Goal-Directed Performance Tuning for Scientific Applications

by

Tien-Pao Shih

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1996

Doctoral Committee:
    Professor Edward S. Davidson, Chair
    Professor John P. Hayes
    Professor William R. Martin
    Professor Trevor N. Mudge

ABSTRACT

Goal-Directed Performance Tuning for Scientific Applications

by
Tien-Pao Shih

Chair: Edward S. Davidson

Performance tuning, as carried out by compiler designers and application programmers to close the performance gap between the achievable peak and delivered performance, becomes increasingly important and challenging as the microprocessor speeds and system sizes increase. However, although performance tuning on scientific codes usually deals with relatively small program regions, it is not generally known how to establish a reasonable performance objective and how to efficiently achieve this objective. We suggest a goal-directed approach and develop such an approach for each of three major system performance components: central processor unit (CPU) computation, memory accessing, and communication.

For the CPU, we suggest using a machine-application performance model that characterizes workloads on four key function units (memory, floating-point, issue, and a virtual "dependence unit") to produce an upper bound performance objective, and derive a mechanism to approach this objective. A case study shows an average 1.79x speedup achieved by using this approach for the Livermore Fortran Kernels 1-12 running on the IBM RS/6000.

For memory, as compulsory and capacity misses are relatively easy to characterize, we derive a method for building application-specific cache behavior models that report the number of misses for all three types of conflict misses: self, cross, and ping-pong. The method uses averaging concepts to determine the expected number of cache misses instead of attempting to count them exactly in each instance, which provides a more rapid, yet realistic assessment of expected cache behavior. For each type of conflict miss, we propose a reduction method that uses one or a combination of three techniques based on modifying or exploiting data layout: array padding, initial address adjustment, and access resequencing. A case study using a blocked matrix multiply program as an example shows that the model is within 11% of the simulation results, and that each type of conflict miss can be effectively reduced or completely eliminated.

For communication in shared memory parallel systems, we derive an array grouping mechanism and related loop transformations to reduce communication caused by the problematic case of nonconsecutive references to shared arrays and prove several theorems that determine when and where to apply this technique. The experimental results show a 15% reduction in communication, a 40% reduction in data subcache misses, and an 18% reduction in maximum user time for a finite element application on a 56 processor KSR1 parallel computer.

To my wife, Yung-Ching Chang

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to Professor Edward Davidson. I have been very lucky to have Professor Davidson as my advisor whose experience, knowledge, and inspiration have guided me to overcome all of the obstacles standing in the way toward my Ph.D. degree. Without his help, this dissertation would have been impossible to accomplish. In addition to improving my research capability, his patient instruction also greatly enhanced my writing and presentation skills.

I would also like to thank my committee, Professors John Hayes, Trevor Mudge, and William Martin for their efforts throughout this dissertation. I wish to thank Dr. Santosh Abraham for serving on my thesis proposal committee and for his help. I also wish to thank Daniel Windheiser for his suggestion to investigate the array grouping technique.

While I have been embedded in the hardship of research, my fellow graduate students not only have provided worthwhile discussions and suggestions but have also shared their exciting daily adventures with me which enriched my studying life. Special thanks to Bill Mangione-Smith, Jia-Jen Lin, Rabin Sugumar, Eric Boyd, Chuan-Hua Chang, Hung-Kuei Ku, Waleed Meleis, Karen Tomko, John-David Wellman, Alexandre Eichenberger, Waqar Azeem, Shih-Hao Hung, Hsien-Hsin Lee, Jude Rivers, Perry Wang, I-Cheng Chen, Chih-Chieh Lee, and Gheith Abandah. I also want to thank our secretaries and the DCO/CAEN staff for their daily behind-the-scenes assistance; in particular, Jeanne Patterson, for her kind offer to proofread my dissertation.

Finally, I want to thank my families: my father for his encouragement (he had passed away in 1990, the year I came to the University of Michigan); my mother for her unconditional financial support; my brother and sister for taking on my share of caring for my mother. Certainly, the persons who share my feelings day and night are my wife, son, and daughter. The naivete of my two loving kids often eased my stress. The understanding and unflagging support of my wife often recovered me from frustration and depression and smoothed out our sometime tense life.

TABLE OF CONTENTS

CHAPTERS

LIST OF TABLES

Table

LIST OF FIGURES

Figure

CHAPTER 1

Introduction

1.1 Performance Tuning Considerations and Approaches

Recent parallel supercomputers tend to use commodity microprocessors as their building blocks: the Cray Research, Inc. T3D/T3E consists of 16 to 2048 Digital ALPHA microprocessors, the HP-Convex Exemplar SPP Series contains 8 to 128 Hewlett-Packard PA-RISC microprocessors, the new Intel Teraflop supercomputer will be based on 9000 Intel Pentium Pro microprocessors, and the IBM SP2 uses IBM POWER2 microprocessors. Many of these microprocessors are high clock rate, RISC machines with multiple issue and multiple pipelined functional units. These microprocessors can generally achieve very high peak performance, as do the corresponding supercomputers. Nevertheless, the real performance in these systems is usually disappointing. Performance tuning, as carried out by compiler designers and application programmers to close this performance gap, becomes increasingly important and challenging as the microprocessor speeds and system sizes increase.

Much of the research on performance tuning, including the research of this dissertation, has focused on scientific (numeric) applications. Scientific applications are often very time-consuming. Hence, tuning performance for this type of application can result in significant rewards. Since the vast majority of the execution time of a scientific application is usually spent in relatively small, well-structured loop nests iterating over large arrays of floating-point data, researchers can typically concentrate on small regions of code containing loop constructs that reference large arrays and perform floating-point operations. Such regions offer more opportunities for exploring new methodologies and technologies to improve the performance.

Even though performance tuning on scientific code usually deals with relatively small program regions, it is still not known how to set up a reasonable performance objective and how to efficiently achieve the performance objective. Two performance figures are usually well known for each machine: the peak performance and the performance on some standard industry benchmarks. The peak performance does not consider application workloads and is rarely if ever achievable. Hence, using peak performance as the performance objective may waste effort in striving for an unreachable target. Benchmark performance results are also inappropriate to use for establishing the performance objective for a real application program under consideration since real programs rarely duplicate the algorithm behavior and data set of a benchmark. Reasonable performance objectives must be application-specific as well as machine-specific. Even if a performance objective can be determined, identifying performance problems requires expertise in both hardware and software. Recently, some tools, such as Mtool [1], Cprof [2], and MemSpy [3], have become available to identify performance problems. However, they do not provide direct solutions. Therefore, users may still have to use costly trial and error processes to find appropriate solutions for perceived performance problems.

Our experience suggests that the most efficient performance tuning method should be goal-directed, and we develop such an approach in this dissertation. The goal-directed approach tunes performance based on models that specify the best achievable performance so that users can know when to stop applying further tuning techniques, and models that indicate specific performance problems so that users can directly select and apply effective techniques. These techniques are also an integral part of the goal-directed approach and are designed to directly solve the types of performance problems that may be identified. In this dissertation, a goal-directed performance tuning method is developed for each of three major components of system performance: central processor unit (CPU), memory, and communication. Although load imbalance is also a critical component of system performance, it is not discussed in this dissertation. There are good known techniques for improving the load balance, which involves workload redistribution among the processors of a parallel system, and one or more of them may be used before, in conjunction with, or after the methods we develop in this dissertation.

We consider CPU performance with ideal memory, then consider memory performance later. The CPU performance is dependent on clock rate, clock cycles per instruction, and instruction count, which are in turn determined by hardware technology, processor organization, instruction set architecture, and compiler technology [4]. Hence, for a given microprocessor, the delivered CPU performance for an application can be improved by an optimizing compiler. To achieve high performance, a compiler must be aggressive enough to ensure that the bottleneck functional units are kept sufficiently busy. This task is difficult even for current microprocessors due to their multiple-issue and multiple-functional-unit features. More effort will be needed to optimize compilers for future microprocessor generations. The high complexity of current microprocessors also makes the performance objective difficult to determine and to achieve. Thus, a tool that can help determine the best achievable performance is an urgent need. Chapter 2 presents a performance bound model that can serve as this purpose.

Systems also require a high bandwidth, low latency memory system to provide data and instructions to the CPU in a timely fashion so that it can sustain the best performance. Since the CPU is usually many times faster than main memory, a memory hierarchy is used to close this gap. In the memory hierarchy, the cache subsystem is most critical. As the latency gap widens between microprocessor operations and memory accesses, the cache performance becomes increasingly critical to program performance.

The cache performance is determined to a first order approximation by its miss ratio. Cache misses in a single processor can generally be divided into compulsory, capacity, mapping, and replacement [5] (or compulsory, capacity, and conflict [6]). Among these types of cache misses, the compulsory misses are unavoidable; their miss penalties can, however, be hidden by latency hiding techniques such as data prefetching [7]. Capacity misses can generally be reduced effectively in scientific codes by loop blocking [8, 9]. However, mapping and replacement misses (collectively called conflict misses) which are caused by conflicts in the cache, are generally more difficult to eliminate.

Conflict misses can be distinguished into self, cross, and ping-pong conflict misses. It is difficult, in general, to determine how much each type of conflict miss degrades the cache performance. Recently some tools, which will be discussed in Chapter 3, have been developed to help identify cache performance problems; however, they do not provide information for all these types of conflict misses nor do they provide the direct solutions for the perceived problems. Although many techniques have been proposed for reducing cache conflict misses, each may only be effective for certain types of cache misses. Furthermore, since the same program may exhibit different cache behavior as a function of input data size, applying any particular technique may not achieve its expected improvement. As a result, selecting suitable techniques and applying them appropriately is an extremely challenging task. In Chapters 3 and 4, a goal-directed approach is presented which consists of a cache behavior model for providing information about the cache misses of each type and a set of conflict miss reduction methods for reducing them. Hence, for each type of cache miss reported by the model, users can directly select the right method to reduce it.

Communication transactions usually have high latency, and although communication transactions do not occur as often as cache misses, they can still severely degrade program performance. In this dissertation, the focus is on cache-based shared-memory parallel systems. In such systems, communication occurs when accessing some particular higher level of the memory hierarchy; hence, its performance will be affected by the locality of data references. While consecutive data references have good (spatial) data locality, other nonconsecutive reference patterns usually do not. Nonconsecutive references to a data structure are generally caused by incrementing an addressing index by a nonunit constant, by computing the next addressing index value using some function of the loop index variables, or by determining the next index indirectly through elements of another array. Such reference patterns are not uncommon in large scientific programs and generally have poor locality.

In a cache-based shared-memory multiprocessor system, these nonconsecutive references cause inefficient communication due to superfluous data within communication transactions, and result in inefficient cache and subcache utilization due to superfluity, false sharing, and pollution. These effects increase average data access time, and thus degrade program performance. Moreover, inefficient communication and false sharing result in an excessive number of communication transactions on the network, which will affect the scalability of an application. In Chapter 5, we show how the communication caused by nonconsecutive array references can be reduced and how we know when to apply this technique.

1.2 Dissertation Overview and Contributions

In this dissertation, we propose a goal-directed performance tuning approach for each of the three major components of the system performance: CPU, memory, and communication. Chapter 2 introduces a machine-application performance upper bound model and a mechanism for approaching this upper bound performance. Chapter 3 presents a method for building an application-specific cache behavior model that can provide information about different types of conflict misses for each array. Chapter 4 provides three conflict miss reduction methods, each capable of eliminating or reducing one type of conflict cache miss reported by the cache behavior model above. Chapter 5 provides a technique to reduce communication caused by nonconsecutive references in a cache-based shared-memory multiprocessor system and derives models that determine where and when to apply this technique. Finally, Chapter 6 summarizes the contributions of this dissertation and suggests directions for future research.

1.2.1 Machine-Application Bound Model for Tuning Processor Performance

In Chapter 2 we first present a performance bounding methodology that explains the performance of loop-dominated scientific applications on particular systems. We model the throughput of key hardware units that are common bottlenecks in concurrent machines. The four units currently used are: memory interface, floating-point, instruction issue, and a pseudo "dependence unit" which is used to model the effects of performance-limiting recurrences. We propose a workload characterization and derive upper bounds on the performance of specific machine-workload pairs. Comparing delivered performance with these bounds focuses attention on areas for improvement and indicates how much improvement might be attainable. To ease this process, a tool has been developed for producing the performance bound automatically.

In addition, we present a performance degradation analysis method that improves the CPU performance step by step. To illustrate this mechanism, we carried out a case study with the IBM RS/6000 and Livermore Fortran Kernels (LFKs) 1-12 serving as the machine and application examples, respectively. Four major performance degradation factors were identified: redundant loads and stores, read after write dependence stalls, floating-point store versus write-back conflicts, and poor fine-grain scheduling. For each of these performance degradation factors, a corresponding solution was derived based on three compiler techniques: register allocation, loop unrolling, and instruction interleaving.

The machine-application bound model produces a lower bound of 1.27 clocks per floating-point operation (CPF) averaged over the 12 LFKs, whereas machine peak performance, which ignores workload, is 0.5 CPF. The IBM V2.01 Fortran compiler attains only 2.43 CPF or 52.4% of the upper bound performance implied by the CPF lower bound. Code improvements in this study have achieved 1.36 CPF, increasing the harmonic mean performance to 93.7% of the bound (97.6% for steady-state inner loop performance). Subsequently, the V2.02 compiler achieved 1.75 CPF (1.60 CPF with carefully chosen preprocessing). A goal-directed compiler with bound knowledge could produce higher performance code more efficiently and automatically.

As mentioned above, system performance is also affected by cache misses. In Chapter 2, we used simple calibration loops to characterize cache performance (a more detailed cache behavior model is introduced in Chapter 3). The machine-application bound model is based on prior work by Mangione-Smith, Abraham, and Davidson [10]. Chapter 2 presents the first empirical evidence that the bound is achievable and derives a mechanism for approaching the performance bound. The work presented in this chapter has been summarized in [11] and [12].

1.2.2 Loop-Based Cache Model and Cache Miss Reduction Techniques

The primary goals of the research on memory performance tuning are two-fold: to derive a methodology for building application-specific cache models that can provide information about different types of cache misses and thus guide the selection of appropriate techniques, and to develop these techniques themselves and demonstrate their effective employment.

Chapter 3 presents a method to build an application-specific cache behavior model for conflict misses. Conflict misses can be classified as self, cross, and ping-pong conflicts. The model treats all three types and reports them separately for each array which enables the selection of appropriate techniques in a direct manner. The modeling method constructs cache models rather efficiently for a variety of cache organizations by using averaging concepts to determine the expected number of cache misses instead of attempting to count them exactly. This method provides a realistic assessment of expected average behavior, rather than a detailed assessment of one instance.

Chapter 4 then proposes three conflict miss reduction methods, one for each type of conflict miss. These methods use three techniques based on modifying or exploiting the data layout of arrays: (1) array padding, which attaches dummy elements to the end of each column, (2) initial address adjustment, which adjusts the initial address of each array, and (3) access resequencing, which reorders the references to achieve a desired sequence. Array padding is used to eliminate self conflict misses, ping-pong conflict misses are removed by using array padding and initial address adjustment, and cross conflict misses are reduced by using all three of the techniques.

We used a version of a blocked matrix multiply program as an example to evaluate the method for building cache behavior models and to verify the effectiveness of the conflict miss reduction methods for removing or reducing conflict misses. The model created by the method for this application can explain the number of cache misses to within 11% of the simulation results. After each application of the methods, the effectiveness of the three conflict miss reduction methods is demonstrated by running simulations of the program on an 8 KByte direct mapped cache with a 32 byte line size (the L1 cache organization of the ALPHA 21064 microprocessor). The results show that the method can effectively reduce each type of conflict miss. For example, for the cases using the largest allowable square block, after the application of the method for reducing cross conflict misses, which also removes self and ping-pong conflict misses, only 24% to 32% of the original total cache misses remain.

1.2.3 Array Grouping for Reducing Communication

In Chapter 5 we propose a systematic method of array reorganization by grouping several arrays into one interleaved layout. Array grouping and related loop transformations are used to improve the performance of programs with nonconsecutive references by reducing the number of communication transactions. We also prove several theorems that determine when and where to apply this technique.

Specifically, the method for reducing communication is as follows: A Program Section Array Table (PSAT) describes the reference types (read, write, consecutive, nonconsecutive) of shared arrays in each program section. Read-only arrays and arrays that are only referenced consecutively are removed from further consideration. Groups of arrays are selected for reorganization by first allocating arrays into the same set if they have the same reference types and the same sizes of nonconsecutively referenced dimensions. Then two sets are combined if i) the sets are accessed in more than one program section and ii) in each program section that accesses these sets nonconsecutively, the same index function or indirect reference array is used for accessing both sets. Finally each set is partitioned into disjoint subsets such that the group of elements of the arrays in each subset that are accessed by one instance of the index function or one element of the indirect reference array can be packed efficiently into communication transaction units of the system (*e.g.* subpages in the KSR1). A new array is then formed for each subset in the partition by interleaving groups of elements from the arrays in the subset. After all new arrays are constructed, the parts of the program that access these arrays are modified, and loop transformation techniques are applied to enhance the locality of access to the new array structures.

Because we group together data that have the same access pattern, all data in the same group will be accessed whenever one is referenced. For this reason, the corresponding transactions and cache lines should contain more useful data, and any previous false-sharing will decrease. In reality, however, the grouped arrays do not always generate more efficient communication. We have proved several theorems that evaluate the effectiveness of grouping and determine whether the number of transactions will actually be reduced.

In this chapter, we used the KSR1 parallel computer and a finite element application to evaluate the technique. The experimental results show a 15% reduction in communication, a 40% reduction in data subcache misses, and an 18% reduction in maximum user time on 56 processors of the KSR1 parallel computer. The work presented in this chapter has been summarized in [13].

CHAPTER 2

Machine-Application Performance Bound Model

This chapter develops a method for determining an upper bound on optimum performance for a scientific application on a highly concurrent processor [10, 14] and a mechanism for approaching this upper bound performance. The methodology is illustrated by deriving and using these bounds to evaluate and improve the performance of the IBM RS/6000 on a set of scientific code kernels, Livermore Fortran Kernels. Section 2.1 discusses the application of a performance bound and related work. Section 2.2 introduces the performance bound model and derives one for the IBM RS/6000. Section 2.3 demonstrates a process guided by the bound that analyzes performance degradation factors and provides solutions for the identified performance problems. Section 2.4 considers the transient effect caused by loop overhead on the program performance. Section 2.5 characterizes cache performance using simple load/store kernels. Section 2.6 draws conclusions.

2.1 Introduction and Related Work

Computer scientists and engineers use performance evaluation as a tool to achieve several different goals. Computer architects are interested in understanding existing and proposed machines in order to improve the design of new machines. The developers of libraries, compilers and operating systems focus on effective utilization of machine resources. Application developers are concerned with optimizing specific programs by understanding performance bottlenecks. End users may only be interested in choosing the fastest or most cost-effective among existing machines and application packages.

An effective performance evaluation technique can provide insight for each of these groups. Many researchers have evaluated scientific computers by focusing on the expected performance. These studies may involve detailed measurement on implemented machines (*e.g.* [15, 16]), the timing of large applications (*e.g.* [17, 18]), or analytic performance models (*e.g.* [19]). We believe that a more appropriate approach to improving performance for scientific applications is to bound the best achievable performance that a machine could deliver *on a particular code* and then try to approach this bound in delivered performance. This approach more accurately reflects the manner in which the scientific computing community views performance and the extent to which they are willing to work to optimize their codes.

2.2 Machine-Application Performance Bound Model

This section develops a method of finding an upper bound on optimum performance for a scientific application on a highly concurrent processor [10, 11, 14]. The methodology is illustrated

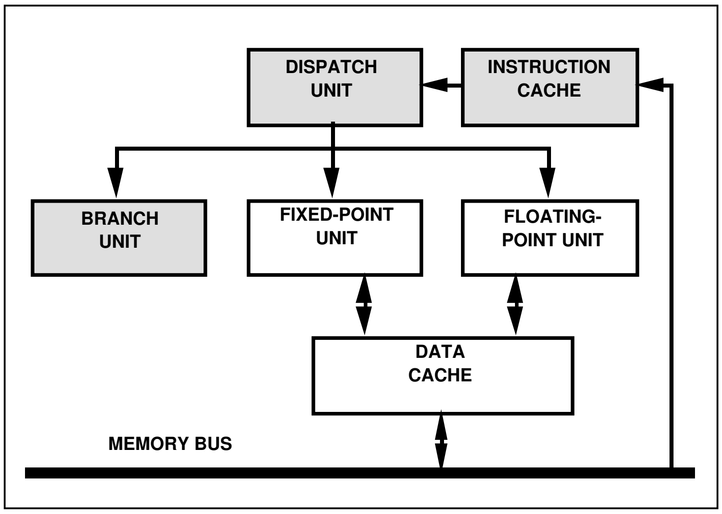


Figure 2.1: RS/6000 processor model 730. Physically, Instruction Cache, Dispatch Unit and Branch Unit are combined in a single chip, Instruction Cache Unit. Fixed-Point Unit and Floating-Point Unit are in separate chips. Data Cache consists of 2 or 4 identical chips with 16 KByte memory each.

$$
\begin{aligned}
&t_l = \max(t_i, t_m, t_f, t_d)\\
&t_f = f_a + f_m + f_{ma} + s_{fl}\\
&t_i = (f_a + f_m + f_{ma} + l_{fl} + s_{fl})/2\\
&t_m = l_{fl} + s_{fl}\\
&t_d = \text{total latency of loop-carried dependence}
\end{aligned}
$$

Figure 2.2: Performance model for loop-dominated code on the IBM RS/6000.

by deriving and using these bounds to evaluate and improve the performance of the IBM RS/6000 (Figure 2.1) on a set of scientific code kernels.

The run time of an application is bounded from below by the product of the minimum amount of time (number of clocks) that must be spent per inner loop iteration, t_l, and the number of iterations. This bound can be made fairly tight for loop-dominated scientific applications. For a highly concurrent machine, t_l is determined by first computing the minimum time that must be spent in each of several potential bottleneck units of the machine and then taking the maximum of these times. The bounds formulation for the IBM RS/6000 (Figure 2.2) identifies these times for four common bottleneck units:

- t_f for the floating-point unit, which is evaluated by counting the number of floating-point operations that appear in the high level application code for one inner loop iteration. These operations can be executed at best at the rate of one operation per clock in the RS/6000 Floating-Point Unit (FPU) [20]. In our workload, these are combinable multiply-add triad operations (f_{ma}), uncombinable adds (f_a), and uncombinable multiplies (f_m). Add and multiply operators that appear in the loop body are checked to see if they are combinable into one of the RS/6000's three-operand multiply-add instructions. Floating-point store instructions pass through the decode stage of the FPU. Thus *essential* stores (see t_m) are counted here as well.

- t_m for the memory port activity, which requires at least one clock per *essential* floating-point load (l_{fl}) and floating-point store (s_{fl}) [21]. Counting only essential loads and stores requires inter-iteration dependence analysis. For m iterations of the inner loop, the number of *distinct* array elements that appear on the left hand side of assignment statements will generally be of the form $am + b$. The number of *essential stores* is taken to be a. The number of *essential loads* is found similarly by examining the right hand side of each statement and counting distinct array elements that appear on the right before they appear on the left side of an assignment statement. Scalars should simply reside in registers during the loop.

- t_i for the instruction issue unit, which models the RS/6000 instruction dispatch bandwidth of the Instruction-Cache Unit (ICU). The issue unit can issue two floating-point arithmetic or memory reference instructions per clock. The RS/6000 can also issue a branch and a test instruction concurrently with the above, or other fixed-point instructions in place of memory instructions; however, since these are either concurrent with the operations counted above or tend not to limit performance in scientific loop-dominated code, they are not counted.

- t_d is used to model a fictitious "loop-carried dependence unit" which can bottleneck the performance of loops with recurrence, *i.e.* a result of one iteration depends on the corresponding result of a previous iteration. Whenever there is such a cycle in the dependence graph of the floating-point arithmetic operations, t_d is computed as the sum of the latencies of the operations in one tour of the longest cycle divided by the number of loop iterations in the cycle (usually 1); otherwise, t_d is 0. The *latency* of an operation is related to pipeline depth and is computed as the minimum number of clocks between issuing that operation and issuing a succeeding operation that uses its result as an operand.

Some source-code transformation techniques that can be used to improve performance, such as blocking for better memory or register locality [22], loop interchange [23, 24, 25], and cyclic reduction [26] (which reduces t_d by increasing the number of iterations in a recurrence cycle), tend to alter the "essential" workload. Such transformations must be viewed as creating a *different* application code for which the CPF bound would have to be recalculated.

When the t_l bound is achieved, the unit(s) selected by the max function is kept continuously busy by the operations that are counted. Other units, other operations, and other factors that may limit concurrency within and among these units are simply ignored. When these do inherently limit performance, the t_l bound will not be achievable in practice. The goal of this research and this case study is to evaluate performance relative to this simple bound, to find mechanisms for approaching the bound performance, and to explain the gaps between achievable and bound performance when they do occur. The first 12 Livermore Fortran Kernels [27] are used as the application workload for the experimental study in this chapter. Table 2.1 lists the essential workload and the bound for LFK1-12. The bound shown is in units of CPF (clocks per floating-point operation), which is computed as $t_l/(2f_{ma} + f_a + f_m)$. The LFK7 bound approaches the RS/6000 peak performance of 0.5 CPF. The average CPF is 1.269 which is equivalent to a harmonic mean of 19.7 MFLOPS on an RS/6000 with a 25 MHz clock rate. Harmonic Mean MFLOPS (the "average" performance if one does an equal number of floating-point operations in each application) = Clock Rate (in MHz)/Average CPF. In general, Clock Rate/CPF Bound is referred to as the "Performance Bound."

LFK	f_m	f_a	f_{ma}	s_{fl}	l_{fl}	t_f	t_m	t_i	t_d	BOUND (CPF)
1	1	0	2	1	2	4	3	3	0	0.80
2	0	0	2	1	4	3	5	3.5	0	1.25
3	0	0	1	0	2	1	2	1.5	0	1.00
4	0	0	1	0	2	1	2	1.5	0	1.00
5	1	1	0	1	2	3	3	2.5	4	2.00
6	0	0	1	0	2	1	2	1.5	0	1.00
7	0	0	8	1	3	9	4	6	0	0.56
8	0	6	15	6	9	27	15	18	0	0.75
9	1	2	7	1	10	11	11	10.5	0	0.65
10	0	9	0	10	10	19	20	14.5	0	2.22
11	0	1	0	1	1	2	2	1.5	2	2.00
12	0	1	0	1	1	2	2	1.5	0	2.00

Table 2.1: Workload and CPF bound of LFK1-12 on IBM RS/6000. (LFK5 has $t_d = 4$ clocks and LFK11 has $t_d = 2$, since their f_a and f_m form a loop-carried dependence and the latency of each operation is 2; the other loops have no recursion and $t_d = 0$.)

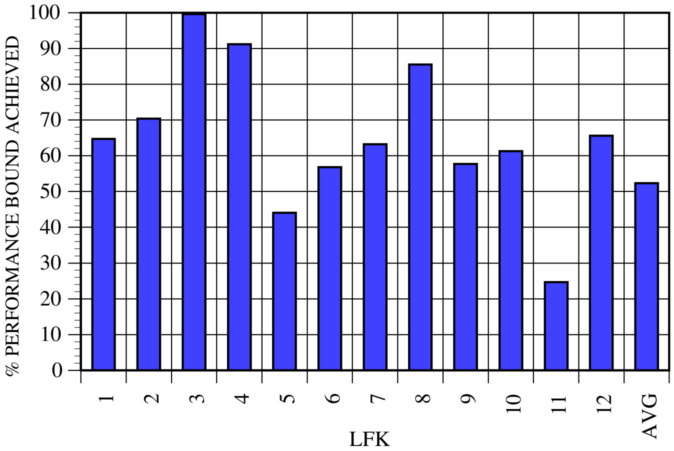


Figure 2.3: Measured performance vs. performance bound. (For IBM RS/6000 model 730 with compiler XLF release 2.01.)

LFK	From Source Code		In Compiled Code		
	Essential		Redundant		# RAW stalls
	s_{fl}	l_{fl}	s_{fl}	l_{fl}	
1	1	2	0	1	2
2	1	4	0	1	1
3	0	2	0	0	0
4	0	2	0	0	0
5	1	2	0	1	1
6	0	2	1	0	0
7	1	3	0	6	5
8	6	9	0	6	0
9	1	10	0	0	8
10	10	10	6	0	1
11	1	1	0	1	0
12	1	1	0	1	0

Table 2.2: Required vs. actual redundant memory operations, and RAW stalls of LFK1-12.

Figure 2.3 shows that actual measured performance for compiled code averages only 52.4% of architectural performance bound[1]. Only LFK3 and 4 achieve 90% or more of their performance bounds; the worst loop (LFK11) achieves only about one fourth of its bound.

2.3 Improving Delivered Performance

The gap between delivered and architectural (bound) performance has two causes: the compiler may fail to optimize the code well (resulting in poor performance) or the model may ignore some performance-limiting aspects of the architecture (resulting in an unattainable bound). By analyzing the compiled code, we have identified four major compiler-related issues that depressed the delivered performance: redundant loads and stores, read after write (RAW) dependence stalls, floating-point store vs. write-back conflicts [28], and a fine grain code scheduling problem. These issues are discussed in this section. Some model and measurement refinement issues are deferred to the subsequent section.

2.3.1 Remove redundant loads/stores

Redundant memory loads occur when data that are reused in later iterations are not saved in register files. Redundant stores occur when intermediate data are stored to the memory when only final data need be. These extra nonessential memory accesses add one clock each to the time required at the memory port (t_m), thereby degrading performance, especially for memory-bound loops. Redundant (actual - essential) loads/stores were found in the compiled inner loop body code

---

[1]

$$\% \text{ Performance Bound Achieved} = \frac{\text{Measured Performance (MFLOPS)}}{\text{Performance Bound (MFLOPS)}} \times 100\% = \frac{\text{CPF Bound}}{\text{Measured CPF}} \times 100\%$$

```
(a).Fortran Source Program
        DO 1 L = 1, Loop
           DO 1 K = 1, n
    1         X(K) = Q + Y(K) * (R*ZX(K+10) + T*ZX(K+11))

(b).Compiled Pseudo Code (the inner loop only)
    CL.2:
        lfd     fp0, ZX(K+10)                 [1]
        lfdu    fp2, ZX(K+11)                 [2]
        fm      fp2, fp3, fp2                 [3] fp3 = T
        lfd     fp4, Y(K)                     [4]
        fma     fp0, fp5, fp0, fp2            [5] fp5 = R
        fma     fp0, fp4, fp0, fp1            [6] fp1 = Q
        stfdu   fp0, X(K)                     [7]
        bc      CL.2

(c).Modified Pseudo Code (the inner loop only)
        lfd     fp0, ZX(K+10)                 [0]
    CL.2:
        fm      fp2, fp5, fp0                 [1] fp5 = R
        lfdu    fp0, ZX(K+11)                 [2]
        fma     fp2, fp3, fp0, fp2            [3] fp3 = T
        lfd     fp4, Y(K)                     [4]
        fma     fp2, fp4, fp2, fp1            [5] fp1 = Q
        stfdu   fp2, X(K)                     [6]
        bc      CL.2
```

Figure 2.4: Eliminating redundant loads: codes for example 1 (stfdu is a double precision floating-point store with address updated automatically).

of 9 of the 12 LFKs (Table 2.2). To assess their effect on performance, we applied two methods to remove them.

In the first method, the registers are re-allocated so that the content of the registers that store reusable data will not be changed until the data is used. Arithmetic operations are then recoded to use the data in these registers, and the redundant loads are discarded. Some initial loads may need to be inserted before the loop. The second method is used only in LFK12 where the first method is difficult to apply. The loop is unrolled twice and different registers are used for odd and even iterations. Redundant stores are simply removed and final stores are inserted after the loop. Since there is no official disassembler for the RS/6000, nor an executable assembly code output from the compiler, we used Goblin [29] to disassemble the compiled code, and performed the necessary modifications on that code. To illustrate the first method, consider the following example.

Example 2.3.1 Excise the redundant loads in the compiled code of LFK1
From the source code of LFK1 (Figure 2.4), it is evident that ZX(K+11) could be used as ZX(K+10) in the next iteration. However, the current compiler loads both of them in every iteration. The modified code reserves fp0 to hold the reusable data, ZX(K+11), for the first instruction of the following iteration, and eliminates the redundant load. Notice that the code

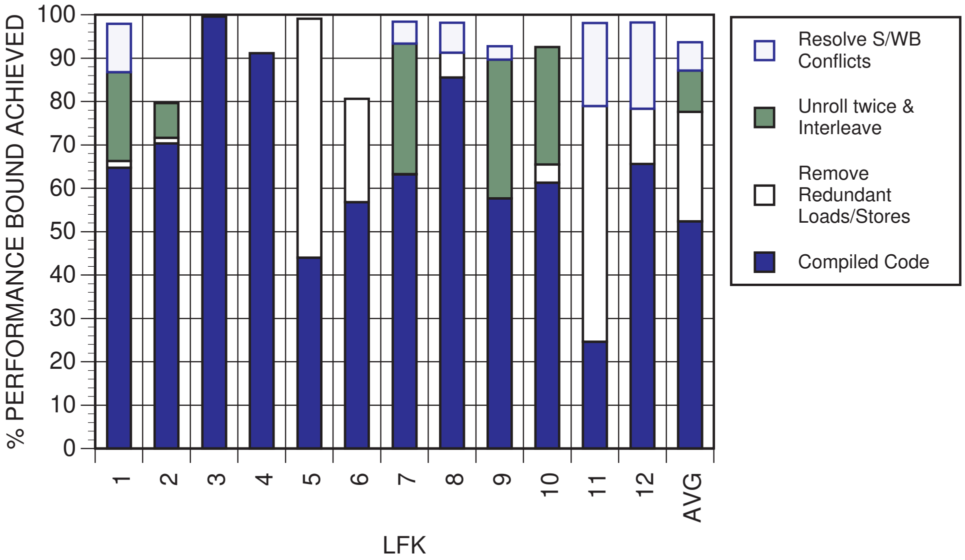


Figure 2.5: Step by step modification results of LFK1-12. The data were measured by running the codes on RS/6000 Model 730 with AIX 3.1.5 operating system and V2.01 compiler.

execution sequence had to be reordered and the initial value of ZX(K+10) had to be loaded before the loop begins. □

Usually, removing redundant memory operations increases performance. However, for loops such as LFK7 and 8, which have more floating-point arithmetic instructions than essential memory operations, eliminating all redundant memory operations does not guarantee higher performance. Moreover, when register reallocation is difficult, necessitating a large degree of unrolling or register moves through the FPU pipeline [30], the burden on the FPU can increase. Consequently, only one memory reference was removed in LFK7, and three in LFK8. All the redundant memory operations were removed in the other loops. The resulting performance improvement is shown in Figure 2.5.

2.3.2 Unroll Twice and Interleave

The next step is to reduce the stalls caused by RAW dependence among FPU arithmetic instructions.[2] In RS/6000, the second instruction of two consecutive dependent FPU arithmetic instructions will typically be stalled for one clock before entering the FPU pipeline (FPU pipeline latency for multiply-add is usually 2, sometimes more when data exceptions are possible) [28]. These stall clocks are added to the time, t_f, required in the FPU. Table 2.2 shows the number of such RAW stall cycles per iteration for the compiled code. For example, LFK9 with no redundant memory operations, suffers at least 8 RAW stall clocks due to this effect, which aggravates the FPU bottleneck. One independent FPU instruction placed between two consecutive dependent

---

[2] In IBM RS/6000, when an lfd (floating-point load double) and a dependent FPU arithmetic instruction are issued in the same cycle and the datum to be loaded is in cache, the operand will arrive before the arithmetic instruction needs it, and no stall cycle occurs. Thus we only consider dependent FPU arithmetic instructions.

FPU instructions will fill an otherwise wasted stall cycle in the pipeline. It may be possible to achieve this by reordering the instructions in the loop body. If not, one straightforward, systematic way to implement this for loops without loop-carried dependence is to unroll the loop twice and then interleave the instructions so that a successive pair of FPU arithmetic instructions from one iteration is separated by one FPU instruction from the other iteration. Unrolling more than twice may be required if data exceptions are suspected. The following example describes this approach. Similarly, we can use cyclic scheduling here [30, 31, 32, 33] (a form of software pipelining) as an alternative. In general, the RAW dependence stall cycles can be eliminated entirely, except when there are not enough registers for the new schedule, or there are not enough independent pairs of FPU operations in a recursive loop.

Example 2.3.2 Reduce the stalls caused by RAW hazards in LFK1 compiled code by unrolling the loop twice and interleaving the instructions.
In Figure 2.4.c, instruction 3 depends on the result of instruction 1, and instruction 5 depends on the result of instruction 3. Two RAW stalls result since there is no FPU instruction between 1 and 3 or 3 and 5. Unrolling and interleaving is possible since FPU instructions from different iterations are independent, *i.e.* there is no recurrence. It is not possible to accomplish this simply by unrolling the source code twice because the compiler does not interleave the instructions; it simply stacks two iterations of the code in Figure 2.4.c consecutively. We used Goblin to disassemble the compiled code and interleaved these instructions as follows:

```
          lfd     fp0, ZX(K+10)          [0]
CL.2:
          fm      fp2, fp5, fp0          [1] fp5 = R
          lfd     fp6, ZX(K+11)          [2]
          fm      fp4, fp5, fp6          [1U]fp5 = R
          lfdu    fp0, ZX(K+12)          [2U]
          fma     fp2, fp3, fp6, fp2     [3] fp3 = T
          lfd     fp6, Y(K)              [4]
          fma     fp4, fp3, fp0, fp4     [3U]fp3 = T
          lfd     fp7, Y(K+1)            [4U]
          fma     fp2, fp6, fp2, fp1     [5] fp1 = Q
          stfd    fp2, X(K)              [6]
          fma     fp4, fp7, fp4, fp1     [5U]fp1 = Q
          stfdu   fp4, X(K)              [6U]
          bc      CL.2
```

Instructions 1, 3, 5 from an odd iteration are interleaved with 1U, 3U, 5U from the next even iteration to form the execution sequence 1, 1U, 3, 3U, 5, 5U, thereby eliminating the two RAW stall cycles. □

2.3.3 Resolve Floating-Point Store (stf) vs. Write-Back Conflicts

A floating-point store instruction can conflict with a preceding FPU arithmetic instruction if the store is in the FPU decode stage while the arithmetic instruction is in the FPU write-back stage. In the absence of data exceptions, there is a one clock delay when the conflicting instructions pass through FPU decode three clocks apart [28]. For the workload in our experiments, this delay seems to occur only when the arithmetic destination and store registers are different. This conflict is very common: LFK1, 7, 8, 9, 11 and 12 each have at least one such conflict. These conflicts were eliminated by unrolling and interleaving three loop iterations, or by a similar cyclic scheduling technique, which can always place a dependent store or a nonstore instruction three clocks after an FPU arithmetic instruction. Although LFK10 also has this problem, it was not eliminated since overall performance was not improved.

2.3.4 Fine-Grain Code Scheduling

In a machine with multiple functional units, effective code schedulers try to reach the bound by keeping the bottleneck functional units busy in every clock cycle. In LFK1-12, the most relevant functional units in the RS/6000 are the Fixed-Point (FXU) and Floating-Point Units, since all the assembly instructions in the inner loop bodies are floating-point loads/stores and floating-point arithmetic instructions. These can be issued by the ICU at the rate of two per clock, but the FXU pipeline can only start execution of one load or store per clock and the FPU decoder (before the pipeline) can only process one floating-point store or arithmetic instruction per clock. Thus loads should be sufficiently interleaved with FPU arithmetic instructions so that the FXU and FPU instruction buffers do not fill up and stall the issue unit. This interleaving was in fact carried out each time we modified the code in previous steps.

2.3.5 Performance Improvement

Figure 2.5 shows the step by step performance gains from these code modifications of LFK1-12. The average performance was increased from 52.4% to 77.6% of the performance bound by eliminating redundant memory operations. This simple procedure (applied initially) contributed most to the average performance improvement. The largest gains are in LFK5 and 11, where the first lfd instruction of each iteration in the compiled code reloads the data value that was stored by the last instruction in previous iteration. These load operations are stalled until the data are written into the data cache from the Pending Store Queue. When these redundant loads were removed, performance was significantly improved. The second step elevated average performance to 87% of the performance bound by unrolling twice and interleaving to separate dependent FPU instructions. Although the net contribution of this procedure is less than that of the first step, several loops posted significant improvement gains. The final average performance achieved after the third step, which removes Store Write-Back conflicts, is 93.7% of the performance bound, representing a 1.79 times speedup over the original compiled code (Table 2.3). These hand coded improvements of the disassembled code were achieved by applying fairly general techniques to approach the performance bound, indicating that the RS/6000 compiler (release 2.01) could improve delivered performance significantly.

2.3.6 Preprocessor and New Compiler

We have used the preprocessor (V2.01) of the RS/6000 to see how much it can improve delivered performance. The preprocessor can be activated by turning on optimization switches or/and by inserting directives. In addition to the default switch settings [34, p. 53], the interesting switches include switch a (allows associative transformations), switch j (generates calls to Engineering/Scientific Subroutine Library routines), switch 2 (performs no dependence checking for arrays containing pointer variables), switch 4 (generates calls to BLAS library routines), switch 7 (inlines called routines automatically), and switch 9 (collapses array subscripts when inlining). After trying the following four switch combinations: {default}, {default, a}, {default, a, 4, 7}, and

LFK	Bound CPF	Actual Compiled	Hand Code	Speedup	%-Bound Hand Code
1	0.800	1.236	0.817	1.513	97.919
2	1.250	1.777	1.570	1.132	79.618
3	1.000	1.004	1.004	1.000	99.602
4	1.000	1.097	1.097	1.000	91.158
5	2.000	4.541	2.019	2.249	99.059
6	1.000	1.760	1.240	1.419	80.645
7	0.560	0.886	0.569	1.557	98.418
8	0.750	0.877	0.764	1.148	98.168
9	0.650	1.127	0.701	1.608	92.725
10	2.222	3.624	2.400	1.510	92.583
11	2.000	8.117	2.039	3.981	98.087
12	2.000	3.048	2.036	1.497	98.232
AVG	1.269	2.425	1.355	1.790	93.701
MFLOPS	19.701	10.309	18.450	1.790	93.701

Table 2.3: Hand code vs. compiled code performance (in CPF and MFLOPS).

{default, all of above}, we found the most effective switch for our LFK benchmark is switch a.

The closest directive to our own performance tuning techniques is "CVD$ UNROLL." By manually inserting the "CVD$ UNROLL" directive to selected kernels and turning on all of the aforementioned optimization switches, the best harmonic mean performance we obtained is 13.4 MFLOPS over the first 12 LFKs, as shown in Table 2.4 column V2.01(P). The performance gain is less than half of the hand code improvement primarily because the compiler does not effectively take advantage of the preprocessor's source code restructuring. It still fails to detect the redundant memory operations, and is not able to remove most of the RAW hazard stalls even when the basic block is enlarged by unrolling.

The new RS/6000 V2.02 compiler, even without the preprocessor, improves the harmonic mean performance to 14.28 MFLOPS (1.39x improvement over V2.01) for the same workload (Table 2.4). Most of this performance gain was achieved by removing the redundant load in each of LFK5 and 11. Since these loads were stalled due to the still pending stores to same locations, performance of these loops increased by 2.25x and 2.69x, respectively. Even though the V2.02 compiler eliminated 6 redundant stores and one RAW hazard in LFK10, its performance improved by only 10%. The other loops were not affected by more than 5% and some actually ran slower. Adding the V2.02 preprocessor gains another 1.10x improvement in the harmonic mean to 15.68 MFLOPS due to restructuring such as loop interchange and unrolling. However, unless the compiler can take advantage of the preprocessor by employing more sophisticated scheduling to remove stalls, the performance improvement that can be obtained by the preprocessor is limited. Moreover, although the V2.02 compiler is much improved over V2.01, it can still be further improved by dealing in a systematic and goal-directed manner with the problems cited in previous sections.

LFK	BOUND CPF	Hand Code	Compiled Code			
			V2.01	V2.01(P)	V2.02	V2.02(P)
1	0.80	0.82	1.24	1.21	1.22	1.24
2	1.25	1.57	1.78	1.83	1.75	1.78
3	1.00	1.00	1.00	1.01	1.02	1.04
4	1.00	1.10	1.10	1.06	1.05	1.07
5	2.00	2.02	4.54	2.64	2.02	2.04
6	1.00	1.24	1.76	1.56	1.78	1.61
7	0.56	0.57	0.89	0.74	0.88	0.74
8	0.75	0.76	0.88	0.88	0.83	0.77
9	0.65	0.70	1.13	0.74	1.13	0.86
10	2.22	2.40	3.62	3.86	3.28	2.69
11	2.00	2.04	8.12	3.79	3.02	2.30
12	2.00	2.04	3.05	3.03	3.03	3.02
AVG	1.27	1.36	2.43	1.86	1.75	1.60
MFLOPS	19.70	18.45	10.31	13.40	14.28	15.68

Table 2.4: Performance of LFK1-12 for the bound, hand-code, and different versions of compiled codes without and with the preprocessor.

2.4 Measuring Steady-State Inner Loop Performance

In Figure 2.5, the final performance achieved for some loops (LFK2 and 6, as well as 4, 9, and 10) is still noticeably less than the bound. To better bound their performance, we now consider effects previously ignored in the model. The model ignored cache miss effects, but these loops fit entirely in cache and no cache miss penalty was observed in our measurements. Even initial "cold start" miss effects are negligible since each code is executed many times repeatedly to get an accurate timing measurement. Other ignored effects, such as branch and iteration count overhead, ignored instruction types, certain dependencies, and hardware interlocks are expected to have little or no effect on the performance of scientific inner loops due to the success of the RS/6000 branch handling and other hardware design features in masking their effects through buffering and concurrent execution [35].

However, there is some outer loop overhead to compute the loop limit and load it into the count register before the loop begins and to execute other code, if any, that is not part of an inner loop body. Since outer loop run time can be amortized over all inner loop iterations, it can be ignored in most cases. However, if the number of inner loop iterations per outer loop iteration is small then the measured performance might include a nonnegligible outer loop overhead.

Since t_l is a lower bound on the steady-state run time of one inner loop iteration, it is interesting to compare the CPF bound against the delivered CPF of the inner loop in steady state. As it happens, each LFK has a parameter n and the average number of inner loop iterations per outer loop iteration is an increasing function of n. By measuring the actual CPF for a range of values of n and curve fitting, we found that the measured CPF can be approximated by

$$k \times n^{-h} + c \tag{2.1}$$

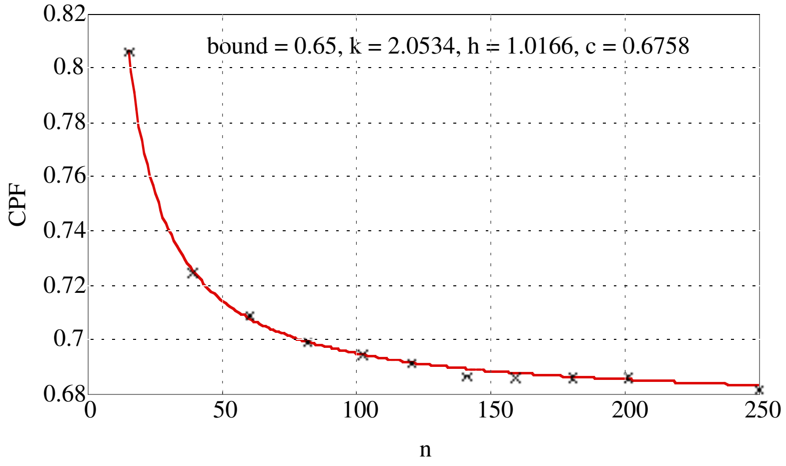


Figure 2.6: Example of curve fitting: Curve fitting by $k \times n^{-h} + c$ for LFK9 with 11 points of n ranging from 15 to 249.

as long as n is kept sufficiently small, *e.g.* so that cache misses do not suddenly change the shape of the function. The term $k \times n^{-h}$, when it fits the measurements well, summarizes all program effects that vary with the program parameter n, *i.e.* all actual run time not associated with steady-state inner loop performance. The constant term, c, is thus considered to be the actual measured CPF during steady-state execution of an inner loop iteration.

Figure 2.6 presents a typical example of the curve fitting, namely for LFK9, and Table 2.5 summarizes the k, h, and c values obtained from curve fitting for LFK2, 4, 6, 9 and 10. The relative errors of the curve fitting are small (less than 1.5%). Note that h is close to 1 and outer loop overhead is thus approximately inversely proportional to n.[3] Figure 2.7 displays the final achieved performance as in Figure 2.5, except that the measured CPF for LFK2, 4, 6, 9 and 10 are taken to be the values of c in Table 4. With this adjustment, every loop is seen to achieve at least 94% of the performance bound in steady state, and the average is 97.6%.

Thus for the RS/6000, this simple bound model is believed to characterize extremely well the achievable (architectural) steady-state performance for scientific inner loops, and c and $k \times n^{-h}$ are believed to characterize, respectively, the achieved steady-state inner loop performance and the run-time overhead of the complete scientific kernels, except for cache miss effects and register spilling which have not occurred here. The issue of register requirements is not within the scope of this dissertation; this problem is treated in [11, 36].

2.5 Cache Characterization

In our LFK experiments, all data arrays of the programs fit entirely within the IBM RS/6000 cache, but the data arrays in other applications or on other machines may not. Due to the progress of VLSI technology, cache system designs are becoming more complex and the memory subsystem descriptions provided by vendors are usually too simple to represent the real performance accurately. In this section, we present a simple methodology that can be developed further to predict real memory performance. This method uses the idea of load/store kernels to learn about

---

[3] For a simple loop with constant outer loop overhead incurred every n inner loop iterations, we would expect the actual measured CPF to be $(k + cn)/n$.

LFK	k	h	c	CPF Bound	CPF Bound/c
2	12.0283	0.7909	1.2586	1.250	99.3%
4	114.0077	1.0601	1.0175	1.000	98.3%
6	28.1210	1.2101	1.0564	1.000	94.7%
9	2.0534	1.0166	0.6758	0.650	95.8%
10	1.8532	0.9868	2.3608	2.222	94.1%

Table 2.5: The achieved and bound performance for LFK2, 4, 6, 9, and 10 using the derived k, h and c values of formula 2.1.

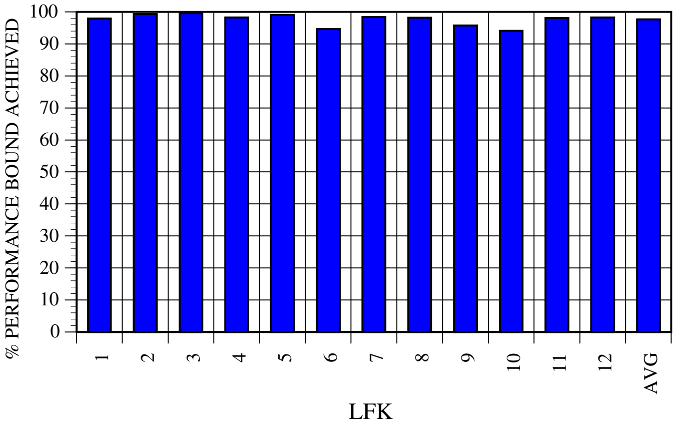


Figure 2.7: Achieved vs. bound performance for LFK1-12 with the measured CPF of LFK2, 4, 6, 9, 10 set to c from formula 2.1.

the cache structure and calibrate its performance [22]. The basic program used here is shown in Figure 2.8a. In this study, we ran the kernel many times to get steady state performance.

Figure 2.8b and 2.8c list one load kernel and one store kernel, respectively. The load kernel repeatedly loads data with a fixed stride from a 1-dimensional array into registers and performs some simple arithmetic and loop control which is guaranteed not to affect performance; while the store kernel simply repeatedly stores data with a fixed stride to a 1-dimensional array. The loop performance is totally memory bottlenecked. Two parameters, ARRAY_SIZE and STRIDE, were varied in order to obtain enough data for analyzing the cache characteristics.

2.5.1 Floating-Point Load Cache Performance

Figure 2.9 shows the cache performance as measured by the load kernel. It is evident that the average time required to load an 8-byte data word depends on the array size and stride. When ARRAY_SIZE is less than 8K, the average access rate is one clock cycle per data element, and we conclude that all referenced data is in cache (the *cache region* [22]). The RS/6000 Model 730 being measured in fact has a 64 KByte cache (= 8K × 8 bytes).

However, when ARRAY_SIZE > 8K, some misses occur (beginning of the *transition region* [22]).

```
(a).  Basic Program
      REAL*8 X(ARRAY_SIZE)
      :
      TS = MCLOCK()
      DO 1 L = 1, LOOPS
         ( LOAD/STORE KERNEL )
   1  CONTINUE
      TE = MCLOCK()

(b).  Load Kernel
      DO 1 I = 1, ARRAY_SIZE, STRIDE*4
         P = P + X(I)*X(I+STRIDE)
         Q = Q + X(I+STRIDE*2)*X(I+STRIDE*3)

(c).  Store Kernel
      DO 1 I = 1, ARRAY_SIZE, STRIDE*4
         X(I) = P
         X(I+STRIDE) = P
         X(I+STRIDE*2) = P
         X(I+STRIDE*3) = P
```

Figure 2.8: Cache calibration programs.

On cache misses, the IBM RS/6000 replaces the referenced line with the least recently used line in the set. If ARRAY_SIZE exceeds the cache size by less than the size of one full cache column (one line of each associative set), some data is never replaced. In this region, the effective access time grows as ARRAY_SIZE increases. Since the access time and hence the miss rate, levels off at ARRAY_SIZE $\geq$ 10K (the *memory region* [22]), we can conclude that a column is 2K $\times$ 8 bytes and that the cache must be 4-way set associative, as documented [37].

Let T_c, T_t, and T_m be the clocks per floating point load in the cache, transient, and memory regions, respectively. For a given stride, T_c and T_m are each constant. Suppose that the array, X, is aligned to a cache line boundary and that ARRAY_SIZE is a multiple of line size, which is the case in our experiments. Then, in the transient region, the value of T_t is determined by the formula:

$$\begin{aligned} T_t &= T_c \frac{\text{ARRAY\_SIZE} - (D+1)(\text{ARRAY\_SIZE} - C)}{\text{ARRAY\_SIZE}} + \\ & \quad T_m \frac{(D+1)(\text{ARRAY\_SIZE} - C)}{\text{ARRAY\_SIZE}} \end{aligned} \tag{2.2}$$

where C is the cache size, and D is the degree of associativity [38]. In the transient region, the cache capacity is exceeded by a number of elements that is less than one column of the cache, *i.e.* ARRAY_SIZE $- C < C/D$. Since the one dimensional array will be placed in contiguous cache space, the cache sets mapped to by these ARRAY_SIZE $- C$ elements are mapped to by $D+1$ lines of elements. Hence, the first reference to a new line in each of these cache sets will replace the least recently used line in the set, which is similar to the scenario in the memory region. The number of elements corresponding to these cache sets is $(D+1)(\text{ARRAY\_SIZE}-C)$. The remaining ARRAY_SIZE $-(D+1)(\text{ARRAY\_SIZE} - C)$ elements will all reside in the cache permanently, *i.e.* their sets are in the cache region. Consequently, we can represent T_t by Equation (2.2). T_t

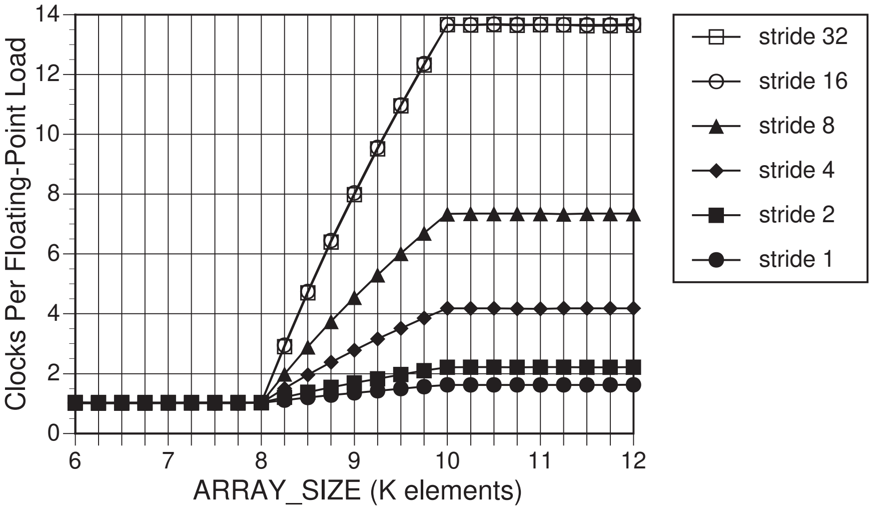


Figure 2.9: Memory performance for floating-point load(IBM RS/6000 model 730).

thus follows a parabolic curve that is convex between its two endpoints. This curve appears to be nearly linear for the cases in Figure 2.9.

Figure 2.9 also shows that the average miss penalty depends on STRIDE. Strides 16 and 32 have the same performance, implying that the cache line size must be $16 \times 8 = 128$ bytes, as documented for the Model 730. From this curve, we also conclude that cache misses can be served at a maximum rate of one miss per 13.6 clocks.

From the final level of the STRIDE = 1 curve, we see that 1 miss followed by 15 hits can be served per 25.9 ($= 10.9 + 15 \times 1$) clocks. Similarly for STRIDE = 2, 1 miss and 7 hits are served per 17.7 ($= 10.7 + 7 \times 1$) clocks. For STRIDE = 4, 1 miss and 3 hits are served per 16.7 ($= 10.7 + 3 \times 2$) clocks. And for STRIDE = 8, 1 miss and 1 hit are served per 14.7 ($= 10.7 + 1 \times 4$) clocks. Consequently, we conjecture that the referenced miss data is brought in first (with an average measured latency of 10.7 to 10.9 clocks) and that successive *pairs* of 8-byte data words in the miss line follow consecutively (2 words per clock to the cache, but at most 1 referenced word per clock to the processor). In IBM parlance, the initial miss reference penalty (*i.e.* miss latency - 1) is the *leading-edge effect* and the various penalties associated with subsequent references that are delayed because the miss has not yet been fully serviced constitute the *trailing-edge effect*. Neither the leading- nor trailing-edge effects are fully documented in the available literature.

2.5.2 Floating-Point Store Cache Performance

Figure 2.10 shows the cache performance as measured by the store kernel. For consecutive floating-point store instructions, the average access time on cache hit is 1.25 clocks per data element (5 clocks per 4 references). In the IBM RS/6000, for consecutive floating-point store instructions, some stall cycles may be introduced because of the limited size of the Pending Store Queue and the difference between the pipeline latencies of the FXU and FPU. This data reflects that the fifth continuous stf will be delayed for one cycle [28]. From this figure, we can conclude by an analysis similar to that for Figure 2.9 that the cache is 4-way set associative with a 128-byte line size and a 64 KByte capacity. The average number of clocks per floating point store depends on the stride and can be read directly from the figure, as listed below:

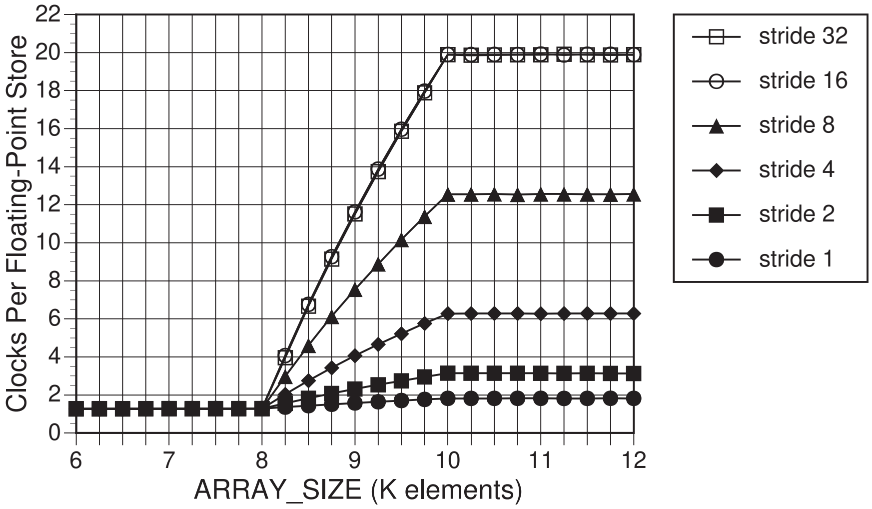


Figure 2.10: Memory performance for floating-point store (IBM RS/6000 model 730).

```
STRIDE = 1    1.8 clocks  (1 miss, 15 hits)
STRIDE = 2    3.1 clocks  (1 miss, 7 hits)
STRIDE = 4    6.3 clocks  (1 miss, 3 hits)
STRIDE = 8    12.6 clocks (1 miss, 1 hit)
STRIDE ≥ 16   19.9 clocks (1 miss)
```

Note that for this store kernel, every miss replaces a dirty line. Also the total time between misses is approximately 20 clocks for STRIDE $\geq$ 16, 29 clocks for STRIDE = 1, and 25 clocks for the other strides.

Mixtures of loads and stores and other stride patterns can also be characterized with corresponding kernels. This method is directly portable to other machines and leads to a straightforward experiment-based derivation of cache access bandwidth, latency, and inferred structure in the absence of full documentation. By characterizing an application's reference sequence, the results of this method can be used (for applications that do not operate entirely in the cache region) to add appropriate memory access latency effects to the bounds discussed above.

2.6 Conclusion

In this chapter, we have suggested a machine-application performance upper bound model for specifying reasonable performance objectives for scientific applications. This model focuses on the bandwidth provided by key machine units, as well as the specific bandwidth requirements of particular applications. Significantly, the upper bound performance resulting from the model was achievable for a set of scientific benchmark kernels on the IBM RS/6000. Achievable bounds help application programmers determine when, where, and how to invest effort to optimize code, by indicating what fraction of the available performance has already been achieved and can often indicate which factors in each region of code account for the remaining performance gap.

These bounds have proven useful for optimizing scientific loop codes on the IBM RS/6000. The RS/6000 is generally credited with good performance on many scientific codes, including the Livermore Fortran Kernels. However, the application-specific performance bounds developed here suggest that the RS/6000 actually loses a great deal of its potential performance, even though the achieved performance is often impressive. Guided by the performance bounds, a few straightforward code optimizations were able to raise the average efficiency on these codes from 52.4% to 93.7% of the bound, which represents a performance speedup of 1.79. Without the performance bounds, it would have been very difficult to decide which kernels to focus effort on, what optimizations to apply, and when to cease optimization work.

It is apparent that the heuristic approaches used in the RS/6000 Fortran compilers often unroll more or less than needed to mask pipeline latency, insert redundant memory accesses, and frequently fail to deliver the best possible performance. The efficiency of both the compilation process and the compiled code would be significantly improved if the compiler were cognizant of the application workload and the machine-application performance bound. It could then simply choose among program transformations and code schedules that attempt to keep the most heavily demanded resource continuously busy performing nonredundant operations. Furthermore, it could avoid wasting compile time seeking unattainable improvements and otherwise needlessly complicating the delivered machine code for no possible performance benefit, and occasionally actually degrading delivered performance.

We have also developed a tool that calculates the bound automatically for a given application loop. The tool employs a two step process. First, SIGMA, a tool kit for building parallelizing compilers and performance analysis systems, builds a database which contains needed information such as expression trees and dependence vectors [39]. Then, the tool backtracks through the expression tree to compute the number of essential floating-point operations, groups memory operations based on their dependence vectors and computes the number of essential load/stores, and computes the length of the maximally-weighted dependence cycle. It is then straightforward to calculate the workload of each of the four units and the performance bound of the loop for given machine parameters [12].

In general, performance is also affected by register limitations and cache misses. Register limitations have been treated in some other related work [36, 11]. Simple load/store kernel experiments are recommended in this chapter for characterizing cache structure and performance. Chapter 3 introduces an analytical methodology for constructing a more detailed cache behavior model.

CHAPTER 3

Application-Specific Loop-Based Cache Behavior Model

This chapter and the next chapter present a goal-directed approach for improving cache performance by reducing conflict misses. The focus is to derive a methodology for building application-specific cache models that can provide information about different types of cache misses and thus guide the selection of appropriate techniques for reducing them. This chapter studies the methodology for building the cache behavior models. Section 3.1 introduces some background and considerations for the methodology. Section 3.2 then contrasts our approach with some related work. In Section 3.3, we introduce some notation and preliminary discussion of some of the concepts used in this chapter. In Section 3.4, we illustrate our method for generating cache behavior models. Then, in Section 3.5, we use the method to build up a cache behavior model for a blocked matrix multiply program and evaluate the model by comparing its results with actual simulation results. Conclusions are presented in section 3.6.

3.1 Introduction

Performance degradation due to cache miss is a critical issue in program performance that becomes increasingly critical as the latency gap widens between processor operations and memory accesses. Several code transformation methodologies, such as loop fusion, loop interchange, and loop blocking [9], and data layout techniques, such as, array grouping [13], array padding, and copying [40], have been proposed for improving locality. Each of these techniques is capable of improving cache performance, but may be effective only for particular types of cache misses. Moreover, the same program may exhibit different cache behavior as a function of input data size, and thus applying any particular technique may not achieve its expected improvement [41]. As a result, selecting a suitable code or data restructuring technique is an extremely challenging task. We propose a methodology for building application-specific cache models that can provide information about different types of cache misses and thus guide the selection of appropriate techniques.

Cache misses in uniprocessors can be due to first time reference to data (*compulsory miss*), insufficient cache size (*capacity miss*), or references to lines that were replaced due to limited associativity or nonoptimal replacement (*conflict misses*) [6]. Because unexpected cache performance generally results from complicated conflict behavior, this dissertation focuses on conflict misses. We also limit this dissertation to array references, the dominant case in scientific code.

Three types of conflict misses are identified for arrays: *self conflict*, *cross conflict*, and *ping-pong conflict*. Ping-pong conflicts are best understood by considering a monotonic reference sequence to an array section, i.e. a sequence of references where the subsequence of references to a particular array section of interest exhibits a monotonically nondecreasing or monotonically nonincreasing sequence of addresses. In any monotonic reference sequence to an array section, all misses, except for those at the first reference within the sequence to each line of the array section, are called *ping-pong conflict misses*. Ping-pong conflict misses are considered separately as a special class because they can severely degrade performance. For example, Temam *et al.* present a program whose run time normally varies within 20% as the data layout in cache is changed, but can become 500% larger when ping-pong conflict misses occur [42]. The remaining conflict misses are either self conflicts or cross conflicts. Self-conflict misses for each particular array can be counted by imagining that only that array is allowed to reside in cache. They occur when the number of distinct lines of that array within one cache set is larger than the degree of associativity. All misses to this array, except those already counted as ping-pong conflict misses, that can be attributed to this phenomenon are called *self conflict misses*. If we now allow two or more arrays to compete for cache space, then the number of all conflict misses that are not already counted as either self conflict misses or ping-pong conflict misses are *cross conflict misses*.

Our model thus treats all three types of conflict misses and reports them separately for each array, which enables the selection of appropriate techniques in a direct manner. For example, in some cases, self, cross, and ping-pong conflict misses can be eliminated or reduced, respectively, by array padding, array grouping, and initial array address adjustment techniques. If we only know that a program has conflict misses, then we may simply have to try all of these techniques and search for some combination that appears to be most effective. If, however, we know that the misses are mostly self conflict misses, then we can immediately pick the array padding technique that is most appropriate for solving this problem. Of course, we must know which array is responsible for these misses so that the technique can be applied to the right array.

Our modeling method constructs cache models rather efficiently for a variety of cache organizations by using averaging concepts to determine the expected number of cache misses instead of attempting to count them exactly. In general, counting cross conflict misses is a highly complex problem. In particular, if the relative placement of the arrays in memory is changed, the cache sets referred to, and hence the references that refer to a particular set, and their sequences, will vary in a manner that is not easily determined. To make the assessment of cross conflict misses tractable, we do not use knowledge of the relative placement of the arrays in memory. Instead, we have developed an averaging technique that deals with the population of possible references to individual sets and their sequences and computes the average number of cross conflict misses over the application. This method provides a realistic assessment of expected average behavior, rather than a detailed assessment of each instance as it actually occurs in a trace.

One noticeable effect of using averages is that the method only needs basic cache parameters plus information that is extractable from the source code. Moreover, since our method characterizes all possible reference sequences regardless of their occurrence, and since for complicated programs, it is much more efficient to characterize the set of all possible reference sequences than to figure out which ones really occur and in what sequence for a particular application run, our method can thus dramatically reduce the time for creating cache behavior models for such programs. The efficiency and flexibility of the method make it a potential candidate for incorporation into a compiler to enable automatic performance tuning.

3.2 Related Work

Several technologies based on hardware monitoring, software profiling, simulation techniques, and analytical methods have been developed to help identify cache performance bottlenecks. For example, the hierarchical bound model MACS12*B [43] relies on hardware monitoring to acquire the actual number of cache misses in a program. It is precise but is limited to machines with special hardware. Mtool [1] uses a software profiling technique to count the number of instructions in each basic block, and estimates the ideal execution time based on these instructions. The difference between the actual and the estimated running time is then attributed to memory overhead. Regardless of how the results are presented, the tools that use these two types of techniques can only provide lumped results for actual cache misses. While this information may be adequate for identifying the bottleneck portion of an application, it is not detailed enough to guide the selection of appropriate techniques for cache performance tuning.

Cprof [2], MemSpy [3] and our model all identify the performance degradation caused by each array. In the uniprocessor environment, Cprof shows compulsory, conflict, and capacity misses while MemSpy classifies misses into first-load (compulsory) and replacement (conflict and capacity) misses. Both these tools consider conflict misses as a whole. Our model provides more fine grain information than these tools, further dividing conflict misses into self, cross, and ping-pong conflict misses, which is more instructive in selecting performance tuning techniques. In addition, Cprof and MemSpy are implemented based on trace simulation techniques rather than on source code analysis. Therefore, they are relatively slow and depend on the quality of the trace for capturing the relevant aspects of cache behavior.

Although analytical models may not be as precise as the aforementioned models, they can generally provide more insights regarding the relevant phenomena. As analytic models can generally be evaluated quickly, they are potential candidates for inclusion within a compiler to enable automatic performance tuning. Models derived by Lam *et al.* [40] and Temam *et al.* [42], as well as ours, belong to this category. Lam *et al.* classify cache misses into intrinsic, self conflict, and cross conflict misses, and derive a model accordingly. However, these misses are not evaluated for individual arrays. In addition, their model does not include ping-pong conflict misses. Temam *et al.* further distinguishes cross conflict into internal and external cross conflict according to whether the amount of overlap between two conflicting arrays will change as a function of the loop indices. If so, the cross conflict is classified as internal; otherwise, it is external. Although this method, which concentrates on overlap between arrays, is more detailed than Lam *et al.*, it is only suitable for direct mapped caches. This model also does not consider ping-pong conflict. Although similar, our model provides more detailed information than Temam *et al.* and is suitable for a variety of cache organizations.

3.3 Notation

We denote cache organization by a three-tuple (D, S, L), where D is the degree of associativity, S is the number of sets, and L is the line size in words (one word = one array element). For a set associative cache $(D > 1)$, we assume that LRU (Least-Recently Used) replacement strategy is employed. Given an application and a cache organization, there is an associated footprint of that application in each cache set. This *footprint* is defined as the set of distinct referenced lines of each array that are in that cache set, *i.e.* the set of lines that would be resident in that cache set after running the application if the cache had an infinite degree of associativity, *i.e.* (∞, S, L) [26].

Let $H_X(i)$ be the number of sets whose footprint contains exactly i lines of array X, and possibly some lines of other arrays. The function, $H_X(i)$ for $i = 0, 1, 2, \ldots$, is then the *footprint histogram* of array X. Given a cache organization, a footprint histogram is determined by the size and shape of a portion of the array, where the portion of interest is in turn determined by the addressing patterns among the various loop levels of a loop nest under consideration. For example,

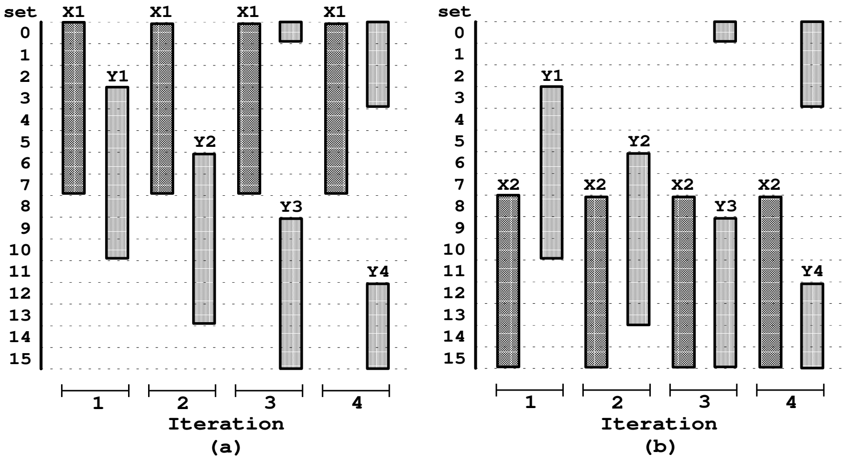


Figure 3.1: Data referenced in a 16-set cache for iterations 1 to 4 of a sample program. In both cases, the data set involves one section of array X ($X1$ or $X2$) and four sections of array Y ($Y1$, $Y2$, $Y3$, and $Y4$). The Y array sections begin at cache set 3, 6, 9, and 12, respectively. Elements of $X1$ begin at set 0 in (a), and $X2$ at 8 in (b).

consider the data referenced in 4 consecutive iterations (1 to 4) in a cache with 16 sets ($S = 16$) as shown in Figure 3.1a. In each iteration, the same X array section ($X1$) is referenced with a different Y array section (Y_1, Y_2, Y_3, or Y_4). Suppose further that each of the 5 sections ($X1$, $Y1$, $Y2$, $Y3$, $Y4$) includes 1 line in each of 8 consecutive sets and that all 40 lines are distinct. Then, the nonzero terms of $H_X(i)$ are $H_X(0) = 8$ and $H_X(1) = 8$; while those of $H_Y(i)$ are $H_Y(1) = 4$, $H_Y(2) = 8$, and $H_Y(3) = 4$.

We also define a *conflict footprint* (CF). We say that a set contains CF (i, j) with respect to array pair (X, Y) if the set contains an i-line footprint of X and a j-line footprint of Y, where $i \geq 0$, $j \geq 0$. The $i = j = 0$ case, however, is excluded since it implies an empty cache set, which is of no interest. Consequently, CF (i, j) with respect to (X, Y) in the above example is defined for $(i, j) = (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 2)$, and $(1, 3)$.

Although the footprint for a data set is easy to derive, two cache sets with the same footprint may cause a different number of cache misses, depending on the actual reference sequence. Define a *reference sequence* for a given cache set to be an ordered subset of all program references that contains exactly the array references to the given cache set. Furthermore, each reference in a reference sequence for a particular cache set is identified by the lower case of the name of the array[1] being referenced and a subscript that indicates which line of that array is being referenced. For convenience, for each such reference sequence and each array X, the subscripts are chosen as follows: mark the first reference to each distinct line of X in the reference sequence and assign subscript 1 to the first marked line, 2 to the second, etc. Note that for two arrays, X and Y, there is no relationship between x_i and y_i and no relationship between x_i for one cache set and x_i for some other cache set. This notation allows a concise representation for reference sequences.

For example, consider Figure 3.1a. Suppose that in each iteration the X and Y sections are

---

[1] In this dissertation, arrays are represented by upper case letters.

referenced alternately and consecutively from beginning to end, starting with the X array. To evaluate cache misses, we consider the reference sequence to each set separately. For simplicity, let's further assume that $L = 1$. Consider the reference sequence in set 3 for the layout shown in Figure 3.1a. We see that one line of X, x_1, and one line of Y, y_1, are referenced in iteration 1, but y_1 is referenced first since y_1 is closer to the beginning of the $Y1$ section than x_1 is to the beginning of the $X1$ section. Thus after iteration 1 the set 3 reference sequence is y_1x_1. In iterations 2 and 3, only line x_1 is referenced in set 3, thereby resulting in reference sequence, $y_1x_1x_1x_1$. In the last iteration, a new line from $Y4$, denoted as y_2, is referenced in set 3, but y_2 is referenced after x_1 in this iteration. Thus the complete reference sequence for this set is $y_1x_1x_1x_1x_1y_2$. Similarly, we can derive the reference sequence for cache set 6 as $y_1x_1y_2x_1x_1x_1$. The reference sequences of these two sets have footprint $\{y_1, x_1, y_2\}$ and are thus both associated with the (X, Y) conflict footprint CF (1, 2). Thus both have three compulsory misses; however, in a direct mapped cache, the second pattern also has a conflict miss when x_1 returns after y_2 is referenced. Clearly, footprints do not contain enough information for determining cache misses, but reference sequences contain more than needed.

For counting cache misses, we define a *track* as a reference sequence where consecutive references to the same line are merged. The tracks for the above two reference sequences are $y_1x_1y_2$ and $y_1x_1y_2x_1$, respectively. Since consecutively referencing the same line never causes additional cache misses, misses can be counted from either their reference sequences or their tracks. Note that one track may represent more than one reference sequence. For example, consider another alignment of the X section relative to the Y sections, as shown in Figure 3.1b. The reference sequences of cache sets 8 and 14 are $x_1y_1x_1y_2x_1x_1$ and $x_1x_1y_1x_1y_2x_1$, respectively. However, these two reference sequences will always have the same misses, and they are both represented by track $x_1y_1x_1y_2x_1$.

Lemma 3.3.1 Rule for determining cache miss of a line in a track
Given a track "$\cdots y_i \cdots y_i \cdots$" of a set in a cache (D, S, L) with LRU replacement strategy, if $\mathcal{L}$ is the set of distinct lines between one appearance of y_i and the next appearance of y_i, then the latter y_i is a miss iff $|\mathcal{L}| \geq D$. Note that y_i itself is not in $\mathcal{L}$.

(The proof of Lemma 3.3.1 is omitted since it is obvious.)

3.4 Methods for Building Cache Behavior Models

In this section, we discuss the basic concept of our method for computing the number of cache misses for a loop nest. Our cache behavior model includes all types of cache misses; however, we will especially focus on the three types of conflict misses, namely, ping-pong, self, and cross conflict misses. In Section 3.4.1, we describe the situation where ping-pong conflicts can occur. Then, in Section 3.4.2, we use an example to discuss how we deal with compulsory and capacity misses, and to illustrate how we compute the numbers of self and cross conflict misses.

3.4.1 When Will Ping-Pong Conflict Misses Occur?

Recall that, except for misses that occur at the first reference to each line, each other miss in a sequence that involves references to more than one array section is called a ping-pong conflict miss. In other words, if there are ping-pong conflict misses in a line, the first reference to that line may also be a miss but will be counted in some other category. For example, consider two array sections, Q_1 and R_1, where Q_1 contains the consecutively stored elements $Q(1)$, $Q(2)$, $Q(3)$, $Q(4)$, $Q(5)$, $Q(6)$, $Q(7)$, $Q(8)$, and R_1 contains the consecutively stored elements $R(1)$, $R(2)$, $R(3)$, $R(4)$,

$R(5)$, $R(6)$, $R(7)$, $R(8)$. Suppose that Q_1 and R_1 are referenced alternately and consecutively from beginning to end, *i.e.* $Q(1), R(1), Q(2), R(2), \ldots, Q(8), R(8)$. It is evident that ping-pong conflict misses of Q_1 and R_1 can occur only in a direct mapped cache since a line of Q can simultaneously reside in cache with any line of R if the cache is at least 2-way associative.

Define an element's location in a cache to be $l = sL + o$, where s is the cache set number (starting from 0) and o is the offset (in elements) to the preceding line boundary. Let δ be the minimum distance (in elements) between the starting locations of the two array sections in the cache (l_1 and l_2), where the minimum distance is defined as $\min\{(l_1 - l_2) \bmod SL, (l_2 - l_1) \bmod SL\}$. Then, even in a direct mapped cache, ping-pong conflict misses will not occur if $\delta \geq L$. Assume a direct mapped cache with line size $L = 4$ for all examples in Figure 3.2. Figure 3.2a shows an example with $\delta = 4 = L$. In this case, when the referenced elements of $Q1$ are in one cache set, the corresponding $R1$ elements being referenced are in another cache set. Each line of these two array sections is kept in the cache between the first and last reference to it, and thus only causes a compulsory miss. Likewise, when $\delta > L$ each line remains in cache as long as it is needed and there are no ping-pong conflict misses.

Consequently, if these two array sections have ping-pong conflict misses, the cache must be a direct mapped cache and δ must be less than L. Suppose first that array sections $Q1$ and $R1$ are aligned to cache line boundaries. Then, as shown in Figure 3.2b, the two array sections can cause ping-pong conflict misses only when they begin in the same cache set, *i.e.* $\delta = 0$. The reference sequence in each of the two cache sets mapped to by elements of $Q1$ and $R1$ will be of form $q_1 r_1 q_1 r_1 q_1 r_1 q_1 r_1$. Every reference is a cache miss. Excluding the first miss to each cache line, there are $(L - 1) = 3$ ping-pong conflict misses per line, which results in a total of $3 \times 2 = 6$ ping-pong conflict misses for each array section.

The above case, namely two line-boundary-aligned array sections that begin in the same set, will generate the most ping-pong conflict misses. In other cases, the number of ping-pong conflict misses per line will be less. Figure 3.2c shows an example in which array sections Q_1 and R_1 are not aligned to cache line boundaries, and $\delta = 3 < L$. In this case ping-pong conflict misses occur at $R(3)$, $Q(4)$, $R(7)$, and $Q(8)$. The 0 or 1 ping-pong conflict misses per line is significantly less than the maximum case in the aligned example above.

Although up to now we have only discussed cases with consecutive references, *i.e.* stride = 1, ping-pong conflict can also occur for nonunit stride cases if the stride is less than L. The stride must be less than L since otherwise each line would have at most one reference. Even if this reference caused a cache miss, it would not be counted as a ping-pong conflict miss. Figure 3.2d illustrates a case with stride = 3. In this case, only the second reference in the second line of $R1$, namely at $R(7)$, is a ping-pong conflict miss.

Each example in Figure 3.2 involves two array sections, and these array sections are referenced from their first elements with the same stride. Similar analyses can be done for three or more such array sections. Ping-pong conflicts can occur among multiple array sections with any of a variety of unusual reference patterns and section alignments; however, we restrict our discussion here to the case of several array sections chosen such that i) the first reference of interest to each section is to the first element of the section and ii) these sections are referenced in a strictly interleaved (round-robin) fashion with the same stride. Ping-pong conflict misses in a cache with degree of associativity D will occur among such a set of array sections only if at least $D + 1$ of these array sections have the property that for each array section in this group there is another array section in the group such that the δ of this pair of sections is less than L. Suppose there are only D array sections that satisfy the aforementioned property. Then, when elements of these D arrays are referenced, the corresponding element referenced in any array section that is not in this group of D array sections must be in a different cache set. Thus since D lines can simultaneously reside

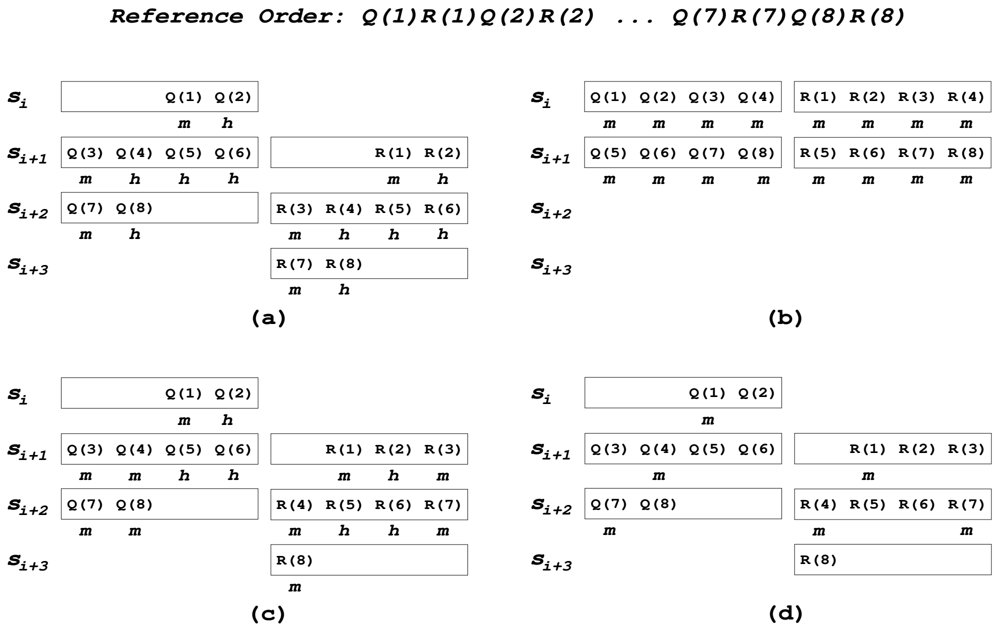


Figure 3.2: Ping-pong conflict misses between array sections $Q1$ and $R1$: (a) $\delta = 4 = L$. This case causes no ping-pong conflict misses. (b) $Q1$ and $R1$ are aligned to cache line boundaries and begin in the same cache set, s_i. This case causes 6 ping-pong conflict misses per array section. (c) $Q1$ and $R1$ are not aligned to cache line boundaries and $\delta = 3$. This causes 2 ping-pong conflict misses per array section. (d) Same as (c) but with stride 3, which causes no ping-pong conflict misses for $Q1$, and 1 ping-pong conflict miss for $R1$.

in a cache set, this group has no ping-pong conflict misses in a D-way set associative cache. If, on the other hand, there is a set of at least $D + 1$ such array sections, there is a possibility of ping-pong conflict.

Pseudo code for counting ping-pong conflict misses of a blocked matrix multiply program is found in Figure 3.4. This procedure needs to consider only 2 aligned array sections, but it can be generalized for more than 2 array sections and for array sections that are not aligned on cache line boundaries.

3.4.2 Loop Cold Start, Self Conflict, and Cross Conflict Misses

Unlike ping-pong conflict misses, the computations of the other types of cache misses are based on the notion of a *coverage loop*. The *coverage loop* is the outermost loop of a set of innermost loops for which we analyze the cache miss patterns. A loop nest of interest may have many loop levels. Let level 1 be the innermost loop, level 2 the next higher level, and so on. In practice, we may only need to analyze the cache miss patterns for the innermost c loop levels, *i.e.* the miss patterns may simply repeat for levels higher than level c. To determine the total number of misses for the loop nest, we can then simply multiply the results of the analysis by the number of entries to the c level loop nest caused by iterations of the outer loops. Note, however, that c cannot be chosen at too low a level, otherwise the analysis may not cover the full range of cache miss patterns; if c is chosen at too high a level, a lot of redundant work is done in the analysis. We call the level c loop the *coverage loop* and the level $c-1$ loop the *inner loop*, and call their iterations the *coverage loop iterations* and *inner loop iterations*, respectively. Loop levels, if any, below the inner loop are not given specific names.

As a running example to illustrate the analysis method, consider a loop nest with the outermost loop being the coverage loop and a cache with $(D, S, L) = (2, 16, 4)$. Suppose that the placements of data referenced in the odd and even numbered coverage loop iterations are like Figures 3.1a and 3.1b, respectively. Each coverage loop iteration consists of four inner loop iterations, and each inner loop iteration references the same X array section and a different Y array section. Note that there may be more levels below the inner loop, *e.g.* to iterate over the elements of the sections. Further assume that the X array section accessed in one coverage loop is not accessed in any other coverage loop iteration; however, the same four Y array sections, $Y1$, $Y2$, $Y3$, and $Y4$, are reused in every coverage loop iteration. An exact analysis of Figure 3.1b would be very similar to the analysis of Figure 3.1a, but has a somewhat different population of conflict patterns. Our method considers the full set of coverage loop iterations and analyzes them as a group.

Note that once ping-pong conflict misses have been evaluated, we can evaluate all other types of cache misses by looking only at the first reference to each line in each inner loop iteration. Hence, in any context where only these other types of cache misses are discussed, we can use a *condensed reference sequence* for each inner loop iteration that contains only its first reference to each line. We thus use condensed reference sequences in the following discussion, and generally refer to them simply as "reference sequences" unless we need to emphasize the difference.

Loop Cold Start Misses

Our model assumes that data that is reused in successive coverage loop iterations may remain in the cache from iteration to iteration, and the analysis will assess this likelihood. The analysis also evaluates misses for data that are referenced within one coverage loop iteration. However, when data is referenced for the first time in an iteration and was not referenced in the immediately preceding iteration, no further analysis is done and this reference is simply assumed to be and counted as a *loop cold start miss*. This assumption is reasonable in the current example since the size of the five sections ($X1$, $Y1$, $Y2$, $Y3$, $Y4$) of the data set referenced in a coverage loop iteration is greater than the cache size. A loop cold start miss may be either a compulsory or a capacity miss from the viewpoint of the entire loop nest or the entire application.

The loop cold start misses in the first iteration involve loading all five array sections, while those in each of the following iterations involve loading only one (X) array section since the four Y array sections are reused. As a result, loop cold start misses of Y only exist in the first coverage loop iteration, and amount to $4 \times 8 = 32$ misses. On the other hand, loop cold start misses of X occur in every coverage loop iteration with 8 misses per iteration. Let $M_L(X)$ and $M_L(Y)$ be the number of loop cold start misses of X and Y per coverage loop iteration, respectively. Assume that there are N coverage loop iterations, then $M_L(X) = 8$, and $M_L(Y) = 32/N$.

Self Conflict Misses

Self and cross conflict misses are counted excluding the loop cold start and ping-pong conflict misses. Since the first coverage loop iteration involves many loop cold start misses, we consider self and cross conflict misses in other iterations, which are referred as *steady-state* iterations. In this section and the next section, we illustrate the computation of the average number of self and cross conflict misses per steady-state iteration, respectively. To do so, our method requires the footprint histogram of each array. Since a footprint histogram is derived based on the sub-array size and shape determined by loops below the coverage loop, it should be the same for every coverage loop iteration. The sizes and shapes of arrays X and Y in this example are shown in Figure 3.1, and the footprint histograms of these two arrays were derived in Section 3.3. The nonzero terms are $H_X(0) = 8$, $H_X(1) = 8$, $H_Y(1) = 4$, $H_Y(2) = 8$, and $H_Y(3) = 4$. Once we know the number of self conflict misses per steady-state iteration for each array, they are added to get the total.

We compute self conflict misses of an array in a cache by assuming that the cache contains only this array. Thus, self conflict misses can occur only if (1) the array is reused in the coverage loop since otherwise the misses will all be loop cold start misses, and (2) at least one cache set footprint has more than D distinct lines of the array since otherwise there will be no self conflict miss (Lemma 3.3.1). Moreover, if a cache set footprint for this array has j lines, where $j > D$, then since these lines are referenced in sequence and since the cache employs LRU replacement strategy, this array will have j self conflict misses associated with this cache set.

Let $M_S(A)$ be the number of self conflict misses of array A per steady-state coverage loop iteration. If either one of the above conditions is violated, then $M_S(A) = 0$. Otherwise, both conditions are satisfied and

$$M_S(A) = \sum_{j=D+1}^{\text{max footprint}} j \cdot H_A(j) \tag{3.1}$$

where $H_A(j)$ is the footprint histogram of A and indicates the number of sets that have a j-line footprint of A. As a result, $M_S(X) = 0$ since the X array section is not reused in the coverage loop, and

$$M_S(Y) = \sum_{j=D+1} j \cdot H_Y(j) = \sum_{j=3}^{3} j \cdot H_Y(j) = 3 \cdot 4 = 12$$

Similarly for a direct mapped cache, $D = 1$, $M_S(X) = 0$ and $M_S(Y) = 2 \cdot 8 + 3 \cdot 4 = 28$. For $D \geq 3$, $M_S(X) = M_S(Y) = 0$.

Cross Conflict Misses

In general, counting cross conflict misses is a highly complex problem. In particular, if the relative placement of the arrays in memory is changed, the cache sets referred to, and hence the references that refer to a particular set, and their sequences, will vary in a manner that is not easily determined. To make the assessment of cross conflict misses tractable, we do not use knowledge of the relative placement of the arrays in memory. Instead, we consider the population of all possible references to individual sets and their sequences, and compute the average number of cross conflict misses over the application. This section describes the assumptions and the process for computing the average number of cross conflict misses.

References to a cache set can be summarized by its conflict footprint. However, a cache set can have many possible conflict footprints. Let $Pr[CF(i,j)]$ be the probability that a given cache set has footprint CF (i,j). To compute $Pr[CF(i,j)]$, we assume that any i-line footprint of an array has a uniform probability of being in any cache set, *i.e.* the probability that a given i-line footprint maps to a particular cache set is equal to $1/S$, where S is the number of cache sets. Hence if there are $H_X(i)$ i-line footprints of X and $H_Y(j)$ j-line footprints of Y, the probability that a given cache set can be mapped to by both an i-line footprint of X and a j-line footprint of Y, CF (i,j) for (X,Y), equals $\frac{H_X(i)H_Y(j)}{S^2}$, *i.e.*

$$Pr[CF(i,j)] = \frac{H_X(i)H_Y(j)}{S^2} \tag{3.2}$$

Therefore, the expected number of cache sets that have CF (i, j), $n_s(i, j)$, can be computed by the following formula

$$n_s(i,j) = SPr[CF(i,j)] = \frac{H_X(i)H_Y(j)}{S} \tag{3.3}$$

A given conflict footprint may have several tracks associated with it. Let $T(i, j)$ be the set of all possible tracks associated with CF (i, j), and $Pr[t|CF(i, j)]$ be the probability of track t given CF (i, j). Since we also know that for a given track, there can be many reference sequences associated with the track, we derive $Pr[t|CF(i, j)]$ based on the relative population of reference sequences. Because we do not have *a priori* knowledge favoring the distribution of these reference sequences, we assume that each of these reference sequences is equally likely. Let $n(t)$ be the number of reference sequences for track t. Then, for a given track t, $Pr[t|CF(i, j)]$ is simply equal to the number of reference sequences associated with t over the total number of reference sequences associated with CF (i, j), *i.e.*

$$Pr[t|CF(i,j)] = \frac{n(t)}{\sum_{t_k \in T(i,j)} n(t_k)} \tag{3.4}$$

Therefore, the probability that a given cache set has track t of CF (i, j), $Pr[t, CF(i, j)]$, can be derived as follows:

$$\begin{aligned} Pr[t, CF(i,j)] &= Pr[CF(i,j)]Pr[t|CF(i,j)] \\ &= \frac{H_X(i)H_Y(j)}{S^2} \frac{n(t)}{\sum_{t_k \in T(i,j)} n(t_k)} \\ &= \frac{n_s(i,j)}{S} \frac{n(t)}{\sum_{t_k \in T(i,j)} n(t_k)} \end{aligned} \tag{3.5}$$

When it is clear from context that we are only discussing a particular conflict footprint, we simply use $Pr[t]$ for $Pr[t, CF(i, j)]$.

Note that different tracks may cause different numbers of cross conflict misses. We let $m_A(t)$ be the average number of cross conflict misses of array A for track t. Then, the expected number per set of cross conflict misses of array A related to CF (i, j), $m_A(i, j)$, can be computed from

$$\begin{aligned} m_A(i,j) &= \sum_{t \in T(i,j)} m_A(t)Pr[t] \\ &= \sum_{t \in T(i,j)} m_A(t) \frac{n_s(i,j)}{S} \frac{n(t)}{\sum_{t_k \in T(i,j)} n(t_k)} \\ &= \frac{n_s(i,j)}{S} \sum_{t \in T(i,j)} \frac{m_A(t)n(t)}{\sum_{t_k \in T(i,j)} n(t_k)} \end{aligned} \tag{3.6}$$

Finally, we can compute the average number of cross conflict misses per steady-state coverage loop iteration for array A, $M_C(A)$, as

$$M_C(A) = S \sum_{i,j} m_A(i,j) \tag{3.7}$$

For each array A, the total number of cross conflict misses is simply equal to the product of $M_C(A)$ by the number of steady-state coverage loop iterations. Summing this number over all arrays, we can get the total number of cross conflict misses for the application.

The key question for computing the number of cross conflict misses is how to compute $m_A(i,j)$ for each CF (i,j). Equation (3.6) shows that $m_A(i,j)$ is a function of $n_s(i,j)$ and $T(i,j)$, as well as $n(t)$ and $m_A(t)\ \forall\ t \in T(i,j)$. Equation (3.3) above describes how to compute $n_s(i,j)$. We now discuss how to compute the other parameters by using the computation of the cross conflict misses for CF (1, 2) as an illustrative example. Note that different array reference behaviors may require different methods to compute $T(i,j)$ and $n(t)$; however, the general issues and methods described here are still valid.

Find all possible tracks, $T(i,j)$ The general process for finding all possible tracks for a conflict footprint is as follows: (1) analyze how the arrays are referenced, (2) derive a necessary and sufficient set of properties that a track must have, and (3) list all tracks that comply with the properties. For the purpose of illustration, we first use a straightforward method to derive the properties of the tracks for CF (1, 2) in the example of Figure 3.1. We then show that the same properties can be obtained by analyzing the program.

Let the three lines of the cache set with conflict footprint CF (1, 2) with respect to array pair (X,Y) be x_1, y_1, and y_2. Consider the two inner iterations in which lines y_1 and y_2 are referenced. Suppose these two iterations are iterations i_1 and i_2, respectively, where $i_1 < i_2$. Note that x_1 is referenced in every iteration. Therefore, iteration i_1 itself has track x_1y_1 if x_1 is closer to the beginning of the X section than y_1 is to the beginning of the Y section; otherwise it has track y_1x_1.[2] Similarly, iteration i_2 itself may have track x_1y_2 or y_2x_1. Obviously, the track for each of the remaining inner iterations, or concatenations of such iterations, is x_1. Let e be the empty track, and note that the prior or post concatenation of e to a nonempty track t, et or te, is simply t.

We now begin to derive the elements of $T(1,2)$. First, consider all possible tracks for iterations from 1 to i_1. If $i_1 \neq 1$, then the track just prior to entering iteration i_1 is x_1; otherwise, it is the empty track, e. Thus, we can merge tracks x_1 and e with the tracks for iteration i_1, x_1y_1 and y_1x_1, and combine consecutive x_1 elements to get all of the possible tracks for the iterations through i_1. The merge operation is shown in the following equation:

$$\left\{ \begin{array}{c} e \\ x_1 \end{array} \right\} \times \left\{ \begin{array}{c} x_1y_1 \\ y_1x_1 \end{array} \right\} = \left\{ \begin{array}{c} x_1y_1 \\ y_1x_1 \\ x_1y_1x_1 \end{array} \right\} \tag{3.8}$$

The right hand side of the equation contains the result: x_1y_1, y_1x_1, and $x_1y_1x_1$. Similarly, since the track between iterations i_1 and i_2 (i_1 and i_2 are excluded) is x_1 if $i_2 > i_1 + 1$ and e otherwise, we can derive all of the possible tracks for iterations 1 through $i_2 - 1$ by

$$\left\{ \begin{array}{c} x_1y_1 \\ y_1x_1 \\ x_1y_1x_1 \end{array} \right\} \times \left\{ \begin{array}{c} e \\ x_1 \end{array} \right\} = \left\{ \begin{array}{c} x_1y_1 \\ y_1x_1 \\ x_1y_1x_1 \end{array} \right\} \tag{3.9}$$

This equation generates no new track patterns. Applying this operation again between these tracks and the tracks in i_2, we can then obtain all of the possible tracks for iterations 1 through i_2. These tracks are shown in the right hand side of Equation (3.10).

---

[2] In general, x_1 and y_1 could be equally far from the beginning of their sections and which of these two tracks results depends on the program order of reference to X and Y. Note that in this case, ping-pong conflict misses may occur if $D = 1$ and $L \geq 2$. This case does not occur in this example.

$$\left\{ \begin{array}{l} x_1y_1 \\ y_1x_1 \\ x_1y_1x_1 \end{array} \right\} \times \left\{ \begin{array}{l} x_1y_2 \\ y_2x_1 \end{array} \right\} = \left\{ \begin{array}{l} y_1x_1y_2 \\ y_1x_1y_2x_1 \\ x_1y_1y_2x_1 \\ x_1y_1x_1y_2 \\ x_1y_1x_1y_2x_1 \end{array} \right\} \tag{3.10}$$

Finally, consider the possible tracks for iterations after i_2. The track is e if i_2 is the last iteration, and x_1 otherwise. From Equation (3.11), we have $T(1,2) = \{y_1x_1y_2,\ y_1x_1y_2x_1,\ x_1y_1y_2x_1,\ x_1y_1x_1y_2,$ and $x_1y_1x_1y_2x_1\}$. These tracks are also listed in the first column of Table 3.1.

$$\left\{ \begin{array}{l} y_1x_1y_2 \\ y_1x_1y_2x_1 \\ x_1y_1y_2x_1 \\ x_1y_1x_1y_2 \\ x_1y_1x_1y_2x_1 \end{array} \right\} \times \left\{ \begin{array}{l} e \\ x_1 \end{array} \right\} = \left\{ \begin{array}{l} y_1x_1y_2 \\ y_1x_1y_2x_1 \\ x_1y_1y_2x_1 \\ x_1y_1x_1y_2 \\ x_1y_1x_1y_2x_1 \end{array} \right\} \tag{3.11}$$

Each of these tracks has exactly one y_1 and one y_2 and they exhibit the common property that each y element is adjacent to an x_1. Furthermore, the five tracks shown for CF (1, 2) are all the tracks that satisfy this property. This property can be also derived by analyzing the reference behavior of the arrays. Whenever a Y line is accessed, the x_1 line is also accessed in that iteration, though it may be earlier or later. Thus, in all tracks an x_1 must appear adjacent to each y element. Hence, tracks such as $x_1y_1y_2$ and $y_1y_2x_1$ are impossible. Tracks for the other conflict footprints of this example can be derived based on this property. In Appendix A.1.1 and Appendix A.1.2, we show cases where binary trees are used to list all possible tracks of all conflict footprints systematically based on the given track properties.

Count the number of reference sequences, $n(t)$, for each track Since an element in a track represents a string of one or more consecutive such elements in a corresponding (condensed) reference sequence, the number of (condensed) reference sequences associated with a track is determined by the total number of each such element in a (condensed) reference sequence and in the track.

In our example, since within a coverage loop iteration all references to each Y line occur within one inner loop iteration, a y element with a given subscript can appear only once in a (condensed) reference sequence, *i.e.* each y element in a track represents exactly one y element in the corresponding reference sequence. Hence, in this example, $n(t)$ is not affected by y elements. However, since x_1 is referenced in every inner iteration, then for B inner iterations ($B = 4$ here), there are B x_1 elements in each reference sequence. When the reference sequence is converted into a track, some x_1 elements may be merged. Hence, the number of reference sequences associated with a track having h x_1 elements is equal to the number of different ways of dividing a sequence of B identical objects into h subsequences; this amounts to the number of the ways of placing $h-1$ cuts in different places among the $B-1$ spaces between objects, *i.e.*

$$n(t) = \mathbf{C}_{h-1}^{B-1} = \frac{(B-1)!}{(B-h)!(h-1)!} \tag{3.12}$$

Each reference sequence formed by this way is legal since its corresponding track is legal. In fact, for any such reference sequence, we can find a data placement of the arrays in the cache such that some cache set will experience this reference sequence. The number of reference sequences associated with each track of conflict footprint (1, 2) is listed in the second column of Table 3.1.

Possible Track, t		$n(t)$	$Pr[t]$	$m_X(t)$	$m_Y(t)$
t_1	$y_1x_1y_2$	1	1/52	0	3/2
t_2	$y_1x_1y_2x_1$	3	3/52	0	3/2
t_3	$x_1y_1y_2x_1$	3	3/52	1	2
t_4	$x_1y_1x_1y_2$	3	3/52	0	2
t_5	$x_1y_1x_1y_2x_1$	3	3/52	0	2

Table 3.1: Tracks with the corresponding numbers of misses and reference sequences for conflict footprint (1, 2) with respect to (X, Y).

After we compute $n(t)$ for all $t \in T(i,j)$, we can then compute $Pr[t]$ according to Equation (3.5) as $n(t)/52$. The 52 results from the fact that $n_s(1,2)/S = H_X(1)H_Y(2)/S^2 = (8 \times 8)/(16 \times 16) = 1/4$, and the total number of reference sequences for $T(1,2)$, $\sum_{k=1}^{5} n(t_k)$, equals 13. The resulting values of $Pr[t]$ for $t \in T(1,2)$ in this example are listed in the third column of Table 3.1.

Count the number of misses, $m_A(t)$, for each track Given a track, the main issue that affects how we compute the number of cross conflict misses is the reusability of arrays in the coverage loop. Regarding reusability, a pair of arrays can have one of three combinations: both reused, neither reused, or one reused and the other not. In this example, arrays X and Y belong to the third case, where Y is reused in successive coverage loop iterations; while array X is not. The discussion below thus includes both types of arrays; therefore, the type of analysis for this example can also be applied to the other two cases[3].

If an array is not reused in successive coverage loop iterations, then since a track corresponds to references in one coverage loop iteration, the misses of this array in track t can be solely determined by track t itself. In addition, the first element of this array in t must be a loop cold start miss. For our example, array X is not reused in the coverage loop. Hence, to compute $m_X(t)$ we only need to consider track t. Since the first x_1 in each track is a loop cold start miss, cross conflict misses for X are the misses caused by the remaining x_1 elements. Since every y element must be adjacent to an x_1 element in a track, two x_1 elements with no x_1 between them can be separated by at most two distinct y elements. According to Lemma 3.3.1, we know that the second x_1 can be a miss only if $D = 1$ or 2. In our example, $D = 2$ and CF = (1, 2), the only track where the second x_1 is a miss is $x_1y_1y_2x_1$, and each such track causes one cross conflict miss of X. The results of CF (1, 2) are listed in the fourth column of Table 3.1.

However, if an array, A, is reused in successive coverage loop iterations, then the calculation of cross conflict misses of this array in track t has to consider not only track t but also the track immediately before t. Let $m_A(t|t')$ be the number of cross conflict misses of A for track t when track t' is immediately before t. For CF (i,j), suppose array A corresponds to the second component, *i.e.* j. Then, we have

$$m_A(t) = \sum_i \sum_{t' \in T(i,j)} m_A(t|t') \frac{n_s(i,j)}{H_A(j)} \frac{n(t')}{\sum_{t_k \in T(i,j)} n(t_k)} \tag{3.13}$$

---

[3] If two arrays are both reused in the coverage loop, then we can consider a simple method similar to that for computing self conflict misses; instead of using the method described here.

where $\frac{n_s(i,j)}{H_A(j)}\frac{n(t')}{\sum_{t_k \in T(i,j)} n(t_k)}$ is the probability that a track $t' \in T(i,j)$ occurs immediately before t, and $H_A(j)$ is the number of cache sets that contain a j-line footprint of A. Since lines of A are reused in successive coverage loop iterations, they must appear in both tracks t' and t. Hence tracks t' and t must have the same j, but can have different i. Therefore, for track t of CF (i,j), we consider track t' for all i. We now discuss how to compute $m_A(t|t')$ which is the only parameter in Equation (3.13) whose computation method has not yet been discussed.

Generally, for a track t, we can determine $m_A(t|t')$ for each t' by simply applying Lemma 3.3.1 for every t', t pair. In our example, array Y is reused in the coverage loop. We thus compute $m_Y(t)\ \forall\ t \in T(1,2)$. Since there are two CFs, CF (0, 2) and CF (1, 2), with a 2-line footprint of Y, t' can be in either $T(0,2)$ or $T(1,2)$, where $T(0,2) = \{y_1y_2\}$. For each track $t \in T(1,2)$, we can certainly compute $m_Y(t|t')$ for all $t' \in T(0,2) \cup T(1,2)$ by applying Lemma 3.3.1 for every t', t pair. We can then compute $m_Y(t)$, based on Equation (3.13), to produce the results listed in the last column of Table 3.1.

For the purpose of illustration, in our example, we discuss the computation of $m_Y(t)$ for all $t \in T(1,2)$ by dividing $T(1,2)$ into two subsets: one contains tracks beginning with x_1, and the other contains tracks beginning with y_1. In this particular example, all tracks beginning with x_1 will have the same $m_Y(t)$, as do all tracks beginning with y_1. First, consider a track $t \in T(1,2)$ beginning with x_1, *i.e.* $t \in \{x_1y_1y_2x_1,\ x_1y_1x_1y_2,\ x_1y_1x_1y_2x_1\}$. Note that track t contains 2 y elements, y_1 and y_2. We know that t' will also contain these two y elements and in that order. Therefore, any y element in t must be at least $2 + 1 = 3$ ($> D = 2$) elements away from its previous appearance (in t') since at least one x_1 and the other y must occur in between, and thus this y is a cache miss according to Lemma 3.3.1. As a result, $m_Y(t|t') = 2$ for any $t \in \{x_1y_1y_2x_1,\ x_1y_1x_1y_2,\ x_1y_1x_1y_2x_1\}$ and $t' \in T(0,2) \cup T(1,2)$. From Equation (3.13), we have $m_Y(t) = 2$, $\forall\ t \in \{x_1y_1y_2x_1,\ x_1y_1x_1y_2,\ x_1y_1x_1y_2x_1\}$.

Now, consider a track $t \in T(1,2)$ beginning with y_1, *i.e.* $t \in \{y_1x_1y_2,\ y_1x_1y_2x_1\}$. Note that the y_1 in t is always followed by an x_1. Similarly, from Lemma 3.3.1 we know that y_2, which is after the first x_1, is a cache miss. Therefore, element y_1 is the only y element in track t that could be a hit, depending on t'. As shown in Table 3.1, for any track in $T(1,2)$ there is at least one x and a y_2 element after y_1. Hence, if $t' \in T(1,2)$, then the y_1 in t must be at least 3 ($> D$) elements away from the y_1 in t', and thus is a miss. On the other hand, if $t' = y_1y_2$, then the y_1 in t is 2 ($= D$) elements away from the y_1 in t', and thus is a cache hit. In summary, $\forall\ t \in \{y_1x_1y_2,\ y_1x_1y_2x_1\}$, $m_A(t|t') = 2$ if $t' \in T(1,2)$, and $m_A(t|t') = 1$ if $t' \in T(0,2)$. As a result, according to Equation (3.13), $\forall t \in \{y_1x_1y_2, y_1x_1y_2x_1\}$

$$
\begin{aligned}
m_Y(t) &= \sum_{i=0}^{1} \sum_{t' \in T(i,2)} m_Y(t|t') \frac{n_s(i,2)}{H_Y(2)} \frac{n(t')}{\sum_{t_k \in T(i,2)} n(t_k)} \\
&= \frac{n_s(0,2)}{H_Y(2)} \sum_{t' \in T(0,2)} m_Y(t|t') \frac{n(t')}{\sum_{t_k \in T(0,2)} n(t_k)} + \\
&\quad \frac{n_s(1,2)}{H_Y(2)} \sum_{t' \in T(1,2)} m_Y(t|t') \frac{n(t')}{\sum_{t_k \in T(1,2)} n(t_k)} \\
&= \frac{4}{8} \times 1 \times 1 + \frac{4}{8} \times 2 \times 1 = \frac{3}{2}
\end{aligned}
$$

Calculate the Expected Number of Misses, $m_A(i,j)$ We can compute $m_X(i,j)$ and $m_Y(i,j)$ according to Equation (3.6) as follows:

$$\begin{aligned}
m_X(1,2) &= m_X(t_3)Pr[t_3] = 1 \times \frac{3}{52} = \frac{3}{52} \\
m_Y(1,2) &= m_Y(t_1)Pr[t_1] + m_Y(t_2)Pr[t_2] + m_Y(t_3)Pr[t_3] + m_Y(t_4)Pr[t_4] + m_Y(t_5)Pr[t_5] \\
&= \frac{3}{2} \times \frac{1}{52} + \frac{3}{2} \times \frac{3}{52} + 2 \times \frac{3}{52} + 2 \times \frac{3}{52} + 2 \times \frac{3}{52} = \frac{24}{52} = \frac{6}{13}
\end{aligned}$$

Although our example actually involves 7 CFs with respect to (X,Y): (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 2), and (1, 3), not all of them are required for computing cross conflict misses. We certainly do not need to consider CFs (0, 1), (0, 2), and (0, 3) for cross conflict misses of X since they contain an empty footprint of X. Similarly, we do not need to consider CF (1, 0) for cross conflict misses of Y. For other CFs, since X is not reused in the coverage loop, the cross conflict misses of X for any track can be determined by the track itself. We can derive from Lemma 3.3.1 that $m_X(i,j) = 0$ if $i + j \leq D$. Hence, with $D = 2$, we only have to consider CFs (1, 2) and (1, 3) for cross conflict misses of X; others cannot cause such misses.

Unlike array X, array Y is reused in the coverage loop. Hence, the cross conflict misses of Y can be affected by the previous track. Therefore, cross conflict misses of Y can occur in a track of CF (i,j) even if $i + j \leq D$ or $i = 0$. Recall that any two tracks can appear in succession only if they have the same number of y elements, *i.e.* same j. Thus, to determine whether a conflict footprint can cause cross conflict misses, we must consider CFs with the same j as a group. All CFs in a group can guarantee no cross conflict misses of Y only if none of the CFs in the group has more than D distinct lines. In this example, we can form three such groups: CFs (0, 1) and (1, 1), CFs (0, 2) and (1, 2), and CFs (0, 3) and (1, 3). Among them, only the group of CFs (0, 1) and (1, 1) is guaranteed not to cause cross conflict misses of Y.

However, if $j > D$, then every y element will be a miss even with no interference with elements of X. These misses have already been counted as self conflict misses in Equation (3.1). In order not to count these misses twice, our method simply assigns $m_Y(i,j) = 0$ for $j > D$. Hence, $m_Y(0,3)$ = $m_Y(1,3) = 0$ in this example. Therefore, we only need to consider CFs (0, 2) and (1, 2) for cross conflict misses of Y.

In summary, for this example, we need to consider CFs (1, 2) and (1, 3) for cross conflict misses of X, and CFs (0, 2) and (1, 2) for cross conflict misses of Y. We have already computed $m_X(1,2)$ and $m_Y(1,2)$. We can similarly compute $m_Y(0,2) = 11/52$ and $m_X(1,3) = 12/176$. According to Equation (3.7), the number of cross conflict misses per steady-state iteration for arrays X and Y, respectively, is as follows:

$$\begin{aligned}
M_C(X) &= S(m_X(1,2) + m_X(1,3)) = 16(\frac{3}{52} + \frac{12}{176}) = 2.01 \\
M_C(Y) &= S(m_Y(0,2) + m_Y(1,2)) = 16(\frac{11}{52} + \frac{6}{13}) = 10.77
\end{aligned}$$

This completes the process of counting cross conflict misses. The complexity of the process has been reduced by using the concept of averages. Although the computation of $m_A(i,j)$ requires some parameters or information such as (1) histogram of arrays, (2) how arrays are referenced, (3) reusability of arrays in the coverage loop, and (4) number of appearances of an element in a (condensed) reference sequence and in a track, all of them can be obtained by analyzing the program.

Model Result

Suppose there are a total of N coverage loop iterations, given the number of loop cold start misses per coverage loop iteration, the number of self conflict misses per steady-state iteration, and

the number of cross conflict misses per steady-state iteration for each array, the cache behavior model for the loop nest, excluding ping-pong conflict misses, is as follows:
For array X,

$$number\ of\ loop\ cold\ start\ misses = M_L(X) \times N = 8N \tag{3.14}$$

$$number\ of\ cross\ conflict\ misses = M_C(X) \times (N-1) = 2.01(N-1) \tag{3.15}$$

For array Y,

$$number\ of\ loop\ cold\ start\ misses = M_L(Y) \times N = 32 \tag{3.16}$$

$$number\ of\ self\ conflict\ misses = M_S(Y) \times (N-1) = 12(N-1) \tag{3.17}$$

$$number\ of\ cross\ conflict\ misses = M_C(Y) \times (N-1) = 10.77(N-1) \tag{3.18}$$

Note that only an array that is reused in the coverage loop can have self conflict misses; thus the model has no self conflict misses of X. Moreover, the number of self conflict misses and cross conflict misses are derived for a steady-state coverage loop iteration, hence we multiply them by $(N-1)$. We do not count the self and cross conflict misses in the first iteration. However, for large N, this error should be negligible.

Actual Number of Conflict Misses Per Steady-State Iteration

For comparison purposes, we also count the actual number of cache misses. Table 3.2 lists real tracks of each cache set for coverage loop iterations 1, 2, and 3. The table also shows whether a cache hit or miss occurs for each y element reference, as identified according to Lemma 3.3.1. From the first iteration, we can count 32 misses of Y, which are all loop cold start misses. In addition, we can count that each iteration has 8 sets with x elements. Each such set contains one loop cold start miss of X, thus, there are 8 loop cold start misses per iteration. These loop cold start miss results coincide with our model results.

For conflict misses of an array, we count self and cross conflict misses of the array over all steady-state iterations, *i.e.* excluding the first iteration, and then take an average to get number of conflict misses per steady-state coverage loop iteration. To do so, it is sufficient to consider only iterations 2 and 3 in our example since every following pair of iterations, *e.g.* iterations 4 and 5, 6 and 7, and so on, will just repeat the track patterns of these two iterations. Counting all misses of y elements in iterations 2 and 3, we get a total of 50 misses, which results in $50/2 = 25$ conflict misses of Y per iteration. Similarly, we have $5/2 = 2.5$ conflict misses of X per iteration. On the other hand, our model results in 22.77 ($M_S(Y) + M_C(Y) = 12 + 10.77 = 22.77$) conflict misses of Y and 2.01 ($M_C(X) = 2.01$) conflict misses of X per iteration, which undercounts conflict misses of Y by 2.23 and those of X by 0.5. Considering only the conflict misses for each steady-state iteration, the model counts about 90% of the actual steady-state conflict misses.

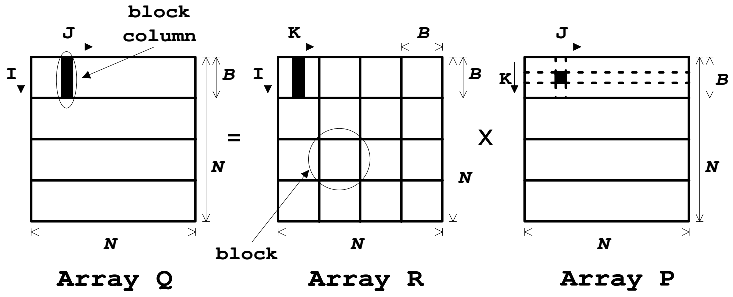
Most of the error comes from $m_Y(0,2)$, *i.e.* the cross conflict misses for track y_1y_2. In our example, track y_1y_2 is always preceded by a track in $T(1,2)$, and thus always causes one or two cache misses as shown in Table 3.2. However, our model also allows another y_1y_2 to appear before the track y_1y_2. Hence, in addition to counting misses of Y for the cases where track y_1y_2 is preceded by tracks in $T(1,2)$, the model also computes the number of misses for the case where y_1y_2 is preceded by another y_1y_2 track and then takes an average of all of these results. Since the case where y_1y_2 is preceded by y_1y_2 will result in 0 misses of Y, the average is less than the number in the actual case which does not include this track sequence. In general, our model will become more accurate if the distribution of the reference sequences is more nearly random.

Cache Set	Coverage Loop Iteration													
	1				2						3			
0	x_1	y_1	x_1	y_2	y_1	y_2					x_2	y_1	x_2	y_2
		m		m	m	h						m		m
1	x_1	y_1			y_1						x_2	y_1		
		m			h							h		
2	x_1	y_1			y_1						x_2	y_1		
		m			h							h		
3	y_1	x_1	y_2		y_1	y_2					y_1	x_2	y_2	
	m		m		m	h					h		m	
4	y_1	x_1			y_1						y_1	x_2		
	m				h						h			
5	y_1	x_1			y_1						y_1	x_2		
	m				h						h			
6	y_1	x_1	y_2	x_1	y_1	y_2					y_1	x_2	y_2	x_2
	m		m		m	m					h		m	
7	y_1	x_1	y_2	x_1	y_1	y_2					y_1	x_2	y_2	x_2
	m		m		m	m					h		m	
8	y_1	y_2			x_1	y_1	x_1	y_2	x_1		y_1	y_2		
	m	m				m		m			m	m		
9	y_1	y_2	y_3		x_1	y_1	x_1	y_2	y_3	x_1	y_1	y_2	y_3	
	m	m	m			m		m	m		m	m	m	
10	y_1	y_2	y_3		x_1	y_1	x_1	y_2	y_3	x_1	y_1	y_2	y_3	
	m	m	m			m		m	m		m	m	m	
11	y_1	y_2			x_1	y_1	y_2	x_1			y_1	y_2		
	m	m				m	m				m	m		
12	y_1	y_2	y_3		x_1	y_1	y_2	x_1	y_3	x_1	y_1	y_2	y_3	
	m	m	m			m	m		m		m	m	m	
13	y_1	y_2	y_3		x_1	y_1	y_2	x_1	y_3	x_1	y_1	y_2	y_3	
	m	m	m			m	m		m		m	m	m	
14	y_1	y_2			x_1	y_1	x_1	y_2	x_1		y_1	y_2		
	m	m				m		m			m	m		
15	y_1	y_2			x_1	y_1	x_1	y_2	x_1		y_1	y_2		
	m	m				m		m			m	m		

Table 3.2: Actual reference sequences and miss (m)/hit (h) information of y_i for the first three coverage loop iterations.

```
DO KK = 1, N, B
  DO II = II, N, B
    DO J = 1, N
      DO K = KK, KK+B-1
        DO I = II, KK+B-1
          Q(I,J) = Q(I,J) + R(I,K) * P(K,J)
        END DO
      END DO
    END DO
  END DO
END DO
```

(a)



(b)

Figure 3.3: Blocked matrix multiply: (a) Code. (b) Operations in the innermost loop.

3.5 Model Evaluation

In this section, we use our method to derive a cache behavior model of a blocked matrix multiply as shown in Figure 3.3a for $Q = R \times P$. Then, we evaluate the model by comparing the model and simulation results. Since in this program, capacity and compulsory misses can be considered together as loop cold start misses, using this program allows us to focus on conflict behavior.

In order to reduce capacity misses, this program divides matrix R into sub-matrices, called *blocks*, and attempts to hold a block in the cache until the block is never used again. Although the number of capacity misses is reduced, the program can still experience a significant number of conflict misses. These conflict misses are mainly due to the reuse of the R blocks being interrupted by other elements of itself (self conflicts), by elements of arrays P (cross conflicts), and/or Q (cross and ping-pong conflicts). We choose loop J as the coverage loop. A coverage loop iteration thus includes one block of R, one block column of P, and one block column of Q. If the coverage loop was chosen as loop K or I, *i.e.* lower than the level of loop J, then a coverage loop iteration would not contain a full R block. Hence, we would not be able to analyze all possible conflict patterns that interrupt the reuse of R. On the other hand, if the coverage loop was chosen as loop II or KK, *i.e.* higher than the level of loop J, then a coverage loop would contain many repetitions of the cache miss patterns, which provides little new information. We therefore analyze the cache miss patterns for the data set associated with a loop-J (coverage loop) iteration and derive the cache behavior model for this program accordingly.

As shown in Figure 3.3b, the program also divides every column into sub-columns, called *block columns*. For simplicity, we assume that the leading dimension of an array and the leading dimension of a block (the size of a block column) are each a multiple of the line size L, and that arrays and block columns start at cache line boundaries. These assumptions are actually reasonable in practice; although an array can generally start at any address, it is convenient to assume that an array can only start at a cache line boundary, and some machines, such as the KSR series, do have a compiler directive that forces this alignment. Moreover, in order to minimize the cache space occupied by a block, it is desirable that the block starts at a cache line boundary and that the leading dimension[4] of the block is a multiple of L. This program requires that the leading dimension of the array is also a multiple of L so that each block can be easily aligned to a cache line boundary. If the leading dimension of the array is not a multiple of L, we can add some dummy elements at the end of each column to fulfill this requirement.

For clarity of illustration, we consider square blocks ($B \times B$) in square matrices ($N \times N$) here. According to the aforementioned assumptions, B and N are each multiples of L. Since $B \times B$ is supposed to be no greater than the cache size $C = D \times S \times L$, it is also reasonable to assume that a block column is no larger than "one way" of the cache, *i.e.* $B \le C/D = S \times L$. Thus every cache line of a block column will be in a distinct set of the cache.

3.5.1 Model Derivation

The cache behavior model consists of four components: loop cold start, self conflict, cross conflict, and ping-pong conflict misses. Each component is discussed in turn below. After we have the number of misses per (steady-state) coverage loop iteration for each component, their sum is multiplied by the number of coverage loop iterations to get total misses.

Loop Cold Start Misses It is generally safe to assume that only the R block can be reused in successive coverage loop iterations, and the block columns of Q and P will need to be loaded from the memory in every coverage loop iteration. Since the size of a block column is B and a block is $B \times B$, for each array the number of loop cold start misses per coverage loop iteration is

$$\begin{array}{l} M_L(P) = M_L(Q) = B/L \\ M_L(R) = (B^2/L)(1/N) \end{array} \tag{3.19}$$

Note that by analyzing the program $P(K,J)$ and $Q(I,J)$ are supposed to be reused in loops II and KK, respectively. However, these reuses are too far apart to be likely. For example, before a rereference to $P(K,J)$ in the next loop-II iteration, the program will have accessed one block of R, N block columns of Q, and N block columns of P. Since to effectively utilize the cache, one block is close to, but no larger than the size of the cache, these $2(N/B)+1$ blocks of data almost certainly exceed the cache capacity. Therefore, the reuse of $P(K,J)$ in loop II is unlikely to happen. Since the data set referenced between two successive reuses of $Q(I,J)$ in loop KK, which includes N/B blocks of R, the whole of array Q, and N block columns of P, is even larger, the reuse of $Q(I,J)$ in loop KK is even less likely.

Self Conflict Misses Before modeling self and cross conflict misses, we need to derive the footprint histograms of the arrays. In this program, footprint histograms are computed based on the data size and shape determined by loops K and I, which include one block column of P, one block column of Q, and one block of R. Since only R is reused in the coverage loop, only references to elements of array R can cause self conflict misses. Given the footprint histogram of R, we can compute $M_S(R)$, the number of self conflict misses of R per steady-state coverage loop iteration, by using Equation (3.1).

Cross Conflict Misses Most of the cross conflict misses in this program come from conflicts

---

[4] In this dissertation, we follow the Fortran program convention that consecutive elements in the same column are placed in contiguous memory space.

```
1.  For each array $X$ in an array pair $\{Q, R\}$ of interest
    • Initialize the elements of $I_X(s)$ to 0
    • For the initial address, $i$, of each column of $X$,
      - Compute the cache set number, $s$, that $i$ maps to
      - Increment $I_X(s)$ by 1
2.  $M_{PP}(Q) = M_{PP}(R) = \sum_{s \in \{1,2,\ldots,S\}} I_Q(s) \cdot I_R(s) \cdot \frac{N}{L} \cdot (L-1)$
```

Figure 3.4: Algorithm for computing ping-pong conflict misses of the blocked matrix multiply program in Figure 3.3a when $D = 1$. (If $D > 1$, $M_{PP}(Q) = M_{PP}(R) = 0$.)

between arrays Q and R, and between arrays P and R. In each case, we can compute the number of cross conflict misses per steady-state coverage loop iteration based on the process described in Section 3.4.2. Since in this process, the most complicated procedure is to compute the expected number of cross conflict misses in a cache set, we describe this procedure for Q-R and P-R conflicts in detail in Appendix A.

Ping-Pong Conflict Misses The algorithm in Figure 3.4 computes the number of ping-pong conflict misses of this program for a direct mapped cache. As discussed above, in each loop-K iteration a block column of Q and a block column of R are accessed. Since they are referenced consecutively and alternately, they can potentially cause ping-pong conflict misses. The numbers of ping-pong conflict misses associated with these arrays for a complete program run are denoted as $M_{PP}(Q)$ and $M_{PP}(R)$, respectively. For these two arrays, ping-pong conflict misses can occur only in a direct mapped cache. Therefore, $M_{PP}(Q)$ and $M_{PP}(R)$ resulting from this algorithm are only for $D = 1$. If $D > 1$, then $M_{PP}(Q) = M_{PP}(R) = 0$.

Now consider a direct mapped cache, *i.e.* $D = 1$. Let $U_R(i, j)$ represent the i^{th} block column in column j of matrix R, and similarly $U_Q(i, j)$ for Q. Then, in fact, any two block columns, $U_Q(i, j)$ and $U_R(i', k)$, will be accessed simultaneously in some loop-K iteration if and only if $i = i'$. Therefore, if $U_Q(1, j)$ and $U_R(1, k)$ begin in the same cache set, then the cache sets mapped to by these two block columns will experience ping-pong conflict misses; furthermore, the cache sets mapped to by $U_Q(2, j)$ and $U_R(2, k)$, by $U_Q(3, j)$ and $U_R(3, k)$, and so on, will also experience ping-pong conflict misses. This implies that the cache sets mapped to by column j of array Q and column k of array R will all experience ping-pong conflict misses. Since a column spans N/L cache sets, from Section 3.4.1 we know that each array will be associated with $\frac{N}{L} \times (L-1)$ ping-pong conflict misses for each pair of columns j, k for which $U_Q(1, j)$ and $U_R(1, k)$ begin in the same cache set.

The key question is then how many pairs of columns have initial addresses that map to the same cache set. A brute-force algorithm that checks every column pair and counts the ping-pong conflict misses would take $O(N^2)$ time. Instead, we implement a rather efficient algorithm (Figure 3.4) that takes $O(N + S)$ time.

In this algorithm, for each column of an array X, where $X = Q$ or R, the first step computes the cache set number mapped to by its initial address. If this set number is s, then $I_X(s)$ is increased by 1. Thus, at the end of this step, $I_X(s)$ contains the number of columns whose initial addresses map to cache set s for array X. As a result, there are $I_Q(s) \times I_R(s)$ pairs of columns, one from each array, whose initial addresses map to the same cache set s and thereby result in

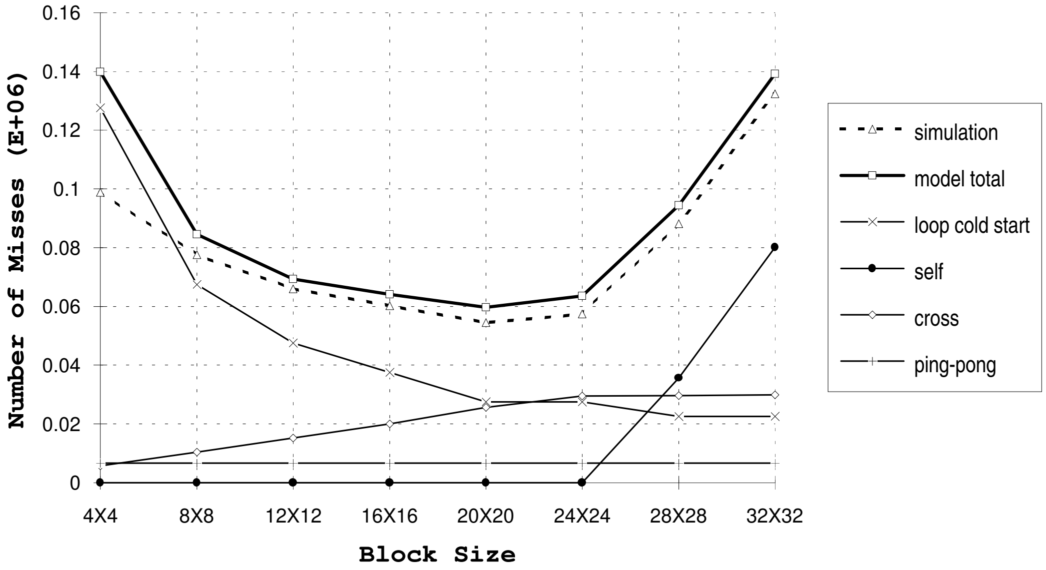


Figure 3.5: Number of Misses vs. Block Size: $(D, S, L) = (1,\ 256,\ 4)$; $N = 100$.

$(N/L)(L-1)I_Q(s)I_R(s)$ ping-pong conflict misses for each array in a direct mapped cache. The second step then accumulates all these misses over all cache sets to compute total ping-pong conflict misses for this array pair.

3.5.2 Model and Simulation Comparison and Discussion

In this section, we compare outputs from the cache behavior model and simulation. The simulation data were obtained by using the Atrace[5] tool from IBM to generate traces of programs, and then using the dinero III trace simulator[6] to simulate the traces. The results show that the model is very close to the simulation. Therefore, we can use the four model components for loop cold start, self conflict, cross conflict, and ping-pong conflict misses, to explain the cache miss behavior fairly well.

Figures 3.5, 3.6, and 3.7 show the number of cache misses as a function of block size ($B \times B$) for $N = 100$, 200, and 300, respectively. The cache organization is $(D, S, L) = (1, 256, 4)$, and the cache size is $C = D \times S \times L = 1024$ (elements), which is equivalent to 8 KBytes since we use double precision arrays in the program. Each figure displays curves for the total misses from the model and from simulation as well as curves for each of the four model components: loop cold start, self, cross, and ping-pong. Since the number of ping-pong conflict misses is independent of the block size, the curve for ping-pong conflict misses is flat for all block sizes. Therefore, ping-pong conflict misses should have a greater impact on small total cache misses than on large total cache misses.

Consider the two curves for the total misses. The model total curve closely follows the simulation curve in Figures 3.6 and 3.7. However, in Figure 3.5, a great discrepancy appears at

---

[5] Atrace is IBM internal software made available to us by Ravi Nair.

[6] Copyright 1985, 1989 Mark D. Hill

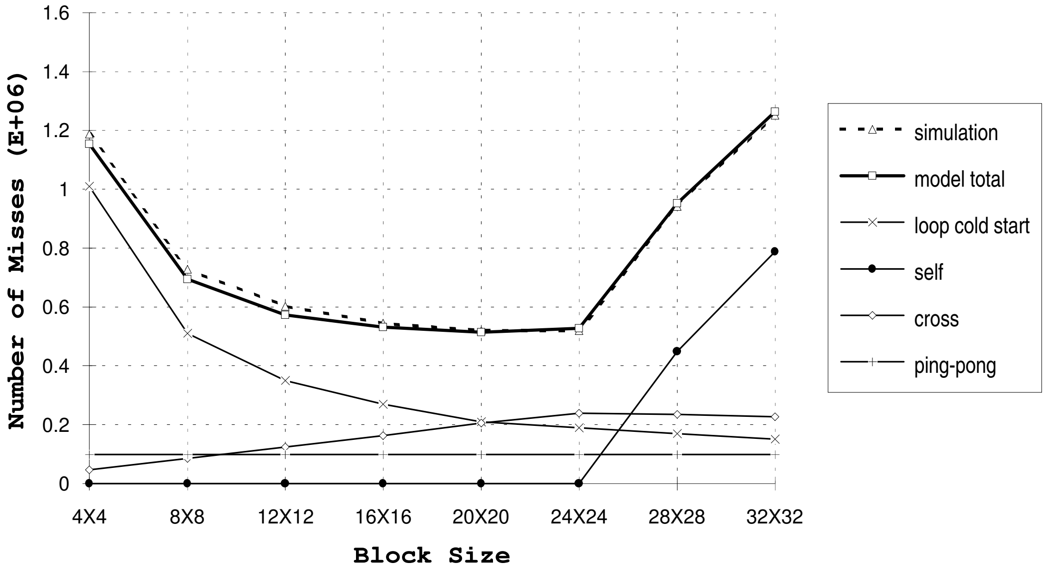


Figure 3.6: Number of Misses vs. Block Size: $(D,S,L) = (1,\ 256,\ 4)$; $N = 200$.

block size 4×4. From the loop cold start curve, we know that the discrepancy is because the model overcounts the loop cold start misses in this particular case. Recall that our model assumes that there is no reuse of $P(K,J)$ between successive loop-II iterations since the data set referenced between two such reuses usually exceeds the cache capacity. However, when $B = 4$ and $N = 100$, this data set, one block of R and N block columns each of P and Q, is equivalent to $B^2/L + NB/L + NB/L = 4 + 100 + 100 = 204$ lines which is actually less than the cache size (256 lines). Hence, some reuses of $P(K,J)$ may actually happen, thus resulting in fewer loop cold start misses than expected. Note that the assumption is reasonable for $B \geq 8$ or $N \geq 200$.

When the block size is small, the loop cold start misses dominate the cache misses. However, as the block size increases, they decrease sharply. The number of coverage loop iterations in this program equals N^3/B^2. Multiplying this number by $M_L(P) + M_L(Q) + M_L(R)$ from Equation (3.19), we have the total number of the loop cold start misses as

$$\frac{N^2}{L} + \frac{2N^3}{BL} \tag{3.20}$$

It is evident that the total number of the loop cold start misses decreases as B increases. In addition, after the new data are loaded into the cache, they are supposed to be reused in a coverage loop iteration unless conflicts occur. For large B, more data is reused and is thus reloaded less often.

Unlike the loop cold start misses, self conflict misses usually do not occur when the block size is small because elements of a small R block are less likely to conflict with one another. Nonetheless, once self conflict misses do occur, they increase rapidly and soon become the dominant type of cache miss. The occurrence of the self conflict misses also affect the trend of the cross conflict curve. The curve of the cross conflict misses initially goes up steadily and then levels off (and

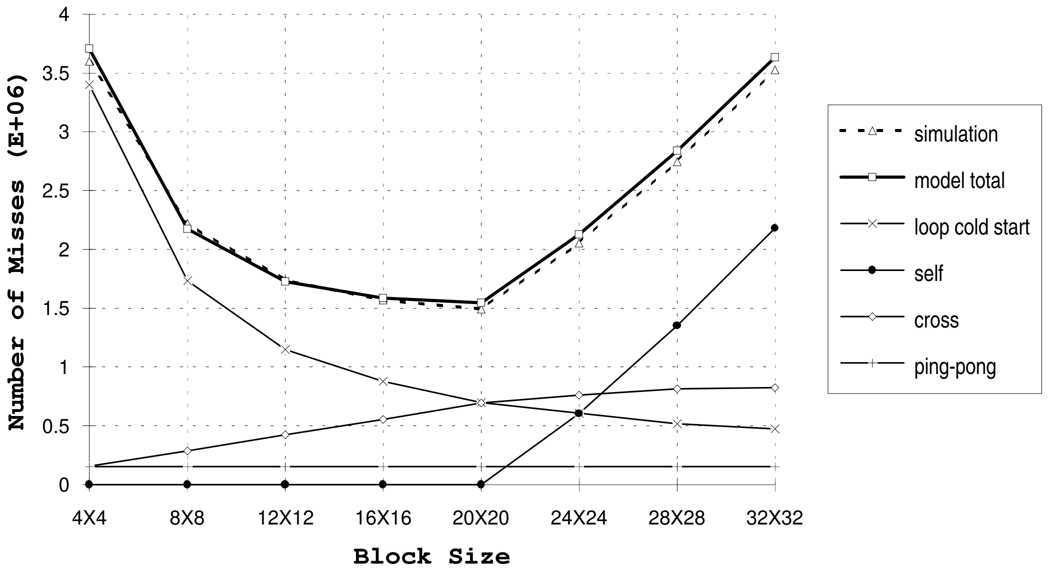


Figure 3.7: Number of Misses vs. Block Size: $(D, S, L) = (1,\ 256,\ 4)$; $N = 300$.

may even decrease) after the block size at which the self conflict misses begin to occur. Note that the cross conflict misses of this program mainly come from the conflicts between arrays R and Q or between arrays R and P. Hence, the greater the number of cache sets in which elements of a R block reside, the more likely the cross conflicts will be, and thus the greater the number of cross conflict misses. However, once self conflict misses begin to occur, cache sets begin to contain multiple lines of the R block and the number of sets occupied by the R block grows less rapidly. Hence, the incremental rate of the number of cross conflict misses is also reduced.

Figure 3.8 shows results for a fixed size (8 KByte) cache with different organizations (D, S, L) = (1, 256, 4), (2, 128, 4), and (4, 64, 4). We use $D = 1$, $D = 2$, and $D = 4$ labels to distinguish among these cache organizations. In each cache organization, the figure displays curves of the model and simulation total misses for array size 300×300. For $D = 4$, it also shows the model component curves of self and cross conflict misses. In each case, we see that the model total and simulation curves are very close.

Note that in this figure, the total number of cache misses for a set associative cache, $D = 2$ or $D = 4$, is significantly less than that for a direct mapped cache, $D = 1$. From Equation (3.20), we know that the line size L is the only cache parameter that can affect the number of loop cold start misses. Since the three cache organizations have the same line size, they should have the same number of loop cold start misses. Hence, the reduction in cache misses in set associative caches relative to direct mapped caches is due to the reduction in conflict misses. In a set associative cache, a line can have more than one place to reside within a set. Therefore, the ping-pong conflict misses between two array sections disappear, and the self and cross conflict misses are reduced. In fact, the cross conflict misses are now like self conflict misses in that they do not occur with small block sizes. In this figure, the cross conflict misses start to occur at block size 20×20 for $D = 2$ and at 28×28 for $D = 4$; the self conflict misses begin at block size 32×32 in either case.

Consequently, when block size $\leq 16 \times 16$ for $D = 2$ and when block size $\leq 24 \times 24$ for $D = 4$,

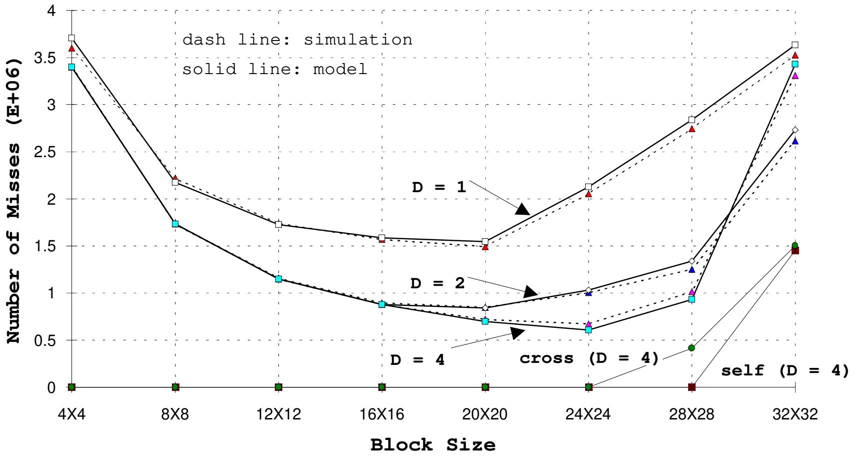


Figure 3.8: Number of Misses vs. Block Size:
$C = 1024$ elements (8 KBytes), $L = 4$; $N = 300$.

the total cache misses consist of only loop cold start misses, and thus the corresponding curves are decreasing up to this point. In each case, the curve starts to turn upward when the cross conflict misses begin to occur and the rate of increase becomes much greater when the self conflict misses begin at block size 32×32. Generally, a cache with large D is less sensitive to conflict misses. This figure presents an exception at block size 32×32 at which the total number of misses for $D = 4$ surpasses that for $D = 2$. From the curves of the self and the cross conflicts, we know that this is not only caused by the self conflict misses but also caused by the surge of cross conflict misses in the case of $D = 4$. One fact worth mentioning here is that our model is capable of capturing this exceptional instance accurately.

Figure 3.9 shows the results for an 8 KByte direct mapped cache with $L = 1$, 2, and 4. The matrix size is also 300×300. Without any exception, this figure shows that this program can always benefit from larger line size. Since this program accesses the arrays consecutively, large line size can dramatically reduce the number of loop cold start misses.

Figures 3.5 through 3.9 demonstrate the accuracy of our model. Figure 3.10 then illustrates how informative our model can be regarding types of misses and the arrays where they occur. In the figure, the height of a bar represents the total number of cache misses for one array. Each bar is further broken down into sections for loop cold start, self conflict, Q-R cross conflict, P-R cross conflict, and ping-pong conflict misses. We ignore the cross conflict misses between arrays P and Q since this case is negligible relative to the other two cases. This fine grain information allows us to identify the worst types of cache misses and to select appropriate techniques to improve cache performance directly and effectively.

3.6 Conclusion

This chapter demonstrated an analytical method for deriving a cache conflict behavior model of an application. The method is suitable for a wide variety of cache organizations. Although

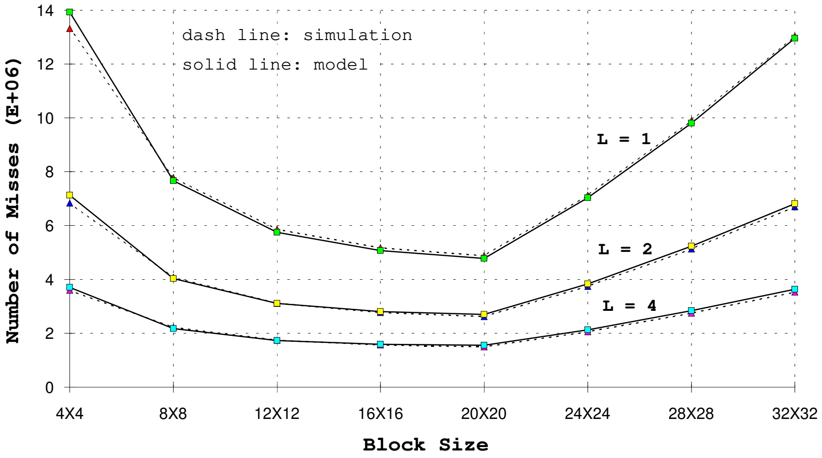


Figure 3.9: Number of Misses vs. Block Size:
$C = 1024$ elements (8 KBytes), $D = 1$; $N = 300$.

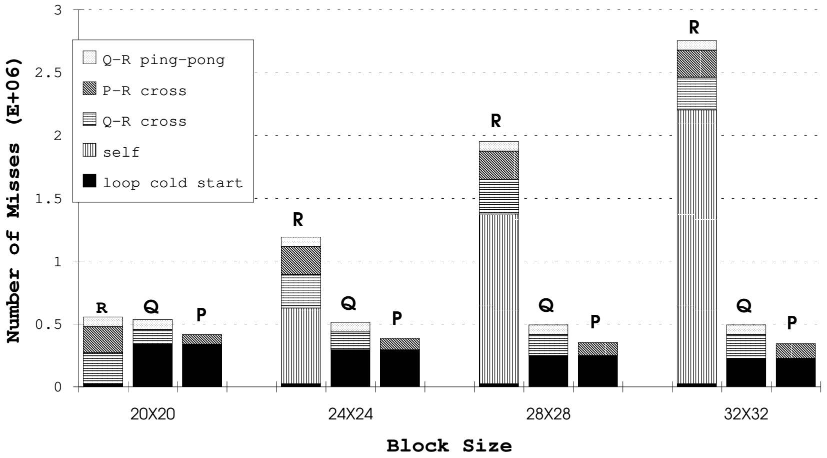


Figure 3.10: Breakdown of cache miss information for each array:
$(D, S, L) = (1, 256, 4)$, $N = 300$.

accuracy is usually a potential problem with analytical models, our results show that our model can generally achieve very good accuracy (within 11% in our experiments except for the case where block size = 4×4 and $N = 100$ which violates our reuse assumption, but contains an undesirably small block size). Since our method uses averages, it will become more accurate if the reference sequences of the application are distributed more randomly. Hence, our model is potentially most appropriate for complex programs that tend to generate random distributions that employ all possible reference sequences.

Due to the nature of analytical methods, the model can provide significant insight into the number of cache conflicts of each type and where they occur. Furthermore, such a model can be integrated within a compiler system to accomplish automatic performance tuning. Such fine grain information, as shown for each array in Figure 3.10, allows us to identify the most severe cache miss problems and to select appropriate techniques that directly address them. Chapter 4 presents a series of such techniques, each capable of reducing one type of conflict miss identified by this model.

CHAPTER 4

Cache Conflict Miss Reduction Techniques

This chapter studies reduction techniques for conflict misses. In Chapter 3, we have classified the conflict misses into *self*, *cross*, and *ping-pong* conflict misses. Each type of conflict miss is associated with a necessary characteristic of how array elements are placed in memory, and thus in a cache. For example, consider a direct mapped cache. If there are self conflict misses, then some cache sets must be mapped into by more than one line of an array. If there are cross conflict misses, then some cache sets must be mapped into by lines from different arrays. If there are ping-pong conflict misses, then some array sections must begin in the cache less than L elements apart. Thus, it is evident that one fundamental method to eliminate or reduce each type of conflict miss is to optimize the data layout of arrays.

We propose three methods, each eliminating or reducing one type of conflict miss. Each method uses one or a combination of the following three techniques that modify or exploit the data layout of arrays: *array padding*, *initial address adjustment*, and *access resequencing*. The array padding technique attaches dummy elements to the end of each column of an array; the initial address adjustment technique adjusts the distance between the initial addresses of pairs of arrays; and the access resequencing technique produces a preferred reference sequence.

In Section 4.1 we first introduce some related work. Then, in Section 4.2 we begin to describe our methods by illustrating ways of eliminating or reducing each type of conflict miss, and in Section 4.3 we discuss the mathematical properties of the data layout techniques. Section 4.4 demonstrates how to apply the methods to a real application by using a block matrix multiply program as an example. In addition, simulations are run to illustrate the effectiveness of these methods. Conclusions are stated in Section 4.5.

4.1 Related Work

Many hardware and software techniques have been proposed for reducing cache conflict misses. The common strategy employed by the hardware techniques is to provide additional space for conflict lines. The most direct method is to increase the degree of set associativity [44]. However, according to the experiments in Chapter 3, increasing the degree of set associativity may not solve the problem as expected. In addition, the hardware cost and cache access latency will be increased [45]. Hence, regarding reduction of conflict misses, most of the research focuses on direct mapped caches. In a direct mapped cache, additional space is sometimes provided by adding a small fully-associative cache, such as a victim cache which retains a few of the most recently replaced lines among all sets [46], by allowing an alternative cache set within the cache, such as in a column-associative cache which maps a conflict line to a different cache set [47], or by directly passing the conflict line through the cache without being stored [48]. The general concerns regarding such hardware methods are that programs can only benefit by running on those machines that implement the solution and that a particular program may only realize a significant benefit with only a particular subset of the possible hardware implementations.

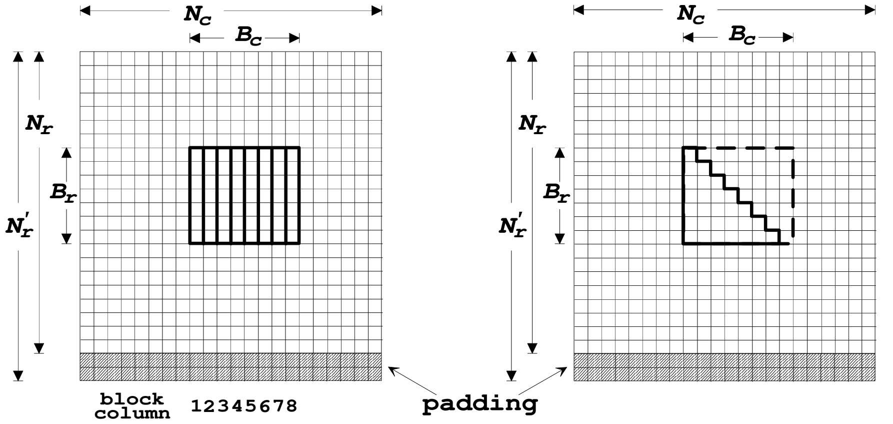
Software techniques can be applied to any system and may enhance the performance of a hardware solution. Unlike the hardware techniques, the software methods usually try to reduce total cache misses by eliminating or reducing the probability that line conflicts occur in each cache set. The software methods are basically divided into two categories: loop transformations and data layout. The loop transformation techniques restructure loop nests to adjust the reference order while the data layout techniques try to optimize array storage patterns. The class of loop transformation techniques includes loop fusion, loop distribution, loop interchange, loop permutation, *e.t.c.* [9, 23, 24, 25, 49, 50]. These techniques as well as the hardware techniques deal with gross characteristics of conflict misses, whereas data layout techniques generally deal with particular classes within a relatively fine-grain classification of conflict misses.

Much of the prior research on data layout techniques has focused on the copy technique [40, 41]. The array copy technique copies array sections of concern into a contiguous temporary array, replaces references to the original arrays with references to the temporary array, and finally updates copies of the elements in the temporary array back to the original arrays, if necessary. Since the temporary array maps to contiguous lines in a cache, it will result in no self conflict misses if the array size is less than the cache size. Given that sections of several arrays can be copied into one copy array, this technique can be used to eliminate each type of conflict miss, yet not without disadvantages. The two most well-known problems are (1) it introduces run time overhead for copying elements back and forth between the temporary and the original arrays, which may be significant; and (2) it needs special modifications within the application code to maintain coherence when data that is shared among different processors is copied. Yamada *et al.* proposed a mechanism similar to the gather-scatter operations in vector machines to reduce the run time overhead by providing hardware support for data copying [51]. However, they do not include any mechanism to ease the coherence problem caused by the copy technique.

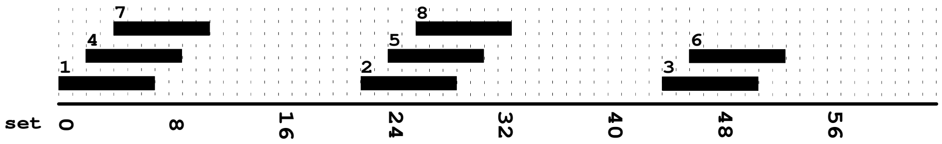
Two of the three techniques used in this study are data layout techniques: namely, array padding and initial address adjustment. These two techniques only modify the storage patterns. Therefore, no program modification is required, nor will any additional operations be introduced at run time. Even though the third technique, access resequencing, may have to modify the program, the resulting run time overhead should be much less than that for copying. Furthermore, all three techniques preserve the references of the program and thus will not complicate the cache coherence problem.

4.2 Data Layout Methods for Reducing Cache Misses

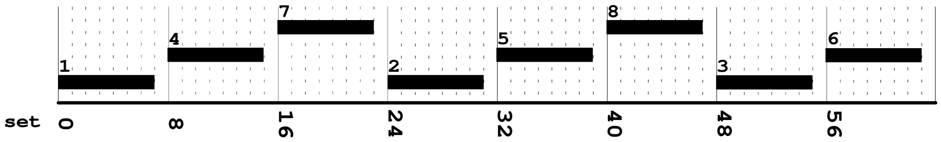
In Sections 4.2.1, 4.2.2, and 4.2.3 we illustrate, respectively, how the self, ping-pong, and cross conflict miss can be removed or reduced by one or a combination of the three proposed techniques: array padding, initial address adjustment, and access resequencing. As in Chapter 3, in this chapter we also assume that word addresses are used where one word is one array element, and use (D, S, L) to represent a cache organization, where D is the degree of set associativity, S is the number of cache sets, and L is the line size. We also use $C = D \times S \times L$ to denote the cache size. Note that cache line size, L, and total size, C, are measured in terms of array elements.



(a) Rectangle data set (b) Triangle data set



(c) Cache Mapping: self conflict misses



(d) Cache Mapping: no self conflict misses

Figure 4.1: Mapping of a block into the cache: (D, S, L) = (1, 64, 1), B_r = 7, B_c = 8, original N_r = 22, and after array padding N_r' = 24.

4.2.1 Eliminating Self Conflict Misses

Consider a cache of $(D, S, L) = (1, 64, 1)$ and a submatrix (block) of size $B_r \times B_c = 7 \times 8$ in an array of size $N_r \times N_c = 22 \times 22$, as shown in Figure 4.1a. Suppose, without loss of generality, that the first block column of the block begins in cache set 0. Then, as shown in the Figure 4.1c, since the first elements of two successive block columns are 22 elements (cache sets) apart, the block columns of the block will thus begin in cache sets 0, 22, 44, 2, 24, 46, 4, and 26, respectively.

Let $\mathcal{S}_b$ be the multiset[1] that contains cache set numbers that correspond to the first element of each block column of a block. Define the *distance* between a pair of cache sets, s_1 and s_2 where $s_2 \geq s_1$, to be $\min\{s_2 - s_1, S - s_2 + s_1\}$. Define the *minimum set distance* of $\mathcal{S}_b$, δ_b, to be the minimum distance over all pairs of elements in $\mathcal{S}_b$. When there is only one element in $\mathcal{S}_b$, we define $\delta_b = S$. The value of δ_b specifies the maximum number of cache sets that can be spanned by a block column without causing self conflict misses. If a block column of the block spans more than δ_b cache sets, then some cache sets will be mapped into by more than one line of the block, thereby resulting in self conflict misses. In the above example, $\mathcal{S}_b = \{0, 22, 44, 2, 24, 46, 4, 26\}$ and $\delta_b = 2$. Since each block spans 7 cache sets, we find that cache sets 2 through 8, 24 through 30, and

---

[1] A *multiset* is a set where more than one element may have the same value.

46 through 50 are all mapped to by more than one line of the block. In a direct mapped cache, all such cache sets will experience self conflict misses. To eliminate these self conflict misses in a direct mapped cache, a method must guarantee that δ_b is no less than the number of cache sets spanned by a block column, *i.e.* $\delta_b \geq 7$ in this example. (Later, we will discuss how to remove self conflict misses in a set associative cache.) Certainly, the block size must be no greater than the cache size C, which is assumed throughout this dissertation.

If the block is read only, then we can use the *copy* technique [40, 41] to eliminate the self conflict misses without any coherence problem. However, there may still be significant copy overhead [41]. Due to the two potential problems in the copy technique (copy overhead and cache coherence), we focus on a different approach, *array padding*, for eliminating the self conflict misses.

The *array padding* technique changes the relative positions of the elements in the same row by attaching dummy elements to the end of each column, *i.e.* it changes the leading dimension of the array. Let N_r' be the new leading dimension. In the example above, if we add 2 dummy elements to the end of each column so that $N_r' = 24$, then $\mathcal{S}_b = \{0, 24, 48, 8, 32, 56, 16, 40\}$, and $\delta_b = 8 > B_r = 7$. Figure 4.1d shows the new mapping of this block. Note that none of the cache sets is mapped to by more than one line of the block, *i.e.* the self conflict misses have been completely eliminated. Given the value of S, the compiler can easily determine the new leading dimension, and no program modification is required. Therefore, this technique does not incur any of the problems of the copy technique. One possible drawback of this technique is that an array may now require a significantly larger memory space; thus, the performance could suffer from additional page faults.

Other data set shapes can also use the array padding technique to resolve self conflicts. Consider a triangular data set that is contained within the rectangular block $B_r \times B_c$ above, as shown in Figure 4.2b. It is evident that if the rectangular block is self-conflict-free, then the triangular data set is also self-conflict-free. In fact, for any data set shape we can apply the array padding technique to determine a self-conflict-free leading dimension by considering a rectangular block that encloses the data set.

Thus far we have only discussed how to apply the array padding technique in a direct mapped cache. The following theorem describes how we can view a set associative cache as a direct mapped cache and eliminate the self conflict misses by the method suggested above.

Theorem 4.2.1 *A method that achieves a self-conflict-free mapping for a direct mapped cache of $D \times S$ sets will also achieve a self-conflict-free mapping for a D-way S-set set associative cache.*

Proof (Sketch):
Note that with a self-conflict-free leading dimension, each line of a direct mapped cache is mapped to by no more than one line of array block elements. Thus any D cache lines are mapped to by at most D lines of elements; in particular, since exactly D lines of the $D \times S$ set direct mapped cache are mapped into each set of the D-way S-set set associative cache. Finally, there are no self-conflict misses in this set associative cache since D lines can be accommodated in any set without conflict. □

4.2.2 Removing Ping-Pong Conflict Misses

Ping-pong conflict misses can occur within one array or between different arrays. The former case is not of interest here because self conflict misses can usually be eliminated completely and when this is done, the ping-pong misses within those arrays themselves will also disappear. Hence, we only consider ping-pong conflict misses between sections of different arrays. For simplicity, we concentrate on two arrays. Since ping-pong conflict misses between two arrays only occur in direct

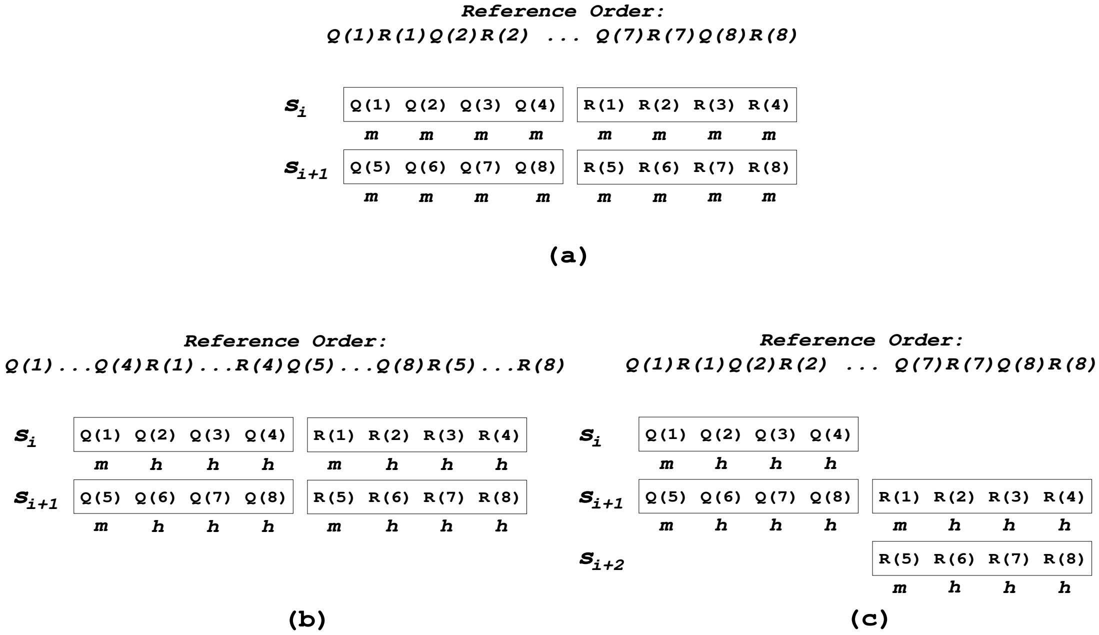


Figure 4.2: Solutions for ping-pong conflict misses: (a) Illustration of ping-pong conflict misses. (b) Rearrange reference order to remove ping-pong conflict misses. (c) Adjust data layout to remove ping-pong conflict misses.

mapped cache, we only consider direct mapped caches here. Furthermore, Section 3.4.1 showed that if two array sections do not cause ping-pong conflict misses when stride = 1, then they will not cause ping-pong conflict misses when stride $\neq$ 1. Thus, a solution with no ping-pong conflict misses for stride = 1 is also a solution for stride $\neq$ 1, and it is sufficient to consider only the case of consecutive references (stride = 1).

Figure 4.2a, which is a copy of Figure 3.2b, depicts ping-pong conflict misses in two consecutively and alternately referenced line-boundary-aligned array sections. The ping-pong conflict misses occur because (1) the two array sections are referenced alternately, and (2) they begin less than L elements apart in the cache. One approach to eliminate the ping-pong conflict misses is to unroll the loop L times and rearrange the memory instructions so that elements of the same line are referenced consecutively. Figure 4.2b shows a new reference order after such a rearrangement. Each line now has only one cache miss, and thereby no ping-pong conflict misses. One disadvantage of this method is that it requires additional registers for storing the accessed data, and thus may inhibit the use of some advanced compiler optimizations that also have a high register requirement, such as software pipelining [52, 36].

Rather than rely on being able to apply reference reordering to eliminate alternate reference patterns, our approach adjusts the data layout instead. We use a method, called *initial address adjustment*, that adjusts the initial addresses of arrays so that the two array sections begin in the cache at least L elements apart. Figure 4.2c shows an example where the initial address of the R array section has been adjusted so that the R section begins in cache set s_{i+1}. In this case, all ping-pong conflict misses are removed. This technique is superior to loop unrolling and instruction

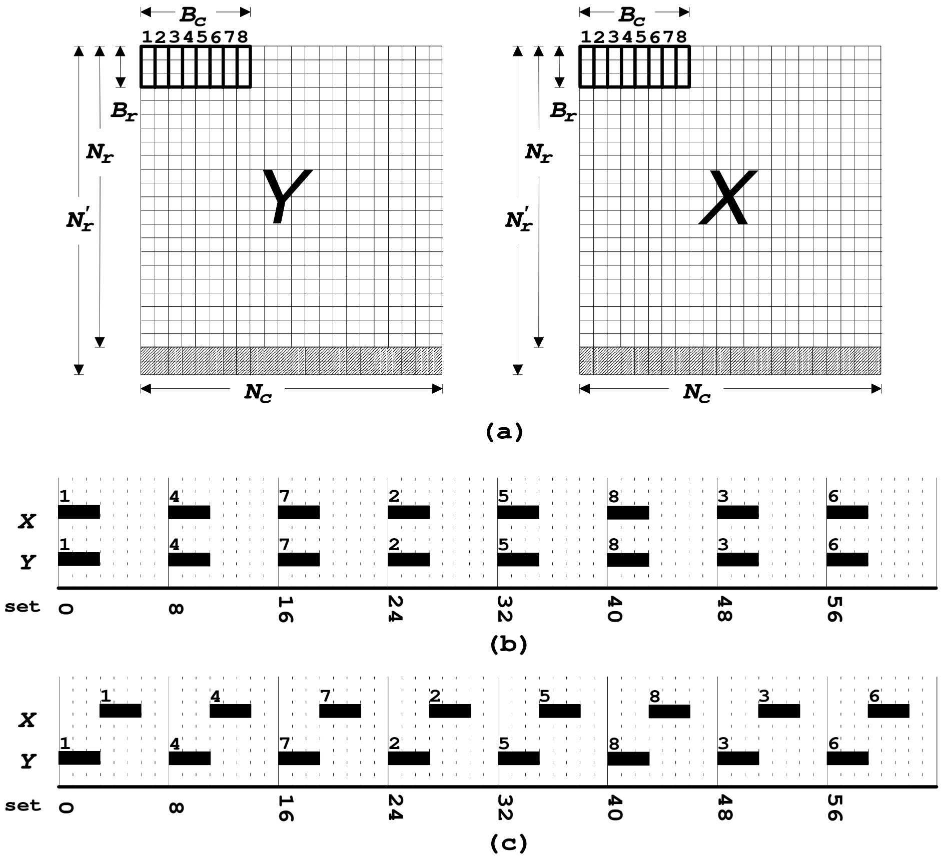


Figure 4.3: One solution for eliminating cross conflict misses.

arrangement since it requires no program restructuring and uses no additional registers. Hence, it has little, if any, effect on other compiler optimizations. One minor disadvantage of this method is that it introduces some dummy elements between arrays.

4.2.3 Reducing Cross Conflict Misses

Cross conflict misses are similar to self conflict misses, except that the conflict is between sections of different arrays. As with self conflict misses, our first attempt is to eliminate cross conflict misses completely. No cross conflict misses occur if no cache set is mapped to by more than D lines among all the array sections of concern. To satisfy this condition, one approach is to apply the array padding technique with a method similar to that for self conflicts. However, to do so, the data set of concern must be no larger than the cache. For example, consider a cache of organization $(D, S, L) = (1, 64, 1)$, and two blocks, one from array X and the other from array Y, as shown in Figure 4.3a. Suppose each array has size 22×22, and each block is of size 3×8. If both X and Y begin in cache set 0, then the mapping of each block in the cache will be like the one in Figure 4.1c, except that now each block column spans only 3 cache sets.

The idea for eliminating the cross conflict misses in this case is illustrated in Figure 4.3b and 4.3c. We first apply the array padding technique to both arrays. The constraint here is that δ_b must be no less than the block column size of the X block plus that of the Y block, which equals 6. (Note that in removing self conflict misses, δ_b need only be greater than or equal to the size of one block column.) One leading dimension that fulfills this constraint is 24. With $N_r' = 24$, $\mathcal{S}_b =$

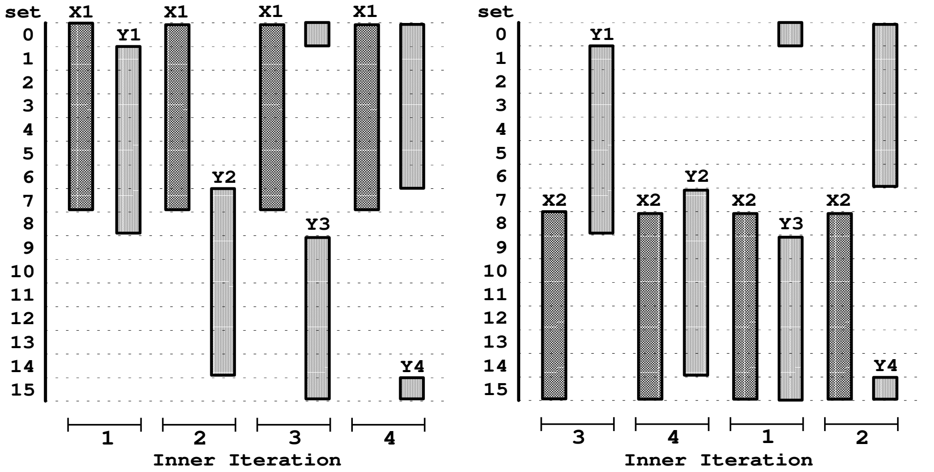
$\{0, 8, 16, 24, 32, 40, 48, 56\}$, and $\delta_b = 8$. Suppose these two arrays still begin in cache set 0. Then, Figure 4.3b displays the mappings of these two array blocks. Assume that array X is placed after array Y in the main memory. Then, we apply the initial address adjustment to shift the mapping of the X block 3 cache sets to the right, as shown in Figure 4.3c. This shift satisfies the condition that no cache set is mapped to by more than one line, thereby removing all conflict misses.

If the cross conflict misses cannot be completely removed, we then try to reduce them. Consider the example in Section 3.4.2, where the mappings of data in the cache for the odd and even numbered coverage loop iterations are shown in Figure 3.1a and 3.1b, respectively. In this example, $(D, S, L) = (2, 16, 1)$, and each coverage loop iteration involves four array sections of Y and one array section of X. Although the four Y array sections can fit in the cache, the total data, which also includes the X section, is over the cache capacity and there will be conflict misses, either self conflict misses of Y or cross conflict misses between X and Y. Since self conflict misses are usually more numerous than cross conflict misses, we first remove the self conflict misses. In this example, any array padding solution that removes all the self conflict misses of Y must result in a situation where every cache set is mapped to by exactly two lines of Y. Thus, the conflicts will occur in half the cache sets and these will have conflict footprint $(1, 2)$ for (X, Y). The kernel idea for reducing the cross conflict misses is to convert the tracks of the most conflict cache sets, if not all, into those tracks that result in the least number of cross conflict misses.

Table 3.1 shows the track with conflict footprint $(1, 2)$ for array pair (X, Y). Note that different tracks can result in different numbers of cross conflict misses. Among all possible tracks in the table, track $t = y_1 x_1 y_2$ results in the least number of cross conflict misses $(m_X(t) + m_Y(t) = 1.5)$. We thus choose it as the desirable track and try to convert other tracks into this track. Suppose that lines y_1 and y_2 are from array sections Yi and Yj, respectively, where $i, j \in \{1, 2, 3, 4\}$. Recall that in this example, each inner iteration references the X array section and one of the Y array sections alternately and consecutively from beginning to end. With no knowledge of which array section is referenced first, the following two conditions must be satisfied so that a cache set can have track $y_1 x_1 y_2$. First y_1 is closer to the beginning of array section Yi than x_1 is to the beginning of its section, but x_1 is closer than y_2 is in section Yj. We say that such a cache set is covered by the *desirable data layout* since the cache set can have the desirable track if the data are referenced appropriately. Second, array sections Yi and Yj must be referenced in the first and last inner iterations, respectively. If these two conditions are satisfied, then all references to X in this cache set will occur after y_1 and before y_2.

Suppose the data can be placed in the cache by the array padding and initial address adjustment techniques as in Figure 4.4a and 4.4b for the odd and even numbered coverage loop iterations, respectively. Then, cache sets 1 to 6 for the odd numbered coverage loop iterations and 9 to 14 for even numbered coverage loop iterations are all covered by the desirable data layout. If we trace the references in Figure 4.4a and 4.4b, then we can find that cache sets 1 to 6 in Figure 4.4a indeed have the desirable track $y_1 x_1 y_2$. However, cache sets 9 to 14 in Figure 4.4b have track $x_1 y_1 y_2 x_1$ instead of $y_1 x_1 y_2$. The reason is that in an even numbered coverage loop iteration $Y2$, which contains y_1, and $Y3$, which contains y_2, are not referenced in the first and last inner loop iterations. To solve this problem, we modify the program so that in the even numbered coverage loop iterations, inner loop iterations will reference the four Y array sections in the order $Y3$, $Y4$, $Y1$, and $Y2$, as shown in Figure 4.4b. This changes the tracks of cache sets 9 to 14 to $y_1 x_1 y_2$, as desired. We call this technique for satisfying the second condition *access resequencing.*

Note that in this example, since the alignment insures that all array sections of X and Y are aligned on cache line boundaries and no two of them begin in the same cache set, there will be no ping-pong conflict misses. In general, this method can eliminate the self and ping-pong conflict misses and reduce the cross conflict misses. From Section 3.4.2, we know that the original number



(a) Odd Coverage Loop Iterations (b) Even Coverage Loop Iterations

Figure 4.4: Access resequencing to reduce cross conflict misses.

of misses per coverage loop iteration is equal to 27.5. By counting misses after the application of this method, we find that the number of misses per average coverage loop iteration is equal to 19, which is a 31% improvement.

4.3 Properties of the Layout Techniques

Before we address the layout techniques themselves, we introduce some general background on congruence theory. The basic definitions, Definition 4.3.1, 4.3.2, and 4.3.3, and the main theorem, Theorem 4.3.1 are summarized from [53] and [54]. A congruence is a statement about divisibility. Congruence theory employs a powerful notation that has lead to many known theorems. Therefore, even though the theorems needed in our discussion could be derived from the remainder properties of integer division, much of what we need already exists among the theorems of congruence theory.

Definition 4.3.1 Congruence
If a nonzero, integer h divides the difference $a - b$, we say that a is congruent to b modulo h *and write* $a \equiv b \pmod{h}$.

This definition is equivalent to saying that $a \equiv b \pmod{h}$ if and only if a and b have the same remainder when divided by h. Since there are h different remainders, $0, 1, 2, \ldots, h-1$, when a number is divided by h, the set of integers can be partitioned into h equivalent classes, called *residue classes*.

Definition 4.3.2 Residue
If $a \equiv b \pmod{h}$ *then b is called a* residue *of a modulo h, and a and b are in the same* residue class.

Definition 4.3.3 Complete Residue System
A set $\{a_1, a_2, \ldots, a_h\}$ *is called a* complete residue system modulo h *if for every integer b there is exactly one a_i in the set such that* $b \equiv a_i \pmod{h}$.

According to this definition, $0, 1, 2, \ldots, h-1$ is a complete residue system modulo h.

Theorem 4.3.1 *Let k be an integer. If $a_1, a_2, \ldots, a_h$ is a complete residue system modulo h and $\gcd(k,h) = 1$, then $ka_1, ka_2, \ldots, ka_h$ is also a complete residue system modulo h.*

Let r be the *remainder* of $a \div h$, represented as $r = a \bmod h$. By definition 4.3.1, a is congruent to r modulo h. Note that the complete residue system modulo h, $\{0, 1, 2, \ldots, h-1\}$ is also the set of all possible remainders when an integer is divided by h. Moreover, if $r \in \{0, 1, 2, \ldots, h-1\}$, and $a \equiv r \pmod{h}$, then $r = a \bmod h$, *i.e.*, the notation $a \bmod h$ is used to represent the smallest nonnegative number congruent to *a modulo h*.

Most caches map a memory address, m, to a cache set, s, using the function, $s = \lfloor m/L \rfloor \bmod S$, where L and S are the line size and the number of sets in the cache, respectively. The following theorems which discuss some properties of the binary operation, *mod* (*remainder*), will help in understanding the set distribution when mapping a particular data layout into the cache. These theorems can easily be proved from Theorem 4.3.1.

Theorem 4.3.2 *Let $H = \{ik \bmod h | 0 \le i \le h-1\}$ and $R = \{0, 1, 2, \ldots, h-1\}$. If $\gcd(k,h) = 1$, then $H = R$.*

Proof: Since $R = \{0, 1, 2, \ldots, h-1\}$ is a complete residue system modulo h, from Theorem 4.3.1, the set, $\{ik | 0 \le i \le h-1\}$, is also a complete residue system modulo h. Therefore, for any $i \in \{0, 1, 2, \ldots, h-1\}$, there is exactly one element $r \in R$, such that $r \equiv ik \pmod{h}$, *i.e.*, $r = ik \bmod h$. Since both sets have h elements, we have

$$\{ik \bmod h | 0 \le i \le h-1\} = \{0, 1, 2, \ldots, h-1\}$$

□

Theorems 4.3.2 deals with the situation when $\gcd(k,h) = 1$. Note that if $d = \gcd(k,h)$, then $\gcd(k/d, h/d) = \gcd(k,h)/d = 1$. Using this fact, we can easily generalize the theorem and adapt the proof above for the case $\gcd(k,h) = d$.

Theorem 4.3.3 *Let $H = \{ik \bmod h | 0 \le i \le h/d - 1\}$ and $P = \{0, d, 2d, \ldots, (h-2d), (h-d)\}$. If $d = \gcd(k,h)$, then $H = P$.*

Now, let's show a simple application of Theorem 4.3.3. Suppose we want to know which sets will be occupied by the first element of the columns of an $N_r \times N_c$ array. Suppose that the array starts at address 0, and that $N_c = S/d$ and $d = \gcd(N_r/L, S)$. Then the addresses of the first elements of the columns will be $0, N_r, 2N_r, \ldots, (N_c - 1)N_r$. Applying Theorem 4.3.3, the cache sets that these addresses map to will be $0, d, 2d, \ldots, (S-2d), (S-d)$, not necessarily in that order. Note that if $N_c > S/d$, more than one of these elements will map into some of these sets, *i.e.*, the sequence of set numbers will repeat in the same order with period S/d as N_c grows. The following theorem describes this periodic behavior.

Theorem 4.3.4 *If $d = \gcd(k,h)$ and $T = h/d$, then for two nonnegative integers i and j,*

$$(i + jT)k \bmod h = ik \bmod h$$

Proof:

$$\begin{aligned}((i+jT)k) \bmod h &= (ik \bmod h + jTk \bmod h) \bmod h \\ &= (ik \bmod h + \frac{jhk}{d} \bmod h) \bmod h\end{aligned}$$

Since d divides k, jk/d is an integer. Hence, $\frac{jhk}{d} \bmod h = 0$. As a result, we have

$$(i+jT)k \bmod h = ik \bmod h$$

□

Consider a more general case in which the array is allowed to start at address m, where m can be greater than 0 but must be a multiple of L, *i.e.* the array must be aligned on a cache line boundary. Then the addresses of the first elements of the columns become $m, m+N_r, m+2N_r, \ldots, m+(N_c-1)N_r$. It can be shown by applying the next theorem (Theorem 4.3.5) that the cache sets that these addresses map to are $m_0, m_0+d, m_0+2d, \ldots, m_0+(S-2d), m_0+(S-d)$, not necessarily in that order, where $m_0 = (m/L) \bmod d$. Since m/L is an integer, the list is just uniformly shifted from those in previous paragraph.

Theorem 4.3.5 *Let* $H = \{(p+ik) \bmod h | 0 \le i \le h/d - 1\}$ *and* $P = \{p_0, p_0+d, p_0+2d, \ldots, p_0+(h-2d), p_0+(h-d)\}$. *If* $d = \gcd(k,h)$ *and* $p_0 = p \bmod d$, *then* $H = P$.

Proof: Let $q = h/d$. According to Theorem 4.3.3, for any integer $i \in \{0, 1, 2, \ldots, q-1\}$, there exists one and only one integer $j \in \{0, 1, 2, \ldots, q-1\}$ such that $ik \bmod h = jd$, and similarly for any j there is exactly one i. Hence,

$$(p+ik) \bmod h = (p \bmod h + ik \bmod h) \bmod h = (p \bmod h + jd) \bmod h$$

for some j. Therefore,

$$\{(p+ik) \bmod h | 0 \le i \le q-1\} = \{(p \bmod h + jd) \bmod h | 0 \le j \le q-1\} \tag{4.1}$$

Let $p \bmod h = ud + p_0$, where u is an integer and $0 \le p_0 \le d-1$. Then the right hand side of (4.1) equals

$$\{(p_0 + (u+j)d) \bmod h | 0 \le j \le q-1\}$$

When $u+j = q$, *i.e.*, $j = q-u$, then since $p_0 < d \le h$ we have

$$(p_0 + (u+j)d) \bmod h = (p_0 + qd) \bmod h = (p_0 + h) \bmod h = p_0 \tag{4.2}$$

For any j such that $q-u < j \le q-1$,

$$\begin{aligned}(p_0 + (u+j)d) \bmod h &= p_0 + (u+j)d - h = p_0 + (u+j-q)d = p_0 + j'd \\ & \quad \textit{where } 1 \le j' \le u-1.\end{aligned} \tag{4.3}$$

Similarly, for any j such that $0 \leq j < q - u$,

$$\begin{aligned}(p_0 + (u+j)d) \bmod h &= p_0 + (u+j)d = p_0 + j'd \\ &\quad \textit{where } u \leq j' \leq q-1.\end{aligned} \tag{4.4}$$

Combining (4.2), (4.3), and (4.4), we have

$$\begin{aligned} & \{(p_0 + (u+j)d) \bmod h | 0 \leq j \leq q-1\} \\ = & \{p_0, p_0 + d, \ldots, p_0 + (u-1)d, p_0 + ud, p_0 + (u+1)d, \ldots, p_0 + (q-1)d\} \\ = & \{p_0, p_0 + d, p_0 + 2d, \ldots, p_0 + (h-d)\}\end{aligned}$$

Finally,

$$p_0 = (p \bmod h) \bmod d = (p \bmod (qd)) \bmod d = p \bmod d$$

So,

$$H = P$$

□

From Theorems 4.3.4 and 4.3.5, we have the following theorem. Note that the periodic characteristic also exists in this general case.

Theorem 4.3.6 *Let* $H = \{(p+ik) \bmod h | 0 \leq i \leq n, n \geq h/d - 1\}$ *and* $P = \{p_0, p_0 + d, p_0 + 2d, \ldots, p_0 + (h-2d), p_0 + (h-d)\}$. *If* $d = \gcd(k, h)$ *and* $p_0 = p \bmod d$, *then* $H = P$. *In addition, if* $T = h/d$, *then* $(p + (i + jT)k) \bmod h = (p + ik) \bmod h$.

4.4 A Case Study: Reduce Cache Misses in the Blocked Matrix Multiply Program

In this section, we use the blocked matrix multiply program (Figure 4.5) as an example and demonstrate how each type of conflict miss in the program can be eliminated or reduced using the techniques illustrated in Section 4.2. We adopt the same assumptions as those in Chapter 3, which include: i) the leading dimension of an array and the leading dimension of a block are each a multiple of the line size L, and ii) array and block columns start at cache line boundaries. In addition, we assume that each of S, L, and C is a power of 2. Furthermore, although Section 4.2 shows that these methods are applicable to set associative caches, we only consider direct mapped caches here.

4.4.1 Array Padding to Eliminate Self Conflict

We have shown that only blocks of array R can cause self conflict misses in the blocked matrix multiply program. Suppose array R has dimension $N_r \times N_c$. Consider an R block of size $B_r \times B_c$. Without loss of generality, let m be the starting address of the array block. Then, since the starting addresses of every pair of successive block columns are N_r apart, the block columns of the block start at addresses m, $m + N_r$, $m + 2N_r$, …, and $m + (B_c - 1)N_r$, respectively, and begin in cache set

$$s_i = \left\lfloor \frac{m + iN_r}{L} \right\rfloor \bmod S \tag{4.5}$$

```
DO 100 KK = 1, $N_c$, $B_c$
   DO 200 II = 1, $N_r$, $B_r$
      do 220 J = 1, $N_c$
         do 240 K = KK, MIN(KK+$B_c$-1,$N_c$)
            DO 300 I = II, MIN(II+$B_r$-1,$N_r$)
               Q(I, J) = Q(I, J) + R(I, K) * P(K, J)
            END DO
         END DO
      END DO
   END DO
END DO
```

Figure 4.5: Blocked matrix multiplication code (rectangular blocks).

for $i = 0, 1, 2, \ldots, B_c - 1$. Recall that $\mathcal{S}_b$ is the multiset of cache set numbers mapped to by the starting addresses of the block columns of a block, and δ_b is the minimum set distance of $\mathcal{S}_b$, which is defined as the minimum difference among all pairs of cache set numbers in $\mathcal{S}_b$, where the difference of cache sets s_i and s_j can be mathematically defined as $\min\{(s_i - s_j) \bmod S, (s_j - s_i) \bmod S\}$. Under the assumptions that a block column, and thus a block, starts at a cache line boundary and that N_r is a multiple of L, we can remove the floor operation in Equation (4.5). Hence,

$$\mathcal{S}_b = \{\frac{m + iN_r}{L} \bmod S | i = 0, 1, 2, \ldots, (B_c - 1)\} \tag{4.6}$$

To avoid self conflict, it must be that

$$\delta_b \geq B_r / L \tag{4.7}$$

Note that Equation (4.7) can be satisfied by either increasing δ_b or decreasing B_r. In addition, from Equation (4.6) and the definition of δ_b, we see that δ_b, for a given cache organization, (D, S, L), is in turn determined by the leading dimension (N_r) and B_c. Hence, adjusting the leading dimension, or B_c, or B_r can each be used to eliminate self conflict misses. However, since B_r and B_c are generally chosen to get a reasonable size block relative to the total cache size, C, and the reference behavior of the application, it is better to let them be chosen for that purpose. Hence, we primarily focus on adjusting the leading dimension, *i.e.* array padding, to eliminate self conflict misses[2]. Recall that we have assumed that the block size, $B_r \times B_c$, is no greater than the cache size, C. In addition, we only discuss the cases where $B_c \geq 2$ here since if $B_c = 1$, then the elements of the only block column should reside in contiguous cache space (Fortran data layout convention), and thus there would be no self conflict misses.

In this section, we present two methods, a search algorithm and an analytical approach for choosing a leading dimension so that $\delta_b \geq B_r / L$, thereby ensuring no self conflict misses. Although both of these methods determine such a self-conflict-free leading dimension by considering only one array block of size $B_r \times B_c$, with this leading dimension, each other array block of the same blocked array will also be self-conflict-free. For a blocked array, since two array blocks of the same block size will differ only in their starting addresses, m, and since δ_b is not affected by m, if one array block is self-conflict-free, then all array blocks with the same dimensions are self-conflict-free. Note, however, that in addition to the $B_r \times B_c$ array blocks, there may be array blocks with small B_c or small B_r or both along the bottom or the right side of the array. However, in comparison

---

[2] In the analytical method discussed below we may finally also need to fine-tune B_c.

```
1.  Let $N'_r = N_r$
2.  For the starting address of each block column
    - Compute $s_i$, the cache set number that the address maps to
    - $\mathcal{S}_b = \mathcal{S}_b \cup \{s_i\}$
3.  Compute the minimum set distance of $\mathcal{S}_b$, $\delta_b$.
4.  If ($\delta_b < B_r/L$) then
        $N'_r = N'_r + L$; Go To Step 2
5.  $N'_r$ is the solution.
```

Figure 4.6: A self-conflict-free leading dimension search algorithm.

with the δ_b and B_r/L associated with the $B_r \times B_c$ block, these array blocks result in either larger δ_b or smaller B_r/L (or both). Therefore, they are also self-conflict-free. Consequently, it is sufficient to compute a self-conflict-free leading dimension from just one $B_r \times B_c$ array block.

We now introduce a theorem that describes some properties of δ_b. These properties will be used in later discussion.

Theorem 4.4.1 Properties of the minimum set distance

1. *The minimum set distance, δ_b, decreases or remains the same as B_c increases.*
2. *Let $d = \gcd(N_r/L, S)$.*
    (a) *If $B_c = S/d$, $\delta_b = d$.*
    (b) *If $B_c > S/d$, then $\delta_b = 0$.*

Proof:

1. When B_c increases, new elements are added into $\mathcal{S}_b$. Since δ_b is defined as the minimum pairwise difference of elements in $\mathcal{S}_b$, adding a new element will never increase δ_b.
2. (a) Let $p = m/L$, $k = N_r/L$, and $h = S$. If $d = \gcd(N_r/L, S) = \gcd(k, h)$ and $B_c = S/d = h/d$, then from Theorem 4.3.5 $\mathcal{S}_b = \{s_0, s_0 + d, s_0 + 2d, \ldots, s_0 + (h/d - 1)d\}$, where $s_0 = p \bmod h = m/L \bmod S$. Hence, $\delta_b = d$.
    (b) According to Theorem 4.3.6, the sequence of values of $\frac{m+iN_r}{L} \bmod S$, as i increases, repeats every S/d elements. Hence, $s_{S/d} = s_0$, $s_{S/d+1} = s_1$, and so on. Therefore, $\delta_b = 0$.

□

A Search Algorithm for a Self-Conflict-Free Leading Dimension

For an array with leading dimension N_r and a block of size $B_r \times B_c$, Figure 4.6 presents a search algorithm for finding a new leading dimension so that the block results in no self conflict misses in a direct mapped cache. Beginning with the given leading dimension, N_r, the algorithm computes $\mathcal{S}_b$ and the corresponding δ_b. If $\delta_b < B_r/L$, then the leading dimension is increased by L, and $\mathcal{S}_b$ and δ_b are recomputed. Otherwise, the current leading dimension, N'_r, is a self-conflict-free leading dimension of the array, and the search stops. The purpose of increasing the leading dimension with a step size of L elements is to maintain a leading dimension that is a multiple of L.

Example 4.4.1 For the example in Figure 4.1a, the search algorithm will result in $N_r' = 24$, *i.e.* padding the end of each column of the array with 2 dummy elements (2 cache lines). With this new leading dimension, $\mathcal{S}_b = \{0, 8, 16, 24, 32, 40, 48, 56\}$, and $\delta_b = 8$. Since $B_r/L = 7 < \delta_b$, there is no self conflict, as shown in Figure 4.1c. □

The search algorithm computes and checks δ_b for every leading dimension that is a multiple of L, starting from N_r and increasing until the condition $\delta_b \geq B_r/L$ is satisfied. Since $\delta_b \geq B_r/L$ is a necessary and sufficient condition for no self conflict in a directed mapped cache, the search algorithm can therefore find the minimum amount of padding that eliminates self conflict; however, the search algorithm is generally slow. On the other hand, the analytical method proposed below is fast, but may not produce the minimum possible padding because it uses an over-constrained condition. Furthermore, the analytical solution guarantees that δ_b will never be less than B_r/L unless δ_b becomes 0 when we increase B_c. This property will ease the implementation of our method for reducing cross conflict misses.

An Analytical Solution for a Self-Conflict-Free Leading Dimension

In the analytical method, since we do not know of any formula for δ_b that can be used to get a self-conflict-free leading dimension analytically, we first let $d = \gcd(N_r/L, S)$, and assume that

$$\delta_b = d \tag{4.8}$$

Thus we are not concerned here with cases where $B_c > S/d$, in which $\delta_b = 0$. Furthermore, although the analytic solution will be self-conflict free for cases where $B_c < S/d$, *i.e.* those in which we may have $\delta_b > d$, we do not attempt to optimize the solution for these cases.

Then, we derive a self-conflict-free leading dimension, N_r', by enforcing the condition that

$$d' = \gcd(N_r'/L, S) \geq B_r/L \tag{4.9}$$

with the constraints:

(1) $d' = \gcd(N_r'/L, S)$ is the minimum divisor of S such that $d' \geq B_r/L$, and

(2) N_r' is the minimum solution that satisfies constraint (1) and is greater than or equal to N_r.

Finally, if necessary, we adjust B_c.

The first constraint attempts to reduce the chance of adjusting B_c (as discussed below), while the second constraint finds the minimum padding that satisfies the first constraint. From Theorem 4.4.1, we know that if the number of block columns, B_c, is greater than S/d', then even if $d' \geq B_r/L$, the actual minimum set distance will be 0 ($< B_r/L$). Thus, the block will still cause self conflict misses. Since assumption (4.8) and condition (4.9) do not consider B_c, the new leading dimension N_r' may result in a S/d' that is less than B_c. If $S/d' < B_c$, we make N_r' a feasible solution with no self conflict misses by simply reducing B_c to S/d'. As mentioned before, values of B_r and B_c are generally chosen for other purposes; thus it is best that the readjustment of B_c is minimal, if not zero. The first constraint, which results in the maximum possible S/d', is used for this purpose.

We now discuss some properties of the analytical solution and derive a closed form expression for N_r'. Considering only the condition $d' = \gcd(N_r'/L, S) \geq B_r/L$, there can be many N_r' solutions for a given d', *i.e.* many N'_r can satisfy $\gcd(N'_r/L, S) = d'$, and there may be more than one feasible d'. We denote the set of all these N'_r solutions as Ω, *i.e.*

$$\Omega = \{N'_r | \gcd(N'_r/L, S) \geq B_r/L\}$$

Let d_0 be the value of d' that satisfies the first constraint. Then the first constraint will narrow down the domain of N'_r solutions to

$$\Omega_0 = \{N'_r | \gcd(N'_r/L, S) = d_0\} \subseteq \Omega$$

Note that for any $N'_r \in \Omega$, since we have assumed that S is always a power of 2, $d' = \gcd(N'_r/L, S)$ will also be a power of 2. Hence, $d_0 = 2^{\lceil \lg(B_r/L) \rceil}$, where lg is the logarithm operation with base 2.

Example 4.4.2 Suppose $(D, S, L) = (1, 64, 1)$, $B_r = 7$, and $N_r = 22$. Then $\Omega = \{N'_r | \gcd(N'_r, 64) \geq 7\}$, which is all nonnegative integer multiples of 8. However, since $d_0 = 2^{\lceil \lg(B_r/L) \rceil} = 8$ and all larger divisors of S are nonnegative integer multiples of $16 = 2 \times 8$, Ω_0 is the set of all other, *i.e.* odd nonnegative integer, multiples of 8. Considering now the second constraint, which requires N'_r to be the smallest element of Ω_0 that satisfies $N'_r \geq N_r$, the analytical solution yields $N'_r = 24$. □

We now show that $N'_r \in \Omega_0$ if and only if $\frac{N'_r}{Ld_0}$ is an odd integer. Recall that $B_r \times B_c \leq C$ and $B_c \geq 2$. Thus, we have $B_r \leq C/2$, and

$$d_0 = 2^{\lceil \lg(B_r/L) \rceil} \leq 2^{\lceil \lg(C/(2L)) \rceil} = 2^{\lceil \lg(S/2) \rceil} = S/2$$

Because both S and d_0 are each a power of 2 and $d_0 \leq S/2$, the value of $\frac{S}{d_0}$ must be a positive integer power of 2. If $N'_r \in \Omega_0$, then $\gcd(N'_r/L, S) = d_0$, which implies that $\frac{N'_r}{Ld_0}$ and $\frac{S}{d_0}$ are relatively prime. Thus, $\frac{N'_r}{Ld_0}$ must be odd since $\frac{S}{d_0}$ is a nonnegative integer power of 2. Conversely, if $\frac{N'_r}{Ld_0}$ is an odd integer, then since $\frac{S}{d_0}$ is a nonnegative integer power of 2, $\gcd(\frac{N'_r}{Ld_0}, \frac{S}{d_0}) = 1$. It follows that $\gcd(N'_r/L, S) = d_0$, *i.e.* $N'_r \in \Omega_0$. As a result,

$$\Omega_0 = \{iLd_0 | i \text{ is an odd integer}\}$$

The smallest N'_r in Ω_0 satisfying $N'_r \geq N_r$ corresponds to the smallest $\frac{N'_r}{Ld_0} \geq \frac{N_r}{Ld_0}$, *i.e.* the smallest $i \geq \frac{N_r}{Ld_0}$. Since i is an odd integer, if $\left\lceil \frac{N_r}{Ld_0} \right\rceil$ is odd, then the smallest i equals $\left\lceil \frac{N_r}{Ld_0} \right\rceil$; otherwise, it equals $\left\lceil \frac{N_r}{Ld_0} \right\rceil + 1$. Consequently, the analytical solution for the self-conflict-free leading dimension is

$$N'_r = \begin{cases} \left\lceil \frac{N_r}{Ld_0} \right\rceil \times L \times d_0 & \text{if } \left\lceil \frac{N_r}{Ld_0} \right\rceil \text{ is odd} \\ (\left\lceil \frac{N_r}{Ld_0} \right\rceil + 1) \times L \times d_0 & \text{otherwise} \end{cases} \tag{4.10}$$

In summary, given a cache organization, (D, S, L), an array of leading dimension N_r, and a block of size $B_r \times B_c$, we can directly compute a self-conflict-free leading dimension, N'_r, based on Equation (4.10). If $B_c > S/d_0$, despite the fact that d_0 was chosen according to the first constraint to be as small as possible, we then reduce B_c to S/d_0.

Using Equation (4.10) we can now solve Example 4.4.2 very easily and use that solution to eliminate the self conflict in Figure 4.1(c), as seen in the following example.

Example 4.4.3 In the example of Figure 4.1(a), $d_0 = 2^{\lceil \lg(7/1) \rceil} = 2^3 = 8$. Since $\left\lceil \frac{N_r}{Ld_0} \right\rceil = 3$ is odd, from Equation (4.10), $N'_r = \left\lceil \frac{N_r}{Ld_0} \right\rceil \times L \times d_0 = \left\lceil \frac{22}{1 \cdot 8} \right\rceil \times 1 \times 8 = 24$. Because $S/d_0 = 8$ is equal to

B_c, no adjustment of B_c is necessary.

The resulting cache mapping is shown in Figure 4.1(d). The cache appears to be equally divided into 8 segments with leading boundaries at cache sets 0, 8, 16, 24, 32, 40, 48, and 56, respectively. Each block column then maps to the first 7 cache sets of a segment, and there is no conflict between block columns. □

Square Blocks If a program requires square blocks, *i.e.* $B_r = B_c$, then the analytical method may not result in a feasible solution since now B_c is no longer allowed to change arbitrarily. In this section, we discuss the largest size square block that still permits the analytical method to result in a feasible solution with no self conflict misses.

Let $B_r = B_c = B$, and $C = 2^\gamma$. From the derivation of the analytical solution, we know that the analytic solution for N_r' must satisfy $\gcd(N_r'/L, S) = 2^{\lceil \lg(B/L)\rceil}$. To ensure no self conflict, the number of block columns must never be greater than $S/\gcd(N_r'/L, S)$, *i.e.*

$$B \leq \frac{S}{2^{\lceil \lg(B/L)\rceil}}$$

Multiplying both sides by $2^{\lceil \lg(B/L)\rceil}$, and then taking logarithm with base 2, we have

$$\lg B + \lceil \lg B - \lg L\rceil \leq \lg S$$

Since L is a power of 2, $\lceil \lg B - \lg L\rceil = \lceil \lg B\rceil - \lg L$. As a result,

$$\lg B + \lceil \lg B\rceil \leq \lg S + \lg L = \lg C = \gamma$$

Now let $\epsilon = \lceil \lg B\rceil - \lg B$, and note that $0 \leq \epsilon < 1$. Substituting $\lceil \lg B\rceil - \epsilon$ for $\lg B$, we obtain

$$2\lceil \lg B\rceil - \epsilon \leq \gamma$$

It follows that

$$\lceil \lg B\rceil \leq \begin{cases} \frac{\gamma}{2} + \frac{\epsilon}{2} & \text{if } \gamma \text{ is even} \\ \frac{\gamma-1}{2} + \frac{\epsilon+1}{2} & \text{if } \gamma \text{ is odd} \end{cases} \tag{4.11}$$

Note that $\lceil \lg B\rceil$ is an integer, and the γ terms of Equation (4.11) are also integers. If γ is even, since $0 \leq \frac{\epsilon}{2} < \frac{1}{2}$, we have $\lceil \lg B\rceil \leq \frac{\gamma}{2}$. The maximum B that satisfies $\lceil \lg B\rceil \leq \frac{\gamma}{2}$ is equal to $2^{\frac{\gamma}{2}}$, which is $\sqrt{C}$. Similarly, if γ is odd, since $\frac{1}{2} \leq \frac{\epsilon+1}{2} < 1$, $\lceil \lg B\rceil \leq \frac{\gamma-1}{2}$. Thus, the maximum B that satisfies this condition is equal to $2^{\frac{\gamma-1}{2}}$, which is $\sqrt{C/2}$. As a result, if γ is even, then the analytical method can generate a self-conflict-free leading dimension for any square block whose size is no greater than the cache size. However, if γ is odd, then the analytical method can only generate a self-conflict-free leading dimension for any square block whose size is no greater than half of the cache size.

Padding Overhead The padding overhead of the padded array is defined as $N_r' - N_r$. The analytical solution, Equation (4.10), is the least N_r' that is no less than N_r that satisfies $\gcd(N_r'/L, S)$ $= 2^{\lceil \lg(B_r/L)\rceil} = d_0$. Hence, unless the original leading dimension, N_r, results in $\gcd(N_r/L, S) = d_0$, the analytical method will always introduce some amount of padding. However, unlike the search algorithm in Figure 4.6, this padding may not be the minimum possible.

Note that from Theorem 4.4.1, for any choice of N_r, we know that the minimum set distance for a given cache organization is either 0 or no less than $\gcd(N_r/L, S)$. It may happen that $\gcd(N_r/L, S)$ is less than B_r/L, whereas the actual minimum set distance is not less than B_r/L. Hence, the analytical solution, which requires $\gcd(N_r/L, S) \geq B_r/L$, may generate unnecessarily large padding. In addition, it may also happen that $\gcd(N_r/L, S)$ is greater than B_r/L, but is not equal to d_0. Thus, the first constraint, which requires using the minimum possible $\gcd(N_r/L, S)$ that is no less than B_r/L, may also cause unnecessarily large array padding.

Example 4.4.4 If $B_c = 3$ in Figure 4.1(a), then $\mathcal{S}_b = \{0, 22, 44\}$, and the actual δ_b equals 20, which is greater than $B_r/L = 7$. Hence, the original leading dimension is already a self-conflict-free leading dimension. However, since for $N_r = 22$, $\gcd(N_r/L, S) = 2 < B_r/L$, the analytical method will attempt to find a new leading dimension. According to Equation (4.10), the new leading dimension is 24, *i.e.* padding overhead = 2. Alternatively, if the initial N_r were 48 with $B_c = 3$, then $\mathcal{S}_b = \{0, 32, 48\}$. In this case, even though $\gcd(N_r/L, S) = 16$ is already greater than B_r/L and no padding is needed for self conflicts, the analytical method will nevertheless pad the leading dimension up to 56, *i.e.* padding overhead = 8, since $\gcd(N_r/L, S) \neq 2^{\lceil \lg(B_r/L) \rceil} = 8$. In both cases, the search algorithm (Figure 4.6) will use the original leading dimension as the self-conflict-free leading dimension. □

Suppose that padding is necessary. Consider any $N_r' \in \Omega$, *i.e.* any N_r' that satisfies the condition that $\gcd(N_r'/L, S) = d' \geq B_r/L$. This N_r' must be a multiple of Ld'. However, since d' cannot be 1 for $B_r > L$, Ω may not contain all nonnegative integer multiples of L. Hence, in general, the minimum self-conflict-free leading dimension may not be in Ω. Because the analytical solution must be in Ω, a nonminimal self-conflict-free leading dimension may result.

Consider N_r' solutions that are in Ω. Although the first constraint reduces the solution domain to a subset of Ω, we can show that when $\left\lceil \frac{N_r}{Ld_0} \right\rceil$ is odd, Equation (4.10) actually generates an N_r' that equals the minimum N_r' solution in Ω. Note that since d' and d_0 are each a power of 2 and $d' \geq d_0$, d' must be a multiple of d_0. Thus any feasible $N_r' \in \Omega$ that is a multiple of Ld' must also be a multiple of Ld_0. The minimum N_r' that is no less than N_r and is a multiple of Ld_0 is $\left\lceil \frac{N_r}{Ld_0} \right\rceil \times L \times d_0$, which is exactly equal to the solution of Equation (4.10) if $\left\lceil \frac{N_r}{Ld_0} \right\rceil$ is odd. Since the solution of Equation (4.10) must be in Ω, we can conclude that if $\left\lceil \frac{N_r}{Ld_0} \right\rceil$ is odd, Equation (4.10) does generate a solution that is equal to the minimum N_r' in Ω. When $\left\lceil \frac{N_r}{Ld_0} \right\rceil$ is even, if the aforementioned minimum value, $\left\lceil \frac{N_r}{Ld_0} \right\rceil \times L \times d_0$, is in Ω, then the solution resulting from Equation (4.10) is Ld_0 more than the minimum solution in Ω; otherwise, it is also equal to the minimum N_r' in Ω.

Example 4.4.5 Suppose $(D, S, L) = (1, 64, 1)$, $N_r = 44$, $B_r = B_c = 7$, and assume that the block begins in cache set 0. Then, $\mathcal{S}_b = \{0, 4, 8, 24, 28, 44, 48\}$, and $\delta_b = 4$. In this example, the minimum padding is 1, *i.e.* $N_r' = 45$, which results in $\mathcal{S}_b = \{0, 7, 14, 26, 33, 45, 52\}$, and $\delta_b = 7$. The minimum N_r' solution in Ω is 48, which occurs when $d' = 16$. (However, in this case, we need to reduce B_c to $S/d' = 64/16 = 4$.) Our analytical method, which uses minimum $d' = d_0 = 8$, produces $N_r' = 56$ and results in $\mathcal{S}_b = \{0, 16, 24, 32, 40, 48, 56\}$ and $\delta_b = 8$. Note that in this case the analytical solution is $Ld_0 = 8$ more than the minimum N_r' in Ω, but B_c does not need to be reduced from 7. □

In general, the total amount of padding overhead in the padded array, $N_r' - N_r$, resulting from the analytical method, can be derived as follows. Since $\frac{N_r}{Ld_0} \leq \left\lceil \frac{N_r}{Ld_0} \right\rceil \leq \frac{N_r}{Ld_0} + 1$,

$$N_r \leq \left\lceil \frac{N_r}{Ld_0} \right\rceil Ld_0 \leq N_r + Ld_0, \text{ and}$$

$$N_r + Ld_0 \leq \left\lceil \frac{N_r}{Ld_0} \right\rceil Ld_0 + Ld_0 \leq N_r + 2Ld_0$$

Thus, from Equation (4.10), we have

$$\begin{cases} N_r \le N_r' \le N_r + Ld_0 & \text{if } \left\lceil \frac{N_r}{Ld_0} \right\rceil \text{ is odd} \\ N_r + Ld_0 \le N_r' < N_r + 2Ld_0 & \text{if } \left\lceil \frac{N_r}{Ld_0} \right\rceil \text{ is even} \end{cases}$$

It follows that

$$\begin{cases} 0 \le N_r' - N_r < Ld_0 & \text{if } \left\lceil \frac{N_r}{Ld_0} \right\rceil \text{ is odd} \\ Ld_0 \le N_r' - N_r < 2Ld_0 & \text{if } \left\lceil \frac{N_r}{Ld_0} \right\rceil \text{ is even} \end{cases}$$

Therefore, the amount of padding for each column is upper bounded by Ld_0 if $\left\lceil \frac{N_r}{Ld_0} \right\rceil$ is odd or $2Ld_0$ if $\left\lceil \frac{N_r}{Ld_0} \right\rceil$ is even. Since $1 \le d_0 \le S/2$, the padding overhead upper bounds, Ld_0 and $2\ Ld_0$, can never exceed $LS/2 = C/2$ and $2LS/2 = C$, respectively. However, because $d_0 = 2^{\lceil \lg(B_r/L) \rceil}$, a small B_r should yield a small padding overhead upper bound. Hence, to have a small padding overhead, we can choose a block with small B_r. A square block is a common choice for many application programs. In a cache of size 2^γ with an even γ, since B of a square block is no greater than $\sqrt{C}$, the padding overhead upper bound should be no greater than $L\sqrt{C}$ if $\left\lceil \frac{N_r}{Ld_0} \right\rceil$ is odd or $2L\sqrt{C}$ if $\left\lceil \frac{N_r}{Ld_0} \right\rceil$ is even. Similarly, in a cache of size 2^γ with an odd γ, the padding overhead upper bound should be no greater than $L\sqrt{C/2}$ and $2L\sqrt{C/2}$ for odd $\left\lceil \frac{N_r}{Ld_0} \right\rceil$ and even $\left\lceil \frac{N_r}{Ld_0} \right\rceil$, respectively.

Summary

In this section, we introduced a search algorithm and an analytical method for array padding to eliminate self conflict misses. The search algorithm uses the necessary and sufficient condition for no self conflict in a direct mapped cache, $\delta_b \ge B_r/L$, to evaluate every leading dimension that is a multiple of L, starting from the original leading dimension, N_r. Thus, it can find a leading dimension that results in the minimum amount of padding.

On the other hand, the analytical method tries to find a minimum N_r' that is greater than or equal to N_r and satisfies $\gcd(N_r'/L, S) = 2^{\lceil \lg(B_r/L) \rceil} = d_0$. The analytical method immediately produces the closed form solution in Equation (4.10). In general, this method may not generate a leading dimension with the minimum padding, but its padding overhead is upper bounded by Ld_0 (if $\left\lceil \frac{N_r}{Ld_0} \right\rceil$ is odd) or $2Ld_0$ (if $\left\lceil \frac{N_r}{Ld_0} \right\rceil$ is even). In addition, the analytical method may require reducing B_c to get a feasible solution.

For a cache size of $C = 2^\gamma$, the analytical method can always find a solution for a square block if the block size is no greater than C and γ is an even number, or if the block size is no greater than $C/2$ and γ is an odd number.

Generally, the analytical method is much faster than the search algorithm. Furthermore, the analytical solution guarantees that δ_b will never be less than B_r/L unless δ_b is 0 due to a B_c that is too large. This property will ease the implementation of our method for reducing cross conflict misses.

4.4.2 Initial Address Adjustment & Array Padding to Remove Ping-Pong Conflicts

In this section, we describe how to use the initial address adjustment and array padding techniques to eliminate ping-pong conflict misses in the blocked matrix multiplication program. As discussed in Section 4.2, our approach for eliminating the ping-pong conflict misses between two line-boundary-aligned array sections is to adjust their initial addresses so that they do not begin in the same cache set. The difficulty of applying this idea to real programs is that a real program, such as the block matrix multiply, usually involves many array section pairs that can potentially cause ping-pong conflict misses. This section presents a systematic analysis process for analyzing array sections, which can usually result in a simple process that guarantees that no pair of array sections with potential ping-pong conflicts begin in the same cache set, thereby ensuring that there are no ping-pong conflict misses. We present the analyses and solution below.

Classify Array Sections Recall that we use $U_A(i,j)$ to denote the i^{th} block column in column j of array A. In the following discussion, we also use "a:b" to denote the index range from a through b inclusive. As discussed in Section 3.5.1, we know that in this program only arrays Q and R can result in ping-pong conflict misses, and a pair of block columns, $U_Q(i,j)$ and $U_R(i',k)$, can cause ping-pong conflict misses only if $i = i'$. Therefore, to guarantee that the program does not have ping-pong conflict misses, we must ensure that no block column in $U_Q(i, 1 : N_c)$ begins in the same cache set as any block column in $U_R(i, 1 : N_c)$, $\forall i = 1, 2, \ldots, \lceil \frac{N_r}{b_r} \rceil$. Let $\Theta_Q(i)$ and $\Theta_R(i)$ be the sets of the cache set numbers mapped to by the initial addresses of block columns $U_Q(i, 1 : N_c)$ and $U_R(i, 1 : N_c)$, respectively. Then, to have no ping-pong conflict misses, the equalities in (4.12) must hold.

$$\begin{cases} \Theta_Q(1) \cap \Theta_R(1) = \emptyset \\ \Theta_Q(2) \cap \Theta_R(2) = \emptyset \\ \vdots \\ \Theta_Q(\lceil \frac{N_r}{b_r} \rceil) \cap \Theta_R(\lceil \frac{N_r}{b_r} \rceil) = \emptyset \end{cases} \tag{4.12}$$

For any column, because the difference between the initial addresses of two successive block columns is the same for arrays Q and R, if the initial addresses of $U_Q(i,j)$ and $U_R(i,k)$ map to different cache sets, the initial addresses of $U_Q(i+1,j)$ and $U_R(i+1,k)$ will also map to different cache sets. As a result, $\Theta_Q(1) \cap \Theta_R(1) = \emptyset$ implies $\Theta_Q(2) \cap \Theta_R(2) = \emptyset$, $\Theta_Q(3) \cap \Theta_R(3) = \emptyset$, and so on. Hence, we only need to make $\Theta_Q(1) \cap \Theta_R(1) = \emptyset$ true in order to eliminate the ping-pong conflict misses in the matrix multiply program.

Characterize $\Theta_Q(1)$ and $\Theta_R(1)$ Let Q_0 and R_0 be the starting addresses of arrays Q and R, respectively, then we know that

$$\begin{aligned} \Theta_Q(1) &= \{\frac{Q_0 + iN_r}{L} \bmod S \mid 0 \le i \le N_c - 1\} \\ \Theta_R(1) &= \{\frac{R_0 + iN_r}{L} \bmod S \mid 0 \le i \le N_c - 1\} \end{aligned}$$

Theorem 4.3.6 implies that

$$\begin{aligned} \Theta_Q(1) &\subseteq \{s_q + id \mid i = 0, 1, 2, \ldots, S/d - 1\} = \Phi_Q \\ \Theta_R(1) &\subseteq \{s_r + id \mid i = 0, 1, 2, \ldots, S/d - 1\} = \Phi_R \end{aligned}$$

where $d = \gcd(N_r/L, S)$, $s_q = (Q_0/L) \bmod d$, and $s_r = (R_0/L) \bmod d$. Therefore, if $\Phi_Q \cap \Phi_R = \emptyset$, then $\Theta_Q(1) \cap \Theta_R(1) = \emptyset$. Our method thus uses the initial address adjustment and array array padding techniques to make $\Phi_Q \cap \Phi_R$ equal to $\emptyset$. Since Φ_Q and Φ_R are supersets of $\Theta_Q(1)$ and $\Theta_R(1)$, respectively, our solution may be conservative.

Solution It is clear that if $d = 1$, then $\Phi_Q = \Phi_R = \{0, 1, 2, \ldots, S-1\}$. Hence, for $\Phi_Q \cap \Phi_R = \emptyset$, d cannot be 1. If $d = 1$, we apply the array padding technique to both arrays in order to have $d > 1$. Since S is a power of 2, if $d = \gcd(N_r/L, S) = 1$, then N_r/L must be an odd number.

Therefore, we can simply add L to the leading dimensions of the two arrays so that the new N_r/L is an even number, which yields $d \geq 2$. When $d > 1$, Φ_Q and Φ_R actually contain two equal distance (d) arithmetic sequences with first terms s_q and s_r, respectively. Since $0 \leq s_q, s_r < d$, if $s_q \neq s_r$, then Φ_Q and Φ_R must have no common elements, *i.e* $\Phi_Q \cap \Phi_R = \emptyset$. Otherwise, $\Phi_Q = \Phi_R$. In this case, we can adjust the initial addresses of the arrays so that $s_q \neq s_r$. One solution is

$$\begin{cases} Q_0 = Q_0 + L \quad if\ Q_0 > R_0; \quad Q_0 = Q_0 \quad otherwise \\ R_0 = R_0 + L \quad if\ Q_0 < R_0; \quad R_0 = R_0 \quad otherwise \end{cases} \tag{4.13}$$

This solution shifts s_q or s_r by one cache set, and thus $\Phi_Q \cap \Phi_R = \emptyset$.

Algorithm We summarize the above results in the following algorithm that removes all ping-pong conflicts in the blocked matrix multiply program:

1. If $d = \gcd(N_r/L, S) = 1$
    (a) Add L to the leading dimension of arrays Q and R
    (b) Recompute d using the new leading dimension
2. If $s_q = s_r$, apply Equation (4.13)

4.4.3 Access Resequencing, Initial Address Adjustment, and Array Padding to Reduce Cross Conflict Misses

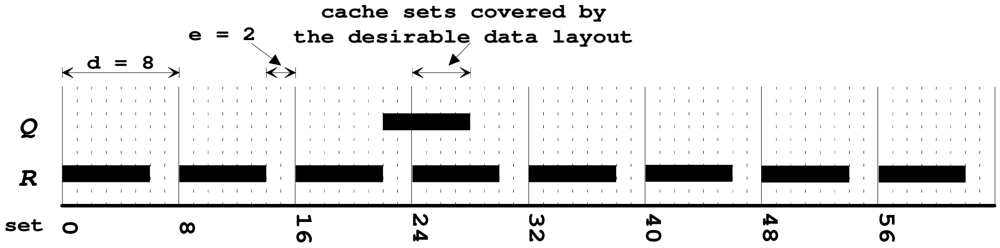
The conflict misses in the matrix multiply program are primarily caused by conflicts between arrays Q and R and between arrays P and R. Since the array padding and initial address adjustment techniques have been discussed in detail in previous sections, we are particularly interested in the case where the access resequencing technique is included in the process of reducing the cross conflict misses. The method including the access resequencing technique will have to take into account how arrays are referenced. Therefore, the same solution may not apply to both Q-R and P-R conflicts since arrays Q and P are referenced differently. We apply the method to reduce the cross conflict misses for Q-R conflicts first, then consider array P in the P-R conflicts.

Apply Array Padding

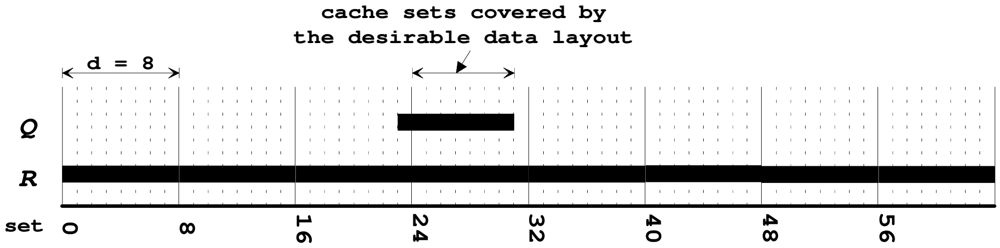
Consider only arrays Q and R. A coverage loop (loop-J) iteration of the program involves one block column of Q and one block of R. As suggested in Section 4.2.3, we first apply the analytical array padding technique to array R so that in a coverage loop iteration a cache set can have a one line footprint of R at most. Furthermore, since one block column is never larger than the cache, a cache set can contain at most one line footprint of Q. Therefore, a conflict cache set will contain one line footprint each of Q and R, and thus may experience tracks r_1q_1, q_1r_1, or $q_1r_1q_1$. Among these possible tracks, track r_1q_1 (or equivalently q_1r_1) can result in the least number of cache misses, and thus is selected as the desirable track. Recall that in an inner loop (loop-K) iteration, the Q block column is referenced with an R block column alternately and consecutively from beginning to end. Suppose that the R block column containing r_1 and the Q block column containing q_1 begin in cache sets s_{r1} and s_{q1}, respectively, and we do not know which block column is referenced first. Then, for a cache set s, a data layout in which $(s - s_{r1}) \bmod S < (s - s_{q1}) \bmod S$ is selected as the desirable data layout for the cache set s.

Apply Initial Address Adjustment

In general, there are three types of cache sets for a coverage loop iteration after we apply the array padding technique: without conflict, conflict with desirable layout (minimal conflict), and conflict with undesirable data layout (could possibly be improved significantly). Applying the



(a) $B_r = 6$, $e = 2$



(b) $B_r = 8$, $e = 0$

Figure 4.7: Suggested alignments of a Q block column relative to an R block column: $(D, S, L) = (1, 64, 1)$, $B_c = 8$, $N'_r = 24$.

initial address adjustment technique to arrays Q and R first maximizes the number of no-conflict cache sets, and then tries to maximize the number of the remaining cache sets that have a desirable data layout.

Recall that the analytical array padding technique guarantees that $d \geq B_r/L$, where $d = \gcd(N'_r/L, S)$, and N'_r is the self-conflict-free leading dimension. Imagine that the direct mapped cache is divided into S/d segments starting from the cache set in which the block begins; each segment covers d cache sets. Then each block column of the R block will map to a distinct segment, but only to the first B_r/L cache sets of that segment. The last $e = (d - B_r/L)$ cache sets in each segment are not mapped to by any element of the R block. Therefore, if $e > 0$, then we can reduce the number of conflict cache sets by mapping the first e lines of a Q block to the last e cache sets of a segment. With this alignment, all conflict cache sets will also be covered by the desirable data layout. On the other hand, if $e = 0$, then all cache sets mapped to by a Q block column are conflict cache sets. The best we can do is to increase the number of cache sets covered by the desirable data layout. This can be accomplished by beginning the first line of a Q block column one cache set ahead of an R block column. Figure 4.7a and Figure 4.7b show the suggested alignments for $e \neq 0$ and $e = 0$, respectively. In both examples, $(D, S, L) = (1, 64, 1)$, $N'_r = 24$, and the R block begins in cache set 0. Note that in the figures, the R block columns begin in cache sets 0, 8, 16, 24, 32, 40, 48, and 56, and one Q block column is arbitrarily selected for the purpose of illustration.

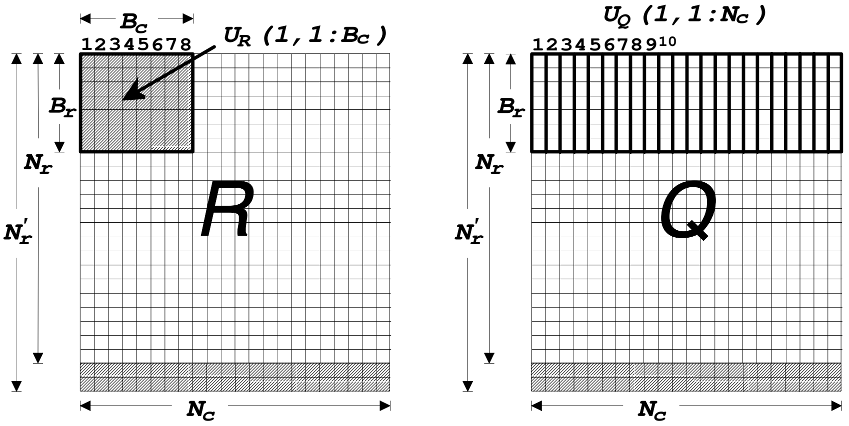
Suppose arrays Q and R begin in cache sets s_q and s_r, respectively. Our method aligns s_q to cache set $(s_r - e) \bmod S$ if $e > 0$ and to cache set $(s_r - 1) \bmod S$ if $e = 0$. Since s_q and s_r, respectively, are also the cache sets in which block columns $U_Q(1,1)$ and $U_R(1,1)$ begin, this alignment directly applies to these two block columns. We will show that in each coverage loop iteration, after changing the leading dimension of array Q to N'_r, the referenced Q block column will be aligned relative to some R block column in the same way that $U_Q(1,1)$ is aligned relative to $U_R(1,1)$, or more generally, relative to a leading boundary of a segment in the same way that s_q is aligned relative to s_r, since some segments will have no R block columns if $B_c < S/d$.

First, consider $B_c = S/d$, *i.e.* every segment is mapped to by one R block column. When $KK = II = 1$, the data set referenced in all of the enclosed coverage loop (loop-J) iterations includes an R block, $U_R(1, 1 : B_c)$, and N_c block columns of Q, $U_Q(1, 1 : N_c)$. Note that in this program, the R block is referenced in every coverage loop iteration. However, a different Q block column is accessed in each coverage loop iteration: $U_Q(1,j)$ is referenced when $J = j$. The initial address adjustment above has actually aligned $U_Q(1,1)$ relative to $U_R(1,1)$ in the coverage loop iteration of $J = 1$. Recall that block columns $U_Q(1,j)$ and $U_R(1,j)$ begin in cache sets $\frac{Q_0+jN'_r}{L} \bmod S$ and $\frac{R_0+jN'_r}{L} \bmod S$, respectively. Note that the difference between the beginning cache sets of two successive Q block columns, $\frac{Q_0+(j+1)N'_r}{L} \bmod S - \frac{Q_0+jN'_r}{L} \bmod S$, is equal to that of two successive R block columns, $\frac{R_0+(j+1)N'_r}{L} \bmod S - \frac{R_0+jN'_r}{L} \bmod S$. Thus, block column $U_Q(1,2)$ will be aligned relative to $U_R(1,2)$ in the same way that $U_Q(1,1)$ is aligned relative to $U_R(1,1)$. Similarly, for any $j = 1, 2, \ldots, N_c$, block column $U_Q(1,j)$ will be aligned relative to block column $U_R(1,j)$ in the same way that $U_Q(1,1)$ is aligned relative to $U_R(1,1)$. From Theorem 4.3.6, we know that the mapping of R repeats every $S/d = B_c$ block columns. Hence, for $j > B_c$, any block column $U_R(1,j)$ must be mapped to the same cache sets as some block column in $U_R(1, 1 : B_c)$. Therefore, although there are only B_c block columns of R, $U_R(1, 1 : B_c)$, involved in this case, we can still have the result that for any $j = 1, 2, \ldots, N_c$, block column $U_Q(1,j)$ is aligned relative to some R block column in the same way that $U_Q(1,1)$ is aligned relative to $U_R(1,1)$.

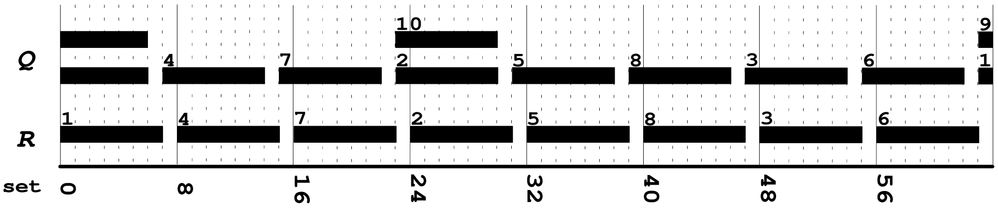
Example 4.4.6 Suppose $(D, S, L) = (1, 64, 1)$, $B_r = 7$, $B_c = 8$, $N'_r = 24$, and $s_r = 0$. Then, $d = 8$, $S/d = 8$, and $e = 1$, we thus align s_q to cache set $(s_r - e) \bmod S = 63$. Figure 4.8a shows the data involved when $KK = II = 1$. Figure 4.8b shows the mapping of block $U_R(1, 1 : B_c)$ and the first 10 block columns of $U_Q(1, 1 : N_c)$, where we can find that every Q block column is aligned relative to an R block column in the same way that $U_Q(1,1)$ is aligned relative to $U_R(1,1)$. In addition, the mapping of Q repeats every $S/d = 8$ block columns, thus, $U_Q(1,9)$ and $U_Q(1,10)$ map to the same cache sets as $U_Q(1,1)$ and $U_Q(1,2)$, respectively. □

When KK is advanced, *e.g.* from 1 to B_c+1 (refer to Figure 4.9a), a new R block, $U_R(1, B_c+1 : 2B_c)$, is referenced. Since this R block is just to the right of the previous one, $U_R(1, 1 : B_c)$, we know that each block column of this block will map to the same cache sets as the corresponding block column of the previous block. Moreover, the same Q block columns, $U_Q(1, 1 : N_c)$, are referenced here. Thus, it is obvious that each Q block column is aligned relative to an R block column in the same way that $U_Q(1,1)$ is aligned relative to $U_R(1,1)$. When II is advanced, *e.g.* from 1 to $B_c + 1$ (refer to Figure 4.9b), the data involved will be $U_R(2, 1 : B_c)$ and $U_Q(2, 1 : N_c)$. The mapping of block column $U_R(2,1)$ is shifted by B_r/L cache sets relative to that of $U_R(1,1)$. Nevertheless, the mapping of $U_Q(2,1)$ is also shifted by B_r/L cache sets relative to that of $U_Q(1,1)$. Hence, $U_Q(2,1)$ should be aligned relative to $U_R(2,1)$ in the same way that $U_Q(1,1)$ is aligned relative to $U_R(1,1)$. Similarly, we see that for any $j = 1, 2, \ldots, N_c$, block column $U_Q(2,j)$ is aligned relative to some R block column in the same way that $U_Q(1,1)$ is aligned relative to $U_R(1,1)$.

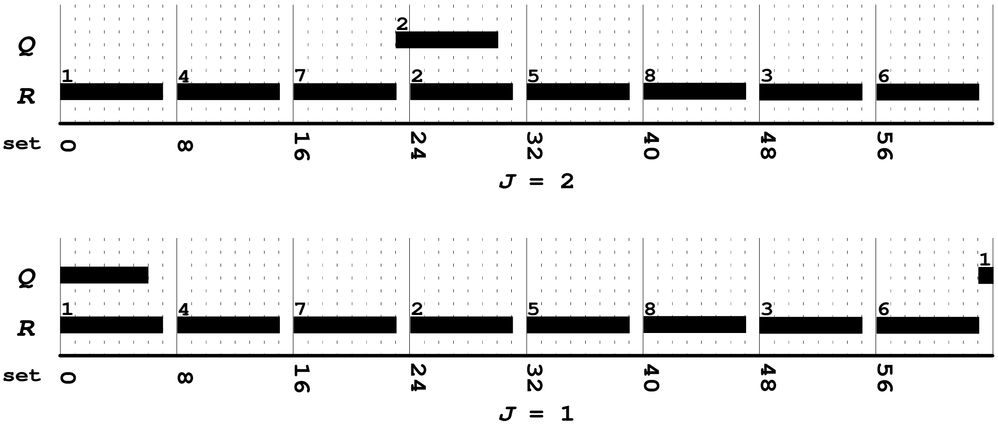
When $B_c < S/d$, some segments may not be mapped to by any block column of R. However, the mappings of other R block columns and Q block columns should remain the same. Therefore, the alignment is similar, except that some Q block columns may be aligned relative to an empty segment. Nevertheless, we can say that the beginning cache set of a Q block column is aligned relative to the leading boundary of a segment in the same way that s_q is aligned relative to s_r. This



(a) The data referenced in the coverage loop iterations enclosed when $KK = II = 1$.

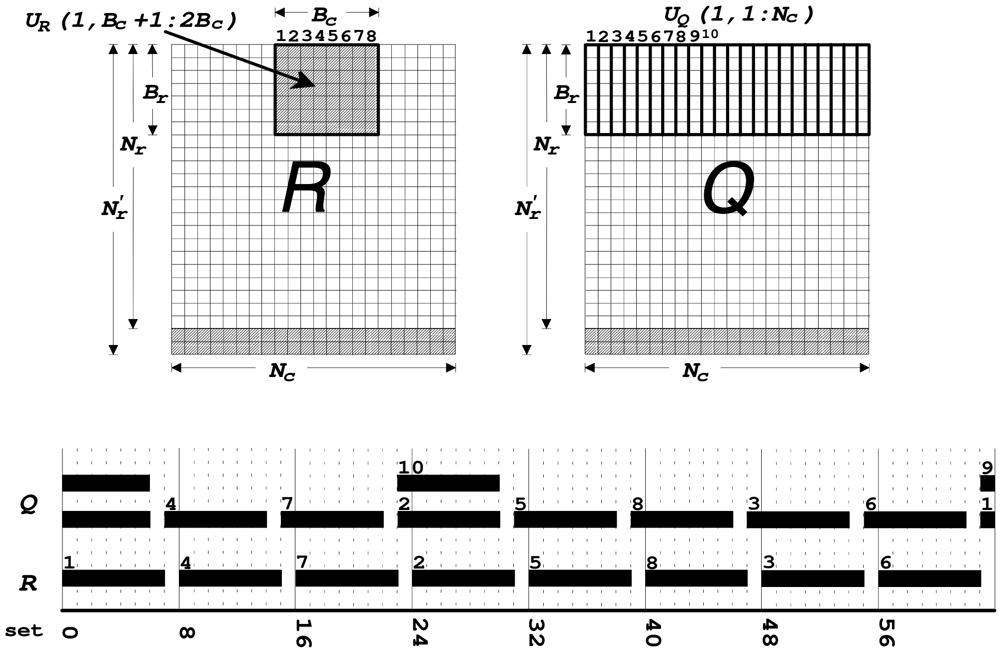


(b) The mapping of the data when $KK = II = 1$.

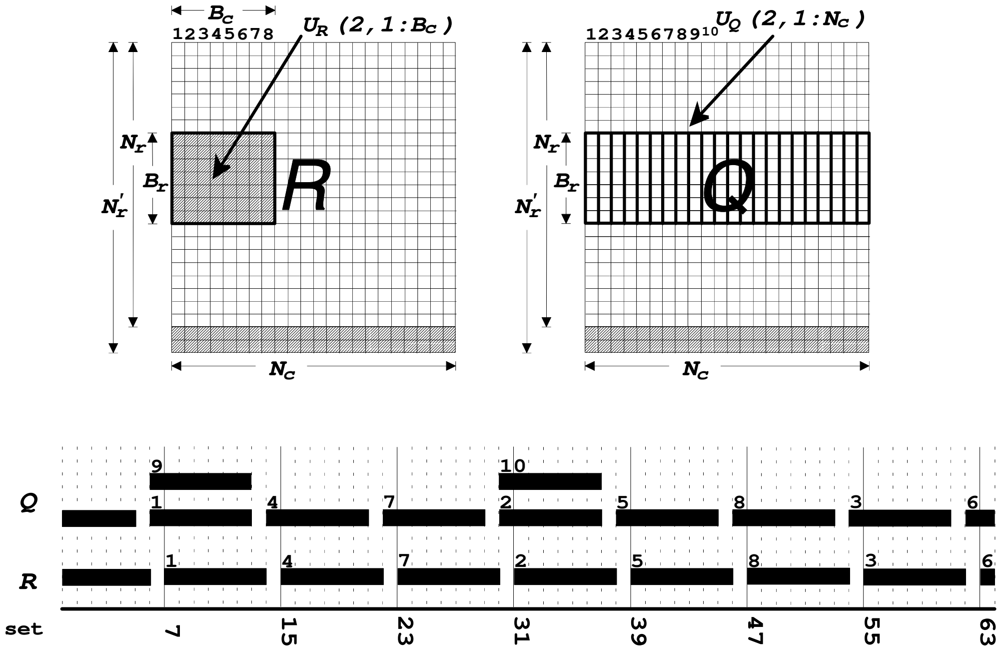


(c) Mapping in two coverage loop iterations (J = 1 and J = 2)

Figure 4.8: Illustration of cache mappings after the application the array padding and initial address adjustment techniques for $KK = II = 1$: (D, S, L) = (1, 64, 1), $B_r = 7$, $B_c = 8$, $N_r = 22$, and $N'_r = 24$.



(a) $KK = B_c + 1$ and $II = 1$



(b) $KK = 1$ and $II = B_r + 1$

Figure 4.9: Illustration of cache mappings after applying the array padding and initial address adjustment techniques: $(D, S, L) = (1,\ 64,\ 1)$, $B_r = 7$, $B_c = 8$, $N_r = 22$, and $N'_r = 24$.

works because the analytical array padding method provides a leading dimension which guarantees that the segment size is at least B_r/L.

On the contrary, if we use the leading dimension that results from the search algorithm, then the size of each segment, d, may be less than B_r/L. This implies that some Q block will begin in the middle of an R block column, and thus have a different alignment. For example, if we let $B_c = 3$ in Example 4.4.6, then the original leading dimension will be the self-conflict-free leading dimension according to the search algorithm. Suppose that array R begins in cache set 0, and the initial addresses of Q and R are adjusted in the same way as before. Then, referring to Figure 4.1c, we find that the fourth block column of Q, $U_Q(1,4)$, which begins in cache set 1, has lost its alignment relative to the R block columns. The associated conflict cache sets are thus not covered by the desirable data layout. Hence, in general, the leading dimension generated by the search algorithm is not appropriate in this mechanism for reducing cross conflict misses, and the one generated by the analytic solution is preferred.

Implement Access Resequencing

We now adjust the access sequence of the loop-K iterations so that the cache sets covered by the desirable data layout can actually result in the desirable track, r_1q_1. Figure 4.10 shows the matrix multiply program with the access resequencing. Since in a coverage loop iteration, the Q block column is referenced with every block column of the R block, the R block column with which the Q block column is aligned must be referenced first. Otherwise, the first elements in the tracks of the cache sets mapped to by the Q block column will be q_1 because the Q block column needs to be accessed with the prior block columns of the R block. For example, consider Figure 4.8c. When $J = 2$, the first elements in the tracks of cache sets 24 to 30 are all q_1. However, if the block columns are referenced in the cyclically permuted order 2, 3, 4, 5, 6, 7, 1, then the tracks of all conflict cache sets become the desirable one, r_1q_1.

The program adds a new variable, l, to keep track of the position of the R block column (in the current block) relative to which a Q block column is aligned. If $l \geq 0$, then it indicates that the Q block column is aligned relative to the $(l+1)^{th}$ block column of the R block. Hence, the first inner loop iteration should reference the $(l+1)^{th}$ block column of the R block, *i.e.* $KK + l$. Loop K is thus split into two loops. The first one references R block columns from $KK + l$ to $KK + B_c - 1$, and the second one, from KK to $KK + l - 1$. If $l < 0$, then it indicates that the Q block column is aligned to no R block column, which may happen when $B_c < S/d$. In this case, it does not matter which block column of R should be referenced first since the cache sets mapped to by the Q block columns should have no conflict now (except for the one mapped to by the first line when $B_r/L = d$). We just let K begin at KK.

For a given KK and II, the block columns of Q and R that are involved must have the same first index, *i.e.* $U_Q(i, 1 : N_c)$, $U_R(i, KK : KK + B_c - 1)$. We know that if $U_Q(i,1)$ is aligned relative to the first block column of the R block when $J = 1$, then $U_Q(i,2)$ will be aligned relative to the second block column of the R block when $J = 2$, and so on, (refer to Figure 4.8). Hence, we can simply increment l by 1 for each following loop-J iteration. However, since there are only B_c block columns in a block, the value of l must be reset when $l = B_c$. If $B_c < S/d$, we reset l to $B_c - S/d$, which indicates that the next $S/d - B_c$ block columns of Q are not aligned to any R block column. After $S/d - B_c$ iterations, the value of l will become 0, and the corresponding Q block column is again aligned relative to the first block column of the R block. If $B_c = S/d$, then we simply reset l to 0, which starts another period. In cases where the first block column of Q, $U_Q(i,1)$, is not aligned relative to the first block column of the R block, say to the $(l_0+1)^{th}$, then we simply have to set l to l_0 before entering into loop J.

For loop J, there are two outer loops, loops II and KK. According to the discussion above, we know that l_0, the position of the R block column in a block relative to which the first Q

```
Double Precision $R(N_r, N_c)$, $P(N_r, N_c)$, $Q(N_r, N_c)$

DO 100 KK = 1, $N_c$, $B_c$
   update $l_0$
   DO 200 II = 1, $N_r$, $B_r$
      $l = l_0$
      do 220 J = 1, $N_c$
         kstrt = KK
         if ($l$ .ge. 0) kstrt = kstrt + $l$
         $l = l + 1$
         if ($l$ .eq. $B_c$) $l = B_c - S/d$
         do 240 K = kstrt, MIN(KK+$B_c$-1,$N_c$)
            DO 300 I = II, MIN(II+$B_r$-1,$N_r$)
               Q(I, J) = Q(I, J) + R(I, K) * P(K, J)
            END DO
         END DO
         do 241 K = KK, MIN(kstrt - 1, $N_c$)
            DO 301 I = II, MIN(II+$B_r$-1,$N_r$)
               Q(I, J) = Q(I, J) + R(I, K) * P(K, J)
            END DO
         END DO
      END DO
   END DO
END DO
```

Figure 4.10: Blocked matrix multiplication code with access resequencing.

block column is aligned, can only be changed when KK is changed, provided that $B_c < S/d$. Basically, when KK is advanced one step, l_0 needs to be updated to $(l_0 + S/d - B_c)$. However, when $(l_0 + S/d - B_c)$ is greater than B_c, some adjustment is actually required. We do not discuss this subtle detail here.

Consider Array *P* in the *P*-*R* Conflicts

Regarding arrays P and R, the data set involved in a coverage loop iteration includes one block column of P and one block of R. After we apply the array padding technique, in a coverage loop iteration a cache set can contain at most a one line footprint each of arrays P, Q, and R. Hence, a P-R conflict cache set will have conflict footprint (1, 1) with respect to (P, R), and thus may experience track r_1p_1, p_1r_1, or $p_1r_1p_1$. Among them, only track $p_1r_1p_1$ is undesirable.

In Section A.1.2 of the Appendix, we show that in an inner loop iteration, a P element is accessed and is multiplied by every element of the R block column that is referenced in that inner loop iteration. This P element is typically accessed before any element of the R block column. If a cache set has track $p_1r_1p_1$, then there must be one element (excluding the last) in line p_1 that is accessed in the same inner loop iteration as the R block column whose mapping includes this same cache set. Since in a coverage loop iteration, only one P element can be accessed in the same inner loop iteration as the R block column, there is at most one cache set that can have the undesirable track, $p_1r_1p_1$, for any R block column whose mapping overlaps with the mapping of the P clock column.

Example 4.4.7 In the blocked matrix multiply program, no P element is accessed more than once in a coverage loop iteration. Hence, if $L = 1$, the conflict cache sets that contain conflict footprint

(1, 1) for (P, R) will have only desirable tracks, p_1r_1 and r_1p_1. Therefore, it is better to discuss P-R conflict by considering $L > 1$. Accordingly, suppose $(D, S, L) = (1, 128, 2)$, $B_r = B_c = 14$, and $N_r' = 48$. Figure 4.11a shows the mapping of the relevant portions of arrays P, Q, and R when $KK = II = J = 1$. Note that in this case, $\gcd(N_r'/L, S)$ is equal to 8, and the first element of block column $U_Q(1, 1)$ is now aligned to cache set 127. The mapping of $U_P(1, 1)$, which is arbitrarily selected for the purpose of illustration, overlaps with the mappings of $U_R(1, 7)$ in cache sets 18 through 22 and $U_R(1, 2)$ in cache set 24.

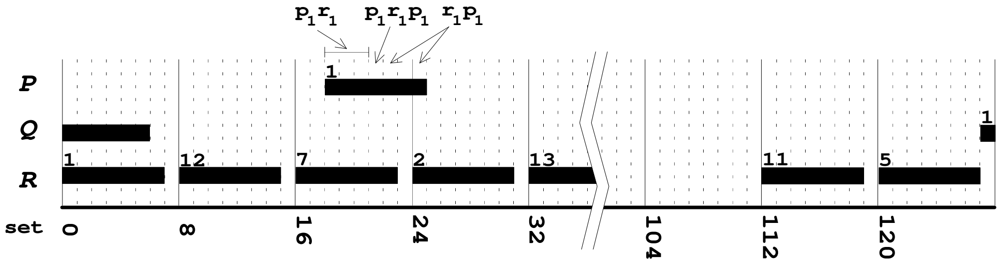
From the innermost loop of the Figure 4.10 program, $Q(I, J) = Q(I, J) + R(I, K) * P(K, J)$, we know that block column $U_R(1, 7)$ is accessed in the same inner loop iteration as the 7th element of block column $U_P(1, 1)$, which is the first element of the P line mapped to cache set 21. Since this P element is accessed before all of the elements of $U_R(1, 7)$, and since the next P element in the same line will be accessed in the next inner loop iteration, cache set 21 has track $p_1r_1p_1$. Moreover, because all of the P elements in the lines mapped to cache sets 18, 19, and 20 are accessed in earlier inner loop iterations than $U_R(1, 7)$ is accessed, and because in the same coverage loop iteration these P elements will not be accessed again, cache set 18, 19, and 20 have track p_1r_1. Similarly, since the P elements in the line mapped to cache set 22 are accessed in later inner loop iterations than $U_R(1, 7)$ is accessed, cache set 22 has track r_1p_1. Consequently, we can see that for those cache sets covered by the mapping overlap of $U_R(1, 7)$ and $U_P(1, 1)$, there is only one cache set, cache set 21, that has the undesirable track, $p_1r_1p_1$.

For the cache set covered by the mapping overlap of $U_P(1, 1)$ and $U_R(1, 2)$, *i.e.* cache set 24, we can derive its track as r_1p_1 with a similar analysis. Thus, there is no cache set with the undesirable track for this overlap. □

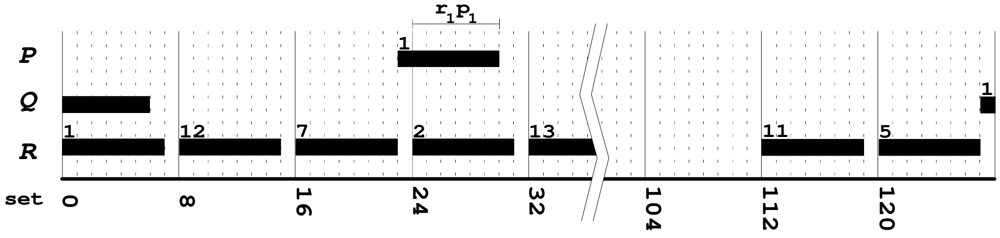
As mentioned before, since arrays Q and P are referenced differently, we do not have a solution that is effective for both Q-R and P-R conflicts. However, since for any R block column whose mapping overlaps with the mapping of the P clock column, there is at most one cache set that can have track $p_1r_1p_1$, *i.e.* most of the cross conflict misses of P already result from desirable track p_1r_1 or r_1p_1, how to place array P so as to reduce the P-R conflicts is not a critical issue in this program after the solution for the Q-R conflicts has been implemented. However, we found that we can still adjust the data layout of P so that if the block shape is square ($B_r = B_c$), then the number of cross conflict misses of P is the least when $KK = II$.

This was accomplished by changing the leading dimension of P to be the same as for arrays R and Q, and adjusting array P so that block column $U_P(1, 1)$ begins in the same cache set as $U_Q(1, 2)$. Figure 4.11b shows the mapping, where all of the P-R conflict cache sets (24 through 29) have desirable tracks, r_1p_1. When $J = 2$, the P and Q block columns involved are $U_P(1, 2)$ and $U_Q(1, 2)$, respectively, and $U_P(1, 2)$ will begin in the same cache set as $U_Q(1, 3)$. Since the access resequencing technique will enforce $U_R(1, 2)$ to be accessed first, the P-R conflict cache sets will also have track r_1p_1. Similarly, the same track will be repeated in each coverage loop iteration.

When KK and II are both advanced one step, the P and R block columns involved are $U_P(2, 1 : N_c)$ and $U_Q(2, 1 : N_c)$. For any $J = j$, the mapping of $U_P(2, j)$ is shifted by B_r/L cache sets relative to that of $U_P(1, j)$. However, the mapping of $U_Q(2, j)$ is also shifted by B_r/L cache sets relative to that of $U_Q(1, j)$. Thus, the P-R conflict cache sets should all experience track r_1p_1 for all cases where $KK = II$. We do not have control over the cases where $KK \neq II$. However, as mentioned before, the number of cross conflict misses of P should not be severe (one undesirable track per mapping overlap between a pair of P, R block columns) after the solution above is implemented.



(a) Block column $U_P(1,1)$ is placed arbitrarily.



(b) Block column $U_P(1,1)$ begins in the same cache set as $U_Q(1,2)$.

Figure 4.11: Illustration of cache mappings when $KK = II = J = 1$: $(D, S, L) = (1,\ 128,\ 2)$, $B_c = 14$, $B_r = 14$, $N'_r = 48$.

4.4.4 Simulation Results

This section shows simulation results of the matrix multiply program after the application of the methods discussed in Sections 4.4.1, 4.4.2, and 4.4.3. The results were collected by using the same tools (IBM Atrace and Dinero III) and procedures as those used in Chapter 3. Three matrix sizes are chosen, 100 × 100, 200 × 200, and 300 × 300. For each matrix size, we run simulations for block sizes 4 × 4, 8 × 8, 12 × 12, 16 × 16, 20 × 20, 24 × 24, 28 × 28, and 32 × 32. All simulations were run for the cache of organization $(D, S, L) = (1,\ 256,\ 4)$.

Figures 4.12, 4.13, and 4.14 show the results for matrices 100 × 100, 200 × 200, and 300 × 300, respectively. Each figure displays six curves. Curve I is the simulation result for the original code. Curves II, III, and IV are the simulation results after the application of the methods in Section 4.4.1, 4.4.2, and 4.4.3, respectively. Curve V shows the number of loop cold start misses. Since the simulator does not report this data, we use the output from the model derived in Chapter 3. Finally, curve VI represents the number of remaining cross conflict misses which is the difference between IV and V.

Generally, we can see a step by step progress when applying the methods in Sections 4.4.1, 4.4.2, and 4.4.3. The original code (I) results in self conflict misses after block sizes 24 × 24, 28 × 28, and 24 × 24 for matrices 100 × 100, 200 × 200, and 300 × 300, respectively. In all cases, the array padding (II) can effectively remove these self conflict misses. Adding the initial address adjustment technique (III) can also eliminate the ping-pong conflict misses. However, because the new alignment may actually increase the number of cross conflict misses, this sequence of techniques does not guarantee a consistent improvement at each step. Nevertheless, in this example

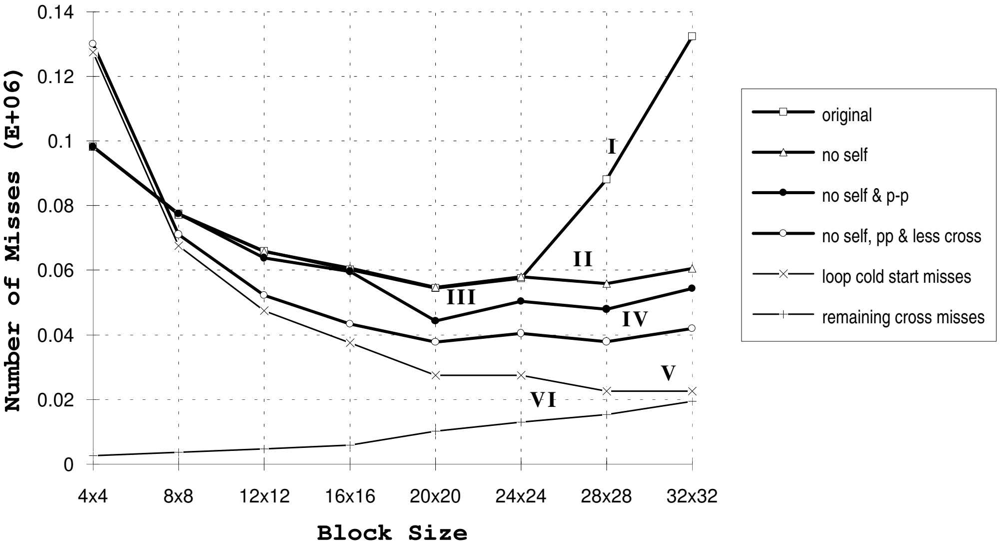


Figure 4.12: Reduction of cache misses: $(D, S, L) = (1, 256, 4)$, $C = 8\text{K}; N = 100$.

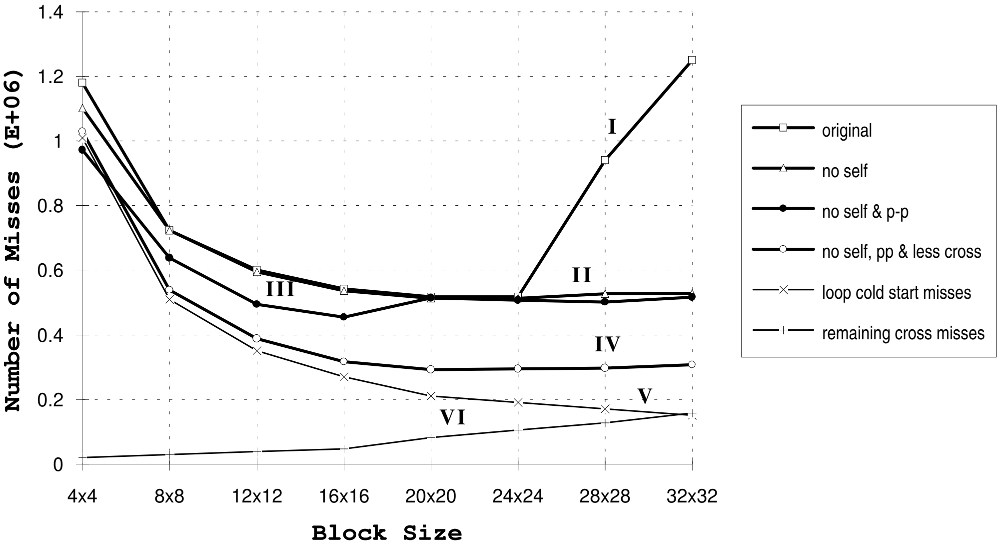


Figure 4.13: Reduction of cache misses: $(D, S, L) = (1, 256, 4)$, $C = 8\text{K}; N = 200$.

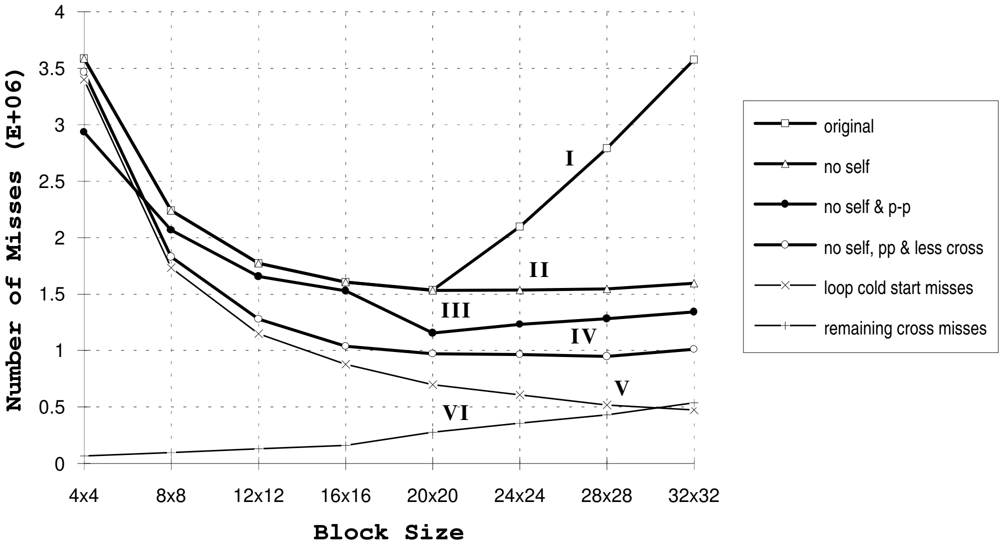


Figure 4.14: Reduction of cache misses: (D, S, L) = (1, 256, 4), C = 8K;N = 300.

the total number of misses is significantly improved at each step.

The method in Section 4.4.3 includes all three techniques (array padding, initial address adjustment, and access resequencing), and can significantly reduce the total cache misses when block size is greater than 4 × 4. For example, at block size 32 × 32, the resulting numbers of cache misses (IV) are about 31.6%, 24.6%, and 28.2% of the original in Figures 4.12, 4.13, and 4.14, respectively. With this implementation, all new data referenced in a coverage loop iteration will always experience cache misses, while with other implementations these new data are not necessary all cache misses. As a result, the total number of cache misses of curve IV at block size 4 × 4, for example, is clearly not optimal in some cases and may even be worse than the original curve. However, 4 × 4 is not an interesting block size and the conflict miss model and the miss reduction procedure are not intended to deal with such effects of undesirably small block sizes.

Another phenomenon worth mentioning is that the curve (IV) decreases quickly as the block size increases from 4 × 4 to 16 × 16, and then becomes fairly flat. This indicates that the number of cache misses is not a critical factor for determining the optimal block size for the program performance after block size 16 × 16, as long as the total capacity of the cache does not become a factor. Loop overhead, which is another noticeable factor in program performance, should play a decisive role after the cache performance is improved by this technology. To reduce loop overhead, the program favors large block size since a large block traverses the outer loops fewer times. As a result, we can simply select the largest block size as the optimal block size.

The cache misses in IV should only consist of loop cold start and cross conflict misses. Hence, the difference between IV and V should be the number of remaining cross conflict misses (VI). Curve VI can be roughly divided into two fairly linear segments: from block size 4 × 4 to 16 × 16, and from 20 × 20 to 32 × 32. The first one increases relatively slowly in comparison with the second one. In fact, for the block sizes covered by the first linear segment, cross conflict misses due to Q-R conflict are completely removed. Note that at block size 32 × 32, the remaining number of cross conflict misses is almost equal to the number of loop cold start misses. This indicates that each distinct line that is referenced will result in approximately two cache misses. In the cases of block size 32 × 32, one R block (except for those at along the bottom or right side of the array) is the same size as the cache itself. For such an R block, after it has been referenced in the $J = 1$ loop for a given KK and II, each line of Q and P referenced in the following loop-J iterations will replace one R line. Since this R line needs to be referenced again, this replacement also causes a later miss to the R line, which results in a total of 2 cache misses for each line of arrays Q and R. However, the final results do not show 2 cache misses per loop cold start miss which may be due to the small R blocks along the bottom and right side of the array, and possibly also to some other minor unidentified effects.

4.5 Conclusion

In this chapter, we have presented the methods for eliminating or reducing a particular type of cache miss based on three techniques that either modify or exploit the data layout: array padding, initial address adjustment, and access resequencing. As a case study, we applied these techniques to a blocked matrix multiply program.

To eliminate self conflict misses, we introduced two methods for applying the array padding technique: a search algorithm and an analytical method. The former uses a necessary and sufficient condition, while the latter uses a conservative condition. Although the analytical method may not generate the minimum padding, it is relatively fast and is also suitable for being used in the method for reducing cross conflict misses. To remove ping-pong conflict misses, we proposed a process that involves the array padding and initial address adjustment techniques. This process ensures that any two columns, one from array Q and the other from array R, begin in a direct mapped cache at least L elements apart, and thus guarantees that no ping-pong conflict misses will occur. Finally, to reduce cross conflict misses, we introduced a method that includes all three of the aforementioned techniques. The central idea here is to maximize the number of cache sets with no conflict and then, for the remaining sets that still have conflicts, maximize the cache sets that have the desirable track.

The simulation results show that these methods can effectively reduce the numbers of self, cross, and ping-pong conflict misses. From the results, the method that deals with the cross conflict misses in Section 4.4.3 is the most effective; however, it is not without disadvantages. First, since the technique takes into account how arrays are typically referenced, arrays that are referenced differently may not benefit from the solution. Hence, additional effort may be required to arrange other arrays properly. Second, since this technique may split a loop into two parts and may add some bookkeeping instructions, it will add some run time overhead. Consequently, it is better to apply this technique when it is known that the cross conflict misses are a dominant performance degradation factor and apply it first to the array pair that has the most cross conflict misses. This information can be obtained from the output of the model discussed in Chapter 3, as shown in Figure 3.10.

These methods and the loop-based cache model introduced in Chapter 3 can form a goal-directed cache performance tuning system: the model can report the number of cache misses for self, cross, and ping-pong conflict misses for each array (as in Figure 3.10). Each reported type of conflict miss can then be removed or reduced by one of the techniques in this chapter. This systematic procedure can replace the expensive trial and error processes that prevail today and should result in significantly better solutions without undue human or compiler effort.

CHAPTER 5

Communication Reduction Technique - Array Grouping

This chapter presents a technique to reduce communication caused by nonconsecutive references in a cache-based shared-memory multiprocessor system, and theorems that specify when and where to apply this technique. Section 5.1 describes the communication problem. Section 5.2 discusses some related work. Section 5.3 describes the program, programming environment, and machine models that underlay the method. Section 5.4 introduces the array reorganization technique and associated procedures including reference pattern analysis, array selection and reorganization, program modification, and loop transformation. Section 5.5 investigates why and when the technique works. Section 5.6 presents and discusses experimental results. Conclusions are presented in section 5.7.

5.1 Introduction

Since consecutive reference patterns are relatively easy to deal with, this chapter focuses on nonconsecutive reference patterns. Nonconsecutive references in parallel programs often significantly degrade program performance because of their poor locality. Nonconsecutive references to a data structure may be caused by incrementing an addressing index by a nonunit constant, by computing the next addressing index using some function of loop index variables, or by determining the next index indirectly through elements of another array. Such reference patterns often exist in large scientific programs. In a cache-based shared-memory multiprocessor system, these nonconsecutive references will cause inefficient communication due to superfluous data in communication transactions, and will result in inefficient cache and subcache utilization due to superfluity, false sharing, and pollution. These effects increase average memory access time, and thus degrade program performance. Moreover, inefficient communication and false sharing result in excessive communication transactions on the network, which will affect the scalability of an application.

Consider the following FORTRAN program segment and a cache-based shared-memory system that contains the first level of the memory hierarchy, the L1 cache, in each node:

```
DOUBLE PRECISION A(m, n), B(m, n)

DO i = 1, n, h
   DO j = 1, m, k
      ··· = A(j,i) + B(j,i)
   END DO
END DO
```

Assume that a cache block is the unit of communication and contains L (> 1) array elements, and that the loop step sizes, h and k, are greater than L. If neither array A nor array B has been referenced before, every array reference in the loop will cause a cache miss, which in turn results in a cache block request message on the network. The node responding to the request will return a transaction containing L elements even though only one is needed. Similarly, once installed, the corresponding cache block will also contain only one useful and $L - 1$ superfluous data elements. Thus, the program performance is expected to suffer due to a poor miss ratio and an excessive number of communication transactions. Furthermore, the hit ratio of subsequent portions of the program is reduced due to pollution of the cache with superfluous and singly referenced data items.

For the sake of simplicity, assume further that all arrays are aligned to cache block boundaries, and that L is a power of 2. Then, an effective way to improve this program segment's performance is to construct an array that interleaves elements of arrays A and B so that in the new array, each element $A(j, i)$ is followed immediately by $B(j, i)$. This new array has better spatial locality: the cache block associated with each communication transaction will contain two useful data elements, $A(j, i)$ and $B(j, i)$. This will cut the number of cache misses and communication transactions in half, and hence will improve program performance. Furthermore, the data for this program segment may occupy fewer blocks in the cache, thereby replacing fewer blocks of the surrounding context.

This simple example demonstrates the reorganization of arrays and data layout to improve program locality and performance. Although the example contains only constant strides, our technique is also applicable to nonconsecutive references caused by irregular (computed) strides and indirect array references. In addition, the complete data reorganization process also includes loop transformation techniques such as loop interchange [23, 24, 25] and loop permutation [55].

5.2 Related Work

Our research aims to improve program performance by optimizing data layout and sequencing memory access in cache-based shared-memory multiprocessor systems.

Many studies have considered data layout techniques to improve memory performance and reduce communication for other types of architectures such as vector supercomputers [56, 57], Single Instruction stream- Multiple Data stream (SIMD) systems [58, 59], and message-passing non-shared-memory systems [60, 61, 62].

For cache-based shared-memory multiprocessor systems, cache performance improvement and communication overhead reduction are often achieved by considering loop transformation and partitioning techniques [50, 63, 64, 65, 66, 67]. Instead of just considering the loop restructuring techniques, Ju [68] studies the problem by involving both data layout and loop transformation in the solution. In his algorithms, for each loop and the arrays in the loop it considers all possible combinations of legal loop transformations and data layout (row-major, column-major, and diagonal), and computes the cache coherence overhead for every combination. Then the algorithm searches for a combination per loop for the whole program with minimum total cache coherence overhead. In contrast to these studies, this chapter focuses on indirect references which prior studies have generally ignored. As in Ju's research, our proposed technique also takes both data layout and loop transformation techniques into consideration.

Interleaving elements (or groups of elements) from several arrays has been used in relational data base applications [69, 70] where records that are typically accessed together are stored contiguously in memory. This idea was also suggested to reduce page faults in paged memory systems [71]. We use this technique to deal with a different class of problems involving communication efficiency

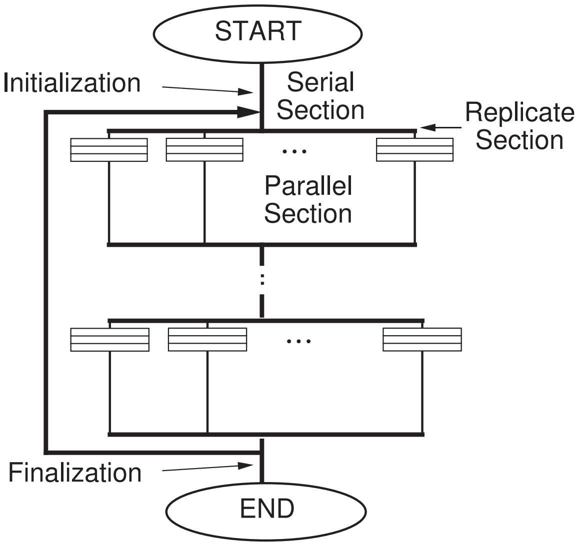


Figure 5.1: Program model.

and cache performance on cache-based shared-memory parallel systems, and present a systematic method of analysis for this class of problems.

5.3 Program and Machine Models

In general, a parallel program consists of a sequence of three types of sections: *serial*, *parallel*, and *replicate*. The work of a serial section is executed by only one processor, that of a parallel section is partitioned and executed concurrently by multiple processors, and that of a replicate section is replicated and executed by every processor [25]. We focus initially on a simple, but common parallel program structure (Figure 5.1). This structure consists of a series of parallel sections, each enclosed by a parallel construct that enforces barriers (replicate sections) at the beginning and end of the section. These parallel sections may be separated by serial sections. The entire sequence is enclosed in an iteration loop where an iteration corresponds, for example, to a *simulation time step*. The entire code may begin with serial initialization code and end with serial finalization code. In programs that are suitable for such machines, extensive work is executed in parallelizable, normalized[1] loops which form the major part of the parallel sections. The following discussion is limited to the important and common case where parallelization is realized by partitioning the loop index space and the portion of the iteration space assigned to a processor is invariant among different simulation time steps.

One processor is initially assigned to be the master processor responsible for executing all serial sections. A team of processors (including the master processor) is designated by the system to execute each enclosed parallel section concurrently. The number of processors in a team exactly equals the number of components in the corresponding parallel section. Processors in a team must be synchronized when they encounter a barrier. No team member can start its component until the master processor finishes the prior serial work. Similarly, the master processor cannot begin executing the following section until every team member has finished its parallel component.

Although the concepts introduced in this chapter are applicable to other cache-based shared-

---

[1] A loop is normalized if its initial loop index and step size are 1 [25].

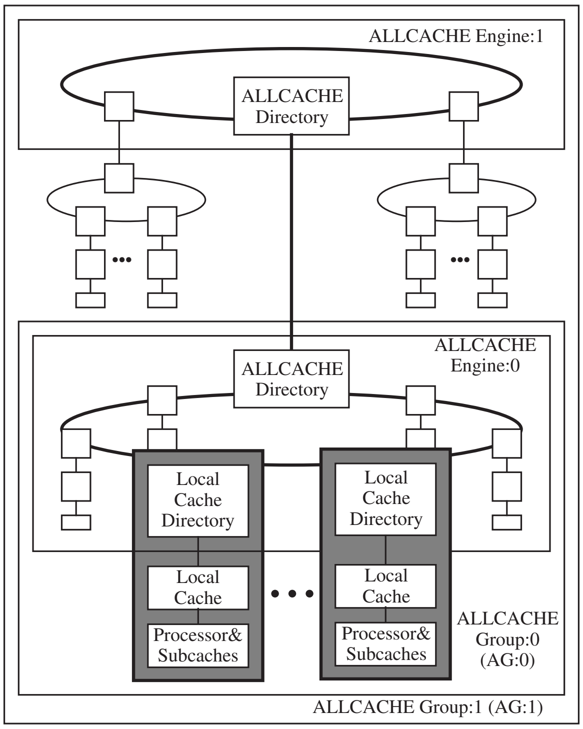


Figure 5.2: The KSR1 architecture.

memory multiprocessor systems, we concentrate on Cache-Only Memory Architecture (COMA) systems, such as the KSR1 [72], DDM [73], and COMA-FLAT [74]. In a COMA system, data items do not have static home addresses, they are instead dynamically moved and replicated among the caches in various nodes in response to an application's reference patterns. The KSR1 was chosen as our experimental testbed because it was the only commercially available COMA system.

The KSR1 (Figure 5.2) [72, 75, 76] is built as a group of ALLCACHE engines, connected by a fat tree hierarchy of unidirectional pipelined slotted rings. Each lowest level ring includes 32 processor cells and two ALLCACHE directories. Each cell contains a processor, data and instruction subcaches, a large local cache and a network interface. In the KSR1's cache design, the *allocation unit* associated with an address tag in the directory and reserved space in the cache is different from the *transfer unit* which is the basic unit transferred to a cache on demand. The allocation units of the subcache and local cache are called *block* and *page*, respectively. Each block or page is further divided into several *subblocks* or *subpages*, which are the transfer units.

Up to 34 such rings can be connected by a single second-level ring for a maximum configuration of 1088 processors. The ring transaction unit is a single subpage (128 bytes) plus some leading header information (16 bytes). Data coherence is maintained within each lowest level ring by a snooping cache coherence protocol. The coherence protocol is extended to two-level ring systems by adding a pair of ALLCACHE directories between each lowest level ring and the second-level ring.

To serve a local cache miss, a request is sent to the ring. For a read request, the first cell whose directory determines that it can respond will place a response including the requested subpage on the ring. The response then completes the tour and returns to the requester. For a write request (to obtain exclusive ownership), each node observing the request must invalidate its copy of the subpage, if present. A valid copy of the subpage is also supplied as above if the requester does not already have one. During the request phase, if the ALLCACHE directory is encountered, a lookup is made to determine if the request can be satisfied within this lower level ring. If not, the request is transferred to the second-level ring, and then proceeds around part of the ALLCACHE Engine:1 and part of a remote ALLCACHE Engine:0 in a similar fashion. The response then completes the tours of the three rings in the reverse order. Invalidates traverse as many rings as necessary in a similar fashion.

The latency of communication is usually much higher than the latency of a cache miss served within the node. In the KSR1, communication latency varies between 135 and 600 cycles, while subcache misses served by the requesting node's own local cache take an average of 23.4 cycles (when the corresponding block is already allocated) or 49.2 cycles (otherwise) [76]. Although nonconsecutive references affect both communication and cache performance, only communication will be considered when deciding which arrays to reorganize. However, array reorganization should also improve subcache utilization.

Because of the large (32 MB) local cache of the KSR1, we assume that data accessed by a processor typically remains in the corresponding local cache between time steps unless invalidated to enforce coherence. This assumption implies that communication is caused only by coherence misses, *i.e.* no communication is caused by any other types of cache misses, including capacity, conflict, and compulsory misses [6] (or capacity, matching, replacement, and compulsory misses [5]). This model thus focuses on the dominant class of communication in a COMA system when executing an interesting, *i.e.* large and not embarrassingly parallel, application on a reasonably large number of processors.

5.4 Array Grouping

In this section we introduce a systematic method of grouping arrays to reduce the performance degradation caused by nonconsecutive references. This method includes the following major steps: (1) record for each program section the patterns of references to the shared arrays, (2) analyze the patterns and form equivalence classes of arrays with interesting patterns, (3) reorganize each equivalence class into new data structures, (4) modify the program as needed to reflect this reorganization, (5) apply loop transformations to the loops that access the reorganized arrays.

Suppose that r arrays are to be grouped into a single array using a subset of d of their dimensions, called the *group dimensions*, whose declared sizes are the same for these arrays. Transpose each array so that the group dimensions appear contiguously in the same order in the rightmost dimensions. Let *group index vector* $\vec{i} = (i_1, i_2, \cdots, i_d)$ represent an instance of the group dimensions. Each $\vec{i}$ identifies a group of elements of each array whose indices in the group dimensions are i_1, i_2, $\cdots$, i_d, respectively. Let $E_j(\vec{i})$ denote the element group of array j for group index vector $\vec{i}$. We define *element group* $\vec{i}$ of the grouped array, $E(\vec{i})$, to be the concatenation of $E_1(\vec{i})$, $E_2(\vec{i})$, $\cdots$, $E_r(\vec{i})$, where r is the number of arrays to be grouped. The size of $E_j(\vec{i})$, e_j, is the product of the sizes of all the dimensions of array j, except the group dimensions. The size of $E(\vec{i})$, e, is then equal to $\sum_j e_j$. Since the group dimensions are rightmost, the elements within an element group occupy consecutive memory locations.

The grouped array can then be constructed by simply ordering the element groups in the memory according to FORTRAN semantics, *i.e.*, $E(\vec{i})$ is placed before $E(\vec{j})$ if $i_k < j_k$, where k is the rightmost position where $i_k \neq j_k$. Figure 5.3 shows three types of element group layouts containing 1, $s/(e+c)$, and s/e element groups per subpage, respectively, where s is the subpage size (in elements) and c is the number of blanks inserted between element groups. In the second layout, c is the minimum integer such that $(e+c)$ divides s. In the third layout, s/e may be noninteger. The example in Figure 5.3 has $s = 16$, $e = 6$. The first case (a) has 1 element group

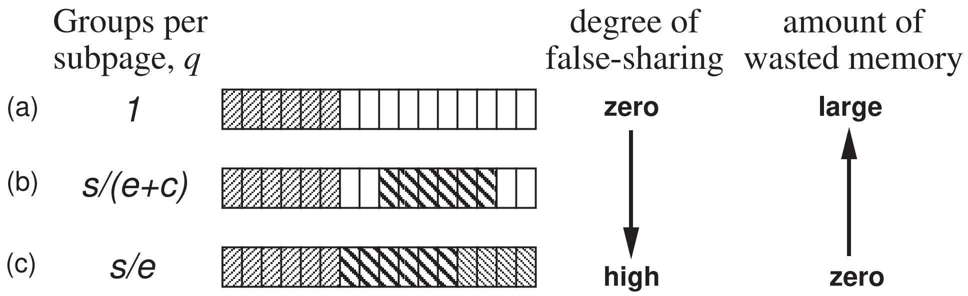


Figure 5.3: Three types of grouped array layouts in a subpage.

and 10 blanks per subpage. The second (b) sets $c = 2$ resulting in 2 element groups, the largest possible number of complete groups, and 4 blanks per subpage. The third (c) uses no blanks and has $\frac{16}{6} = 2\,\frac{2}{3}$ element groups per subpage. Note that the next subpage will contain all or part of 4 different element groups.

Our method of grouping guarantees that whenever one element of an element group is referenced by a processor, all elements of that group are referenced by that processor. False sharing (which occurs when two or more processors share a subpage, but do not reference the same element groups in the subpage) can occur whenever more than one element group occupies a subpage. Since false sharing increases coherence misses and since inserted blanks waste space and increase misses of other types, case (a) may achieve the fewest misses for sufficiently large caches (or small arrays) or for array groups where false sharing dominates, (c) may be best in the opposite case, and (b) is a compromise between the two. Each final array group may use its own best-case layout. We do not consider false sharing in the process of selecting which arrays to group. However, the degree of false sharing should generally decrease after grouping since a subpage of the grouped array will never contain more element groups than a subpage of any of the given arrays when allocated separately.

We suggest only these three types of arrangements because they yield straightforward indexing into the grouped array. Regardless of the chosen layout, the final grouped array can be declared as a $d+1$ dimension array; the first dimension has size $s/q = e + c$, where q is the number of element groups in a subpage and the other dimensions are the group dimensions. Array accessing is done as before for the group dimensions with some modified indexing to access the desired element in an element group.

5.4.1 The Program Section Array Table (PSAT)

The *Program Section Array Table (PSAT)* is used to record and analyze array reference patterns. Cell (i, j) of the PSAT describes how array j is referenced in program section i. Typically, however, the initialization and finalization sections (Figure 5.1) are ignored. In our study, we classify array reference patterns as C (consecutive, *i.e.* unit stride, on all dimensions), $I^d(X)$ (indirect on dimension d through array X), and $F^d(f)$ (nonunit stride on dimension d using a function, f, of loop index variables). Each reference pattern is assigned an R or W prefix to indicate read or write access, respectively. If an array has more than one reference pattern in a particular program section, all patterns are listed in order in the corresponding cell.

	Equiv. Class 1	Equiv. Class 2	Equiv. Class 3
P_1	$WI^3(X)$	$RI^3(X)$, $WI^3(X)$	WC
S_1	RC	$RI^3(X)$	RC
P_2	$WI^3(X)$	$RI^3(X)$, $WI^3(X)$	$RI^3(X)$, $WI^3(X)$
S_2	RC	$RI^3(X)$	$RI^3(X)$

Figure 5.4: PSAT of arrays in the three equivalence classes and 4 program sections in which S_1 and S_2 are serial, and P_1 and P_2 are parallel program sections. All arrays are assumed to have the same declared sizes of their nonconsecutively referenced dimensions.

5.4.2 Choosing candidate array groups for reorganization

Choosing arrays for grouping is based on communication as determined by analyzing the array reference patterns in the PSAT. If an array element is accessed by processor P in one program section and by Q in another section, we assume that P and Q will never be the same processor. This worst case assumption yields the most coherence misses. For indirect array references, it is hard to guarantee when P and Q may be the same processor; thus the worst case assumption appears to be the most reasonable assumption.

Communication is reduced by increasing the number of useful data in a transaction. Arrays that are read-only (RC, RI^d, RF^d) in all program sections are removed from further consideration because as many copies as needed may remain valid, resulting in no coherence communication. Arrays that are referenced only consecutively (RC, WC) are also not considered further since the only preventable coherence communication for these arrays arises due to false sharing of at most 2 subpages per array section. When the boundaries of an array section allocated to a processor are not aligned with subpage boundaries, the first and the last subpages of the array section may contain elements that are not used by this processor, which are the only communication transactions that can be saved by the array grouping technique since all other subpages already contain only useful data. These cases, read-only and reference-consecutively-only, cannot benefit much from the array grouping technique. We partition the remaining arrays into equivalence classes. Each equivalence class is called a *candidate group* (of arrays) for grouping and contains arrays that share the same access patterns.

Two arrays are in the same equivalence class if they have the same declared sizes of their nonconsecutively referenced dimensions and, for each program section, their PSAT entries are symbolically identical. Then two equivalence classes are merged if they have the same declared sizes of their nonconsecutively referenced dimensions and, for each program section, they have the same communication patterns. If only a few program sections are known to dominate program performance, considering the communication patterns in only these program sections may result in sufficiently good performance improvement.

To compare communication patterns, consider the example in Figure 5.4. Recall that we assume that data once initially loaded will remain in a processor's local cache until invalidated, and that communication occurs only when coherence misses occur. When S_1 is executed, array elements in equivalence class 2 are read through the elements of array X. Since these elements were invalidated in P_1, reading these elements will result in communication transactions. Furthermore, although S_1 references every array element in equivalence class 1, only those accessed through the elements of array X and written in P_1 have been invalidated. Therefore, the communication transactions in both classes are determined solely by the elements of array X, and the arrays of these classes thus have the same communication patterns in S_1.

When executing P_2, each processor reads and writes array elements in equivalence class 2 through the elements of array X. Since these elements were supposed to be invalidated in P_1, the reads will result in communication transactions for loading valid copies of these elements in the processor's local cache. When the processor's local cache has the valid copies of these elements, the writes will only initiate invalidation messages to the rings; no subpage transfers should occur. In the same program section, each processor also writes array elements in equivalence class 1 through the elements of X. When the processor tries to obtain the exclusive ownership of these elements, transactions that contain the valid copies of these data elements will occur since these elements were also invalidated in P_1. Note that the communication transactions in both classes are determined solely by the elements of array X. Thus, the arrays in these two classes also have the same communication patterns in P_2. Recall that a PSAT only records program sections that are executed repeatedly. When S_2 and P_1 are executed, it is easy to see that the communication patterns of arrays in equivalence classes 1 and 2 in these two sections are the same as those in S_1 and P_2, respectively. Because equivalence classes 1 and 2 have the same communication patterns in all four sections, they can be merged.

On the contrary, when executing S_1, the processor will find that all of the array elements in equivalence class 3 need to be reloaded. Thus, the arrays in equivalence class 3 have a different communication pattern from those in the other two equivalence classes. If equivalence class 3 were combined with equivalence class 1 (or 2), the reloading of array elements in equivalence class 3 would result in loading all array elements in equivalence class 1 (or 2) as well, which is obviously not necessary. Therefore, this combination is undesirable and the arrays in equivalence class 3 are not grouped with any arrays in class 1 or 2.

5.4.3 Choosing final array groups for reorganization

Although all the arrays in a candidate group can be grouped together, it may be preferable to split a candidate group due to the limited size of the transaction unit. Furthermore, although using all nonconsecutively referenced dimensions as the group dimensions will maximize the number of consecutive, useful data elements in an element group, adding other (consecutively referenced) dimensions to the group dimensions may be preferable. Splitting candidate groups and adding group dimensions are performed for each candidate group by a two phase process: phase 1 defines all possible choices and phase 2 selects the final array groups and their group dimensions.

Phase 1 algorithm: (1) Partition the given candidate group of arrays into disjoint subsets in all possible ways. (2) For each subset of arrays in each partition determine all possible additional group dimensions and for each subset of these additional group dimensions compute the corresponding element group size. (3) For each partition, list for each subset of arrays in the partition the set of all possible additional group dimensions.

Phase 1 example: Suppose $\{A(3,3,4,1000), B(3,7,3,1000), C(4,1000)\}$ is the given candidate group and the dimension with declared size 1000 is the group dimension. There are five possible partitions: $\{\{A,\ B,\ C\}\}$, $\{\{A,\ B\}, \{C\}\}$, $\{\{A,\ C\}, \{B\}\}$, $\{\{B,\ C\}, \{A\}\}$, and $\{\{A\}, \{B\}, \{C\}\}$. The partition $\{\{A\}, \{B\}, \{C\}\}$ is not considered further because it implies no grouping. For each subset of the other four partitions, we need to determine all possible additional (consecutively referenced) dimensions that can also be group dimensions and the corresponding element group sizes. There are 7 distinct subsets in the partitions: $\{A,\ B,\ C\}$, $\{A,\ B\}$, $\{A,\ C\}$, $\{B,\ C\}$, $\{A\}$, $\{B\}$, and $\{C\}$. Since $\{A, B, C\}$ and $\{B, C\}$ each have no common declared sizes among

Array Partition	Additional Group Dimensions	Group Size
$\{A, B\}, \{C\}$	{3, 3}, {4}	11, 1
$\{A, B\}, \{C\}$	{3}, {4}	33, 1
$\{A, B\}, \{C\}$	{}, {4}	99, 1
$\{A, C\}, \{B\}$	{4}, {3, 7, 3}	10, 1
$\{A, C\}, \{B\}$	{}, {3, 7, 3}	40, 1
$\{B, C\}, \{A\}$	{}, {3, 3, 4}	67, 1
$\{A, B, C\}$	{}	103

Table 5.1: Possible partitions and additional group dimensions.

their consecutively referenced dimensions, no group dimensions can be added. The corresponding element group sizes are $3 \times 3 \times 4 + 3 \times 7 \times 3 + 4 = 103$ and $3 \times 7 \times 3 + 4 = 67$, respectively. The single array subsets $\{A\}$, $\{B\}$, and $\{C\}$ are not grouped, all dimensions are chosen as group dimensions, and each array has element group size 1. Subset $\{A, B\}$ may add 0, 1,or 2 group dimensions of size 3. The corresponding element group sizes are 99, 33, and 11, respectively. For subset $\{A, C\}$, no group dimensions or one of size 4 could be added, and the element group sizes are 40 and 10, respectively. For each partition, we can then list for each subset of arrays in the partition the set of all possible additional group dimensions. Table 5.1 presents such lists as well as the corresponding element group sizes for all possible partitions in this example.

In phase 2, one combination, *i.e.* a partition and a subset of each maximum set of group dimensions, must be chosen based on the programmers' knowledge of the program or on predetermined criteria that can be applied (automatically) to a broad range of programs. In our experiments, we choose a combination with the maximum average element group size (not counting ungrouped single arrays) subject to the constraint that no element group exceeds the subpage size. This heuristic attempts to make element groups as large as possible, but insures, for the type (a) and type (b) grouped array layouts in Figure 5.3, that acquiring an element group requires only one communication transaction. However, if a type (c) layout is used, then some element groups may not be accessed in one communication transaction since a type (c) layout may split one element group across two subpages. This effect must be considered when choosing the type of grouped array layout to be used.

5.4.4 Program Modification and Loop Transformations

A grouped array is declared as "$(s/q, \cdots$ group dimensions $\cdots)$", and we use a FORTRAN "EQUIVALENCE" statement to specify the relative positions of the original arrays in the grouped array. Thus only the order of array indices needs to be adjusted to conform to the new order of array dimensions. For example, suppose arrays $A(3,4,1000)$ and $B(3,4,1000)$ are to be grouped together with group dimensions = $\{3, 1000\}$. The corresponding element group size is thus 8. In the modified program, A and B are declared as $A(8,3,1000)$ and $B(8,3,1000)$ with EQUIVALENCE ((A(5, 3, 1000), B(1, 3, 1000)). Finally, wherever A or B appear in the the inner loop body, the leftmost two index expressions are interchanged.

This grouped data structure layout should improve the efficiency of communication. Loop transformations such as loop interchange [23, 24], loop permutation [55], loop distribution [77], or loop fusion [9, 78] as well as additional array dimension reordering may improve the locality of reference, further improving overall performance. Although finding the best transformations may take exponential time [68], the time may be reduced by prior array grouping since arrays with similar memory access patterns are gathered together.

Read-only and consecutively-referenced arrays were not considered since grouping them does not reduce coherence communication. However, grouping such arrays may improve cache locality by grouping elements in conflicting cache blocks into the same block. Our grouping technique is easily adapted to treat this case as well.

5.5 Effect of Grouping on the Number of Transactions

Unfortunately, not all applications of the grouping technique reduce communication, especially when the grouped array contains some blanks between element groups. This issue is studied here by deriving some analytic expressions for the number of transactions before and after grouping. Conditions that determine the benefits of array grouping *a priori* will then be described by a series of theorems. These results provide a theoretical basis for evaluating grouping and for deciding when to apply it.

We use the following assumptions. The first two follow from the previous sections; the rest are mainly for the sake of simplicity of illustration:

1. Communication is caused only by coherence misses.
2. The size of an element group (e) is less than the size of the subpage (s).
3. False sharing is not explicitly considered in this section.
4. Arrays are aligned on subpage boundaries.
5. The rightmost dimension is the only group dimension. This assumption is not actually a restriction because multiple group dimensions can always be linearized to a single dimension.
6. The number of element groups in a subpage, q, is an integer and divides s. This assumption guarantees that no element group will span two subpages.

From assumptions 1 and 3, and because true sharing within the same program section other than multiple read-only access is not allowed by the program model, each desired datum that is invalid on entry to a program section is reloaded exactly once in that section. Let $I = (i_1, i_2, \cdots, i_n)$ be an ascending list of the invalidated element group indices referenced by a particular program section. Let $T_U(I)$ and $T_G(I)$ represent the number of transactions necessary for reloading these element groups for the ungrouped and grouped array layouts, respectively.

For clarity of illustration, in this section we first consider the simple case where (1) $e_j = g$ (constant element group size) for all arrays j in a given group, and (2) a subpage of an array before grouping contains an integer number of element groups, *i.e.* g divides s. The effect of removing these additional conditions is discussed in Theorem 5.5.4.

Given an index set $I = (i_1, i_2, \cdots, i_n)$ and an array (either ungrouped or grouped) in which each subpage contains exactly f element groups, the number of subpages containing element groups referenced by I, and thus the number of transactions, $t(I)$, can be derived for the following two cases, where $d_k = i_k - i_{k-1}$ is the *reference distance.* If $\forall k \in [2, n]$, $d_k \geq f$, each referenced element group is in a distinct subpage and $|I| = n$ transactions are required. If $\forall k \in [2, n]$, $1 \leq d_k < f$, then a set of $(x + (s/f) \times (i_n - i_1 + 1) + x')/s$ consecutive subpages is referenced, where x is size of the initial unreferenced portion of the first subpage, and x' is the final unreferenced portion of the last subpage. In summary,

$$t(I) = \begin{cases} n, & if\ \forall k \in [2, n], d_k \geq f \\ \frac{(s/f)(i_n - i_1 + 1) + x + x'}{s}, & if\ \forall k \in [2, n], 1 \leq d_k < f \end{cases} \tag{5.1}$$

Theorem 5.5.1 *For index set* $I = (i_1, i_2, \cdots, i_n)$ *and constant element group size,* g*, over* r (> 1) *ungrouped arrays, i.e.* $\forall j \in [1, r]$, $e_j = g$*:*
(1) if $\forall k \in [2, n]$, $d_k \geq s/g$*, then* $T_G(I) = T_U(I)/r$*,*
(2) if $\forall k \in [2, n]$, $d_k \in [q, s/g)$*, and* $c = 0$*, then* $T_G(I) \leq T_U(I)$*, and*
(3) if $\forall k \in [2, n]$, $d_k \in [1, q)$*, and* $c = 0$*, then* $T_G(I) \leq T_U(I)$*.*

Proof:

(1) Since g is the constant element group size for the ungrouped arrays, s/g is the number of element groups in a subpage for an ungrouped array. The condition $\forall k \in [2, n]$, $d_k \geq s/g$ implies that each referenced element group of an ungrouped array is in a distinct subpage. Thus, references to each ungrouped array result in n communication transactions. As a result, for r ungrouped arrays,

$$T_U(I) = r \times n$$

Because the element group size of the grouped array is greater than that of an ungrouped array, *i.e.* $e > g$, the number of element groups in a subpage for the grouped array, q, is less than that for an ungrouped array, s/g. Thus, $\forall k \in [2, n]$, $d_k > q$. Hence, each referenced element group of the grouped array is also in a distinct subpage. Therefore, we have

$$T_G(I) = n$$

Thus $T_G(I) = T_U(I)/r$.
(2) In this case, since $\forall k \in [2, n]$, $d_k \geq q$, we have

$$T_G(I) = n$$

However, because $\forall k \in [2, n]$, $d_k < s/g$, for each ungrouped array the n referenced element groups are in a set of consecutive subpages where each subpage in the set contains at least one referenced element group. Moreover, since each of the ungrouped arrays is assumed to have the same element groups size and to be aligned on a subpage boundary, these arrays will have the same initial unreferenced portion of their first referenced subpage and the same final unreferenced portion of their last referenced subpage. Let y and y' be these two unreferenced portions, respectively. Then, substituting g, y, and y' for s/f, x, and x' in Equation (5.1), we see that the number of communication transactions for each ungrouped array is $\frac{g(i_n - i_1 + 1) + y + y'}{s}$. This number of transactions is minimal when $d_k = i_k - i_{k-1} = q$, $\forall k \in [2, n]$. Therefore, we have

$$\begin{aligned} T_U(I) &= \frac{g(i_n - i_1 + 1) + y + y'}{s} \times r \\ &= \frac{g((i_n - i_{n-1}) + (i_{n-1} - i_{n-2}) + \cdots + (i_2 - i_1) + 1) + y + y'}{s} \times r \\ &\geq \frac{gq(n-1) + g + (y + y')}{s} \times r \end{aligned}$$

Since the number of blanks inserted between element groups in the grouped array, c, is 0, $q =$

$s/(e+c) = s/e$. Note that $gr = e$. Hence, $gqr = s$. It follows that

$$T_U(I) \geq \frac{s(n-1)+e+r(y+y')}{s} = (n-1) + \frac{e+r(y+y')}{s}$$

The fact that $T_U(I)$ is an integer indicates that the second term, $(e+r(y+y'))/s$, can be replaced by its ceiling function. Since $(e+r(y+y'))/s \geq e/s > 0$, this ceiling function is at least 1. As a result,

$$T_U(I) \geq (n-1)+1 = n = T_G(I)$$

(3) Since $\forall k \in [2,n]$, $d_k < q < s/g$, for each array (either an ungrouped array or the grouped array), the n referenced element groups are in consecutive subpages. As in case (2) above, let y and y', respectively, be the initial unreferenced portion of the first subpage and the final unreferenced portion of the last subpage for an ungrouped array. In addition, let z and z' be their counterparts for the grouped array. Then, according to Equation (5.1), and recalling that $gr = e$ and $c = 0$, we see that

$$\begin{aligned} T_U(I) &= \frac{g(i_n - i_1 + 1) + y + y'}{s} \times r = \frac{e(i_n - i_1 + 1) + r(y+y')}{s} \\ T_G(I) &= \frac{(e+c)(i_n - i_1 + 1) + z + z'}{s} = \frac{e(i_n - i_1 + 1) + z + z'}{s} \end{aligned}$$

Hence,

$$\begin{aligned} T_U(I) - T_G(I) &= \frac{e(i_n - i_1 + 1) + r(y+y')}{s} - \frac{e(i_n - i_1 + 1) + z + z'}{s} \\ &= \frac{ry + ry' - z - z'}{s} = \frac{(ry - z) + (ry' - z')}{s} \end{aligned}$$

To complete the proof, we show that $ry \geq z$ and $ry' \geq z'$.
Let $(i_1 - 1)g = ks + k_1 g$, where k and k_1 are integers and $k \geq 0$ and $0 \leq k_1 < s/g$. Then, the number of subpages spanned by the first $i_1 - 1$ element groups of an ungrouped array is

$$\frac{(i_1 - 1)g}{s} = k + \frac{k_1 g}{s}$$

By definition, $y = k_1 g$. Thus,

$$ry = rk_1 g = k_1 e$$

On the other hand, the number of subpages spanned by the first $i_1 - 1$ element groups of the grouped array is

$$\frac{(i_1 - 1)e}{s} = \frac{(i_1 - 1)gr}{s} = kr + \frac{k_1 gr}{s} = kr + \frac{k_1 e}{s}$$

and

$$z = (k_1 e) \bmod s = (ry) \bmod s$$

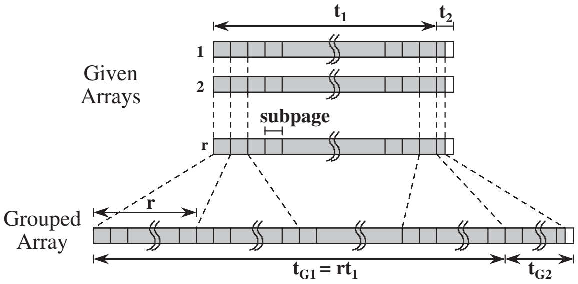


Figure 5.5: Ungrouped and grouped layouts.

Therefore,

$$ry \geq z$$

Similarly, let $i_n g = ks - k_1 g$, where k and k_1 are integers and $k \geq 0$ and $0 \leq k_1 < s/g$. Then, the final unreferenced portion of the last subpage of an ungrouped array is equal to $k_1 g$, *i.e.* $y' = k_1 g$, and

$$ry' = rk_1 g = k_1 e$$

On the other hand, the number of subpages spanned by the first i_n element groups of the grouped array is

$$\frac{i_n e}{s} = \frac{i_n g r}{s} = kr - \frac{k_1 g r}{s} = kr - \frac{k_1 e}{s}$$

and $z' = (k_1 e) \bmod s = (ry') \bmod s$. As a result,

$$ry' \geq z'$$

Since $ry \geq z$ and $ry' \geq z'$, $T_U(I) - T_G(I) \geq 0$, *i.e.* $T_G(I) \leq T_U(I)$.
□

Theorem 5.5.1 considers the reference distances in three ranges: $[s/g, \infty)$, $[q, s/g)$, and $[1, q)$, where s/g and q are the number of element groups in a subpage for an ungrouped array and the grouped array, respectively. This theorem gives us a quick check to identify some cases where the grouping is known to be beneficial (or at least not harmful). Note that in the first case of the theorem, the blank space inserted between element groups of the grouped array, c, is not necessarily 0. The major restriction of Theorem 5.5.1, however, is that all of the reference distances, d_k, must be in the same one of the three ranges. If $c = 0$, then Theorem 5.5.2 shows that for arbitrary reference distances $T_G(I) \leq T_U(I)$.

Theorem 5.5.2 *If* $\forall j \in [1, r]$ $e_j = g$ *and* $eq = s$ *(i.e.* $c = 0$*), then* $T_G(I) \leq T_U(I)$.

Proof: Since $eq = s$ and $rg = e$, the number of element groups per subpage of each of the given arrays, $s/g = rq$, is also an integer. Therefore, all element groups in the k^{th} subpage of each given array will be reallocated in the k^{th} set of r consecutive subpages of the grouped array as in Figure 5.5.

Let h be the declared size of the group dimension, and $t_1 = \lfloor \frac{g \times h}{s} \rfloor$, then I can be divided into I_1 and I_2, such that $I = I_1 \circ I_2$ (where $\circ$ stands for concatenation), $I_1 = I \cap \{1, 2, 3, \cdots, t_1 \frac{s}{g}\}$, and

$I_2 = I \cap \{t_1\frac{s}{g}+1, t_1\frac{s}{g}+2, \cdots, h\}$. Since the subpages of I_1 are disjoint from those of I_2, both for the given arrays and the grouped array, we can prove $T_G(I) \leq T_U(I)$ by proving that $T_G(I_1) \leq T_U(I_1)$ and $T_G(I_2) \leq T_U(I_2)$.

(1) $T_G(I_1) \leq T_U(I_1)$: If element groups 1 through $t_1\frac{s}{g}$ are all referenced, $T_U(I_1) = T_G(I_1) = rt_1$. Whenever a subpage in a given array is not referenced, resulting in a total of r non-referenced subpages for the r given arrays, the corresponding r subpages are unreferenced in the grouped array. Thus $T_G(I_1) \leq T_U(I_1)$.

(2) $T_G(I_2) \leq T_U(I_2)$: If $I_2 = \emptyset$, then $T_G(I_2) = T_U(I_2) = 0$. If $I_2 \neq \emptyset$, then since the element groups associated with I_2 are all in the last subpage of each given array, $T_U(I_2) = r$. However, since there are no more than r subpages in T_{G2}, $T_G(I_2) \leq r$. Hence, $T_G(I_2) \leq T_U(I_2)$.

□

The above theorems specify some special cases where array grouping should always be considered. Theorem 5.5.3 deals with the more general case where c is not necessarily 0.

Theorem 5.5.3 *For* $I = (i_1, i_2, \cdots, i_n)$, $t = T_U(I)/r$, *and* $\forall j \in [1, r]$, $e_j = g$:
(1) if $e \geq ng/t$, $T_G(I) \leq T_U(I)$
(2) if $e \leq ng/qt$, $T_G(I) \geq T_U(I)$

Proof: Since t is the number of transactions for a given array, ng/t is the average number of useful data elements in a transaction before grouping. Since after grouping each transaction has at least e useful data elements, if $e \geq ng/t$, then $T_G(I) \leq T_U(I)$. However, since after grouping a transaction has at most eq useful data elements, if $eq \leq ng/t$, then $T_G(I) \geq T_U(I)$.

□

Note that Theorem 5.5.3 does not resolve whether grouping has a positive benefit for arbitrary reference patterns with $c \neq 0$ and $ng/qt < e < ng/t$. However, intuitively, if e is close to ng/t, then there is a good chance that the grouped array will cause less communication than the given arrays; on the other hand, if e is close to ng/qt, then the possibility of reducing the communication is low. So far, we have only considered constant element group sizes over all ungrouped arrays. Theorem 5.5.4 extends Theorem 5.5.3 to allow different ungrouped arrays to have different element group sizes.

Theorem 5.5.4 *For* $I = (i_1, i_2, \cdots, i_n)$,
(1) if $e \geq n/\min\{t_j/e_j | j \in [1, r]\}$, $T_G(I) \leq T_U(I)$
(2) if $e \leq n/(q\max\{t_j/e_j | j \in [1, r]\})$, $T_G(I) \geq T_U(I)$
where t_j *is the number of transactions for ungrouped array* j.

Proof: Note that $n/\min\{t_j/e_j | j \in [1, r]\} = \max\{(ne_j)/t_j | j \in [1, r]\}$, which is the maximum average number of useful data elements in a transaction among all ungrouped arrays. Since after grouping, each transaction has at least e useful data elements, if $e \geq \max\{(ne_j)/t_j | j \in [1, r]\}$, then $T_G(I) \leq T_U(I)$. On the other hand, $n/\max\{t_j/e_j | j \in [1, r]\} = \min\{(ne_j)/t_j | j \in [1, r]\}$, which is the minimum average number of useful data elements in a transaction among all ungrouped arrays. Since after grouping a transaction has at most eq useful data elements, if eq is no greater than $\min\{(ne_j)/t_j | j \in [1, r]\}$, then $T_G(I) \geq T_U(I)$.

□

The above theorems provide insight into why and when array grouping works. They can be used *a priori* to estimate whether an array grouping is beneficial. However, to do so, in most cases we need to know the nonconsecutive reference index set. If these nonconsecutive references are represented by a function of loop induction variables, then they can be obtained by substituting the appropriate ranges (or sets) of values for the loop induction variables in this function. However, if arrays are referenced indirectly through the elements of another array, then we may have to insert appropriate reference-collection instrumentation code in the program and collect these references at run time. Similar methods have been used in other studies for indirect reference arrays, *e.g.* [79]. These methods are, however, beyond the scope of this dissertation, and are not discussed further here.

5.6 Experiments and Discussion

An actual finite element application code was used to demonstrate the effectiveness of array grouping. The data set contains more than 20000 nodes and 20000 elements and a *node connectivity array* to maintain the element-node connection information. Each node is associated with a number of parameter arrays indexed by the node identifier. Among them, we considered only the force and moment arrays. Each node's force and moment are updated in every simulation time step. Terms associated with an element are first computed based on the motion of the element, and are saved in *partial_force* and *partial_moment* arrays. A node's final force and moment are then obtained by summing all of the relevant terms by using the elements of the node connectivity array to index into the *partial_force* and *partial_moment* arrays, *i.e.* by indirect array references.

The original program was first restructured by interchanging two loop levels to increase its performance and to reduce other effects in the results. The data grouping technique was then applied to this loop-interchanged program. In the array-grouped program, the *partial_force* and *partial_moment* arrays were grouped according to the result of the PSAT reference pattern analysis. Since this chapter is concerned with the effect of communication and data subcache misses, we present only these parts of the experimental results. The measured data were obtained from the KSR1's *pmon* (performance monitor) and represent the measured performance of this program section from simulation time steps 2 to 100. Step 1 is eliminated due to initialization effects.

5.6.1 Local Cache Misses

In the KSR1, the total local cache misses (summed over all caches) directly reflects the amount of ring communication. The loop interchange only changes the array reference order while maintaining the same workload in each processor. Thus it should produce the same number of local cache misses as the original program if only coherence misses occurred. However, the results in Figure 5.6 show that on average the loop-interchanged program has $\approx 4\%$ fewer local cache misses than the original program. This result is evidence that even for a large number of processors, there are still other types of misses in the local cache.

As the number of processors, p, increases, the total compulsory misses may increase due to initial loads of shared cache blocks in multiple caches, but the total conflict misses may decline due to the effective increase in set associativity spread over multiple caches; however, the number of misses of these types should not be impacted significantly. In contrast, total capacity misses should decrease rapidly to 0 and coherence misses should increase significantly as p increases. Figure 5.6 also shows that the total local cache misses increase monotonically after 16 processors. This increase is obviously dominated by the coherence misses and is least severe for the array-grouped program which eliminates 4% of all misses relative to the original program section for

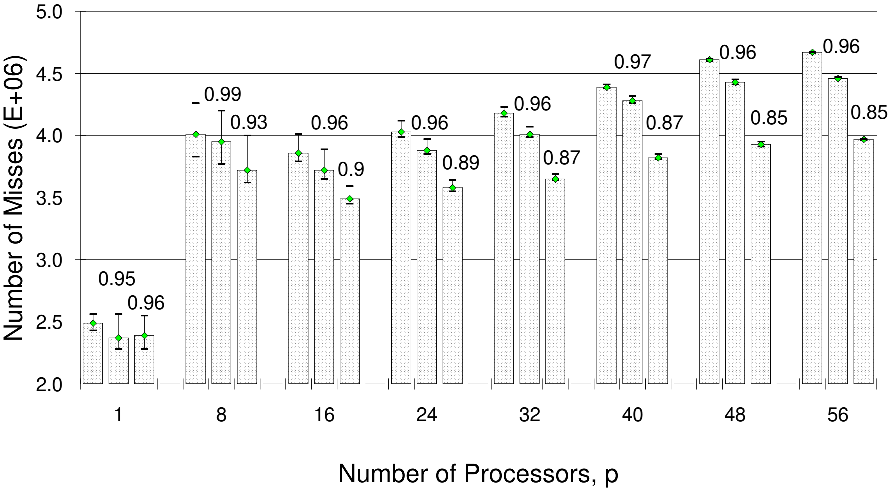


Figure 5.6: Total local cache misses: For each p, the three bars from left to right represent the original, loop-interchanged, and array-grouped programs. Each bar shows the average of five measurements (minimum and maximum are also shown). The numbers above the 2 right bars give the bar height relative to the leftmost bar for that p.

$p = 1$, up to 15% for $p = 56$. Thus for this parallel application, the array-grouping technique significantly reduces this fundamental performance degradation factor.

For local cache misses, the *pmon* measured data are always less than the worst-case model prediction, as shown in Figure 5.7. Apparently not all data expected to be invalidated are actually invalidated, *i.e.* none of the codes actually follow the worst-case communication assumption and the actual number of local cache misses is thus less than the model prediction. In this example, however, the predicted improvement due to array grouping relative to the original layout is remarkably close to the measured result.

5.6.2 Data Subcache Misses

Figure 5.8 shows that the loop-interchanged program dramatically reduces the total misses, summed over all data subcaches, by 25% to 44% relative to the original program over all p. The array-grouped program further reduces the misses by an additional 12% to 15%. The loop-interchanged program reduces the number of iterations between reusing array elements; thus these elements are more likely to remain in the data subcache until reuse. The array-grouped program increases the number of useful data in a cache block, and thus has a smaller working set, which further reduces subcache block replacements.

Although the improvement is significant, it decreases as p increases. In the original program, a processor needs a large cache to hold all the reusable data. Hence, the limited size of the KSR1's data subcache (256 KB) causes many capacity misses, and cache misses of this type will decrease significantly as p increases and the reusable data per processor decreases. The other two versions of the program, as explained above, generate much fewer capacity misses, even for $p = 1$. The other data subcache misses include compulsory and coherence misses whose total increases with p, and conflict misses whose total decreases. On balance, Figure 5.8 shows a very slight increase

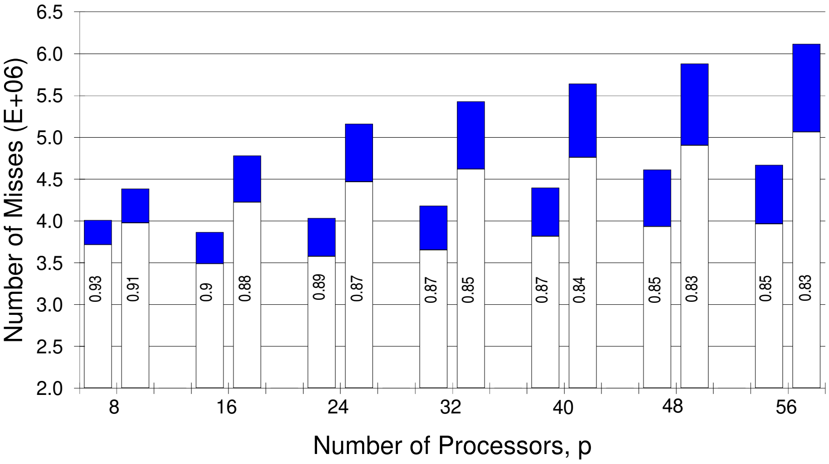


Figure 5.7: Measured vs. worst-case local cache misses: For each p, the left bar is the *pmon* measured data, and the right bar, the worst-case model prediction. The lower segment of each bar is the number of local cache misses of the array-grouped program, while the full bar is for the original program. The lower bar height relative to the full bar is shown numerically.

with p. As the original program improves with increasing p, the relative subcache miss benefit of the improved programs is reduced.

5.6.3 User time

Figures 5.9 and 5.10 show the total user time (summed over all threads) and the maximum user time (worst thread user time), respectively. The total user time represents the overall workload, and the maximum user time is typically used to represent the execution time of the program. Total user time gives an upper bound on the speedup, which is achievable only with perfect load balance. It also indicates the overhead due to parallel computing, namely the growth in total processor resources consumed as p increases. Actual speedup is measured by dividing the maximum user times. In our experiments, both times are reduced by the loop interchange and further reduced by the array grouping. On average, the loop-interchanged program reduces the total time of the original program by 11 to 29%, and reduces the maximum time by 4 to 29%. The array-grouped program reduces the total user time by an additional 8 to 12% and the maximum user time by an additional 7 to 18%.

In Figure 5.9, from $p = 1$ to $p = 8$ the total user time jumps due to the increase in communication. Then, between $p = 8$ and $p = 28$ it increases very slightly (almost unchanged). After $p = 28$, it grows steadily. For $p > 28$, some processors are located on the other ring, *i.e.* some communication must travel over multiple (3) rings. Although there are actually 32 processors per ring, several processors on each ring are reserved for system use in our installation. The occurrence of these multiple-ring communication transactions whose latency is at least 2 times greater than the latency of single-ring communication results in the significant increase in the total user time at $p = 32$ relative to that at $p = 28$. Because the amount of multiple-ring communication increases as p increases, the total user time increases monotonically after $p = 28$. The same effect also disrupts

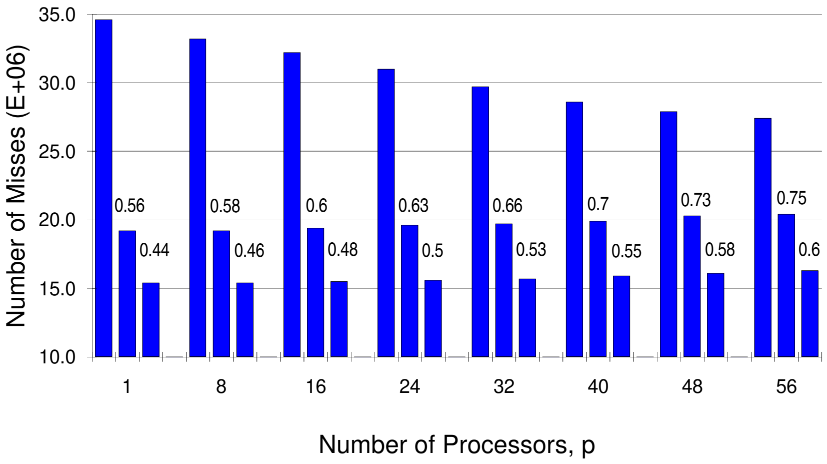


Figure 5.8: Total data subcache misses: For each p, the bars from left to right represent the original, loop-interchanged, and array-grouped programs, respectively. Each bar shows the average values from the five measurements (the measurements are within ± 0.6%). The numbers above the 2 right bars give the bar height relative to the leftmost bar for that p.

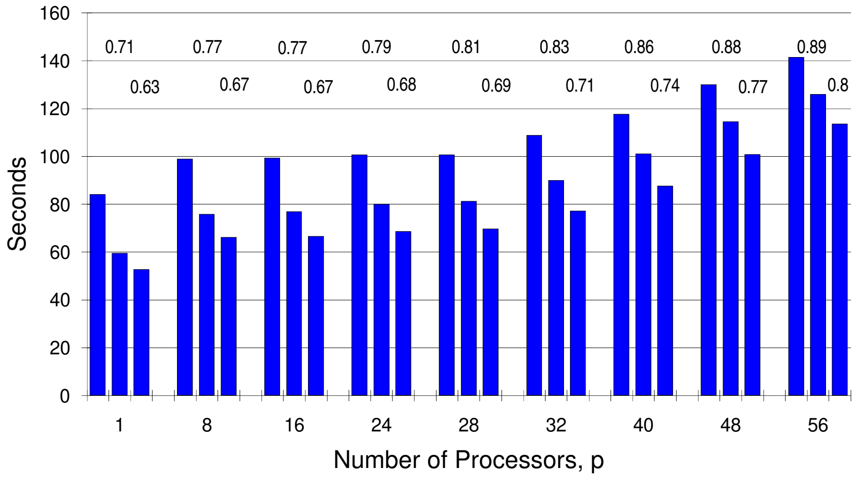


Figure 5.9: Total user time: In each processor category, the bars from left to right represent the original, loop-interchanged, and the array-grouped programs, respectively. Each bar shows the average value of the five measurements (the measurements are within ± 2%). The numbers above the 2 right bars give the bar height relative to the leftmost bar for that p.

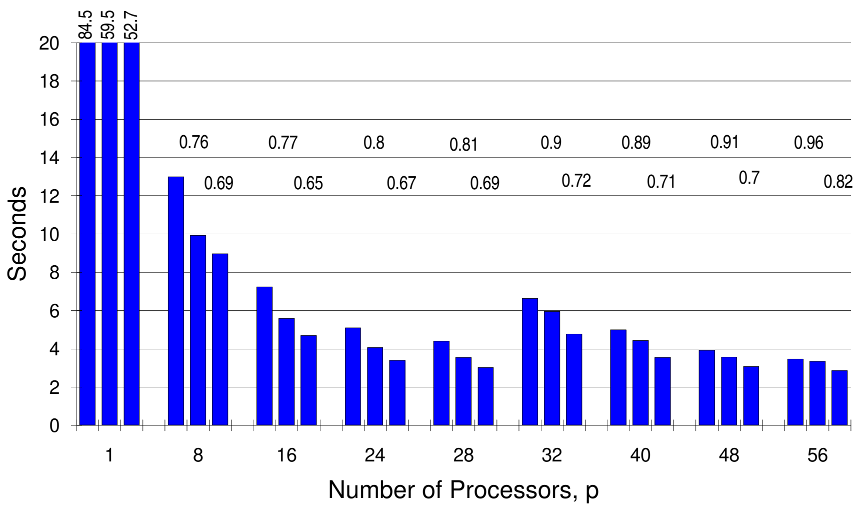


Figure 5.10: Maximum user time: In each processor category, the bars from left to right represent the original, loop-interchanged, and the array-grouped programs, respectively. Each bar shows the average value of the five measurements (the measurements are within ± 3%). The numbers above the 2 right bars give the bar height relative to the leftmost bar for that p, except for $p = 1$ where the 3 numbers show the absolute bar height.

the monotonic decrease of the maximum user time at $p = 32$ (Figure 5.10).

Figure 5.11 shows the speedup of the three programs. By dividing the user time of a single processor run of each program by the maximum user time and by the average user time (total user time/p), we obtain two speedup vs. p curves for each program. The difference between each pair of curves is caused by load imbalance. Note that the aforementioned multiple-ring communication effect causes the maximum user time speedup curve to decrease from $p = 24$ to $p = 32$; however, although the average user time speedup curve is nearly flat between $p = 28$ and $p = 32$, its slope is never negative in these experiments as shown in Table 5.2. For all p, the original program has better speedup (although worse run time) than the loop-interchanged and array-grouped programs. This occurs primarily because the original program contains many capacity misses in the subcache and these are reduced significantly as p increases, thus adding to this program's speedup.

Unlike the original program, the other two programs have very stable subcache performance; thus, their speedup behavior is more heavily influenced by communication. The array-grouped program, which has less communication than the loop-interchanged program, should have somewhat better speedup; and Figure 5.11 shows that it does. However, the difference in speedup between these programs is limited. Compared with $p = 28$ (the maximum number of processors allocated to an application in one ring), the loop-interchanged and array-grouped programs only have slightly better speedup at $p = 56$ (maximum number of allocated processors in 2 rings), as shown in Table 5.2. Therefore, considering cost/performance it seems worthless to run these programs on a two-ring system.

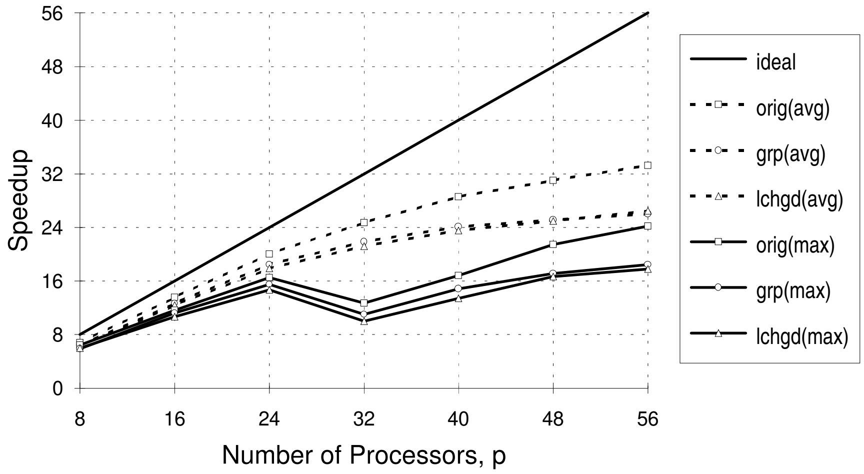


Figure 5.11: Speedup of the original, loop-interchanged, and array-grouped programs.

	Maximum			Average		
p	Org.	L-chnged	Arr-grped	Org.	L-chnged	Arr-grped
8	6.47	6.00	5.88	6.80	6.29	6.37
16	11.62	10.65	11.23	13.55	12.39	12.67
24	16.50	14.67	15.50	20.03	17.86	18.46
28	19.13	16.75	17.46	23.37	20.52	21.19
32	12.68	10.01	11.04	24.72	21.19	21.87
40	16.81	13.41	14.83	28.57	23.55	24.09
48	21.44	16.69	17.12	31.02	24.96	25.13
56	24.20	17.79	18.45	33.27	26.47	26.02

Table 5.2: Speedup of the original, loop-interchanged, and array-grouped programs (in maximum user time and average user time).

5.7 Conclusion

We have suggested an array-grouping technique to mitigate the problems resulting from nonconsecutive references of arrays in parallel programs. The methodology identifies nonconsecutively referenced arrays and includes algorithms to choose final array groups, methods to reorganize arrays and modify the program, and applications of appropriate loop transformations to improve memory access sequences. The purpose of the entire procedure is to improve locality, and thus improve program performance. We have also derived a number of theorems that specify the conditions under which applying this technique can be beneficial. The effectiveness of the technique was evaluated experimentally by running a finite element program on the KSR1. Compared with the original program, the array-grouped program on average reduced total local cache misses by 15%, total subcache misses by 40%, and maximum user time by 18% for 56 processors.

CHAPTER 6

Conclusion

6.1 Summary of Results

In this dissertation, we have proposed a goal-directed approach to answer the two performance tuning questions: how to establish a performance objective and how to efficiently achieve this performance objective. A goal-directed method has been proposed for each of three major components of system performance: CPU, cache, and communication. Each goal-directed method contains a set of methods, each designed for solving one of the identified performance problems together with a model or a set of theorems to indicate for an application under consideration (1) where the performance problems are, (2) how much they can be improved, and (3) when the application of the performance tuning techniques should stop. The corresponding case studies have shown the effectiveness of this approach.

In Chapter 2 we have suggested a machine-application model that yields an upper bound on CPU performance to be used as a performance objective, and derived a mechanism for approaching this performance goal. To verify this approach, we have carried out a case study using the IBM RS/6000 and the Livermore Fortran Kernels 1-12 as running examples. Although the RS/6000 is generally credited with good performance on several scientific codes, its original performance averaged over these 12 Livermore Fortran Kernels only reached 52.4% of the performance bound. We have successfully identified its four performance degradation factors: redundant loads and stores, read after write dependence stalls, floating-point store versus write-back conflicts, and poor fine-grain scheduling, and we derived techniques for solving each of these performance problems. Using these techniques, we raised the average performance from 52.4% to 93.7% of the performance bound which represents a performance speedup of 1.79. Since this machine-application bound model is achievable, using this bound as a performance goal should focus the tuning effort on the most effective techniques. To ease the process of determining a performance objective, we have designed a tool that analyzes application workloads and produces the performance bound automatically.

Chapters 3 and 4 provide a goal-directed approach for tuning memory performance. In Chapter 3 we have demonstrated an analytical method for deriving application-specific cache conflict behavior models. The method models self, cross, and ping-pong conflict misses separately for each array, and is suitable for a variety of cache organizations. In Chapter 4 we have presented three conflict miss reduction methods, each for eliminating or reducing one type of conflict miss. These methods are based on three techniques that either modify or exploit the data layout: array padding, initial address adjustment, and access resequencing. To evaluate the accuracy of the model and the effectiveness of the cache miss reduction methods, we have performed a case study using a blocked matrix multiply program as an example. In our experiments, the accuracy of the model is within 11% of the simulation results, and each type of conflict miss can be effectively reduced by a proposed conflict miss reduction method. For the cases that use the largest possible square block in our experiments, the total number of cache misses was reduced to 24% to 32% of the original. Since each of the three conflict miss reduction methods was designed for the specific performance problems reported by the model, the model and these methods together can directly and effectively solve the cache performance problems.

In Chapter 5 we have proposed an array grouping technique to mitigate the problems resulting from nonconsecutive array references in parallel programs. To guide the application of this technique, we have also proved a number of theorems that specify the conditions under which application of this technique can be beneficial. The methodology identifies nonconsecutively referenced arrays and includes algorithms to choose final array groups, methods to reorganize arrays and modify the program correspondingly, and applications of appropriate loop transformations to improve memory access sequences. The purpose of the entire procedure is to improve locality, and thus improve program performance. The effectiveness of the technique was evaluated experimentally using a finite element program on the KSR1. Compared with the original program, the array-grouped program reduced total local cache misses by 15%, total subcache misses by 40%, and maximum user time by 18% for 56 processors.

Given an application, the following procedure is suggested for tuning its performance. First, achieve the best delivered CPU performance by using the approach introduced in Chapter 2. Although the techniques discussed in this chapter are primarily compiler techniques, some of them can actually be implemented in source code. Second, enhance communication efficiency by the array grouping technique proposed in Chapter 5. Note that this step is skipped if the program involves no nonconsecutive array references. Finally, improve cache performance with the model and cache-miss-reduction techniques described in Chapters 3 and 4. Performance tuning for communication performance is considered before cache performance so that they can be treated independently and still achieve their intended purposes. Otherwise, if cache performance is dealt with before communication performance, then since the new grouped array may have poor relative data layout with respect to other arrays, the once tuned cache performance may again become worse and require tuning.

The array grouping technique, as mentioned in Chapter 5, can also reduce cross conflict misses between different arrays. Thus, it can be considered as a cache-miss-reduction technique. Nevertheless, its application is limited in comparison with the techniques discussed in Chapter 4, namely array padding, initial address adjustment, and access resequencing. For example, no two arrays in the blocked matrix multiply program are appropriate for grouping according to the mechanism in Chapter 5, while the total cache misses of this program were significantly reduced by the techniques in Chapter 4.

6.2 Suggestions for Future Research

Although the goal-directed performance tuning approach has been shown to be effective, several issues need to be addressed before this approach can be used in general practice. Some of these issues are discussed below.

First, the method for creating cache behavior models may be too complicated for general users. The complexity must be hidden from the users so that they are willing to use this method. Note that the parameters used in constructing the model are extractable from the source program. Hence, one possible solution is to generate a set of library routines which implement the critical procedures of the method, and simply provide these parameter values to the users. A user could then quickly build a cache behavior model for his application. To achieve a robust implementation, further work is needed to enhance the applicability of these routines. In addition, this method should be extended to parallel computing environments, for example by including coherence misses in the model.

Second, communication performance needs to be improved further. Although the array-grouping technique can eliminate a significant amount of communication, many communication transactions may remain. Most of these are due to accesses to shared data among the processors rather than to superfluity in communication. Hence, the next direction to improve communication performance further is to model and provide techniques for the data migration problem. The ultimate objective is to build a uniform goal-directed performance tuning tool for communication.

Finally, the goal-directed method is not yet automatic. Goal-directed performance tuning should eventually be made automatic for ease of use. As this was considered at the beginning of this research, all of the models were derived using analytical methods and software techniques that are suitable for incorporation into a compiler system or for implementation in a separate automatic performance tuning tool. However, only the machine-application upper bound model is currently implemented in an automatic tool.

In addition, the cache behavior model and the processor-based machine-application upper bound model can be combined and used to predict uniprocessor performance with cache effects. Our experimental results show that the cache model is fairly accurate. Hence, if the number of cache misses can be converted into cycles, this information can be incorporated within the machine-application bound model. Converting cache misses into cycles is trivial for a cache with uniform miss penalties. However, to reduce average miss penalties, most current microprocessors have been designed with nonuniform miss penalty caches. Although this adds difficulties, this issue is worthy of future exploration.

APPENDIX

APPENDIX A

Computing the Expected Number of Cross Conflict Misses in a Cache Set for the Matrix Multiply Example

The number of conflict misses is considered for each array pair and for each conflict footprint (CF). In the matrix multiply program, the cross conflict misses are mainly due to the conflicts between arrays Q and R and between arrays P and R. We thus only consider array pairs (Q, R) and (P, R). Recall that in this program, loop J is the coverage loop, and a coverage loop iteration includes one block column each of P and Q, and one block (B block columns) of R. A block column is assumed to be no larger than "one way" of the cache, *i.e.* a cache set can be mapped into by at most one line of P, one line of Q, and B lines of R. Therefore, the possible CFs for (P, R) and those for (Q, R) all have (i, j) that satisfy $i \in \{0, 1\}$, $j \in \{0, 1, 2, \ldots, B\}$, where i and j are not both 0.

Among these CFs for array pairs (P, R) and (Q, R), we only need to consider a) CFs $(1, j)$, $\forall j \geq D$ for computing the numbers of cross conflict misses of arrays P and Q, respectively, and b) CFs $(0, D)$ and $(1, D)$ for computing the number of cross conflict misses of R. The reasons are as follows:

a) Since arrays P and Q are not reused in the coverage loop, from Section 3.4.2 we know that their misses in a track can be determined by the track itself, without considering the prior track. To have cache misses, the track must contain some lines of the array, and the total number of distinct lines of the track must be greater than the degree of associativity, D. This implies that the cross conflict misses of Q in Q-R conflict and those of P in P-R conflict can only occur in a track of CF (i, j), where $i = 1$ and $j \geq D$.

b) However, since array R is reused in the coverage loop, the initial status of the r lines in a track is affected by the behavior of the previous track. Hence, the number of cross conflict misses of R in a track is determined not only by the track but also by the track immediately before it. Furthermore, since only tracks with the same number of r elements can appear in succession, CFs $(0, j)$ and $(1, j)$ should be considered together. Recall that once ping-pong conflict misses have been evaluated, our method counts all remaining conflict misses of an array that is reusable in the coverage loop as self conflict misses if the number of distinct lines of this array in the footprint is greater than D. Hence, it is not necessary to count cross conflict misses of R for CFs $(0, j)$ and $(1, j)$ if $j > D$. On the other hand, if $j < D$, then no track of CFs $(0, j)$ or $(1, j)$ causes any conflict misses of R. As a result, we have to consider only CFs $(0, D)$ and $(1, D)$ for computing cross conflict misses of R in either the Q-R or P-R conflict case.

From Section 3.4.2, we know that the key question for computing the number of cross conflict misses is how to compute the expected number per set of cross conflict misses of an array for each CF (i, j) of concern, *i.e.* $m_A(i, j)$, for array A, where $A = P$, Q, or R, since the appropriate linear

$m_A(i,j)$	the expected number per set of cross conflict misses of array A related to CF (i,j)
$m_A(t)$	the expected number of cross conflict misses of array A for track t
$Pr[t]$	probability that a given cache set has track t of CF (i,j)
$n_s(i,j)$	the expected number of cache sets that have CF (i,j)
$T(i,j)$	set of all possible tracks associated with CF (i,j)
$n(t)$	number of reference sequences for track t
$m_A(t\|t')$	number of cross conflict misses of A for track t when track t' is immediately before t

Table A.1: Summary of notation and definitions.

combination of the $m_A(i,j)$ values yields total cross conflict misses. From Equation (3.6), we know that

$$m_A(i,j) = \sum_{t \in T(i,j)} m_A(t)Pr[t] = \frac{n_s(i,j)}{S} \sum_{t \in T(i,j)} \frac{m_A(t)n(t)}{\sum_{t_k \in T(i,j)} n(t_k)}$$

Since $n_s(i,j)$ can be directly computed from the footprint histogram of the arrays by using Equation (3.3), we now show the methods for computing $T(i,j)$, $n(t)$, and $m_A(t)$ for the block matrix multiply program in Sections A.1, A.2, and A.3, respectively. Table A.1 summarizes and defines some useful terminology.

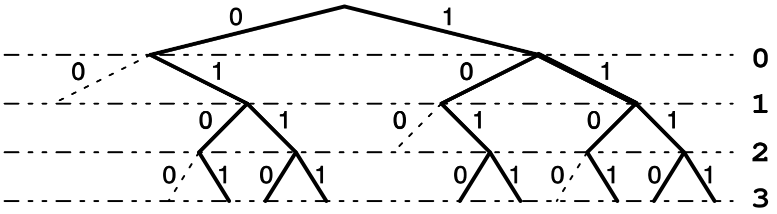
A.1 Find All Possible Tracks, $T(i,j)$

With respect to either (Q,R) or (P,R), it is evident that there is only one track for CF $(0,j)$, which is $r_1 r_2 \cdots r_j$. However, there may be many tracks for CF $(1,j)$. The common characteristic of these tracks is that each of them contains a one line footprint for the P (or Q) array. This characteristic allows us to represent a track by a binary string. By doing so, we can then list all possible tracks in a binary tree in which each path from the root to a leaf corresponds to a binary string that represents a possible track. We call this binary tree the *track tree*. In the following, we discuss how to represent a track by a binary string using the notations of Q-R conflict. The case of P-R conflict is similar.

Consider CF $(1,j)$ for array pair (Q,R). Note that in the sequence of the j-line footprint of R, there are $j+1$ positions where a q_1 can be located as shown below:

`Position:` 0 1 2 3 ... j-1 j
`Footprint:` $r_1\ r_2\ r_3 \cdots r_{j-1}\ r_j$

These positions include the two ends, *i.e.* before r_1 and after r_j, and the $(j-1)$ spaces between two adjacent r elements. Denote the position before r_1 as position 0, the position between r_1 and r_2 as position 1, and so on. A possible track of CF $(1,j)$ can be formed by placing a q_1 at any of these positions under the constraints implied by the track properties. If we assign 1 to each position where a q_1 is located and 0 to the others, then the $(j+1)$-digit binary string, from position 0 to j, can uniquely represent the track. Due to different track properties for the cases of Q-R conflict and P-R conflict, the rules that generate these binary strings are different.



(a) Track tree for $B \geq 4$, $1 \leq j \leq 3$.

---

1. A 1-edge can follow either a 0-edge or a 1-edge.
2. A 0-edge can only follow a 1-edge.
3. The number of 1-edges in a path cannot exceed B.

(b) Rules for generating the track tree.

Figure A.1: Track tree and its rules for Q-R conflict.

A.1.1 Tracks of CF $(1, j)$ for Array Pair (Q, R)

To find the properties of the tracks of CF $(1, j)$ for (Q, R), we analyze how arrays Q and R are referenced in a coverage loop (loop-J) iteration. In this program, each coverage loop iteration contains B inner loop (loop-K) iterations. Each inner loop iteration references a block column of Q and a block column of R. The Q block column is the same for all inner loop iterations in a given coverage loop iteration and is changed for each new coverage loop iteration. In contrast, the R block column is different in different inner loop iterations; however, the block columns of the same R block are reused (in the same order) in successive coverage loop iterations. The way that arrays Q and R are referenced in a coverage loop iteration in this program is thus similar to the way that arrays X and Y are referenced in the example of Section 3.4.2. Hence, the tracks here should have the same properties as the tracks in that example, namely, each r element is always adjacent to an appearance of the single q element, q_1. In addition, since each inner iteration can contribute at most one q_1 in a track, a track can have at most B q_1 elements.

Figure A.1b lists the set of rules, corresponding to the observations above, for generating the track tree for CF $(1, j)$ with respect to array pair (Q, R). The first two rules prohibit consecutive 0-edges in a path, which is equivalent to enforcing the property that every r element has an adjacent q_1. In addition, the rules restrict the maximum number of 1-edges in each path to B, which limits the number of q_1 elements in a track.

Figure A.1a shows the track tree generated up to 4 levels by the rules in Figure A.1b with $B \geq 4$. All paths terminated at horizontal line j, where $j = 1, 2, 3, \ldots$, correspond to all possible tracks of CF $(1, j)$. For example, with respect to (Q, R) the possible tracks of CF (1,2) correspond to all paths terminated at horizontal line 2 in Figure A.1a. These paths produce the binary strings 010, 011, 101, 110, and 111 which correspond, respectively, to tracks $r_1q_1r_2$, $r_1q_1r_2q_1$, $q_1r_1r_2q_1$, $q_1r_1q_1r_2$, and $q_1r_1q_1r_2q_1$. From the tree, we also see that when $j \geq 1$, each path has at least one 1-edge since otherwise the corresponding conflict footprint would be CF $(0, j)$, instead of CF

$(1, j)$.

Note that the tree in Figure A.1a can be extended to as many levels as needed for the maximum j of interest and that this one tree suffices for all smaller values of j down to $j = 1$. Furthermore, the tree can be constructed for the largest B of interest and used for smaller B by simply pruning illegal branches.

A.1.2 Tracks of CF $(1, j)$ for Array Pair (P, R)

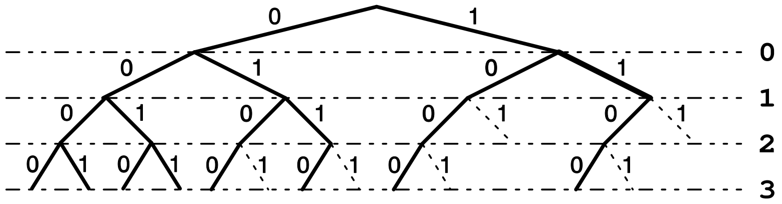
We now analyze how arrays P and R are referenced in a coverage loop iteration for deriving the properties of the tracks of CF $(1, j)$ for (P, R). In Section A.1.1, we discussed how array R is referenced in both coverage loop and inner loop iterations. We now describe how array P is referenced. Actually, in a coverage loop iteration, array P has the same reference behavior as array Q. However, in an inner loop iteration, only one element of P is accessed (rather than a whole block column as for Q). The P element is multiplied by every element of the R block column referenced in the same inner loop iteration. A typical compiler would access this P element once (from the cache) before any element of the R block column and save it in a register. Since successive inner loop iterations will reference consecutive P elements, each P line is referenced once in each of L successive inner loop iterations with L consecutive block columns of the R block. Therefore, a track of CF (1, B) must contain one r element for the references to one line for each of the B block columns and therefore must look like

$$r_1\ r_2\ \cdots r_{i-1}\ p_1\ r_i\ p_1\ r_{i+1}\ p_1\ r_{i+2}\ \cdots\ p_1\ r_{i+L-1}\ r_{i+L} \cdots r_B$$

For CF $(1, j)$ where $j < B$, $B - j$ block columns of the R block do not have any line that is mapped to this set. Thus, only j r elements will appear in the track. Furthermore, if a missing column is one of the block columns that is interleaved between p_1 references, the p_1 references immediately before and immediately after this block column is referenced will merge into one element of the track. Thus, the number of p_1 elements may be less than L. However, the nature of the interleaving between the p_1 elements and a sequence of r elements remains unchanged, regardless of the number of p_1 elements.

According to these observations, we can summarize the properties of a track of CF $(1, j)$ for (P, R) as follows: (1) p_1 appears in an interleaved fashion with a sequence of r elements; (2) the first p_1 can be at any position and the rest follow in consecutive positions with a single r element between them; (3) the number of p_1 cannot exceed L or $j + 1$, whichever is less. These properties are transformed into the set of rules in Figure A.2b for generating the track tree for conflicts between P and R. The first three rules can generate all of the paths whose corresponding tracks satisfy the track properties, as well as a path with all 0-edges. Since when $j \geq 1$, every path must contain at least one 1-edge, we add the fourth rule to remove this all 0-edge path. Figure A.2a shows a track tree generated up to 4 levels by the rules for $L = 2$. Similarly, all paths terminating at line j, where j = 1, 2, 3, ..., correspond to all possible tracks of CF (1, j) for (P, R) when L = 2, plus the all 0-edge path which is subsequently removed by rule 4.

Like the track tree in Figure A.1a, the tree in Figure A.2a can also be extended to as many levels as needed for the maximum j of interest, and this one tree suffices for all smaller values of j down to $j = 1$. After we have track trees similar to those in Figure A.1a and Figure A.2a, we can then list all possible tracks for CF $(1, j)$ for array pairs (Q, R) and (P, R). In addition, the tree structure can help to calculate $n(t)$ and $m_A(t)$ for each track t, as discussed in the following sections.



(a) Track tree for $L = 2$, $j \leq 3$.

---

1. A 0-edge can follow either a 0-edge or a 1-edge.
2. A 1-edge can follow a 0-edge if the path does not contain an 1-edge; otherwise, it can only follow another 1-edge.
3. The number of 1-edges in a path cannot exceed L.
4. Temporarily remove the path containing only 0-edges whenever a particular value of j is considered.

(b) Rules for generating the track tree.

Figure A.2: Track tree and its rules for *P*-*R* conflict.

A.2 Count the Number of Reference Sequences, $n(t)$, for Each Track

With respect to either (Q, R) or (P, R), the number of reference sequences, $n(t)$, for a track t of CF $(0, j)$ is certainly 1, however, that for a track of CF $(1, j)$ does need some effort to obtain. For the case of Q-R conflict, we know from Section A.1.1 that the same q_1 line will be referenced in every inner loop iteration. Hence, there are B q_1 elements in a (condensed) reference sequence. Therefore, from Equation (3.12), if a track t contains h q_1 elements, then

$$n(t) = \mathbf{C}_{h-1}^{B-1} = \frac{(B-1)!}{(B-h)!(h-1)!} \tag{A.1}$$

Similarly, in the case of P-R conflict, from Section A.1.2 we know that a reference sequence should have L p_1 elements. Thus, if a track t contains h p_1 elements, then

$$n(t) = \mathbf{C}_{h-1}^{L-1} = \frac{(L-1)!}{(L-h)!(h-1)!} \tag{A.2}$$

In both cases, $n(t)$ for track t of CF $(1, j)$ can be computed if h is known since B and L are both known parameters. Since the value of h for a track is equal to the number of 1-edges of the corresponding path in a track tree, we can derive h for each track by constructing an algorithm that searches for the number of 1-edges in each path. We can then calculate $n(t)$ for each track by using Equation (A.1) or (A.2).

A.3 Count the Number of Misses, $m_A(t)$, for Each Track

In this section, we describe how to compute the number of cross conflict misses of arrays P, Q and R for each track t, *i.e.* $m_P(t)$, $m_Q(t)$, and $m_R(t)$, in the P-R and Q-R conflicts of the matrix multiply program.

Recall that we only need to consider CF $(1, j)$ for the cross conflict misses of Q in Q-R conflict and those of P in P-R conflict. In addition, since P and Q are not reused in the coverage loop, their misses in a track can be determined by the track itself, without considering the prior track. We first consider $m_P(t)$. From Section A.1.2 we know that in a track of CF $(1, j)$ for (P, R), the p_1 elements must appear in an interleaved fashion with a sequence of r elements, *i.e.* two p_1 elements with no p_1 between them in the track are separated by exactly one r element. From Lemma 3.3.1, we know that no p_1 after the first can be a cache miss unless $D = 1$. In fact, if $D = 1$, every p_1 in a track is a cache miss. Since the first miss of p_1 in a track is counted as a loop cold start miss, $m_P(t) = h - 1$, where h is the number of p_1 elements in the track, which can be counted as in Section A.2 using a track tree like the one in Figure A.2a. If $D \geq 2$, then $m_P(t) = 0$.

Similarly, Section A.1.1 has shown that in a track of CF $(1, j)$ with respect to (Q, R), two q_1 elements with no q_1 between them in a track can be separated by one or two r elements. Therefore, according to Lemma 3.3.1, the second q_1 can be a miss only if $D = 1$ or $D = 2$. As with $m_P(t)$, if $D = 1$, then $m_Q(t) = h - 1$ for a track t of h q_1 elements, where h can be counted as in Section A.2 using a track tree like the one in Figure A.1a. On the other hand, if $D = 2$, then the only subtrack that the second q_1 is a miss is $q_1 r_i r_{i+1} q_1$. Each such subtrack will cause one cross conflict miss of Q. Since a such subtrack corresponds to a 101 sequence of edges in a path of a track tree like the one in Figure A.1a, we can search for the number of such patterns in each path to get $m_Q(t)$. If $D \geq 3$, then $m_Q(t) = 0$.

Finally, we consider $m_R(t)$. Since the cross conflict misses of R can occur in conflicts between P and R and between Q and R, we have to compute $m_R(t)$ for each case. In either case, the only possible conflict footprints, CF (i, j), have $i = 0$ or 1. Furthermore if $j < D$ then all R references are hits, and if $j > D$ then all R references are self conflict misses. Thus we only need to consider CF $(0, D)$ and CF $(1, D)$. Since array R is reused in the coverage loop, its misses in a track need to be determined by considering both the track and the track immediately before it. Using Equation (3.13), and summing over $t \in T(0, D) \cup T(1, D)$, we obtain

$$m_R(t) = \sum_{i=0}^{1} \sum_{t' \in T(i,D)} m_R(t|t') \frac{n_s(i, D)}{H_R(D)} \frac{n(t')}{\sum_{t_k \in T(i,D)} n(t_k)}$$

where $m_R(t|t')$ is the number of cross conflict misses of R for a track t when track t' is immediately before it. We have already shown how to compute $T(i, D)$ and $n_s(i, D)$, as well as $n(t)\ \forall\ t \in T(i, D)$, for $i = 0$ and 1. The rest of this section is thus devoted to discussing the method for computing the remaining unknown, $m_R(t|t')$.

For each track $t \in T(0, D) \cup T(1, D)$, we can certainly compute $m_R(t|t')$ for each $t' \in T(0, D) \cup T(1, D)$ by simply applying Lemma 3.3.1 for every t, t' pair. However, for the case here, we can derive a simple formula that computes $m_R(t|t')$ as follows:

$$m_R(t|t') = \min(D, D - f + l) \qquad \text{(A.3)}$$

where

$$
\begin{aligned}
f &= \begin{cases} \text{the first } p_1 \text{ (or } q_1\text{) position in track } t & \text{if } t \in T(1,D) \\ D & \text{if } t \in T(0,D) \end{cases} \\
l &= \begin{cases} \text{the last } p_1 \text{ (or } q_1\text{) position in track } t' & \text{if } t' \in T(1,D) \\ 0 & \text{if } t' \in T(0,D) \end{cases}
\end{aligned}
$$

We say that a p_1 (or q_1) is in position i of a track if it appears immediately after the ith r; it is in position 0 if it appears before the first r. Note that a track of CF $(0, D)$ or CF $(1, D)$ contains exactly D r elements, and these r elements appear in the same order in all tracks associated with these CFs. If both tracks t and t' contain only r elements, *i.e.* $t, t' \in T(0, D)$, then any r element in t is D elements away from the same r element in t', and thus no cache misses occur. However, if there are any p_1 (or q_1) elements between these two same r elements, then the r element in track t becomes at least $D + 1$ elements away from its previous appearance in t', and thus is a cache miss according to Lemma 3.3.1. Therefore, if in track t the first p_1 (or q_1) appears at position f (let $f = D$ if $t \in T(0, D)$), then each r element in t after position f must be a cache miss since it is at least $D + 1$ elements away from its previous appearance in t'. Similarly, if in track t' the last p_1 (or q_1) in t' appears at position l (let $l = 0$ if $t' \in T(0, D)$), then each r element before position l in t must also be a cache miss. The former case will result in a total of $D - f$ misses of R, while the latter case will result in a total of l misses of R. If $l \leq f$, then the total number of misses of R equals $(D - f) + l$. However, if $l > f$, then some r elements may be counted as misses in $(D - f)$ and counted again in l. In this case, the number of misses of R equals D. As a result, we have $m_R(t|t') = \min(D, D - f + l)$.

For example, let $D = 4$, $t' = p_1 r_1 p_1 r_2 r_3 r_4$, and $t = r_1 r_2 r_3 p_1 r_4$. We can find two cache misses of R in track t, namely, r_1 and r_4. In this example, since $f = 3$ and $l = 1$, we get $m_R(t|t') = \min(D, D - f + l) = \min(4, 4 - 3 + 1) = 2$ from Equation (A.3), as expected. The values of f and l for each track can also be obtained by searching each path of a track tree.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Aaron J. Goldberg and John L. Hennessy. Mtool: An integrated system for performance debugging shared memory multiprocessor applications. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):28–40, January 1993.

[2] Alvin R. Lebeck and David A. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer*, 27(10):15–26, October 1994.

[3] Margaret Martonosi, Anoop Gupta, and Thomas E. Anderson. Tuning memory performance of sequential and parallel programs. *IEEE Computer*, 28(4):32–40, April 1995.

[4] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Mateo, California, 1990.

[5] Rabin A. Sugumar and Santosh G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 24–35, Santa Clara, California, May 1993.

[6] Mark Donald Hill. *Aspects of Cache Memory and Instruction Buffer Performance.* PhD thesis, University of California, Berkeley, November 1987. UCB/CSD 87/381.

[7] Tien-Fu Chen. *Data Prefetching for High-Performance Processors.* PhD thesis, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, July 1993.

[8] Kyle Gallivan, William Jalby, Ulrike Meier, and Ahmed Sameh. The impact of hierarchical memory systmes on linear algebra algorithm design. Technical report, Center for Supercomputing Research & Development, University of Illinois, September 1987. CSRD Rpt. No. 625.

[9] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* Research Monographs in Parallel and Distributed computing. The MIT Press, Cambridge, Massachusetts, 1989.

[10] William H. Mangione-Smith, Santosh G. Abraham, and Edward S. Davidson. Architectural vs. delivered performance of the IBM RS/6000 and the Astronautics ZS-1. In *Proc. Twenty-Fourth Hawaii International Conference on System Sciences*, pages 397–408, January 1991.

[11] William H. Mangione-Smith, Tien-Pao Shih, Santosh G. Abraham, and Edward S. Davidson. Approaching a machine-application bound in delivered performance on scientific code. *Procedings of the IEEE: Special Issue on Computer Performance Evaluation*, 81(8):1166 – 1178, August 1993.

[12] Eric L. Boyd, Waqar Azeem, Hsien-Hsin Lee, Tien-Pao Shih, Shih-Hao Hung, and Edward S. Davidson. A hierarchical approach to modeling and improving the performance of scientific application on the KSR1. In *Proceedings of the 1994 International Conference on Parallel Processing*, volume 3, pages 188–192, St. Charles, Illinois, August 1994.

[13] Tien-Pao Shih and Edward S. Davidson. Grouping array layouts to reduce communication and improve locality of parallel programs. In *1994 International Conference on Parallel and Distributed Systems*, pages 558 – 566, Hsinchu, Taiwan, R.O.C., December 1994.

[14] William H. Mangione-Smith, Santosh G. Abraham, and Edward S. Davidson. A performance comparison of the IBM RS/6000 and the Astronautics ZS-1. *IEEE Computer*, 24(1):39–46, January 1991.

[15] Daniel Windheiser and William Jalby. Behavioral characterization of decoupled access/execute architectures. In *Proceedings of 1991 International Conference on Supercomputing*, pages 28–39, Cologne, Germany, June 1991.

[16] Joel S. Emer and Douglas W. Clark. A characterization of processor performance in the VAX-11/780. In *Proceedings of the 11th International Symposium on Computer Architecture*, pages 301–309, Ann Arbor, Michigan, June 1984.

[17] Jaswinder P. Singh, Wolf Dietrich-Webber, and Anoop Gupta. Splash: Stanford Parallel Applications for Shared-Memory. Technical Report 596, Stanford, April 1991.

[18] M. Berry, D. Chen, P. Koss, and D. Kuck. The Perfect Club Benchmarks: Effective performance evaluation of supercomputers. CSRD Report 827, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, May 1989.

[19] T. N. Mudge and H. B. Al-Sadoun. A Semi-Markov model for the performance of multiple-bus systems. *IEEE Trans. on Computers*, 34(10):934–942, October 1985.

[20] R. R. Oehler and R. D. Groves. IBM RISC System/6000 processor architecture. *IBM Journal of Research and Development*, 34(1):23–36, January 1990.

[21] Gregory F. Grohoski. Machine organization of the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34(1):37–58, January 1990.

[22] Kyle Gallivan, Dennis Gannon, William Jalby, Allen Malony, and Harry Wijshoff. Experimentally characterizing the behavior of multiprocessor memory systems: A case study. *IEEE Transactions on Software Engineering*, 16(2):216–223, February 1990.

[23] John R. Allen and Ken Kennedy. Automatic loop interchange. *ACM SIGPLAN Notices*, 19(6):233–246, June 1984. Proceedings of the *ACM SIGPLAN'84 Symposium on Compiler Construction.*

[24] Michael Wolfe. Advanced loop interchanging. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 536–543, August 1986.

[25] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* Addison-Wesley, New York, 1991.

[26] Harold S. Stone. *High-Performance Computer Architecture.* Addison-Wesley, New York, third edition, 1993.

[27] F. H. McMahon. The Livermore Fortran Kernels: A computer test of the numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, December 1986.

[28] Henry S. Warren Jr. Predicting execution time on the IBM RISC RS/6000. Technical Report IBM Research Report RC 16994(#75050), IBM, June 1991.

[29] Criss Stephens, Bryce Cogswell, John Heinlein, Gregory Palmer, and John P. Shen. Instruction level profiling and evaluation of the IBM RS/6000. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 180–189, Toronto, Canada, May 1991.

[30] Christine Eisenbeis, William Jalby, and Alain Lichnewsky. Squeezing more CPU performance out of a Cray-2 by vector block scheduling. In *Proc. of Supercomputing '88*, pages 237–246, November 1988.

[31] B. R. Rau, C. D. Glaeser, and R. L. Picard. Efficient code generation for horizontal architectures: Compiler techniques and architectural support. In *Proc. Int. Symp. on Computer Architecture*, pages 131–139, April 1982.

[32] R. F. Touzeau. A Fortran compiler for the FPS-164 scientific computer. In *Proc. ACM SIGPLAN '84 Symp. on Compiler Construction*, pages 48–57, June 1984.

[33] Ju-ho Tang, Edward S. Davidson, and Johau Tong. Polycyclic vector scheduling vs. chaining on 1-port vector supercomputers. In *Proc. of Supercomputing '88*, pages 122–129, November 1988.

[34] *AIX XL FORTRAN Compiler/6000 User's Guide Version 2.2*, second edition, September 1991. SC09-1354-01.

[35] C. Brian Hall and Kevin O'Brien. Performance characteristics of architectural features of the IBM RS/6000. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 303–309, Santa Clara, California, April 1991.

[36] William H. Mangione-Smith, Santosh G. Abraham, and Edward S. Davidson. Register requirements of pipelined processors. In *Proc. International Conference on Supercomputing*, pages 260–271, Washington D.C., USA, July 1992.

[37] William R. Hardell, Dwain A. Hicks, Lawrence C. Howell Jr., Warren E. Maule, Robert Montoye, and David P. Tuttle. Data cache and storage control units. In *IBM RISC System/6000 Technology*, pages 44–50. IBM Corporation, 1990.

[38] Gheith A. Abandah and Edward S. Davidson. Characterizing shared memory and communication performance: A case study of the Convex SPP-1000. Technical report, University of Michigan, 1996. CSE-TR-277-96.

[39] Dennis Gannon, Jenq kuen Lee, Bruce Shei, Sekhar Sarukkai, Srinivas Narayana, Neelakantan Sundaresan, Daya Atapattu, and Francois Bodin. SIGMA II: A tool kit for building parallelizing compilers and performance analysis systems. In *Proceedings of the IFIP WG 10.3 Workshop on Programming Environments for Parallel Computing*, pages 17–36, Edinburgh, Scotland, April 1992.

[40] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, April 1991.

[41] Olivier Temam, Elana D. Granston, and William 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*, pages 410–419, Portland, Oregon, November 1993.

[42] Olivier Temam, Christine Fricker, and William Jalby. Cache interference phenomena. In *Performance Evaluation Review special issue: 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, volume 22, pages 261–271, Nashville, Tennessee, May 1994.

[43] Eric Logan Boyd. *Performance Evaluation and Improvement of Parallel Applications on High Performance Architectures.* PhD thesis, Department of Electrical Engineering and Computer Science, The University of Michigan, 1995.

[44] Alan Jay Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

[45] Mark D. Hill. A case of direct-mapped cache. *IEEE Computer*, pages 25–40, December 1988.

[46] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 364–373, Seattle, Washington, May 1990.

[47] Anant Agarwal and Steven D. Pudar. Column-associative caches: A technique for reducing the miss rate of direct-mapped caches. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 179–190, San Diego, California, May 1993.

[48] Scott McFarling. Cache replacement with dynamic exclusion. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 191–200, Gold Coast, Australia, May 1992.

[49] Steve Carr, Ken Kennedy, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. Technical Report COMP TR92-195, Rice University, November 1992.

[50] Ken Kennedy and Kathryn S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. Technical Report COMP TR92-189, Rice University, August 1992.

[51] Yoji Yamada, John Gyllenhall, Grant Haab, and Wen-Mei Hwu. Data relocation and prefetching for programs with large data sets. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 118–127, San Jose, California, November 1994.

[52] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, Atlanta, Georgia, June 1988.

[53] Ivan Niven, Herbert S. Zucherman, and Hugh L. Montgomery. *An Introduction to The Theory of Numbers.* John Wiley & Sons, Inc., 1991.

[54] William J. Leveque. *Elementary Theory of Numbers.* Dover Publications Inc., 1990.

[55] Utpal Banerjee. A theory of loop permutations. In David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, Research Monographs in Parallel and Distributed computing, pages 54–74. The MIT Press, Cambridge, Massachusetts, 1989. (Second Workshop on Language and Compilers for Parallel Computing, August, 1989, Urbana, Illinois.).

[56] Mary E. Mace. *Memory Storage Patterns in Parallel Processing.* Kluwer Academic, Boston, 1987.

[57] Mary E. Mace and Robert A. Wagner. Globally optimum selection of memory storage patterns. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 264–271, August 1985.

[58] J.M. Frailong, W. Jalby, and J. Lenfant. XOR-SCHEMES: A flexible data organization in parallel memories. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 276–283, August 1985.

[59] Kathleen Knobe, Joan D. Lukas, and Guy L. Steele. Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, February 1990.

[60] Jingke Li and Marina Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, July 1991.

[61] J. Ramanumjam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–482, August 1991.

[62] Manish Gupta and Prithviraj Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[63] Sesh Venugopal and William Eventoff. Automatic transformation of FORTRAN loops to reduce cache conflicts. In *Proceedings of 1991 International Conference on Supercomputing*, pages 183–193, Cologne, Germany, June 1991.

[64] Michael E. Wolf and Monica S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[65] Daniel Windheiser. *Data Locality Optimization.* PhD thesis, IRISA, University of Rennes, 1992.

[66] David E. Hudak and Santosh G. Abraham. Compile-time optimization of near-neighbor communication for scalable shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 15:368–381, 1992.

[67] Karen A. Tomko and Santosh G. Abraham. Iteration partitioning for resolving stride conflicts on cache-coherent multiprocessors. In *Proceedings of the 1993 International Conference on Parallel Processing*, volume 2, pages 95–102, August 1993.

[68] Yeun-Jyr Ju. *Compiler Data Layout and Code Transformation for Reducing Cache Coherence Overhead.* PhD thesis, Purdue University, December 1991.

[69] E.F. Codd. Further normalization of the data base relational model. In Randall Rustin, editor, *Courant Computer Science Symposium 6, Data Base Systems*, pages 33–64. Prentice-Hall, May 1972.

[70] James Martin. *Computer Data-Base Organization.* Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

[71] Lung-Yu Chang and Henry G. Dietz. Data layout optimization and code transformation for paged memory systems. Technical Report TR-EE 90-43, Purdue University, April 1990.

[72] Kendall Square Research Corporation, 170 Tracer Lane, Waltham, MA 02154-1379, USA. *KSR1 Principles of Operation*, October 1992. Rev. 6.0.

[73] Erik Hagersten, Anders Landin, and Seif Haridi. DDM - a cache-only memory architecture. *IEEE Computer*, 25(9):44–54, September 1992.

[74] Per Stenstrom, Truman Joe, and Anoop Gupta. Comparative performance evaluation of cache-coherent NUMA and COMA architectures. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 80–91, Gold Coast, Australia, May 1992.

[75] Daniel Windheiser, Eric L. Boyd, Eric Hao, Santosh G. Abraham, and Edward S. Davidson. KSR1 multiprocessor: Analysis of latency hiding techniques in a sparse solver. In *the 7th International Parallel Processing Symposium*, pages 454–461, April 1993.

[76] Eric L. Boyd and Edward S. Davidson. Communication in the KSR1 MPP: Performance evaluation using synthetic workload experiments. In *8th ACM International Conference on Supercomputing*, pages 166–175, July 1994.

[77] David J. Kuck. *The Structure of Computers and Computations, vol. 1.* John Wiley & Sons, New York, 1978.

[78] W. Abu-Sufah, D. Kuck, and D. Lawrie. Automatic program transformations for virtual memory computers. In *Proceedings of the National Computer Conference*, volume 48, pages 969–974. AFIPS Press, 1979.

[79] L.-C. Lu and M. Chen. Parallelizing loops with indirect array references or pointers. In *Proceedings of The Fourth International Workshop on Languages and Compilers for Parallel Computing*, pages 201–217, Santa Clara, California, August 1991.