
	Innermost	14	0	1	0	13	0	0	279	300
NASA7	Outer	56	0	25	14	17	0	0	80	6
	Innermost	73	0	10	13	46	3	1	697	190
TOMCATV	Outer	6	0	0	0	6	0	0	255	255
	Innermost	11	0	0	0	11	0	0	256	255

Table 2.3 Loop Iteration Counts of PERFECT and SPEC

Bench Name	Loop Type	# of Loops Invoked	Number of Loops with Average Increment					Average Iter. Per Loop	Average Iter. Per Invocation
			0 to 1	1	1 to 2	2	More		
ADM	Outer	43	0	43	0	0	0	1.00	1.00
	Innermost	86	0	83	0	3	0	1.03	1.17
SPICE	Outer	14	0	14	0	0	0	1.00	1.00
	Innermost	70	0	70	0	0	0	1.00	1.00
MGD	Outer	12	0	12	0	0	0	1.00	1.00
	Innermost	30	0	30	0	0	0	1.00	1.00
TRACK	Outer	23	0	23	0	0	0	1.00	1.00
	Innermost	57	0	57	0	0	0	1.00	1.00
BDNA	Outer	44	0	44	0	0	0	1.00	1.00
	Innermost	151	0	151	0	0	0	1.00	1.00
OCEAN	Outer	49	0	46	0	3	0	1.06	1.09
	Innermost	72	0	70	0	0	2	1.08	1.00
DYFESM	Outer	79	0	79	0	0	0	1.00	1.00
	Innermost	133	0	133	0	0	0	1.00	1.00
ARC2D	Outer	72	0	72	0	0	0	1.00	1.00
	Innermost	107	0	107	0	0	0	1.00	1.00
FLO52	Outer	61	0	57	0	4	0	1.07	1.03
	Innermost	107	0	102	0	5	0	1.05	1.01
TRFD	Outer	18	0	18	0	0	0	1.00	1.00
	Innermost	15	0	15	0	0	0	1.00	1.00
SPEC77	Outer	117	0	109	0	6	2	1.14	1.51
	Innermost	259	0	237	0	22	0	1.08	1.59
MATRIX300	Outer	2	0	2	0	0	0	1.00	1.00
	Innermost	14	0	14	0	0	0	1.00	1.00
NASA7	Outer	56	0	55	0	0	1	1.05	1.00
	Innermost	73	0	73	0	0	0	1.00	1.00
TOMCATV	Outer	6	0	6	0	0	0	1.00	1.00
	Innermost	11	0	11	0	0	0	1.00	1.00

Table 2.4 Loop Increments for PERFECT and SPEC

Table 2.4 shows the average increment of each loop. **ADM**, **OCEAN**, **FLO52**, **SPEC77** and **NASA7** have a few loops whose increment is not one. Especially, **SPEC77** has two as the increment in the half of the loop invocations. The increments for all other benchmarks are one.

2.3 Pcode Cache Simulation

In addition to the loop profile information, some cache simulation information is useful to determine whether a loop nest should be transformed or not. Since the purpose of the transformation is to reduce the cache misses, any loop nest with few cache misses should not be transformed in order to avoid the possible performance penalty due to the instruction overhead introduced by the transformation. Simulation is the process of selecting a set of inputs for a program, executing the program with these inputs, and simulating the cache behavior of the program. The following information is simulated by the Pcode cache simulator:

(1) The cache compulsory misses, capacity misses and conflict misses.

(2) The cache anti-conflict hits.

Pcode cache simulation is supported by three tools: a probe insertion program, a cache simulator and a program to summarize cache simulation information. There are three steps to perform Pcode cache simulation:

(1) The probe insertion program assigns a unique identifier or line number to each array reference and inserts a probe just before the array reference in Pcode. The probe is inserted only for reads if the cache has does not allocate line when a write-miss occurs. Whenever the prove is activated, it produces the associated memory address and identifier.

Bench Name	Array Access in Par_loop		Other Array Accesses (%)	Total Array Access Counts
	Outer (%)	Innermost (%)		
TRACK	17,704,495 (33.4)	14,862,168 (28.1)	20,412,350 (38.5)	52,979,013
BDNA	91,292 (0.0)	318,842,870 (100.0)	833 (0.0)	318,934,995
OCEAN	1,909,884 (0.1)	1,602,459,335 (97.3)	42,489,908 (2.6)	1,646,859,127
ARC2D	1,441,725 (0.1)	1,459,178,576 (99.9)	2 (0.0)	1,460,620,303
MATRIX300	300 (0.0)	432,000,000 (100.0)	0 (0.0)	432,000,300
NASA7	1,221,370 (0.1)	2,080,443,485 (99.9)	162,807 (0.0)	2,081,827,662
TOMCATV	0 (0.0)	325,218,099 (100.0)	0 (0.0)	325,218,099

Table 2.5 Array Accesses of Each Loop Type of Several Benchmarks

(2) The Probed Pcode is reverse-translated into C code. The C code is compiled by a standard C compiler and linked with the cache simulator program.

(3) The executable program is executed for a set of inputs. The generated cache simulation data for each line is collected.

(4) Finally, the summary cache simulation information is generated by the program using the cache simulation data for each line. The cache simulation data for each loop nest is also generated using the loop nest information. This information is then merged back into the Pcode representation of the program.

Table 2.5 shows some characteristics of array accesses of several benchmarks. In all benchmarks except **TRACK**, the *Innermost* loops contain the vast majority of the array accesses in the program. The array accesses in **TRACK** occur not only in the *Innermost* loops but also in *Outer* loops and the other regions where the accesses are not within loops.

Bench Name	Loop Type	Number of Par_loops					
		Array Access Rate					Total
		0	0.1	1.0	10.0	more	
TRACK	Outer	3	5	4	2	1	15
	Innermost	6	15	15	16	0	52
BDNA	Outer	14	24	0	0	0	38
	Innermost	49	78	6	4	2	139
OCEAN	Outer	0	8	0	0	0	8
	Innermost	4	25	15	11	4	59
ARC2D	Outer	1	14	0	0	0	15
	Innermost	41	50	13	33	0	137
MATRIX300	Outer	0	1	0	0	0	1
	Innermost	1	0	0	0	1	2
NASA7	Outer	0	15	0	0	0	15
	Innermost	0	20	11	14	3	48
TOMCATV	Outer	0	0	0	0	0	0
	Innermost	1	5	0	2	3	11

Table 2.6 Array Access Statistics for Several Benchmarks

Table 2.6 shows the number of loops with each array access counts for several benchmarks. **BDNA** and **ARC2D** have many loops that do not read from arrays. Also, in **BDNA** a few loops contain the vast majority of the array accesses.

2.4 Experiments

The experiments in this section show the effect of the cache structure for numerical programs. Since there exist a lot of possible cache configurations, testing all configurations is impossible with available computer resources and time. To test the effect of a particular cache parameter, a base cache was defined and a single parameter was varied

on each simulation. The selected base cache was a 8K, direct mapped cache with 16 byte cache line, using a write-back and no-allocate-on-write-miss policy.

2.4.1 Cache Size

The first cache characteristic is the size. The cache simulations examined the effects of increasing cache size from 8K to 64K bytes. As shown in Figure 2.2, for all the benchmarks, the total cache misses as well as the cache capacity misses decreased as the data cache size increased. The cache conflict misses not necessarily decreased as the cache size increased since the conflict misses come from the lack of the cache line associativity, not directly from the lack of the cache size. Regarding the capacity misses, increasing the cache size decrease the cache capacity misses in the most cases. For **TOMCATV**, however, doubling the cache size from 16K did not decrease the cache capacity misses as well as the total cache misses. This may happens if the working set size is larger than the cache size. Cache size does not affect the cache line overhead.

2.4.2 Set Associativity

The second cache parameter examined was set associativity. The cache simulations examined the effects of increasing set associativity from direct-mapped to 8-way set. As shown in Figure 2.3, for most benchmarks, the total cache misses did not necessarily decrease as the set associativity increased. Although increasing the set associativity can decreases the cache conflict misses in almost all cases, it increases the capacity misses

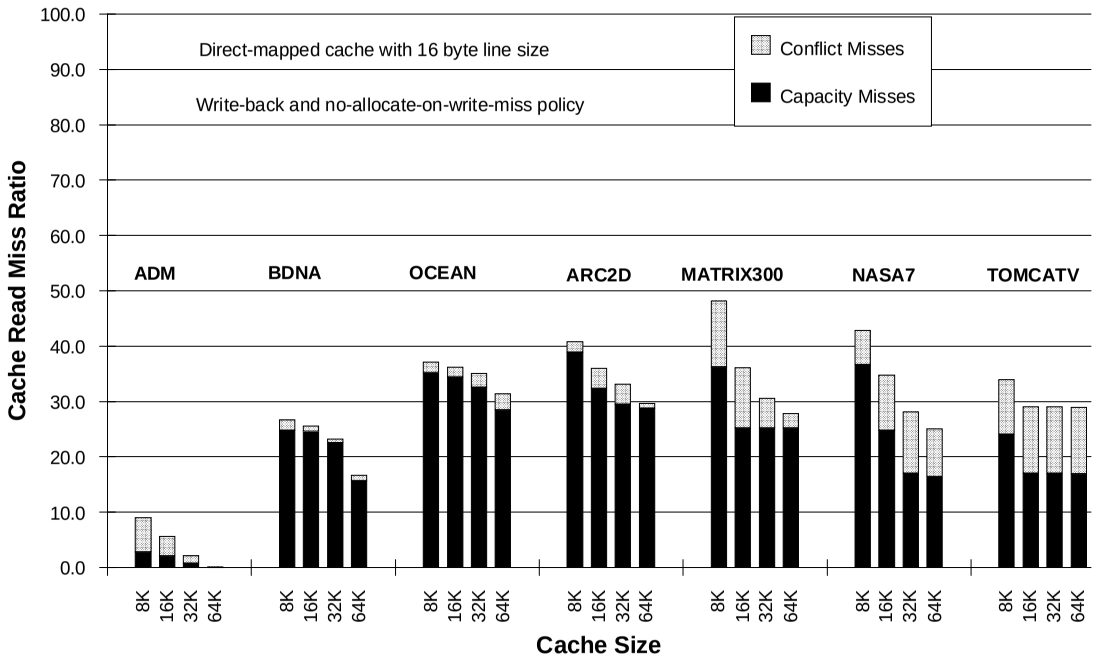

Figure 2.2 Effect of Cache Size on Cache Miss Ratios

instead. The improvement of the conflict misses is easily canceled by the increase of the capacity misses. Like cache size, set associativity does not affect the cache line overhead.

In general, the cache with higher set associativity need more hardware to compare the tags. Also, the extra hardware may slow down the cache speed. Especially, a direct-mapped cache is much faster than a set associative cache since it can fetch the data from cache into a data register while comparing the tag. If the tag is not matched, the content of the data register can be just ignored.

2.4.3 Line Size

The third cache parameter examined was cache line size. The cache simulations examined the effects of increasing cache line from 32 to 8 bytes as shown in Figure 2.4.

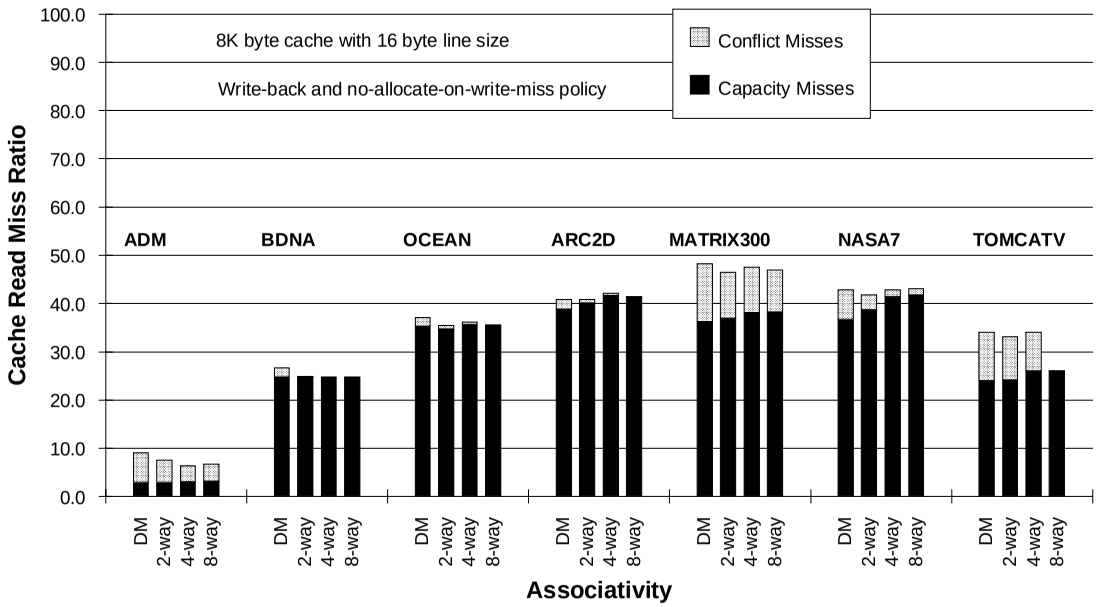

Figure 2.3 Effect of Set Associativity on Cache Miss Ratios

Overall, decreasing the cache line size increases the cache misses since smaller cache line cannot take the advantage of the locality of the reference. Although smaller cache line can effectively decrease the cache line fetch, the advantage is easily cancelled by the significant increase of the cache misses due to the small cache line.

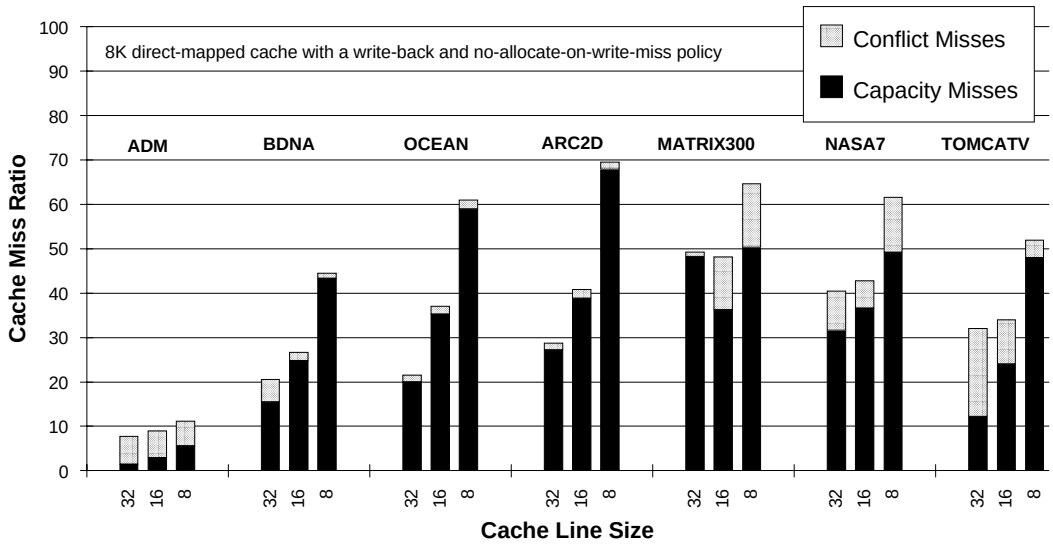

Figure 2.4 Effect of Cache Line Size on Cache Miss Ratios

CHAPTER 3

LOOP BLOCKING

Loop blocking is an optimization technique to increase the chance for data reuse in the cache. The basic idea is to reduce the number of accesses between two accesses to the same datum. In this method, as illustrated in Figure 3.1, a large iteration space (Figure 3.1a) is divided into smaller sub-iteration spaces (Figure 3.1b) so that the array accesses that occur within each sub-iteration space can fit within the cache. This more localized memory reference pattern can increase data reuse within the cache and reduce processor stalls due to cache misses.

3.1 Method

The original program is transformed in order to divide the iteration space of loops into smaller sub-iteration spaces. Using this method, no special hardware is necessary at run-

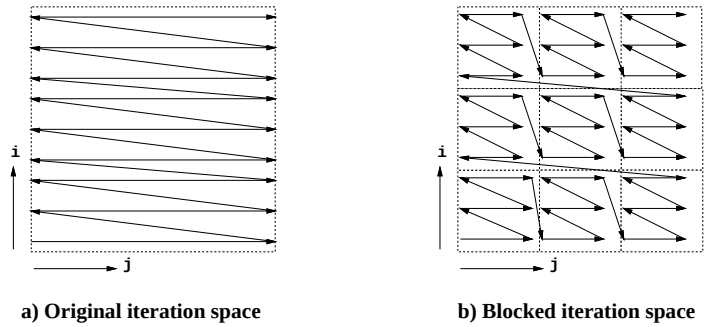

Figure 3.1 Concept of Loop Blocking

time. Instead, a sophisticated compiler is essential to perform the blocking transformation correctly and effectively. The compiler need identify the following:

- the loop nests that are valid to block.
- the loop nests that should be blocked to improve cache performance.
- the blocking size that should be used to improve cache performance effectively.

The automatic blocking transformation has been implemented as one of the transformations for the IMPACT compiler.

3.2 Compiler Support

The loop blocking transformation consists of three phases:

- Phase 1: Validity Analysis
- Phase 2: Blocking Decision
- Phase 3: Code Transformation

3.2.1 Validity Analysis

Validity analysis is done as the first phase in order to confirm that the blocking transformation is valid, which means that the result of the computation after the blocking transformation is exactly the same as that before the transformation. In the current implementation, only perfectly nested and structured loops are considered as the candidate

for loop blocking. Loop nests that contain multiple inner loops, return, goto, or break statements are excluded, as well as which contain subroutine calls with possible side effects. Then, for the remaining loop nests, the dependence analysis checks the validity of loop blocking.

The dependence analysis and the criteria for loop blocking are same as those for the loop interchange. Loop blocking is an iteration space reordering transformation, and therefore, may reverse dependence directions. Therefore, the dependence direction vector can be used to check whether or not a loop nest can be legally blocked. If the dependence direction vectors for the transformed loop nest are not lexically positive, the transformation will be invalid. The dependence analyzer used by my implementation is a version of the Omega test tailored to IMPACT by Grant Haab.

3.2.2 Blocking Decision

In the second phase of loop blocking, a fast compile time cache simulation is done in order to determine the effective blocking size. In the current implementation, the address of each array reference is calculated by a fast approximate method which uses a constant number for the size of each array dimension and the coefficient of each loop iteration variable in the array subscript. Therefore, the loop nests which contain arrays whose subscripts include variables other than loop iteration variables are excluded. Furthermore, the starting address of the array itself is very hard or impossible to determine at compile time; thus, some approximation is necessary in order to perform cache simulation at compile time.

Using a given cache size, the simulator estimates the cache misses for a blocked loop nest, starting from the smallest blocking size or one. Then, the blocking size is expanded in each dimension. In order to speedup the blocking size determination, each dimension for which blocking is to be applied is expanded by factors of power of 2 (i.e. 2, 4, 8, 16 and so on). Then, the cache miss ratio for each expanded blocking size is estimated. The blocking size with the lowest miss ratio among the expanded blocked loop nests is picked. This expansion process is repeated until the blocking size is expanded a predefined number of times. If the estimated cache misses for the finally selected blocking size is less than a predefined threshold, use this blocking size for the transformation. Otherwise, this loop nest is not blocked. Thus, the computational complexity of the algorithm is O (DlogN), where D is the number of dimensions and N is the number of iterations in each dimension.

3.2.3 Code Transformation

As the final phase of the blocking transformation, the loop nest is actually modified into the blocked loop nest with the blocking size determined by the previous phase.

A doubly-nested loop is given as an example in Figure 3.2a. The accesses to array B are sequential within each row, while the accesses to array C are sequential within each column, causing low utilization of the cache storage. The loop nesting after the blocking transformation is shown in Figure 3.2b.

a) Original code

```
for (i=0;i<N;i++)
  for (j=0;j<N;j++)
    A[i][j]=B[i][j]*C[j][i];
```

b) After blocking

```
for (ii=0;ii<N;ii+=Bi)
  for (jj=0;jj<N;jj+=Bj)
    for (i=ii;i<min(N,ii+Bi);i++)
      for (j=jj;j<min(N,jj+Bj);j++)
        A[i][j] = B[i][j] * C[j][i];
```

Figure 3.2 Code Transformation for Loop Blocking

3.3 Experimental Evaluation

In this section, the cache miss ratio of the loop blocking are evaluated through cache simulations.

3.3.1 Statistics for Loop Blocking

Table 3.1 shows the statistics of blockable loops of several PERFECT benchmarks. As described in Section 3.2.1, unstructured loops, imperfectly nested loops and loops that contain calls with possible side effects are excluded. Then, dependence analysis is performed for the remained loops. The last column of the Table 3.1 shows the number of blockable loop nests. However, most of them are singly-nested loops that do not benefit from loop blocking since the access pattern would not change.

For the blockable loops, the cache miss is estimated using various blocking sizes. As described before, the singly-nested loops has show no cache performance improvement through loop blocking as shown in Table 3.2. The last column of Table 3.2 shows the number of the blocked loop nests. Several benchmarks have blocked loop nests, but the

Bench	# of Loops Analyzed	Excluded Loops				Blockable Loops				
		Unstruct Loops	Imperfect Nest	S. Effect Calls	Depend Invalid	# of Loops	Nest Depth 3	2	1	Total Loopnests
ADM	302	13	50	51	20	198	9	25	121	155
SPICE	348	87	22	245	0	55	0	0	55	55
MGD	52	5	19	6	2	29	0	0	29	29
TRACK	97	25	15	29	2	53	0	4	45	49
BDNA	270	26	40	71	0	158	0	8	142	150
OCEAN	132	4	34	22	26	79	0	8	63	71
DYFESM	264	15	62	83	9	124	0	18	88	106
MG3D	111	1	42	27	0	62	0	13	36	49
ARC2D	233	2	31	9	0	198	5	46	91	142
FLO52	188	14	35	22	4	128	7	27	53	87
TRFD	78	4	30	3	0	37	0	8	21	29
SPEC77	407	29	110	63	18	249	0	28	193	221
Total	2482	225	490	631	81	1370	21	185	937	1143

Table 3.1 Blockable Loops for PERFECT Benchmarks

number of the blocked loop nests is relatively small compared with the total number of the loop nests.

3.3.2 Cache Miss Ratios for Blocked Loop Nests

Table 3.3 shows dynamic array access statistics for the blocked loops. Among the eight benchmarks that have the blocked loop nests, only two benchmarks, **BDNA** and **ARC2D**, have blocked loop nests that contain relatively high percentages of array access counts.

Figure 3.3 shows the cache miss ratio of the blocked loop nest and the entire loop nests of **BDNA**. **BDNA** has a very frequently executed loop nest that contains many arrays, and therefore, loop blocking is very successful in improving the cache performance.

Figure 3.4 shows the cache miss ratio of the blocked loop nest and the entire loop nests of **ARC2D**. Unlike **BDNA**, **ARC2D** has relatively many blocked loop nests, but

Bench	Blockable Loops				No Improvement				Blocked Loops			
	Nest Depth			Total	Nest Depth			Total	Nest Depth			Total
	3	2	1	Loopnests	3	2	1	Loopnests	3	2	1	Loopnests
ADM	9	25	121	155	1	3	121	125	8	22	0	30
SPICE	0	0	55	55	0	0	55	55	0	0	0	0
MDG	0	0	29	29	0	0	29	29	0	0	0	0
TRACK	0	4	45	49	0	4	45	49	0	0	0	0
BDNA	0	8	142	150	0	3	142	145	0	5	0	5
OCEAN	0	8	63	71	0	0	63	63	0	8	0	8
DYFESM	0	18	88	106	0	16	88	104	0	2	0	2
MG3D	0	13	36	49	0	13	36	49	0	0	0	0
ARC2D	5	46	91	142	3	26	91	120	2	20	0	22
FLO52	7	27	53	87	6	25	53	84	1	2	0	3
TRFD	0	8	21	29	0	2	21	23	0	6	0	6
SPEC77	0	28	193	221	0	21	193	214	0	7	0	7
Total	21	185	937	1143	10	113	937	1060	11	72	0	83

Table 3.2 Blocked Loop Nests for PERFECT Benchmarks

Bench	Total Array Access	Blocked Loops			
		# of Loopnests		Array Access	% of Total Array Access
		Source	Executed		
ADM	462,824,761	30	0	0	0
BDNA	436,028,173	5	5	285,750,080	65.5
OCEAN	2,669,877,945	8	0	0	0
DYFESM	410,046,035	2	2	1,873,872	0.5
ARC2D	1,981,154,040	22	14	561,837,852	28.4
FLO52	736,423,318	3	1	5,715,360	0.7
TRFD	666,159,027	6	0	0	0
SPEC77	2,530,747,172	7	0	0	0

Table 3.3 Array Access Counts of PERFECT

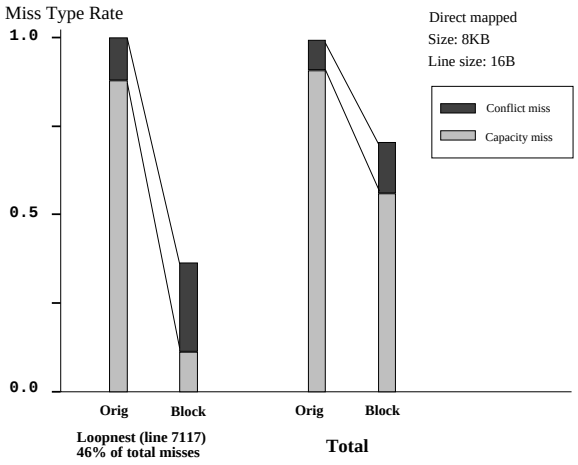

Figure 3.3 Cache Miss Detail of BDNA

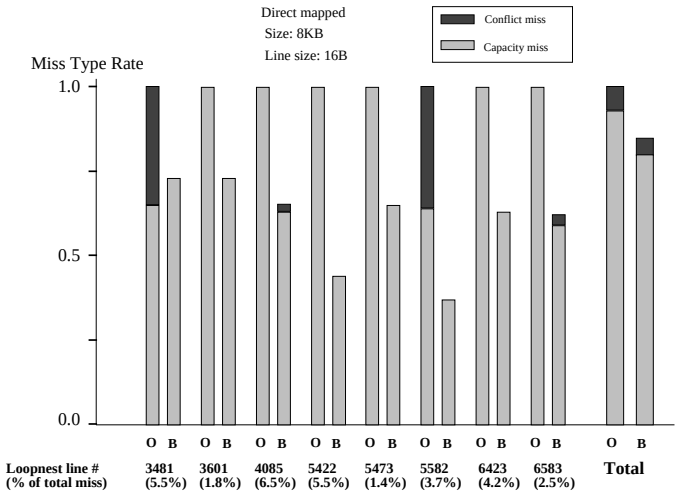

Figure 3.4 Cache Miss Detail of ARC2D

Effective ? Somewhat effective Not effective

	Capacity misses	Conflict misses	Line fetch overhead
Large Cache	?	×	N/A
Highly Associative Cache	×	?	N/A
Small Cache Line Size	×	×	●
Loop Blocking	?	×	N/A

Table 3.4 Motivation for a More Aggressive Method

each execution percentage is small. Therefore, the entire benchmark cache performance is not improved significantly.

Loop blocking as well as changing cache parameters as described in Section 2.4 does not solve all cache problems at the same time. This motivates a more aggressive method that can solve all the cache problems at the same time (Table 3.4).

CHAPTER 4

DATA RELOCATION AND PREFETCHING

4.1 Method

I propose a compiler-supported, hardware-based technique called data relocation and prefetching in order to improve the data cache performance. In this method, the array references in the inner loop of a nest are sequentially mapped in the cache before they are accessed. The relocation operations are invoked by explicit instructions inserted by the compiler. The compiler also inserts a declaration into the original code for the relocation buffer that allocates space for the relocated data in memory. Special hardware that is attached to the cache unit maps and compresses the data into the virtual buffer space so that the relocation can be performed while prefetching the data from the memory to the cache and without stalling the *CPU*. In order to access the relocated data (instead of the original array data) during the computation, the compiler replaces the original array references with corresponding relocation buffer references.

Because the array data is relocated, the prefetch is binding. During the computation, the newly assigned address in the relocation buffer space is used to access the data rather than the original address. Consequently, the relocation must be completed before the computation on the same data begins. If the relocated, cached data is replaced by

some other original address accesses, such as scalar accesses, the relocated data must be written back to the relocation buffer in memory since the accesses in the computation use the address of the relocated data. To insure write-back of the relocated data, the dirty bit is set when the cache line for the data is allocated. When the computation that uses the relocated data is finished, all modified, relocated data are written back from the relocation buffer to their original memory locations using an explicit machine instruction.

Data Relocation and Prefetching (*DRP*) can improve the spatial locality of array accesses for a loop nest. Figure 4.1 shows how array data elements accessed in the first iteration of the outer loop are copied to sequential cache locations that map to the relocation buffer in memory. Array **A** is accessed with a stride of two, and array **B** is accessed in column order during the execution of the inner loop. Accesses to these array elements can result in poor performance because:

(1) The accesses may not exhibit any spatial locality because of the non-unit access stride, resulting in wasted cache capacity which may lower the cache hit rate.

(2) The sets of accesses for different arrays may conflict with each other because they happen to be mapped to some of the same locations in the cache.

(3) The accesses for a single array may conflict with each other because of a large access stride.

If the accessed elements of these arrays are relocated in the cache, spatial locality can be improved by packing elements of the arrays into contiguous locations. Also, since only necessary elements are brought into the cache, the extra memory requests and time to

a) Original Code

```
for (i = 0; i < N; i++)
   for (j = 0; j < 3; j++)
      ... = A[i][2*j] + B[j][i];
```

b)Transformed Code

```
for (i = 0; i < N; i++)
   for (j = 0; j < 3; j++)
      ... = A'[j] + B'[j];
```

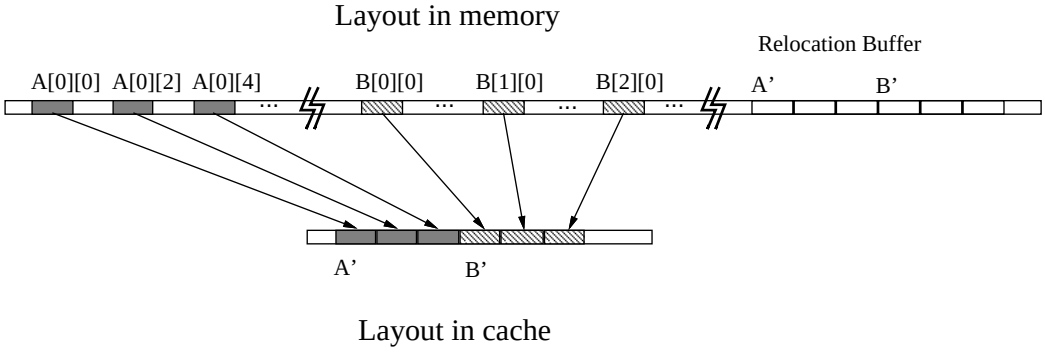

Figure 4.1 Concept of Data Relocation

fill the cache line due to the non-unit stride accesses are reduced. Furthermore, if the total size of the relocated array elements is smaller than the cache size, the compression guarantees that the references to the relocated data do not conflict with each other in the cache. Finally, cache space is conserved by packing elements of the arrays.

In order to reduce the instruction-fetch overhead due to the inserted relocation instructions, each instruction contains enough information to operate on several elements of the array in sequence. Also, in order to accommodate the latency of the relocation of array data, the relocation and computation phases are separated in time by software-pipelining the outer loop. Further details are given in Section 4.4.

4.2 Architectural Support

Implementing the mechanism for the *DRP* technique requires extra instructions as well as extra hardware. Five instructions to support *DRP* are added to the processor instruction set: *precollect*, *distribute*, *preallocate*, *await* and *finishup*.

4.2.1 Precollect

The *precollect* instruction, which has five operands, collects the array data referenced in a computation into consecutive locations in the cache before the data is needed for the computation. The *precollect* instruction fetches the data at the original addresses from the cache. If the cache misses the original address, memory is accessed to get the data instead. After fetching the data from either the cache or memory, the data is stored into the associated cache line with the tag of the relocated addresses. The dirty bit of the cache line is set so that the relocated data can be written back to memory at the relocated address if it is replaced with other data. Therefore, the relocated data in memory can be referenced by the transformed code even if the cache misses the relocated address. This instruction is non-blocking: It does not stall the processor even if it causes cache misses. Also, this instruction is non-trapping: It does not interrupt the processor even if this instruction accesses memory illegally. Therefore, the execution of this instruction can be overlapped with the execution of the instructions which perform computations in the loop nest.

The first operand is the address of the first element of the array to be relocated, whereas the second operand is the address of the first element of the relocated array in the relocation buffer. The third through the fifth operands are the size of each array element in bytes, the stride of the array accesses in bytes, and the number of array elements to be collected. The information given by the third through the fifth operands is necessary since the *DRP* technique does not relocate entire cache lines but rather relocates array elements that will be accessed in the computation.

4.2.2 Distribute

The *distribute* instruction writes the relocated data which were updated by the computation back to their original array locations in memory. First, the cache is accessed to read the updated data using the relocated address. If the cache misses the relocated address due to the replacement, memory is accessed instead. After reading the updated data from either the cache or memory, the data is written in the cache using the original address. If the cache misses the original address, the data is written in memory instead. Like the *precollect* instruction, the *distribute* instruction does not stall the processor.

The format of the *distribute* instruction is the same as that of the *precollect* instruction: The first operand specifies the starting address of the array elements before relocation, and the second operand specifies the starting address of the array data in the relocation buffer.

```
a) Original code

    for (i=0;i<N;i++)
      for (j=0;j<8;j++) {
        A[i][j] = 1.0;
        ... = A[i][j] * 2.0;
      }

b) Transformed code using preallocate operation

    for (i=0;i<N;i++) {
      /* preallocate A'[0]..A'[7] */
      for (j=0;j<8;j++) {
        A'[j] = 1.0;
        ... = A'[j] * 2.0;
      }
      /* distribute A'[0]..A'[7] to A[i][0]..A[i][7] */
    }
```

Figure 4.2 Example of Preallocate Operation

4.2.3 Preallocate

The *preallocate* instruction only allocates the necessary cache lines instead of collecting the array data into consecutive locations within the cache and can be used instead of the *precollect* instruction if the array data is not read before being written during the computation as shown in Figure 4.2. In this case, there is no need to fetch the data into the cache before the computation. Unlike the *precollect* operation, the dirty bit of the cache line is not set since the data is not fetched into the cache line. Typically, tags of the cache lines can be modified rapidly by the hardware. Also, since this operation does not access memory at all, it does not increase the bus traffic at all. This instruction is also non-blocking.

The *preallocate* instruction has the same operands as the *precollect* instruction, except that the first operand (the original starting address) and the fourth operand (the stride of the original array accesses) are not needed.

4.2.4 Await

The *await* instruction provides a simple synchronization mechanism to avoid accessing the relocated array data before the *precollect* or *preallocate* operation is completed. This instruction has a single operand that must match the second operand of the associated *precollect* or *preallocate* instruction. If the operation of the associated *precollect* or *preallocate* instruction has not been completed, the *await* instruction blocks the data cache until the operation is finished. The processor is not blocked until it accesses the data cache.

4.2.5 Finishup

After the loop nest finishes execution, there may be unfinished *distribute* instructions which must be completed before the array data is accessed in subsequent code. The *finishup* instruction provides a synchronization mechanism to insure that all *distribute* instructions are completed by blocking the data cache. No operands are necessary.

4.3 Hardware Support

The execution of the *DRP* instructions is handled by special hardware called the *Data Relocation and Prefetching unit* which can be attached to the existing cache unit (Figure 4.3). The *DRP* unit shares the *MMU* with the cache and also shares the cache itself with the *CPU*. In this configuration, both the processor and the cache have higher priority than the *DRP* unit in accessing the shared resources. This priority hierarchy helps to ensure that the *DRP* unit doesn't significantly slow down the execution of an application program while executing *precollect*, *preallocate* and *distribute* instructions unless an *await* or *finishup* instruction is executed. The data cache configuration necessary to support the *DRP* unit is a write-back cache that does not allocate a cache line on a write miss. If a high-bandwidth memory system like a split-transaction bus system is used, the *DRP* unit can utilize the bandwidth by pipelining the read and write requests to the memory system. Figure 4.4 illustrates the components of the *DRP* unit data path.

The *DRP* unit insures that the cache need not block due to a data fetch generated by any *DRP* instruction by performing these fetches in the *DRP* unit itself. Since the cache does not block while writes are serviced unless the write buffer is full, writes misses generated by the *DRP* instructions are handled by the cache instead of dedicated *DRP* hardware. Furthermore, the *DRP* unit itself does not block when fetching data from a memory location.

As program execution proceeds, *precollect*, *preallocate*, and *distribute* instructions from the processor are placed in the instruction queue. If the instruction queue is full,

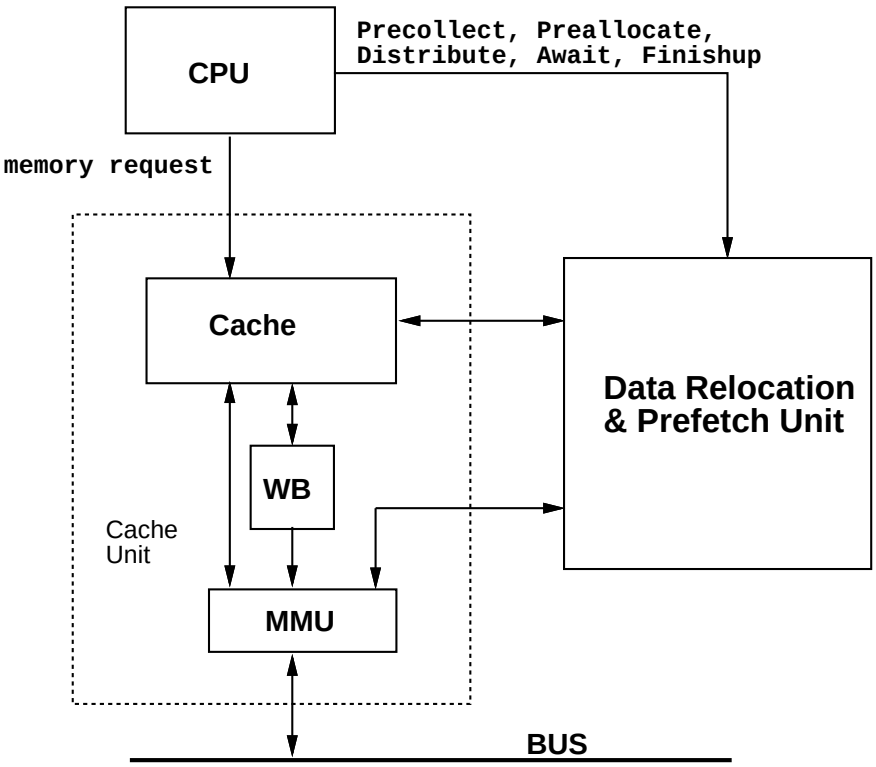

Figure 4.3 Data Relocation and Prefetch Unit Interfaces

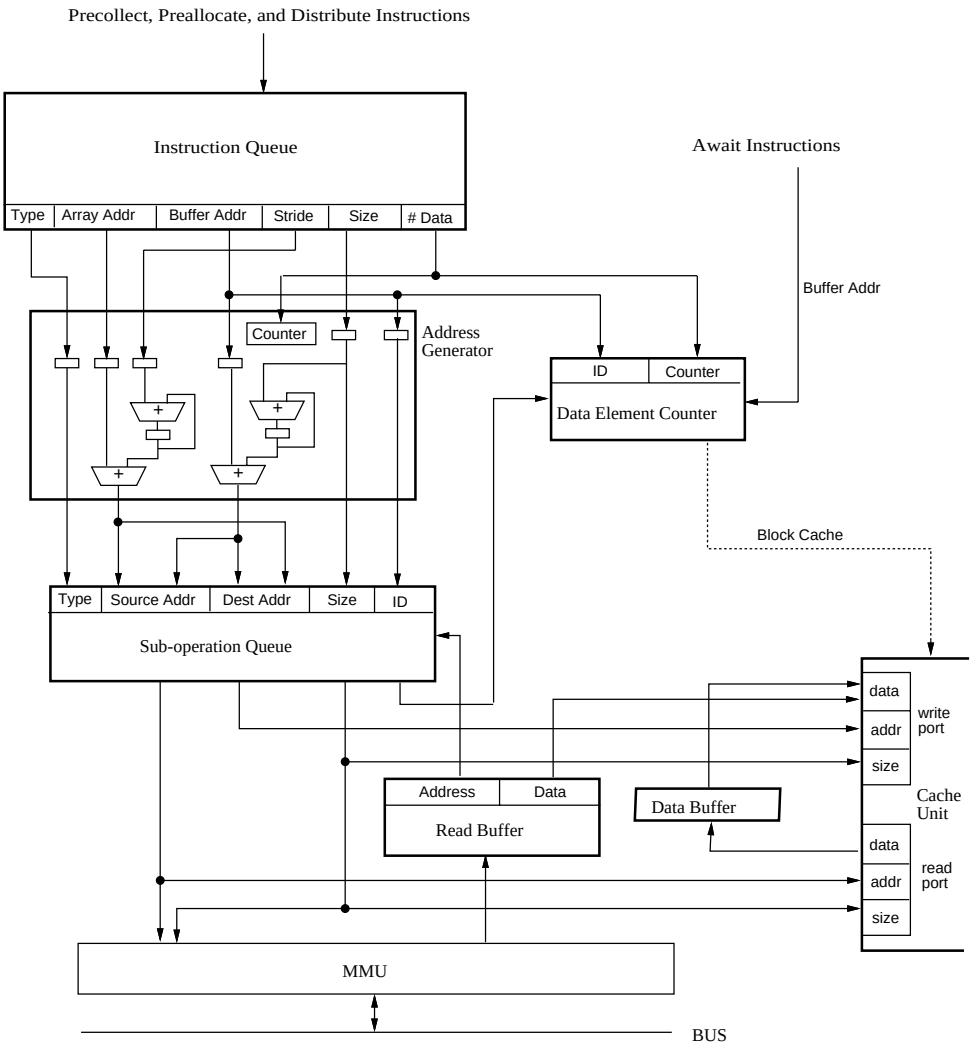

Figure 4.4 Data Relocation and Prefetch Unit Data Path

the *DRP* unit stalls the processor until there is an empty entry in the queue. Each instruction at the head of the queue is processed completely by the address generator before the next may proceed.

The address generator calculates the original address of each array element using the starting array address and the stride information. The address generator also uses the

starting relocation buffer address and the element size to calculate the relocation buffer address for each array element.

As each pair of addresses is generated, an entry is stored in the sub-operation queue for processing. For the *precollect* and *preallocate* instructions, the original addresses are stored in the source address field, and the buffer addresses are stored in the destination field of the sub-operation queue. For *distribute* instructions, the addresses are reversed.

To begin processing a *precollect* sub-operation, the source address field is used to send a special read request to the cache. If the data is present in the cache, it is first stored in the *DRP* unit data buffer then written to the cache using the destination address. Also, the sub-operation is removed from the queue since it is finished.

If the data is not present in the cache, the cache does not send a read request to fetch the data, but instead, the *DRP* unit sends the read request to the *MMU* using the source address. At this point, the next sub-operation in the sub-operation queue begins processing, but the pending sub-operation is left in the queue for further processing. If the sub-operation queue has enough entries, the queue does not block while a sub-operation is waiting for memory access to be completed.

When the data returns from memory to the read buffer, the source address fields of the sub-operation queue are searched associatively using the address from the read buffer to obtain the destination address and size of the data from the same sub-operation. Then, the appropriate cache line is allocated, and the data from the read buffer is written to the cache using the destination address and size. All blocks written with relocated data in the cache are marked dirty.

To begin processing a *preallocate* sub-operation, the source address field is used to access the cache tag store. For each address that misses in the cache, a cache line is allocated for the relocated data, but the array data is not fetched into the cache. After the line is allocated, it is marked clean. Then, the sub-operation is removed from the sub-operation queue. The *preallocate* sub-operation must finish before the relocated address is written by the computation in order to insure correct results are obtained for program execution.

For a *distribute* sub-operation, the relocated data is read from the cache using the source address field from the entry at the head of the sub-operation queue. If the data is present in the cache, it is placed in the *DRP* unit data buffer. If the data is not present in the cache (which can occur if non-relocated data conflicts with relocated data in the cache), the cache does not fetch the data; but instead, the *DRP* unit sends a read request to the *MMU* using the source address. At this point, the next sub-operation in the queue begins processing, but the pending sub-operation is left in the queue for further processing. When the data returns from memory, the address in the read buffer is used to associatively search the source address fields of the sub-operation queue for the destination address and size fields.

After the relocated data has been read, a write request is sent to the cache using the destination address and data in either the read buffer or the data buffer, depending on whether or not the read of the relocated data resulted in a cache miss or hit, respectively. The sub-operation is removed from the queue, and the data is written to the original address in the cache if a write hit occurs or in the memory if a write miss occurs.

When each instruction is passed from the instruction queue to the address generator, an entry is allocated in the data element counter for it, and the total number of data elements is used to initialize the counter. As the sub-operation queue finishes the processing of each data element, the data element counter is decremented by one. Therefore, a *preallocate*, *precollect*, or *distribute* instruction has a corresponding non-zero data element counter entry during execution. An *await* instruction causes the data cache to block until the data element counter entry with the matching buffer address identifier reaches zero. A *finishup* instruction causes the data cache to block until all the data element counter entries contain zero.

An interlock mechanism insures that a *precollect* or *preallocate* instruction will not begin execution until after all previously issued *distribute* instructions which use the same relocation buffer have completed execution. The identifier (buffer address) of the instruction at the head of the instruction queue is used to search the data element counter associatively. If an instruction with the same identifier is still being executed, address generation is delayed until the previous instruction has been completed.

4.4 Compiler Support

Compiler support is essential to transforming the source code so that the proposed scheme can improve the cache performance. The *IMPACT* research prototype compiler [17] supports high-level transformations and optimizations, superscalar and *VLIW* optimizations and scheduling [18], as well as classical and machine-specific optimizations.

Some high-level loop analysis information such as loop carry dependences is passed to the low-level code so that the low level optimization can be performed effectively.

The compiler support for the *DRP* technique consists of five phases: loop analysis, array analysis, *DRP*-transformation decision, high-level code transformation and low-level code transformation. The first three phases are performed at the Pcode level, while the last phase is performed at the Lcode level.

4.4.1 Loop Analysis

In this phase, each loop nest is analyzed to make the transformation valid and effective. First, innermost loops which contain return or goto are excluded since the transformation would be invalid. Furthermore, loop nests that contain subroutine calls with possible side effects are also excluded because the subroutine could modify global arrays which have been relocated. Then, loop information is generated using static and profile information. The profile information contains iteration counts, array access counts, and cache misses for each loop. Since transformations may introduce instruction overhead and code expansion, only important loops should be transformed. The array access counts and cache misses for each loop are used in order to determine whether the loop is important. If it is considered as a unimportant loop, the loop is removed from consideration. Iteration counts for each loop are necessary to determine whether the inner-most loop has enough array elements to be relocated, and whether the outer loop has enough iterations to reduce the startup overhead.

```
Example:

  for (ii=0;ii<N;ii+=Bi)
    for (jj=0;jj<N;jj+=Bj)
      for (i=ii;i<min(N,ii+Bi);i++)
        for (j=jj;j<min(N,jj+Bj);j++)
          A[i][j] = B[i][j] + A[i][j];

 - i is dominated by ii
 - j is dominated by jj
```

Figure 4.5 Relationships of Iteration Variables in Blocked Loop

Also, in this phase, the relationships between iteration variables in a loop nest are determined. This information is necessary to prefetch elements of an array which will be used in the next inner-most loop invocation. The index of the elements are determined using this information. For example, if the loop nest is blocked as shown in Figure 4.5, the initial value of j is determined by jj. In other words, the iteration variable j is dominated by jj, which means that if jj is incremented, j also need to be incremented. Similarly, the iteration variable i is dominated by ii. Therefore, the next iteration of the i loop is determined by the current ii loop iteration and the relationships as illustrated in Figure 4.6.

4.4.2 Array Analysis

The next phase is array analysis. In this phase, all array references in each loop nest are analyzed and stored in a data structure. Data dependence analysis is performed using

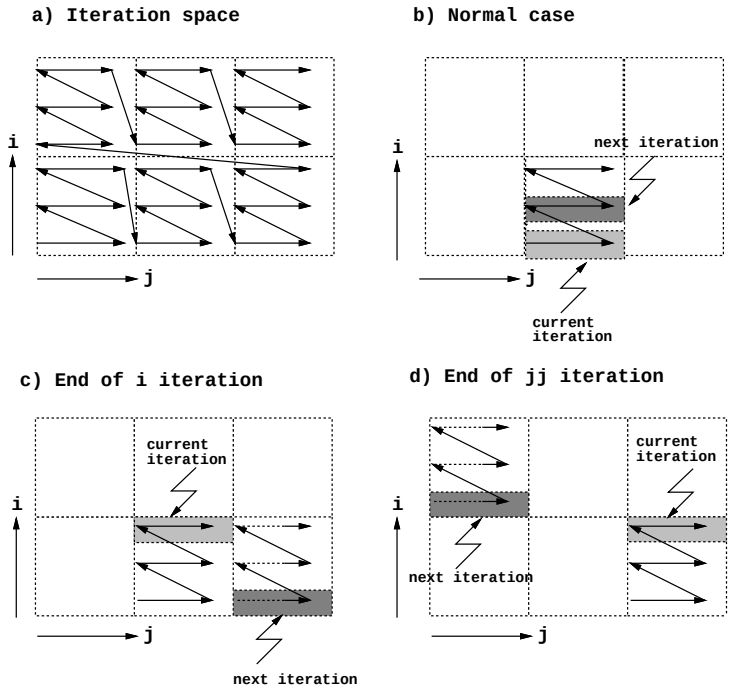

Figure 4.6 Iteration Space in Blocked Loop

Omega test [14][15]. The data dependence information is used to insure the validity of software-pipelining the data relocation and computation phases.

The relationships among the array references are also determined in this phase. If the relationship between two array references is determined to be *in line*, they are grouped in one array identifier and they can share the relocation buffer. For example, as shown in Figure 4.7, if the array references in a loop nest are A[i][j] and A[i][j+1], these array references are overlapped with regard to loop j and the relationship is considered as *in line*. It is much more effective to use one shared buffer for both array references since the buffer space can be saved and the overhead for data relocation can be reduced. Furthermore, if the buffer is not shared by the two array references, an inconsistency

```
a) Original code

    for (i=0;i<N;i++)
      for (j=0;j<8;j++)
        A[i][j+1] = A[i][j+1] + A[i][j];

b) Transformed code without sharing relocation buffer

    for (i=0;i<N;i++) {
      /* precollect A[i][0]..A[i][7] to A'[0]..A'[7] */
      /* precollect A[i][1]..A[i][8] to A''[0]..A''[7] */
      for (j=0;j<8;j++)
        A''[j] = A''[j] + A'[j];
      /* distribute A''[0]..A''[7] to A[i][1]..A[i][8] */
    }

c) Transformed code with sharing relocation buffer

    for (i=0;i<N;i++) {
      /* precollect A[i][0]..A[i][8] to A'[0]..A'[8] */
      for (j=0;j<8;j++)
        A'[j+1] = A'[j+1] + A'[j];
      /* distribute A'[1]..A'[8] to A[i][1]..A[i][8] */
    }
```

Figure 4.7 Example of Shared Relocation Buffer

problem may arise between **A'** and **A"** as shown in Figure 4.7b. Sharing the buffer eliminates the possible inconsistency problem.

Another relationship between array references is *in line cross iteration*. This relationship exists if the array subscripts containing the inner loop index variable (j in Figure 4.8a) are the same and if the difference in the array subscripts containing the index of the outer-loop variable (i in Figure 4.8a) is exactly one (Figure 4.8a). The iteration space that was updated by the previous *i-1* iteration is referenced at the current i iteration. In this case, the relocation buffer can be forwarded across the iteration so

```
a) Original code

    for (i=0;i<N;i++)
      for (j=0;j<8;j++)
        A[i+1][j] = A[i+1][j] + A[i][j];

b) Transformed code without forwarding relocation buffer

    for (i=0;i<N;i++) {
      /* precollect A[i][0]..A[i][7] to A'[0]..A'[7] */
      /* precollect A[i+1][0]..A[i+1][7] to A''[0]..A''[7] */
      for (j=0;j<8;j++)
        A''[j] = A''[j] + A'[j];
      /* distribute A''[0]..A''[7] to A[i+1][0]..A[i+1][7] */
    }

c) Transformed code with forwarding relocation buffer

    /* precollect A[0][0]..A[0][7] to A'[0]..A'[7] */
    for (i=0;i<N;i++) {
      /* precollect A[i+1][0]..A[i+1][7] to A''[0]..A''[7] */
      for (j=0;j<8;j++)
        A''[j] = A''[j] + A'[j];
      /* distribute A''[0]..A''[7] to A[i+1][0]..A[i+1][7] */
      /* forward A'' to A' */
    }
```

Figure 4.8 Example of Forwarded Relocation Buffer

that the unnecessary *precollect* operation can be avoided (Figure 4.8c). If the *precollect* operation is overlapped with the computation by software pipelining, forwarding buffer is essential since the *precollect* operation must be invoked before the computation of the previous iteration is completed. In other words, the elements have not been updated by the time the *precollect* operation is invoked. Therefore, if the dependence distance regarding i loop is not determined by the dependence analyzer, the array reference should not be transformed using the *DRP* technique to avoid the possibility of incorrect execution.

Some intelligent schemes are used to reduce the *DRP* operation overhead. As described in Section 4.2.3, *preallocate* operations are used if the array analysis determines that the array data is not read before being written during the computation (array **A** in Figure 4.2). In this case, there is no need to fetch the data into the cache before the computation since the data is updated by the computation anyway. The overhead of the *preallocate* operation is much smaller than the *precollect* operation since it only sets the tag field of the cache rather than fetching data.

A write buffer is often attached to the cache in order to reduce the processor stalls due to cache write misses. Often, a cache line is not allocated even if the cache misses for a write access. This policy is called no-write-allocate. If the cache has a no-write-allocate policy, the cache write miss does not stall the processor unless the write buffer is full, which is a rare case. In other words, these array references may not benefit from the *DRP* technique. Therefore, if an array is written and not read thereafter in a innermost loop, it is more effective for these arrays not to be transformed by *DRP* in order to reduce the *DRP* operation overhead which is shown in Figure 4.9. In the case of array **B**, the += operator is devided into two operators, = and +, so that only the read access can be transformed for *DRP*.

After grouping the array references and determining the characteristics of the array grouped, the information for the operands of *DRP* operations is determined. The starting address of array to be relocated is determined using the loop analysis information described in Section 4.4.1 since it needs the index of array reference for the next outer loop iteration. The element size of an array is determined by information gathered from

a) Original code

```
for (i=0;i<N;i++)
  for (j=0;j<8;j++) {
    A[i][j] = 1.0;
    B[i][j] += 2.0;
  }
```

b) Transformed code by DRP technique for all accesses

```
for (i=0;i<N;i++) {
  /* preallocate A'[0]..A'[7] */
  /* precollect B[i][0]..B[i][7] to B'[0]..B'[7] */
  for (j=0;j<8;j++) {
    A'[j] = 1.0;
    B'[j] += 2.0;
  }
  /* distribute A'[0]..A'[7] to A[i][0]..A[i][7] */
  /* distribute B'[0]..B'[7] to B[i][0]..B[i][7] */
}
```

c) Transformed code by DRP technique for all accesses except last stores

```
for (i=0;i<N;i++) {
  /* precollect B[i][0]..B[i][7] to B'[0]..B'[7] */
  for (j=0;j<8;j++) {
    A[i][j] = 1.0;
    B[i][j] = B'[j] + 2.0;
  }
}
```

Figure 4.9 Example of Application of *DRP* to All Array Accesses Except Last Stores

the Pcode representation and machine description. The stride of the array references is determined by the difference between two array reference addresses, which may not be able to be completely determined at compile time. Therefore, the stride is an expression such as **&A[i][1] - &A[i][0]**. This expression may be reduced into a constant number by the low-level optimizations. The number of elements is determined by the loop information such as the initial value, final value and increment of the inner-most loop variable. Like the stride, the number of elements may not be able to be determined at compile time. For example, it is impossible to determine the number of iterations for a blocked loop at compile time since the final trip count value for a blocked loop is usually expressed by a function which checks the loop boundary at run time. However, if some transformation such as loop peeling is used for the blocked loop as described in Section 4.4.3.6, the function to check the loop boundary can be removed from the loop nest. In this case, the number of elements may be reduced to a constant by an intelligent optimization in low-level code.

4.4.3 High-level Code Transformation

The data relocation and prefetching optimization can be applied most directly to loops nested at least two deep. Figure 4.11a shows an example loop nest that illustrates different array access patterns. Before each invocation of the inner-most loop, array data accessed in the inner-most loop are first relocated and prefetched, after which the computation proceeds until the inner-most loop is finished. The transformation is always applied for the two inner-most loops for loop nests which are nested more deeply than

two. For this case, I refer to the outer of the two inner-most loops as the outer loop, and the inner-most loop as the inner loop. The high-level code transformations employed for data relocation and prefetching are declaration of new variables for *DRP*, loop strip-mining, insertion of *DRP* operations, replacement of array references with relocation buffer references, loop unrolling and loop peeling. Each of these transformations are discussed in greater detail in the following subsections.

4.4.3.1 Declaration of New Variables for *DRP*

First, as shown in Figure 4.10, the relocation buffer is declared as an array that has the same size as the cache size. This guarantees that the relocated arrays in the relocation buffer have no cache line conflicts among them. Then, new array variables to access the relocation buffer are declared as pointers. The starting address of each relocated array in the relocation buffer is assigned to the associated variable. In my implementation, the new array variables in Figure 4.11 are assigned in the relocation buffer as shown in Figure 4.10. Actually, cache is subdivided to a very high degree to accommodate widely varying array data sizes. Thus, the same pointer variables need not be reassigned even if the the number of arrays are different for each loop nest, which eliminates the overhead of pointer variable assignments of new variables for each loop nest.

In Figure 4.10, $\mathbf{A}'$, $\mathbf{B}'$, and $\mathbf{C}'$ are pointers to the relocation buffer starting addresses for the elements of arrays **A**, **B**, and **C** (respectively) accessed in the first unrolled outer-loop body, and $\mathbf{A}''$, $\mathbf{B}''$, and $\mathbf{C}''$ point to the relocation buffer starting addresses for array

elements accessed in the second unrolled outer-loop body. Loop unrolling is described in Section 4.4.3.5.

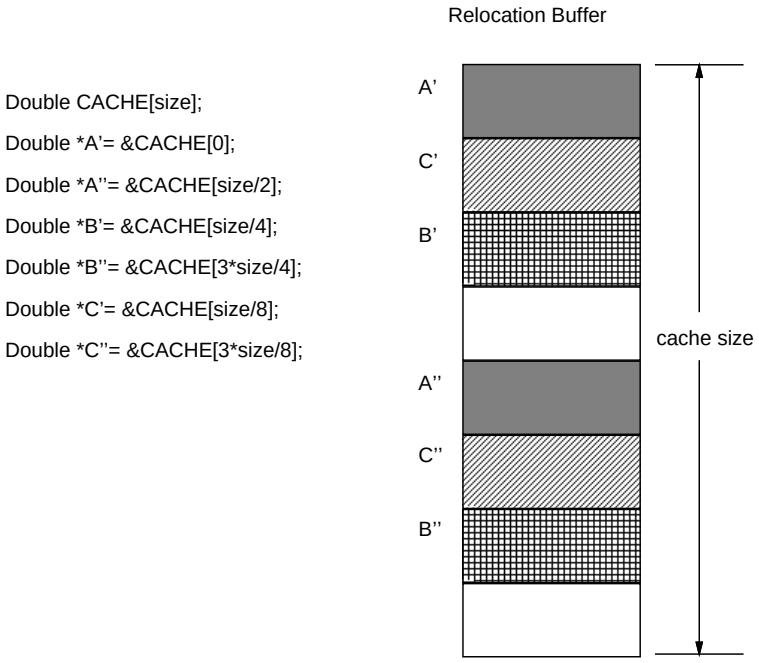

Figure 4.10 Declaration of New Variables for *DRP*

4.4.3.2 Loop Strip-mining

For the *DRP* technique, strip-mining of the inner loop is performed in order to create a doubly nested loop if the loop nest consists of only a singly nested loop. Strip-mining is also used in order to reduce the amount of data relocated for the inner-loop computation if that amount is too large to fit in the cache. This is necessary since all elements that are relocated and prefetched for the inner loop computation must be allocated to unique cache locations to prevent possible conflict misses. at the same time. Furthermore, strip-mining can remove some restrictions that prevent the loop nest from being transformed for the *DRP* technique. For example, if the original outer loop includes calls with possi-

a) Original Loop Nest

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++) {
    C[i][j] = A[i][2*j] + B[j][i];
    D[i][j] = C[i][j] - B[j+1][i];
  }
```

b) After Strip Mining Inner Loop

```
for (i=0; i<N; i++)
  for (jj=0; jj<N; jj+Bj)
    min_j = min(jj+Bj, N);
    for (j=jj; j<min_j; j++) {
      C[i][j] = A[i][2*j] + B[j][i];
      D[i][j] = C[i][j] - B[j+1][i];
    }
```

Figure 4.11 Strip-mining Transformation

ble side-effect, the restrictions due to the outer loop can be ignored by strip-mining the innermost loop and applying the *DRP* technique to the two innermost loops. As shown in Figure 4.11, loop strip-mining does not alter the data access pattern of the original loop and is, therefore, always a valid loop transformation. However, strip-mining introduces the *min* function that is used to check the iteration boundary. The *min* function is actually implemented as *if* statements, thus it introducing unnecessary execution overhead. These *if* statements can be eliminated by *loop peeling*, described in Section 4.4.3.6.

4.4.3.3 Insertion of *DRP* Operations

After creating a doubly-nested loop by strip-mining, new operations are inserted in the high-level code to perform the *precollect*, *preallocate*, *await*, *distribute*, and *finishup* operations. (See Figure 4.12.) These high-level operations are replaced by the corresponding machine instructions at the assembly code level.

Before the outer loop, *precollect* and/or *preallocate* operations are inserted for the first inner loop computation. Since these operations cannot be overlapped with any other computation in the loop body, these operations constitute the start-up overhead

a) Strip-Mined Loop

```
for (jj=0; jj<N; jj+Bj)
  min_j = min(jj+Bj, N);
  for (j=jj; j<min_j; j++) {
    C[i][j] = A[i][2*j] + B[j][i];
    D[i][j] = C[i][j] - B[j+1][i];
  }
```

b) After Inserting New Operations

```
precollect(&A[i][0], A', 8, 16, Bj);
precollect(&B[0][i], B', 8, Bd, Bj+1);
preallocate(C', 8, Bj);
for (jj=0; jj<N; jj+=Bj)
  precollect(&A[i][jj+Bj], A'', 8, 16, Bj);
  precollect(&B[jj+Bj][i], B'', 8, Bd, Bj+1);
  preallocate(C'', 8, Bj);
  await(A'); await(B'); await(C');
  min_j = min(jj+Bj, N);
  for (j=jj; j<min_j; j++) {
    C[i][j] = A[i][2*j] + B[j][i];
    D[i][j] = C[i][j] - B[j+1][i];
  }
  distribute(&C[i][jj], C', 8, 8, min_j-jj);
}
finishup();
```

Figure 4.12 Insertion of Operation for *DRP*

of software pipelining. One or more *await* operations are inserted just before the inner loop to insure that each of the *precollect* and *preallocate* operations is completed before the inner-loop array computation begins. *Distribute* operations are inserted just after the inner loop in order to restore all modified data in the relocation buffers to their original locations in the memory. Also, *precollect* and/or *preallocate* operations for the next outer-loop iteration are inserted before the inner loop. These operations need to use different buffers from those of the current iteration since precollecting the data for the next iteration is overlapped with the computation for the current iteration by software pipelining. Finally, a *finishup* operation is placed just after the outer loop in order to insure that all *distribute* operations are completed before execution proceeds to other computations which may involve the same array data.

a) Before Replacing Array References

```
precollect(&A[i][0],A',8,16,Bj);
precollect(&B[0][i],B',8,Bd,Bj+1);
preallocate(C',8,Bj);
for (jj=0; jj<N; jj+=Bj)
  precollect(&A[i][jj+Bj],A'',8,16,Bj);
  precollect(&B[jj+Bj][i],B'',8,Bd,Bj+1);
  preallocate(C'',8,Bj);
  await(A'); await(B'); await(C');
  min_j = min(jj+Bj,N);
  for (j=jj; j<min_j; j++) {
    C[i][j] = A[i][2*j] + B[j][i];
    D[i][j] = C[i][j] - B[j+1][i];
  }
  distribute(&C[i][jj],C',8,8,min_j-jj);
}
finishup();
```

b) After Replacing Array References

```
precollect(&A[i][0],A',8,16,Bj);
precollect(&B[0][i],B',8,Bd,Bj+1);
preallocate(C',8,Bj);
for (jj=0; jj<N; jj+=Bj)
  precollect(&A[i][jj+Bj],A'',8,16,Bj);
  precollect(&B[jj+Bj][i],B'',8,Bd,Bj+1);
  preallocate(C'',8,Bj);
  await(A'); await(B'); await(C');
  min_j = min(jj+Bj,N);
  for (j=jj; j<min_j; j++) {
    C'[j] = A'[j] + B'[j];
    D[i][j] = C'[j] - B'[j+1];
  }
  distribute(&C[i][jj],C',8,8,min_j-jj);
}
finishup();
```

Figure 4.13 Replacement of Array References for *DRP*

4.4.3.4 Replacement of Array References with Relocation Buffer References

Once the *DRP* operations have been inserted, the array references for the computation within the inner loop are modified so that the relocation buffer locations are accessed instead of the original array locations. Some array references that do not need to be relocated are left as the original array references. Replaced array references are all one-dimensional even if the original array references are multi-dimensional.

4.4.3.5 Loop Unrolling

In order to overlap the data relocation and prefetching for the next outer-loop iteration with the computation for the current iteration, the relocation and prefetching phase is software-pipelined with the computation phase as shown in Figure 4.14. This software-pipelining scheme requires two relocation buffers. The inner loop is duplicated

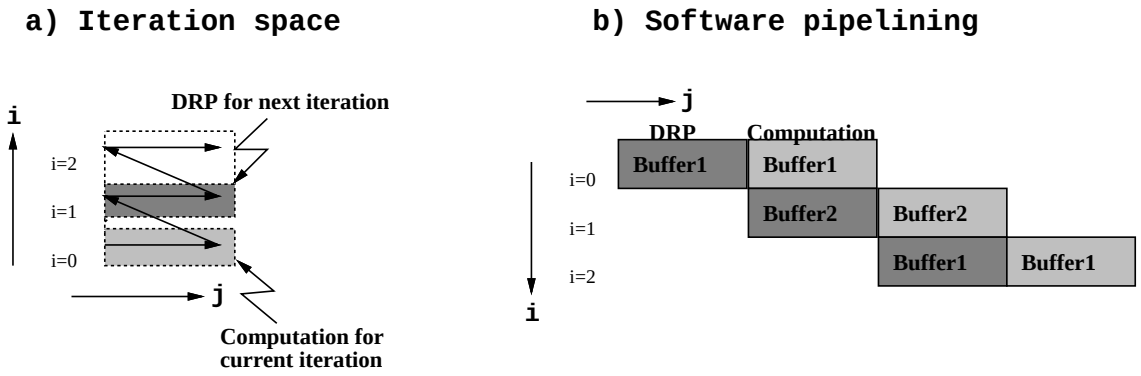

Figure 4.14 Software Pipelining for *DRP*

by unrolling the outer loop once, as shown in Figure 4.15. In the first outer-loop body, the data relocation proceeds into the second relocation buffer, while the computation is performed using the data already relocated in the first buffer. For the second outer-loop body, the same method is used as for the first outer-loop body except that the buffers are switched. Loop unrolling introduces outer-loop code expansion and *if* statements that check the iteration boundary for the second body.

4.4.3.6 Loop Peeling

The final transformation is loop peeling. This transformation peels the last iteration of the outer loop in order to reduce the overhead of the *if* statements that have been introduced by the strip-mining and loop unrolling as described in previous sections. Loop peeling guarantees that the loop body is executed a certain number of times without checking the iteration boundary, eliminating need for the *if* statements in the loop. Although the peeled body still needs the *if* statements, the execution percentage for the

a) Before Unrolling

```
for (jj=0; jj<N; jj+=Bj)
  precollect(&A[i][jj+Bj],A'',8,16,Bj);
  precollect(&B[jj+Bj][i],B'',8,Bd,Bj+1);
  preallocate(C'',8,Bj);
  await(A'); await(B'); await(C');
  min_j = min(jj+Bj,N);
  for (j=jj; j<min_j; j++) {
    C'[j] = A'[j] + B'[j];
    D[i][j] = C'[j] - B'[j+1];
  }
  distribute(&C[i][jj],C',8,8,min_j-jj);
}
finishup();
```

b) After Unrolling

```
for (jj'=0; jj'<N; jj'+=2*Bj)
  jj=jj';
  precollect(&A[i][jj+Bj],A'',8,16,Bj);
  precollect(&B[jj+Bj][i],B'',8,Bd,Bj+1);
  preallocate(C'',8,Bj);
  await(A'); await(B'); await(C');
  min_j = min(jj+Bj,N);
  for (j=jj; j<min_j; j++) {
    C'[j] = A'[j] + B'[j];
    D[i][j] = C'[j] - B'[j+1];
  }
  distribute(&C[i][jj],C',8,8,min_j-jj);
  jj+=Bj;
  if (jj<N) {
    precollect(&A[i][jj+Bj],A',8,16,Bj);
    precollect(&B[jj+Bj][i],B',8,Bd,Bj+1);
    preallocate(C',8,Bj);
    await(A''); await(B''); await(C'');
    min_j = min(jj+Bj,N);
    for (j=jj; j<min_j; j++) {
      C''[j] = A''[j] + B''[j];
      D[i][j] = C''[j] - B''[j+1];
    }
    distribute(&C[i][jj],C'',8,8,min_j-jj);
  }
}
finishup();
```

Figure 4.15 Loop Unrolling Transformation for *DRP*

peeled body is generally much smaller than that for the loop body. However, like loop unrolling, loop peeling introduces code expansion (Figure 4.16).

4.4.4 Low Level Code Transformations

After the high-level code transformations have been performed, the high-level code is translated into the low-level code. Then, the low-level code is also transformed by the *DRP* technique. The low-level code transformations employed for *DRP* are replacements of high-level *DRP* operations with low-level instructions and creation of flow dependence among *DRP* instructions in order to prevent invalid code schedules.

4.4.4.1 Replacement of High-level *DRP* Operations with Low-level Instructions

The transformed high-level code (*Pcode*) is translated into low-level code (*Lcode*), which is a machine-level intermediate code in order to optimize and schedule the code at the machine instruction level. Since the high-level *DRP* operations are still in the form of function calls, they need to be translated to low-level instructions. Since *prec-ollect* and *distribute* operations for the *DRP* technique have five operands including at most four register operands, neither operation is likely to be implemented by one 32-bit machine instruction. Therefore, each of these high-level operation is replaced with two low-level instructions. The other *DRP* operations can be replaced with a single machine instruction.

a) Before Loop Peeling

```
for (jj'=0; jj'<N; jj'+=2*Bj)
  jj=jj';
  precollect(&A[i][jj+Bj],A'',8,16,Bj);
  precollect(&B[jj+Bj][i],B'',8,Bd,Bj+1);
  preallocate(C'',8,Bj);
  await(A'); await(B'); await(C');
  min_j = min(jj+Bj,N);
  for (j=jj; j<min_j; j++) {
    C'[j] = A'[j] + B'[j];
    D[i][j] = C'[j] - B'[j+1];
  }
  distribute(&C[i][jj],C',8,8,min_j-jj);
  jj+=Bj;
  if (jj<N) {
    precollect(&A[i][jj+Bj],A',8,16,Bj);
    precollect(&B[jj+Bj][i],B',8,Bd,Bj+1);
    preallocate(C',8,Bj);
    await(A''); await(B''); await(C'');
    min_j = min(jj+Bj,N);
    for (j=jj; j<min_j; j++) {
      C''[j] = A''[j] + B''[j];
      D[i][j] = C''[j] - B''[j+1];
    }
    distribute(&C[i][jj],C'',8,8,min_j-jj);
  }
}
finishup();
```

b) After Loop Peeling

```
for (jj'=0; jj'<N-2*Bj; jj'+=2*Bj)
  jj=jj';
  precollect(&A[i][jj+Bj],A'',8,16,Bj);
  precollect(&B[jj+Bj][i],B'',8,Bd,Bj+1);
  preallocate(C'',8,Bj);
  await(A'); await(B'); await(C');
  for (j=jj; j<jj+Bj; j++) {
    C'[j] = A'[j] + B'[j];
    D[i][j] = C'[j] - B'[j+1];
  }
  distribute(&C[i][jj],C',8,8,Bj);
  jj+=Bj;
  precollect(&A[i][jj+Bj],A',8,16,Bj);
  precollect(&B[jj+Bj][i],B',8,Bd,Bj+1);
  preallocate(C',8,Bj);
  await(A''); await(B''); await(C'');
  for (j=jj; j<jj+Bj; j++) {
    C''[j] = A''[j] + B''[j];
    D[i][j] = C''[j] - B''[j+1];
  }
  distribute(&C[i][jj],C'',8,8,Bj);
}
jj=jj';
if (jj < N) {
  precollect(&A[i][jj+Bj],A'',8,16,Bj);
  precollect(&B[jj+Bj][i],B'',8,Bd,Bj+1);
  preallocate(C'',8,Bj);
  await(A'); await(B'); await(C');
  min_j = min(jj+Bj,N);
  for (j=jj; j<min_j; j++) {
    C'[j] = A'[j] + B'[j];
    D[i][j] = C'[j] - B'[j+1];
  }
  distribute(&C[i][jj],C',8,8,min_j-jj));
  jj+=Bj;
  if (jj < N) {
    await(A''); await(B''); await(C'');
    min_j = min(jj+Bj,N);
    for (j=jj; j<min_j; j++) {
      C''[j] = A''[j] + B''[j];
      D[i][j] = C''[j] - B''[j+1];
    }
    distribute(&C[i][jj],C'',8,8,min_j-jj));
  }
}
finishup();
```

Figure 4.16 Loop Peeling Transformation for *DRP*

Figure 4.17 shows the six machine level instructions that are added to the instruction set for the *DRP* technique. Five instructions respectively correspond to the five high-level operations that are described in Section 4.2. One new machine-level instruction is *Lop_setup*. This instruction is used with the *Lop_precollect* and *Lop_distribute* instructions in order to implement the *precollect* and *distribute* operations, respectively. *Lop_setup* takes the element size and the access stride of an array as its two operands, and encodes them into a register. A *Lop_setup* instruction is inserted just before the corresponding *Lop_precollect* instruction. The *Lop_distribute* instruction can use the register that is generated by the previous *Lop_setup* instructions for *Lop_precollect* instruction. Although the original *precollect* operations are inserted in the outer loop body, the *Lop_setup* instruction may be moved out of the outer loop by the Lcode optimizer since the two operands are likely to be loop-independent. Therefore, the overhead of the additional *Lop_setup* instructions should be small.

The *Lop_precollect* and *Lop_distribute* instructions take four operands. The first operand is the relocation buffer address. The second operand is the array starting address to be relocated. The third operand is the register that holds the encoded data generated by the *Lop_setup* instruction. The last operand is the number of relocated array elements. This operand will be constant in the outer loop body. However, in the peeled body, the value for this operand is determined by the *min* function as described in Section 4.4.3.2. All operands except the last operand are read from general purpose registers. The last operand can be read either from a register or directly from the immediate value in the instruction.

a) Precollect Operation

```
Lop_setup [Rt] [(element_size)(stride)]
Lop_precollect [buffer_addr] [(array_addr)(Rt)(# of data)]
```

b) Distribute Operation

```
Lop_setup [Rt] [(element_size)(stride)]
Lop_distribute [buffer_addr] [(array_addr)(Rt)(# of data)]
```

c) Preallocate Operation

```
Lop_preallocate [buffer_addr] [(element_size)(# of data)]
```

d) Await Operation

```
Lop_await [buffer_addr]
```

e) Finishup Operation

```
Lop_finishup []
```

Figure 4.17 Machine Level Instructions for Each *DRP* Operation

4.4.4.2 Creation of Dependences among *DRP* Instructions

After replacing the high-level operations with low-level instructions, classical optimizations are applied. Then, *ILP* increasing optimizations such as loop unrolling and superblock formation [18] are also performed. Next, the code is scheduled, register allocated, and optimized for a specific machine. Before performing the instruction-level scheduling, data dependences among the *DRP* Instructions are created in order to prevent the scheduler from producing an invalid schedule. The dependences are created by the register that holds the relocation buffer address as shown in Figure 4.18. Each instruction takes a set of source operands and one destination operand. The first square bracket of each instruction represents the destination field, and the second square bracket represents the source field. The destination field has only one operand while the source field has multiple operands, which is represented by parenthesis. The number of the source operands depends on the instruction.

In the Figure 4.18, *R1* represent a register that hold the relocation buffer address. This register is used to create flow dependence chain among the *DRP* instructions so that these instructions can be scheduled in the same order as the high-level operations inserted for *DRP*. Some instructions in the Figure 4.18 takes the relocation buffer address operand as both the source and the destination. This is done just for the scheduling purpose, therefore one operand for the buffer address is enough for the actual machine instruction.

```
Lop_ld [R1] [(&CACHE)(0)]
 ...
Lop_setup [Rt] [(element_size)(stride)]
 ...
Lop_precollect [R1] [(R1)(array_addr)(Rt)(# of data)]
 ...
Lop_await [R1] [R1]
for (i=0; i<N; i++) {
   ...
  Lop_ld [Rn][(R1)(offset)]        ; Rn <- Mem(R1+offset)
   ...
  Lop_st [Rm][(R1)(offset)]        ; Mem(R1+offset) <- Rm
   ...
}
Lop_distribute [R1] [(array_addr)(Rt)(# of data)]
```

Figure 4.18 Creation of Dependences among *DRP* Instructions

Since the *store* instruction has no destination register, an artificial dependence arc is added from the *store* instruction to the *Lop_distribute* instruction before scheduling the code.

CHAPTER 5

EXPERIMENTAL EVALUATION

In this chapter, the effectiveness of the data relocation and prefetching technique is evaluated through simulations of a set of array-based benchmarks.

5.1 Methodology

5.1.1 Compiler Transformation Statistics

The benchmarks used for the experimental evaluation of the *DRP* technique consist of fourteen numeric programs: **ADM**, **SPICE**, **MGD**, **TRACK**, **BDNA**, **OCEAN**, **DYFESM**, **ARC2D**, **FLO52**, **TRFD**, and **SPEC77** from the *PERFECT* benchmark suite citeBeCK89; **MATRIX300** from the *SPEC'89* benchmark suite; and **NASA7** and **TOMCATV** from the *SPEC'92* benchmark suite. All benchmarks are profiled at the loop level to obtain the number of loop invocations and iterations in order to apply the *DRP* transformations selectively and effectively. Tables 5.1 - 5.3 summarize the *DRP* compiler transformation statistics for each of the benchmarks.

Table 5.1 shows the inner-most loops that were excluded from consideration based on static analysis of the loop characteristics. The second column of the table gives the total number of inner-most loops for each benchmark, while the third through the fifth

Benchmark Name	# of Loop Nests	Excluded Loops			Remaining Loops					
		Side Effect Calls	Goto Stmts	Return Stmts	Depth					Total
					1	2	3	4	5	
ADM	188	29	8	0	54	63	38	0	0	155
SPICE	310	223	0	62	40	10	1	0	0	51
MGD	35	4	0	3	10	9	9	1	0	29
TRACK	69	16	1	8	22	18	6	3	0	49
BDNA	212	56	0	7	84	58	6	2	0	150
OCEAN	79	8	0	0	22	37	12	0	0	71
DYFESM	165	57	1	2	34	43	22	7	0	106
ARC2D	150	7	0	1	68	56	18	0	0	142
FLO52	118	22	0	10	20	43	24	0	0	87
TRFD	32	3	0	0	1	3	6	13	6	29
SPEC77	279	41	0	19	62	118	37	4	0	221
MATRIX300	15	11	0	0	2	2	0	0	0	4
NASA7	73	13	0	0	14	28	13	5	0	60
TOMCATV	12	0	0	1	6	5	0	0	0	11

Table 5.1 Inner-most Loops Excluded by Static Loop Characteristics

columns give the numbers of loops excluded because of subroutine calls which possibly modify global arrays, "goto" statements which causes a branch out of the loop, and "return" statements. A single inner-most loop may be represented in the counts for any or all of these three columns. The number and loop nesting depths of the inner-most loops that remain after the loops are excluded are given in the remaining columns of the table.

After static loop-characteristic analysis has been applied, static array characteristics and dynamic loop iteration counts are used to exclude more loops from being considered for *DRP* transformation. Table 5.2 summarizes the array reference characteristics which prevent certain references from being relocated as well as loops that were excluded due to small loop iteration counts. The second column of the table gives the number of inner-most loops which have not been excluded by the loop characteristics analysis phase. The

Benchmark Name	# of Innermost Loops Before	# of Innermost Loops							# of Innermost Loops After
		No Array in Loop	Write Access	Modified Iteration Variable	No Inner Iteration Variable	Complex Array Subscript	Flow Depend	Small Iter.	
ADM	155	6	110	26	24	0	19	85	14
SPICE	51	0	30	3	21	0	4	25	1
MGD	29	3	9	11	10	0	7	11	3
TRACK	49	0	29	0	8	2	1	31	5
BDNA	150	7	63	25	43	33	7	63	13
OCEAN	71	1	19	0	4	5	10	1	40
DYFESM	106	17	48	2	17	7	6	51	2
ARC2D	142	2	48	0	2	0	4	34	89
FLO52	87	4	31	9	8	0	1	16	40
TRFD	29	6	11	7	6	0	4	8	1
SPEC77	221	5	91	19	52	4	7	91	57
MATRIX300	4	0	3	0	0	0	0	0	1
NASA7	60	0	24	2	10	0	2	4	30
TOMCATV	11	0	1	0	0	0	0	1	9

Table 5.2 Results of Static Array Reference and Dynamic Loop Iteration Count Analyses

third column gives the number of inner-most loops which have no array accesses within them. These loops are automatically excluded from consideration. Of the remaining inner-most loops, the next five columns give the number of loops which contain specific kinds of array reference characteristics which prevent these arrays from being relocated, although other arrays in the loop may still benefit from relocation. The second-from-the-last column shows the number of inner-most loops which have too few iterations to overcome the transformation overhead; either without applying strip-mining, or after applying loop strip-mining if it is necessary. Finally, the last column relates the number of inner-most loop nests which remain viable for *DRP* transformation after this phase of the analysis.

There exist several reasons why individual array references are not relocated by the *DRP* transformation. First, if the last access to an array in the inner-most loop is a write, this access is not relocated in order to eliminate the overhead of adding the *DRP*

operations. Second, if the iteration variable of either the inner-most or next-outer loop is modified in the next-outer loop body and occurs in a subscript of an array reference, transforming this reference could be invalid. However, if the modified iteration variable is associated with the next-outer loop, then strip-mining the inner-most loop can sometimes remove this restriction. The number of loops which contain array reference subscripts with modified iteration variables which cannot be rendered harmless by strip-mining is given in Table 5.2. Third, if the inner-most loop iteration variable does not occur in any of the subscripts of an array reference, then the reference is a scalar with respect to the inner-most loop and is not relocated. Fourth, if any array reference subscript is a non-linear function of the iteration variables or contains both the inner-most and next-outer loop iteration variables, then that reference cannot be relocated using our *DRP* implementation. Last, if a reference has a undetermined distance cross-iteration flow dependence carried by the next-outer loop, relocating it could cause an inconsistency problem with the relocation buffers. Again, strip-mining the inner-most loop can sometimes remove this restriction, and the data given in Table 5.2 reflects those restrictions which could not be removed by loop strip-mining. With the exception of the last-write-access restriction, if one array reference is excluded from relocation consideration then all other references to the same array in the inner-most loop are also excluded. If all array references in the inner-most loop are excluded from relocation consideration, the loop will not be transformed by *DRP*.

Table 5.3 shows the number of inner-most loops which are transformed as well as statistics for the transformed loops based on profile information. The second column

Benchmark Name	Remaining Innermost Loops	Small Array Access & Misses	Transformed Innermost Loops	% of Total Array Accesses	Estimated Cache Misses		
					Original	DRP	Improve
ADM	14	10	4	3.8	9.0	8.2	8.9%
SPICE	1	1	0	0	7.9	7.9	0%
MGD	3	3	0	0	2.1	2.1	0%
TRACK	5	2	3	0.4	11.1	10.9	1.8%
BDNA	13	9	4	63.2	26.7	13.8	48.3%
OCEAN	40	27	13	23.0	36.9	26.0	29.5%
DYFESM	2	2	0	0	5.7	5.7	0%
ARC2D	89	52	37	87.6	40.8	2.2	94.6%
FLO52	40	37	3	1.6	11.9	11.5	3.4%
TRFD	1	0	1	0.3	17.3	17.1	1.2%
SPEC77	57	54	3	0.3	12.1	11.8	2.5%
MATRIX300	1	0	1	100.0	48.2	0.0	100%
NASA7	30	12	18	60.6	42.8	9.3	78.3%
TOMCATV	9	5	4	95.8	34.0	2.0	94.1%

Table 5.3 Results of Effectiveness Analyses for *DRP*-transformed Loops

of the table gives the number of inner-most loops which have not been excluded by the analysis summarized in Table 5.2. Of those inner-most loops, column three shows the number of loops which possess either small cache miss ratios or a relatively tiny number of dynamic array accesses as compared to the entire benchmark. If an inner-most loop has a small cache miss ratio, the performance benefit of transforming it may not outweigh the transformation overheads. Also, if the loop has a very small percentage of the dynamic array references for the benchmark, then the added code expansion due to transformation may cause instruction cache performance degradation which could outweigh the performance benefits of transforming it. The fourth column gives the number of inner-most loops which are actually transformed by the *DRP* technique.

The remaining columns of Table 5.3 relate information which predicts the effectiveness of the *DRP* transformation. The fifth column of the table shows the number of dynamic relocated array accesses as a percentage of the array accesses for the entire benchmark. Notice that only seven benchmarks have transformed array accesses per-

centages greater than two percent. These are the benchmarks used in Section 5.2 for the experimental evaluation. The other benchmarks have characteristics which are unsuitable for the *DRP* transformation. The last three columns give information about the data cache miss ratio for the benchmarks derived from Pcode cache simulation results. The first of these columns gives the original measured cache miss ratio while the second gives the cache miss ratio estimated by reducing to zero the number of cache misses in all transformed loops. The estimated percentage improvement for the cache miss ratio due to the *DRP* technique is given in the last column. Among the fourteen benchmarks, seven benchmarks have the improvement of the estimated cache misses: **ADM**, **BDNA**, **OCEAN**, **ARC2D**, **MATRIX300**, **NASA7**, and **TOMCATV**. These seven benchmark were used for the detailed simulation.

5.1.2 Transformation Correctness Verification via Emulation

In order to provide a realistic evaluation of the *DRP* technique, the code is optimized using the *IMPACT* compiler. Classical optimizations are applied, then *ILP* increasing optimizations such as loop unrolling and superblock formation [18] are performed. The code is scheduled, register allocated, and optimized for a four-issue, scoreboarded, superscalar processor with register renaming. The register file contains 64 integer registers and 64 double-precision floating-point registers. Each of the four functional units are pipelined and can execute any type of instruction.

To verify the correctness of the code transformations, emulation of the generated code is performed on a Hewlett-Packard PA-RISC 7100 workstation. The *precollect* and

Function	Latency	Function	Latency
Int ALU	1	FP ALU	2
memory load	2	FP multiply	2
memory store	1	FP divide (SGL)	8
branch	1 / 1 slot	FP divide (DBL)	15

Table 5.4 Instruction Latencies for Simulation Experiments

distribute instructions are emulated using machine language subroutines that perform the data relocation from memory to memory instead of to and from the cache. Thus, the transformed code must relocate the data and reference it using the correct addresses for the emulation to produce valid results.

5.1.3 Simulation Experiments

The emulator drives a simulator that models the processor and the *DRP* unit to determine benchmark execution time, cache performance, and bus utilization. The simulation latencies used are those of a Hewlett-Packard PA-RISC 7100 microprocessor, as given in Table 5.4.

The processor model includes separate instruction and data caches. Each cache is a direct-mapped, 8k-byte blocking cache with a 16-byte block size. The data cache is a multiported, write-back, no write-allocate cache that satisfies four load or store requests per cycle from the processor. The 8-entry write buffer combines write requests to the same cache line. Streaming of data from load misses minimizes the load miss penalty. The instruction cache and data cache share a common, split-transaction memory bus, with 64 bits/cycle data bandwidth. A pipelined memory model is used with a 10-cycle latency.

A direct-mapped branch target buffer with 1024 entries is used to perform dynamic branch prediction using a 2-bit counter. Hardware speculation is supported, and the branch misprediction penalty is approximately two cycles.

The simulation model for the *DRP* unit is based on the description in Chapter 4. However, infinite-size queues and buffers are modeled so that no blocking occurs in the *DRP* unit due to insufficient entries. Also, the data cache has one read and one write port dedicated to service *DRP* unit accesses.

Since simulating the entire benchmark programs at this level of detail would be impractical, uniform sampling is used to reduce simulation time [19]. The samples are 200,000 instructions in length and are spaced evenly every 20,000,000 instructions, yielding a 1% sampling ratio. Most of the benchmarks used have more than a billion dynamic instructions, at least 50 samples, and thus, more than 10,000,000 instructions are simulated. For smaller benchmarks, the time between samples is reduced to maintain at least 50 samples (10,000,000 instructions). From experience with the emulation-driven simulator, it has been determined that sampling with at least 50 samples introduces very little error in our performance estimates. Typically, the statistics generated with sampling are within 1% of those generated without sampling.

5.2 Experimental Results

Experimental results are presented for some of the loop nests as well as the entire benchmarks. From the view of the point of memory optimization, three types of processor

stalls are evaluated for each experiment. The first one is processor stall due to the cache read misses. The second one is processor stall due to the full write buffer. The last one is processor stall due to the *await* operations as described in Section 4.2.4. The last type exists only for the *DRP* technique.

5.2.1 Individual Loop Nest Results

In order to show the full performance benefit of the *DRP* technique, experimental results are presented for all benchmark program loop nests which are modified by the *DRP* transformation. Performance statistics for individual loop nests are obtained by marking the *DRP*-transformed loop nests as execution regions for the simulation. Consequently, simulating the execution of the entire program once is sufficient to gather results for all transformed loop nests in the context of the entire program execution.

Table 5.5 shows some array information of some transformed loop nests that were frequently executed. The transformed loop nests are identified by function name, the *Fortran* outer **DO**-loop number, and the loop iteration variable if necessary. The second column is the original code execution time for each of the loop nests selected by the *DRP* technique as a percentage of the total original code execution time. For **BDNA**, **MATRIX300** and **TOMCATV**, a single loop nest has a very high percentage of the execution cycles. Transforming these loop nests will greatly affect the entire benchmark results. The entire benchmark results are discussed in Section 5.2.2. The third column of Table 5.5 shows the array access counts of each loop nest as a percentage of the total array accesses in the original code. This is almost proportional to the percentage of the

Benchmark Loop Nest	% of Total Execution	% of Total Array Acc	Cache Miss Ratio (% of Total Miss)	# of Arrays		# of Buff	Strip-mine Size	Estimated Outer Iter
				Orig	Trans			
ARC2D								
STEPFX.232	9.9%	8.7%	49.8% (10.6%)	17	17	10	10	8
STEPFY.435J	7.0%	8.7%	29.5% (6.3%)	17	17	10	17	17
XPENTA.11	5.5%	4.0%	72.8% (7.2%)	12	12	12	10	8
STEPFX.212	5.4%	4.6%	34.8% (3.9%)	9	9	9	10	8
STEPFY.430	5.4%	4.5%	50.1% (5.5%)	9	9	9	17	16
ADM								
LEAPFR.30	1.9%	2.4%	20.6% (5.4%)	3	3	3	28	27
DCTDX.10	0.7%	0.8%	25.1% (2.3%)	1	1	1	8	8
BDNA								
ACTFOR.350	65.7%	62.7%	20.4% (47.7%)	42	21	15	26	24
OCEAN								
IN.10	9.8%	10.3%	50.0% (14.0%)	1	1	1	50	50
OUT.10	5.9%	8.8%	44.7% (10.6%)	1	1	1	44	42
MATRIX300								
SAXPY.10	97.6%	97.6%	48.2% (100.0%)	2	2	2	18	16
NASA7								
CFFT2D2.30K	16.6%	10.1%	43.8% (10.3%)	8	8	4	12	10
GMTRY.8K	11.2%	12.0%	56.7% (15.9%)	3	2	2	19	17
CFFT2D1.130	9.9%	8.8%	25.0% (5.1%)	8	8	4	16	16
VPENTA.11	7.9%	5.5%	80.6% (10.3%)	18	18	18	12	10
VPENTA.15	5.6%	4.6%	80.0% (8.7%)	15	15	11	12	10
TOMCATV								
MAIN.250	42.3%	56.0%	28.6% (47.1%)	28	28	6	16	15
MAIN.401I	22.5%	15.9%	50.1% (23.5%)	8	8	8	16	15
MAIN.501I	15.7%	15.9%	25.2% (11.8%)	8	8	6	16	15
MAIN.290I	9.0%	8.0%	50.1% (11.8%)	4	4	4	16	15

Table 5.5 Array Information of Transformed Loop Nests

execution time in the second column. The forth column is the cache miss ratio due to array accesses in each loop nest and the percentage of the total cache misses of the entire benchmark. The number of the fifth and the sixth column show the number of arrays in original code and the number of arrays that are transformed respectively. The seventh column is the number of the relocation buffer for one iteration. The number in the eight column shows the strip-mining size, and the the number of the last column show the estimated iterations of the transformed outer loop.

Figure 5.1 presents the measured speedup of the *DRP* technique which is calculated by dividing the original loop nest execution time by the *DRP*-transformed loop nest

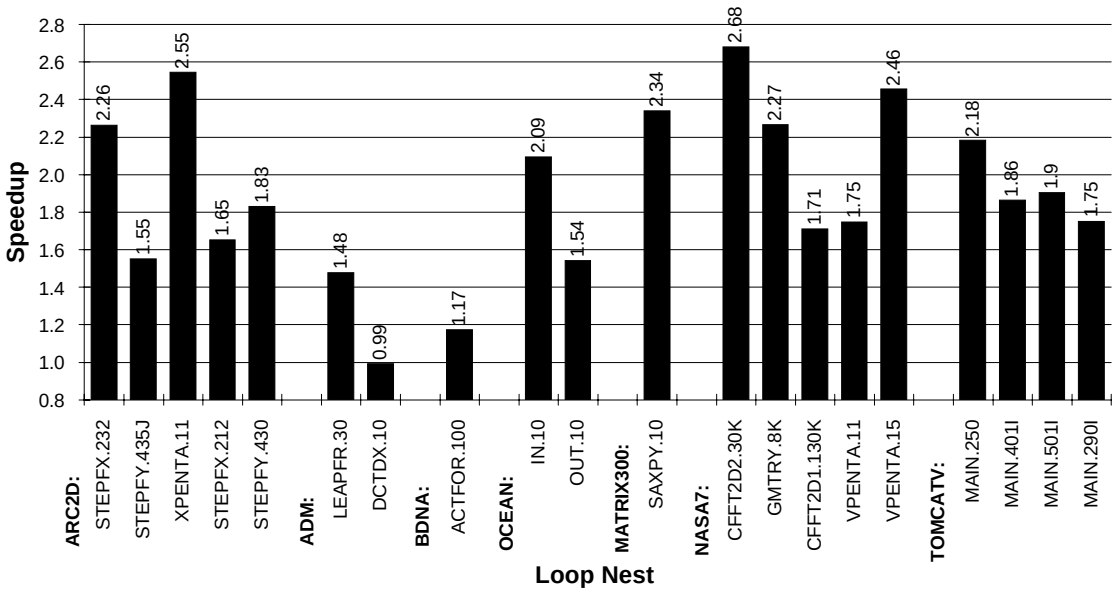

Figure 5.1 Speedup of the *DRP*-transformed Code Over the Original Code for the Loop Nests

execution time. Measured speedup for most loop nests is relatively large, demonstrating the high performance improvement obtainable using the *DRP* technique.

The stalls due to the cache read misses directly correspond to the cache read misses since each cache read miss stalls the processor. Figure 5.2 shows the data cache miss ratio for the original code and the *DRP*-transformed code for the loop nests. Since a no write-allocate cache is used for these experiments, the miss ratios for both original and transformed code are calculated by dividing the number of cache read misses by the number of cache read requests in the original code. This method of calculating the cache misses ensures a fair comparison if the number of cache accesses for the transformed code is different from the number for the original code. This ratio does not include requests

initiated by the *DRP* unit. Since these cache reads do not stall the *CPU*, they have a minimal impact on performance.

Note that cache misses are nearly eliminated for most of the loop nests. However, the execution speedup obtained for each loop nest is not always highly correlated to the reduction in cache miss ratio because of the complexity of scheduling a multiple-issue processor.

For the loop nest *ACTOFOR.100* in the **BDNA**, the cache misses were not completely eliminated by *DRP* technique. The loop nest contains forty two array references. However, twenty one of them were transformed for *DRP* and the rest of the array references were not transformed due the restrictions described in Chapter 4. Therefore, there still exist cache misses due to the non-transformed array references. The cache misses contains many conflict misses between the transformed and non-transformed array references. For the loop nest *STEPFX.212* in the **ARC2D**, some cache misses are not eliminated. The loop nest contains twenty scalar variables, which are not relocated, and they conflict the cache line with the transformed array references. The 98% of the cache misses are the conflict misses due to this reason.

Figure 5.3 shows the stall overhead due to the full write buffer. The overhead is represented as the percentage of execution cycles for which the processor is stalled due to the full write buffer for the transformed loop nests. The stalls due to the full write buffer are increased for *DRP*-transformed code. In general, for the cache with no-write-allocate policy, the write misses do not affect the processor perfomance so much since they do not stall the processor if the write buffer is not full. Without *DRP*, the bus

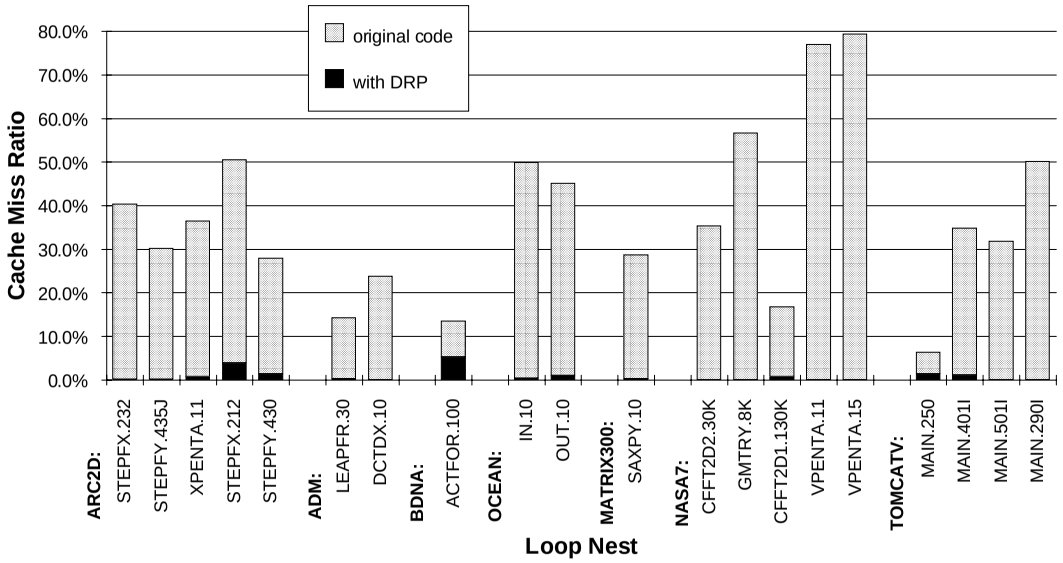

Figure 5.2 Cache Miss Ratios of the *DRP*-transformed Code and the Original Code for Loop Nests

utilization is low enough that the write buffer is unlikely to become full. In the case of *DRP*, however, since the write buffer has lower priority on the memory bus than the *DRP* unit, *precollect* operations will block the write buffer. Therefore, if there are many cache write misses in the transformed loop nest which contains many *precollect* operations, the write buffer easily becomes full, causing the processor stalls. As shown in Figure 5.4, *DRP* technique increases the cache write misses in the trasformed loop nests, resulting in the many processor stalls due to the full write buffer.

Figure 5.4 shows that the *DRP* technique increases the cache write miss ratio in most cases. There are two reasons that the cache write miss ratio increased. One reason is that since the *DRP* technique relocates arrays for read accesses, the original addresses of the arrays unlikely reside in the cache when the array is referenced by write accesses

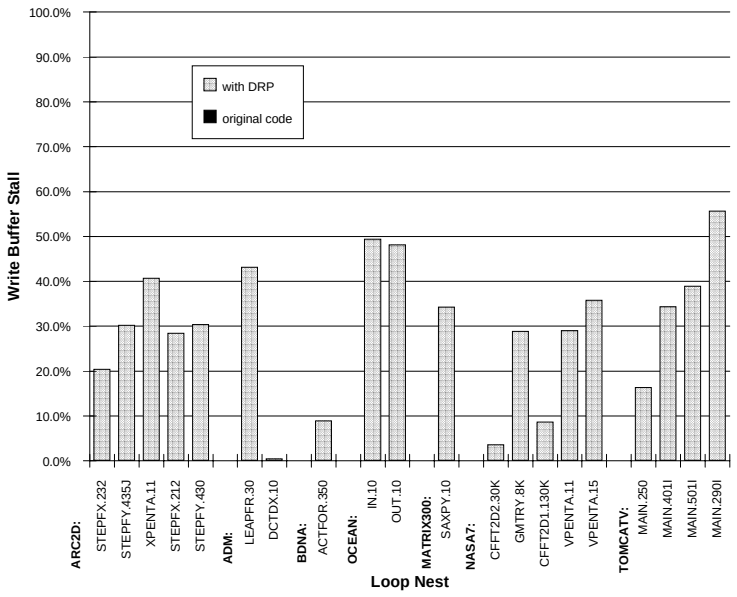

Figure 5.3 Stalls Due to the Full Write Buffer

using the original addresses. Another reason is that most write accesses are not *DRP*-transformed for no-write-allocate cache, thus write accesses do not benefit from the *DRP* technique very much. The solution to this problem is discussed in Section 5.2.3 - 5.2.5.

Figure 5.5 shows the *await* operation overhead. The overhead is represented as percentage of execution cycles for which the *await* instruction stalls the processor for the transformed loop nests. As long as the loop nests shown in the Figure 5.5, there are not many *await* operation stalls, which means that most *precollect* operations are completed when the corresponding *await* operation is invoked.

The memory bus utilization for the transformed and original loop nests is displayed in Figure 5.6. Since *precollect* operation may access memory when the bus is not used, it increases the bus utilization. As long as the bus utilization does not exceed 100%, the

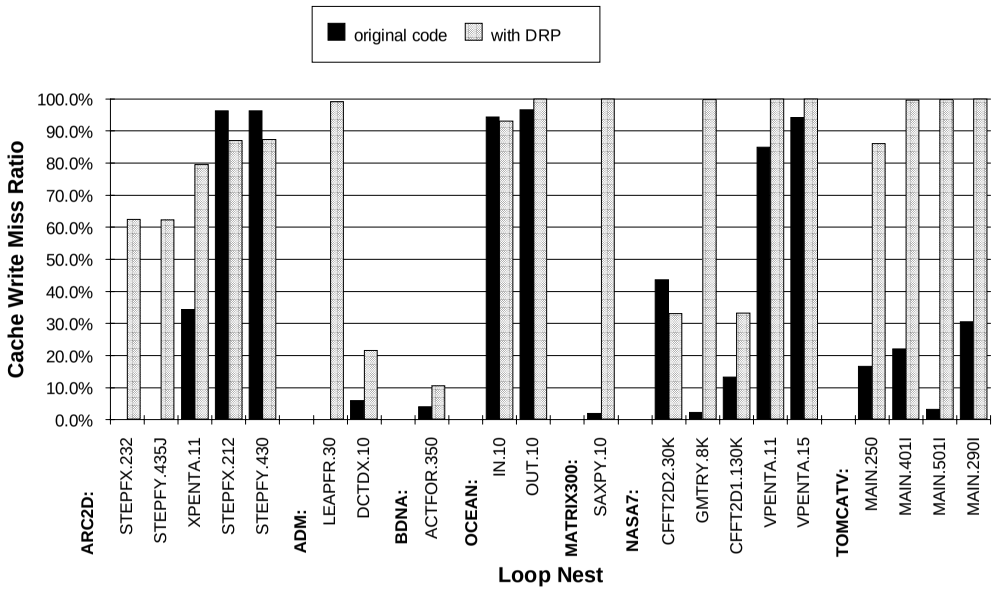

Figure 5.4 Cache Write Miss Ratio of Each Loop Nest

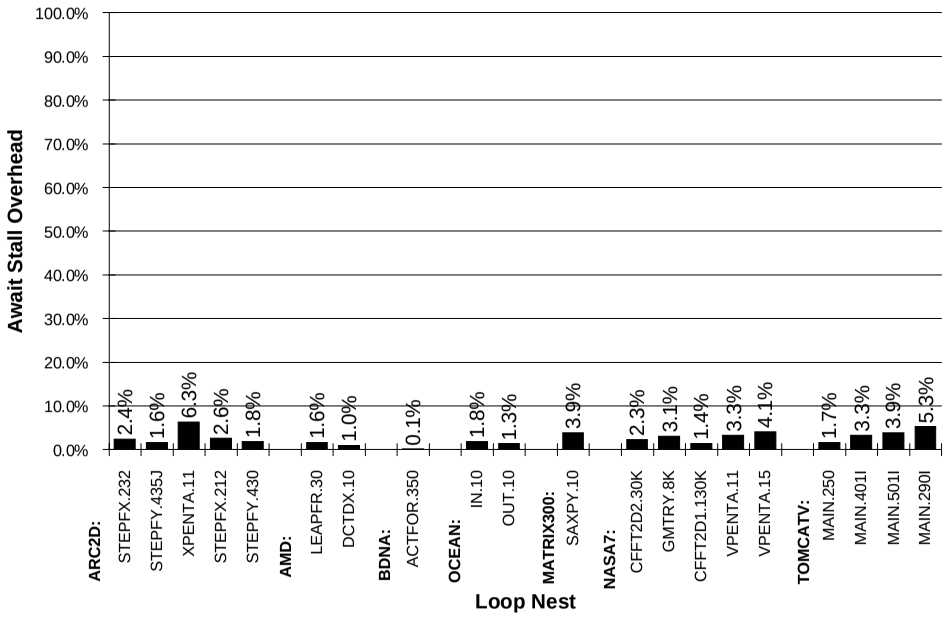

Figure 5.5 *Await* Stall Overhead for the *DRP*-transformed Code and the Original Code for Loop Nests

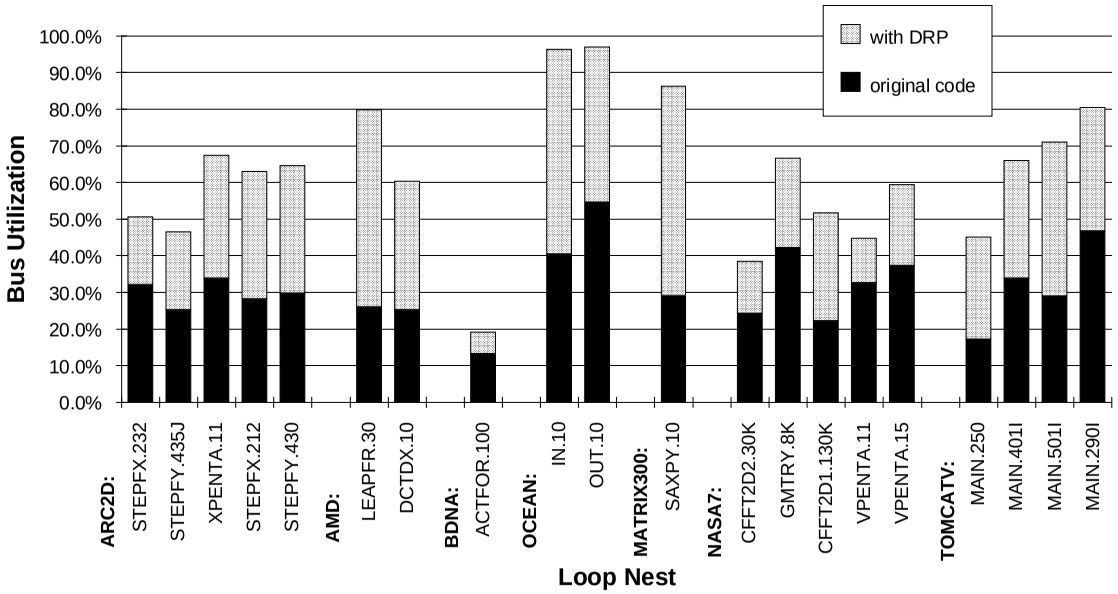

Figure 5.6 Bus Utilization for the *DRP*-transformed Code and the Original Code for Loop Nests

precollect operation can effectively fetch the data from memory minimizing stalls of the cache or processor since *precollect* operation has the lower priority than the cache. Once the bus utilization starts to saturate, the *precollect* operation and the write buffer can not get the bus effectively, introducing many processor stalls due to the incomplete data relocation and the full write buffer, respectively.

5.2.2 Entire Benchmark Results

Table 5.6 show some statistics of entire benchmark transformed for *DRP*. The second column is the number of the inner-most loops which are transformed. The third column is the number of dynamic relocated array accesses as a percentage of the array accesses for the entire benchmark. The fourth column gives the original measured cache miss ratio

Benchmark Name	Transformed Innermost Loops	% of Total Array Accesses	Estimated Cache Misses			% of Total Execution
			Original	DRP	Improve	
ADM	4	3.8	9.0	8.2	8.9%	3.2%
BDNA	4	63.2	26.7	13.8	48.3%	66.3%
OCEAN	13	23.0	36.9	26.0	29.5%	20.2%
ARC2D	37	87.6	40.8	2.2	94.6%	89.9%
MATRIX300	1	97.6	48.2	0.0	100%	97.6%
NASA7	18	60.6	42.8	9.3	78.3%	73.0%
TOMCATV	4	95.8	34.0	2.0	94.1%	89.3%

Table 5.6 Statistics of *DRP*-transformed Entire Benchmark

while the fifth column gives the cache miss ratio estimated by reducing to zero the number of cache misses in all transformed loops. The estimated percentage improvement for the cache miss ratio due to the *DRP* technique is given in the sixth column. The last column is the original code execution time for the loop nests selected by the *DRP* technique as a percentage of the total original code execution time. This is almost proportional to the percentage of the array access counts in the third column. All benchmarks except for **ADM** and **OCEAN** have high percentage of the execution time for the transformed loop nests. For **ARC2D**, **MATRIX300**, and **TOMCATV**, the estimated cache miss ratios are expected to be improved greatly by the *DRP* technique.

The increase in the dynamic instruction count for *DRP* over the original code is displayed in Figure 5.7. The figure shows that the instruction overhead due to the *DRP* transformation is relatively small. For **TOMCATV**, the dynamic instruction count for *DRP*-transformed code is smaller than that for the original code. This will happen if there exist many complicated address calculation such as multidimensional array in the original code. The complicated address calculation may need more than one instruction

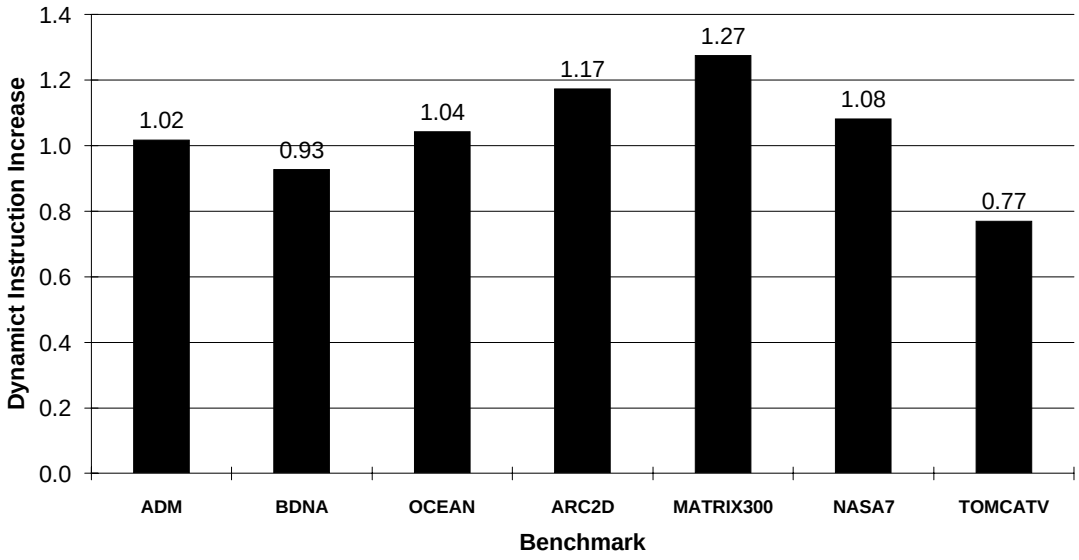

Figure 5.7 Instruction Overhead of *DRP*-transformed Code Over the Original Code

while the *DRP*-transformed array takes only one instruction since the array is always a single dimension. This is a hidden advantage of the *DRP* technique.

Speedups for the simulated execution of the *DRP*-transformed code over the original code for the seven benchmarks are given in Figure 5.8. The total speedup of **ARC2D**, **MATRIX300**, **NASA7** and **TOMCATV** is high although the rest of the benchmarks have small speedup. For **ADM** and **OCEAN**, the small speedup is attributable to the fact that the percentage of the execution time spent in the transformed loop nests is relatively small due to the restrictions for *DRP* transformations as described in Section 4.4. In the case of **BDNA**, a single loop nest dominates the entire execution time. As described in Section 5.2.1, not all array references in the loop nest were transformed for *DRP*, resulting in the small speedup.

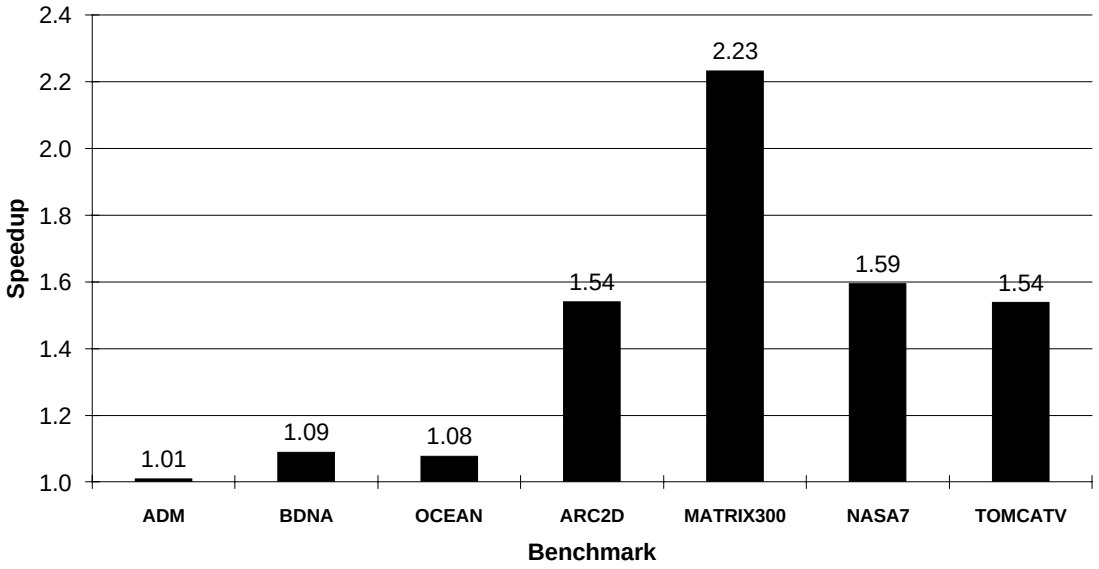

Figure 5.8 Speedup of *DRP*-transformed Code Over the Original Code

The cache read miss ratios show promising improvement for almost all benchmarks shown in Figure 5.9. For **ARC2D**, **MATRIX300**, and **TOMCATV**, especially, the cache miss ratios are reduced to only a few percent of the read accesses. The binding-prefetching mechanism of the *DRP* technique guarantees that most of the necessary data for the computation resides in the cache. Also, the compression of the data in the relocation buffer increases the utilization of the cache so that more data can reside in the cache at the same time.

Figure 5.10 shows the processor stall overhead due to the full write buffer. As described in Section 5.2.1, the *DRP* technique increases processor stalls due to the full write buffer since *precollect* operations block the write buffer and the cache write misses increase for the *DRP*-transformed loop nests.

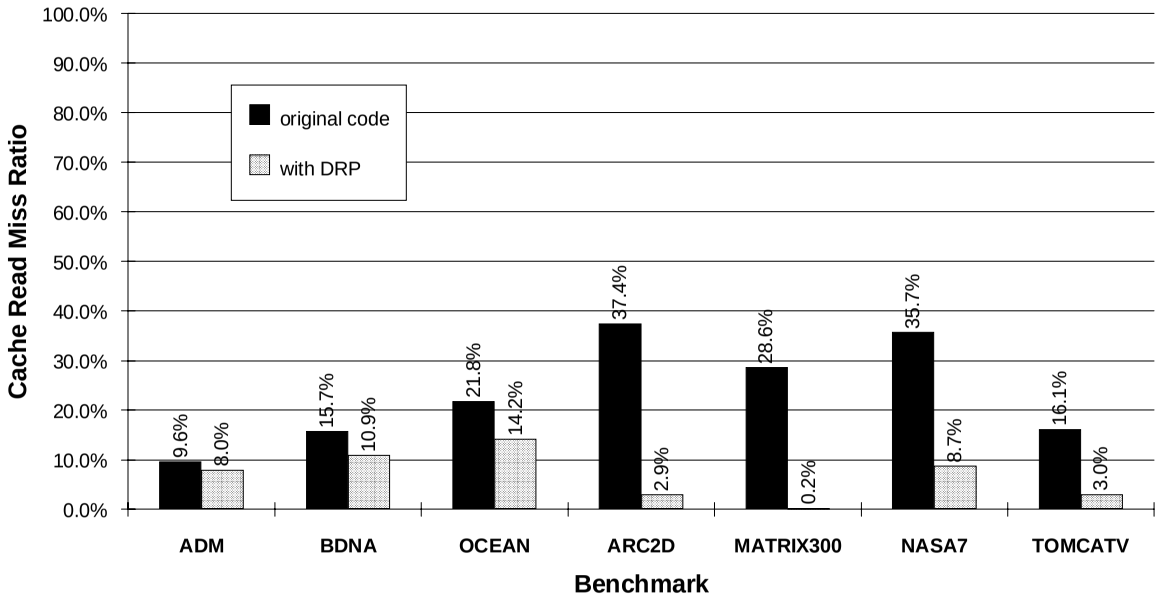

Figure 5.9 Cache Read Miss Ratio for Original Code and *DRP*-transformed Code

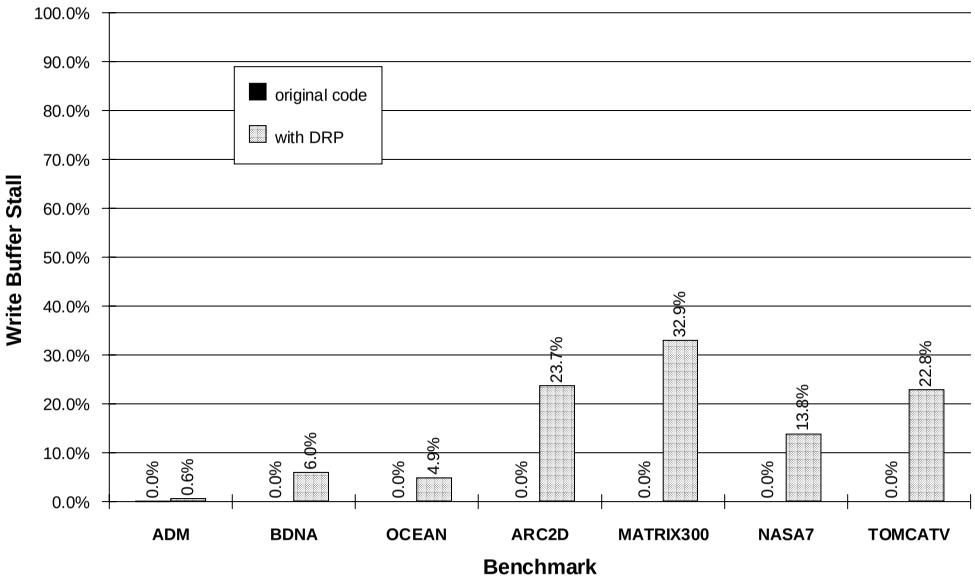

Figure 5.10 Stalls Due to the Full Write Buffer for Original Code and *DRP*-transformed Code

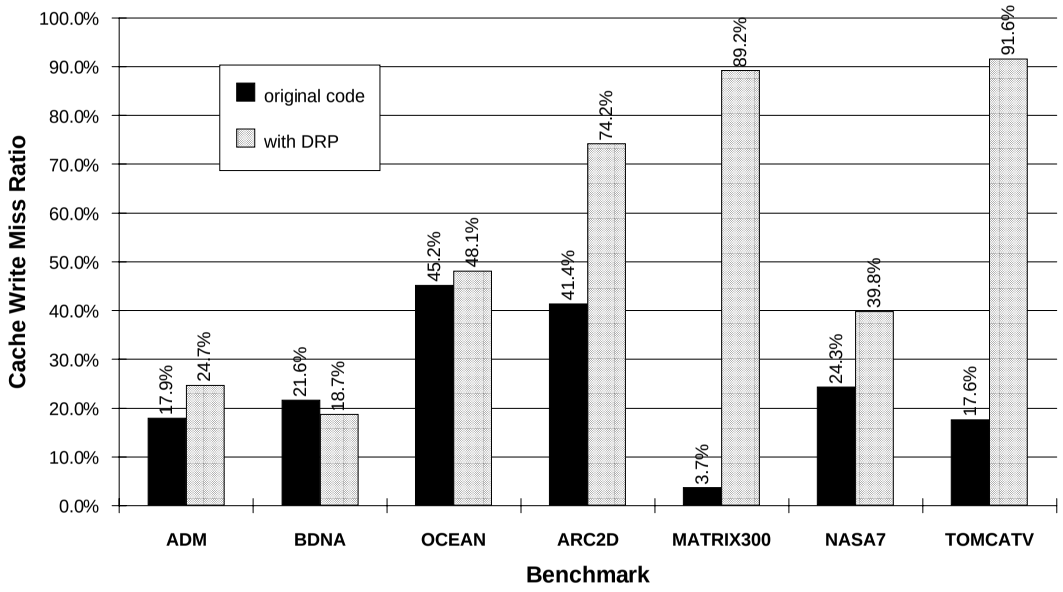

Figure 5.11 Data Cache Write Miss Ratio for Original Code and *DRP*-transformed Code

Among the benchmarks used, **ARC2D**, **MATRIX300**, **NASA7** and **TOMCATV** have the highest percentage of stalls due to the full write buffer since these benchmarks contain *DRP*-transformed loop nests with high execution frequency and the cache write misse ratios increased significantly as shown in Figure 5.11.

Figure 5.12 shows *await* operation stall overhead. An *await* operation stalls the processor if the corresponding *precollect* or *preallocate* operation is not completed at that time. This will happen if there are too many memory requests by *precollect* operations to complete to fetch all requested data into the cache within the computation which is software-pipelined with the *precollect* operations. For the benchmarks studied, there is very little stall overhead due to *await* operations as shown in Figure 5.12. In other words, the software pipelining of *precollect* operations works very well, and invoking *precollect*

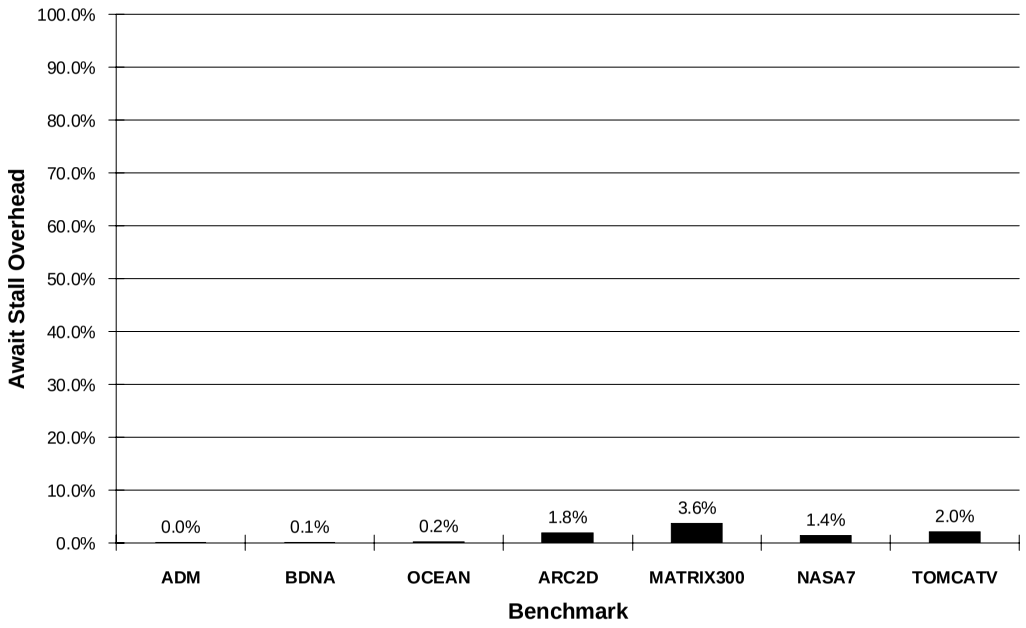

Figure 5.12 *Await* Stall Overhead for *DRP*-transformed Code

operations at one iteration ahead of the outer loop is early enough to finish to fetch all requested data into the cache.

The memory bus utilization for the *DRP*-transformed code and original code is displayed in Figure 5.13. As described in Section 5.2.1, *DRP* technique increase the bus utilization due to the *precolloect* operations and the increase of the cache write misses. Although the bus utilization for the entire benchmark is not saturated, it may happen that the bus untilization for each loop nest is saturated.

5.2.3 Effect of *DRP* for Write Accesses

The experiment in Section 5.2.2 showed that there were many stalls due to the full write buffer. One possible solution to this problem is to apply the *DRP* technique to the

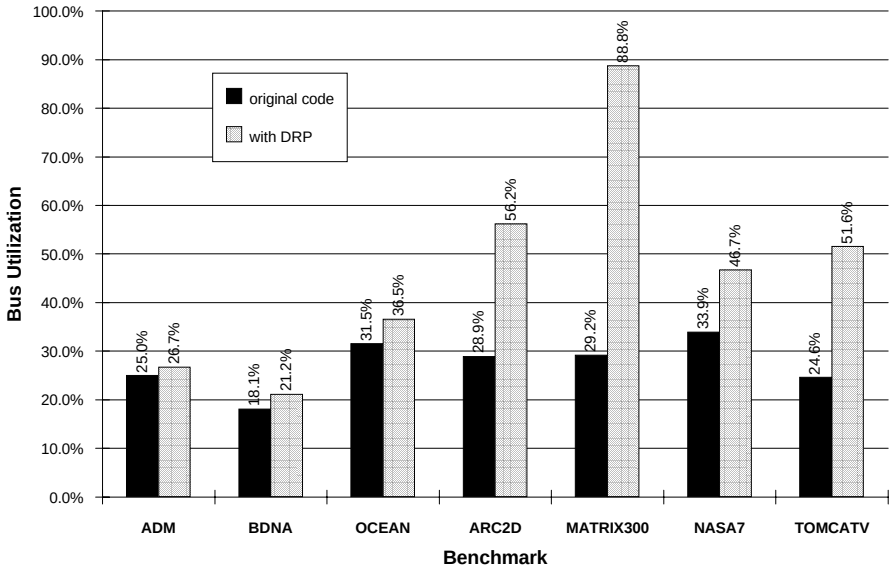

Figure 5.13 Bus Utilization for the *DRP*-transformed Code and the Original Code

array accesses for all stores as well as for loads. Originally, if array references for stores are not referenced later in the loop nest, these array reference were not transformed for *DRP* since write accesses to the cache with a no-write-allocate policy do not stall the processor if the write buffer is not full. In this section, the *DRP* transformation is applied to the array accesses for all stores regardless the loads thereafter in order to examine the effect on the write buffer stall overhead.

Applying *DRP* technique to array accesses for stores does not increase *precollect* operations but *preallocate* and *distribute* operations as described in Section 4.4.2. *Preallocate* operations do not affect bus accesses at all since the operation modifies only the cache tag without fetching the data. Since the number of *distribute* operations increases, the operation affects bus accesses, therefore the write buffer.

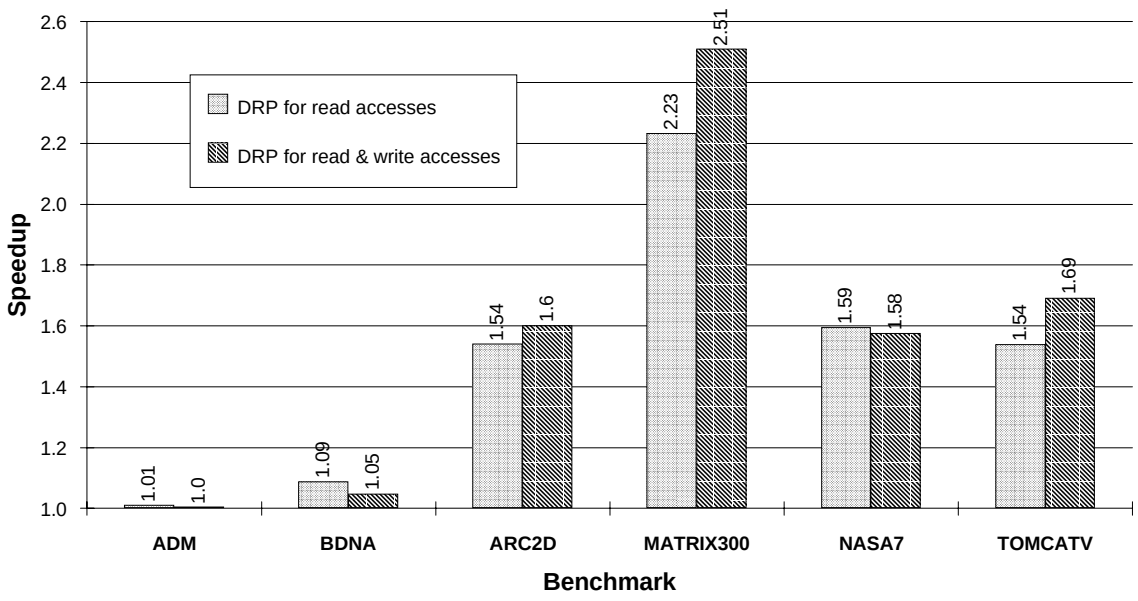

Figure 5.14 Effect of *DRP* for Write Accesses on Speedup

Regarding the speedup which results from transforming stores for *DRP*, only **MATRIX300** and **TOMCATV** have relatively large improvement while the others have a few or less improvement as shown in Figure 5.14. Figure 5.15 shows the cache read miss ratios. The cache read miss ratio increased for almost all benchmarks. The reason is that transforming stores as well as loads for *DRP* increases the space of relocation buffer in the cache, causing to perge out the other un-relocated data from the cache, which may be reused in un-transformed loop nests. For **NASA7**, the cache read miss ratio is increased, resulting in the less improvement of speedup.

Figure 5.16 shows the write buffer stall overhead. For most benchmarks except **BDNA**, write buffer stalls are greatly reduced by transforming the array accesses for stores since the *DRP* technique greatly reduces the cache write miss ratios as shown in

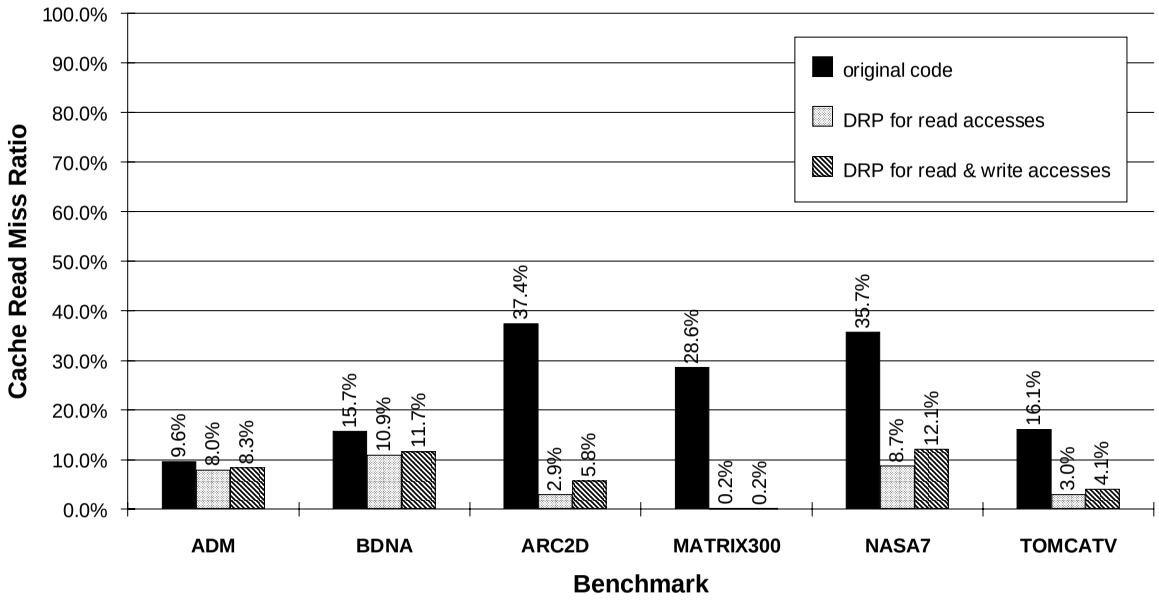

Figure 5.15 Effect of *DRP* for Write Accesses on Cache Read Miss

Figure 5.17. For **BDNA**, as described in Section 5.2.1, not all the array references in the loop nest were transformed for *DRP*. Therefore, several un-transformed array references cause the cache write misses, resulting in the write buffer stalls. This is the reason for the no improvement of speedup for **BDNA**.

Figure 5.18 shows the *await* stall overhead. For **ARC2D**, **MATRIX300**, **NASA7**, and **TOMCATV**, the *await* stall overhead increased. Applying *DRP* to stores increases the number of *preallocate* operations, resulting in the increase of *await* stall overhead since more cache line tag need to be set even though *preallocate* operations do not fetch data at all. The large increase of the *await* stall overhead for **MATRIX300** comes from the bus saturation as described below.

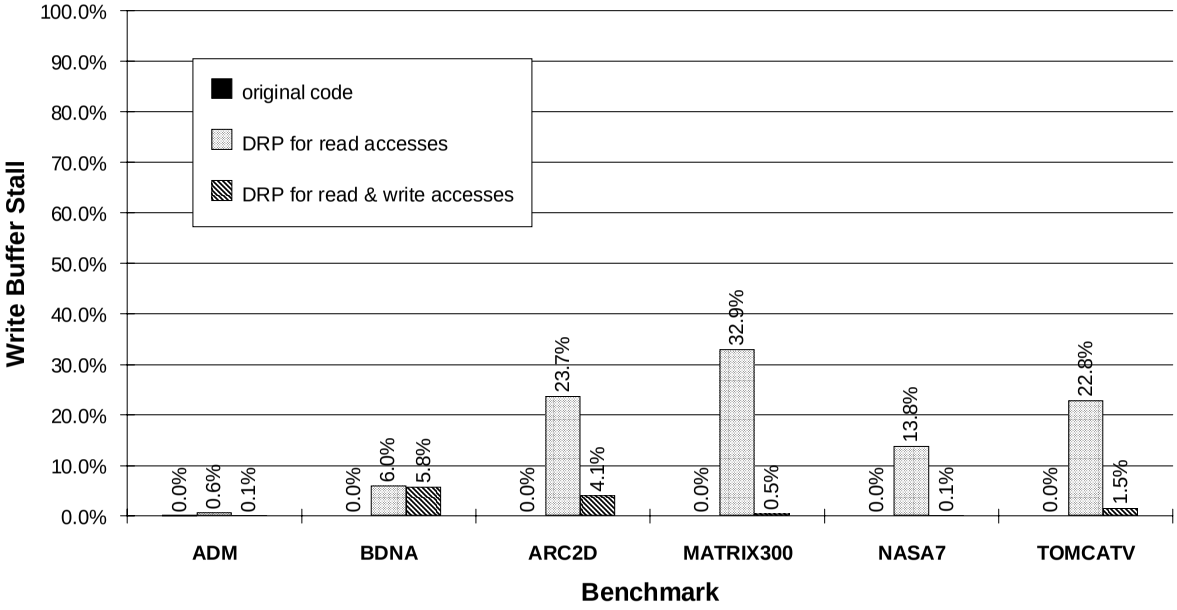

Figure 5.16 Effect of *DRP* for Write Accesses on Write Buffer Stall

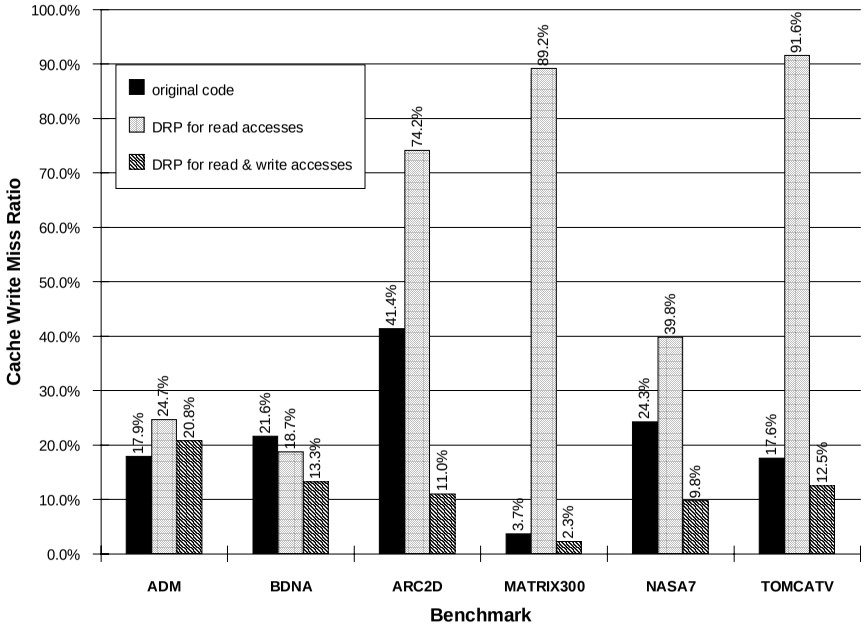

Figure 5.17 Effect of *DRP* for Write Accesses on Cache Write Miss

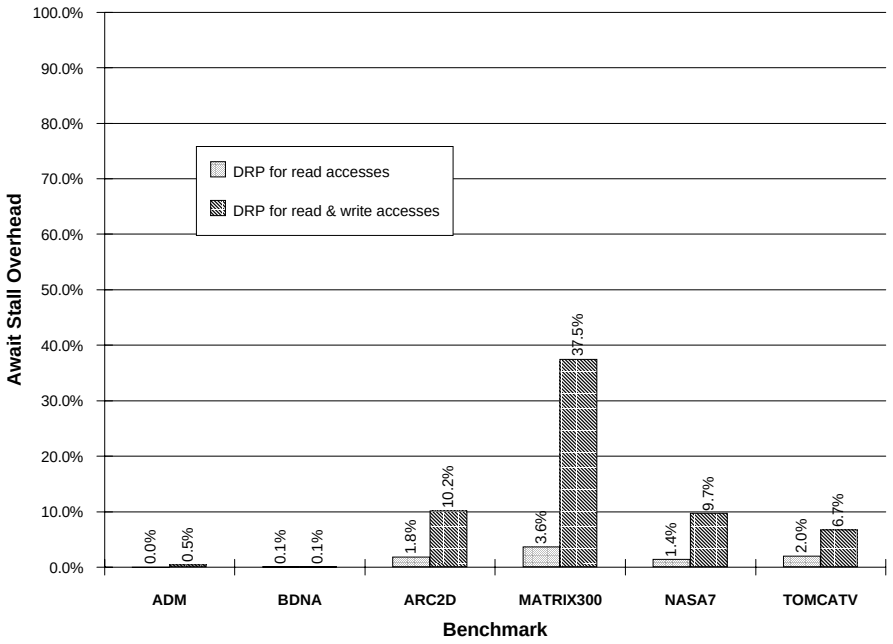

Figure 5.18 Effect of *DRP* for Write Accesses on *Await* Stall Overhead

As shown in Figure 5.19, applying *DRP* to stores increases the bus utilization. The reason is not the increase of the memory accesses but the reduction of the execution cycles, which comes from the reduction of the write buffer stalls. For **MATRIX300**, especially, the bus utilization reached almost 100%, which means that memory is always accessed during the execution cycles. This is the major reason that the *await* stalls for **MATRIX300** are very high as shown in Figure 5.18. Some scheme to reduce the bus traffic is necessary in order to solve this problem. This is discussed in Section 6.2.

5.2.4 Effect of Write Buffer Size

Another possible solution for the write buffer stalls is to increase the size of the write buffer. The size was originally 8 entries, and it was enough large to handle the cache

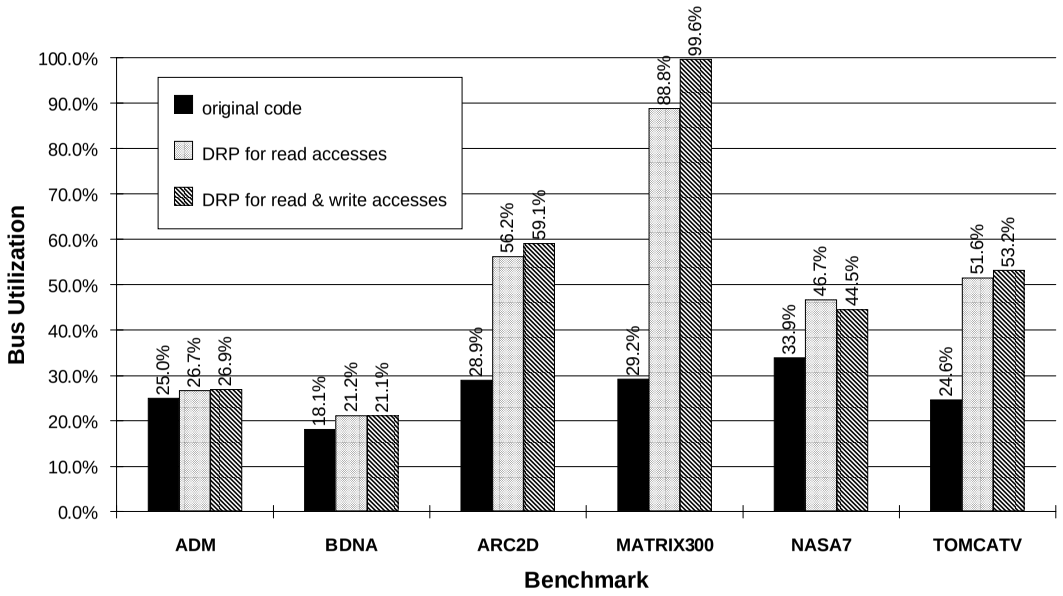

Figure 5.19 Effect of *DRP* for Write Accesses on Bus Utilization

write misses for the normal cache as shown in Figure 5.10. However, for the cache with the *DRP* technique, which has much higher utilization than a normal cache, the size was not large enough. The effect of the write buffer size is examined by changing the size from 8 to 128 entries.

Figure 5.20 shows the speedup for each write buffer size. Increasing the write buffer size is helpful to improve the speedup up to a certain write buffer size. After that, the improvement diminishes. Also, in a real implementation of the write buffer, the hardware complexity of the full associative search limits the maximum size.

Figure 5.21 shows the cache read miss ratios. Regarding the cache miss ratio, there are not much differences since the write buffer stalls affects mainly not the cache state but the execution cycles of the processor. However, increasing the write buffer size has

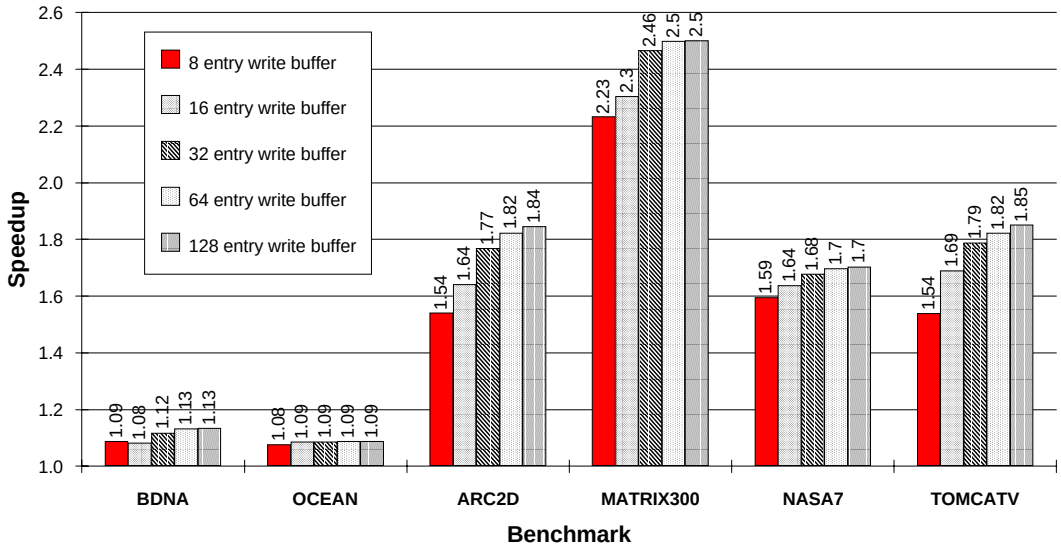

Figure 5.20 Effect of Write Buffer Size on Speedup

a very slight improvement of cache misses since larger write buffer can hold more data, which may be referenced by the processor, resulting in the reduction of the cache misses.

Figure 5.22 shows the stall overhead due to the full write buffer. The speedup shown in Figure 5.20 comes from the improvement of the write buffer stalls. Since cache write miss ratios are not affected much by the write buffer size as shown in Figure 5.23 due to the same reason as for cache read misses just as describe above, the improvement of the write buffer stalls comes from simply the fact that larger write buffer does not easily become full. The improvement of the write buffer stall diminishes after a certain size, which is large enough to hold all the data that the cache misses for stores during the time when the bus is not available for the write buffer.

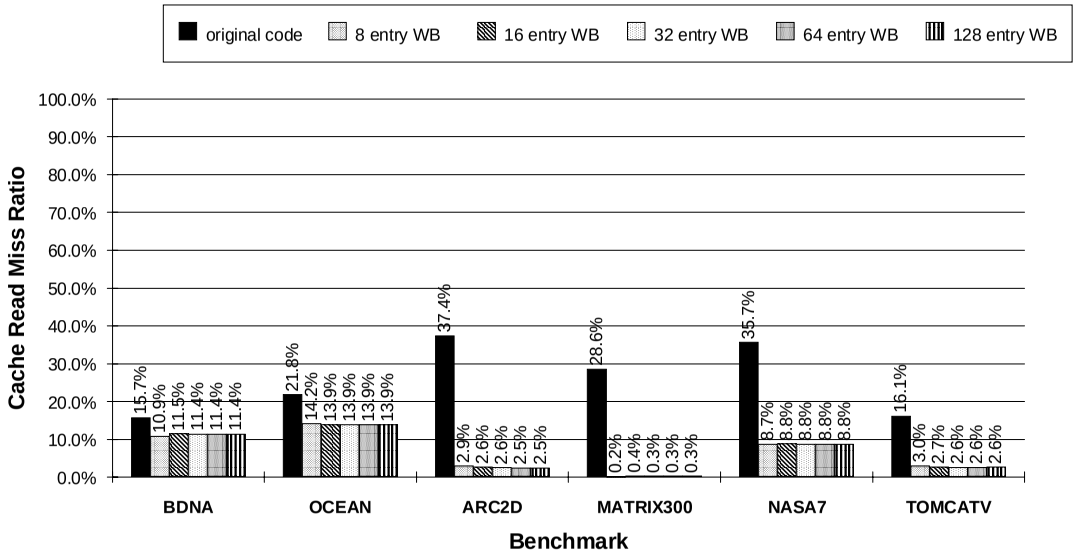

Figure 5.21 Effect of Write Buffer Size on Cache Read Miss

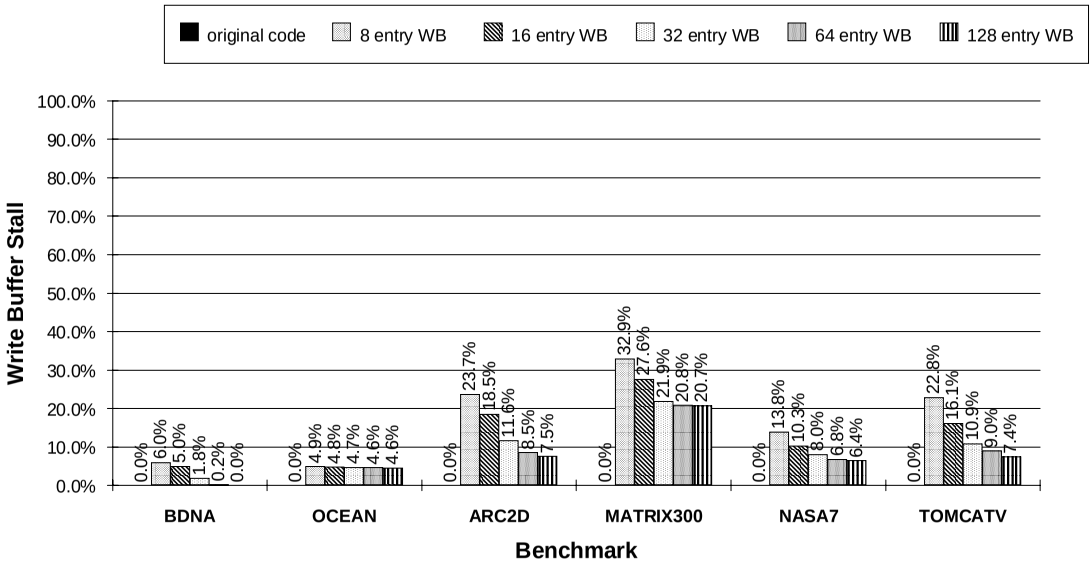

Figure 5.22 Effect of Write Buffer Size on Write Buffer Stall

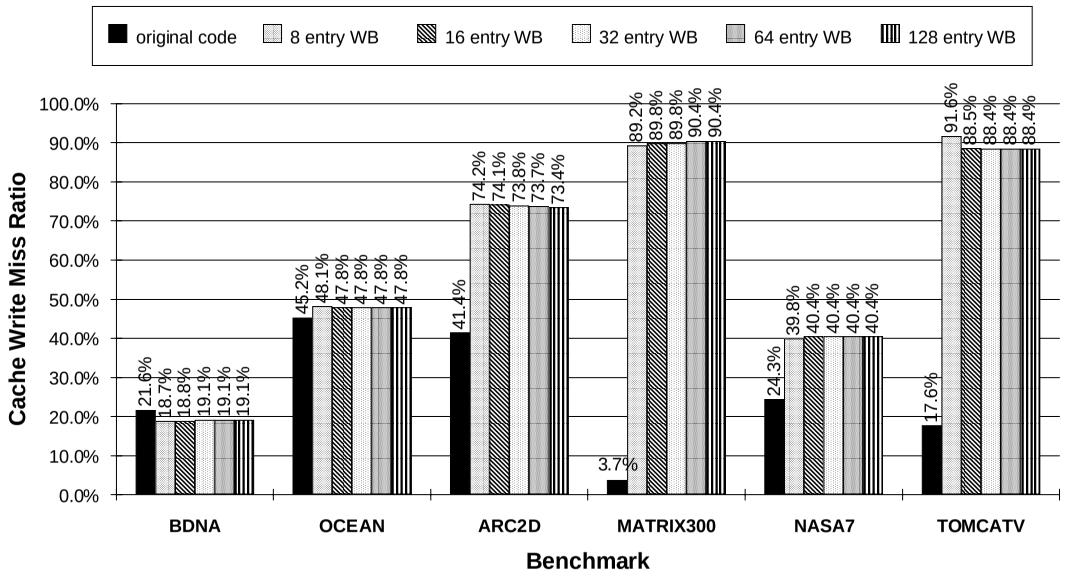

Figure 5.23 Effect of Write Buffer Size on Cache Write Miss

Figure 5.24 shows the *await* stall overhead. Increasing the write buffer size does not affect the *DRP* operations much since the write buffer is mainly used for the processor and the *DRP* operations have higher priority than the write buffer. The slight increase of the *await* stall overhead comes from the reduction of the execution cycles.

The bus utilization is increased by increasing the write buffer size as shown in Figure 5.25. This comes from the fact that the execution time itself is reduced by increasing the write buffer size. The usage time of the bus itself is not affected by the write buffer size.

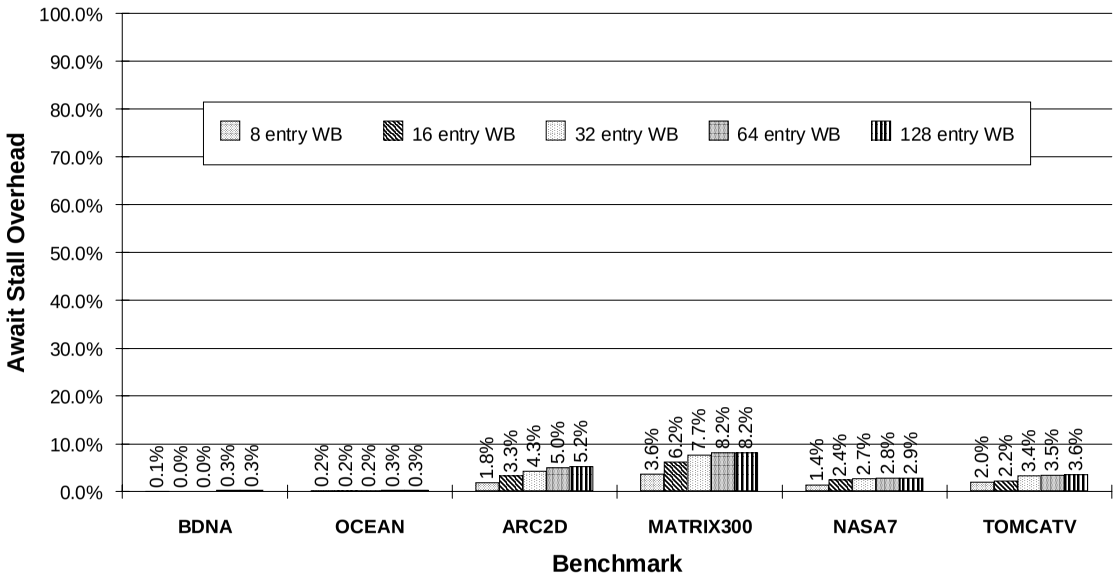

Figure 5.24 Effect of Write Buffer Size on *Await* Stall Overhead

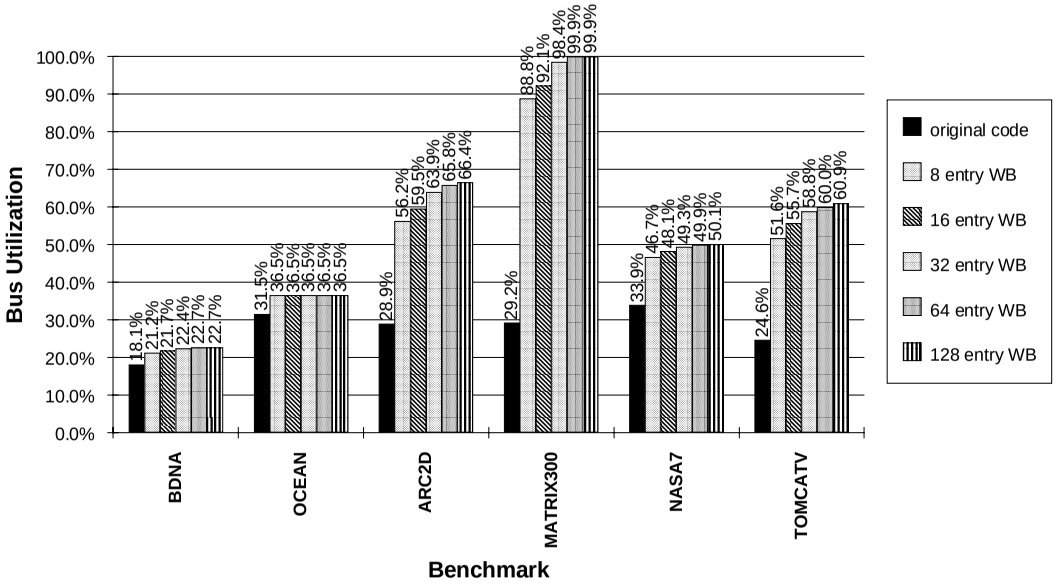

Figure 5.25 Effect of Write Buffer Size on Bus Utilization

5.2.5 Effect of Priority of *Precollect* Operation

Another way to solve the full write buffer problem is to change the priority of the *DRP* unit on the bus. Like a normal prefetch operation, the *precollect* operation had the higher priority than the write buffer. However, in the *DRP* technique, assigning the write buffer priority higher than the *precollect* priority is a reasonable way to reduce the stalls due to the full write buffer since the *precollect* operation may generate many memory accesses.

Figure 5.26 shows the effect of the priority on the speedup. For **ARC2D**, **MATRIX300**, **NASA7** and **TOMCATV**, there is 10% through 20% improvement in the speedup. These benchmarks had a high percentage of write buffer stalls when the priority of the *precollect* operation was higher than that of the write buffer. By switching the priority, these write buffer stalls are reduced, resulting in the improvement of speedup as shown in Figure 5.26. For **ADM**, **BDNA** and **OCEAN**, there is not much improvement since the main reason for the poor performance of these benchmarks was not the write buffer stalls but other factors described in Section 5.2.2.

Figure 5.27 shows the cache read miss ratios of the original code, *DRP* with higher priority than write buffer, and *DRP* with lower priority than write buffer. There is not much difference between the cache miss ratios of the two priority policies since the priority policy changes only the order of the bus accesses between the write buffer and the *DRP* unit, not the memory access itself of the processor.

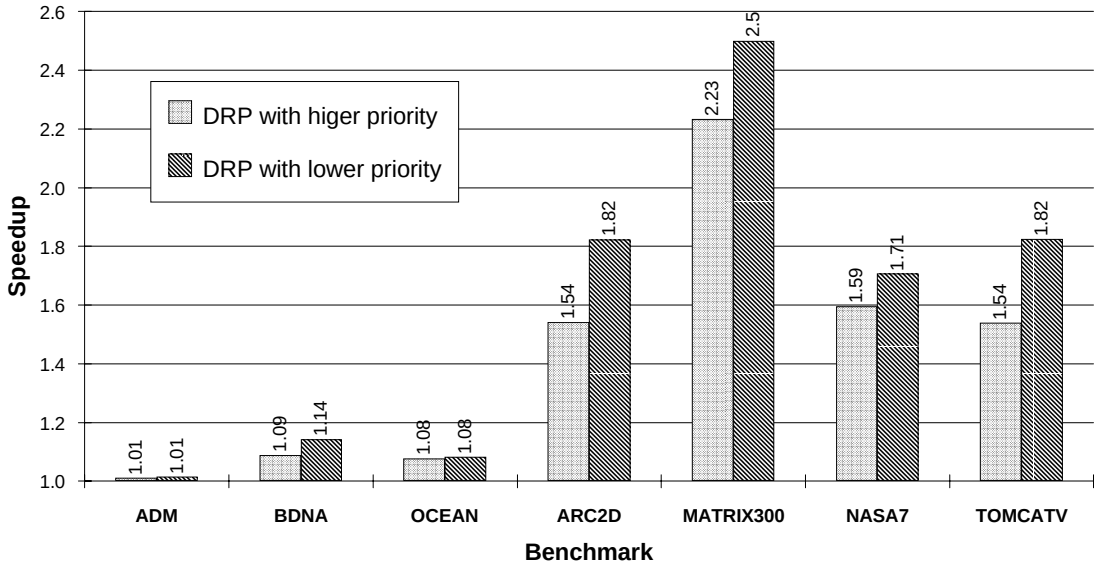

Figure 5.26 Effect of Priority of *Precollect* Operation on Speedup

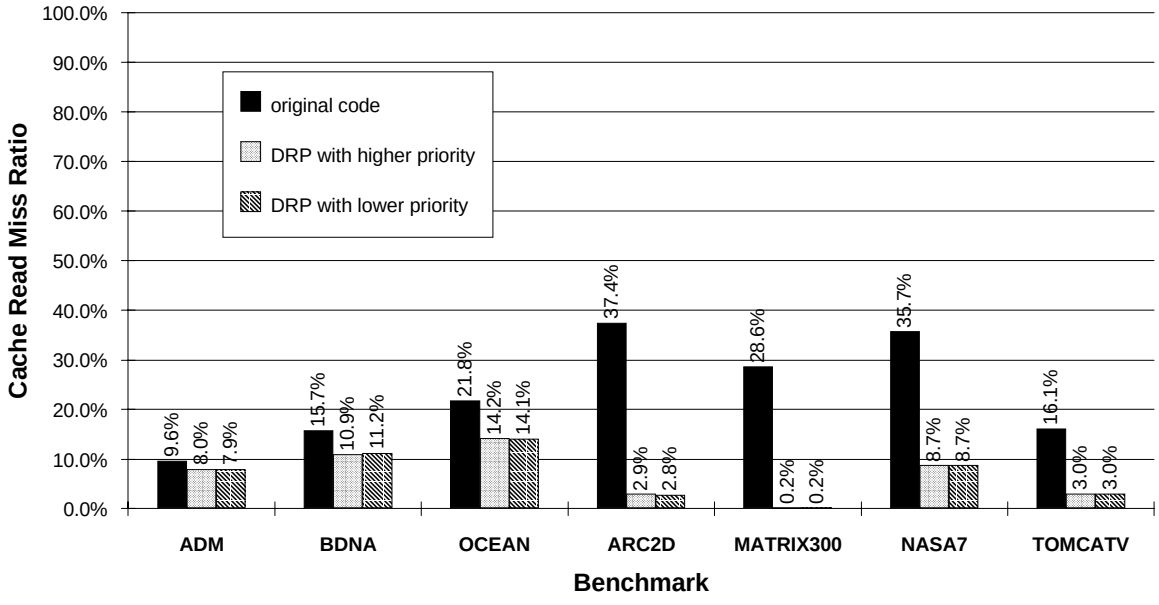

Figure 5.27 Effect of Priority of *Precollect* Operation on Cache Read Miss

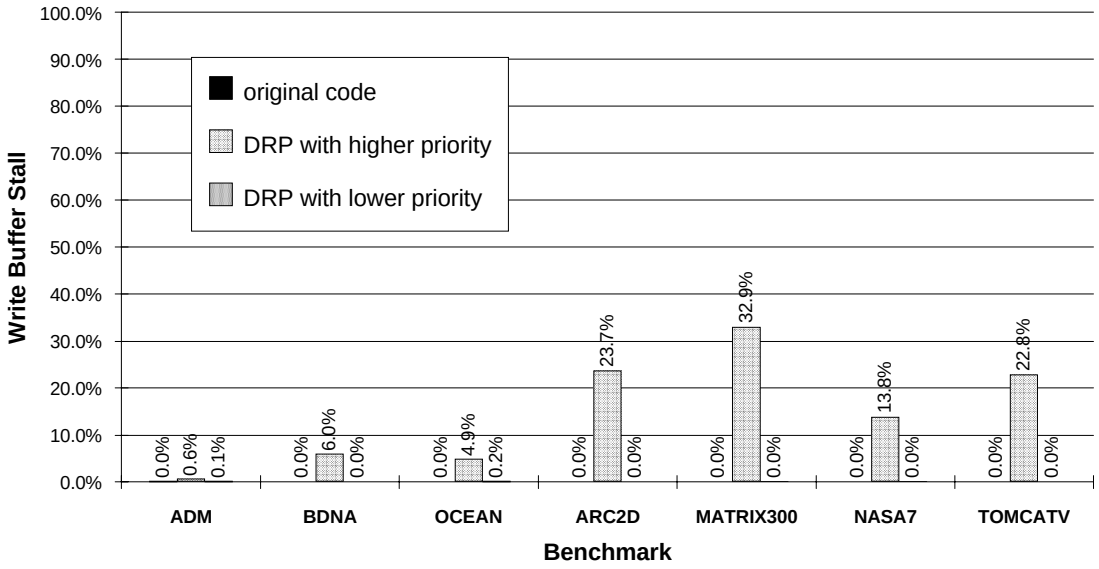

Figure 5.28 Effect of Priority of *Precollect* Operation on Write Buffer Stalls

Figure 5.28 shows the effect of the priority on the write buffer stalls. By changing the priority of the *DRP* to lower than that of the write buffer, most of the write buffer stalls are eliminated, resulting in the improvement of speedup as shown in Figure 5.26. Since the cache write miss ratios is hardly affected by the priority as shown in Figure 5.29 due to the same reason as the cache read miss ratios just as described above, the reduction of the write buffer stalls comes from the fact that the *DRP* operations do not block the write buffer due to the change of the priority.

On the other hand, changing the priority of the *precollect* operation affects the *await* stall overhead. Since *precollect* operations are blocked if the write buffer uses the bus, prefetching of data from memory by *precollect* operations may be delayed due to a bus conflict with the write buffer, introducing the many *await* stalls as shown in Figure 5.30.

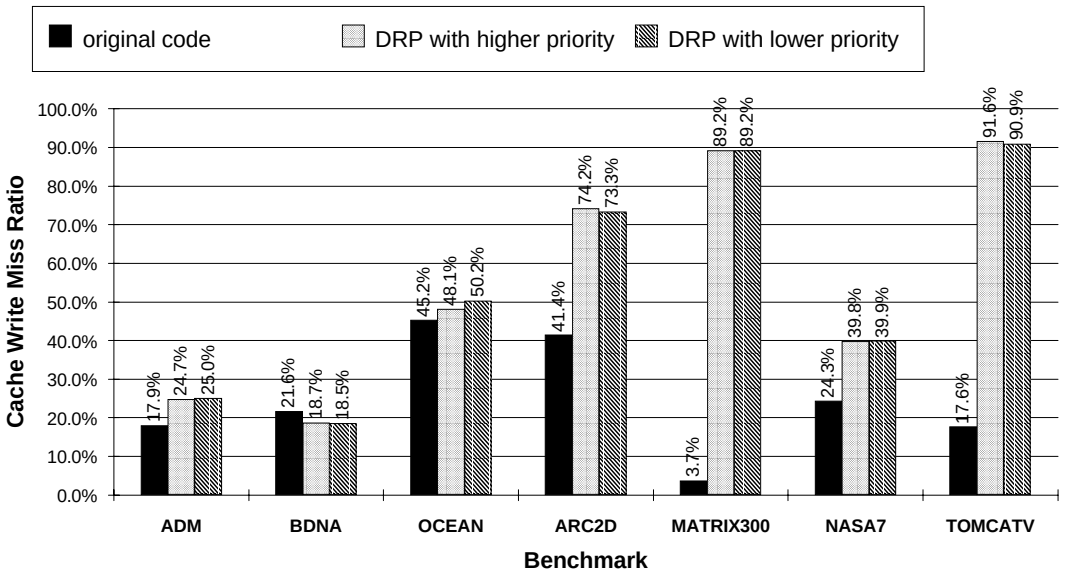

Figure 5.29 Effect of Priority of *Precollect* Operation on Cache Write Miss

Therefore, changing the priority of the *precollect* operation can reduce the write buffer stalls while increasing the *await* operation stalls. For all the benchmarks used, the reduction of the write buffer stall overhead is larger than the increase of the *await* stall overhead, resulting in the improvement of speedup as shown in Figure 5.26.

Figure 5.31 shows the effect of the priority on the bus utilization. Changing the priority of *precollect* operation does not affect the bus untilization much due to the same reason as changing the write buffer size as described in Section 5.2.4. The increase of the bus utilization in the Figure 5.31 comes from the reduction of the execution time. Especially, for **MATRIX300**, the bus utilization reached almost 100%. This is the major reason that there are a large number of the *await* operation stalls for **MATRIX300** as

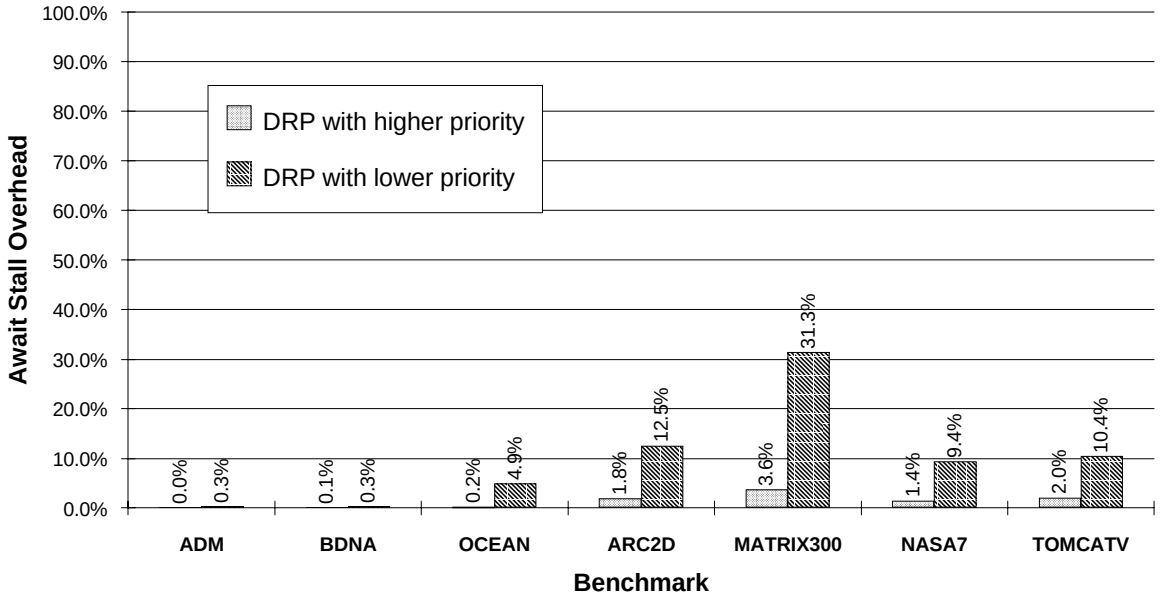

Figure 5.30 Effect of Priority of *Precollect* Operation on Await Stall

shown in Figure 5.30. Some schemes to reduce the bus traffic are necessary in order to solve this problem. This is discussed in Section 6.2.

5.2.6 Effect of *Distribute* Operation

Among the benchmarks used for the experimental evaluation, **ARC2D** and **NASA7** have loop nests that contain an array reference transformed for the *distribute* operation. Each loop nest was executed for only 1% of the entire execution time. All other benchmarks do not use the *distribute* operation at all. If *distribute* operations are not used at all, *preallocate* and *finishup* operations are not necessary any more. The experimental results show that transforming array write accesses for *DRP* is not worth adding three

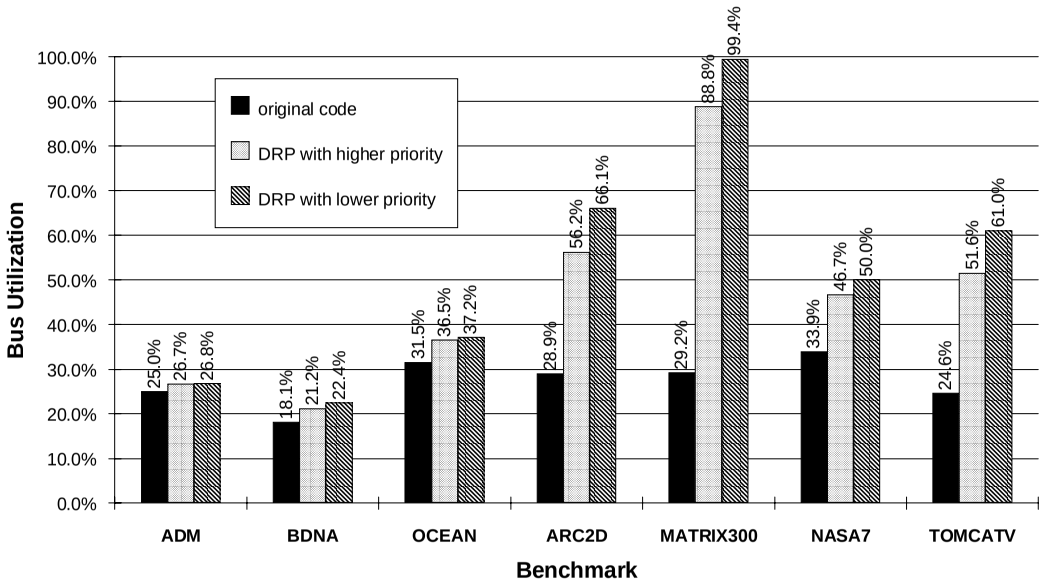

Figure 5.31 Effect of Priority of *Precollect* Operation on Bus Utilization

new operations. By removing these three operations, the control unit of the *DRP* unit as well as the instruction set can be simplified.

5.2.7 Comparison with Larger Caches

Figure 5.32 shows the speedup for normal caches with 16Kbytes though 64Kbytes and an 8Kbyte cache with *DRP* over an 8Kbyte normal cache. For **ARC2D**, **MATRIX300**, **NASA7**, and **TOMCATV**, the improvement from the *DRP* technique is much larger than that from the 64Kbyte cache. Therefore, the *DRP* technique can achieve better performance than the 64Kbyte cache. The extra hardware cost of the *DRP* unit will be smaller than the cost of the 64Kbyte cache.

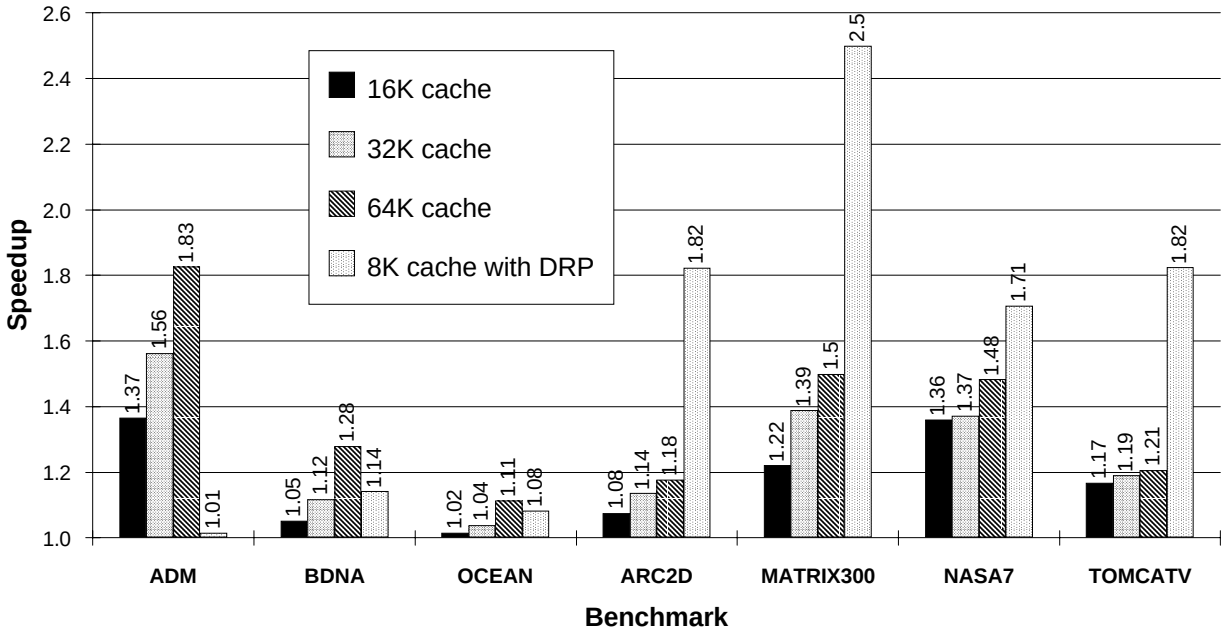

Figure 5.32 Comparison of Speedup by *DRP* with Larger Caches

CHAPTER 6

CONCLUSIONS

6.1 Summary

An architectural extension, referred to as data relocation and prefetching, is proposed to perform data relocation and compression during prefetching. Data relocation is employed to remove array-data mapping conflicts by compressing the accesses in a loop nest into sequential locations in the cache. Compression also improves utilization of the cache by transforming non-unit stride and array column accesses into sequential accesses that require fewer cache lines for storage. Furthermore, reduction of the cache space used to hold the data in the loop nest can increase the data reuse across transformed loop nests.

By combining the data relocation and prefetching hardware with supporting compiler transformations, the performance of loop nests is greatly improved for a set of array-based benchmarks. Also, I have shown that application of the data relocation and prefetching technique greatly improves the cache performance.

6.2 Future Research

For the programs that contains many transformed arrays, some technique is necessary to combine multiple bus accesses into single access in order to reduce the bus traffic.

In addition, the compiler transformations can be expanded and improved in order to transform more loop nests effectively. Further experiments are warranted to study the performance of this technique using various implementation parameters for the *DRP* hardware.

REFERENCES

[1] M. S. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *Proc. Fourth Int'l Conf. on Architectural Support for Prog. Lang. and Operating Systems*, pp. 63–74, Apr. 1991.

[2] O. Temam, E. D. Granston, and W. Jalby, "To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts," in *Proceedings of Supercomputing '93*, (Los Alamitos, California), pp. 410–419, IEEE Computer Society Press, Nov. 1993.

[3] D. Gannon and W. Jalby, *The characteristics of parallel programs*, ch. The influence of memory hierarchy on algorithm organization: Programming FFTs on a vector multiprocessor. MIT press, 1987.

[4] K. Gallivan, W. Jalby, U. Meier, and A. Sameh, "The impact of hierarchical memory systems on linear algebra design," Tech. Rep. CSRD-625, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, 1987.

[5] J. W. C. Fu and J. H. Patel, "Data prefetching in multiprocessor vector cache memories," in *Proc. 18th Ann. Int'l Symp. Computer Architecture*, (Toronto, Canada), pp. 54–63, June 1991.

[6] S. G. Abraham and D. E. Hudak, "Compile-time partitioning of iterative parallel loops to reduce cache coherence traffic," *J. Parallel and Distributed Computing*, vol. 2, pp. 318–328, 1991.

[7] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proceeding of Supercomputing '91*, pp. 176–186, Nov. 1991.

[8] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *Proc. Fifth Int'l Conf. on Architectural Support for Prog. Lang. and Operating Systems*, pp. 62–73, Oct. 1992.

[9] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W. W. Hwu, "Data access microarchitectures for superscalar processors with compiler-assisted data prefetching," in *Proc. 24th Ann. Workshop on Microprogramming and Microarchitectures*, (Albuquerque, NM.), Nov. 1991.

[10] W. Y. Chen, S. A. Mahlke, W. W. Hwu, T. Kiyohara, and P. P. Chang, "Tolerating data access latency with register preloading," in *Proceedings of the 6th International Conference on Supercomputing*, July 1992.

[11] R. M. Russell, "The Cray-1 computer system," *Communications of the ACM*, vol. 21, pp. 63–72, Jan. 1978.

[12] P. P. Chang, S. A. Mahlke, and W. W. Hwu, "Using profile information to assist classic code optimizations," *Software Practice & Experience*, Dec. 1991.

[13] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, "Profile-guided automatic inline expansion for c programs," *Software Practice & Experience*, May 1992.

[14] W. Pugh, "A practical algorithm for exact array dependence analysis," *Communications of the ACM*, vol. 35, pp. 102–114, Aug. 1992.

[15] W. Pugh and D. Wonnacott, "Eliminating false data dependences using the omega test," in *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 140–151, June 1992.

[16] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orzag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, and J. Martin, "The PERFECT club benchmarks: Effective performance evaluation of supercomputers," Tech. Rep. CSRD-827, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, May 1989.

[17] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proc. 18th Ann. Int'l Symp. Computer Architecture*, (Toronto, Canada), pp. 266–275, June 1991.

[18] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective technique for VLIW and superscalar compilation," *Journal of Supercomputing*, vol. 7, pp. 229–248, Jan. 1992.

[19] J. W. C. Fu and J. H. Patel, "How to simulate 100 billion references cheaply," Tech. Rep. CRHC-91-30, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, 1991.

VITA

Yoji Yamada received the Bachelor of Engineering degree and the Master of Engineering degree in physical electronics from the Tokyo Institute of Technology, Japan, in 1980 and 1982, respectively. He has been with Mazda Motor Corporation since 1982. His research interests are in computer architecture with an emphasis on memory optimization.