Empirical Performance Modeling of Multiprocessors Based on Data-Sharing Analysis

Kangwoo Lee

CENG 97-24

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213) 740-4475

December 1997

EMPIRICAL PERFORMANCE MODELING OF MULTIPROCESSORS BASED ON DATA-SHARING ANALYSIS

by

Kangwoo Lee

A Dissertation Presented to the

FACULTY OF THE GRADUATE SCHOOL

UNIVERSITY OF SOUTHERN CALIFORNIA

In Partial Fulfillment of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

(ELECTRICAL ENGINEERING)

December, 1997

Dedicated to my family...

Acknowledgments

This work marks the completion of my Ph. D. degree at USC. I am immensely grateful for the opportunity of having received what I consider the finest education available. I would like to thank some of the people that have made this possible.

I would like to thank my advisor without whom none of this work would have been possible. I would like to thank Professor Michel Dubois for serving as my advisor. In addition to excellent technical advice, he showed me much about how to work with people. His early help in defining a research problem was especially critical. Additionally, he lent industrial insight to the work and gave much help with publications.

I am indebted to all those who helped in the preparation of this thesis, particularly Professor Peter Beerel, Professor J-L. Gaudiot, Professor Sandeep Gupta, and Professor Clifford Neuman, for their detailed and constructive criticism.

I would like to thank the members of our research group. Also, I would like to thank the current and the alumni members of Yonsei University at USC for showing me the great warmness. I would also like to thank all my friends here and in Korea.

For moral support, much of which was needed in the years completing this work, I would like to again thank my parents for their unfailing support. I would also like to thank my wife, daughter, and son who also contributed greatly to my general state of well being as my best friends for showing me the single minded determination to graduate that led to the completion of this work.

Contents

List of Figures

List of Tables

Abstract

In the development of large scale multiprocessor systems and parallel applications, it is critical to predict performance before physical implementation. Software simulations and analytical models are widely used for this purpose. However, software simulations are time-consuming and resource-intensive and it is difficult or even impossible to simulate realistic applications on large target systems. Furthermore, the predictive power of conventional analytical models is very limited. The major goal of this thesis is to explore a methodology to predict accurately the performance of very large applications executed on large-scale multiprocessors.

In this thesis an empirical modeling methodology is developed for shared-memory applications. In this methodology, a few samples of a performance metric are collected by simulating small problems on small system configurations. Additionally, a parametric model of the metric is found through static data-sharing analysis. We limit ourselves to scientific SPMD applications, in which data-sharing can be quantified by analyzing the array indexes of shared data structures. A statistical robust parameter estimation technique is then applied to estimate the parameters of the model. The outcome is an analytical model to predict the value of the metric. Although the ultimate measure of performance is the execution time, we focus on the cold and coherence misses, which are essential in parallel applications.

With this modeling technique, we have achieved quick and accurate performance predictions for problems and systems that are so large that they are difficult or even impossible to simulate. In most cases, the prediction error falls below 1%. This result demonstrates that prediction based on empirical models is extremely accurate even for applications whose behavioral characteristics are irregular.

Because the data-sharing analysis is applicable not only to the number of cache misses but also to other performance metrics such as the numbers of instructions and data accesses, it could be used to predict the execution time in many cases. With the help of a parallelizing compiler, we can envision that the whole modeling procedure can be part of an evaluation framework to develop large scale applications and architectures for high-performance scientific computing.

Keywords: Empirical Modeling, Performance Prediction, Sharing Analysis, scientific SPMD applications, Curve Fitting, Robust Parameter estimation, Shared-Memory Multiprocessors, Cache, Cache Misses

Chapter 1

INTRODUCTION

To evaluate high performance shared-memory multiprocessor systems, researchers and designers can compare designs through software simulations without building hardware prototypes. A set of potential application programs are ported and run on the simulator for diverse possible configurations of the architecture. Various types of performance metrics can be obtained. Unfortunately, software simulations are usually slow and consume large amounts of memory, which limits the problem sizes and target systems to be simulated. As the data set size and the number of processors [36, 56] grow, the amount of memory needed to store the code and the data for both the simulator and the application program may become prohibitive. Moreover, the simulation time often increases much faster than the data set size and the number of processors. This explains why simulation results are only reported on small data sets and small systems while researchers still want to see how their design performs for very large problems.

A tempting approach is to build analytical models and to employ them for driving software simulators. In the past, analytical models have been widely used. Synthetic, parameterized workloads mimic the behaviors of real programs so that the architectures can be compared. Analytical models are useful for several reasons. First, they can be used in quick performance evaluation to prune the design space. Second, analytical models help understand the workload and its characteristics when various parameters are changed. Finally, if extrapolation can be done reliably, the model is used to predict workload behavior and architecture performance for the cases that would be very difficult or even impossible to simulate.

Existing analytical models are structural models abstracted from the expected behavior of the workload and formalized with a set of statistically defined parameters. For example, in [24], a stochastic model was developed in which the workload is characterized by the probabilities of accessing and modifying a shared block. Whereas the model can drive an architecture simulation and show the effects of the parameter values, it is in general difficult to estimate the parameters of the model for a particular application, let alone to estimate the effects of variables such as the data set size or the number of processors in the underlying system. Thus a model is only valid for a particular application with a given data set size and a given number of processors.

Because simulation and general-purpose hardware have become so efficient, it is now possible to develop empirical models. In an empirical performance model, a few samples of performance metrics are collected by simulating a few small-size problems running on small systems. Additionally, the magnitude orders of those performance metrics are found from static workload analysis. Then, statistical extrapolation techniques produce an analytical model which can be applied to estimate the values of performance metrics for realistically large data set size and number of processors.

In this thesis, an empirical technique for modeling cache misses due to data sharing and interprocessor communication is introduced. Although the ultimate measure of performance is the execution time, cache misses have been at the center of attention because of their significant impacts on execution time. In a parallel system, a large number of cache misses stem from the data-sharing among processors as specified in program statements. These data-sharing misses can be estimated through the analysis of data-sharing in parallel applications.

Due to its importance in the performance evaluation of parallel computer systems, data-sharing analysis has been a popular research subject. Many researchers suggested their own analysis methodology and introduced diverse analytical performance models established on top of the knowledge they earned from the sharing analysis. The sharing analysis can be carried on from the traces collected from the actual execution of parallel programs, or directly from the parallel application codes. In general, trace-collection facilities are not so widely available and the traces are valid only for the hardware and software configurations from which they are collected. For these reasons, the accuracy and generality of data-sharing analysis is better when the application codes are used.

One of the most general application classes of parallel systems is scientific computations programmed in the Single-Program-Multiple-Data (SPMD) model. As the speed of today's computer systems gets ever faster, the amount of data to be processed is accordingly growing. These data objects are normally represented by array data structures and the computations on them are performed within iterative loops such as the `for` loop construct. The information regarding the memory locations of the data objects that a processor accesses is given in the program. That is to say, the index expressions of array variable bear the information regarding possible data-sharing in the program. Therefore, the data-sharing analysis must start with array index expression analysis.

The SUIF compiler [40, 73] is the product of the one of the state-of-the-art research in this field. The SUIF compiler statically analyzes the locality of data accesses in a sequential program and produces a parallel program that can be run with minimized amount of data-sharing or data communication among the processors. The locality analysis in the SUIF compiler is done on top of the index expression analysis. Their target application area is dense matrix computations. Indeed, the array index expressions in general scientific applications such as those in SPLASH [67] and SPLASH-2 [82] benchmarks are generally too complicated to perform array index expression analysis. This is why the researchers participating in the development of the SUIF compiler confine themselves to dense matrix computations.

The people in the High Performance Fortran Forum [42] perceived that static array index expression analysis is too complex unless the compiler is provided with useful information so that it can generate high performance parallel codes. To achieve this goal, they develop High Performance Fortran (HPF), a language in which added directives guide the compiler in decisions about some fundamental factors affecting the performance of a parallel program such as the degree of available parallelism, exploitation of data locality, and choice of appropriate task granularity. HPF directives appear as structured comments that suggest implementation strategies or assert facts about a program to the compiler.

In another active project called ALPSTONE [16, 45], a formal description of data access behavior in a parallel application is provided as an input to the data-sharing analyzer, instead of the actual program codes.

These projects are evidence that data-sharing analysis based on array index expression analysis is hard to make and a tempting approach is to provide the sharing analyzer with formally described sharing information. This information can be obtained in many ways. In HPF or ALPSTONE, the programmer or researcher must understand the behavioral characteristics of the underlying algorithms in their parallel applications beforehand. Based on this understanding, they can specify the sharing information according to the pre-defined syntax of the description tools.

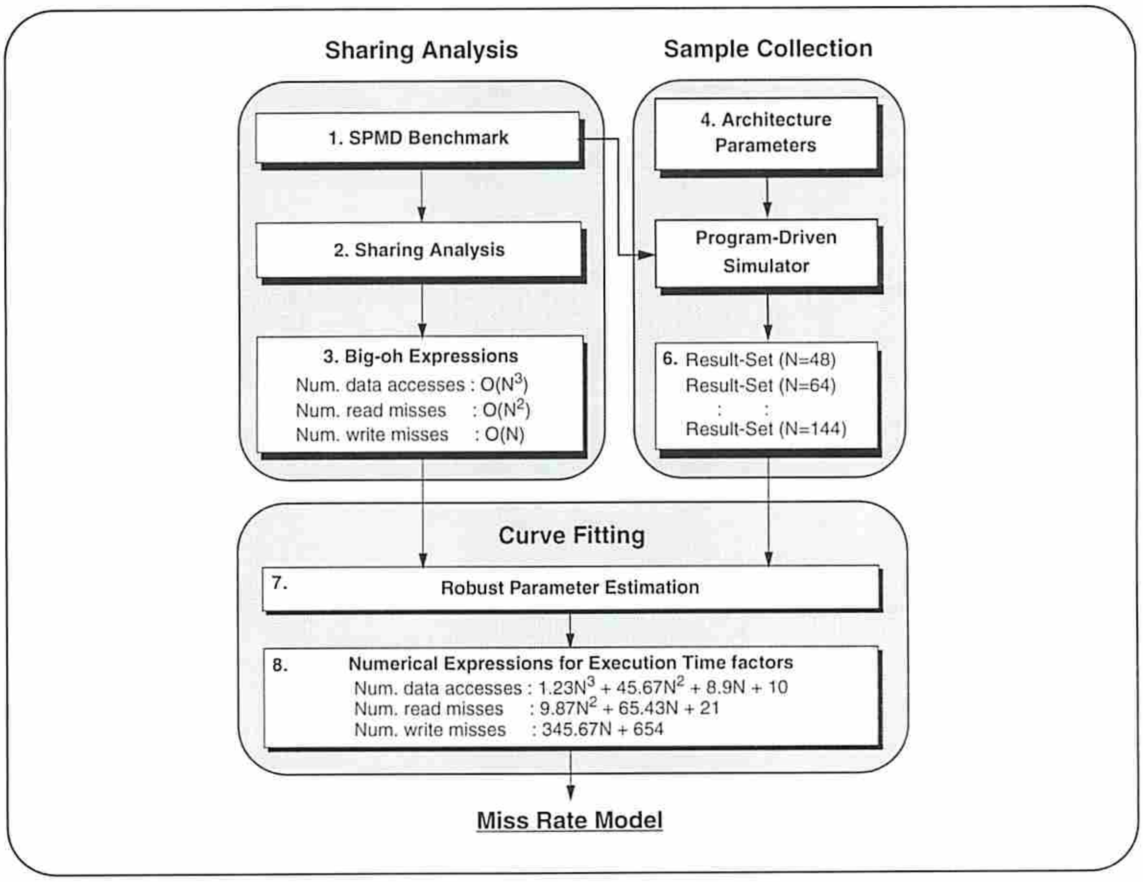
In this research, the data-sharing analysis does not need the knowledge regarding the sharing activities in an application. Instead, the translation of the source code into a formal description called the *Access Pattern Description Statement* (APDS) is merely necessary. The core of this research is to establish a systematic method with which the APDS can be easily constructed from the given parallel applications. The APDS can then be input to the analyzer to compute the magnitude order of the number of data references and the number of cache misses in the big-oh notation. The outputs of the sharing analyzer are used in the curve fitting stage together with previously collected simulation samples to find the a numerical expression for the best fit curve, where two independent variables are the data set size and the number of processors. The final results are the empirical models for the miss rates and, as will be seen later, they are extremely accurate.

To establish a method to translate the source code into an APDS, we should first characterize the data-sharing patterns for various types of array index expressions that are found in general scientific parallel applications. First of all, data-sharing is not observed at every moment during the execution of a parallel program. Rather, it takes place only when certain necessary conditions are met. Therefore, we first investigated what are these conditions and which types of factors are associated with them. In practice, the number of possible values that the sharing factors can take is so large that they need to be *categorized* into a finite number of groups. Although fine grouping may increase the accuracy of sharing analysis, complicated treatments of minute differences among the categories often degrade the efficiency of the work. Our policy is to keep the categories as coarse as possible while a certain level of analysis accuracy is maintained. The next step is to map the array index expressions in application codes into the *groups* that we have defined.

Generally, the overall quality of an analysis and modeling technique is determined by comparing the resulting analytical models against the experimental results. In addition, the prediction accuracy for very large data sets and processor numbers is also important in judging the usefulness of analytical models. Once the model is shown to be accurate, the whole technique including the definition of sharing factors, grouping their values into categories, capturing the values of sharing factors from applications and formularizing the factors into a numeric expression is appraised to be accurate, as well. In this regard, the extreme accuracy of our empirical models for miss rates based on sharing analysis is evidence to the high quality of the modeling technique introduced in this thesis.

1.1 Overview

Figure 1 shows the overall empirical performance modeling procedure with respect to the data set size N. It is largely composed of three subprocedures: sharing analysis, sample collection, and curve fitting. The first component is to establish polynomial expressions (models) for the magnitude order of the number of cache misses in the *big-oh* notation (Figure 1, boxes 1, 2, 3). This step is the core of the work. Performance data are then collected by simulating the execution of the program with small data set (boxes 4, 5, 6). Then, the robust parameter estimation technique (box 7) is used to find the best fit numerical expression (models) for each performance metric (box 8).



Sharing Analysis
Sample Collection
1. SPMD Benchmark
2. Sharing Analysis
3. Big-oh Expressions
Num. data accesses : O(N^3)
Num. read misses : O(N^2)
Num. write misses : O(N)
4. Architecture Parameters
Program-Driven Simulator
6. Result-Set (N=48)
Result-Set (N=64)
Result-Set (N=144)
Curve Fitting
7. Robust Parameter Estimation
8. Numerical Expressions for Execution Time factors
Num. data accesses : 1.23N^3 + 45.67N^2 + 8.9N + 10
Num. read misses : 9.87N^2 + 65.43N + 21
Num. write misses : 345.67N + 654
Miss Rate Model

Figure 1. Overview: miss rate prediction system.

Since our target is the miss rate, the numbers of data references and cache misses are first modeled in terms of data set size and number of processors. Prediction is performed by substituting a large value of data set size into the model obtained in boxes 7 and 8. The same can be done with P, the number of processors.

1.2 Scope of the Thesis

In shared-memory multiprocessor systems, the mechanisms to maintain data consistency are broadly classified into write-invalidate protocols and write-update protocols [74]. In this thesis, among many possible protocols, we only consider a three state write-invalidate protocol [74]. However, since the source of all coherency events in as protocol is commonly the data sharing which is inherent in applications, the methodology introduced in this thesis can be applied to other protocols.

In addition, among parallel applications, we consider scientific SPMD programs since they are the most popular primarily due to the ease of program coding. The benchmark applications used in our research are from the SPLASH [67] and SPLASH-2 [82] benchmark suites.

In the simulation, the caches have infinite size. This helps fully understand the data-sharing effects by eliminating the replacement misses.

1.3 Organization of the Thesis

This thesis is organized as follows.

Chapter 2 provides the basic background knowledge for the work in this thesis. A set of issues that often characterize parallel programs are briefly introduced. The issues include data partitioning, task scheduling, and synchronization. In addition, we extract the essential features of scientific SPMD programs. These features of SPMD programs are at the foundation of our research.

Recall that the empirical models to be built in this thesis will be able to predict the miss rates for very large data sets and number of processors. That is to say, the only two independent variables in the model are the data set size and the number of processors, and the number of data references and the number of cache misses should be dependent on them. A simple, but important, illustration that the number of cache misses is proportional to the two independent variables in the model is given in Chapter 2. At the end of this chapter is found the mathematical presentation of the curve fitting technique is given.

At the beginning stage of the work, we needed a sound evidence that miss rates vary along with the data set size of an application. Chapter 3 presents our very first models. These miss rate models were established based on a simple algorithmic complexity analysis idea instead of the sharing analysis. In this chapter, a small table that collectively shows the empirical models and predictions results of the benchmarks is given. The results for individual applications are put into the Appendix A.

Encouraged by the accurate prediction results of the model in Chapter 3, we started the actual research, *i. e.*, the construction of empirical models for cache miss rates based on the sharing analysis. When investigating the data-sharing patterns in Chapter 4, we do not focus on the cache misses, yet. First, the sharing factors that affect the amount of data-sharing are defined. Once we understand the sharing factors and the ways they affect data-sharing, the array index expression analysis (or, data-sharing analysis) is carried on. The components used in array index expressions and the composite index expressions composed of the index expression components are enumerated. The data-sharing patterns for all types of index expressions are also provided. At this moment, the effects of the data set size and the number of processors on the amount of data-sharing are discussed.

In Chapter 5, the models for the number of data references and the number of cache misses are built based on the knowledge we obtained in Chapter 4. The ideas about the data-sharing patterns are briefly summarized to apply them to measuring cache misses. Since there are usually many shared arrays whose behavioral characteristics are distinct from each other, individual arrays are dealt with separately. A miss rate model is established for each shared array. The global miss rate model is then built by summing up those models. At the end of the chapter, formal algorithms to count the number of data references and the number of cache misses are provided.

The prediction results of the empirical models built in Chapter 5 are tabulated in Chapter 6. For each application, two tables are shown; one for the prediction for large data sets and the other for large number of processors. The full descriptions of each application, their behavioral characteristics and APDS, and the simulation results for various data sets are all summarized in this chapter. Finally, the improvements of the prediction accuracy of the models established based on the sharing analysis over the crude models built in Chapter 3 are tabulated.

The concluding remarks of this thesis are given in Chapter 7 and the related work of other researchers follows in Chapter 8.

Chapter 2

BACKGROUND

2.1 Parallel Programming

We first summarize fundamental issues in parallel programming and relevant characteristics of scientific SPMD programs. Generally, parallel programs involve:

- partitioning of the computation into tasks,
- distribution of tasks among processes,
- coordination of data accesses and communication, and
- assignment of processes to processors.

2.1.1 Data Partitioning and Distribution

Data partitioning and distribution are usually independent of the underlying architecture. They determine how the work is broken up among cooperating processes. In many classes of applications, especially in scientific applications, the partitioning of the data and of the task are so strongly related that it is unnecessary to distinguish them. Therefore, we regard them as equivalent and focus on data partitioning.

The data partitions may or may not be of equal size. We refer to these two cases as *uniform* or *nonuniform* partitioning, respectively. Uniform partitioning is preferred when data partitioning is based on the memory locations of data (*location-dependent partitioning*) due to several reasons. First, it is simple to devise an algorithm and to write a program. Second, performance analysis and prediction are easy. Finally, one can achieve well-balanced workloads. Data partitioning may also be based on data values (*value-dependent* partitioning). This approach is usually taken when the program behavior is dependent on the values of data and the data partitions may be either uniform or nonuniform. Unlike location-dependent partitioning, we cannot statically understand or predict the runtime behavior of applications especially when data are nonuniformly partitioned. Therefore, applications with nonuniform value-dependent data partitioning are not considered in our research.

In parallel programs, shared data structures are partitioned and distributed among processors. We define the processor associated to a particular data to be the *Home* of the data. The home is of great interest since most accesses to a data are made by home because data partition is intended to maintain the locality of accesses and to reduce communication overheads. Additional accesses to the data may be made by other processors between two consecutive access runs by the Home.

Data structures are typically partitioned in block, cyclic, block-cyclic or random manner. Assume P processes and an N-element shared array which is partitioned into b-element blocks. In block partitioning, an array is broken into P pieces of equal size ($b=N/P$). The block size varies with the data set size and the number of processes. In cyclic partitioning, each element is considered as a block ($b=1$). In block-cyclic methods, $bk=N/P$ where k is an arbitrary positive integer so that k blocks are allocated to the same process in a particular manner such as round-robin. In cyclic and block-cyclic partitioning, the block size is fixed regardless of data set size or number of processes. Finally, the array can also be randomly partitioned into blocks of arbitrary size which varies in an unspecified manner as the data set size and the number of processes change.

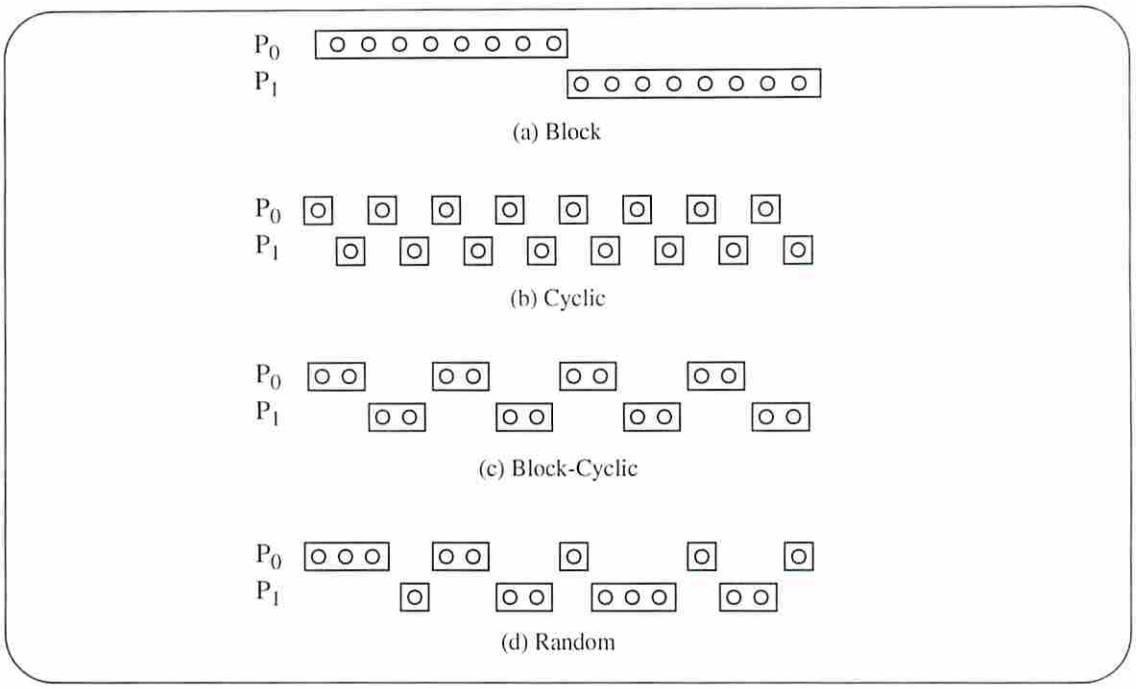


Figure 2. Data partitioning schemes.

2.1.2 Process Coordination and Task Assignment

The purpose of the coordination and assignment is to use available mechanisms to accomplish the following goals correctly and efficiently.

2.1.2.1 Naming and Accessing Shared Data

The fundamental issues in naming are: which shared data can be addressed at the hardware or user level, how they are addressed, and which operations are provided to access them. In *distributed address space* systems, the data in each processing element are independently managed so that remote processing elements cannot directly access them. In *shared address space* systems, all processes are able to access any shared data location with a single memory operation. A shared address space means that, when an address is generated by a processor, the hardware will access the specified memory location without additional processor intervention regardless of where the data is located in the system. In my research, only shared address space systems are considered.

2.1.2.2 Data Communication

Communication among processes is done by either *massage passing* or *shared-memory accesses*. In the message-passing paradigm, the sending and receiving of messages are implemented by specific primitives which are the basis of orchestrating individual activities. Shared memory systems employ conventional memory operations to provide data communication through shared addresses as well as special atomic operations such as lock or test-and-set. The communication model adopted in our study is the shared memory.

2.1.2.3 Interprocessor Synchronization and Execution Ordering

In message passing, a synchronization event is implicitly associated with the transmission or arrival of a message. At the hardware level, an event causes either a program on a processor or a state machine controller to take some action. At the user level, returning from the send call implicitly conveys synchronization information as does returning from the receive call. On the other hand, in shared address spaces, additional operations using synchronization primitives such as locks are required to enforce mutual exclusion. In addition, other primitives such as pauses and barriers are also used for correct execution ordering and interprocessor synchronization.

2.1.2.4 Concurrent Task Scheduling

In the management of parallelism, load balancing can be achieved by a *static* or *dynamic* assignment of concurrent tasks to processors. A *static* assignment is typically an algorithmic matter where the allocation of tasks to processors depends on the data set size, the number of processors and data partitioning. In many SPMD applications, the number of processes and the number of processors are identical and a particular process is tied to a specific processor. Static techniques do not cause much task management overhead. However, for the sake of good load balance, the work in each task must be predictable.

Dynamic techniques are categorized into two classes. In *semi-static* techniques the assignment is determined algorithmically before a computation phase but assignments are recomputed periodically for better load balance. The task granularity in semi-static assignments is predictable between successive time-steps. As in static techniques, a particular process is assigned in each execution phase to a specified processor. *Dynamic* tasking handles the cases where the task granularity or the machine environment is unpredictable. In this approach, the program specifies a mechanism by which tasks are assigned to processors during the computation. Essentially, a pool of available tasks is maintained, and each process repeatedly takes a task from the pool and executes it until there are no tasks remaining.

To summarize, dynamic techniques are adopted for runtime load balancing when the work to be done by a processor is not predictable. While they generally provide good load balance, task management is expensive. Static techniques are therefore usually preferable when they can provide good load balance and are, thus, used in the majority of scientific applications. Also, semi-static techniques are very common. In our study, since the dynamic task assignment is too difficult to statically model or analyze, we focus on static and semi-static task assignment techniques. That is to say, the benchmarks used in our study can associate tasks with processors in a way that each process runs on exactly one and only one processor and there is no task migration.

2.1.3 Summary

The common characteristics of applications we use in the study are summarized below. In addition, the applications in SPLASH and SPLASH 2 benchmark suites are tabulated in Table 1 with their characteristics related to task partitioning and assignment.

- Programs are running on the shared address space system,
- number of processes is equal to that of processors,
- data communication is achieved through shared data,
- representative synchronization primitives are locks, pauses and barriers,
- tasks are assigned to processors in static or semi-static manner, and
- no task migration is allowed.

Benchmarks		Task partitioning			Task assignment
		Partition Size	Location- or Value-dependent	Block, Cyclic, Block-cyclic or Random	Static, Semi-Static or Dynamic
Used in the study	LU	Uniform	Location	BC, C	Static
	MP3D	Uniform	Location	BC	Static
	WATER	Uniform	Location	B	Static
	OCEAN	Uniform	Location	BC	Static
	FFT	Uniform	Location	B	Static
	BARNES-HUT	Uniform	Value	R	Semi-Static
	RADIX	Uniform	Location	B	Static
Cannot be Used	FMM	*Nonuniform*	Value	R	Semi-Static
	CHOLESKY	*Nonuniform*	Value	R	*Dynamic*
	LOCUSROUTE	*Nonuniform*	Value	R	*Static/Dynamic*
	PTHOR	*Nonuniform*	Value	R	*Static/Dynamic*
	RAYTRACE	Uniform	Location	B	*Static/Dynamic*
	RADIOSITY	*Nonuniform*	Value	R	*Static/Dynamic*
	VOLREND	Uniform	Location	B	*Static/Dynamic*

Table 1. Parallel application characteristics.

2.2 SPMD Programs

In scientific applications, computers are used to simulate physical phenomena that are usually impossible or very costly to observe through empirical means by discretizing continuous problems in both space and time into numerically approximated algorithms.

Discretized times are represented as *time-steps* in programs and discretized spaces form large regular grids or *arrays*. The computations executed on array elements are *uniform*, and the most popular program structure for uniform computations is *iterative loops* such as `for` loops.

Typically in SPMD programs, all processes are *symmetric* and execute a *common stream of instructions*. Therefore, the most suitable problems for SPMD style are the ones with large amounts of uniform computations which can be partitioned and executed by multiple processes, simultaneously. In consequence, many scientific applications nowadays are programmed in the SPMD style. Now, we summarize a set of observations regarding scientific SPMD programs.

Observation 1. (*Iterative program structure*) Discretized times in scientific applications are implemented as time-steps in programs.

Observation 2. (*Array data structure*) Discretized spaces in scientific applications are represented as array data structures.

Observation 3. (*Data homogeneity*) The elements in an array are homogeneous in the sense that the computations executed on them are uniform or identical.

Observation 4. (*Iterative computation loop*) Uniform computations on array elements are implemented by iterative loops. The iterative loop in SPMD programs mentioned in Observation 4 is equivalent to the FORALL loop [42].

Observation 5. (*Processor symmetry*) The processors are symmetric or identical and they execute a single instruction stream on their portion of partitioned array.

2.2.1 SPMD Programs on Shared-Memory Systems

Initially, a single process called a master process is started by the operating system. The master process performs initialization procedures required for parallel processing of operations in the program. Then, it creates worker processes, or slave processors, which directly enter the parallel section. The master process itself enters the parallel section so that all created processes execute the same code image until they exit from the program and terminate. This does not mean they proceed in lock-step manner or even execute the same instructions at a given moment since they may follow different control paths in the code.

Control over the shared data distribution and the assignment of work to processors are maintained by private variables that acquire distinct values for different processes. For instance, every process obtains a unique process identifier between 0 to the number of processors minus one upon the creation. This pid is used to determine which data partitions are assigned to which processes by calculating the array indices of each chunk of data. Such private variables are used as loop bounds to limit the address space of partitioned portion of a shared array. The code that performs the actual computations is essentially identical to that in the sequential program.

2.2.2 Data Communication

Data communication, or data-sharing, is of great significance since it causes performance degradation due to the coherency overhead for the correct execution of programs. This is why the reduction of data-sharing and its overhead has been one of the most important goals in parallel processing. To this goal, many efforts have been made to better analyze and model data-sharing.

In this section, we like to provide the first intuition that data-sharing is defined by the index expressions of shared arrays and the amount of data-sharing can be estimated from the values of array index components. In addition, we also like to show how the amount of data-sharing is dependent on both data set size and number of processors. Complete discussions will be made in Sections 4 and 5.

As pointed in Observation 4, the computations in SPMD programs are performed within FORALL loops. During the parallel execution, the loop control variables take the values bounded by the index range of array elements allocated to each processor. The array element designated by the loop variables is called the *center of computation*. Elements that participate in the computation are called the *participating data* or *participant(s)*[1]. In general, the indices of participants are expressed numerically using the loop control variables, constant numbers, or some other general variables. Then, the primary factors affecting data-sharing are:

- data set size and number of processors
- data partitioning method, and
- distance of participating data from the center of computation

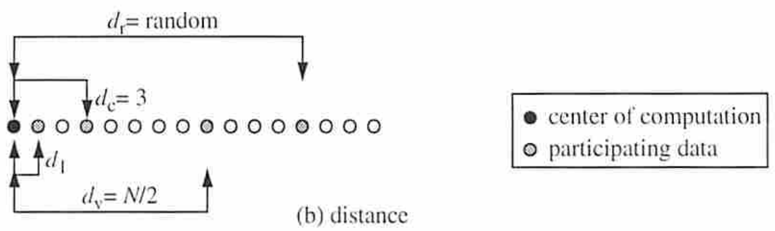
In addition, the number of participating data and the dimensions of shared array also influence the data-sharing as will be seen in Sections 4 and 5.

The distance between the center of computation and each participant is denoted by d and its possible values are d_1, d_c, d_v, and d_r, which stand for the unit distance (adjacent elements), constant distance, variable distance, and random distance respectively. *Variable* and *constant* are distinguished according to whether the distance is dependent on the data set size or the number of processors, or not. See the example program in Figure 3 (a) where an N-element array (N=16) is partitioned into blocks and distributed to P processors (lines 1, 2, 3). In each iteration of the `for` loop (line 4), the center of computation is `A[I]` (line 5). The distances between participants in lines 6 through 9 from the center of computation are shown in Figure 3 (b). Note that a variable `r` in line 8 and 12 is the one whose value can be only known at run time. The number of participants is similarly defined and denoted by n_1, n_c, n_v, or n_r (Figure 3 (a), lines 10 through 14 and Figure 3 (c)). Arrays can

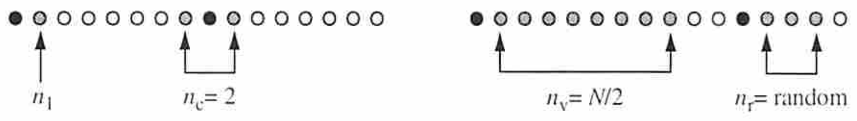
1. Formal definitions of the center of computations and participants will be made in Section 4.1.3

```
int A[N];                         /* array of N integers */
int first = N / P * my_pid;       /* block partitioning */
int last= N / P * (my_pid + 1);/* block partitioning */
FOR (I=first;I<last;I++) {
     ... A[I] ... ;               /* center of computation */
     ... A[I+1] ... ;                /* d = d_1 */
     ... A[I+3] ... ;                /* d = d_c = 3 */
     ... A[I+r] ... ;                /* d = d_r: r */
     ... A[I+N/2] ... ;              /* d = d_v = N/2 */
     ... A[I+1] ... ;                /* n = n_1 */
     ... A[I-1], A[I+1] ... ;        /* n = n_c = 3 */
     ... A[I+1] to A[I+r] ... ;      /* n = n_r: r */
     FOR (J=0;J<N/2;J++)
          ... A[I+J] ... ;           /* n = n_v = N/2 */
}
```

(a) Program



(b) distance



(c) number of associated data

Figure 3. Data-sharing parameters: distance and number of associated data.

be block (p_b), block-cyclic (p_{bc}), cyclic (p_c) or random (p_r) partitioned. Finally, the array may be one-, two-, or higher dimensional.

Following examples show how the factors mentioned above influence the amount of data-sharing. Among many combinations of possible cases, we present a few examples where one- and two-dimensional arrays are partitioned in p_b and p_{bc} manner with d=1 and, n_c=2 and n_c=4. A 16 element one-dimensional array and a 16x16 element two-dimensional array are partitioned between two processors into variable size blocks (p_b) and fixed size blocks (p_{bc}) in Figure 4 (a) and (b), respectively. In general, the data set size and the number of processors proportionally affect the amount of data-sharing. In Figure 4, we see that the amount of data-sharing does not change when the size of a block-partitioned one-dimensional array grows. It is because the block size varies with the data set size. The number of processors in p_{bc} does not affect the amount of data-sharing, either, since two adjacent processors share the same amount of data regardless of the number of processors. The shaded areas in Figure 4 (b) indicate the set of data shared by p_0 and p_1, and p_1 and p_2.

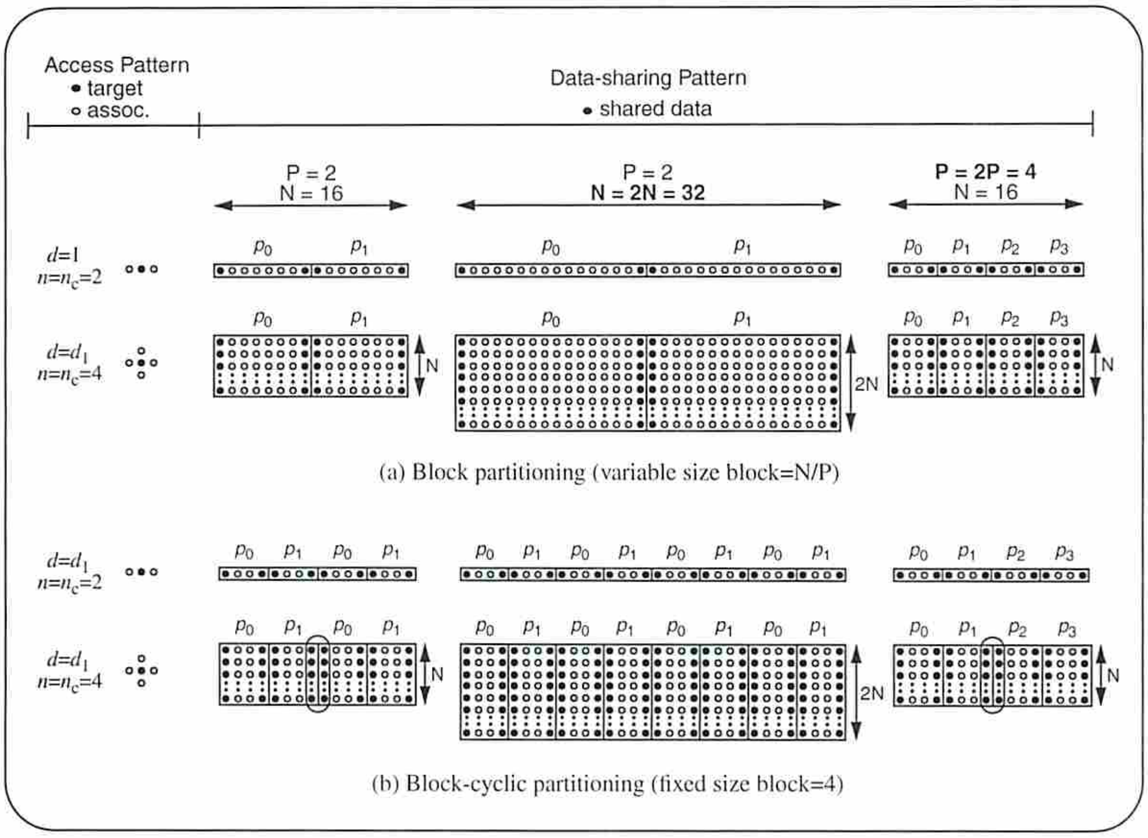


Figure 4. Effect of data set size and the number of processors on data-sharing.

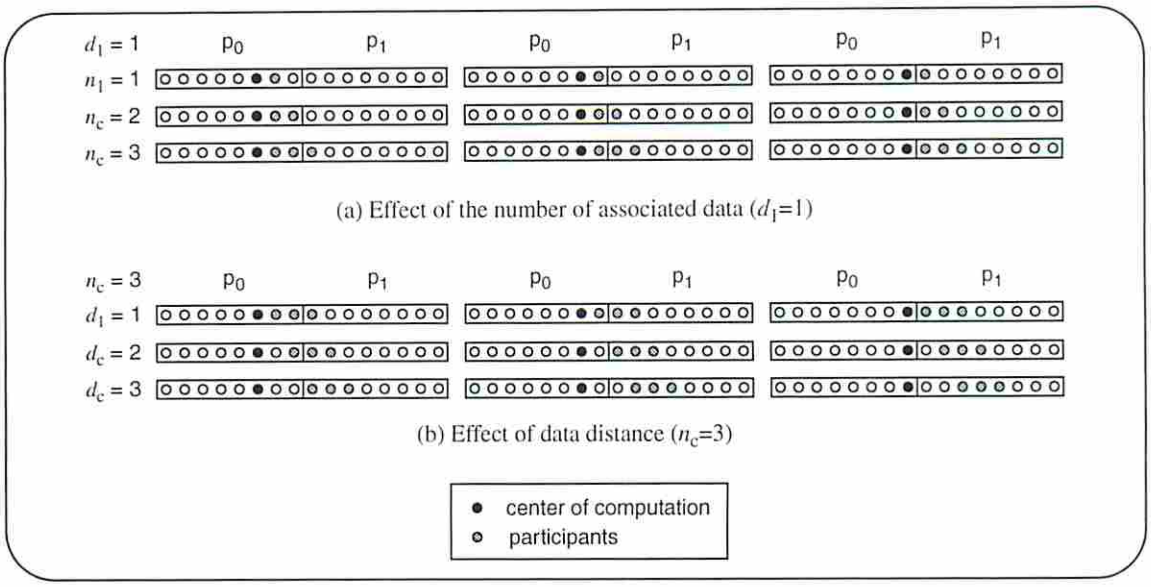


Figure 5. Effect of the number of participants and access distance.

In addition, a computation may apply to data elements of different arrays. There are two possibilities. First, the memory locations of an array and other arrays are *dependent*. That is, the indices of array elements in an array are given by functions of index variables which are also used for other array elements. In this case, both arrays normally have same size and are partitioned in the same manner. Second, the array indices of two arrays are *independent*. For example, memory locations of array elements of an array are the results of earlier computations while the elements of other arrays are specified by a loop control variable. Then, we regard the data location designated by earlier computation results as *random*. The `Cells` and `Ares` in MP3D, `ctab` and `ltab` in BARNES-HUT and `key` in RADIX are such arrays. Generally, these arrays cause large amount of data-sharing, especially when the size of secondary array is much smaller than that of main array (`Cells` and `ctab`).

In consequence, the amount of data-sharing varies along with the data set size, the number of processors or both as well as with the access distance of a participant, the number of participating data elements and the partitioning method.

2.2.3 Number of Iterations

The data-sharing analysis discussed so far deals with data sharing within an iteration. Another factor that affects the amount of data-sharing is the number of iterations or time-steps in an application. The total amount of data-sharing can be computed by multiplying the amount of data-sharing within an iteration by the number of time-steps.

2.3 Cache Misses

In cache-coherent shared-memory systems, some shared data accesses must be accompanied by data or signal transfers between the processor modules and memory modules. The occurrence of events that cause such transactions depends on the coherency protocol. We will use a bus-based three-state writeback invalidation protocol where possible cache block states are invalid, shared or dirty as in Figure 6. It is assumed that, when a processor executes a write operation, an invalidation signal is broadcast through the bus to all other processors so that only one signal suffices and no acknowledgment collection is required. Furthermore, when a processor has a dirty block copy, it must write the block back on receiving the invalidation signal.

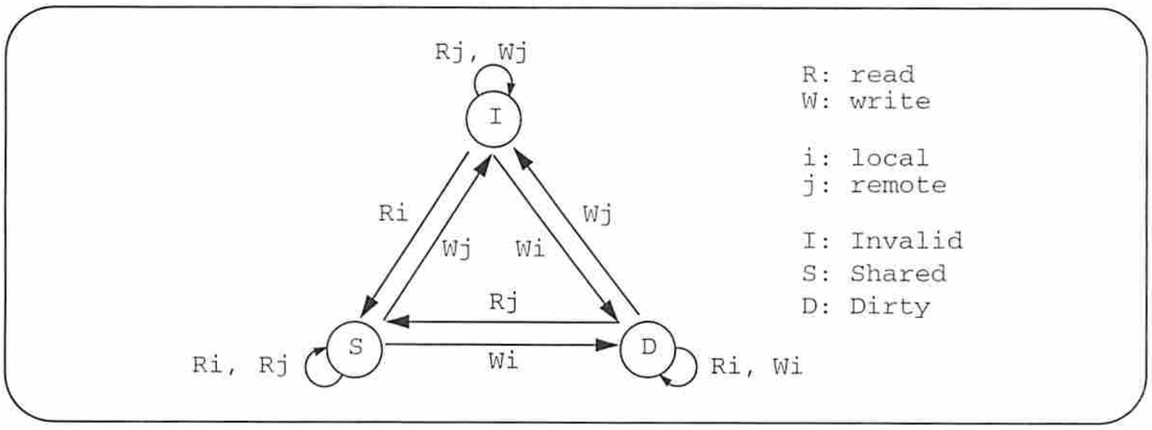
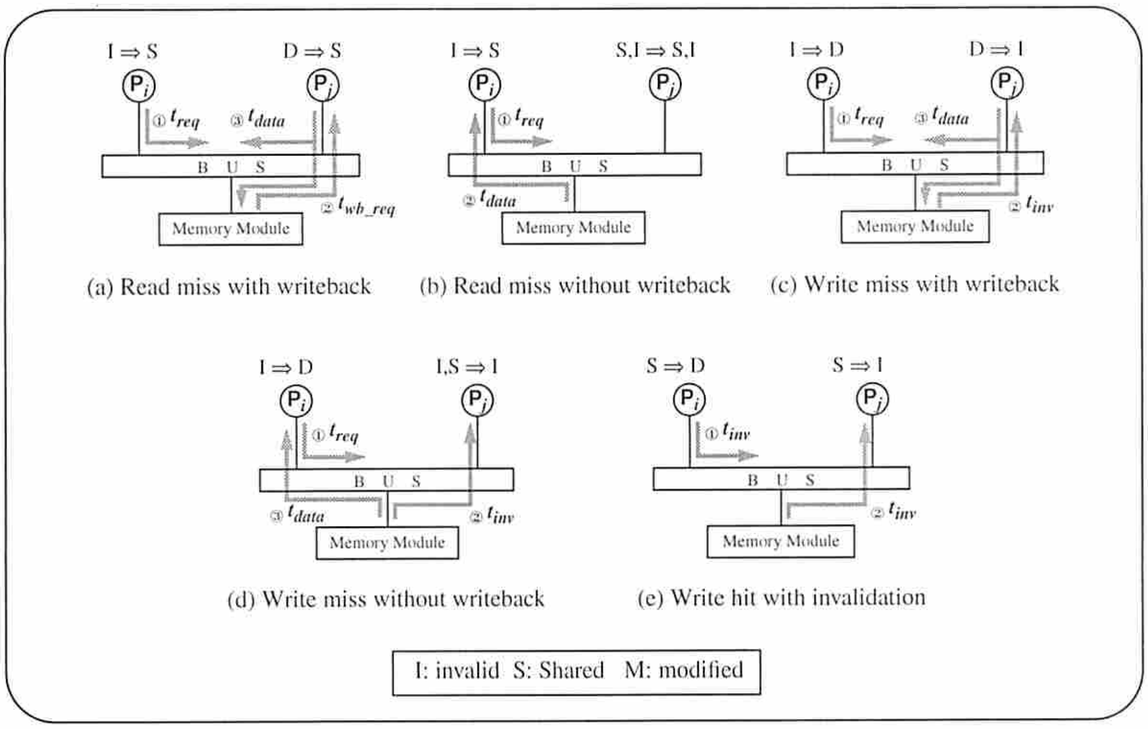


Figure 6. Bus-based 3-state write-back invalidation protocol and coherency events.

Figure 7 illustrates the coherency transactions performed in the three-state write-back invalidation protocol (Figure 6) to maintain data consistency. All coherency transactions are assumed to be initiated by P_i, and P_j is an arbitrary processor other than P_i. The circles denote processors and state changes of shared data block are placed over them. Shaded arrows denote the paths of data or coherence messages.



I ⇒ S
D ⇒ S
P_i
P_j
① t_{req}
③ t_{data}
② t_{wb_req}
B U S
Memory Module
(a) Read miss with writeback
I ⇒ S
S,I ⇒ S,I
② t_{data}
(b) Read miss without writeback
I ⇒ D
D ⇒ I
② t_{inv}
(c) Write miss with writeback
I,S ⇒ I
(d) Write miss without writeback
S ⇒ D
S ⇒ I
① t_{inv}
(e) Write hit with invalidation
I: invalid S: Shared M: modified

Figure 7. Coherency events in bus-based 3-state write-back invalidation protocol.

• **Read miss with writeback**: P_i experiences a read miss when P_j has a dirty copy in its cache. Then, P_i sends a request message for a copy of the block first. P_j must perform a writeback when it receives a signal so that a recent copy can be supplied to P_i. A copy of a block can be sent to P_i from either P_j or memory module. The former case shown in the figure.

• **Read miss without writeback**: P_i experiences a read miss when no processor has a dirty copy of the block. That is to say, other processors may have either a clean copy or none. In this case, a copy of the block must be provided from either processor with a clean copy or a memory module. The figure shows the latter case.

• **Write miss with writeback**: P_i experiences a write miss when P_j has a dirty copy of data. P_i sends a request and P_j should writeback its dirty copy of data when a signal arrives. Then, similarly to the case of Figure 7 (a), a copy of the block is eventually supplied by P_j.

• **Write miss without writeback**: P_i experiences a write miss when no cache has a dirty copy. Some processors may have a clean copy. The clean copies in other processors must be invalidated. Then a copy will be sent to P_i from either any one processor or the memory. Memory sends a copy in the figure.

• **Write hit with invalidation**: P_i experiences a write hit when P_j has a clean copy. This is the only case when a cache hit initiates a coherency transaction. The clean copies in other processors must be invalidated due to the write operation of P_i.

2.3.1 Cache Miss Classification

In a system with infinite caches, traditionally, cache misses are classified as *cold misses* or *invalidation misses*. A cold miss is the first miss to a block by a processor; invalidation misses are all other misses and are caused by invalidations. For precise performance analysis, researchers have built schemes to classify the misses [29, 76].

In our research, since we like to concentrate the effect of data-sharing without the effects of *replacement misses*, we assume infinite caches. The classification algorithm introduced by Dubois *et al.* [26] is used in this proposal. If a cold miss is preceded by an update from another processor, it is classified as a *cold true sharing miss* (CTSM) or as a *cold false sharing miss* (CFSM) depending on whether the updated value is communicated to processors during the lifetime of the block in the cache. *Pure cold misses* (PCM) are not preceded by an update from a different processor. Among invalidation misses we distinguish between *pure true sharing misses* (PTSM) and *pure false sharing misses* (PFSM). As opposed to a pure false sharing miss, a pure true sharing miss communicates a new data value to a processor.

The PCM, CTSM, CFSM and PTSM are called *essential misses* and their numbers cannot be reduced. The PFSM are called *useless misses* and they can be reduced by changing the hardware configuration of underlying systems or coherency protocols.

2.4 Curve Fitting

Generally, an *estimator* $y(x) = y(x;a_1,...,a_M)$ is a linear combination of any M specified functions of independent variable x and unknown coefficients a_j, where the general linear combination is

$$y(x) = a_1 + a_2x + ... + a_Mx^{M-1} = \sum_{k=1}^{M} a_kX_k(x) \qquad \text{(Eq. 1)}$$

where $X_k(x)$ are called the basis functions which can be nonlinear functions of x such as sines, cosines or exponential functions. The term "linear" only refers to the dependence of the model on its parameters, a_j. A procedure to find the coefficients called *parameter estimation* [47, 61] consists in fitting N data points (x_i, y_i), i=1,...,N, to an estimator with M adjustable parameters, a_j, j=1,..,M. Two types of estimators, least-square and robust estimators, will be used in my thesis.

2.4.1 Least-Square Estimators

Suppose the measurement errors for every observed data point y_i are independent and identically distributed random variables with a normal (Gaussian) distribution around the exact model $y(x)$. The probability (likelihood) of occurrence of the data points is the product of the probabilities of occurrence of every point [79]

$$P = \prod_{i=1}^{N}\left\{\exp\left[-\frac{1}{2}\left(\frac{y_i - y(x_i)}{\sigma_i}\right)^2\right]\Delta y\right\} \qquad \text{(Eq. 2)}$$

where N and Δy are constants. Thus, given N data points (x_i, y_i), i=1,...,N, to maximize the likelihood function we must find the parameter values that minimize $\sum_{i=1}^{N}\left[\frac{y_i - y(x;a_1,\ldots,a_M)}{\sigma_i}\right]^2$. Then, the solutions are given as [61]

$$a_j = \sum_{k=1}^{M}[\alpha]_{jk}^{-1}\beta_k \text{ with} \qquad \text{(Eq. 3)}$$

$$\alpha_{jk} = \sum_{i=1}^{N}\frac{X_j(x_i)X_k(x_i)}{\sigma_i^2} \quad \text{or equivalently} \quad [\alpha] = \mathbf{A}^T\cdot\mathbf{A}\text{, where } A_{ij} = \frac{X_j(x_i)}{\sigma_i} \qquad \text{(Eq. 4)}$$

$$\text{and } \beta_j = \sum_{i=1}^{N}\frac{y_i X_k(x_i)}{\sigma_i^2} \quad \text{or equivalently} \quad [\beta] = \mathbf{A}^T\cdot\mathbf{b}\text{, where } b_i = \frac{y_i}{\sigma_i} \qquad \text{(Eq. 5)}$$

where σ_i is the measurement error of the i^{th} data point. If not known, σ_i may be set to the constant value, for example, σ_i=1 [61].

2.4.2 Robust Estimators

Robust estimators are less sensitive than least-square estimators to departures from the idealized assumptions for which the estimator is optimized. Such departures are due to *outliers* which do not follow the general trend and thus distort the estimation. Instead of Eq. 2, the likelihood function for a robust estimator is given by [43]

$$P = \prod_{i=1}^{N}\{\exp[-\rho(y_i, y(x_i;a_1,\ldots,a_M))]\Delta y\} \qquad \text{(Eq. 6)}$$

where ρ is the negative logarithm of the probability density of measurement error. In case of local *M-estimator*, ρ is only dependent on the difference of measured y_i and predicted $y(x_i)$ when scaled by some weight factors σ_i for each point. Then, ρ becomes a function of a single variable $z \equiv (y_i - y(x_i))/\sigma_i$. Therefore, maximizing Eq. 6 is equivalent to minimizing ($\mathbf{a} = a_1,\ldots,a_M$)

$$\sum_{i=1}^{N} \rho\left(\frac{y_i - y(x_i;\mathbf{a})}{\sigma_i}\right) \qquad \text{(Eq. 7)}$$

over **a**. Solving Eq. 7 is finally equivalent to solving the M simultaneous equations [47]

$$\sum_{i=1}^{N} \psi\left(\frac{y_i - y(x_i;\mathbf{a})}{\sigma_i}\right) X_j(x_i) = 0, \qquad j = 1,2,\ldots,M \qquad ,\text{where } \psi = \frac{d\rho}{dz} \qquad \text{(Eq. 8)}$$

The solution, called the weighted least squares method [49], is iterative and starts with an initial estimate of **a** obtained by the least square method. In [43], we have $\mathbf{a}_2 = (\mathbf{A}^T\mathbf{W}\mathbf{A})^{-1}\mathbf{A}^T\mathbf{W}\mathbf{Y}$, where **A** is an NxN matrix with $A_{ij} = \frac{X_j(x_i)}{\sigma_i}$ and **Y** = $[y_i]$. **W** is a diagonal matrix of w_i defined as

$$w_i = \frac{(\psi(y_i - y(x_i;\mathbf{a})))/\sigma_i}{(y_i - y(x_i;\mathbf{a}))/\sigma_i} \qquad i = 1,2,\ldots,N \qquad \text{(Eq. 9)}$$

There are several possible choices for the weight functions. In this paper, we use Cauchy's function, so that $\rho(z) = \ln(1+z^2)$ and $\psi(z) = 2z/(1+z^2)$. When σ_i is unknown, it must be estimated. Among many suggestions [43], $\sigma_i = (\sqrt{(y_i - y(x_i;\mathbf{a}))^2})/N$ is used in this paper.

2.4.3 Computation Time

	Operation	Description	Multiply/Adds (Upper-bound)
Least Squares	$[\alpha] = A^T A$	[MxN] x [NxM]	$N \times M^2 < N^3$
	$[\beta] = A^T b$	[MxN] x [Nx1]	$N \times M < N^2$
Robust Estimation: $a_2 = (A^T W A)^{-1} A^T W Y$	$T_1 = A^T W$	[MxN] x [NxN]	$N^2 \times M < N^3$
	$T_2 = T_1 A$	[MxN] x [NxM]	$N \times M^2 < N^3$
	$T_3 = (T_2)^{-1}$	[MxM] inversion	$M^3 < N^3$
	$T_4 = T_3 A^T$	[MxM] x [MxN]	$N \times M^2 < N^3$
	$T_5 = T_4 W$	[MxN] x [NxN]	$N^2 \times M < N^3$
	$T_6 = T_5 Y$	[MxN] x [Nx1]	$N \times M < N^2$

Table 2. Computation cost of curve fitting (M<N).

Two fundamental computational algorithms in least-square and robust estimations are matrix inversion and matrix multiplication. For NxN matrix inversion, the Gauss-Jordan method [44] takes N^3 multiply/adds. The multiplication of two NxN matrices also takes N^3 multiply/adds and the multiplication of an NxN matrix with an Nx1 matrix takes N^2 multiply/adds.

Given N samples for M unknowns, as shown in Table 2, at most N^3+N^2 multiply/adds for the least squares and $5N^3+N^2$ multiply/adds for the robust estimation are required. The procedure for robust estimation is iterative and usually converges very quickly, within 100 iterations. Thus, its cost is less than $500N^3+200N^2$ multiply/adds. Since we have less than 10 samples (N=10) most of the time, the computation of the coefficients of a fitting curve is very efficient.

Chapter 3

EARLY EXPERIENCE: EMPIRICAL MODELS FOR MISS RATES WITHOUT SHARING ANALYSIS

At the early stage of this research, we first built models for the number of data references and for the number of cache misses. The miss rate model was obtained by dividing the number of cache misses by the number of data references. At this time, no sharing analysis was performed. Instead, simple algorithmic complexity analysis was adopted.

In complexity theory, the execution time of a sequential program is characterized by a function $f(N)$ where N is the input data set size and $f(N)$ is an expression in the big-oh notation. $f(N)$ is though of as *the number of instructions or the number of data accesses*. In other words, the number of instructions and the number of data references in a program vary according to the input data set size with a certain trend which is numerically expressed in terms of data set size. We call the data set size the *problem scaling variable.*

In case of parallel programs, the number of instructions and the number of data references of each processor depend on the number of processors, which is called the *system scaling variable*. Therefore, they are both expressed in terms of two *independent* scaling variables. Since the primary data in parallel programs are shared data structures, data accesses are ordinarily understood to mean shared data accesses. Of course, private variables are also indispensable and they are mainly loop variables of array index variables. In both cases, private data accesses are incident to shared data accesses. In consequence, what defines the number of shared data accesses is the algorithmic complexity of an application which is a function of input data set size expressed in the big-oh notation.

3.1 Modeling Data Accesses and Cache Misses

The cache misses we modeled in this thesis are the ones in Section 2.3.1.

3.1.1 Number of Data Accesses

As mentioned above, *the number of instructions* or *the number of data accesses* is given by a function $f(N)$ where N is the input data set size and $f(N)$ is expressed in the big-oh notation.

3.1.2 Number of Cold Misses

At the initial stage of my research, PCM, CTSM and CFSM defined in Section 2.3.1 are combined into *cold misses* (CM). The estimation of the number of cold misses starts with trivial assumptions.

Assumption 1: Every data element in shared data structures of an application is accessed at least once by at least one processor during the execution time.

Assumption 2: The number of processors (P) and cache block size (B) of underlying architecture model are constants.

Then, the *lower-bound* and *upper-bound for the number of cold misses* of an application program with $O(h(N))$ shared data elements running on a P processor shared-memory multiprocessor system with B byte cache blocks are both $O(h(N))$, where N is a variable that defines the data set size and $h(N)$ is the actual number of shared data elements in an application[1]. The reasons are as follow. First, the smallest number of cold misses is achieved when every cache block is accessed by only one processor during the entire execution. Then, the number of cache misses is $\lceil h(N) / B \rceil$ which is $O(h(N))$. Similarly, the largest number of cold misses is achieved when every cache block is accessed by all P processors during the entire execution. Then, the number of cache misses is $\lceil h(N) / B * P \rceil$ which is $O(h(N))$, too. Therefore, having identical expressions of both lower- and upper-bound, we conclude the number of cold misses in an application is expressed as $O(h(\mathrm{N}))$ when the assumptions are held.

3.1.3 Number of Sharing Misses

Given an application running on a target system, the occurrences of essential sharing misses (PTSM) [26] are determined by data-sharing patterns in the application and by some parameters of the architecture such as the cache block size. As discussed in Section 2.2.2, the amount of data-sharing and the number of PTSM vary with the data set size and the number of processors (which is assumed to be constant in this section).

When a processor experiences a PFSM, there must have been one or more stores to the block, but not to the data element accessed since the last load of the block. Therefore, unlike PTSM, occurrences of PFSM depend on the physical placement of data in cache blocks, which seems too complex to analyze statically. PFSM are not necessarily observed on every shared variable. In any case, if the data around the one to be accessed are truly shared, the number of PFSM will be proportional to the amount of data-sharing. If not, since the cache block must be located across a data partition boundary and the number of PFSM must be proportional to the total length of partition boundaries.

Since the amounts of PTSM and PFSM are commonly dependent both on the total length of partition boundaries and on the amount of data-sharing (Section 2.2.2), we con-

1. For example, in matrix applications such as LU, the number of array elements is denoted by NxN, or N^2.

sider the number of PFSM to be approximately proportional to that of PTSM. Therefore, they will have the same order of complexity. In this section, the number of PTSM will be estimated by a big-oh expression derived from the algorithmic characteristics of an application. Then, the same estimate will be used for the number of PFSM.

3.2 Simulation and Benchmarks

The simulation environment is the Cache-Mire testbench, a program-driven simulator developed at Lund University [12]. Cache-Mire executes the application and generates memory references, which are then simulated on an architectural simulator. Instruction fetches, private data accesses, shared data accesses and synchronization (test-and-set) operations are monitored on the fly as the execution progresses. The architectural simulator simulates a bus-based cache-coherent shared-memory multiprocessor system with eight processors. The coherency protocol is described in Section 2.3 and Figure 6 (a). The caches have infinite size and block size is 32 bytes.

Seven application programs in Table 3 were run: MP3D, WATER, OCEAN, LU, BARNES, FFT and RADIX. They are parallel applications developed at Stanford University and the first three are part of the SPLASH suite [67] and the last three are from SPLASH-2 [82]. These applications are written in C using the Argonne National Laboratory macro package [11].

Table 3 shows the (small) data sets used for the samples and the (large) data sets. Table 3 also lists the shared data structures in each application. Of course, applications have more shared variables. However, some of them are not truly shared but are accessed by one processor only. The rest of them maintain programs' statistics; for instance, in MP3D, one variable keeps track of the number of particles' collisions, and another keeps track of the particle population. Since their number of misses are not significant or are not dependent on the data set size, I do not include them.

Benchmark	Sample data set sizes	Prediction		Shared variables
		DSS-1	DSS-2	
LU	48, 64, 80, 96, 112, 128, 144	288	512	A, L, thisPivot
MP3D	2K, 4K, 8K, 16K, 32K, 64K, 128K	256K	512K	Particles, Cells, Ares
WATER	32, 48, 64, 80, 96, 112, 128	360	720	VAR
OCEAN	34, 66, 130	258	514	q_multi, rhs_multi + 23 2-d arrays
FFT	2^6, 2^8, 2^{10}, 2^{12}, 2^{14}	2^{16}	2^{18}	x, trans
BARNES	512,768,1024,1280,1536,1792,2048	4096	8192	bodytab, ctab, ltab
RADIX	2K, 4K, 6K, 8K, 10K, 12K, 14K	512K	1024K	key, densities, ranks

Table 3. Size of data sets used to find fitting functions and shared variables.

3.3 Prediction Results

In Table 4, the empirical models for overall cache miss rates are given. The denominators of miss rate models are the empirical models (fitting functions) for the total number of data accesses and the numerators are the models for the total number of cache misses. The prediction results were computed by substituting the desired large data sizes into miss rate models. All detail simulation and prediction results for each type of cache misses and data references are found in Tables 46 through 52 in Appendix.A. Brief discussions of the simulation and prediction results are also provided. In the tables of the appendix, the number of essential misses are computed by summing up the numbers of CM and PTSM. The prediction error is computed as:

$$\text{prediction error } (\%) = \frac{\text{simulation result} - \text{prediction result}}{\text{simulation result}} \times 100 \qquad \text{(Eq. 10)}$$

In Appendix A, brief description of benchmark applications and their algorithmic behavior are provided. The tables include empirical models and prediction results for all kinds of performance metrics of every shared data structure in each application.

Bench-mark	Empirical Model for Miss Rate	Prediction - 1			Prediction - 2		
		sim.	pred	err%	sim	pred	err%
LU	$\frac{1.250N^2 + 12.426N - 63.308}{1.994N^3 + 8.764N^2 - 178.70N + 3849.08}$	0.2220	0.2220	0.000	0.1237	0.1238	-0.080
MP3D	$\frac{21.452N - 3650.3}{348.845N + 41097.4}$	6.1494	6.1513	-0.030	6.1332	6.1504	-0.280
WATER	$\frac{4.490N^2 + 151.164N - 221.48}{540.016N^2 - 2938.3N + 644.47}$	0.8964	0.8953	0.122	0.8646	0.8636	0.115
OCEAN	$\frac{6.911N^2 + 2137.86N + 9554.16}{4596.93N^2 - 53728.1N - 1562387}$	0.3121	0.3221	-3.204	0.2285	0.2375	-3.938
FFT	$\frac{2.315n^2 + 22.908n - 0.672}{197.952n^2 - 2614.8n + 24184.5}$	1.3149	1.2787	2.753	1.1875	1.2234	-3.023
BARNES	$\frac{34.943N + 11965.7\log N - 24249.6}{1491.8N\log N + 3278.6N - 2741459\log N + 6796905}$	0.4822	0.4947	-2.592	0.4216	0.4213	0.071
RADIX	$\frac{34009N + 219.030}{1.437N + 3575.31}$	4.3686	4.2448	2.833	4.3624	4.2348	2.924

*** sim: simulation results, pred: prediction results, err: prediction errors**

Table 4. Empirical models and prediction results.

3.4 Discussion

As we have seen in Table 4, the empirical modeling strategy works very well. The predictions are very accurate even when the sharing behaviors are quite irregular (MP3D, BARNES), when the amount of computations or the number of iterations are dependent on the run time value of the data (OCEAN, BARNES) or when barely enough number of samples (OCEAN and FFT) are available. Especially, three is the minimum number of samples required to compute three unknown parameters of OCEAN's $O(N^2)$ fitting function. The prediction errors calculated according to the Equation 10 fall below 4% and, in most cases, below 1%.

Given the possibility of building accurate empirical performance model for miss rates, a systematic methodology to obtain the polynomial expressions of performance metrics is needed. The analysis and characterization of data-sharing behavior will be the starting point. Since the characteristics of each shared data structure in an application are quite different, each shared data structure will be dealt with separately. Scaling effects of the problem size correspond to the case where the size of memory in a multiprocessor system is increased while maintaining the same number of processors; for a complete performance prediction model, the scaling effects of the number of processors will be incorporated in the performance model, as well.

Chapter 4

DATA-SHARING ANALYSIS

One of the most general application classes of parallel systems is scientific computations and most scientific programs are written according to the Single-Program-Multiple-Data (SPMD) model. As the speed of today's computer systems gets ever faster, the amount of data to be processed is accordingly growing. These data objects are normally represented by array data structures and the computations on them are performed within iterative loops such as the `for` loop construct. The information regarding the memory locations of the data objects that a processor accesses is given in the program. That is to say, the index expressions of array variable bear the information regarding possible data-sharing in the program. Therefore, the data-sharing analysis must start with the array index expression analysis.

In this section, we do not focus on the cache misses, yet. First, a set of sharing factors that affect the amount of data-sharing are defined. Then, their unique contributions to data-sharing are illustrated. The components used in array index expressions and the composite index expressions composed of the index expression components are enumerated. The data-sharing patterns for all types of index expressions are also provided. At this moment, the effects of the data set size and the number of processors on the amount of data-sharing are discussed.

4.1 Data-Sharing Factors

As the first step of sharing analysis, we define data-sharing factors.

4.1.1 Data Set Size and Number of Processors

As shown in Figure 4, data-sharing is affected by the *data set size* and the *number of processors*. The effects of these two sharing factors were discussed in detail in Section 2.2.2. They are the variable parameters in the data-sharing characterizer and performance model.

4.1.2 Data Partitioning

As shown in Figure 4, various partitioning methods have a different impact on data-sharing. In fact, what makes the effects of partitioning methods on data-sharing distinct is the partition block size. For example, the block size (or the number of partition blocks) in Figure 4 (a) is larger (smaller) than that in Figure 4 (b). Smaller partition blocks causes shows more data-sharing. Therefore, in the sharing analysis, the size of partition blocks and the partitioning method are simultaneously taken into account.

Figure 8 illustrates data partitioning examples for one- and two-dimensional arrays. Given N as the data set size in one dimension of an array and P as the number of processors, the partition block size and the number of partition blocks are provided. One may bring up more complicated partitioning schemes such as value-dependent partitioning. However, we only consider location-dependent partitioning schemes where cyclic, block-cyclic and block partitioning are known as the most popular schemes [40, 42, 73].

4.1.3 Access Distance

While the three factors described above are related to data placements in parallel applications, the access distance is related only to the program code. Before going any fur-

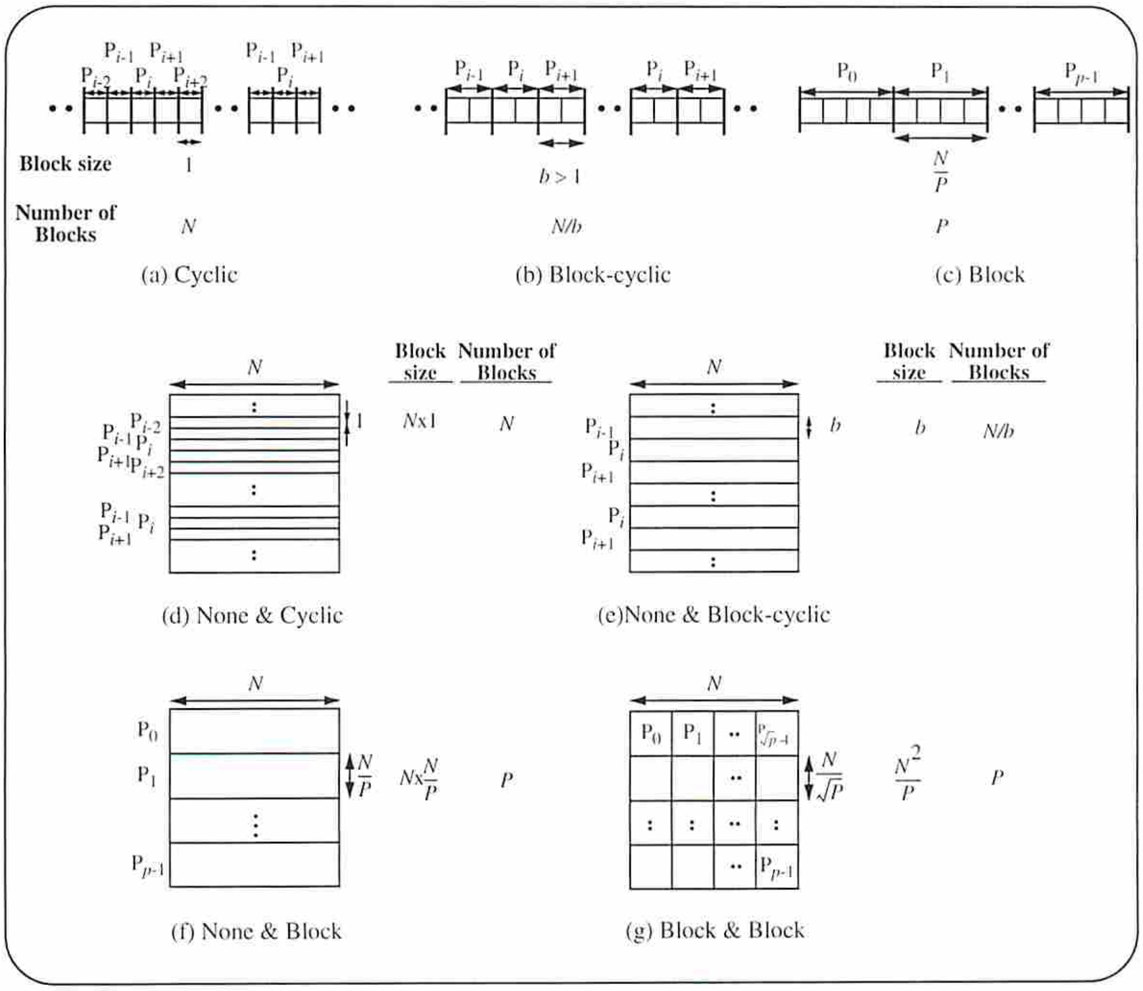


Figure 8. Partition block size and length of partition boundaries.

ther, it is necessary and helpful to make the following definitions. Assume an N element array `A` and a FORALL loop with variable `I`.

Definition 1. (*Pure index and Applied index*) The index expression of '`I`' in `A[I]` is called the *pure index* and all other index expressions are called the *applied index*.

Definition 2. (*Center of computation*) An array element designated by pure index, `A[I]`, is the center of computation (Section 2.2.2).

Definition 3. (*Participants in computation*) All array elements engaged in the computation, except for `A[I]`, are called the participants in computation.

Definition 4. (*Access distance*) The access distance of a participant is the number of array elements lying between `A[I]`, inclusively, and the participant, exclusively.

Definition 5. (*Vicinities in computation*) The participants whose indices are in the form of `A[I+d]`, where `d` is the access distance, are called the vicinities in computation.

Definition 6. (*Strangers in computation*) The participants whose indices are not in the form of `A[I+d]`, where `d` is the access distance, are called the strangers in computation.

If a FORALL loop contains only pure indices, no data-sharing occurs since each processor finds all the required data within its local partition block. In this case, the access distance of an array element, designated by the pure index from the center of computation, is zero. In contrast, if there are participants defined in the FORALL loop whose access distances are not zero, there will always be data-sharing. The sharing behavior of a participant is determined by the value of the access distance as will be shown below.

Of participant types, first consider the vicinities. The vicinities are the elements whose locations are specified by relative distances from the center of computation. Given an array `A` partitioned into b element blocks and accessed in a FORALL loop of variable `I`, the vicinities specified as `A[I+`d`]` form a block of the same size at d elements away from the center of computation (Figure 9 (a)). In Figure 10 where `A` is partitioned into blocks and distributed among P processors, lightly shaded elements are accessed by the Home while heavily shaded elements are accessed by a remote processor. In (a), only the center of computation is accessed without data-sharing. On the other hand, in (b), element `A[I+1]` is the vicinity whose access distance is 1. The access distance in (c) is 3. From the figures, we can see that the amount of data-sharing is proportional to the access distances of the vicinities.

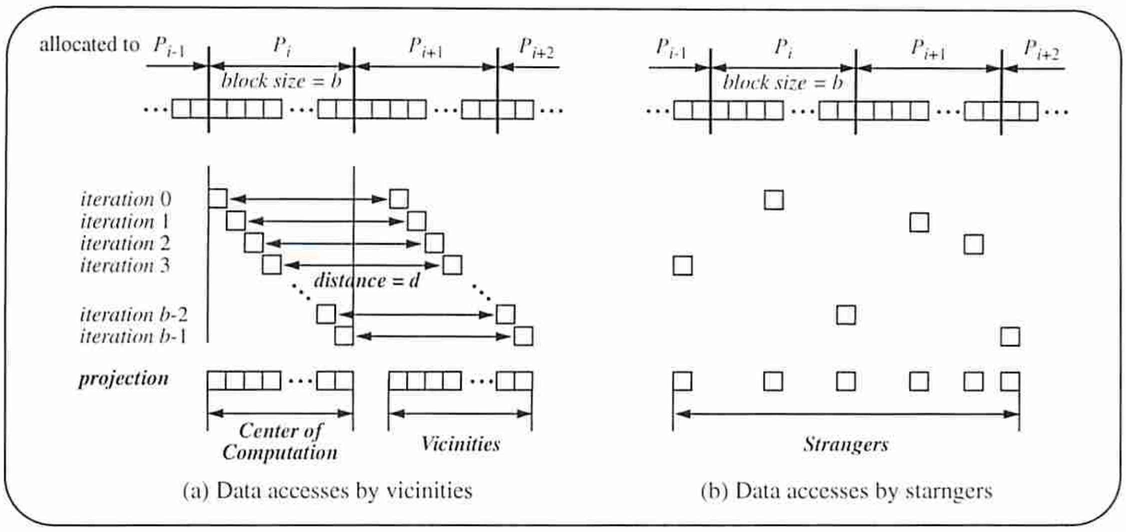


Figure 9. Vicinities and strangers.

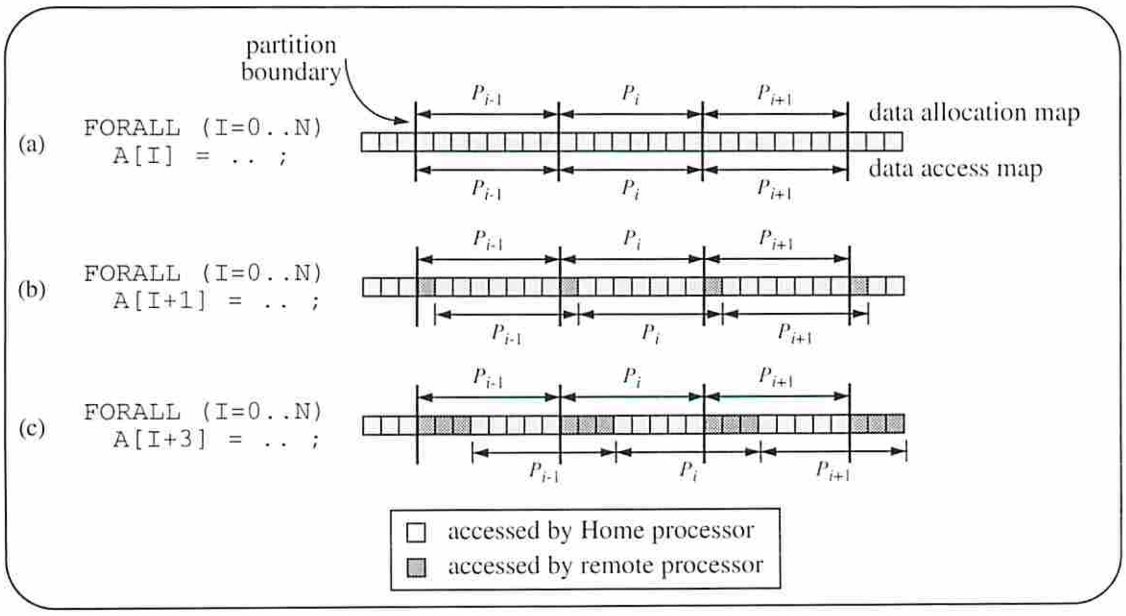


Figure 10. Effects of access distance on data-sharing.

In contrast, strangers are the elements whose locations are determined regardless of the center of computation. The "*strangers*" are named by opposition to the "*vicinities*". Figure 9 (b) presents data accesses of strangers whose indices are determined by a certain run time variable. The array elements may be accessed by a remote processor or by the

Home. Generally, in this case, the probability that an element is accessed by the Home is inversely proportional to the number of processors. In other words, the larger the number of processors, the more data-sharing by remote accesses are observed.

4.2 Array Index Analysis

Data-sharing analysis can be done in two phases. The first phase is to identify how an N element array is partitioned and allocated to P processors. At this stage, the first three sharing factors (data set size, number of processors and partitioning schemes) are simultaneously taken into account. Given data distribution information, the next phase is to determine the method of data-sharing by the processors. Being one of the causes of data-sharing, the access distance will be investigated in detail, in this section. Since the access distance is defined by the index of an array, the sharing analysis can be seen in the same perspective as array index analysis.

Since the number of possible index expression types is uncountable, efficient grouping is necessary so that one may have a manageably finite number of index expression types. The grouping must guarantee that an arbitrary index expression should fall into exactly one category. In categorizing index expressions, we first define *index expression components* and, then, *compound index expressions* that are composed of the index expression components as building blocks. Index expression types will be introduced with brief explanations of their access patterns. Their sharing patterns will be analyzed in the next section for various data partitioning schemes.

Prior to introducing the methodology, we assume the *meaningfulness of data set size*. It implies the data set size, N, is large enough so that we may say $N >> P$ or $N/P >> 1$, where P is the number of processors, although exact values of N and P are not known.

Array indices are numerical expressions consisting of loop variables, numeric constants and/or general variables. We separate general variables into *loop-invariant vari-*

ables and *loop-variant variables*. Memory locations of array elements may be fixed through all iterations or varying in each iteration according to whether their indices are loop-invariant or loop-variant, respectively. General variables are further grouped according to whether their values can be evaluated at compile time or not. We refer to general variables whose values can be known at compile time as *compile time variables* and those whose values can be known only at runtime as *runtime variables*. The memory locations specified by compile time variables can be known via static analysis. By contrast, the memory locations specified by run time variables cannot be known since they depend on the run time computation results. In the rest of the paper, we use the terms *particular* and *arbitrary* to denote that a memory location is determined by compile time variables and runtime variables, respectively.

Given an index expression in SPMD programs, it can be stated that *each processor accesses a particular or an arbitrary array element whose location is variant or invariant with respect to the loop variables*. This interpretation is made from the view point of processors. In our methodology, sharing analysis is done from the view point of data. The interpretation of data accesses seen from the view point of data is that *an array element is accessed by particular or arbitrary processors which may vary as the loop advances*. Figure 11 compares the two interpretations from the view points of the processors and data.

The processors accessing an array element are named by the entries in Table 5. Among them, the *Home* and *All* are self-explanatory. The others are first distinguished according to whether an array element is accessed by particular or arbitrary processors (Determinism). The determinism is explained as follows. In Figure 11 (b), `c` is a compile time variable so that the memory location of `A[c]` is considered *particular* and the processor accessing `A[c]` is the Home of the *particular* array element. The Home of *particular* element is considered to be *particular*, too. In contrast, if `c` is a run time variable (Figure 11 (c)), the memory location is considered as *arbitrary* and the processor accessing `A[c]`

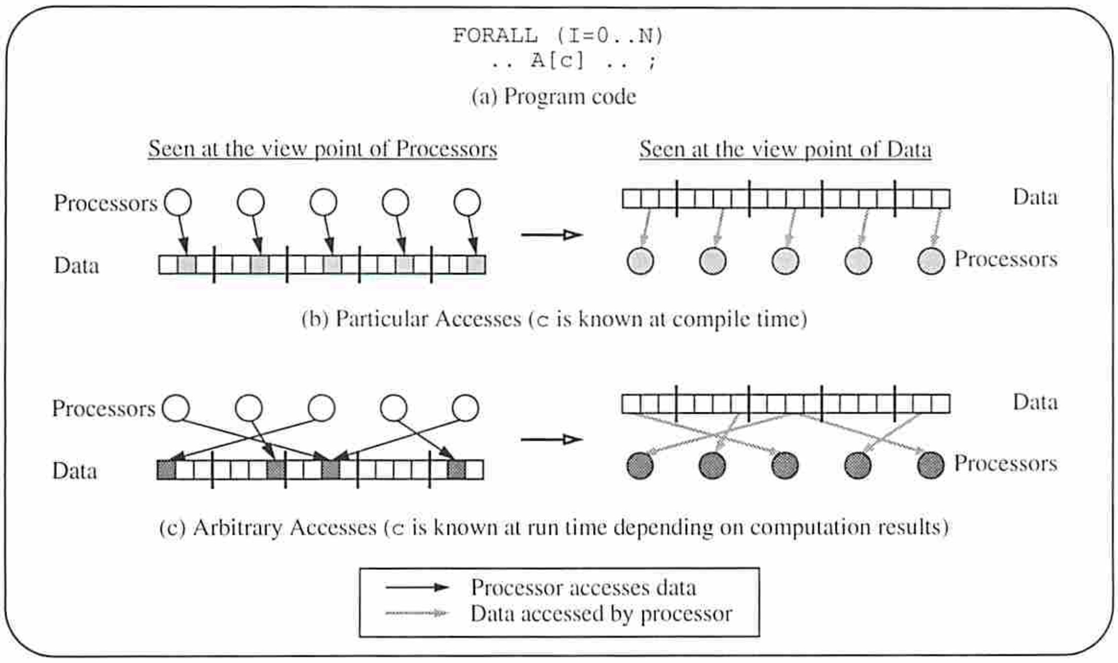

FORALL (I=0..N)
.. A[c] .. ;
(a) Program code
Seen at the view point of Processors
Seen at the view point of Data
Processors
Data
Data
Processors
(b) Particular Accesses (c is known at compile time)
Processors
Data
Data
Processors
(c) Arbitrary Accesses (c is known at run time depending on computation results)
Processor accesses data
Data accessed by processor

Figure 11. Data access interpretation methods.

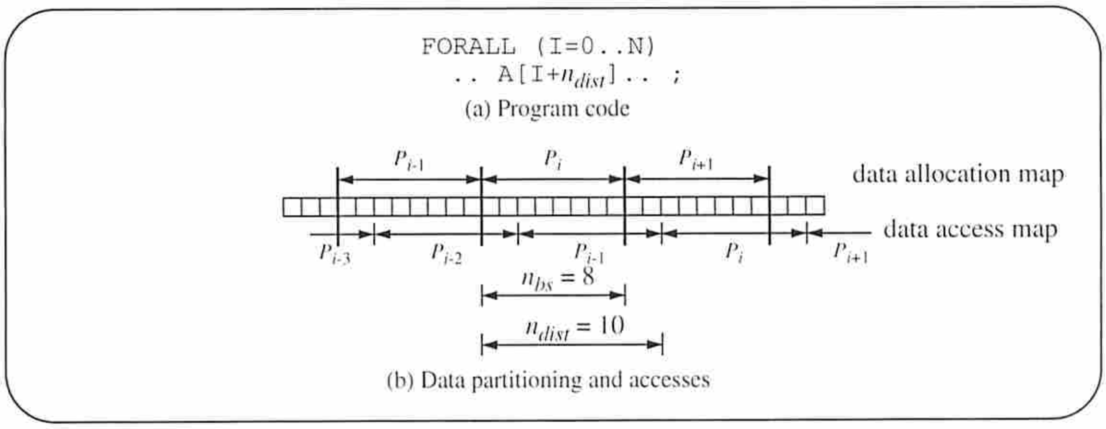
is the Home of the *arbitrary* array element. The Home of *arbitrary* element is considered to be *arbitrary*.

The processor attributes in Table 5 are further distinguished according to whether an element may be accessed by the Home or not (Home inclusion). In Figure 11 (c), an array element, arbitrarily selected by run time variable c, could be accessed by either its Home or any remote processor. On the other hand, imagine an array partitioned into blocks of size b and the program codes including A[I+d] where I is the loop variable and $d > b$. No element is accessed by their Home (Figure 12) since $d = 8$ and $b = 10$.

The last distinction in Table 5 is made according to whether an element is accessed by a single processor or multiple processors (Cardinality). In Figure 12 (b), a certain element is accessed by only one processor while, in Figure 11 (c), it may be accessed by two or more processors.

Processors	Definition	Deter-minism	Home-inclusion	Cardinal-ity
Home	a processor to which a particular data is allocated	Yes	Yes	singular
One	one particular processor	Yes	Yes	singular
AnyOne	one arbitrary processor	No	Yes	singular
Other	one particular processor except Home	Yes	No	singular
AnyOther	one arbitrary processor except Home	No	No	singular
Some	one or more particular processor	Yes	Yes	multiple
Any	none or more arbitrary processor	No	Yes	multiple
Others	one or more particular processor except Home	Yes	No	multiple
AnyOthers	none or more arbitrary processor except Home	No	No	multiple
All	all processors	Yes	Yes	multiple

Table 5. Processor attributes by which a certain data is accessed.

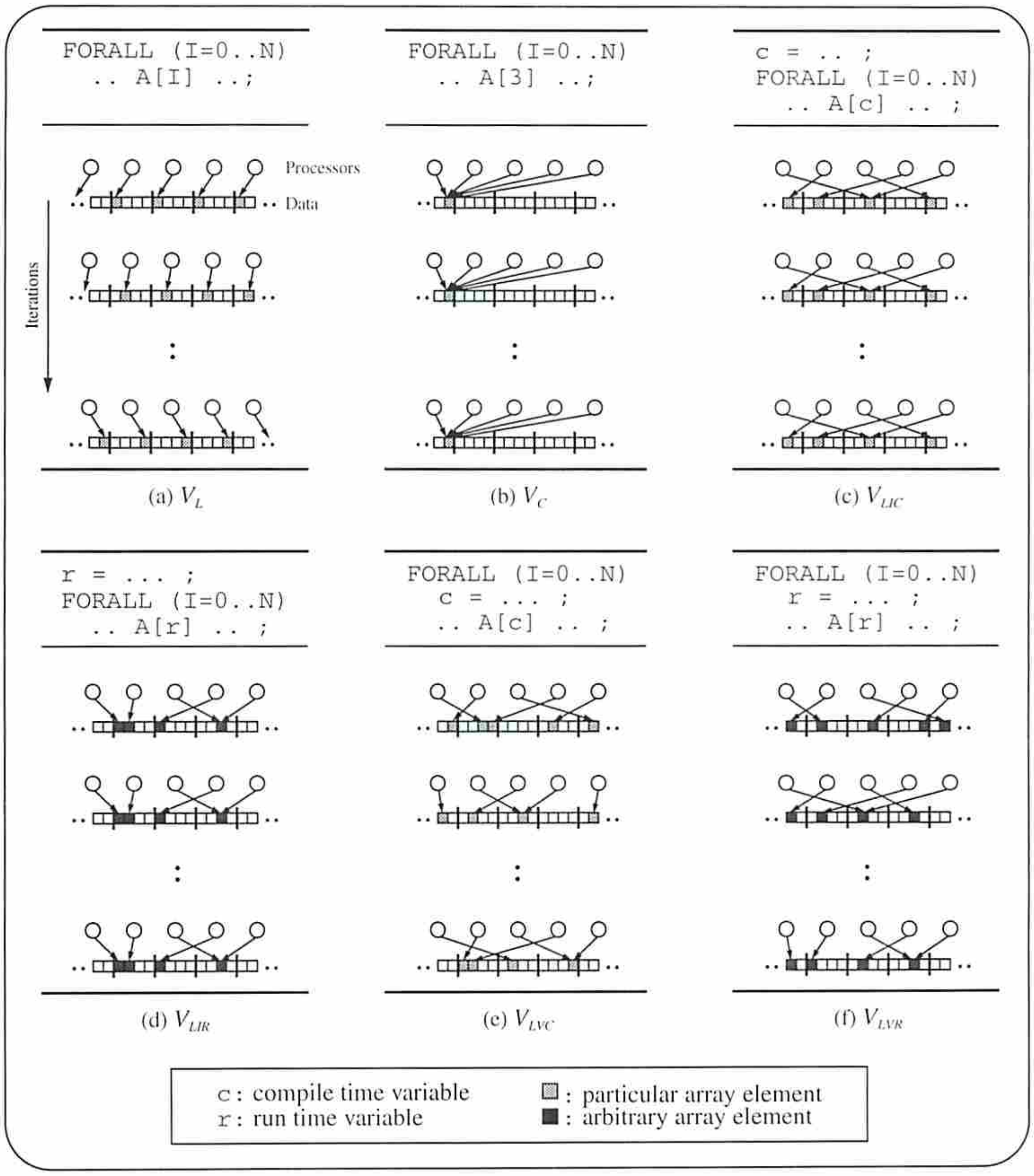


```
FORALL (I=0..N)
   .. A[I+n_dist] .. ;
```

Figure 12. Home-inclusion property of the PRC.

4.2.1 Index Expression Components

The array index components that form an array index expression are introduced in this section. Sample program codes and access patterns are illustrated in Figure 13.



```
FORALL (I=0..N)
  .. A[I] ..;
```

```
FORALL (I=0..N)
  .. A[3] ..;
```

```
c = .. ;
FORALL (I=0..N)
  .. A[c] .. ;
```

```
r = ... ;
FORALL (I=0..N)
  .. A[r] .. ;
```

```
FORALL (I=0..N)
  c = ... ;
  .. A[c] .. ;
```

```
FORALL (I=0..N)
  r = ... ;
  .. A[r] .. ;
```

c : compile time variable
r : run time variable
▨ : particular array element
■ : arbitrary array element

Figure 13. Array index expression components.

4.2.1.1 Loop Variable

A loop variable, V_L, designates all elements that are accessed by their Home through the FORALL loop. Array elements designated by V_L are accessed only by their Home as shown in Figure 13 (a).

4.2.1.2 Numeric Constant

A numeric constant, V_C, designates a single *particular* element whose location is fixed through iterations. The element is accessed by all processors in every iteration as shown in Figure 13 (b).

4.2.1.3 Loop-invariant Compile Time Variable

A loop-variant compile time variable, V_{LIC}, designates *particular* elements whose locations are fixed through all iterations. Once a processor selects an element, it keeps accessing the same element in all iterations. Consequently, every selected element is interpreted as accessed by a *particular processor* including its Home, for *N*/*P* times, which is the loop size executed by a processor. The corresponding value in Table 5 is *One*.

4.2.1.4 Loop-invariant Run Time Variable

A loop-invariant run time variable, V_{LIR}, designates *arbitrary* array elements whose locations are fixed through iterations (Figure 13 (d)). Once a processor selects an element, it keeps accessing the same element in all iterations. Consequently, every selected array element is interpreted as accessed by an *arbitrary processor* including its Home, for *N*/*P* times. This corresponds to the processor *AnyOne* in Table 5.

4.2.1.5 Loop-variant Compile Time Variable

A loop-variant compile time variable, V_{LVC}, designates *particular* array elements whose location changes in every iteration (Figure 13 (e)). In contrast to the V_{LIC} in Section 4.2.1.3, the loop-variant variable c designates a new element in each iteration. Through *N* iterations, therefore, an element is very likely to be selected more than once by at least two processors. That is to say, every array element is interpreted as accessed by *particular processors* including the Home. The corresponding processor value in Table 5 is *Some*.

4.2.1.6 Loop-variant Run Time Variable

A loop-variant run time variable, V_{LVR}, designates *arbitrary* array elements whose location changes in every iteration (Figure 13 (f)). Through N iterations, as in Section 4.2.1.5, an element is very likely to be selected more than once by at least two processors. That is to say, every array element is interpreted as accessed by *arbitrary processors* including the Home. The corresponding processor value in Table 5 is *Any*.

4.2.1.7 Summary

Note that V_L is a private variable and V_C is a shared constant in SPMD programs. Other index expression components (V_{LIC}, V_{LIR}, V_{LVC} and V_{LVR}) can be either shared or private variables. When they are shared variables, a common array element is selected and accessed by all processors. Although this situation is theoretically possible, they are not often implemented in common SPMD programs. Furthermore, they are against data homogeneity (Observation 3). For this reason, the focus is mainly put on the cases where V_{LIC}, V_{LIR}, V_{LVC} and V_{LVR} are private variables.

In summary, Table 6 shows the processors that access array elements indexed by each index expression component.

Loop Components	V_L	V_C	V_{LIC}	V_{LIR}	V_{LVC}	V_{LVR}
Processors	Home	All	One	AnyOne	Some	Any
Data Homogeneity	Yes	No	No	No	Yes	Yes

Table 6. Processor attributes for index expression components.

4.2.2 Compound Index Expressions

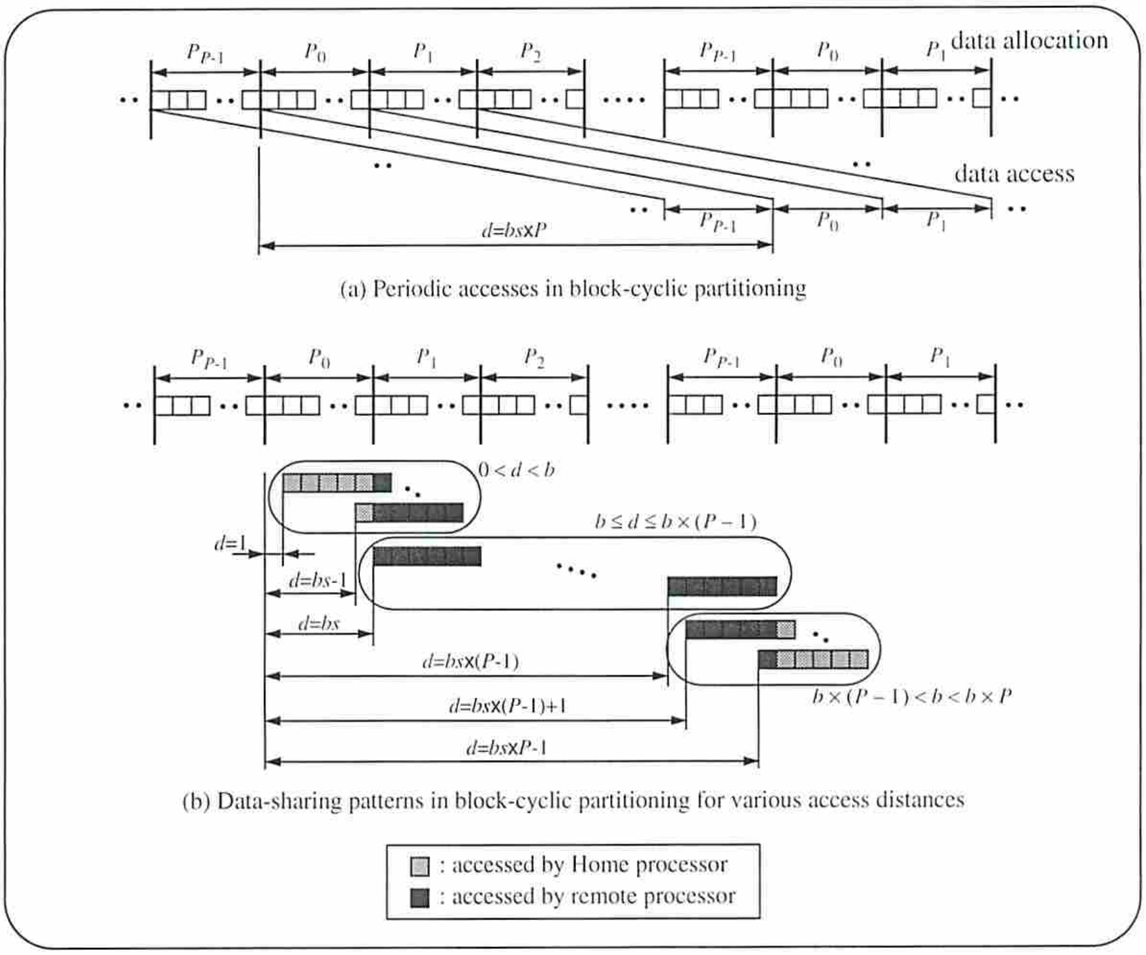
This section presents index expressions which are composed of index expression components.

Data partitioning is ordinarily done so that as little data communication as possible is required. Once partitioned, data are mainly accessed in a *sequential manner* within FORALL loops. The variables of type V_L are used for this purpose and array elements indexed by them are called the centers of computation. As noted before, other elements called participants often take part in computations and the accesses to them may require data communications. The participants were classified into strangers and vicinities in Definitions 5 and 6. In subsequent sections, compound index expressions and data-sharing patterns of vicinities are discussed first and the strangers will follow.

4.2.2.1 Compound Index Expressions of Vicinities

Compound expressions of vicinities, E_{VIC}, are in the form of $\mathtt{A}[V_L+d]$, where the access distance d is a function of V_C,V_{LIC} and V_{LIR}. Note that E_{VIC} does not include V_{LVC} or V_{LVR}. The reason will be found in Section 4.2.2.2. The access distance is formularized as $d = f_d(V_C,V_{LIC},V_{LIR})$, where f_d is a multi-variable function. The effects of d on data-sharing for various partitioning methods are discussed below.

Block-cyclic partitioning with b element blocks is known as periodic with the period of bP elements. Therefore, should d be multiple of bP, the element $\mathtt{A}[V_L+d]$ is accessed by its Home (Figure 14 (a)). In order to look at a single period, we set $d = |d \bmod Pb|$. The data-sharing pattern varies according to the ranges of d; $0 < d < b$, $b \le d \le b \times (P-1)$ and $b \times (P-1) < b < b \times P$. Figure 14 (b) shows the data-sharing pattern caused by P_0's accesses. When $0 < d < b$, some elements are accessed by the Home and others by the adjacent processor which lies in the positive (negative) direction if the distance is positive (negative). The entry in Table 5 for the adjacent processor is *Other* meaning a particular processor which is not the Home of the accessed data. The number of remote array elements is d. For $b \le d \le b \times (P-1)$, all elements are accessed by the remote processor whose entry in Table 5 is *Other*. Finally, when $b \times (P-1) < d < b \times P$, some ele-



P_{P-1}
P_0
P_1
P_2
data allocation
data access
$d=bs\times P$
(a) Periodic accesses in block-cyclic partitioning
$0 < d < b$
$b \le d \le b \times (P-1)$
$d=1$
$d=bs-1$
$d=bs$
$d=bs\times(P-1)$
$d=bs\times(P-1)+1$
$d=bs\times P-1$
$b \times (P-1) < b < b \times P$
(b) Data-sharing patterns in block-cyclic partitioning for various access distances
▨ : accessed by Home processor
■ : accessed by remote processor

Figure 14. Effect of access distance in block-cyclic partitioning.

ments are accessed by the Home and others by the adjacent processor which lies in the negative (positive) direction if the distance is positive (negative). The processor value in Table 5 is *Other*. The number of remote array elements is $b-(d \bmod b)$. As before, the entry in Table 5 for the adjacent processor is *Other*.

Cyclic partitioning is an extreme of the block-cyclic method with b = 1 and the period is P elements. If d is a multiple of P, all elements are accessed by their Home. If not, all elements are accessed by a remote processor (Figure 15 (a) and (b)) whose corresponding entry in Table 5 is *Other*. In Figure 15, data-sharing patterns caused by P_0's accesses are shown.

Finally, block partitioning is another extreme of block-cyclic partitioning with $b = N/P$ and no period. If $0 < d < b$, first b - d elements are accessed by the Home and other d elements by the adjacent processor which lies in the positive (negative) direction if d is positive (negative). If $b \leq d$, all element are accessed by a remote processor (Figure 15 (c)). The entry in Table 5 for the remote processor is *Other*.

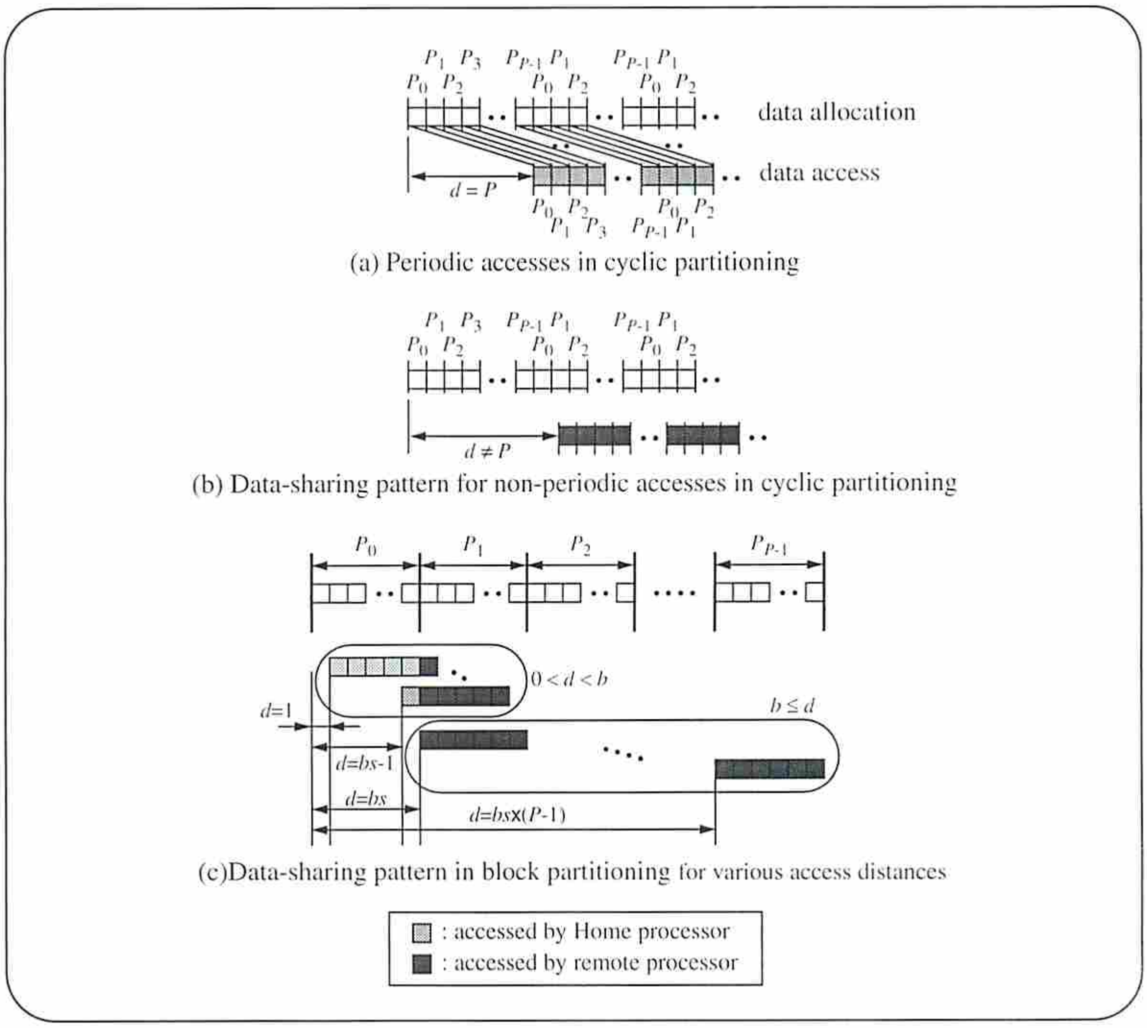

data allocation
data access
$d = P$
(a) Periodic accesses in cyclic partitioning
$d \neq P$
(b) Data-sharing pattern for non-periodic accesses in cyclic partitioning
$0 < d < b$
$b \leq d$
d=1
d=bs-1
d=bs
d=bsx(P-1)
(c)Data-sharing pattern in block partitioning for various access distances
: accessed by Home processor
: accessed by remote processor

Figure 15. Effect of access distance in cyclic and block partitioning.

The access distance of A[V_L+d] from A[V_L] can be known at compile time, depending on the existence of V_{LIR} in f_d. If known at compile time without V_{LIR}, data-sharing analysis is straightforward as shown thus far. Otherwise, worst case estimation is made by assuming the value of d to be large enough so that every element is involved in data-sharing. This yields the most data-sharing. In this case, every element is interpreted as accessed by a certain processor which might be the Home or an arbitrary remote processor. Such processor's corresponding entry in Table 5 is *AnyOne*.

Vicinity Expressions (in the form of A[V_L+d], No V_{LVC} or V_{LVR})

d	Conditions		Processor	No. accessed elements per block	Processor	No. accessed elements per block
Without V_{LIR}	Cyclic Partition (b=1)	$d = nP$	Home	1	-	-
		$d \neq nP$	Other	1	-	-
	Block-cyclic Partition (b=c)	$0 < d^* < b$	Home	$(b - d)$	Other	d
		$b \leq d^* \leq b \times (P-1)$	Other	b	-	-
		$b \times (P-1) < b^* < b \times P$	Home	d mod b	Other	$b - (d$ mod $b)$
	Block Partition (b=N/P)	$0 < d < b$	Home	$(b - d)$	Other	d
		$b \leq d$	Other	N/P	-	-
With V_{LIR}	In all cases		AnyOne	b	-	-

** A: given array, N: data set size, P: number of processors, b: partition block size, d: access distance, $d^*=|d \bmod bP|$, c and n: arbitrary non-zero integers

Table 7. Data access pattern summary for vicinity expressions.

Data-sharing characteristics of the array elements with vicinity expressions (E_{VIC}) are summarized in Table 7. When there is only one processor value in a row, all array elements present a common sharing behavior. Otherwise, in the shaded rows, some elements and the others are accessed by different types of processors exhibiting different sharing behaviors. Distinct sharing behaviors are observed when b consecutive array elements accessed by a certain processor fall across the partition boundaries between the Home and adjacent processor. In shaded rows, the number of elements accessed by a remote processor is determined by various factors as shown in the last column of Table 7.

4.2.2.2 Compound Index Expressions of Strangers

In the previous section, it was shown that the location of the center of computation is designated according to the applied data-partitioning method and the locations of vicinities are determined by the relative distances from the center of computation. Therefore, the access patterns of compound expressions of vicinities were distinguished for different data partitioning methods in Section 4.2.2.1 and Table 7. On the other hand, as shown in Figure 9, the locations of strangers are not associated with the center of computation nor the loop variable, V_L. For this reason, the analysis of sharing pattern of strangers will be performed without distinguishing data partitioning methods.

First of all, consider a compound index expression of strangers, E_{STR}, including V_{LVR} that designates distinct arbitrary array elements in each iteration. Once V_{LVR} exists, the whole expression becomes a loop-variant and its value is known only at run time regardless of other components. Resulting sharing pattern is identical to what is shown in Section 4.2.1.6. We call such an expression the *loop-variant run time index expression* (E_{LVR}).

Without V_{LVR}, V_{LVC} dominates index expressions in that they are loop-variant. The determinism depends on whether V_{LIR} is included in the expression or not. If V_{LVC} and V_{LIR} are coincident, since the value of the expression is not known at compile time due to V_{LIR}, array elements are arbitrarily selected regardless of other components. The sharing pattern is identical to what is shown in Section 4.2.1.6. If no V_{LIR} (and no V_{LVR}) is in the expression, there is no run time component and selected array elements are particular. The sharing pattern is identical to what is shown in Section 4.2.1.5. The former is of type *loop-variant run time index expression* (E_{LVR}), and the latter is called the *loop-variant compile time index expression* (E_{LVC}). In consequence, if an index expression includes V_{LVR} or V_{LVC}, it designates arbitrary memory locations of stranger elements, regardless of the other index expression components.

Now, consider compound index expressions for stranger elements without V_{LVR} or V_{LVC}. If an expression includes no V_L, it is only composed of V_C, V_{LIC} and/or V_{LIR}. Any combinations of these three types of variables are loop-invariant. If there is V_{LIR}, the elements are selected arbitrarily, regardless of the other two, and the expression is called the *loop-invariant run time expression* (E_{LIR}). The sharing pattern is given in Section 4.2.1.4. If there is no V_{LIR}, the expression is composed of V_C and V_{LIC}, and it is called the *loop-invariant compile time expression* (E_{LIC}). The sharing pattern is the same as mentioned in Section 4.2.1.3.

If a compound index expression includes V_C, and it is not in the form of the vicinity expression, it must be written as A[$f(V_L, V_C, V_{LIC}, V_{LIR})$]. By restricting the function to be affine with respect to the loop variable V_L [40, 42, 73], we can rewrite the compound expression into A[$g(V_C, V_{LIC}, V_{LIR}) \times V_L + h(V_C, V_{LIC}, V_{LIR})$], where g and h are functions of V_C, V_{LIC} and V_{LIR}. These are discussed in the following section.

Table 8 shows the processors accessing array elements indexed by E_{STR}. The entries for E_V and E_C are put in this table for convenience although they do not specify the stranger array elements. Data access patterns for various types of E_{STR} are illustrated in Sections 4.2.1.1 through 4.2.1.6. Unlike Table 7, each sharing pattern of E_{STR} can be represented by a single processor type although the sharing behaviors of data in some cases are not homogeneous. For E_C, E_{LIC} and E_{LIR}, if the data set size is N, only 1 or P elements are selected and accessed $O(N)$ or $O(N/P)$ times, respectively. None of the remaining elements are accessed. For E_V, E_{LVC} and E_{LVR}, sharing behaviors of all elements are homogeneous and the number of accesses made to a single element is estimated as $O(1)$.

4.2.2.3 Sparse Index Expressions of Strangers

In this section, the index expression of A[$g(V_C, V_{LIC}, V_{LIR}) \times V_L + h(V_C, V_{LIC}, V_{LIR})$] for strangers is presented. Defining the sparsity of accessed elements, integer function $g(V_C$, Stranger Expressions (not in the form of A[V_L+d] or A[gV_L+h])

Expression	Must include	Must not include	Don't Cares	Processor	No. elements
E_L	V_L	V_C, V_{LIC}, V_{LIR}, V_{LVC}, V_{LVR}	-	Home	All
E_C	V_C	V_L, V_{LIC}, V_{LIR}, V_{LVC}, V_{LVR}	-	All	1
E_{LIC}	V_{LIC}	V_{LVR}, V_{LVC}, V_{LIR}	V_L, V_C,	One	P
E_{LIR}	V_{LIR}	V_{LVR}, V_{LVC}	V_L, V_C, V_{LIC}	AnyOne	P
E_{LVC}	V_{LVC}	V_{LVR}, V_{LIR}	V_L, V_C, V_{LIC}	Some	All
E_{LVR}	V_{LVC}, V_{LIR}	V_{LVR}	V_L, V_C, V_{LIC}	Any	All
	V_{LVR}	-	V_L, V_C, V_{LIC}, V_{LIR}, V_{LVC}		

** A: given array, P: number of processors

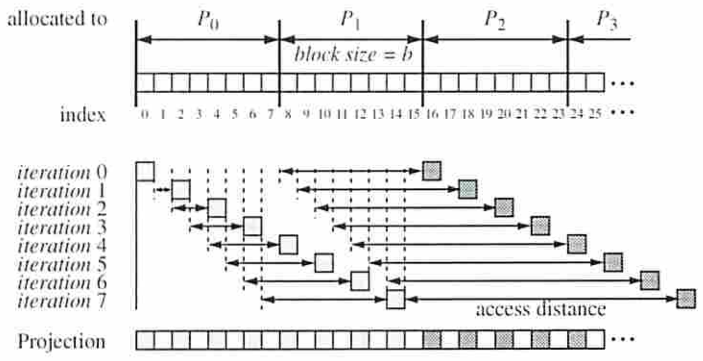
Table 8. Data access pattern for stranger expressions.

V_{LIC}, V_{LIR}) is called the *sparsity function*. Consider the code segment in Figure 16 (a). The access distance from the center is $d = g(V_C, V_{LIC}, V_{LIR}) \times V_L + h(V_C, V_{LIC}, V_{LIR}) - V_L$ which varies as the loop progresses. The largest distance from the center is found when the loop variable points to the last element in a partition block. It is $d_{max} = g(V_C, V_{LIC}, V_{LIR}) \times (b\text{-}1) + h(V_C, V_{LIC}, V_{LIR}) - (b\text{-}1)$. As shown in Figure 16, only a few first elements of an array are accessed by the Home; all others are accessed by a particular remote processor (*One* in Table 5). The number of elements accessed by the Home is $\lceil b/g \rceil - h$.

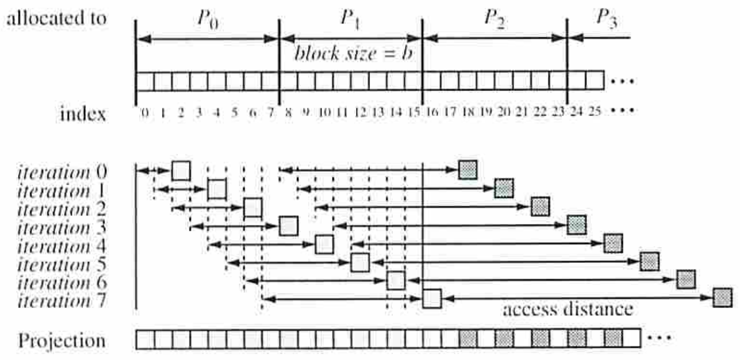
Without losing generality, we may assume array elements are accessed in a wrap-around fashion. For example, given an N element array A and a sparse index expression A[2I], A[0] is accessed when I = N/2, instead of A[N] which is out of the index range. This assumption is valid for all other types of index expressions. As another example, given an index expression A[I+2], A[2] is accessed when I = N-1, instead of A[N+1] which is out of the index range. Based on this assumption, the number of array elements accessed in a FORALL loop can be measured. If b and g are *not relatively prime* numbers,

```
my_start = N / P * my_pid;  /* N : data set size */
my_last = my_start + N / P; /* P : number of processors */
for ( I = my_start ; I < my_last ; I++ )
   .. A[2I] .. ;
```

(a) Program segment for sparse array index expression



(b) Access pattern of sparse index expression A[2I]



(c) Access pattern of sparse index expression A[2I+2]

☐ Accessed by P_0
▨ Accessed by P_1

Figure 16. Access patterns of sparse array indices.

only *N*/*g* elements of the array are accessed; otherwise, all elements are accessed. We can summarize to the following observations.

- Being an integer function, $g \geq 2$.
- The indexes of accessed elements are periodic with period of *g*.

- If b and g are relatively prime numbers,
    - total number of accessed elements is N, the size of FORALL loop.
    - each element is accessed $O(1)$ times in a FORALL loop.
    - each element is accessed by one processor.
- If b and g are *not* relatively prime numbers,
    - total number of accessed elements is N/g, the size of FORALL loop.
    - accessed elements are accessed $O(g)$ times in a FORALL loop.
    - accessed elements are accessed by g or P processors whichever is smaller.

In consequence, data-sharing characteristic of sparse index expression is similar to that of E_{LVC} in that the memory locations of array elements are analyzible at compile time and they vary as the loop advances. Furthermore, in both sparse index expression and E_{LVC} cases, majority of array elements are engaged in data-sharing. Due to these reasons, for convenience purpose, sparse array expressions are categorized into E_{LVC} in this dissertation.

4.2.3 Index Expressions for Multiple Array Elements

Thus far, an index expression is used to denote one array element in an iteration of the given FORALL loop. In many scientific computations, it is often wanted to access a group of array elements in an iteration. For example, some computations in image or signal processing application require the values of a certain data object and its neighbors. Another example is found in molecular science where the interactions between a certain molecule object and its neighbors are calculated. The index expression within a FORALL loop is in the form of Figure 17 (a). α and β specify the neighboring array elements and they must be either constants or some compile time variable. Multiple element index expressions can be associated with any types of array index expressions discussed until now (Figure 17 (b)).

```
FORALL ( I=0; I<N; I++ ) {
      for ( j=α; j<β; j++ )
           .. A[I+j] ..;
}
```

(a) Multiple element index expression for E_L

```
FORALL ( I=0; I<N; I++ ) {
      for ( j=α; j<β; j++ )
           .. A[E + j] ..;
}
```

(b) Multiple element index expression for a certain expression type E

Figure 17. Index expression for multiple array elements.

4.3 Sharing Pattern Analysis

In Section 4.2, the types of index expressions and their access patterns were provided. Data-sharing patterns of those index expressions will be analyzed in this section for various data partitioning methods. In addition, to complete the sharing pattern analysis, the effects of data set size and number of processors will be discussed.

Among data partitioning methods, data-sharing patterns of index expressions for block-cyclic method will be investigated first. The cases of cyclic and block partitioning schemes will follow.

4.3.1 Single Element Vicinity Index Expression

4.3.1.1 Block-Cyclic Partitioning

First note that the partition block size in block-cyclic partitioning is a constant. An index expression for a single vicinity, $E_{VIC\text{-}S}$, was discussed in Section 4.2.2.1 where distinct access patterns were presented according to the access distance, d.

Indeed, what is desired is to first analyze the data-sharing patterns for a specific size of a data set, N, and the number of processors, P. Then, the effects of N and P on data-sharing will be investigated. In this context, by eliminating the distinction caused by *P* in Table 7, we merge the lower two rows into one. So, we are left with only two cases; $0 < d < b$ and $d \geq b$. When $0 < d < b$, the first *d* elements in a block are accessed by *Other* processor while the last $b - d$ elements by the Home. Since the number of blocks in block-cyclic partitioning is *N/b*, where *b* is a constant, the amount of overall data -sharing is proportional to *O(N)*. However, the number of processors has no impact.

When $d \geq b$, every array element is interpreted to be accessed by the Home or *Other*; corresponding value in Table 5 is *One*. Since all *N* elements are potentially engaged in remote accesses, the amount of data-sharing is proportional to *O(N)*. In addition, among *P* processors, the probability that a processor accessing a certain element is the Home is *1/P*. That is, the amount of data-sharing due to remote accesses is proportional to $O((P-1)/P)$. This result is due to the periodic distribution of partition blocks to the processors.

4.3.1.2 Cyclic Partitioning

As given in Table 7, all elements of an array are accessed by the Home if $d = nP$, by a remote processor if $d \neq nP$. In this study, the worst case estimation is assumed; we assume $d \neq nP$, regardless of the value of *d*, and all elements are accessed by a remote processor. The effect of data set size on the amount of data-sharing is *O(N)*. As in block-cyclic partitioning, due to the periodicity of data distribution, the amount of data-sharing is proportional to $O((P-1)/P)$.

4.3.1.3 Block Partitioning

As in block-cyclic partitioning case in Section 4.3.2.1, there are two cases; $0 < d < b$ and $d \geq b$ (Table 7). However, the data set size and the number of processors have quite different influences on data-sharing. First of all, when $0 < d < b$, *d* elements among *N/P* elements in each partition block are engaged in data-sharing in each block. While there are N/b (= $O(N)$) blocks in block-cyclic method, there are P blocks in the block partitioning case. Thus, the amount of data-sharing is proportional to $O(P)$ while there are no effects due to the data set size. When $d \geq b$, all *N/P* elements in each partition block are engaged in data-sharing. With P blocks, the amount of data-sharing is proportional to $O(N)$ while the number of processors has no effect.

4.3.1.4 Summary

The effects of the data set size and number of processors on data-sharing in $E_{VIC\text{-}S}$ are summarized in Table 9.

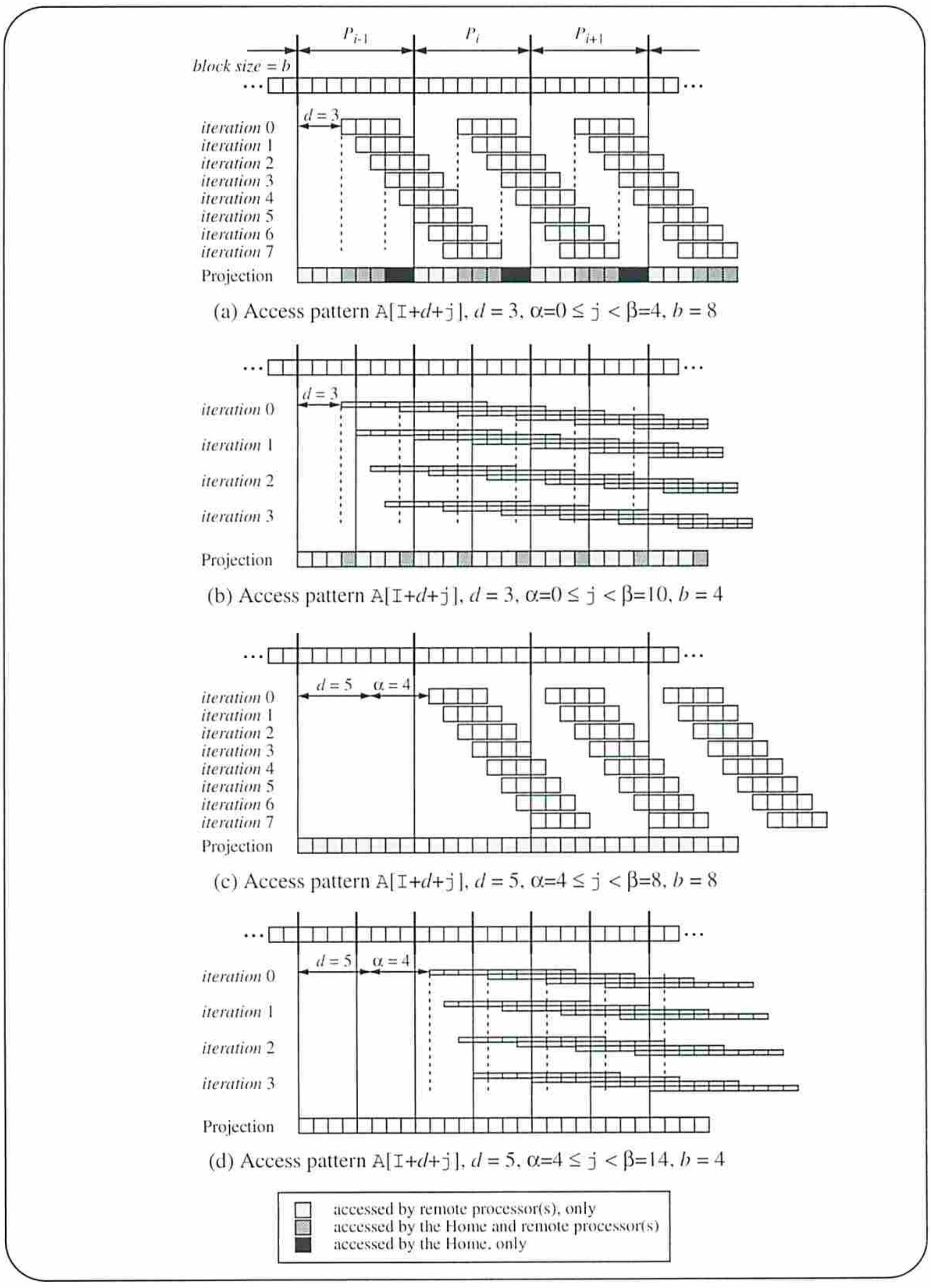
Partition Method	Conditions	Data Set Size	Number of Processors
Cyclic	-	$O(N)$	-
Block-Cyclic	$d < b$	$O(N)$	-
	$d \geq b$	$O(N)$	$O((P\text{-}1) / P)$
Block	$d < b$	$O(N)$	$O(P)$
	$d \geq b$	$O(N)$	-

Table 9. Effects of data set size and number of processors in $E_{VIC\text{-}S}$.

4.3.2 Multiple Element Vicinity Index Expression

4.3.2.1 Block-Cyclic Partitioning

An index expression for multiple vicinity, $E_{VIC\text{-}M}$, is in the form of `A[I+`d`+j]`, where `I` is the FORALL loop variable, d is the access distance of $E_{VIC\text{-}S}$, and `j` is a variable with a range of $\alpha \leq$ `j` $< \beta$ (Figure 17 (b)). The sharing pattern of $E_{VIC\text{-}M}$ is changing according to various conditions; $d + \alpha < b$ and $|\beta - \alpha| < b$, $d + \alpha < b$ and $|\beta - \alpha| \geq b$, $d + \alpha \geq b$ and $|\beta - \alpha| < b$, $d + \alpha \geq b$ and $|\beta - \alpha| \geq b$. The reason for the distinction is as follows.



(a) Access pattern A[I+d+j], $d = 3$, $\alpha=0 \leq$ j $< \beta=4$, $b = 8$

(b) Access pattern A[I+d+j], $d = 3$, $\alpha=0 \leq$ j $< \beta=10$, $b = 4$

(c) Access pattern A[I+d+j], $d = 5$, $\alpha=4 \leq$ j $< \beta=8$, $b = 8$

(d) Access pattern A[I+d+j], $d = 5$, $\alpha=4 \leq$ j $< \beta=14$, $b = 4$

Figure 18. Data-sharing in multiple vicinity index expression.

Basically, the value of $d + \alpha$ determines whether a processor starts its execution of the FORALL loop by accessing its own array element ($d + \alpha < b$, Figure 18 (a) and (c)) or not ($d + \alpha \geq b$, Figure 18 (b) and (d)). The value of $|\beta - \alpha|$ provides several meanings. First, while an array element is accessed only once in the case of $E_{VIC\text{-}S}$, it is accessed $|\beta - \alpha|$ times in the $E_{VIC\text{-}M}$ case. Second, in $E_{VIC\text{-}S}$, if a processor executes computations on its b element partition block, it accesses another set of b consecutive elements that are d elements away from its own partition block. By contrast, in $E_{VIC\text{-}M}$, a processor accesses $b + |\beta - \alpha|$ consecutive elements which are $d + \alpha$ elements away from its own block (Figure 18). Third, while an array element is accessed by one processor in $E_{VIC\text{-}S}$, it may be accessed by multiple processors in $E_{VIC\text{-}M}$. If $|\beta - \alpha| < b$, the first and last $|\beta - \alpha|$ elements among $b + |\beta - \alpha|$ elements that are accessed by a processor during the execution on its b element block are accessed by two processors and other elements in the middle are accessed by a single processor. In addition, if $d + \alpha < b$ and $|\beta - \alpha| < b$, merely $b - (d + |\beta - \alpha|)$ out of $b + |\beta - \alpha|$ are accessed by the Home, only. In consequence, $d + |\beta - \alpha|$ elements per partition block are involved in data-sharing in this case. However, if $|\beta - \alpha| \geq b$, all elements are accessed by multiple processors and the number of processors accessing an array element is either $\lceil |\beta - \alpha| / b \rceil$ or $\lceil |\beta - \alpha| / b \rceil + 1$.

It is very important to note that the access order of an array element by multiple processors is not strictly well-arranged as shown in the figures. It is because the processors execute program instructions not in lock-step fashion. Therefore, $|\beta - \alpha|$ accesses to an element by multiple processors are assumed to be made in an arbitrarily interleaved manner. In this situation, each element yields $O(|\beta - \alpha|)$ data-sharing. In addition, even when $d + \alpha \geq b$, a processor may start execution of the FORALL loop by accessing its own array element, and even when $|\beta - \alpha| \geq b$, array elements may be accessed by their Home. This is due to the periodicity in block-cyclic partitioning.

Summarily, when $d + \alpha < b$ and $|\beta - \alpha| < b$, $d + |\beta - \alpha|$ elements per partition block are involved in data-sharing. In other cases, all elements are involved in data-sharing. So, the total number of elements involved in data-sharing is always *O(N)*. In addition, each element is involved in data-sharing $|\beta - \alpha|$ times so that the total amount of data-sharing is $O(N) \times O(|\beta - \alpha|)$. Finally, when $d + \alpha < b$ and $|\beta - \alpha| < b$ (Figure 18 (a)), the number of processors has no impact. Otherwise, due to the periodicity of block-cyclic partitioning, the amount of data-sharing is proportional to $O((P-1)/P)$ (Figure 18 (b), (c), (d)).

4.3.2.2 Cyclic Partitioning

Cyclic partitioning is a special case of block-cyclic partitioning with $b = 1$. Given d, α and β in $E_{VIC\text{-}M}$, $d + \alpha \geq b$ and $|\beta - \alpha| \geq b$ are always true. Therefore, all elements become involved in data-sharing so that the number elements involved in data-sharing is *O(N)*. Each element is accessed $|\beta - \alpha|$ times as in the block-partitioning case so that $O(|\beta - \alpha|)$ data-sharing is observed per array element. It is known that the cyclic partitioning is also periodic like block-cyclic partitioning. This results in the effect of the number of processors to be $O((P-1)/P)$. In contrast to the block-cyclic partitioning where an element might be accessed several times by the same processor (Figure 18), all accesses to an element in the cyclic partitioning case are made by distinct processors if $|\beta - \alpha| \leq P$. If $|\beta - \alpha| > P$, every processor accesses an element $O(|\beta - \alpha|/P)$ times.

4.3.2.3 Block Partitioning

Block partitioning is another special case of block-cyclic partitioning with $b = N/P$. Data-sharing patterns in this case are almost identical to those in block-cyclic partitioning. The only difference is the scaling effects on the amount of data-sharing.

When $d + \alpha < b$ and $|\beta - \alpha| < b$ in block-cyclic partitioning, *d* elements per block are engaged in data-sharing. In block partitioning, there are *P* partition blocks so that the amount of data-sharing is proportional to *O*(*P*), regardless of data set size. In all other cases ($d + \alpha < b$ and $|\beta - \alpha| \geq b$, $d + \alpha \geq b$ and $|\beta - \alpha| < b$, $d + \alpha \geq b$ and $|\beta - \alpha| \geq b$), data-sharing is observed on every element. The amount of data-sharing is *O*(*N*). The effect of the number of processors on data-sharing is also *O*(*P*). The reason is as follows. We know that an element is accessed $|\beta - \alpha|$ times and the number of processors accessing an array element is $\lceil |\beta - \alpha| / b \rceil$ or $\lceil |\beta - \alpha| / b \rceil + 1$. As the number of processors increases, the block size shrinks and the number of processors accessing an element increases.

4.3.2.4 Summary

The effects of data set size and number of processors on data-sharing in $E_{VIC\text{-}M}$ are summarized in Table 10. Note that a single array element presents $O(|\beta - \alpha|)$ data-sharing.

Partition Methods	Conditions	Data Set Size	Number of Processors	Data-sharing per element
Cyclic	-	$O(N)$	$O((P\text{-}1) / P)$	$O(|\beta - \alpha|)$
Block-Cyclic	$d + \alpha < b$, $|\beta - \alpha| < b$	$O(N)$	-	
	$d + \alpha < b$, $|\beta - \alpha| \geq b$	$O(N)$	$O((P\text{-}1) / P)$	
	$d + \alpha \geq b$, $|\beta - \alpha| < b$	$O(N)$	$O((P\text{-}1) / P)$	
	$d + \alpha \geq b$, $|\beta - \alpha| \geq b$	$O(N)$	$O((P\text{-}1) / P)$	
Block	$d + \alpha < b$, $|\beta - \alpha| < b$	-	$O(P)$	
	$d + \alpha < b$, $|\beta - \alpha| \geq b$	$O(N)$	$O(P)$	
	$d + \alpha \geq b$, $|\beta - \alpha| < b$	$O(N)$	$O(P)$	
	$d + \alpha \geq b$, $|\beta - \alpha| \geq b$	$O(N)$	$O(P)$	

Table 10. Effects of data set size and number of processors in $E_{VIC\text{-}M}$.

4.3.3 Single Element Stranger Index Expression

As pointed earlier in Section 4.2.2.2, no distinction between $E_{STR\text{-}S}$ and $E_{STR\text{-}M}$ is necessary for the purposes of sharing pattern analysis. It has been mentioned many times that the accesses to array elements with E_L cause no data-sharing. So, the discussion will start with E_C.

Given E_C, Figure 13 (b) shows only one element is selected regardless of the data set size and it is accessed by all processors. The amount of data-sharing is thus proportional to the number of processors, or $O(P)$. In addition, once an element is selected, it is accessed N times within the FORALL loop so that it presents $O(N)$ data-sharing.

For E_{LIC} and E_{LIR}, only P elements are accessed regardless of the data set size. The probability that an element is selected by its Home is $1/P$ so that the amount of data-sharing is proportional to $O((P-1)/P)$. That is to say, the overall effect of the number of processors is $O((P-1))$. In addition, as in E_C, once an element is selected, it is accessed $O(N)$ times within the FORALL loop. So, each selected element shows $O(N)$ data-sharing.

In case of E_{LVC} and E_{LVR}, on the average, it is very likely that every element is accessed once through the loop. With P processors given in the system, the probability that a certain element is accessed by the Home is $1/P$. Therefore, the amount of data-sharing is proportional to $O(N)$ and $O((P\text{-}1) / P)$.

The following table summarizes the discussion made in this section. Note that in E_C, E_{LIC} and E_{LIR}, only one or P array elements are accessed while, in E_{LVC} and E_{LVR}, all elements are accessed.

4.3.4 Multiple Element Stranger Index Expression

The multiple element stranger index expression, $E_{STR\text{-}M}$, allows a processor to first select an element with corresponding E_{STR}, then, to access it and its $|\beta - \alpha|$ neighbors. The

Index Expression Types	Data Set Size	Number of Processors	Data-sharing per element ($E_{STR\text{-}M}$, only)
E_L	-	-	$O(\lvert\beta - \alpha\rvert)$
E_C	$O(N)$	$O(P)$	
E_{LIC}	$O(N)$	$O(P\text{-}1)$	
E_{LIR}	$O(N)$	$O(P\text{-}1)$	
E_{LVC}	$O(N)$	$O((P\text{-}1) / P)$	
E_{LVR}	$O(N)$	$O((P\text{-}1) / P)$	

Table 11. Effects of data set size and number of processors in $E_{STR\text{-}S}$ and $E_{STR\text{-}M}$.

selection method of the first element, the basic sharing patterns, and the effects of data set size and number of processors on the amount of data-sharing are all the same as the corresponding index expressions in Section 4.3.3 and Table 11.

The only difference is that $|\beta - \alpha|$ times more elements are selected in $E_{STR\text{-}M}$ than in $E_{STR\text{-}S}$. Thus, for E_C, E_{LIC} and E_{LIR}, the number of accessed elements grows by factor of $|\beta - \alpha|$. For E_{LVC} and E_{LVR}, on the other hand, all elements are accessed $|\beta - \alpha|$ times so that every element yields $|\beta - \alpha|$ data-sharing.

4.3.5 Summary

Thus far, we have seen the sharing patterns and the effects of the data set size and the number of processors on the amount of data-sharing for all kinds of array index expressions. All works have been possible owing to the fundamental inherent nature of SPMD programs (Observations 1 to 5). Especially, the properties of data homogeneity and FORALL loop-based structure of SPMD programs are invaluably useful in performing data-sharing analysis in this study. In this section, before closing the sharing patterns analysis based on array index expression analysis, we make a few additional comments.

First of all, the difference between array elements designated by compile time variables and by run time variables is the determinism of their memory locations. Obviously, describing their sharing behaviors as they are accessed by *particular* or *arbitrary* processors is useful in many aspects of sharing analysis. Nevertheless, in measuring their data-sharing amounts, we see no difference at all as shown in Table 11. For this reason, no distinction will be imposed on them in the rest of this paper. Consequently, the array expressions of E_{LIC} and E_{LIR} are merged into E_{LI}, and E_{LVC} and E_{LVR} into E_{LV}.

As mentioned many times, one of the most useful properties in SPMD programs is data homogeneity. That claims every element should experience the same computation within the same FORALL loop. In this regard, the access patterns of array elements with E_C and E_{LI} (E_{LIC} and E_{LIR}) violate the property of data homogeneity. In fact, such access patterns are quite rarely found in practical applications. Especially in SPLASH and SPLASH-2 benchmark suites, no application presenting such access patterns is found. Additionally, since $N >> P$ due to the meaningfulness of data set size (Section 4.2), the amount of data-sharing observed on those elements that are selected and accessed by E_C and E_{LI} seems to be negligible as compared to total amount of data-sharing. For this reason, and to make further analysis simple, their consideration will be put aside in subsequent sections. Consequentially, the index expressions are finally classified into E_L, E_{LV}, and E_{VIC}. Among them, E_{LV} and E_{VIC} are subdivided into $E_{LV\text{-}S}$, $E_{LV\text{-}M}$, $E_{VIC\text{-}S}$, and $E_{VIC\text{-}M}$.

Up to now, we have investigated the amount of data-sharing incurred by each type of index expressions. The data-sharing is the source of diverse coherency events defined by the underlying architecture. One of the popular coherency events is the cache miss. In Section 5, E_L, $E_{LV\text{-}S}$, $E_{LV\text{-}M}$, $E_{VIC\text{-}S}$, and $E_{VIC\text{-}M}$ are investigated further to see how their sharing characteristics influence the number of cache misses.

In Sections 4.3.2 and 4.3.4, when $E_{VIC\text{-}M}$ or $E_{STR\text{-}M}$ is used, it was shown that an array element is accessed $|\beta - \alpha|$ times and the accesses are normally made by distinct processors in interleaved manner. This may or may not result in the total amount of data-sharing multiplied by the factor of $|\beta - \alpha|$ depending on whether the access type is a read or write operation. Let's consider the case where the access is a read operation. If an element has been modified by remote processor(s) during the most recent access, the first access should invoke a coherency transaction to obtain an up-to-date data copy. Subsequent accesses, however, will not cause any coherency transaction because the first copy has brought in the valid copy already. Therefore, multiple accesses of the read operation do not influence the amount of data-sharing regardless of the size of $|\beta - \alpha|$. In contrast, if the access is a write operation, each access will cause coherency transactions to first obtain a valid data copy, before the write operation, which has been modified by another processor executing the same computation in the same FORALL loop, and to propagate the modified data copy, after the write operation[1]. That is to say, multiple accesses of the read operation increases the amount of data-sharing by factor of $|\beta - \alpha|$.

Finally, $|\beta - \alpha|$ is often a function of the data set size. Then, if the access type is a write operation, the effect of the data set size in Tables 10 and 11 is multiplied by $|\beta - \alpha|$.

4.4 Multi-dimensional Arrays

So far, we have presented analysis of all possible sharing patterns for one-dimensional shared arrays. In practice, arrays in scientific SPMD applications are n-dimensional, where n is an arbitrary positive integer. Each dimension of an n-dimensional array may be partitioned in cyclic, block-cyclic or block manner. In addition, for certain dimension(s), no partitioning may be applied. Figure 8 shows some examples of data partitioning of two-dimensional arrays. This section is dedicated to the sharing pattern analysis of n-dimensional arrays.

Before starting the discussion, let's consider a case where a particular dimension of an n-dimensional array is not partitioned. The examples are in Figure 8 (d), (e) and (f).

1. The types of required coherency transaction are variously defined according to the predefined cache coherency protocol.

In this case, no matter which types of index expressions may be given for the dimension, there is no data-sharing. This is equivalent to the case where an expression type for the dimension is E_L, regardless of the partitioning methods in that there is no data-sharing taking place.

In the following, sharing pattern analysis for two-dimensional arrays will be made first. Then, the sharing patterns in n-dimensional arrays will be deduced based on the knowledge earned from the one- and two-dimensional cases.

4.4.1 Two-dimensional Array

The basic procedure of sharing analysis is as follows.

Step 1: Perform sharing pattern analysis to obtain sharing amount for one row.
Step 2: Having N rows, multiply the sharing amount obtained in Step 1 by N.
Step 3: Perform sharing pattern analysis to obtain sharing amount for one column.
Step 4: Having N columns, multiply the sharing amount obtained in Step 3 by N.
Step 5: Sum up the amounts of data-sharing obtained in Step 2 and Step 4.

This way of sharing analysis is valid regardless of the data partitioning methods. In addition, it is also valid for both singular and multiple element index expressions. In Steps 1 and 3, the data set size is N as before, but the number of processors varies according to the partitioning methods applied to each dimension. For example, in block-block partitioning (Figure 8 (g)), there are $\sqrt{P}$ processors in a dimension.

Recall that the index expression types we have are E_L, E_{VIC}, and E_{LV}. For two-dimensional array A of size $N \times N$, possible combinations of index expressions are A[E_L][E_L], A[E_L][E_{VIC}], A[E_{VIC}][E_{VIC}], A[E_L][E_{LV}], A[E_{VIC}][E_{LV}] and A[E_{LV}][E_{LV}], where the order of index expressions in dimensions is not important. We consider them, one by one, for the case when A is partitioned in a block-cyclic manner in both dimensions. For better discussion, assume we have a one-dimensional array B of size N.

4.4.1.1 E_L and E_L

Since all elements are accessed by their Home in both dimensions, no data-sharing is observed.

4.4.1.2 E_L and E_{VIC}

As an example, notice the sharing pattern of `A[I][J+1]` shown in Figure 19 (a). The sharing pattern is observed as there are identical *N* rows each of which has the same sharing pattern as `B[J+1]`. At first, we measure the amount of data-sharing in a row, by dealing with the row as a one-dimensional array. The data-sharing analysis for one-dimensional array was described in Section 4.3. The only difference is that *N* elements in a row are distributed among $\sqrt{P}$ processors. In the next step, we expand the number of rows to *N* by multiplying the amount of data-sharing obtained in the first step by *N*.

4.4.1.3 E_{VIC} and E_{VIC}

Figure 19 (b) shows the sharing pattern of `A[I+1][J+1]`. This situation can be understood that the data-sharing is the union of the data-sharing caused by `A[I][J+1]` and by `A[I+1][J]`. For each of them, by repeating the analysis procedure in Section 4.4.1.2, the amount of data-sharing can be obtained. The sum is the total amount of data-sharing in `A[I+1][J+1]`.

Note that, although the elements filled with black are counted twice, their amount is negligible especially when expressed in big-oh notation.

4.4.1.4 E_L and E_{LV}

Figure 19 (c) is for `A`[E_L][E_{LV}]. There are *N* rows each of which has the same sharing pattern as `B`[E_{LV}], while no data-sharing is observed in the vertical direction. As in Section 4.4.1.2, the sharing analysis is first done for a row by assuming an *N* element one-

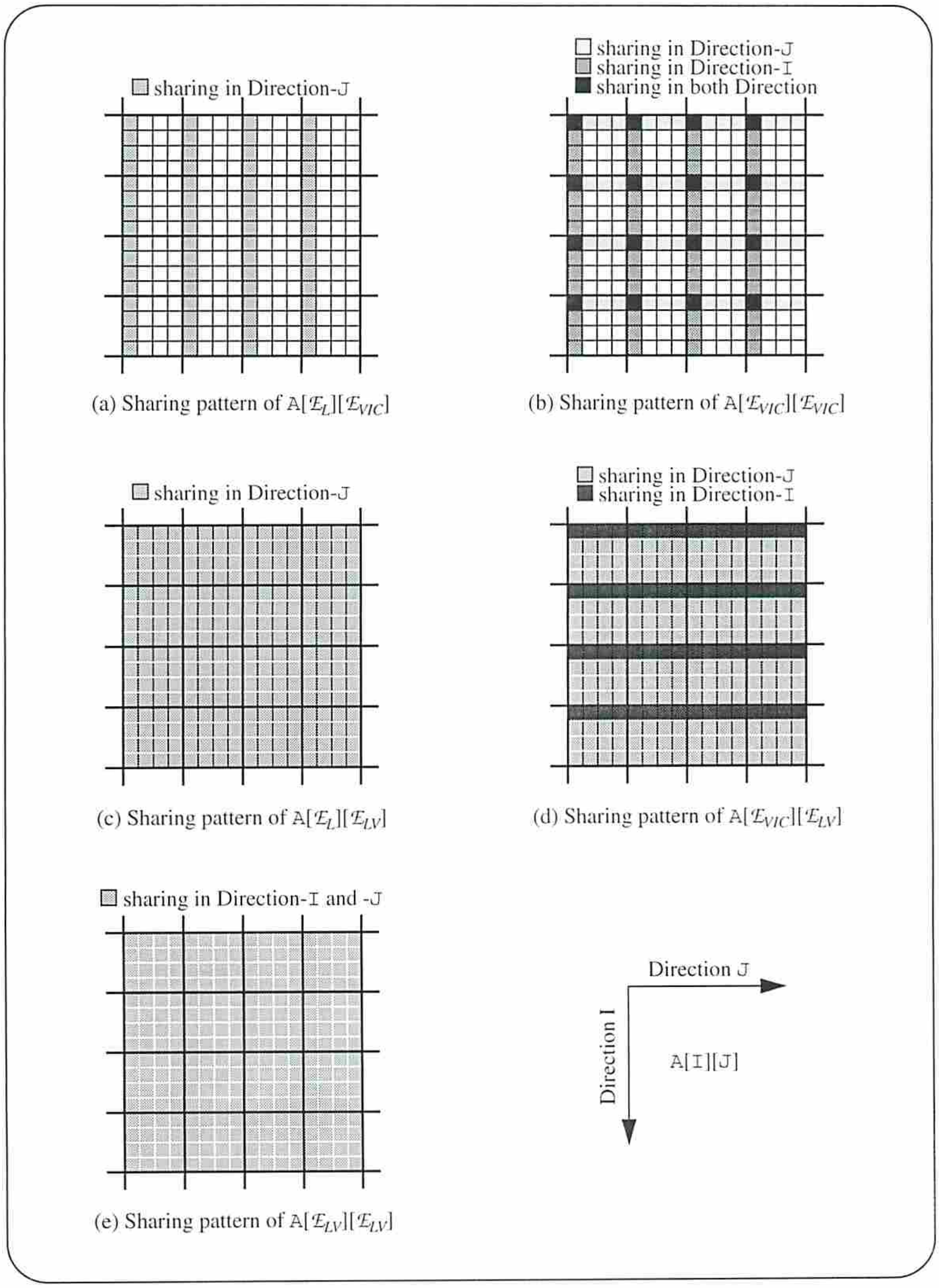

▨ sharing in Direction-J
(a) Sharing pattern of A[$\mathcal{E}_L$][$\mathcal{E}_{VIC}$]
□ sharing in Direction-J
▨ sharing in Direction-I
■ sharing in both Direction
(b) Sharing pattern of A[$\mathcal{E}_{VIC}$][$\mathcal{E}_{VIC}$]
▨ sharing in Direction-J
(c) Sharing pattern of A[$\mathcal{E}_L$][$\mathcal{E}_{LV}$]
▨ sharing in Direction-J
■ sharing in Direction-I
(d) Sharing pattern of A[$\mathcal{E}_{VIC}$][$\mathcal{E}_{LV}$]
▨ sharing in Direction-I and -J
(e) Sharing pattern of A[$\mathcal{E}_{LV}$][$\mathcal{E}_{LV}$]
Direction J
Direction I
A[I][J]

Figure 19. Data-sharing in two-dimensional arrays.

dimensional array is distributed among $\sqrt{P}$ processors. Complete data-sharing analysis for E_{LV} is found in Section 4.3. Then, the amount of data-sharing obtained earlier is multiplied by N.

4.4.1.5 E_{VIC} and E_{LV}

Figure 19 (d) presents the sharing pattern observed when a two-dimensional array is indexed as A[E_{VIC}][E_{LV}]. At first, for the horizontal direction, data-sharing analysis for E_{VIC} is made as in Section 4.4.1.4. Then, for the vertical direction, the methodology in Section 4.4.1.2 is applied.

Normally, the data-sharing analysis for a dimension of multi-dimensional arrays is performed as described in Section 4.3 where one-dimensional arrays are considered. The only difference between one-dimensional arrays and multi-dimensional arrays is the number of processors involved. For example, it was pointed earlier that $\sqrt{P}$ processors access the elements in a row or column of two-dimensional array which is partitioned in block-block manner (Figure 8 (g)).

However, if a certain dimension (dim_i) is indexed by E_{LV}, special attention must be paid when analyzing sharing patterns in the other dimension (dim_j). Assume that $f(P)$ ($=\sqrt{P}$, in this section) processors are invloved along the dim_i direction and $g(P)$ ($=\sqrt{P}$, too) processors are invloved along the dim_j direction. The index expression of type E_{LV} in dim_i direction means that $f(P)$ processors access any single element. Therefore, the number of processors invloved in the accesses to the elements in the dim_j direction is not $g(P)$ any longer; rather it is $f(P) \times g(P)$.

4.4.1.6 E_{LV} and E_{LV}

If both directions (dim_i and dim_j) are indexed by E_{LV}, the sharing analysis in each direction will be made by including P processors. Since, the dim_j is indexed by E_{LV}, the number of processor to be considered in the sharing analysis of *dim$_i$* direction is $\sqrt{P} \times \sqrt{P}$ for *N* elements. With *N* elements in the *dim$_j$* direction, the amount of data-sharing is multiplied by *N*. Vice versa is true for the analysis of *dim$_j$* direction. The total amount of data-sharing of two-dimensional array indexed by E_{LV} in both direction is proportional to $O(N^2)$ and $O(P)$.

4.4.2 Higher-dimensional Array

Now, from the knowledge gained from one- and two-dimensional arrays, the sharing analysis method for a general *n*-dimensional array is described in this section. The data set size in each dimension and the number of processors allocated to each dimension are assumed to be *N* and $f_i(P)$, where $0 \le i \le n-1$. The basic procedure of sharing analysis introduced in Section 4.4.1 is revised as follows.

For each dimension, *i*, $0 \le i \le n-1$, repeat Steps 1 and 2.

Step 1: For one dimension, *i*, perform data-sharing pattern analysis and obtain the amount of data-sharing as described in Section 4.3.

If there are other dimensions with E_{LV}, dim_{LV}, the number of processors involved in accesses to elements in dimension *i* is

$$f_i(P) \times \left(\prod_{k=0,\, k \ne i}^{n-1} f_k(P), \text{ if dimension } k \text{ is } dim_{LV} \right).$$

Step 2: Having *n* dimensions, multiply the amount of data-sharing obtained in Step 1 by N^{n-1}.

Step 3: Accumulate the amount of data-sharing obtained in Steps 1 and 2.

Finally, note that for the dimensions in which no data partitioning is applied, none of the index expressions result in any data-sharing.

Chapter 5

MODELING MISS RATES

In the event of a cache miss, a processor has to stall its current execution until the required data are brought into its local cache. The long latency in the data path including memory accesses and data block transfer via the interconnect is costly in terms of performance. As the speed of processors continually increases, the stall penalty that a processor has to suffer becomes relatively larger. Therefore, cache misses are regarded as one of the most important performance metrics that affect the overall execution time.

Given an application running on a specific architecture, the origins of cache misses are the inherent data-sharing in the application. Therefore, by understanding the sharing behavior, we should be able to estimate the number of cache misses when cache sizes are infinite. In this section, a systematic methodology is proposed to derive polynomial models for the number of cache misses and miss rates based on the sharing analysis. The models are expressed by numerical expressions in terms of data set size (*N*) and number of processors (*P*). The power of the model lies in the accurate prediction of cache misses and miss rates for large data sets and large number of processors which are usually difficult or even impossible to simulate.

The overall system looks as shown in Figure 20. The components are described in the following subsections.

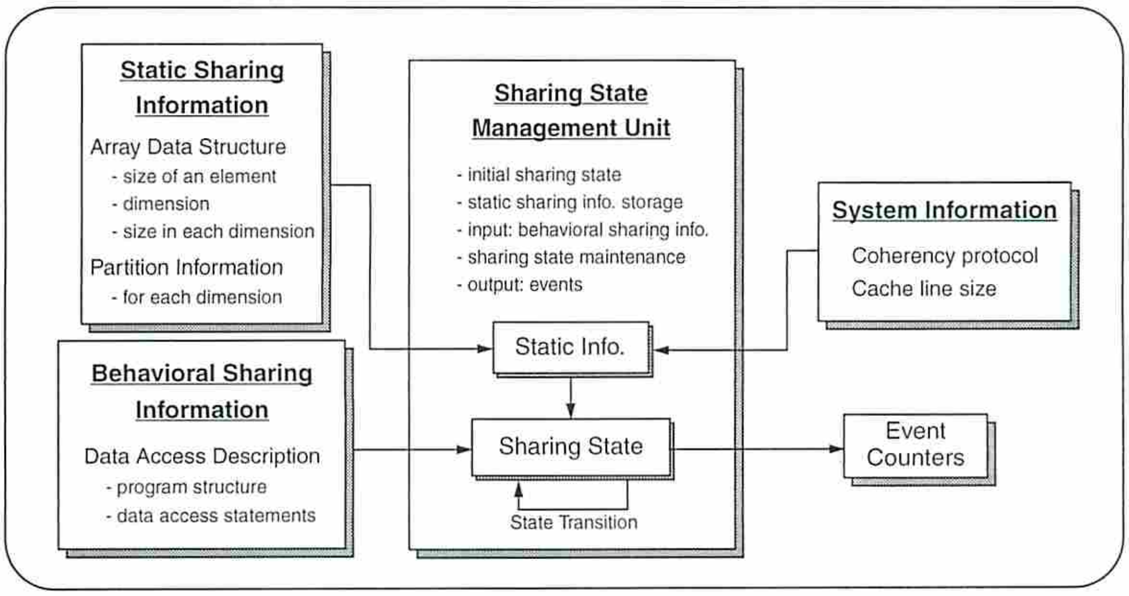


Figure 20. Overview of miss rate modeling system.

5.1 System Information

The first component includes the information related to the underlying architecture. From an architectural viewpoint, the cache coherency protocol to governs sharing state transitions activated by incoming data accesses. The occurrences of relevant events such as cache misses are also determined by the coherency protocol. Another architectural factor needed in measuring cache misses is the cache block size. In this study, the three state write invalidate protocol is adopted and the cache block size is32 bytes.

5.2 Static Sharing Information

On the application side, the static sharing information is related to the shared data structures. The static sharing information includes the size of each array element, the number of dimensions, and the number of elements in each dimension of shared arrays. In addition, the partitioning methods applied to each dimension of shared arrays is also provided. Usually, many shared arrays are used in a program. They are frequently of different sizes with different partitioning schemes and they present different sharing behaviors. The unique characteristics of shared arrays will result in different miss rate models. For this reason, in this study, the shared arrays are dealt with separately.

Among static information, the size of an element, the number of dimensions and the number of elements of an array are given in the declaration area of applications. Data partitioning information is found at the top portion of parallel applications before the parallel section is started. Along with the system information, static sharing information is stored inside the sharing state management unit. They are both used in controlling the sharing state transitions and in measuring the number of coherency events of particular interest.

5.3 Behavioral Sharing Information

The behavioral sharing information of applications is related to the shared data accesses in the computation body of applications. Its role is to drive the sharing state management unit to initiate the sharing state transitions. Being used as an input to the sharing state management unit, the behavioral sharing information is syntactically well-defined as will be shown later.

Given a scientific problem, the algorithm to solve it generally consists of a sequence of numerical equations. According to the equations, required computations are executed on proper data objects. Since the data structures are in array format (Observation 2) in SPMD programs, the application of a common computation on array elements (Observation 3) is implemented by `for` loops (Observation 4). That is, an SPMD program is another form of a sequence of numerical equations represented by a sequence of `for` loops.

In view of data-sharing analysis, a `for` loop construct is regarded as an execution unit whose sharing pattern is represented by a description statement which will be introduced shortly. As discussed in Section 4, the array index expressions within FORALL loops can be used to describe the sharing patterns of shared arrays. Consider a FORALL loop in an SPMD application and the program statements within the loop given in Figure 21, where x is a loop-variant variable.

```
1. FORALL (I = 0 .. N) {
2.     x = ( A[I+1] + B[I+2] / 2 ) ;
3.         C[I] = D[x] ;
4. }
```

Figure 21. Example FORALL loop and data accesses.

The index expression for individual arrays are specified as follows:

- for array A, index is $E_{VIC\text{-}S}$, $d = 1$, $\alpha = 0$, $\beta = 0$ and the operation is *read*,
- for array B, index is $E_{VIC\text{-}S}$, $d = 1$, $\alpha = 0$, $\beta = 0$ and the operation is *read*,
- for array C, index is E_L, and the operation is *write*, and
- for array D, index is $E_{LV\text{-}S}$, variable is x, $\alpha = 0$, $\beta = 0$, the operation is *read*.

Recall that shared arrays are dealt with separately in data-sharing analysis. We now define the syntax for behavioral sharing information description as follows.

$$Expression\_Type\text{: } d,\ label,\ \alpha,\ \beta,\ op\_type,$$

where d is used only for E_{VIC}, *label* for E_{LV}, α and β for multiple element index expressions. When no value is specified, the field is omitted. In summary, the syntax of each type of array index expressions is given in Table 12. The fields of d, *label*, α, β and *op_type* should be substituted by its proper values accordingly. The d, α, and β are integer numbers, *label* is character string and *op_type* is either *read* or *modify*.

The resulting behavioral sharing information description for the program segment given in Figure 21 is shown in Figure 22. Note that the FORALL loop statement (line 1 in

Expressions	Syntax
E_L	E_L: *op_type*
$E_{VIC\text{-}S}$	$E_{VIC\text{-}S}$: *d, op_type*
$E_{VIC\text{-}M}$	$E_{VIC\text{-}M}$: *d*, α, β, *op_type*
$E_{LV\text{-}S}$	$E_{LV\text{-}S}$: *label, op_type*
$E_{LV\text{-}M}$	$E_{LV\text{-}M}$: *label*, α, β, *op_type*

Table 12. Syntax of behavioral sharing information description.

Figure 21) is not shown in the behavioral sharing information description. It is because each statement in Figure 22 implicitly presents the data-sharing pattern within the FORALL loop. The behavioral sharing information presents the access pattern of a shared array. So, the behavioral sharing information is called the *access pattern description* (APD, in short) and the statement in the APD is the *access pattern description statement* (APDS, in short). These two terms will be used in the rest of the thesis.

In practical applications, there are more complicated numerical equations, array index expressions, or data-sharing patterns, which are discussed further in the following subsections.

$E_{VIC\text{-}S}$: 1, Read

(a) For array A

$E_{VIC\text{-}S}$: 1, Read

(b) For array B

E_L: Modify

(c) For array C

$E_{LV\text{-}S}$: x, Read

(d) For array D

Figure 22. Behavioral sharing information description for Figure 21.

5.3.1 Multiple Index Expressions of One Shared Array

In Figure 21, each shared array is used only once. However, in many cases, a shared array may be accessed many times within a FORALL loop as shown in Figure 23 (a). The resulting APD is given in Figure 23 (c). Note that the access types of three index expressions are all reads.

```
FORALL (I = 0 .. N) {
    x = ( A[I-1] + A[I] + A[I+1] ) / 3 ;
       :  :  :
}
```

(a) Program segment (example 1)

```
FORALL (I = 0 .. N)
    a = A[I-1] ;
FORALL (I = 0 .. N)
    b = A[I] ;
FORALL (I = 0 .. N)
    c = A[I+1] ;
```

(b) Program segment (example 2)

$E_{VIC\text{-}S}$: -1, Read

E_L: Read

$E_{VIC\text{-}S}$: 1, Read

(c) Behavioral sharing description

Figure 23. Multiple index expressions of one shared array in a FORALL loop.

One natural problem regarding the APD is that the program codes in Figures 23 (a) and (b) would result in identical APDs. Obviously, the computations given in the two program examples are different from each other. However, for the measurement of cache misses (or, other data coherency events), they both yield the same result. In the following, we will show that the programs in Figure 23 (a) and (b) produce the same magnitude order of the number of cache misses.

Cache misses are classified into cold misses and invalidation misses. First, we consider the cold misses. Assume the program statements above are executed at the very beginning of the program. Figure 24 illustrates data accesses made by a processor, P_i, on array elements which are allocated to it. The heavily shaded accesses denote the accesses experiencing cold misses while the lightly shaded accesses denote cache hits. Figure 24 (a) is the result for the program in Figure 23 (a) and Figure 24 (b) is for Figure 23 (b). We see that the number of cold cache misses are the same in both cases.

In addition, assume there has been a write operation so that some of the elements accessed by P_i are modified by remote processors. The black dots in Figure 25 (a) denote such element. When executing program segments in Figure 23 (a) and (b), P_i experiences the same number of invalidation misses as shown in Figure 25 (a) and (b). In consequence, the APD for the multiple read operations to the same array within a FORALL loop can be written in the formats shown in Figure 23 (c) without causing any differences in the number of cache misses.

Now, consider the case when one of the multiple accesses to a shared array within a FORALL loop is a write operation. The example code is given in Figure 26 (a) and the accesses by P_{i-1}, P_i and P_{i+1} to the array are shown in Figure 26 (b). Attention should be paid to the elements filled in black. They are the ones to be accessed by two processors. At first glance, the program line 2 does not seem to cause any problem. However, since SPMD programs are *not* executed *in lock-step* manner, the sequence of computations to be performed to a certain element is not deterministic. The tagged element e_i is supposed to be modified and read by P_i, and then read by P_{i-1}. But in reality, the order of the access sequence is by no means guaranteed. For example, if P_i experiences a page fault before

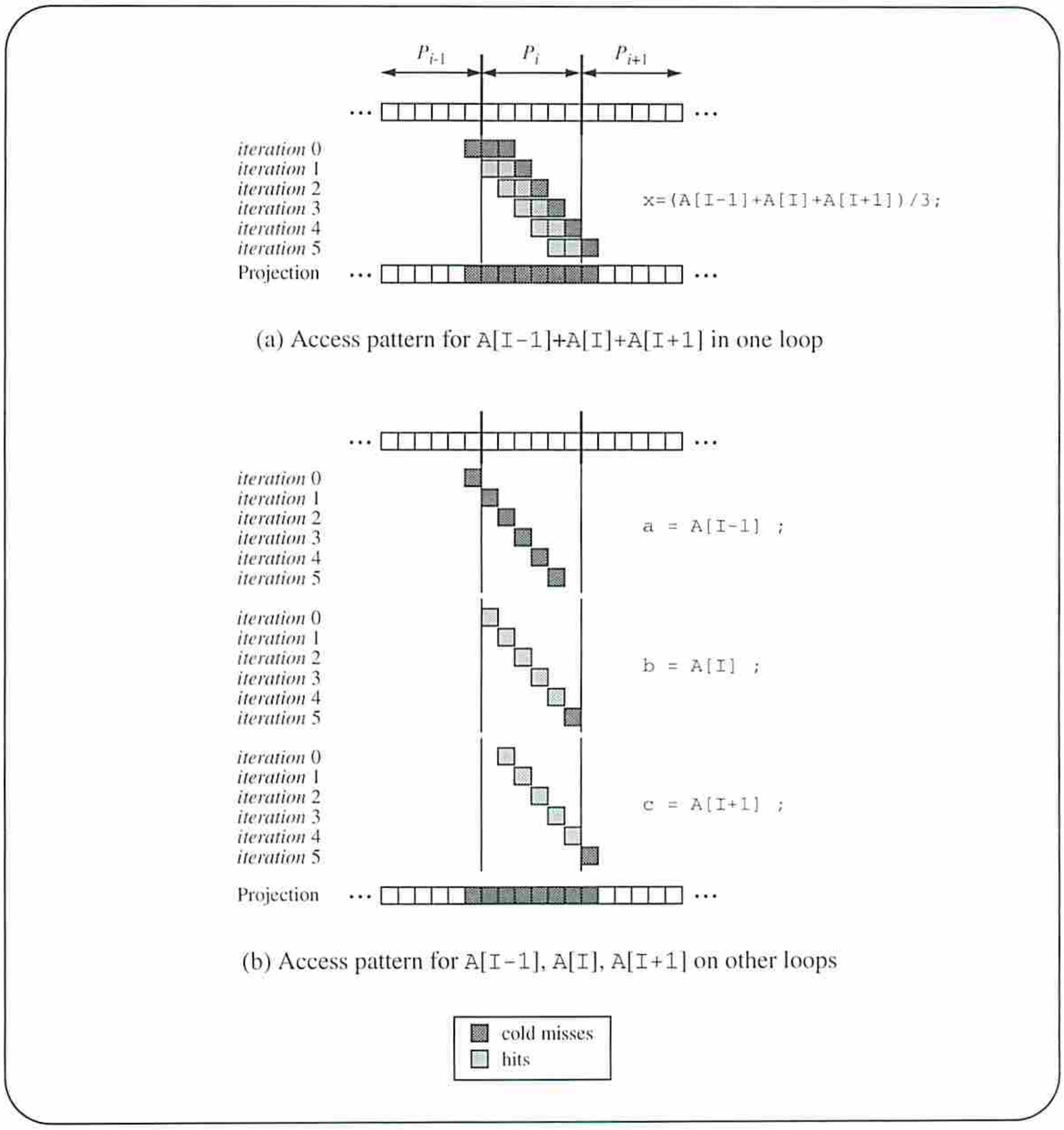
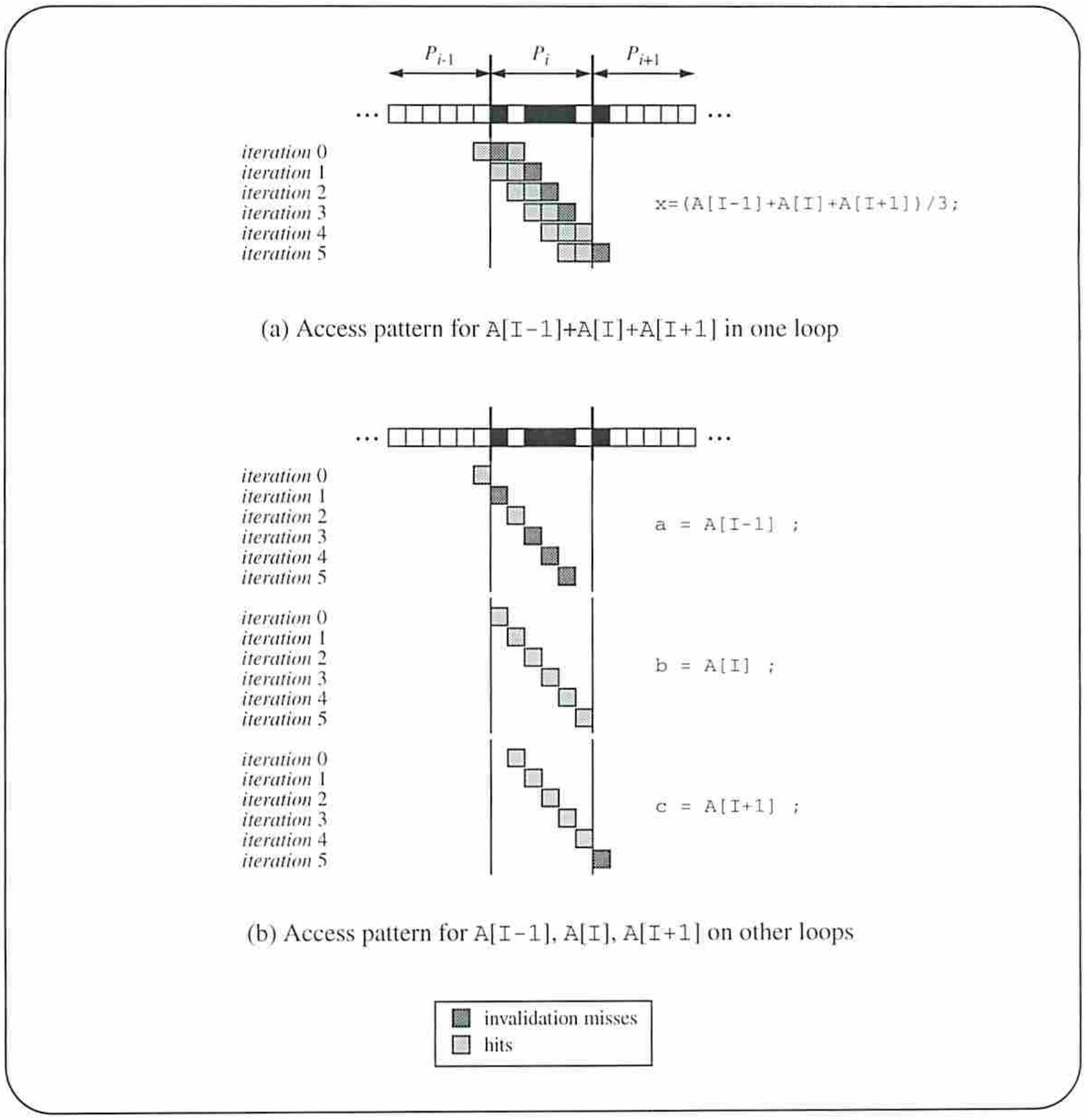


Figure 24. Access patterns for multiple array accesses.

accessing e_i, e_i may be accessed by P_i; this is not what the programmer or algorithm designer expects. In real life, to avoid this problem, a temporary array (B) of the same size is adopted to store the computation results as shown in line 2 of Figure 27 (a). Then, array B may be used in subsequent program statements as in line 3 or the results are copied back to array B for further computation as in line 6. The APDS in Figures 27 (b) and (c) will be



(a) Access pattern for `A[I-1]+A[I]+A[I+1]` in one loop

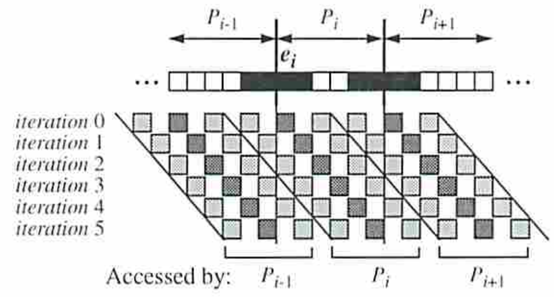
(b) Access pattern for `A[I-1]`, `A[I]`, `A[I+1]` on other loops

Figure 25. Access patterns for multiple array accesses.

derived. An alternative is to impose synchronization primitives such as *lock*, *pause* or *barrier*. This method usually suffers from synchronization overhead, which seriously degrades overall performance. This is why the former scheme is preferred by programmers. In other words, within a FORALL loop, no element of a shared array is accessed by multiple processors if at least one of the access types is a write operation.

```
FORALL (I = 0 .. N) {
    A[I] = ( A[I-2] + A[I+2] ) / 2 ;
          :  :  :
}
```

(a) Program segment



(b) Access pattern

■ write operation
□ read operation

Figure 26. Access patterns for multiple array accesses.

Nevertheless, if there is a FORALL loop where multiple processors are accessing some shared array elements and at least one of the accesses is a write operation, the following approach is taken. As was shown in Figure 26 (b), the read accesses of a processor, P_i, may be made either before or after another processor, P_{i-1}, executes its write operation. Then, the read by P_i may observe either an invalidation miss or a cache hit. Assume that the largest number of possible cache misses is $O(f_1(N))$ and $O(g_1(P))$, and the largest number of possible cache hits is $O(f_2(N))$ and $O(g_2(P))$, where N is the data set size and P is the number of processors. Further assume that the probability that P_i experiences an invalidation miss is p and the probability that P_i experiences a cache hit is $1 - p$. The numbers of cache misses and cache hits that are measured during the execution of the pro-

```
FORALL (I = 0 .. N) {
    B[I] = ( A[I-2] + A[I+2] ) / 2 ;
    x = B[I] ... ;
}
FORALL (I = 0 .. N) {
    A[I] = B[I] ;
}
```

(a) Program segment

$E_{VIC\text{-}S}$: -2, Read

$E_{VIC\text{-}S}$: 2, Read

E_L: Write

(b) Behavioral sharing description of array A

E_L: Write

E_L: Read

E_L: Read

(c) Behavioral sharing description of array B

Figure 27. Multiple index expressions of one shared array in a FORALL loop.

gram are then expressed as $p \times O(f_1(N))$ and $p \times O(g_1(P))$, and $(1-p) \times O(f_2(N))$ and $(1-p) \times O(g_2(N))$, respectively. Since the access order is independent of the data set size or the number of processors, p and $1-p$ are not functions of N or P. Therefore, since these expressions should eventually be given only in terms of N and P, the terms p and $1-p$ will disappear. As a result, we will have $O(f_1(N))$ and $O(g_1(P))$ cache misses, and $O(f_2(N))$ and $O(g_2(P))$ cache hits.

The method above is called the *worst case estimation* of performance metrics and such estimation can be achieved by presenting the APD for Figure 26 (a) as in Figure 28. The first three lines correspond to line 2 of Figure 26 (a). The last two are for the worst case estimation where they will observe the largest number of cache misses.

$E_{VIC\text{-}S}$: -2, Read
$E_{VIC\text{-}S}$: 2, Read
E_L: Write
$E_{VIC\text{-}S}$: -2, Read
$E_{VIC\text{-}S}$: 2, Read

Figure 28. Access patterns for multiple array accesses.

5.3.2 `for` Loops

Consider the program segment in Figure 29. The loop statement in line 1 governs the index range of array `A`, while the loop statement in line 2 shows the index range of array `B`. We call the loops whose variables are used in designating array elements *IC loops* (*I*ndex *C*overing loops) and their control variables are called *IC variables*. When an *n*-dimensional array is involved in the computation, exactly *n IC* loops are nested. For example, the `I`-loop in Figure 29 is the *IC* loop with respect to `A` and the `K`-loop is the *IC* loop with respect to `B`. One *IC* loop is used to specify each element of `A` and another *IC* loop deals with `B`. Additionally, we call `A` the *IC* array of `I`-loop and `B` the *IC* array of `K`-loop.

The other type of loop is called *RC loops* (*R*epetitive *C*omputation loops). Given an array, all loops which are not *IC* loops with respect to a particular array are *RC* loops with respect to that array. For example, the `I`-loop is the *RC* loop with respect to `B` whose loop variable is `K`, and the `K`-loop is the *RC* loop with respect to `A` whose loop variable is `I`. The `I`-loop executes N^2 read operations on one data element, `B[K]`. The `K`-loop executes M write operations on each data element, A[I]. That is to say, the *RC* loops denote the number of computations executed on a single array element. Similarly, we call A the *RC* array of K-loop and B the *RC* array of I-loop.

```
int A[N], B[M];
1. FORALL (I = 0 .. N)
2.     FORALL (K = 0 .. M)
3.         A[I+2] += B[K-2] ;
```

Figure 29. Nested FORALL loops for different arrays.

When presenting the APD, *RC* loops are explicitly specified while, as before, *IC* loops are not. The difference between two arrays is whether the *RC* loop is executed inside or outside the *IC* loop. If the *RC* loop is executed inside the *IC* loop, as for array A, the specified data accesses are made M times in each iteration of the *IC* loop. If the *RC* loop is executed outside the *IC* loop, as for array B, the *IC* loop of size M is executed N times. The final resulting APD of Figure 29 is given in Figure 30.

```
For array A:      E_VIC-S: 2, Write, M Times

For array B:      RC-loop (N times) BEGIN
                      E_VIC-S: 2, Write
                  RC-loop END
```

Figure 30. Behavioral sharing description of nested FORALL loops for different arrays.

5.3.3 Multi-Dimensional Arrays

In this section, an example of the program segment and its corresponding APD are given. Given two-dimensional arrays A and B, lines 1 and 2 in Figure 31 (a) are the *IC* loops of A and lines 3 and 4 are the *RC* loops of A. The lines 3 and 4 are the *IC* loops of B and lines 1 and 2 are the *RC* loops of B. The APD is found in (b). Given an *n*-dimensional array, the access classes ($E_{VIC\text{-}S}$, E_L and *etc.*), *d*, *label*, α and β are written *n* times in the order from the first dimension to the last. The access type (read or write) is given only once.

```
int A[N][N], B[M][M];
1. FORALL (I = 0 .. N)
2.     FOALL (J = 0 .. N)
3.         FORALL (K = 0 .. M)
4.             FORALL (L = 0 .. M)
5.                 A[I+2][J-2] += B[K][L-2] ;
```

(a) Program segment

For array A:

{ $E_{VIC\text{-}S}$: 2 and $E_{VIC\text{-}S}$: -2} Write, M^2 Times

For array B:

RC-loop (N^2 times) BEGIN

{ E_L: and $E_{VIC\text{-}S}$: -2} Write

RC-loop END

(b) Behavioral sharing description

Figure 31. Behavioral sharing description for multi-dimensional arrays.

5.3.4 Example

The representative array `Particles` in MP3D of the SPLASH benchmark suite is taken as an example. It is a one-dimensional array of size *N* partitioned in the block-cyclic manner. Each element of `Particles` is 36 bytes in size and one partition block is composed of 16 elements. The static information and APDS are given in Figure 32. As shown in the figure, the sharing information of arrays in the real application is relatively short, and its sharing behaviors is simple to characterize.

```
Array Particle
   Size of an element:            36 bytes
   Dimension:                     1
   Size in each dimension:        N
   Partition:Block-Cyclic, block = 16 elements
```

(a) Static sharing information

```
1.  /* For array Particles */
2.     RC-loop ($N^2$ times) BEGIN
3.           $E_L$:    Read
4.           $E_L$:    Read
5.           $E_L$:    Write
6.           $E_L$:    Read
7.           $E_L$:    Read
8.           $E_{LV}$:   pn, Read
9.           $E_L$:    Read
10.          $E_{LV}$:   pn, Read
11.    RC-loop END
```

(b) Behavioral sharing description

Figure 32. Static and behavioral sharing description for particles in MP3D.

5.4 Sharing State Management and Event Counts

Given static sharing information and the APDS, the sharing state management unit is used to measure particular performance metrics of interest. The static information pre-defines the set of the interesting performance metrics such as the number of cache misses, the rules that govern sharing state transitions, and other rules that control the occurrences of events. Sharing state management unit takes in the APDS to execute proper sharing state transitions and to count the performance events.

Since the target is the miss rates, the events of interest are *the number of data references* and *the number of cache misses*. Given an n-dimensional array A with N_i, $0 \le i < n$, elements in each dimension that are used in the computations within a FORALL loop, there are $\prod N_i$, $0 \le i < n$, data references to A. As shown in Figure 31, the APDS implicitly represent data accesses executed within the FORALL loops. Therefore, each APDS for A yields $\prod N_i$, $0 \le i < n$, data references. In addition, the *RC* loops also influence the number of data references. If there are m nested *RC* loops of size l_{RC_j}, $0 \le j < m$, surrounding the computations on A, the total number of data references becomes $\prod l_{RC_j} \times \prod N_i$, $0 \le j < m$ and $0 \le i < n$. The number of data references is measured directly from the APD without any consideration for data-sharing. The number of processors is not a factor affecting the number of data reference.

Cache misses are experienced by a processor whenever it accesses data objects which are not in its local cache. Given an APDS, the information as to (1) which data objects a processor keeps in its cache and (2) which data objects it tries to access is necessary and sufficient to count the cache misses. The state transition rules adopted in this study is the three state write invalidate protocol as shown in Figure 6. At the beginning, a processor has no data in its cache. As the processor accesses data, new data are brought in and out of the cache. The events that load data into a processor's cache are data accesses (read or write) by the processor and events that remove data out of the cache are remote writes.

In Section 2.3.1, according to the situation where cache misses are observed, they were classified into pure cold misses (PCM), cold true sharing misses (CTSM), cold false sharing misses (CFSM), pure true sharing misses (PTSM) and pure false sharing misses

(PFSM). Being dependent on the sharing behavior of data objects, PCM and true sharing misses (CTSM and PTSM) can be characterized and measured based on data-sharing analysis. Unlike true sharing misses, false sharing misses (CFSM and PFSM) are observed when one or more write operations have been executed on the elements that are not accessed during the lifetime of the cache block. The occurrences of false sharing misses depend on the physical placement of cache blocks, which seems to be almost impossible to statically analyze. Since true sharing misses and false sharing misses are commonly dependent on data-sharing, the number of false sharing misses is estimated to be approximately proportional to that of true sharing misses. That is to say, they have the same order of magnitude in the big-oh notation.

In this regard, only the numbers of data references, PCM, CTSM and PTSM are to be modeled. On a PCM, a cache block containing a data object that caused the miss is brought into the local cache from the memory. No remote caches are involved during this coherency transaction. On the other hand, on a CTSM or a PTSM, if a modified copy of the data that caused the miss is kept in a remote cache, a remote processor is involved in the coherency transaction. The coherency transactions for CTSM and PTSM are identical. Furthermore, both the CTSM and PTSM are observed when a processor tries to access a data object that has been modified by a remote processor(s) since its last access. For these reasons, CTSM and PTSM are grouped into TSM (true sharing misses) and CFSM and PFSM are grouped into FSM (false sharing misses).

Therefore, in order to measure the number of cache misses, the sharing state management unit must keep track of data movements among processors so that the information as to *which elements a processor keeps in its cache after executing a behavioral sharing description statement* is maintained. Second, the sharing state management unit should also know *which array elements a processor tries to access during the execution of a behavioral sharing description statement.*

In Section 4, we learned about *general* sharing behaviors of diverse data access types. Being interested in cache misses, we will summarize the data-sharing analysis, focusing on the above-mentioned two requirements of the sharing state management unit.

5.4.1 Sharing Analysis for Cache Misses

Since every processor in an SPMD application is symmetric (Observation 5), we consider a processor, P_i, as the representative of all. We also consider an arbitrary partition block, B_{Pi}, which is allocated to P_i as the representative of all partition blocks. Note that, in Section 4.3.5, the access types were classified into E_L, $E_{LV\text{-}S}$, $E_{LV\text{-}M}$, $E_{VIC\text{-}S}$ and $E_{VIC\text{-}M}$. Given a data access type in the APD, only the execution of P_i on B_{Pi} is considered. P_i may access only data elements in B_{Pi} or other elements according to whether the access type is E_L or others ($E_{LV\text{-}S}$, $E_{LV\text{-}M}$, $E_{VIC\text{-}S}$ or $E_{VIC\text{-}M}$), respectively.

The array elements accessed by P_i *during* the execution of P_i on B_{Pi} are regarded as *elements accessed by P_i with respect to B_{Pi}* (e_{acc}). If the access type is E_L, $E_{LV\text{-}S}$ or $E_{VIC\text{-}S}$, the number of elements accessed by P_i with respect to B_{Pi} is the same as the number of elements in B_{Pi}. On the other hand, if the access type is $E_{LV\text{-}M}$ or $E_{VIC\text{-}M}$, the number of elements accessed by P_i with respect to B_{Pi} is larger than the number of elements in B_{Pi}.

In addition, the data residing in P_i's cache *at the completion* of the execution of P_i on B_{Pi} are regarded as the *elements residing in P_i with respect to B_{Pi}* (e_{res}). If the access is a read operation, e_{acc} and e_{res} are identical to each other. On the other hand, if the access is a write operation, because there can be only one modified copy of data in the system, we have $e_{acc} \supset e_{res}$.

Since an array element is modified by multiple processors in $E_{LV\text{-}S}$, $E_{LV\text{-}M}$, and $E_{VIC\text{-}M}$ with a write operation, the processors accessing the modified data should experience TSMs. The misses observed by P_i during the execution of P_i on B_{Pi} are regarded as the *misses observed during the accesses by P_i with respect to B_{Pi}*. The term n_{miss} is used to denote the number of these cache misses. Similarly, n_{acc} denotes the number of elements in e_{acc} and n_{res} denotes the number of elements in e_{res}.

In the sharing analysis of this section, the array is at first assumed to be partitioned in block-cyclic manner with b elements in B_{Pi}. When it is not necessary to distinguish event types such as data accesses or cache misses, they are simply called *events* and their number is denoted by n_e. Having $N/b = O(N)$ partition blocks (since b is a constant in block-cyclic partitioning) in block-cyclic partitioning, the total number of n_e will be multiplied by $N/b = O(N)$, that is, $n_e \times O(N)$. Although n_e is normally dependent on b ($O(1)$), they often depend on data set size. For example, n_{miss} may be measured as $f(N)$. Then, the effect of data set size on n_{miss} is $f(N) \times O(N)$. The effects of the number of processors are of two types. Section 4 show that, if data-sharing takes place between two adjacent processors, the number of processors has no impact on n_e. On the other hand, if data-sharing is observed on more than two processors or on two nonadjacent processors, the effect of the number of processors was known as $O((P-1)/P)$, in Section 4.

In cyclic partitioning where $b = 1$, there are also $O(N)$ ($= N/1 = N$) partition blocks. The discussion made for the block-cyclic partitioning is the same for the cyclic partitioning case. The only difference is that the elements accessed by P_i with non-E_L access types are all supposed to be remote data while, in block-cyclic partitioning case, some of them may be the local data to P_i.

In the block partitioning case with $b = N/P$, since there are only P partition blocks, the total number of n_e is simply multiplied by P to become $n_e \times O(P)$. The effect of data set size on n_e exists only when n_e is dependent on the data set size. In addition, since b is a function of N, n_e is frequently dependent on the data set size. The effects of the number of processors on n_e is always $O(P)$ if the access type is E_{VIC}. Recall that, in cyclic and block-cyclic cases, the number of processors does not affect or its effect is

$O((P-1)/P)$ depending on whether data-sharing is observed among more than two adjacent processors or not. This is due to the periodicity in data distribution. However, in block partitioning, since data distribution is not periodic, no such distinction is necessary. If the access type is E_{LV} in block partitioning, the effects of the number of processors on n_e is $O((P-1)/P)$ due to the irregularity of the access pattern.

In consequence, in the rest of the this section, the analysis will focus on the e_{acc}, e_{res} and n_{miss} for various access types. At the end of this section, a table will be provided to summarize the effects of data set size and number of processors on them.

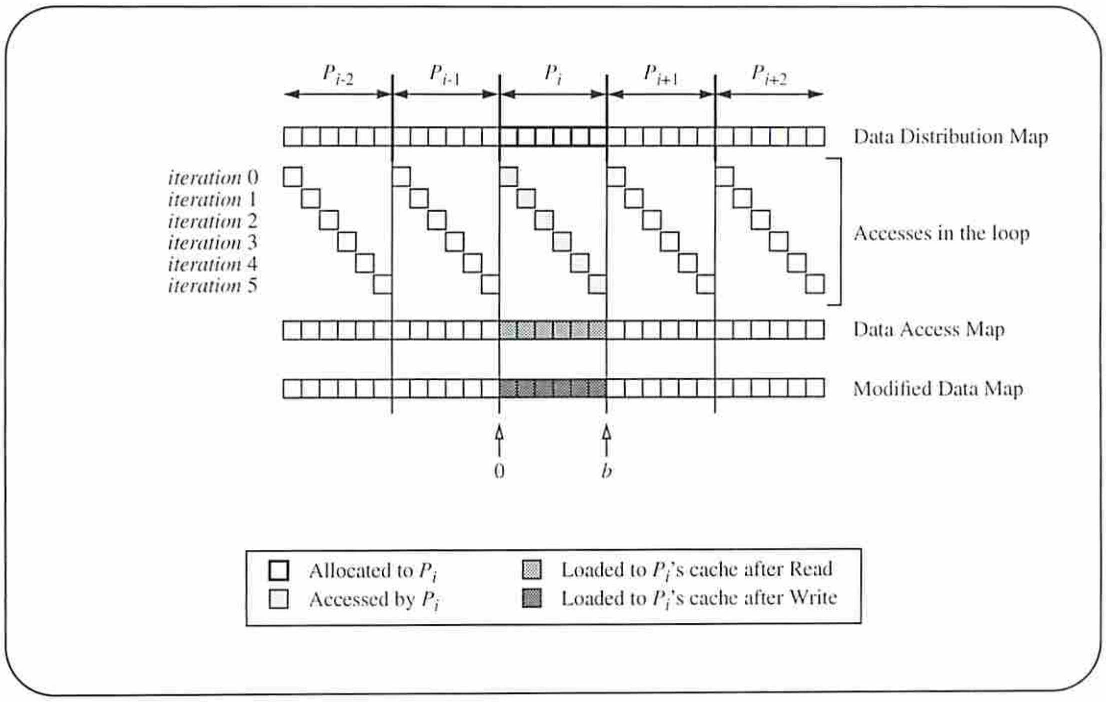


Figure 33. Data accesses of E_L.

Again, in Section 4.3.5, index expressions were finally categorized into five classes; E_L, $E_{VIC\text{-}S}$, $E_{VIC\text{-}M}$, $E_{LV\text{-}S}$ and $E_{LV\text{-}M}$. Figure 33 shows data accesses of E_L. There are four levels in the figure. At the top is the *data distribution map* to describe data distribution among processors. The second level presents data accesses of P_i with respect to B_{Pi} in each iteration of a FORALL loop. The *data access map* follows to show array elements that are accessed by P_i with respect to B_{Pi}, which is the projection in the figure, regardless of read or write. The elements marked in the data access map are loaded into P_i's cache at least once. They will all reside in P_i's cache at the completion of the read operation. If the access is a write operation, some of them have to move out of P_i's cache following the invalidation due to the remote processor's write operation. The diagram at bottom is the *modified data map* used for the write accesses of P_i with respect to B_{Pi}. If the access is a read, the modified data map is empty. The elements marked in the modified data map are in P_i's cache at the completion of the write access. In this figure, because the elements are exclusively accessed by their Home, they do not moving out to another cache during the accesses of P_i on B_{Pi}. The case where the data are loaded into a processor's cache and moved out to another cache will be shown shortly in $E_{VIC\text{-}M}$, $E_{LV\text{-}S}$ and $E_{LV\text{-}M}$.

As shown in Figure 33, for E_L, P_i accesses all of its local data in B_{Pi}, but no remote data. Once a data is accessed, it stays in P_i's cache. In other words, all data are accessed only by their Home processors and stay in their caches.

Figure 34 shows the data accesses of type $E_{VIC\text{-}S}$. As in E_L, all data are accessed only by one processor; either the Home or an adjacent processor. With respect to B_{Pi}, P_i accesses b consecutive elements which are d elements away from the elements in B_{Pi}. In other words, all processors access the elements that are d elements away from the elements allocated to them. If the elements are accessed, they stay at the accessing processor's cache until the completion of the accesses of P_i with respect to B_{Pi}, whether it is a read or write operation.

Figure 35 shows data accesses of $E_{VIC\text{-}M}$. With respect to B_{Pi}, P_i accesses $b + |\beta - \alpha|$ consecutive elements which are $d + \alpha$ elements away from B_{Pi}. All data are accessed $|\beta - \alpha|$ times by various number of processors (see Section 4.3.2 and Figure 18).

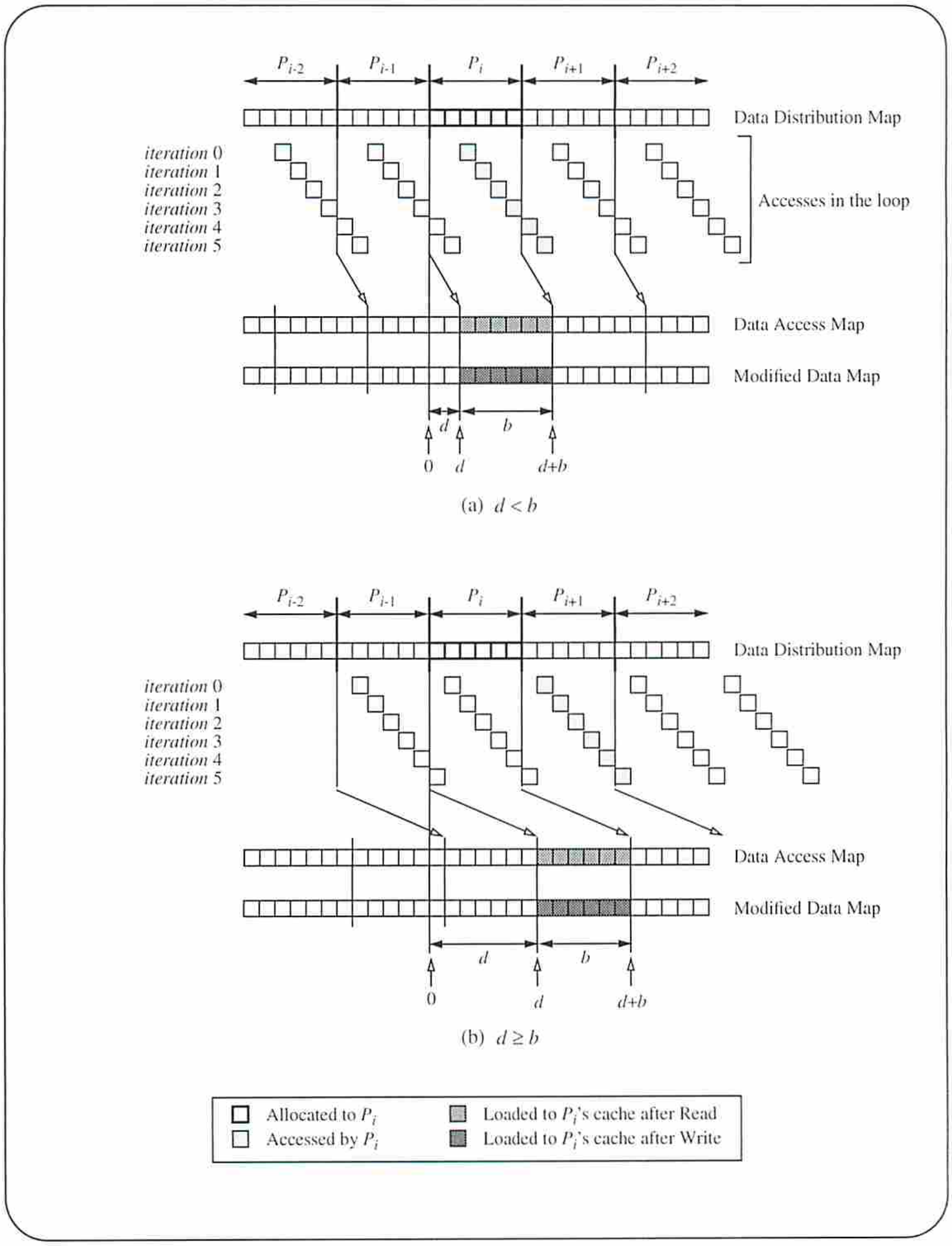


Figure 34. Data accesses of $E_{VIC\text{-}S}$.

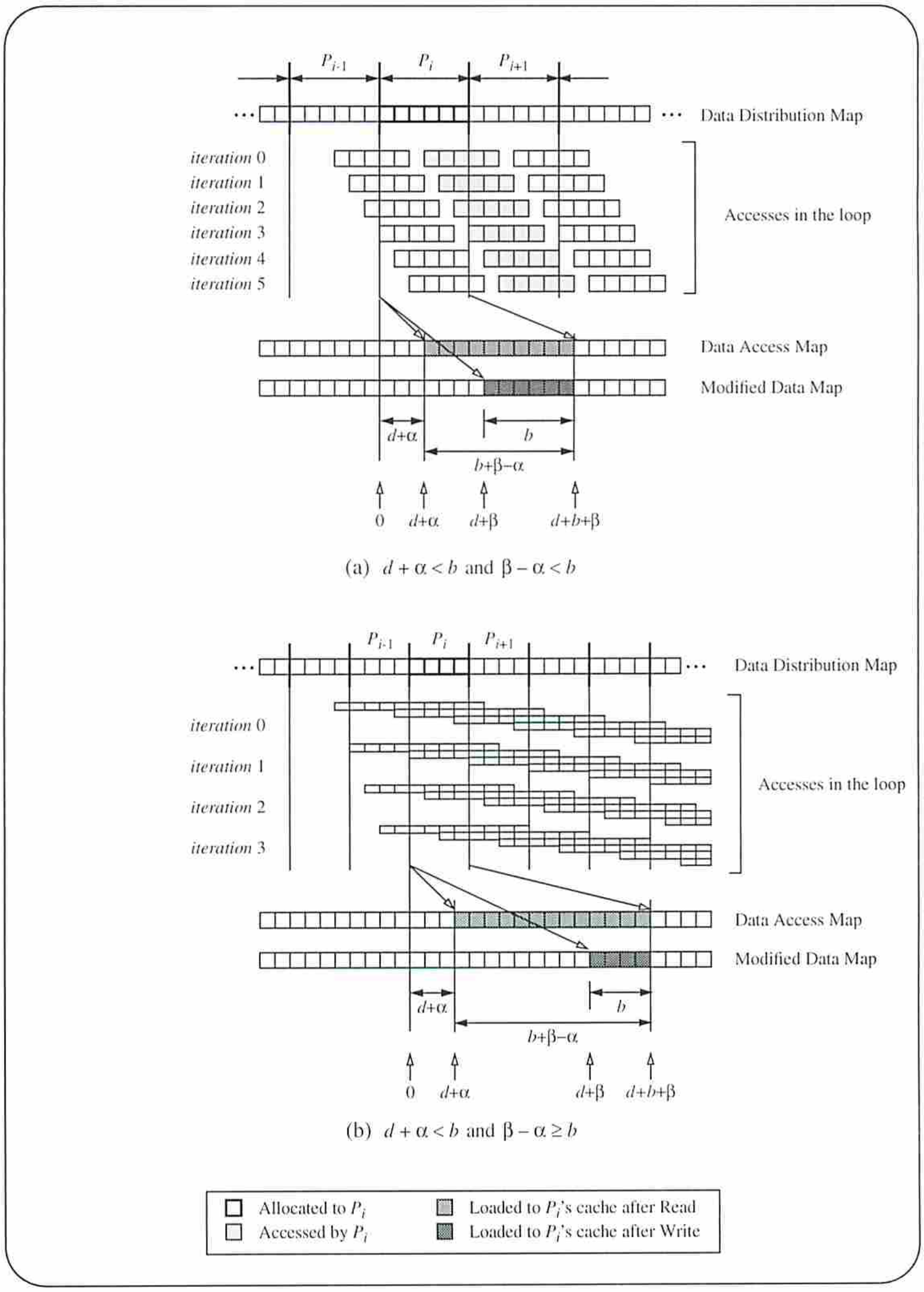
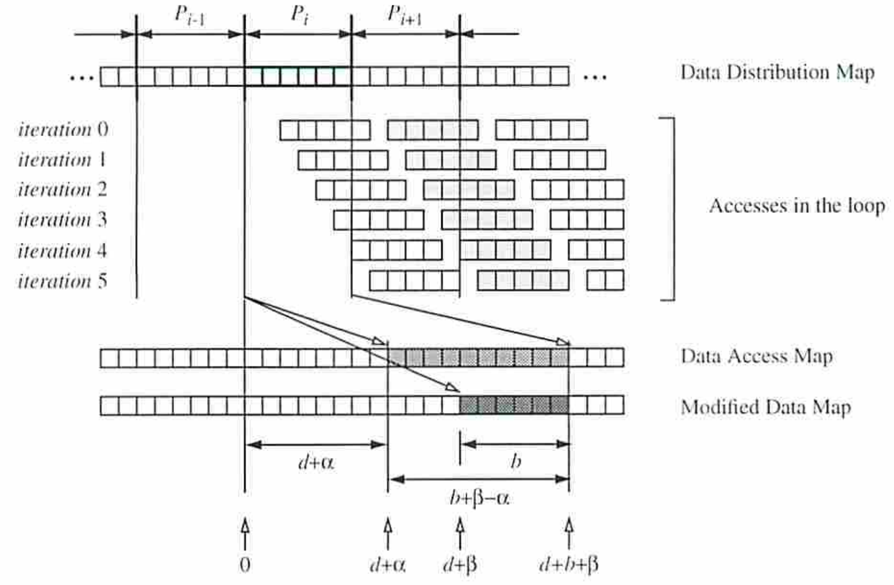
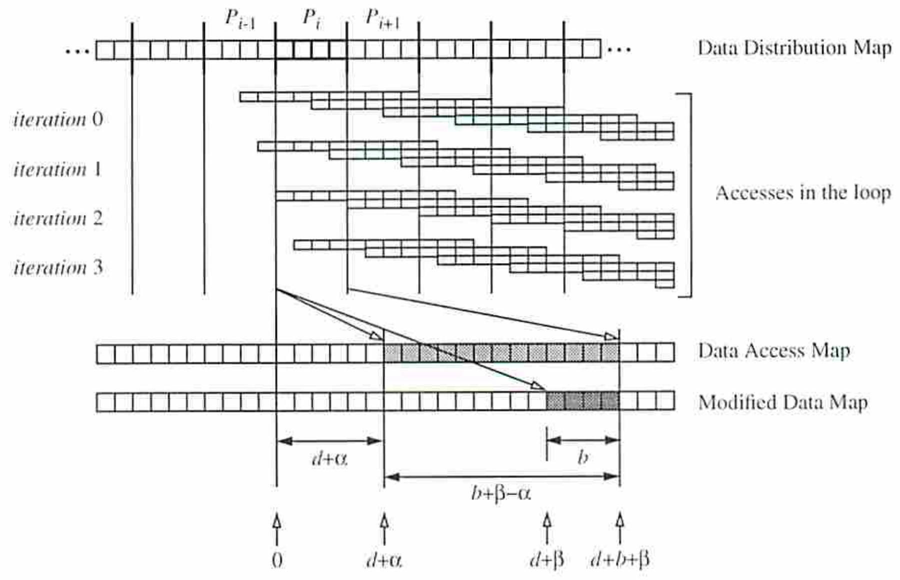


Figure 35. Data accesses of $E_{VIC\text{-}M}$.



(c) $d + \alpha \geq b$ and $\beta - \alpha < b$



(d) $d + \alpha \geq b$ and $\beta - \alpha \geq b$

□	Allocated to P_i	▨	Loaded to P_i's cache after Read
□	Accessed by P_i	■	Loaded to P_i's cache after Write

Figure 35. (Cont'd) Data accesses of $E_{VIC\text{-}M}$.

If a read operation is performed, the elements are copied into multiple processors' caches and stay there till the end of P_i's execution with respect to B_{P_i} according to the values of d, α and β. In contrast, if the access is a write operation, only one data copy per array element is allowed. As shown in Figure 35, P_i is approximately assumed to keep only b consecutive array elements in its cache from the one that is $d + \beta$ elements away from the leftmost element in B_{Pi}.

The approximation such that P_i keeps b consecutive elements in its cache is based on the lock-step fashion execution. Since we are dealing with the magnitude order in *big-oh* notation, this approximation will not yield any significant errors. To be precise, among b consecutive elements, a few elements may be taken away from P_i's cache to another processor's cache. In the modeling procedure, the coefficients are computed in the curve fitting stage using the simulation results. Thus, the difference between the approximation and practical value will eventually be eliminated completely.

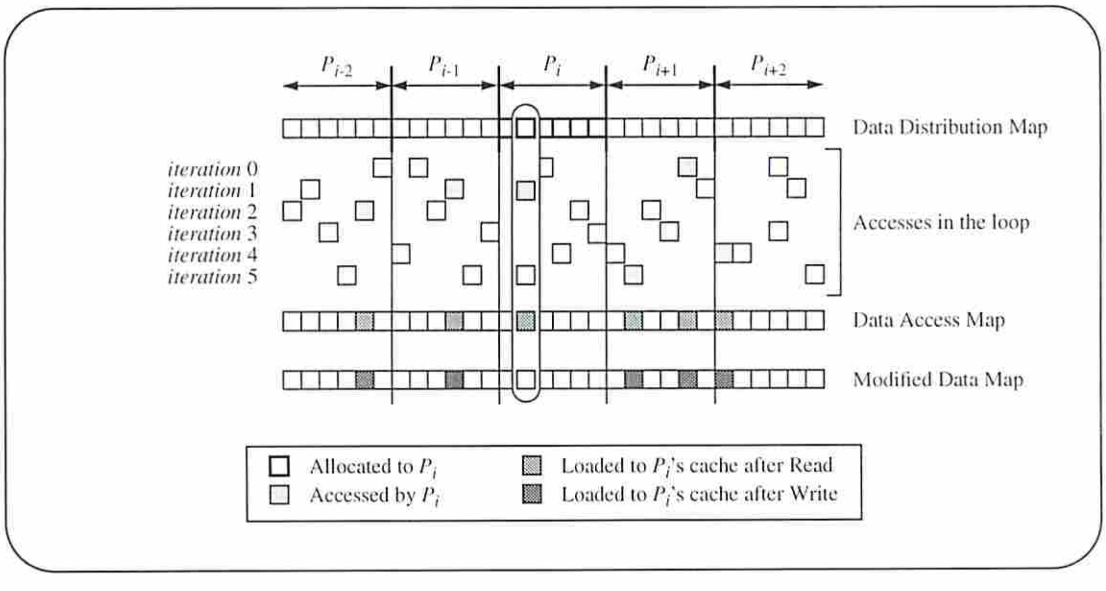


Figure 36. Data accesses of $E_{LV\text{-}S}$.

The access pattern for $E_{LV\text{-}S}$ is shown in Figure 36. With respect to B_{Pi}, P_i accesses b elements which are scattered over the index range. Although some elements may be accessed more than once by more than one processors, there is no differences in the magnitude orders of n_{acc} and n_{res}. They are both proportional to B_{Pi}. Interestingly, the number of elements accessed more than once is also proportional to the data set size, as well. That is, the magnitude orders of the total number of elements accessed by at least one processor and of the total number of elements residing in at least one cache are given by $O(N)$.

If the access is a read operation, exactly b elements are accessed and they stay in P_i's cache regardless of whether they are accessed by multiple processors or not. For the write operation, consider the second element of B_{P_i} in Figure 36. It is accessed by P_i first in iteration 1, and by a remote processor, P_j, in iteration 5. (P_j experiences a TSM at this point.) The data copy that was residing in P_i's cache in iteration 1 is taken away in iteration 5. This implies certain remote processors may have more than b elements in their caches. However, on the average, a processor contains b elements in its cache.

Summarily, this situation is approximated as follows. P_i accesses b elements with respect to B_{Pi}. $O(N)$ elements are accessed, in all, and $O(N)$ elements are accessed many times by multiple processors. On a write operation, $O(N)$ TSMs are observed.

In the case of $E_{LV\text{-}M}$, with respect to B_{Pi}, P_i accesses $|\beta - \alpha| \times b$ elements. In other words, an element is accessed $|\beta - \alpha|$ times. If the access is a read operation, the $|\beta - \alpha| \times b$ elements are accessed and they will stay in P_i's cache. However, at the completion of a write operation, there must be only b elements remaining in P_i's cache, on the average. This means that $|\beta - \alpha| \times (b-1)$ elements have gone out of P_i's cache, or each remote processors should observe $|\beta - \alpha| \times (b-1) = O(|\beta - \alpha| \times b)$ TSMs.

Discussions made in this section are summarized below.

- If the access type is E_L, $E_{VIC\text{-}S}$ or $E_{VIC\text{-}M}$, the locations of array elements accessed by P_i can be described by d, b, α and β.

- If the access type is $E_{LV\text{-}S}$ or $E_{LV\text{-}M}$, the locations of array elements accessed by P_i can not be described by d, b, α and β.
- For E_L, $E_{VIC\text{-}S}$ or $E_{LV\text{-}S}$, b elements are accessed and they are supposed to stay in P_i's cache regardless of the operation type at the completion of the accesses of P_i with respect to B_{Pi}.
- In $E_{VIC\text{-}M}$ and $E_{LV\text{-}M}$, if the access is read operation, $b + |\beta - \alpha|$ and $|\beta - \alpha| \times b$ elements stay in P_i's cache at the completion of the accesses of P_i with respect to B_{Pi}, respectively.
- In $E_{VIC\text{-}M}$ and $E_{LV\text{-}M}$, if write operation, only b elements will eventually remain in P_i's cache at the completion of the accesses of P_i with respect to B_{Pi}. That is to say, $|\beta - \alpha| \times (b - 1)$ TSM are observed during the accesses of P_i with respect to B_{Pi}, respectively.
- Not only $E_{VIC\text{-}M}$ and $E_{LV\text{-}M}$ but also $E_{LV\text{-}S}$ yields TSMs whose amount is proportional to $O(b)$, during the accesses of P_i with respect to B_{Pi}.
- If all elements in B_{Pi} were modified by remote processors in previous APDS, the first access to a certain element in B_{Pi} yields a TSM. If the access is a read operation, subsequent accesses to that element of $E_{LV\text{-}S}$, $E_{VIC\text{-}M}$ and $E_{LV\text{-}M}$ will not produce any more TSMs. If the access is a write operation, they will produce TSMs due to the interleaved accesses by distinct processors.
- In Table 13, a set of important metrics and events are summarized. They were all measured during the execution of P_i on the elements in B_{Pi}. The first row in the table also presents the elements that will reside in P_i's cache after the completion of read operations.
- To reflect the effects of data set size, the metrics presented in Table 13 are multiplied by the number of partition blocks which are N, $N/b = O(N)$ and N/P in cyclic, block-cyclic and block partitioning, respectively.

Events or Metrics	Data Partition	Access Types				
		E_L	$E_{VIC\text{-}S}$	$E_{VIC\text{-}M}$	$E_{LV\text{-}S}$	$E_{LV\text{-}M}$
Num. accessed elements (n_{acc})	Cyclic	1	1	$1+\lvert\beta-\alpha\rvert$	$O(1)$	$O(\lvert\beta-\alpha\rvert)$
	B-cyclic	b	b	$b+\lvert\beta-\alpha\rvert$	$O(b)$	$O(b\times\lvert\beta-\alpha\rvert)$
	Block	N/P	N/P	$N/P+\lvert\beta-\alpha\rvert$	$O(N/P)$	$O((N/P)\times\lvert\beta-\alpha\rvert)$
Distance of accessed elements from B_{Pi}	Cyclic	0	d	$d+\alpha$	*unspecifiable*	
	B-cyclic					
	Block					
Num. accesses per elements ($n_{ref\text{-}e}$)	Cyclic	1	1	$\lvert\beta-\alpha\rvert$	$O(1)$	$O(\lvert\beta-\alpha\rvert)$
	B-cyclic					
	Block					
Num. total data accesses	Cyclic	$n_{acc}\times n_{ref-e}$				
	B-cyclic					
	Block					
Num. modified elements in cache after write op.	Cyclic	1				
	B-cyclic	b				
	Block	N/P				
Distance of modified elements (after write op.) from B_{Pi}	Cyclic	0	d	$d+\beta$	*unspecifiable*	
	B-cyclic					
	Block					
Num. TSM during a single write op.	Cyclic	0	0	$O(n_{acc}\times n_{ref-e})$		
	B-cyclic					
	Block					

Table 13. Important metrics during the execution of P_i with respect to B_{Pi}.

- Finally, the effects of the number of processors are summarized in Table 14. All entries in the rows of $E_{LV\text{-}S}$ and $E_{LV\text{-}M}$ are due to the non-deterministic sharing behavior; the probability that a certain event takes place in remote data range is proportional to $O((P-1)/P)$. The reason for all entries in the rows of $E_{VIC\text{-}S}$ and $E_{VIC\text{-}M}$ for cyclic partitioning ($b = 1 < d$) is the periodicity of data distribution. In addition, the entries in the rows of $E_{VIC\text{-}S}$ and $E_{VIC\text{-}M}$ for block-cyclic partitioning with $O((P-1)/P)$ are also due to the periodic data distribution. Finally, the effects of the number of processors in block partitioning case are all $O(P)$ except for $E_{VIC\text{-}S}$ with $d \geq b$.

Access Type	Conditions	Partitioning Methods		
		Cyclic	Block-Cyclic	Block
E_L	-	-	-	-
$E_{LV\text{-}S}$	-	$O((P-1)/P)$	$O((P-1)/P)$	$O((P-1)/P)$
$E_{LV\text{-}M}$	-	$O((P-1)/P)$	$O((P-1)/P)$	$O((P-1)/P)$
$E_{VIC\text{-}S}$	$d < b$	$O((P-1)/P)$	-	$O(P)$
	$d \geq b$	$O((P-1)/P)$	$O((P-1)/P)$	-
$E_{VIC\text{-}M}$	$d+\alpha < b$, $\beta-\alpha < b$	$O((P-1)/P)$	-	$O(P)$
	$d+\alpha < b$, $\beta-\alpha \geq b$	$O((P-1)/P)$	$O((P-1)/P)$	$O(P)$
	$d+\alpha \geq b$, $\beta-\alpha < b$	$O((P-1)/P)$	$O((P-1)/P)$	$O(P)$
	$d+\alpha \geq b$, $\beta-\alpha \geq b$	$O((P-1)/P)$	$O((P-1)/P)$	$O(P)$

Table 14. Effect of number of processors on the event in Table 13.

5.4.2 Number of References

In the previous section, we performed sharing analysis for measuring cache miss rates. Based on the knowledge obtained, in this and the next subsections, we will introduce the methodology to compute the magnitude orders of the number of references and of the numbers of various cache misses.

The formal algorithm to measure the number of data references is shown in Figure 37. The APD is assumed to be in the form of Figure 32 where each statement is serially numbered. There is a variable and an array to maintain the information regarding *RC* loops; `RC_Depth` to denote the depth of current *RC* loop and `RC_Size[MAX_RC]` to denote the size of current *RC* loop. An additional data structure `N_Ref_List[MAX_STMT]` is used to record the numbers of data accesses incurred by each APDS.

```
1.    DataSetSize = f(N);
2.    RC_Depth = 0;
3.    RC_Size[MAX_RC_Loop] = 1;
4.    N_Ref_List = empty;
5.    for (each behavioral description statement) {
6.       if( RC_Loop BEGIN statement ){
7.          RC_Depth++;
8.          RC_Size[RC_Depth]=RC_Size[RC_Depth-1] *
9.                                      current RC loop size;
10.      }
11.      else if ( RC_Loop END statement ){
12.         RC_Size[RC_Depth] = RC_Size[RC_Depth-1];
13.         RC_Depth--;
14.      }
15.      else /* regular data access statement */{
16.         num_references = find from Table 13 and multiply
                       with the number of partition blocks ;
17.         if ( implicit RC_Loop with size S )
18.            num_references *= S;
19.         num_references *= RC_Size[RC_Depth];
20.         add num_references into N_Ref_List;
21.      }
22.   }
23.   sum up magnitude orders in N_Ref_List;
```

Figure 37. Algorithm to measure the number of data references.

Initialization is done in lines 1 through 4. The data set size (`DataSetSize`) of the shared array is obtained from the static sharing information. `DataSetSize` defines the *IC* (or, FORALL) loop size of each APDS. At the very beginning, no *RC* loop is outstanding so that the depth of the *RC* loop for the current statement (RC_Depth) is zero. For all possible *RC* loops, their sizes (RC_Size[MAX_RC]) are initially assumed to be one. This implies that the body statements in the APD are executed only once. Finally, the list of the numbers of data accesses to be observed by the APDS (N_Ref_List[MAX_STMT]) is assumed to be empty.

Lines 5 through 23 form the main body of the algorithm. An example will help understand the way the algorithm works. Assume an array with data set size *f(N)* and its APD given in Figure 38. On S_1, RC_Depth becomes 1 and RC_Size[1] becomes n_1. On S_2, RC_Depth is 2 and RC_Size[2] is $n_1 \times n_2$. These are shown by lines 7, 8, 9 in Figure 37. $stmt_1$ is executed on *f(N)* array elements and the number of data accesses is $n_1 \times n_2 \times f(N)$, which is the result of lines 16, 19 in Figure 37. In line 16, the sharing analysis results obtained in Section 5.4.1 and Table 13 are used. On S_4, RC_Depth becomes 1 and RC_Size[2] is reset to 1. The number of accesses due to $stmt_2$ is the product of *f(N)* and RC_Size[1]; $n_1 \times f(N)$. S_6 in Figure 38 implies that there is an *RC* loop within the *IC* loop (see array A in Figure 31 (b)). In this case, the number of data references is $n_1 \times n_3 \times f(N)$ which is the result of lines 16, 18, 19 in Figure 37. Having no *RC* loop around, $stmt_4$ yields *f(N)* references. The number of references computed for each statement is stored into the N_Ref_List. After all statements are processed, the last step is to sum up the magnitude orders stored in the N_Ref_List.

5.4.3 Number of PCM

In Table 13, the number of data elements accessed by P_i during its execution with respect to B_{Pi} for all possible cases is given. The total number of data elements accessed by any processor should be at least *O(N)*. But, since an element brings in exactly one cold miss for a processor, the upper bound of the number of cold misses is *O(N)*. Hence, the number of cold misses is always proportional to *O(N)*. As a result, the goal in measuring

```
$S_1$.    RC-loop ($n_1$ times) BEGIN
$S_2$.        RC-loop ($n_2$ times) BEGIN
$S_3$.                stmt$_1$
$S_4$.        RC-loop END
$S_5$.                stmt$_2$
$S_6$.                stmt$_3$, $n_3$ times
$S_7$.        RC-loop END
$S_8$.    stmt$_4$
```

Figure 38. Example for the measurement of data references.

the number of cold misses is to find out the number of processors observing a cold miss from an array element. This is identical to measuring the sharing degree of array elements.

A pure cold miss (PCM) is observed when a processor accesses a cache block for the first time which has not been modified by any other processors. So, *the behavioral sharing description statements appearing before the first write statement and the write statement itself are the only sources of PCM. RC* loops do not affect the number of PCM since the PCMs are measured only once per element for a processor.

Table 15, which is based on Table 14, shows the effect of the number of processors on the number of PCMs. The first row represents the case where there is only one APDS, or there are two or more exactly same statements, before and during the first write statement. If the index expression in the statement(s) is for a single array element (E_L, $E_{VIC\text{-}S}$, or $E_{LV\text{-}S}$), every array element is accessed by one processor. This results in the fact that the number of PCMs is not affected by the number of processors. For the other cases, the entries in the remaining rows are applied. For instance, if the index expression is for multiple array elements ($E_{VIC\text{-}M}$, or $E_{LV\text{-}M}$), an array element may be accessed by multiple processors. If there are two or more different APDSs before the first write, an array element should be accessed by multiple processors as well.

Access Types		Conditions	Partitioning Methods		
			Cyclic	Block-Cyclic	Block
Single or same multiple behavioral statements of type E_L, $E_{LV\text{-}S}$ or $E_{VIC\text{-}S}$ before the first write			-	-	-
Multiple different behavioral sharing description statements	E_L	-	-	-	-
	$E_{LV\text{-}S}$	-	$O((P-1)/P)$	$O((P-1)/P)$	$O((P-1)/P)$
	$E_{LV\text{-}M}$	-	$O((P-1)/P)$	$O((P-1)/P)$	$O((P-1)/P)$
	$E_{VIC\text{-}S}$	$d < b$	$O((P-1)/P)$	-	$O(P)$
		$d \geq b$	$O((P-1)/P)$	$O((P-1)/P)$	-
	$E_{VIC\text{-}M}$	$d + \alpha < b$, $\beta - \alpha < b$	$O((P-1)/P)$	-	$O(P)$
		$d + \alpha < b$, $\beta - \alpha \geq b$	$O((P-1)/P)$	$O((P-1)/P)$	$O(P)$
		$d + \alpha \geq b$, $\beta - \alpha < b$	$O((P-1)/P)$	$O((P-1)/P)$	$O(P)$
		$d + \alpha \geq b$, $\beta - \alpha \geq b$	$O((P-1)/P)$	$O((P-1)/P)$	$O(P)$

Table 15. Effects of number of processors on the number of PCM.

The algorithm in Figure 39 shows that the number of PCMs is computed merely by executing a table look-up (lines 10 and 15). What is necessary in static information is the partitioning method (line 1). Instead of the `N_Ref_List` for the number of data references used in Figure 37, `N_PCM_List` is used to store the magnitude order of the number of processors affecting the number of PCM observed during each APDS.

The main body of the algorithm takes care of only the regular data accesses with read operations before the first write statement (line 10) and the first write statement itself (line 15). For each of those statements, table look-up is performed. As soon as the first write statement is met (line 7), the loop is broken (line 8) and the control in the algorithm jumps to the point (lines 15) where the magnitude order of the number of processors in the first write statement is to be found.

```
1.    /* Data Partitioning Method is known */
2.    N_PCM_List = empty ;
3.    for (each behavioral description statement) {
4.       if ( RC_Loop BEGIN OR RC_Loop END statements )
5.          do nothing ;
6.       else /* regular data access statement */ {
7.          if ( write operation )
8.             break for-loop and go to line 15 ;
9.          else {
10.            look up Table 15;
11.            add the result into N_PCM_List;
12.         }
13.      }
14.   }
15.   look up Table 15 for PCM in the first write statement ;
16.   add the result into N_PCM_List;
17.   sum up magnitude orders in N_PC_List;
```

Figure 39. Algorithm to measure the number of PCM.

5.4.4 Number of TSMs

A true sharing miss (CTSM or PTSM) is a miss which is preceded by a write operation from another processor. At the completion of a write operation in a FORALL loop, all array elements are in *dirty* or *modified* state. The processor that keeps the exclusive dirty copy of a certain array element is determined from the access types and access parameters. In addition, the elements a processor, P_i, tries to access is also diversely defined according to the access type. If P_i tries to access an element which was loaded into a remote cache after the write operation, it has to experience a TSM.

5.4.4.1 Sharing State after a Write Statement

The first requirement to measure the number of TSMs is to know the distribution of modified data after the write statement. In Section 5.4.1, the complete sharing analysis for cache misses and the distribution of modified data for various access types were introduced. The *modified data maps* in Figures 33 through 36 show the modified array elements that the processor P_i keeps in its cache after completing its write operations with respect to B_{Pi}. They were summarized in Table 13. In this section, we build another table based on Table 13. Table 16 more specifically describes the index ranges of modified elements processor P_i keeps in its cache after completing its write operations with respect to B_{Pi}.

Data Partition		Access Types				
		E_L	$E_{VIC\text{-}S}$	$E_{VIC\text{-}M}$	$E_{LV\text{-}S}$	$E_{LV\text{-}M}$
Cyclic	from	0	d	$d+\beta$	*unspecifiable (scattered over index range)*	
	to	1	$d+1$	$d+\beta+1$		
B-cyclic	from	0	d	$d+\beta$		
	to	b	$d+b$	$d+\beta+b$		
Block	from	0	d	$d+\beta$		
	to	N/P	$d+N/P$	$d+\beta+N/P$		

Table 16. Locations of modified elements of P_i after a write operation with respect to B_{Pi}.

The entries in the table denote the relative distance from the leftmost element in B_{Pi}. The elements designated by the *from* rows are inclusive while those designated by the *to* rows are exclusive. For better understanding, see Figures 33 to 36. Note that the conditions (such as $b < d$ or $d + \alpha \geq b$) which were applied to each partitioning scheme in sharing analysis are not taken into account in describing the locations of modified elements of P_i after a write operation with respect to B_{Pi}.

5.4.4.2 Access Pattern of a Statement

The other requirement to measure the number of TSMs is to know the elements to be accessed during the execution of the APDS. In Section 5.4.1, the complete sharing analysis for data access patterns for various access types were introduced. The *data access maps* in Figures 33 through 36 shows the array elements that the processor P_i accesses during its execution with respect to the elements in B_{Pi}. They are summarized in Table 13. In this section, we build another table based on Table 13. Table 17 more specifically describes the index ranges of the elements accessed by processor P_i during its execution with respect to B_{Pi}.

Data Partition		Access Types				
		E_L	$E_{VIC\text{-}S}$	$E_{VIC\text{-}M}$	$E_{LV\text{-}S}$	$E_{LV\text{-}M}$
Cyclic	from	0	d	$d + \alpha$	*unspecifiable (scattered over index range)*	
	to	1	$d + 1$	$d + \beta + 1$		
B-cyclic	from	0	d	$d + \alpha$		
	to	b	$d + b$	$d + \beta + b$		
Block	from	0	d	$d + \alpha$		
	to	N/P	$d + N/P$	$d + \beta + N/P$		

Table 17. Locations of array elements accessed by P_i during its execution with respect to B_{Pi}.

In Table 17, the entries in the table denote the relative distance from the leftmost element in B_{Pi}, and the elements designated by the *from* rows are inclusive while those designated by the *to* rows are exclusive. The conditions such as $b < d$ and $d + \alpha \geq b$ are not taken into consideration in this table. The differences between Tables 16 and 17 are the entries in the shaded table cells appearing in $E_{VIC\text{-}M}$ case. Obviously, the elements to appear in E_{LV} cases of both tables must be distinct from each other, although they are specifiable. As in Table 16, the entries in the table denote the relative distance from the leftmost element in B_{Pi}, and the elements designated by the *from* rows are inclusive while those designated by the *to* rows are exclusive. The conditions such as $b < d$ and $d + \alpha \geq b$ are not considered in this table.

5.4.4.3 TSMs on Read Statement

Using the information obtained from earlier sections, we can estimate the number of TSMs. In this section, we consider read statements following a write statement. Assume the recent write statement is of type E_L, $E_{VIC\text{-}S}$, or $E_{VIC\text{-}M}$. Then, processor P_i keeps some modified elements at deterministic locations in its cache. If the read statement that follows the write statement is also of type E_L, $E_{VIC\text{-}S}$, or $E_{VIC\text{-}M}$, the number of elements that processor P_i cannot find in its cache are measurable using access parameters (d, α and β). In Figure 40, at the top level are the modified elements residing in P_i's cache after the write operation of P_i with respect to B_{Pi}. The second level shows the elements accessed by P_i during its read operation with respect to B_{Pi}. Then, the shaded elements at the bottom level denote the elements not found in P_i's cache during its read operation with respect to B_{Pi}. They are the TSM. The figure is an example where the write statement and read statements are both of type $E_{VIC\text{-}S}$.

Given a write statement of type E_L, $E_{VIC\text{-}S}$, or $E_{VIC\text{-}M}$, assume a read statement is of type $E_{LV\text{-}S}$ or $E_{LV\text{-}M}$. Whenever P_i makes its access to an element designated by an index expression of type $E_{LV\text{-}S}$ or $E_{LV\text{-}M}$, the probability that the element is kept in a remote cache is $(P-1)/P$, regardless of the access type of the recent write statement. This is true even for the case where the write statement is of type $E_{LV\text{-}S}$ or $E_{LV\text{-}M}$. The example in Figure 41 (a) shows the TSM observed on $E_{LV\text{-}S}$ read statement after $E_{VIC\text{-}S}$ write statement, both of which are executed by P_i with respect to B_{Pi}. Another example is in Figure 41 (b) where both read and write statements are of type $E_{LV\text{-}S}$ executed by P_i with respect to B_{Pi}. The

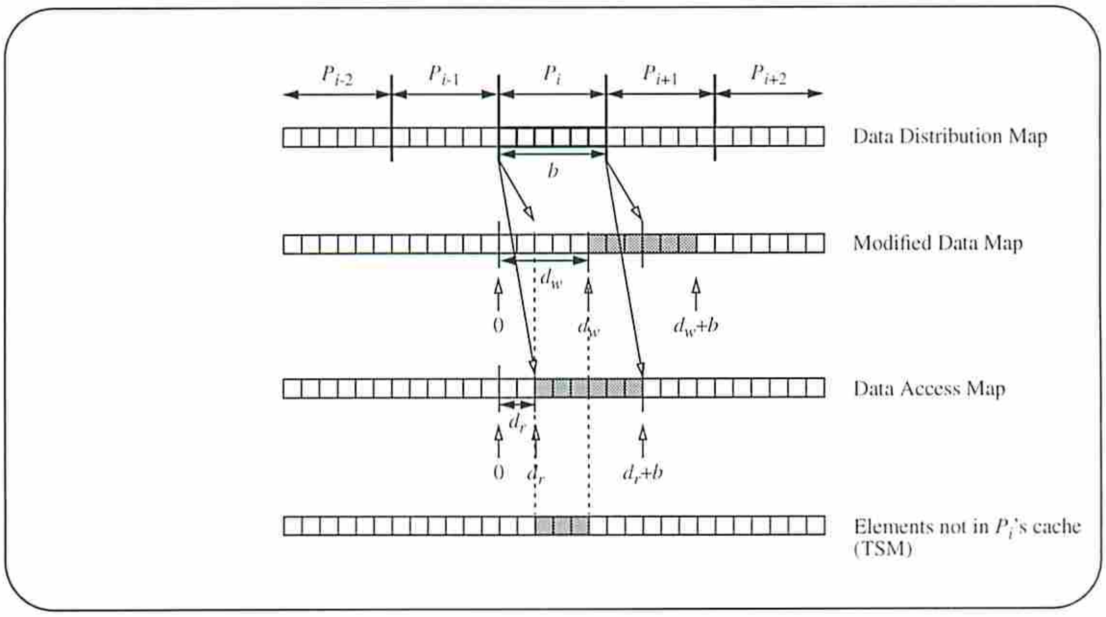


Figure 40. TSM on $E_{VIC\text{-}S}$ read statement after $E_{VIC\text{-}S}$ write statement.

number of TSMs observed by P_i during its read operation with respect to B_{Pi} is estimated as $O(b)$ and $O(b + |\beta + \alpha|)$ for $E_{LV\text{-}S}$ and $E_{LV\text{-}M}$, respectively.

Finally, if the write statement is of type $E_{LV\text{-}S}$ or $E_{LV\text{-}M}$, P_i keeps some modified elements at arbitrary locations in its cache. When P_i makes its access to an array element during its execution with respect to B_{Pi}, the probability that the element is not in P_i's cache is $(P-1)/P$, regardless of the access type of the read statement. This is the probability that P_i experiences a TSM on its access to an element. The number of TSMs that P_i experiences is proportional to the number of accesses it makes during the execution with respect to B_{Pi}. An example is presented in Figure 42.

When a series of read statements are executed after a write statement, the estimated number of TSMs are *summed up*. For example, assume an APD in Figure 43 (a). Three consecutive read statements follow a write statement. Figure 43 (b) shows the sharing patterns of processor P_i. The elements in the figure at the top level are modified by their Home (by E_L) and stay in their Homes' caches. The numbers of TSMs observed

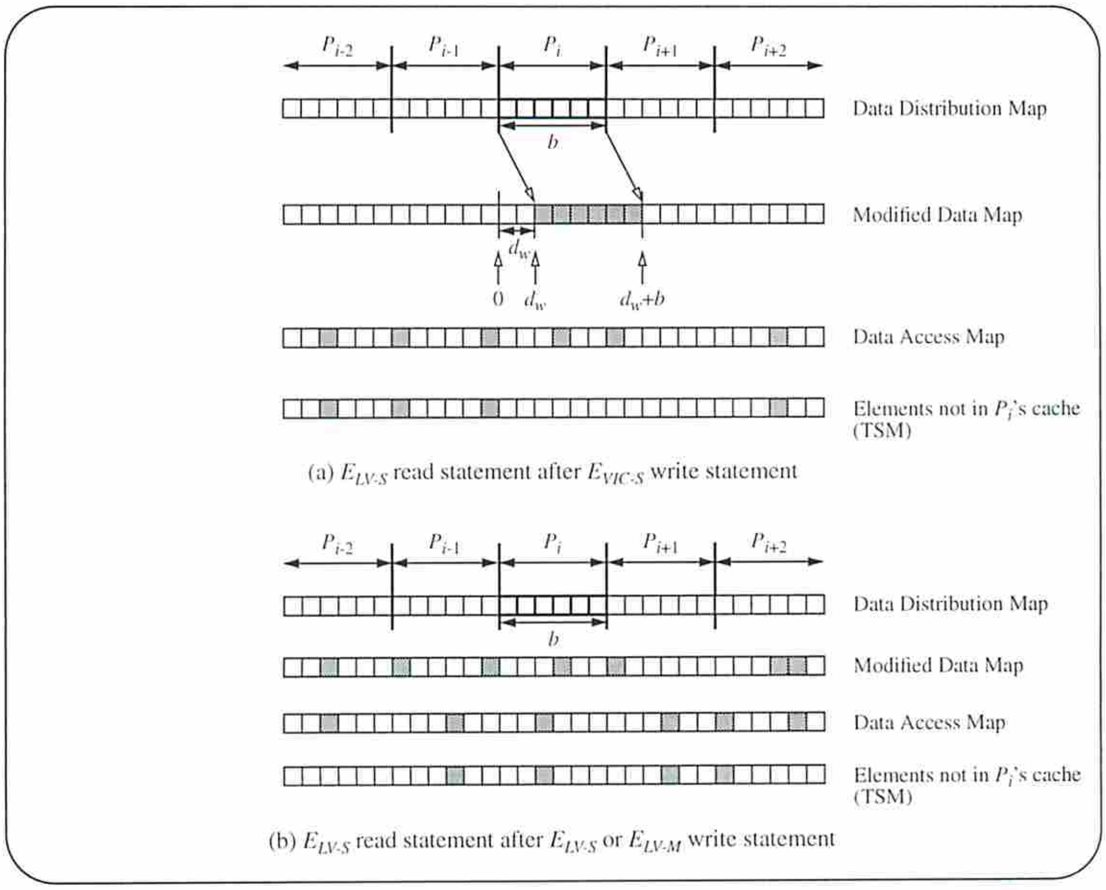

P_{i-2}
P_{i-1}
P_i
P_{i+1}
P_{i+2}
Data Distribution Map
b
Modified Data Map
d_w
0
d_w
d_w+b
Data Access Map
Elements not in P_i's cache (TSM)
(a) $E_{LV\text{-}S}$ read statement after $E_{VIC\text{-}S}$ write statement
Data Distribution Map
b
Modified Data Map
Data Access Map
Elements not in P_i's cache (TSM)
(b) $E_{LV\text{-}S}$ read statement after $E_{LV\text{-}S}$ or $E_{LV\text{-}M}$ write statement

Figure 41. TSM on $E_{LV\text{-}S}$ or $E_{LV\text{-}M}$ read statement after a write statement.

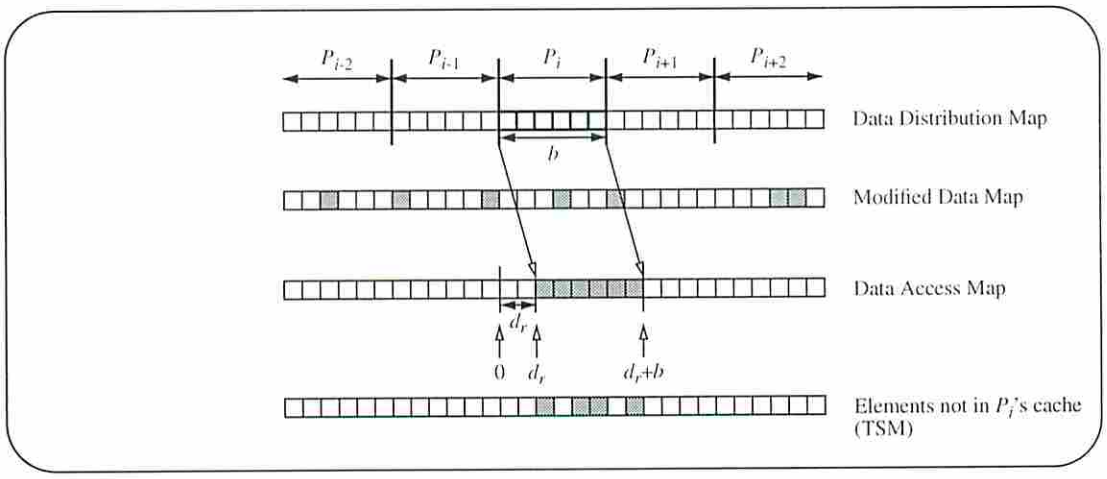


Figure 42. TSM on $E_{VIC\text{-}S}$ read statement after $E_{LV\text{-}S}$ or $E_{LV\text{-}M}$ write statement.

when S_2, S_3 and S_4 are individually executed immediately after S_1 are shown as well as the number of TSMs when S_1, S_2, S_3 *and* S_4 are executed serially. We observe that the total number of TSMs caused by serial read statements, whose magnitude order is expressed in big-oh notation, is same as the sum of the magnitude orders of the number of TSMs caused by individual read statements executed immediately after the write.

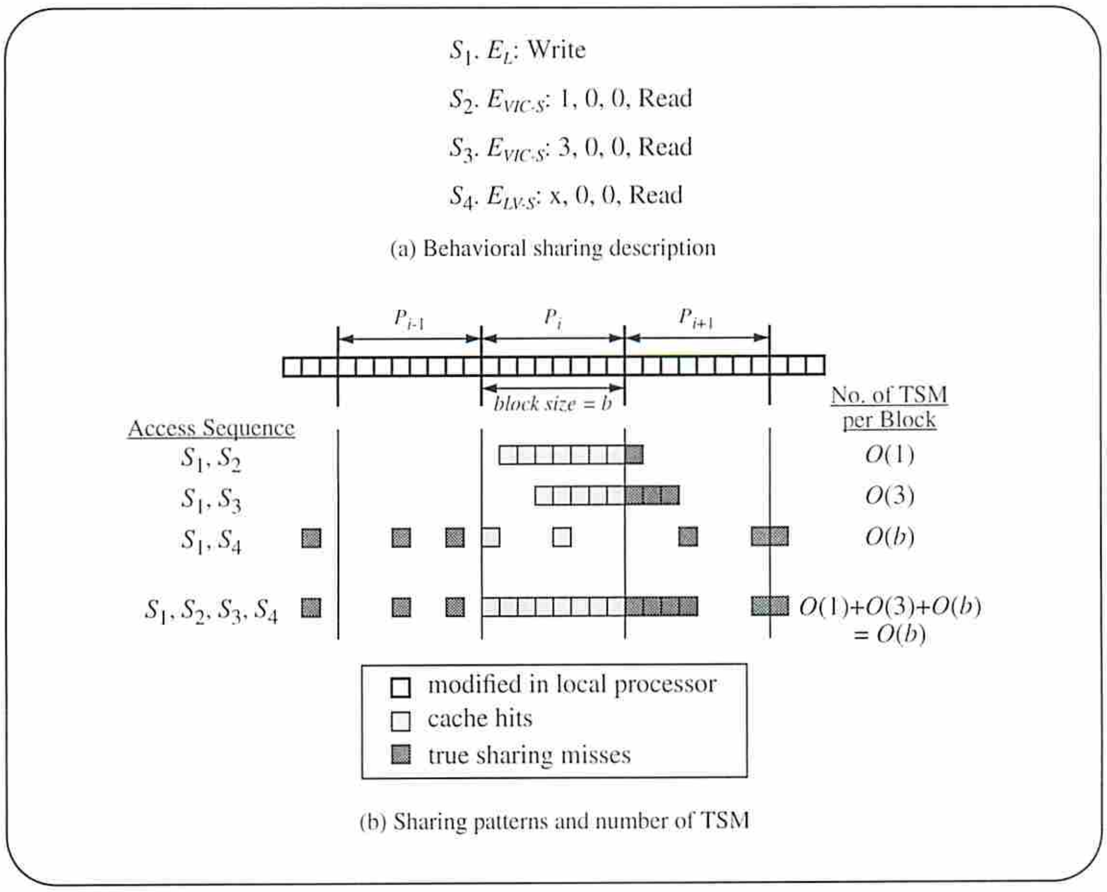

S_1. E_L: Write
S_2. $E_{VIC\text{-}S}$: 1, 0, 0, Read
S_3. $E_{VIC\text{-}S}$: 3, 0, 0, Read
S_4. $E_{LV\text{-}S}$: x, 0, 0, Read
(a) Behavioral sharing description
P_{i-1}
P_i
P_{i+1}
block size = b
Access Sequence
S_1, S_2
S_1, S_3
S_1, S_4
S_1, S_2, S_3, S_4
No. of TSM per Block
$O(1)$
$O(3)$
$O(b)$
$O(1)+O(3)+O(b) = O(b)$
modified in local processor
cache hits
true sharing misses
(b) Sharing patterns and number of TSM

Figure 43. Number of TSM for consecutive reads after a write.

More interestingly, assume that a read statement, S_r, that produces $O(f(N))$ and $O(g(P))$ TSMs is executed twice. In practice, the first execution of S_r results in $O(f(N))$ and $O(g(P))$ TSMs, while the second execution produces cache hits, only. Therefore, the total number of TSMs produced by two read operations is $O(f(N))$ and $O(g(P))$. However, in our scheme, $O(f(N))$ and $O(g(P))$ TSM are recorded for both executions of the read statements given in the program code. Having $O(f(N)) + O(f(N)) = O(f(N))$ and $O(g(P)) + O(g(P)) = O(g(P))$, we get the same number of TSMs. As a result, the number of TSMs observed by consecutive read statements following a write is measured for individual read statements and added together to obtain the total number of TSMs.

From the number of TSMs observed by P_i during its read accesses with respect to the elements in B_{Pi} (Table 18), the total number of TSMs can be computed using the effects of data set size and number of processors as discussed in Section 5.4.1.

	E_L	$E_{VIC\text{-}S}$	$E_{VIC\text{-}M}$	$E_{LV\text{-}S}$	$E_{LV\text{-}M}$
E_L	Estimation using access parameters (d, α, β)			$O(b)$	
$E_{VIC\text{-}S}$				$O(b)$	
$E_{VIC\text{-}M}$				$O(b + \lvert\beta - \alpha\rvert)$	
$E_{LV\text{-}S}$	$O(b)$			$O(b)$	
$E_{LV\text{-}M}$	$O(b \times \lvert\beta - \alpha\rvert)$			$O(b \times \lvert\beta - \alpha\rvert)$	

Table 18. Summary of TSM on read operation by P_i with respect to B_{Pi}.

5.4.4.4 TSM on Write Statement

In this section, we consider the case where the *current* statement in the APD that follows a *previous* write statement is also a write statement. Let's first assume the previous write statement is of type E_L, $E_{VIC\text{-}S}$ or $E_{VIC\text{-}M}$ so that a processor P_i has b consecutive elements at deterministic locations, in its cache, after the completion of its write operation with respect to B_{Pi}. If the current write statement is of type E_L or $E_{VIC\text{-}S}$, the number of TSMs that are observed by P_i during its execution with respect to B_{Pi} can be computed using access parameters, d, α and β. This procedure is identical to the case where the current statement is a read operation. The only difference is the resulting sharing state of the modified elements after the write operation.

If the previous write statement is of type E_L, $E_{VIC\text{-}S}$ or $E_{VIC\text{-}M}$ and the current write statement is of type $E_{VIC\text{-}M}$, b consecutive elements at the distance of $d + \alpha$ from the left-most element in B_{Pi} are accessed *first* by P_i, and, later, by other processors. The other elements that P_i will access are preceded by remote processors' write operations (Figure 44). Therefore, the TSM that P_i observes during its execution with respect to B_{Pi} are classified into two kinds. The first kind of TSM occurs due to the previous write operation. The TSMs observed on the b elements between the relative distances of $d + \alpha$ (inclusive) and $d + \alpha + b$ (exclusive) from the first element of B_{Pi} belong to this category. The number of TSMs in this case is estimated by using access parameters. Another kind of TSM is due to the write operations from remote processors executed in the *current* write statement. In this case, all accesses yield TSMs. The number of TSMs observed by P_i during its execution with respect to B_{Pi} is proportional to $O(|\beta - \alpha|)$. The total number of TSMs that P_i observes during its execution with respect to B_{Pi} is the sum of the number of TSMs measured in two cases.

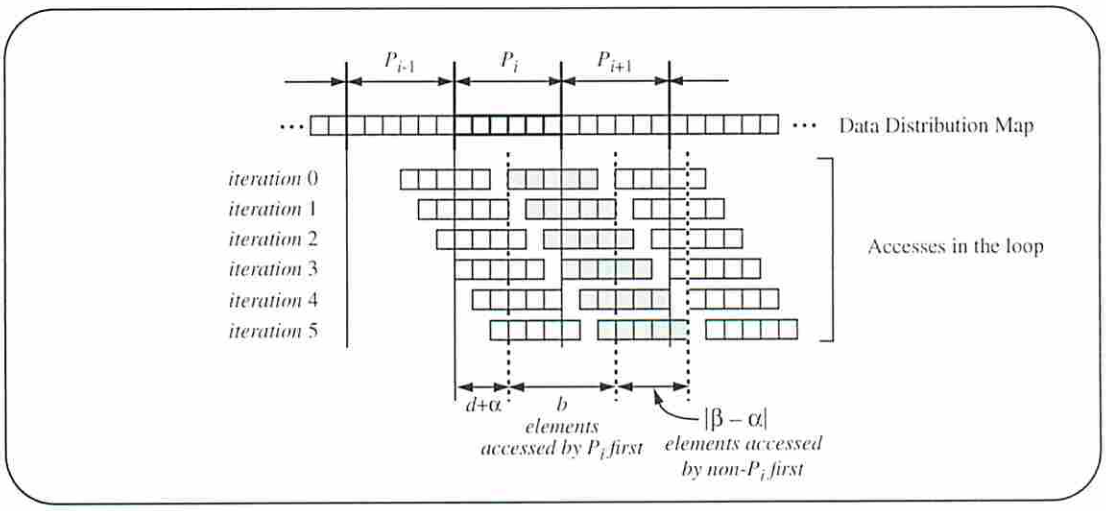


Figure 44. Data accesses of $E_{VIC\text{-}M}$.

Consider the previous write statement is of type E_L, $E_{VIC\text{-}S}$ or $E_{VIC\text{-}M}$ and the current write statement is of type $E_{LV\text{-}S}$ or $E_{LV\text{-}M}$. In this case, no matter how the modified elements may be distributed among processors, each access of P_i would produce a TSM with the probability of $O((P-1)/P)$. The number of TSMs that P_i experiences during its execution of write operation with respect to B_{Pi} is proportional to $O(b)$ or $O(b \times |\beta - \alpha|)$ for $E_{LV\text{-}S}$ or $E_{LV\text{-}M}$, respectively.

Finally, if the previous write statement is of type $E_{LV\text{-}S}$ or $E_{LV\text{-}M}$, every data access of P_i produces a TSMs with the probability of $O((P-1)/P)$, regardless of the current access type. This is because of the non-deterministic distribution of modified data elements as a result of the previous write statement.

From the number of TSMs observed by P_i during its write accesses with respect to the elements in B_{Pi} (Table 19), the total number of TSMs can be computed using the effects of the data set size and the number of processors as discussed in Section 5.4.1.

	E_L	$E_{VIC\text{-}S}$	$E_{VIC\text{-}M}$	$E_{LV\text{-}S}$	$E_{LV\text{-}M}$
E_L	Estimation using access parameters			$O(b)$	
$E_{VIC\text{-}S}$	(d, α, β)			$O(b)$	
$E_{VIC\text{-}M}$	Estimation using access parameters + $O(\lvert\beta - \alpha\rvert)$			$O(b + \lvert\beta - \alpha\rvert)$	
$E_{LV\text{-}S}$	$O(b)$			$O(b)$	
$E_{LV\text{-}M}$	$O(b \times \lvert\beta - \alpha\rvert)$			$O(b \times \lvert\beta - \alpha\rvert)$	

Table 19. Summary of TSM on write operation by P_i with respect to B_{Pi}.

5.4.4.5 Algorithm for TSM Estimation

In Sections 5.4.4.3 and 5.4.4.4, the number of TSMs observed by P_i during its write accesses with respect to the elements in B_{Pi} were completely explained and tabulated. In addition, Section 5.4.1 showed the way to expand the number of TSMs to reflect effects of data set size and number of processors. This knowledge is essential in measuring the total number of TSMs. In this section, the formal algorithm for estimating the total number of TSMs is introduced.

```
/* Given static sharing information */
/* Given behavioral sharing description where
   the statements are serially numbered */
1.    stmt_no = RC_Depth = RC_Idx = Crt_RC_Idx = 0 ;
2.    RC_Info[MAX_RC_Loops] = empty ;
3.    Global_TSM_List_for_DSS[MAX_RC_Idx] = empty ;
4.    Global_TSM_List_for_NP[MAX_RC_Idx] = empty ;
5.    while ( ! end_of_behavioral_sharing_description ) {
6.       if ( behavioral_stmt[stmt_no]->type == RC_BEGIN )
7.          RC_LOOP_BEGIN_Handler (stmt_no) ;
8.       else if ( behavioral_stmt[stmt_no]->type == RC_END )
9.          RC_LOOP_END_Handler (stmt_no) ;
10.      else /* regular statements */ {
11.         compute magnitude orders of data set size and
               number of processors using access parameters or
               in the way shown in Sections 5.4.4.3 and 5.4.4.4 ;
12.         if ( RC_Depth > 0 )
13.            add the results to crt RC_Info TSM lists ;
14.         else
15.            add the results to Global TSM Lists ;
16.         if ( behavioral_stmt[stmt_no]->op == WRITE )
17.            change data modification state ;
18.      }
19.      stmt_no++ ;
20.   }
21.   sum up magnitude orders in
         Global_TSM_List_for_DSS and in
         Global_TSM_List_for_NP ;
22.   return final results to users ;
```

Figure 45. Algorithm for TSM measurement (Main routine).

Figure 45 presents the main routine for measuring the number of TSM in terms of data set size and number of processors. The static information and the APD are assumed to be provided beforehand. The statements in the APD are supposed to be serially numbered as in Figure 32 (b). There are a few variables and data structures needed in the algorithm. All of them are global in the program scope. The `stmt_no` is used to point to the statements in the APD. The `RC_Idx` assigns a unique number to each *RC* loop found in the APD. The `RC_Depth` denotes the depth of the current *RC* loop and the `Crt_RC_Idx` is defined to store the `RC_Idx` of current *RC* loop. The initial values of all these variables are zero (Line 1).

There are two *list* data structures (`Global_TSM_List`) that store the estimated number of TSMs in terms of data set size and number of processors. They are used to store the number of TSMs measured from the statements that are not within an *RC* loop and the total number of TSM measured inside an (nested) *RC* loop. Finally, the data structure called `RC_Info` is used to store the information of each *RC* loop including the `stmt_no` in the APD and the size of the *RC* loop. The `RC_Info` also includes two lists to maintain the amount of TSMs measured from the statements within the current *RC* loop. All these data structures are initialized to be empty.

The main body of the algorithm starts from line 7. The loop is executed until the end of the APD is detected. Every statement is checked if it is the beginning of an *RC* loop (Line 6), the end of an *RC* loop (line 8) or a regular data access statement (line 10). For the first two cases, proper handler subroutines are invoked (lines 9, 11) which will be explained shortly.

The actual estimation of the number of TSMs for each APDS is done in line 11. The methodology is completely described in Sections 5.4.4.3 and 5.4.4.4. If the current statement is within an *RC* loop, the estimation results are stored into the corresponding lists of `RC_Info[Crt_RC_Idx]` (line 13). If the current statement is not surrounded by

```
1.     void RC_LOOP_BEGIN_Handler (stmt_no)
2.     {
3.        RC_Info[RC_Idx]->stmt_no = stmt_no ;
4.        RC_Info[RC_Idx]->size = RC loop size;
5.        RC_Info[RC_Idx]->TSM_List_for_DSS = empty ;
6.        RC_Info[RC_Idx]->TSM_List_for_NP = empty ;
7.        if ( RC_Depth > 0 )
8.           RC_Info[RC_Idx]->parent = Crt_RC_Idx ;
9.        else
10.          RC_Info[RC_Idx]->parent = None ;
11.       Crt_RC_Idx = RC_Idx ;
12.       RC_Idx++ ;
13.       RC_Depth++ ;
14.    }
```

Figure 46. Algorithm for TSM measurement (RC_LOOP_BEGIN_HANDLER).

any *RC* loop, the results are added to the list of `Global_TSM_Lists` (line 15). After TSM estimation, the current statement is checked if it is a write operation to update the modified data distribution among processors (lines 16, 17).

Finally, the entries in `Global_TSM_Lists` are the number of TSMs expressed in terms of the data set size and number of processors. They are all summed up to yield the total number of TSMs.

Figure 46 is the subroutine performing the operations needed when a *new RC* loop is started. The variable `RC_Idx`, which is monotonously increasing, points to a new space of data structure `RC_Info`. The information regarding the new *RC* loop are stored into `RC_Info[RC_Idx]`. At the beginning of *RC* loop, the lists of TSM for data set size and for number of processors are both empty (lines 3 to 6). All TSM estimation results from the statements within this *RC* loop are stored into the lists as specified in lines 12 and 13 of Figure 45.

It is checked if there was any other *RC* loop surrounding the current *RC* loop (nested *RC* loop). If so, `Crt_RC_Idx` which is the `RC_Idx` of previous *RC* loop is recorded into the *parent* field of `RC_Info[RC_Idx]`. The goal for this is as follows. Once the estimation of TSM from the statements within a *child RC* loop is finished, the total number of TSM within the child *RC* loop is stored into the `TSM_List_for_DSS` and `TSM_Lict_for_NP` of the parent *RC* loop. This will be shown in detail when we introduce the `RC_LOOP_END_HANDLER` subroutine.

After the initialization, the `Crt_RC_Idx` pointing to the current entry of the *RC* loop data structure (`RC_Info`) is updated. Similar cases are applied to the variables `RC_Idx` and `RC_Depth` whose values are used for the next *RC* loop to be opened within the current *RC* loop.

The last part of the algorithm for TSM estimation is the `RC_LOOP_END_HANDLER` in Figure 47 which is executed when the current *RC* loop is terminated. First of all, the entries in `TSM_List_for_DSS` and `TSM_List_for_NP` of the current *RC* loop are added together. The results are the number of TSMs measured during *one* iteration of the current *RC* loop. To see how these results should be dealt with, let's assume that there is at least one write statement within the *RC* loop.

In the first iteration, the number of TSMs observed from the read statements ahead of the first write in the current *RC* loop are measured based on the modified data distribution due to the most recent write statement executed before entering current *RC* loop. At the end of the first iteration, we must have different modified data distribution due to the last write statement within the *RC* loop. Then, in the second iteration of the *RC* loop, the occurrences of TSMs during the executions of the read statements ahead of the first write must be different from what we had in the first iteration. Since the modified data distribution at the end of every *RC* loop iteration is the same, the numbers of TSM estimated in the second and later iterations are the same. In consequence, if the size of an *RC* loop is s, the total estimated number of TSMs is given by

$$TSM_{Total} = TSM_{first} + (TSM_{second} \times (s-1))$$

where TSM_{first} is the number of TSMs in the first iteration and TSM_{second} is the number of TSMs measured in the second iteration of the *RC* loop.

If there is no write statement in an *RC* loop, the total TSM measured in the *RC* loop of size s becomes

$$TSM_{Total} = TSM_{first} \times s .$$

In the algorithm, lines 2 to 17 compute the number of TSMs when there is at least one write statement in the *RC* loop. Note that the effects of the number of processors on TSM are not multiplied by the *RC* loop size. This is because while the *RC* loop size may be dependent on data set size, it is not dependent on the number of processors. The lines 16 and 17 in Figure 47 should be replaced with the following if there is no read statement in the *RC* loop.

```
16.  Total_Order_of_DSS =Tmp_T_Order_of_DSS[0]*
                              RC_Info[Crt_RC_Idx]->size;
17.  Total_Order_of_NP =Tmp_T_Order_of_NP[0];
```

Once the total amount of TSMs for a whole *RC* loop is computed, the results must be saved. If the *RC* loop has its parent *RC* loop, the results are added to the `TSM_List_for_DSS` and `TSM_Lict_for_NP` of the parent *RC* loop (lines 23 to 27) since the current *RC* loop is regarded as *one* statement by its parent *RC* loop. If no parent *RC* loop is found, the results are added into the `Global_TSM_List_for_DSS` and `Global_TSM_List_for_NP` (lines 25 to 30).

Finally, proper bookkeeping operations including the decrement of `RC_Depth` and adjustment of `Crt_RC_Idx` are done.

```
1. void RC_LOOP_END_Handler (stmt_no) {
2.     sum up the entries in
3.            RC_Info[Crt_RC_Idx]->Order_of_DSS_List and in
4.            RC_Info[Crt_RC_Idx]->Order_of_PN_List ;
5.     and save them into
6.            Tmp_T_Order_of_DSS[0] and
7.            Tmp_T_Order_of_NP[0] ;
8.     clear RC_Info[Crt_RC_Idx]->Order_of_DSS_List and
9.            RC_Info[Crt_RC_Idx]->Order_of_PN_List ;
10.    run current RC loop one more iteration
11.           from  tmp_stmt_no = RC_Info[RC_Idx]->stmt_no
12.           to    tmp_stmt_no = stmt_no - 1 ;
13.    ditto lines 2 through 9 using
14.           Tmp_T_Order_of_DSS[1] and
15.           Tmp_T_Order_of_NP[1] ;
16.    Total_Order_of_DSS = Tmp_T_Order_of_DSS[0] +
          ( Tmp_T_Order_of_DSS[1] * RC_Info[Crt_RC_Idx]->size );
17.    Total_Order_of_NP =  Tmp_T_Order_of_NP[0] +
                            Tmp_T_Order_of_NP[1] ;
18.    if ( RC_Depth > 1 ) {
20.       add Total_Order_of_DSS into
21.       RC_Info[RC_Info[Crt_RC_Idx]->parent]->TSM_List_for_DSS
22.       add Total_Order_of_NP into
23.       RC_Info[RC_Info[Crt_RC_Idx]->parent]->TSM_List_for_NP
24.    }
25.    else {
26.       add Total_Order_of_DSS into
27.          Global_TSM_List_for_DSS and
28.       add Total_Order_of_NP into
29.          Global_TSM_List_for_NP
30.    }
31.    RC_Depth-- ;
32.    Crt_RC_Idx = RC_Info[RC_Info[Crt_RC_Ind]->parent] ;
33.}
```

Figure 47. Algorithm for TSM measurement (RC_LOOP_END_HANDLER).

Chapter 6

PREDICTION RESULTS BASED ON DATA-SHARING ANALYSIS

This section provides all simulation and prediction results. Simulations are performed in two aspects; one for variable data set size and the other for variable number of processors. At first, when we vary the data set sizes, the number of processors is fixed at 8. The small and large data sets are identical to those given in Table 3. Second, as the number of processors is changing, the data set sizes in applications take distinct values. They are summarized in Table 20.

Benchmark	Sample Number of Processors	Fixed Data Set Size	Prediction	
			NP-1	NP-2
LU	2, 4, 8, 16	256	32	64
MP3D		32K		
WATER		256		
OCEAN		130		
FFT		2^{16}		
BARNES		1024		
RADIX		128K keys		

Table 20. Number of processors used to find fitting functions.

At the beginning of each subsection, brief algorithmic descriptions for the programs and representative shared variables are introduced with their static and behavioral sharing information for the shared variables in every application are also provided. An additional table follows after the sharing information that includes the effects of data size and number of processors on the number of data references and the number of various types of cache misses.

Besides, a set of graphs for simulation results obtained from the small data sets are provided in each benchmark. Even though we are modeling and predicting the number of misses categorized into PCM, TSM and FSM only, the graphs were drawn for each type of cache misses (PCM, CTSM, CFSM, PTSM, and PFSM) to help understand detail behavioral charactersitics in the application. In the graphs, the x-axis and y-axis stand for the data set size and the number of the events, respectively.

Finally, two tables per application will show the prediction results for large data sets and large number of processors. In tables, the empirical models for individual shared variables are found. At the bottom of tables, models for overall performance metrics are given which are obtained by summing up the models for all shared variables.

At the end of this section, the empirical models and prediction results for large data sets and large number of processors of all applications are collected together. The last table will compare the prediction results for data set size obtained in Section 3 and those obtained in this section.

6.1 LU

Shared variables: LU application performs the LU-decomposition of a dense matrix. The problem size N is the number of rows of the matrix as well as the number of iterations. There are two large matrices, `A` and `L`, of size N-by-N and one vector, *`this-Pivot`*, of size N are modifiable. `A` and `L` are stored column-wise in shared memory space. Columns *i* of `A` and `L` are statically allocated to the *Home*, processor *i* `mod` *P*, where *P* is the number of processors. Each column of `A` is accessed by its *Home* only, whereas a column *i* of `L` and an element *i* of `thisPivot` are first modified once by the *Home* and then read by all the *Homes* of columns with indices greater than *i*.

Size of an element:	8 bytes
Dimension:	2
Size in dimension 1:	N
Size in dimension 2:	N
Partition in dimension 1:	Cyclic
Partition in dimension 2:	None

(a) Static sharing information

```
1. RC-loop (N times) BEGIN
2.     { $E_L$: and $E_L$: }, Write;
3. RC-loop END
```

(b) Behavioral sharing description

Figure 48. Static and behavioral sharing description for array `A` in LU.

Size of an element:	8 bytes
Dimension:	2
Size in dimension 1:	N
Size in dimension 2:	N
Partition in dimension 1:	Cyclic
Partition in dimension 2:	None

(a) Static sharing information

```
1. { $E_L$: and $E_L$: }, Write;
2. RC-loop (N times) BEGIN
3.     { $E_{VIC\text{-}S}$: and $E_L$: }, Write;
4. RC-loop END
```

(b) Behavioral sharing description

Figure 49. Static and behavioral sharing description for array `L` in LU.

```
Size of an element:            4 bytes
Dimension:                     1
Size in dimension 1:           N
Partition in dimension 1:      Cyclic
```

(a) Static sharing information

```
1. $E_L$:, Write;
2. RC-loop (N times) BEGIN
3.     $E_{VIC-S}$:, Write;
4. RC-loop END
```

(b) Behavioral sharing description

Figure 50. Static and behavioral sharing description for array `thispivot` in LU.

Array	Metrics	Scaling Effects	
		Data Set	Processor
A	References	$O(N^3)$	-
	PCM	$O(N^2)$	-
L	References	$O(N^3)$	-
	PCM	$O(N^2)$	-
	TSM	$O(N^2)$	$O(P)$
This	References	$O(N^2)$	-
	PCM	$O(N)$	-
	TSM / FSM	$O(N)$	$O(P)$

Table 21. Effects of data set size and number of processors.

Simulation results: Overall miss rate in LU is dominated by cold misses of arrays `A` and `L` (Figure 51 (b), (c)). Only a few sharing misses are observed in `thisPivot` (Figure 51 (d), (e)). The static and behavioral sharing information of arrays are given in Figures 49 to 50. They show that the first accesses to every array are made by the Home and the operation type is a write. This implies that the home is the only processor experiencing PCM since subsequent accesses of all remote processors are made to the elements modified by the Home. Therefore, the numbers of PCM of all arrays are not affected by the number of processors (Table 21). Instead, those accesses bring in TSM on arrays `A` and `thisPivot`. *RC* loops surrounding the array accesses result in $O(N^3)$ and $O(N^2)$ data references for two-dimensional and one-dimensional arrays, respectively. Figures 49 to 50 and Table 21 show that overall PCM and TSM of arrays *A* and *L* increase at the speed of $O(N^2)$ whereas the number of references increases at the speed of $O(N^3)$. Therefore the miss rate in LU will vanish quite rapidly as the data set size increases.

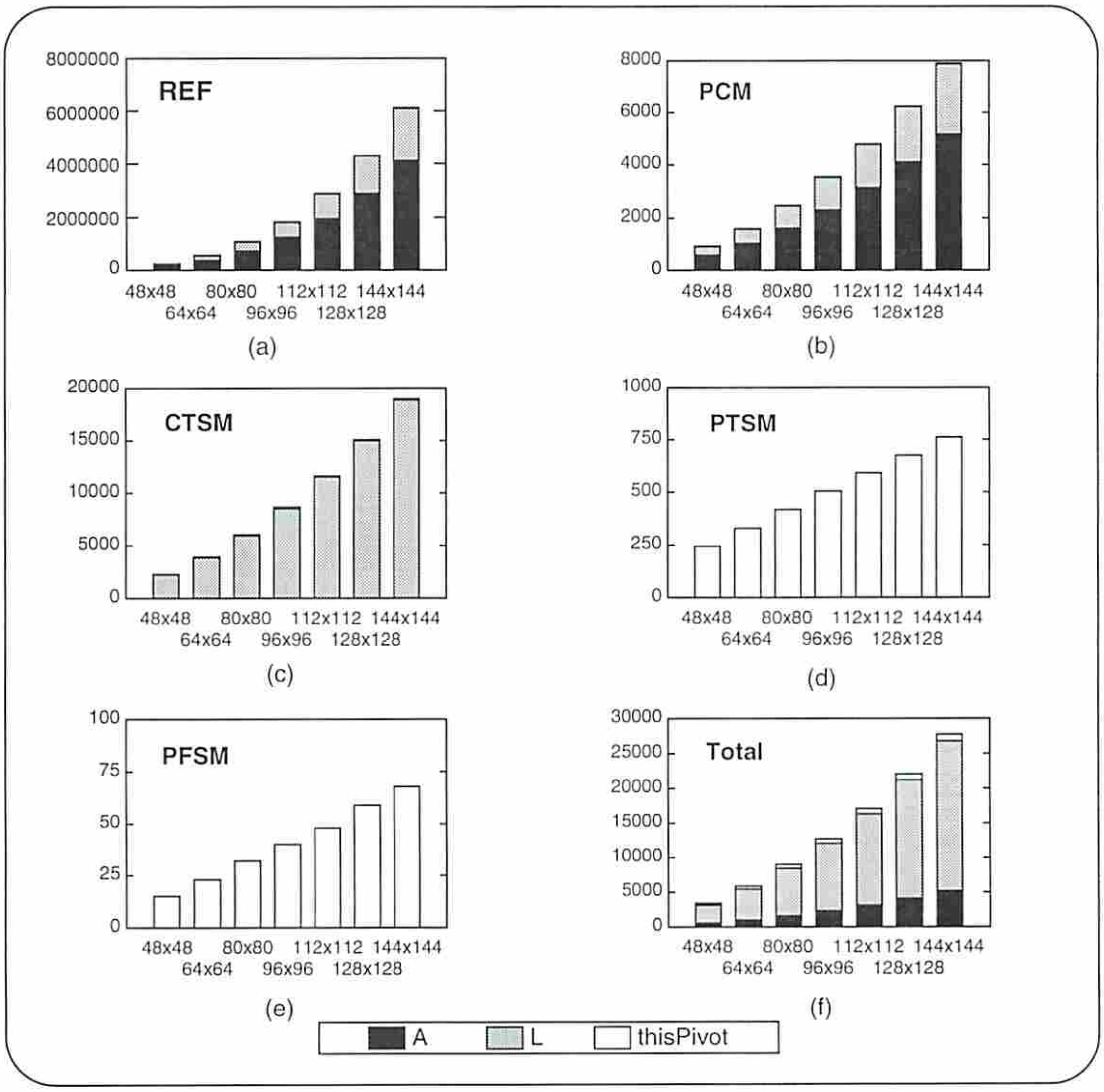


Figure 51. Simulation results in LU.

Prediction results: In LU, as shown in Tables 22 and 23, due to the regularity of data communications among processors, the prediction errors of each type of misses are extremely small. In particular, the predicted values for the total *miss rates* for large data sets and 64 processors are perfect.

			Prediction-1: 288			Prediction-2: 512		
		Empirical Model	simulation	prediction	error%	simulation	prediction	error%
A	Reference	$1.332N^3+6.358N^2+-27.840N+891.126$	32349490	32341280	0.025	180532208	180444752	0.048
	PCM	$0.250N^2-0.036N+4.100$	20739	20745	-0.030	65538	65570	-0.050
L	Reference	$0.667N^3-0.032N^2+2.275N-85.238$	15925054	15925757	-0.004	89478144	89485728	-0.008
	PCM	$0.125N^2+0.772N-2.472$	10583	10580	0.026	33151	33136	0.043
	CTSM	$0.875N^2+5.272N-40.482$	74049	74046	0.004	232032	232010	0.009
thisPivot	Reference	$1.000N^2+2.000N-2.968$	83517	83517	0.000	263165	263165	0.000
	PCM	$0.125N+1.000$	37	37	0.000	65	65	0.000
	TSM	$0.875N+3.000$	1793	1793	0.000	3193	3196	-0.093
	FSM	$0.562N-12.932$	148	148	-0.552	276	274	0.500
Total	Reference	$1.999N^3+7.327N^2-23.566N+803.06$	48358061	48350554	0.016	270273517	270193645	0.030
	PCM	$0.375N^2+0.861N+2.629$	31359	31362	-0.010	98754	98771	0.000
	TSM	$0.875N^2+11.543N-51.624$	75842	75839	0.003	235225	235208	0.007
	FSM	$0.562N-12.932$	148	148	0.000	276	274	0.725
	Total Miss	$1.250N^2+12.967N-61.928$	107349	107349	0.000	334255	334253	0.000
	MissRate(%)	$\frac{1.250N^2 + 12.967N - 61.928}{1.999N^3 + 7.327N^2 - 23.5566N + 803.06}$	**0.2220**	**0.2220**	**0.000**	**0.1237**	**0.1237**	**0.000**

Table 22. Empirical models and prediction results for data set size (LU).

			Prediction-1: 32 processors			Prediction-2: 64 processors		
		Empirical Model	simulation	prediction	error%	simulation	prediction	error%
A	Reference	22764012	22764012	22764012	0.000	22764012	22764012	0.000
	PCM	16384	16384	16384	0.000	16384	16384	0.000
L	Reference	11184638	11184638	11184638	0.000	11184638	11184638	0.000
	PCM	8256	8256	8256	0.000	8256	8256	0.000
	TSM	$8057.018 * P - 6689.170$	254511	251135	1.326	508959	508960	-0.000
thisPivot	Reference	66045	66045	66045	0.000	66045	66045	0.000
	PCM	32	32	32	0.000	32	32	0.000
	TSM	$221.650 * P - 74.557$	7325	7018	3.476	14111	14111	0.000
	FSM	$10.032 * P - 3.019$	318	318	0.000	639	639	0.000

Table 23. Empirical models and prediction results for number of processors (LU).

Total	Reference	34014695	34014695	34014695	0.000	34014695	34014695	0.000
	PCM	24672	24672	24672	0.000	24672	24672	0.000
	TSM	8278.668 * P - 6763.727	261836	258153	1.406	523070	523071	0.000
	FSM	10.032 * P - 3.019	318	318	0.000	639	639	0.000
	Total Miss	8288.700 * P - 6766.746	286828	283143	1.284	548381	548382	0.000
	MissRate(%)	$\frac{8288.700P - 6766.746}{34014695}$	**0.8432**	**0.8324**	**1.280**	**1.6121**	**1.6121**	**0.000**

Table 23. Empirical models and prediction results for number of processors (LU).

6.2 MP3D

Shared Variables: MP3D repeatedly calculates the positions and velocities of particles during a preset number of time steps. The data set size, N, is the number of molecules in the one dimensional array `Particles`. The size of each element of `Particles` is 36 bytes. 16 molecules in `Particles` array form a clump and the clumps are allocated to processors statically so that particle *i* is allocated to processor (*i*/ 16) `mod` *P*, where *P* is the number of processors. `Cells` is a three dimensional array of 48 byte elements, each of which is a location of the space where the molecules are moved. Unlike `Particles`, elements in `Cells` are shared by all processors. When a molecule is moved, the `Cells` component is modified.

In each iteration, every `Particles` element is accessed and moved in cell space. Each time a particle is moved, an element of `Cells` is updated to reflect the changes of the three-dimensional coordinates of the particle. The shared memory locations accessed by a processor are largely unpredictable. When a processor moves one particle in cell space, a randomly generated number decides if the processor should simulate a collision. These collisions are the only source of coherence misses for `Particles`. Moreover, the memory location of an element of `Cells` is selected by the space coordinates of the particle and is independent of the processor or the index of the particle. This unpredictable behavior might cause serious prediction errors. However, robust estimation is rooted in statistics and the basic algorithmic step does not change as the data set size increases.

Static and behavioral sharing description for MP3D shared arrays are in Figure 52 and 53. As shown there, the `Particles` elements are mainly accessed by the Home while `Cells` elements are mostly accessed by remote processors. The *IC* loop (or, FORALL loop) for `Particles` (line 1 in Figure 52) is regarded as an *RC* loop of size N (line 3 in Figure 53). There is no processor effect on the number of PCM for `Cells` since the first access is a write operation. The accesses of type $E_{LV\text{-}S}$ result in the magnitude order of $O((P-1)/P)$ as shown in Table 24.

Size of an element:	36 bytes
Dimension:	1
Size in dimension 1:	N
Partition in dimension 1:	Block-Cyclic, Block=16

(a) Static sharing information

```
1. RC-loop (10 times) BEGIN
2.     $E_L$: Read;
3.     $E_L$: Read;
4.     $E_L$: Write;
5.     $E_L$: Read;
6.     $E_L$: Read;
7.     $E_{LV-S}$: pn, Read;
8.     $E_L$: Read;
9.     $E_{LV-S}$: pn, Read;
10. RC-loop END
```

(b) Behavioral sharing description

Figure 52. Static and behavioral sharing description for array `Particles` in MP3D.

Simulation results: When particles collide, a processor may modify the coordinates of a particle which is allocated to a different processor. However, due to the low collision probability, the resulting number of true sharing misses of `Particles` is not outstanding (Figure 54 (e))). Accesses to 36-byte `Particles` elements result in false sharing misses (Figure 54 (f)) because the cache block size is normally a power of two.

```
Size of an element:          48 bytes
Dimension:                   3
Size in dimension 1:         14
Size in dimension 2:         24
Size in dimension 3:         7
Partition in dimension 1:    Cyclic
Partition in dimension 2:    Cyclic
Partition in dimension 3:    None
```

(a) Static sharing information

```
1. RC-loop (10 times) BEGIN
2.     { $E_L$: and $E_L$: and $E_L$: } Write;
3.     RC-loop (N times) BEGIN
4.         { $E_{LV-S}$: and $E_{LV-S}$: and $E_{LV-S}$: } Read;
5.         { $E_{LV-S}$: and $E_{LV-S}$: and $E_{LV-S}$: } Write;
6.         { $E_{LV-S}$: and $E_{LV-S}$: and $E_{LV-S}$: } Write;
7.     RC-loop END
8. RC-loop END
```

(b) Behavioral sharing description

Figure 53. Static and behavioral sharing description for array `Cells` in MP3D.

Array	Metrics	Scaling Effects	
		Data Set	Processor
Particles	References	$O(N)$	-
	PCM	$O(N)$	$O((P-1)/P)$
	TSM / FSM	$O(N)$	$O((P-1)/P)$
Cells	References	$O(N)$	-
	PCM	$O(N)$	-
	TSM / FSM	$O(N)$	$O((P-1)/P)$

Table 24. Effects of data set size and number of processors.

Most of true and false sharing misses are observed in `Cells` (Figure 54 (e), (f)) while negligible amount of cold misses (PCM, CTSM and CFSM) are experienced due to the small size of array `Cells`. Thus, in Table 24, the orders of the numbers of PCM, CTSM and CFSM of `Cells` are omitted because `Cells` has the constant size of 94,080

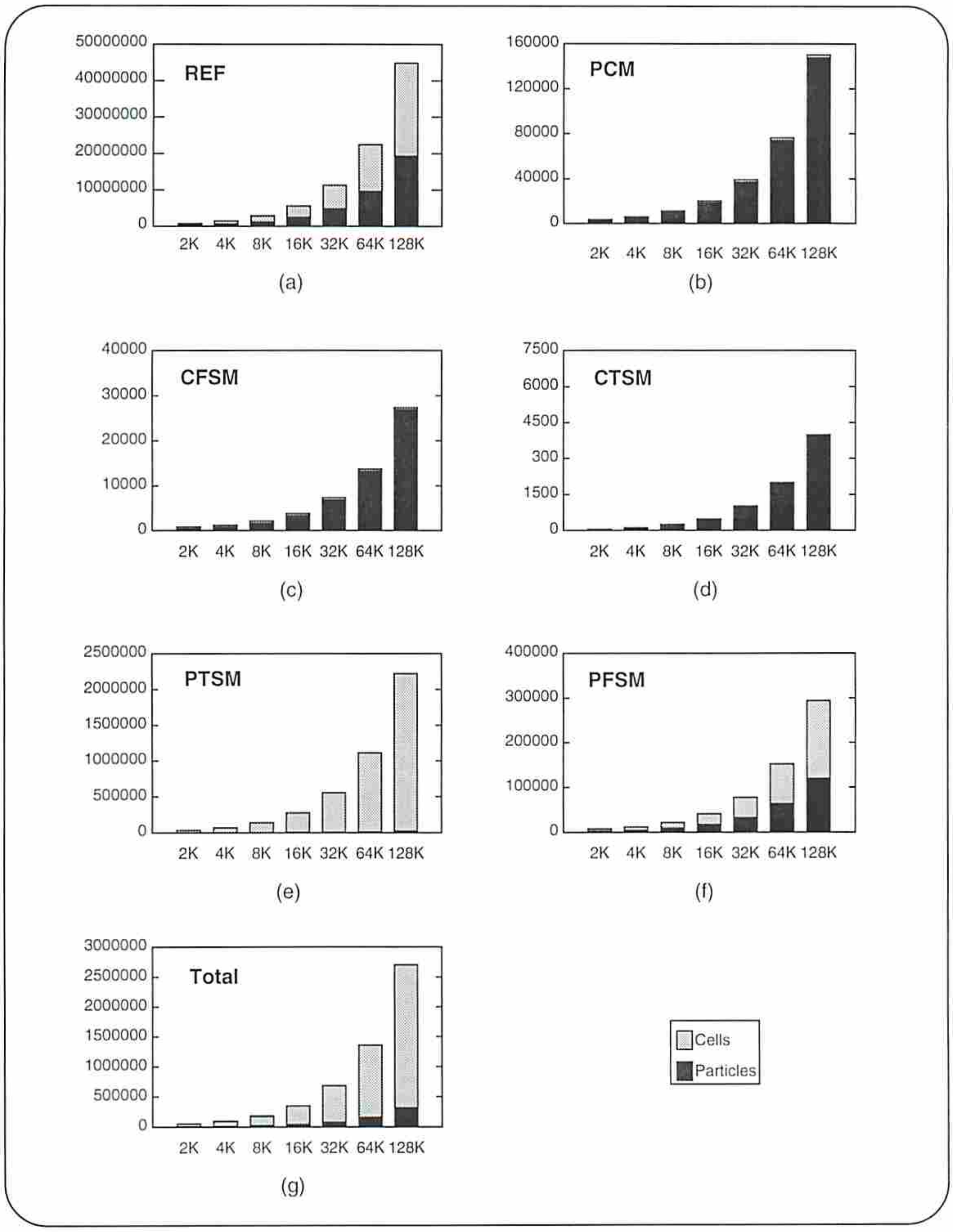


Figure 54. Simulation results in MP3D.

bytes.The collisions in `Ares` cause true sharing misses (Figure 54 (e), (f)) and, unlike `Particles`, every particle in reservoir space experience exact one collision.

			Prediction-1: 256K			Prediction-2: 512K		
		Empirical Model	simulation	prediction	error%	simulation	prediction	error%
Particles	References	146.780 N + 686.656	38475880	38476804	-0.002	76961752	76954296	0.009
	PCM	1.127 N +1.314	295420	295410	0.003	590648	590818	-0.028
	TSM	0.170 N -1.703	44370	44365	0.011	89578	88732	0.944
	FSM	1.101 N +1608.389	292524	290424	1.064	543500	579251	-6.578
Cell	References	195.085 N + 42710.3	51182884	51183032	-0.000	102322920	102323352	-0.000
	TSM	16.821 N -1034.9	4411209	4408433	0.062	8840015	8817901	0.250
	FSM	1.319 N + 2071.14	345947	347954	-0.580	686345	693838	-1.091
Total	Reference	341.865 N + 43396.956	89658764	89659836	-0.001	179284672	179277648	0.004
	PCM	1.127 N +1.314	295420	295410	0.003	590648	590818	-0.028
	TSM	16.991 N -1036.603	4455579	4452798	0.062	8929593	8906633	0.257
	FSM	2.420 N + 3679.529	638471	638378	0.015	1229845	1273089	-3.516
	Total Miss	20.538 N + 2644.240	5389470	5386586	0.054	10750086	10770540	-0.190
	MissRate(%)	$\frac{20.538N + 2644.240}{341.865N + 43396.956}$	**6.0111**	**6.0078**	**0.055**	**5.9961**	**6.0077**	**-0.193**

Table 25. Empirical models and prediction results for data set size (MP3D).

			Prediction-1: 32 processors			Prediction-2: 64 processors		
		Empirical Model	simulation	prediction	error%	simulation	prediction	error%
Particles	References	4808082	4808082	4808082	0.000	4808082	4808082	0.000
	PCM	-63.942 (P-1) / P + 36971.992	36907	36910	-0.008	36909	36909	0.000
	TSM	7030.883 (P-1) / P - 544.440	5972	6266	-4.935	6394	6376	0.272
	FSM	26709.281 (P-1) / P + 17392.337	43267	43266	0.000	43977	43684	0.665
Cell	References	6434519	6434519	6434519	0.000	6434519	6434519	0.000
	TSM	790231.812 (P-1) / P - 114912.906	654388	650624	0.575	695470	662971	4.672
	FSM	18123.068 (P-1) / P + 34385.582	43762	51942	-18.692	686345	693838	-1.091
Total	Reference	11242601	11242601	11242601	0.000	11242601	11242601	0.000
	PCM	-63.942 (P-1) / P + 36971.992	36907	36910	-0.008	36909	36909	0.000
	TSM	797262.688 (P-1) / P --115457.344	660360	656890	0.525	701864	669347	4.633
	FSM	26711.701 (P-1) / P + 51777.922	87029	95208	-9.398	730322	737522	-0.986
	Total Miss	823910.438 (P-1) / P + 797203.062	784296	789008	0.054	1469095	1443778	-0.190
	MissRate(%)	$\frac{823901.438 \times (P-1)/P + 797203.062}{11242601}$	**6.9761**	**7.0180**	**-0.601**	**13.0672**	**12.8420**	**1.723**

Table 26. Empirical models and prediction results for number of processors (MP3D).

Prediction results: In MP3D, shared memory locations accessed by a processor are decided randomly. This unpredictable behavior may cause serious variations in the observations. Using the robust estimation method, even if some of the observations include *outliers*, the effect of the outlier was eliminated. Nevertheless, the prediction results in Tables 25 and 26 are highly accurate and the prediction errors of overall miss rates are mostly near 1% for larger data sets and number of processors.

6.3 WATER

Shared variables: WATER performs an N-body molecular dynamics simulation of the forces and potentials in a system of water molecules. The data set size parameter N is the number of molecules in array *VAR*. The molecules are represented by 600 byte objects in *VAR*, which is the only shared variable in WATER. $1/P$ molecules are allocated statically to each processor. In each iteration every element of *VAR* is accessed to compute the intra-molecular and inter-molecular interactions with $N/2$ elements ahead of it (Figure 55, lines 8, 9 17).

The intra- and inter-interactions of atoms are simulated during a predetermined number of time steps which is a constant. Thus the number of references is $O(N^2)$ and the PCM count is $O(N)$. In addition, for each element of *VAR*, a processor must accessed $N/2$ components, which were mostly modified by the Home of each element. Thus, the TSM count is $O(N^2)$. The number of cache misses is estimated as $O(P)$ due to the accesses to the consecutive elements whose access distances from the center of computation are larger than the partition block size ($N/2 > N/P$).

Simulation results: There are only PCM, CTSM or PTSM and no FSM.

Prediction results: Due to regular program behaviors, the prediction is precise.

```
Size of an element:         600 bytes
Dimension:                  1
Size in dimension 1:        N
Partition in dimension 1:   Block
```

(a) Static sharing information

```
1.  RC-loop (10 times) BEGIN
2.      E_L: Read;
3.      E_L: Write;
4.      E_L: Read;
5.      E_L: Write;
6.      E_L: Read;
7.      E_L: Write;
8.      E_VIC-M: 0, 0, N/2, Read;
9.      E_VIC-M: 0, 0, N/2, Write;
10.     E_L: Read;
11.     E_L: Write;
13.     E_L: Read;
14.     E_L: Write;
15.     E_L: Read;
16.     E_L: Write;
17.     E_VIC-M: 0, 0, N/2, Read;
18. RC-loop END
```

(b) Behavioral sharing description

Figure 55. Static and behavioral sharing description for array *VAR* in WATER.

Array	Metrics	Scaling Effects	
		Data Set	Processor
VAR	References	$O(N^2)$	-
	PCM	$O(N)$	$O(P)$
	TSM / FSM	$O(N^2)$	$O(P)$

Table 27. Effects of data set size and number of processors.

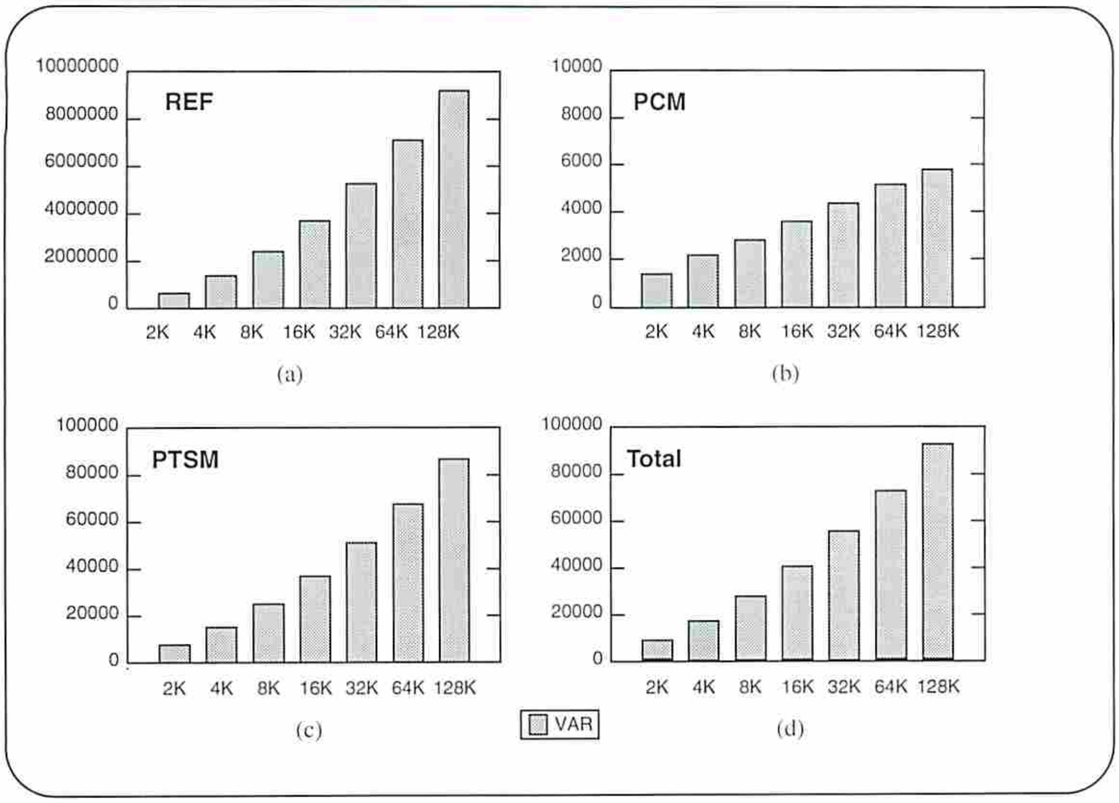


Figure 56. Simulation results in WATER.

			Prediction-1: 512K			Prediction-2: 1024K		
		Empirical Model	simulation	prediction	error%	simulation	prediction	error%
VAR/Total	Reference	$540.016N^2+2938.3N+644.47$	71035088	71044592	-0.013	282029536	282060832	-0.011
	PCM	$20.000N-0.001$	7200	7199	0.000	14401	14399	0.006
	TSM	$4.490N^2+131.164N-221.482$	629556	628841	0.114	2424102	2421590	0.104
	Total Miss	$4.490N^2+151.165N-221.482$	636756	636040	0.112	2438503	2435989	0.103
	MissRate(%)	$\frac{4.490N^2+151.164N-221.48}{540.016N^2-2938.3N+644.47}$	**0.8964**	**0.8953**	**0.122**	**0.8646**	**0.8636**	**0.115**

Table 28. Empirical models and prediction results for data set size (WATER).

			Prediction-1: 32 processors			Prediction-2: 64 processors		
		Empirical Model	simulation	prediction	error%	simulation	prediction	error%
VAR/Total	Reference	36138850	36138850	36138850	0.000	36138850	36138850	0.000
	PCM	5120	5120	5120	0.000	5120	5120	0.000
	TSM	4102.024 P + 295092.562	427572	426357	0.154	557622	557622	0.000
	Total Miss	4102.024 P + 300212.562	427572	431477	-0.913	562742	562742	0.000
	MissRate(%)	$\frac{4102.024P + 300212.562}{36138850}$	**1.1831**	**1.1939**	**-0.913**	**1.5572**	**1.5572**	**0.000**

Table 29. Empirical models and prediction results for number of processors (WATER).

6.4 OCEAN

Shared variables: The OCEAN program simulates the prediction of large-scale ocean movements. The algorithm is composed of many predefined constant time steps. In every time step several spatial partial differential equations are solved on sets of two-dimensional fixed-sized grids. Each set corresponds to an horizontal cross-section of the ocean basin. The solution method is a multi-grid Gauss-Seidel iteration with Successive Over Relaxation (SOR).

There are 25 double precision two-dimensional arrays that store discretized functions associated with the equations of the model. *q_multi* and *rhs_multi* are multi-level grids; their size is $O(N^2)$. Thus the number of cold misses is $O(N^2)$. In each iteration, two grids are accessed repetitively and the number of repetitions is independent of data set size. Hence, the number of references is $O(N^2)$. Data sharing is observed only at the partition boundaries, and thus the PTSM count is $O(N)$.

Except for the *q_multi* and *rhs_multi*, behavioral characteristics of arrays are uniform and simple. Furthermore, since two or three of them form another data structure, those data structures are considered instead of each array. The *q_multi* and *rhs_multi* implement the multi-level grids and the working level of the grids moves up and down according to whether the eddies and the mean flow reach to the balance or not. As the level is changed, does the grid size, too, and smaller grid size presents more data-sharing. The size of all shared variables are in Table 30. They are commonly $O(N^2)$.

Variables	fields	fields2	wrk1	wrk2	wrk3	wrk4	wrk5	wrk6	frcng	guess
VAR	$4N^2$	$2N^2$	$3N^2$	N^2	$3N^2$	$4N^2$	$4N^2$	N^2	N^2	$2N^2$

Table 30. Numbers of data elements of shared arrays in OCEAN.

Array	Metrics	Scaling Effects	
		Data Set	Processor
q_multi	References	$O(N^2)$	-
	PCM	$O(N^2)$	$O(\sqrt{P}^2)$
	TSM / FSM	$O(N)$	$O(\sqrt{P})$
rhs_multi	References	$O(N^2)$	-
	PCM	$O(N^2)$	$O(\sqrt{P}^2)$
	TSM / FSM	$O(N)$	$O(\sqrt{P})$
Other arrays	References	$O(N^2)$	-
	PCM	$O(N^2)$	$O(\sqrt{P}^2)$
	TSM / FSM	$O(N)$	$O(\sqrt{P})$

Table 31. Effects of data set size and number of processors.

Simulation results: Among many arrays, `q_multi` presents the most PTSM (Figure 60 (e)). But the number of the combined misses of other arrays including huge amount of PCM and PFSM (Figure 60 (b), (f)) exceeds the number of misses of `q_multi` (Figure 60 (g)).

Prediction results: OCEAN has been the most challenging program analytical model in two aspects. First, because of the huge memory requirement we could only collect sample for three small data sets. This is the minimum needed to estimate the three unknown coefficients of a polynomial of order N^2. Furthermore, the execution time of OCEAN is strongly dependent on the values of the data since it runs until the changes in

```
Size of an element:         8 bytes
Dimension:                  2
Size in dimension 1:        N
Size in dimension 2:        N
Partition in dimension 1:   Block
Partition in dimension 2:   Block
```

(a) Static sharing information

```
1. { $E_L$: and $E_L$: } Read;
2. RC-loop ( $O(1)$ times) BEGIN
3.    { $E_{VIC-S}$: 1 and $E_{VIC-S}$: 1} Read;
4.    { $E_L$: and $E_L$: } Write;
5.    { $E_{VIC-M}$: 0,0,3 and $E_{VIC-M}$: 0,0,3} Read;
6.    { $E_L$: and $E_L$: } Write;
7.    { $E_L$: and $E_L$: } Write;
8.RC-loop END
9. { $E_{VIC-S}$: 1 and $E_{VIC-S}$: 1} Read;
10.$E_L$: Write;
11.RC-loop ( $O(1)$ times) BEGIN
12.   { $E_L$: and $E_L$: } Read;
13.   RC-loop ( $O(1)$ times) BEGIN
14.      { $E_L$: and $E_L$: } Read;
15.      RC-loop ( $O(1)$ times) BEGIN
16.         { $E_{VIC-S}$: 1 and $E_{VIC-S}$: 1} Read;
17.         { $E_L$: and $E_L$: } Write;
18.         { $E_{VIC-M}$: 0,0,3 and $E_{VIC-M}$: 0,0,3} Read;
19.         { $E_L$: and $E_L$: } Write;
20.         { $E_L$: and $E_L$: } Write;
21.      RC-loop END
22.      { $E_L$: and $E_L$: } Write;
23.      { $E_L$: and $E_L$: } Read;
24.      { $E_L$: and $E_L$: } Write;
25.      { $E_L$: and $E_L$: } Read;
26.   RC-loop END
27.RC-loop END
```

(b) Behavioral sharing description

Figure 57. Static and behavioral sharing description for array `Q_multi` in OCEAN.

```
Size of an element:        8 bytes
Dimension:                 2
Size in dimension 1:       N
Size in dimension 2:       N
Partition in dimension 1:  Block
Partition in dimension 2:  Block
```

(a) Static sharing information

```
1. { $E_L$: and $E_L$: } Read;
2. { $E_L$: and $E_L$: } Write;
3. { $E_{VIC\text{-}S}$: 2 and $E_{VIC\text{-}S}$: 2} Read;
4. { $E_L$: and $E_L$: } Write;
5. RC-loop ( $O(1)$ times) BEGIN
6.     { $E_L$: and $E_L$: } Read;
7.     RC-loop ( $O(1)$ times) BEGIN
8.         { $E_L$: and $E_L$: } Read;
9.         RC-loop ( $O(1)$ times) BEGIN
10.            { $E_{VIC\text{-}S}$: 1 and $E_{VIC\text{-}S}$: 1} Read;
11.            { $E_L$: and $E_L$: } Write;
12.            { $E_{VIC\text{-}S}$: 1 and $E_{VIC\text{-}S}$: 1} Read;
13.            { $E_L$: and $E_L$: } Write;
14.        RC-loop END
15.        { $E_L$: and $E_L$: } Write;
16.        { $E_L$: and $E_L$: } Read;
17.    RC-loop END
18.RC-loop END
```

(b) Behavioral sharing description

Figure 58. Static and behavioral sharing description for other arrays in OCEAN.

```
Size of an element:         8 bytes
Dimension:                  2
Size in dimension 1:        N
Size in dimension 2:        N
Partition in dimension 1:   Block
Partition in dimension 2:   Block
```

(a) Static sharing information

```
1. { $E_L$: and $E_L$: } Write;
2. { $E_L$: and $E_L$: } Read;
3. RC-loop ( $O(1)$ times) BEGIN
4.     { $E_L$: and $E_L$: } Read;
5.     RC-loop ( $O(1)$ times) BEGIN
6.         { $E_L$: and $E_L$: } Read;
7.         RC-loop ( $O(1)$ times) BEGIN
8.             { $E_{VIC-S}$: 1 and $E_{VIC-S}$: 1} Read;
9.             { $E_L$: and $E_L$: } Write;
10.            { $E_{VIC-M}$: 0,0,3 and $E_{VIC-M}$: 0,0,3} Read;
11.            { $E_L$: and $E_L$: } Write;
12.            { $E_L$: and $E_L$: } Write;
13.        RC-loop END
14.        { $E_L$: and $E_L$: } Write;
15.        { $E_L$: and $E_L$: } Read;
16.        { $E_L$: and $E_L$: } Write;
17.        { $E_L$: and $E_L$: } Read;
18.    RC-loop END
19.RC-loop END
```

(b) Behavioral sharing description

Figure 59. Static and behavioral sharing description for array *Q_multi* in OCEAN.

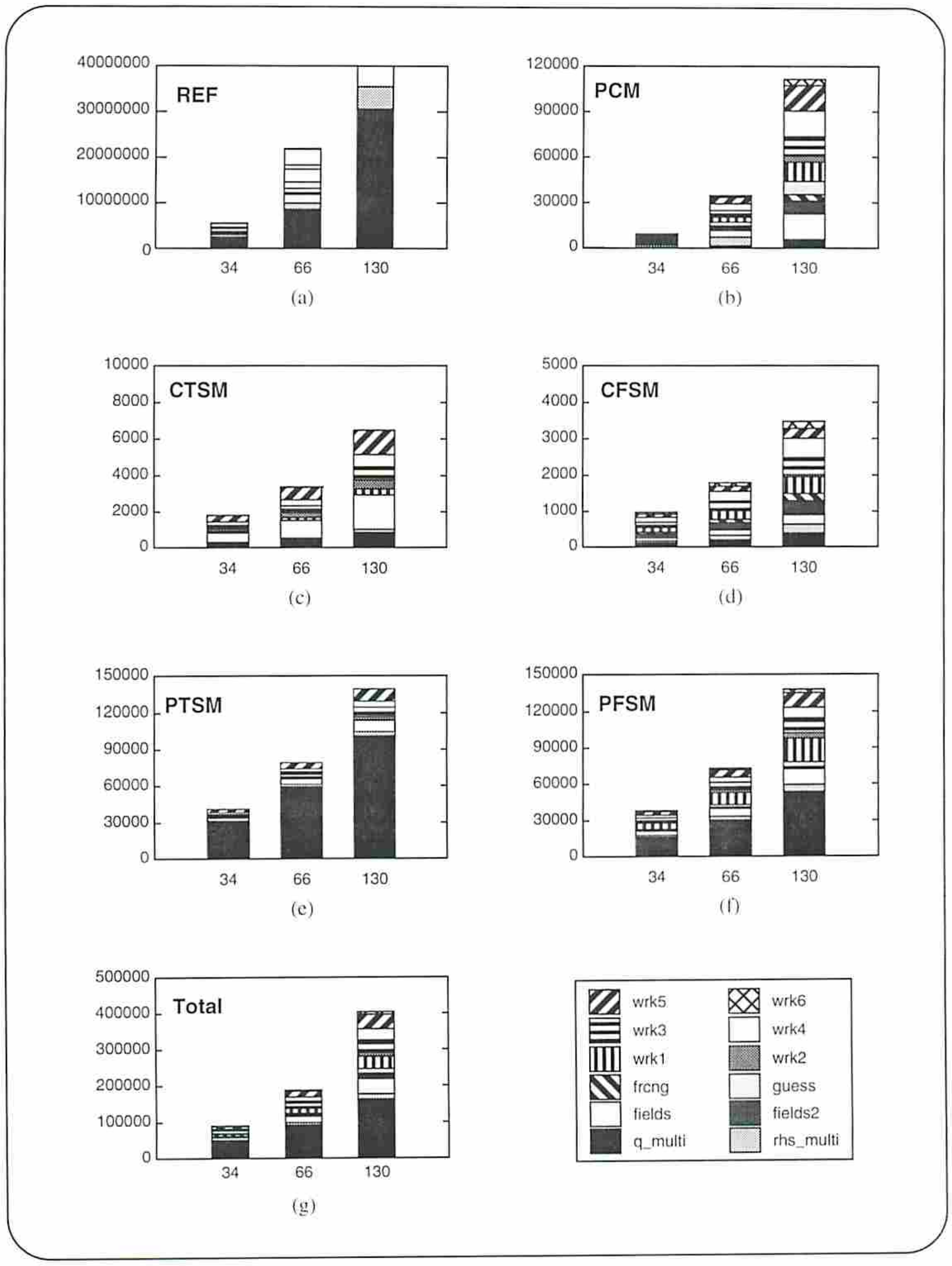


Figure 60. Simulation results in OCEAN.

computed values are smaller than a preset threshold. Table 32 include prediction results of all kinds of metrics of all variables for large data sets with extreme accuracy. The overall miss rate is almost perfectly predicted. Table 33 shows the prediction results for large number of processors. in there, we combined all shared arrays, except for *q_multi* and *rhs_multi*, into a set of rows since their sharing behavior are quite similar to each other. The prediction results in this table are also highly accurate. Consequently, the performance model built on the basis of sharing pattern works accurately even when there are critical difficulties as discussed earlier.

			Prediction-1: 512K			Prediction-2: 1024K		
		Empirical Model	simulation	prediction	error%	simulation	prediction	error%
q_multi	References	$1601.66N^2+66699N-2884388$	122349648	120936976	1.154	454333088	454551456	-0.048
	PCM	$0.332N^2+0.547N+16.560$	22259	22258	0.000	88019	88018	0.000
	TSM	$743.669N+5466.656$	202766	197331	2.680	375802	387709	-3.168
	FSM	$394.174N+2809.42$	107470	104506	2.758	200657	205415	-20371
rhs_multi	References	$259.86N^2+10586.8N-463576$	19787814	19565040	10125	73597080	73631600	-0.046
	PCM	$0.332N^2+0.109N+0.436$	22130	22129	0.000	87778	87777	0.000
	TSM	$23.236N+400.526$	6400	6399	0.016	12234	12344	-0.899
	FSM	$42.816N+351.367$	11676	11398	2.381	22847	22359	2.136
fields	References	$444.00N^2-960.24N+992.28$	29307692	29307710	-0.000	116810424	116810624	-0.000
	PCM	$1.000N^2+0.001N+0.959$	66565	66564	0.000	264197	264195	0.000
	TSM	$88.451N+53.098$	22812	22873	-0.267	45658	45516	0.311
	FSM	$103.547N+19.069$	26712	26735	-0.086	52964	53245	-0.531
fields2	References	$12.001N^2-2.427N+17.222$	798199	798226	-0.003	3169046	3169388	-0.010
	PCM	$0.500N^2+0.000N+0.000$	33282	33282	0.000	132098	132098	0.000
	FSM	$11.924N+3.152$	3095	3079	0.157	6213	6132	1.304
frcng	References	$26.000N^2-0.019N+2.780$	1730666	1730669	-0.000	6869098	6869118	-0.000
	PCM	$0.250N^2+0.000N-0.019$	16641	16640	0.000	66049	66048	0.000
	FSM	$3.984N-4.468$	1019	1022	-0.294	2052	2042	0.487
guess	References	$48.000N^2-96.061N+98.078$	3170400	3170408	0.000	12632160	12632220	0.000
	PCM	$0.500N^2+3.000N-5.989$	34050	34050	0.000	133634	133633	0.000
	FSM	$33.000N$	8514	8514	0.000	16962	16962	0.000

Table 32. Empirical models and prediction results for data set size (OCEAN).

wrk1	References	$210.00N^2+64.069N+36.962$	13994992	13994972	0.000	55514096	55513992	0.000
	PCM	$0.750N^2+0.003N-0.022$	49923	49922	0.000	198147	198145	0.000
	TSM	$3.500N+71.000$	974	974	0.000	1868	1868	0.000
	FSM	$155.090N-85.05$	39861	39927	-0.166	79128	79631	-0.636
wrk2	References	$348.00N^2-1150.1N+1099.1$	22868724	22868836	0.000	91350256	91350920	0.000
	PCM	$0.250N^2+0.250N+0.471$	16706	16705	0.000	66178	66177	0.000
	TSM	$66195.5N+72.75$	5781	5780	0.000	11445	11444	0.000
	FSM	$35.824N-57.136$	9180	9185	-0.054	18400	18356	0.239
wrk3	References	$684.00N^2-2304.0N+2310.6$	44937648	44937652	0.000	179528112	179528112	0.000
	PCM	$0.750N^2+0.003N-0.022$	49923	49922	0.000	1989147	198145	0.000
	TSM	$63.000N+198.000$	16452	16452	0.000	32580	32580	0.000
	FSM	$96.969N+77.062$	25226	25094	0.523	50896	49918	1.922
wrk4	References	$216.00N^2-479.95N+383.62$	14254368	14254325	0.000	56820000	56819804	0.000
	PCM	$1.000N^2+0.000N+0.000$	66564	66564	0.000	264196	264196	0.000
	TSM	$42.000N+65.992$	10902	10901	0.009	21654	21653	0.005
	FSM	$690146N+24.000$	17791	17862	-0.399	35952	35564	1.079
wrk5	References	$839.99N^2-2783.8N+2655.8$	55198176	55197784	0.000	220496352	220494848	0.000
	PCM	$1.000N^2+0.000N+0.000$	66564	66564	0.000	264196	264196	0.000
	TSM	$83.999N+191.999$	21864	21863	0.005	43368	43367	0.002
	FSM	$90.271N-60.587$	24051	24189	-0.574	47269	47634	-0.772
wrk6	References	$24.000N^2+0.016N+427.26$	1597964	1597962	0.000	6341132	6341118	0.000
	PCM	$0.250N^2+0.000N+1.014$	16642	16642	0.000	66050	66050	0.000
	TSM	$0.000N+7090$	7090	7090	0.000	7090	7090	0.000
	FSM	$23.645N-10.790$	6172	6089	1.345	12315	12142	1.405
Total	Reference	$4713.52N^2+69573.5N-3339941$	329996291	328360560	0.496	1277460884	1277713200	-0.020
	PCM	$6.914N^2+3.913N+13.389$	461249	461241	0.002	1828689	1828678	0.001
	TSM	$1069.981N+6616.99$	282645	282664	-0.007	544708	556578	-2.179
	FSM	$1062.879N+3086.639$	280767	277299	1.235	545655	549399	-0.686
	Total Miss	$6.914N^2+2136.77N+9717.01$	1030064	1021204	0.860	2919052	2934655	-0.535
	MissRate(%)	$\frac{6.914N^2+2136.77N+9717.01}{4713.52N^2+69573.5N-3339941}$	**0.3121**	**0.3110**	**0.352**	**0.2285**	**0.2297**	**-0.525**

Table 32. Empirical models and prediction results for data set size (OCEAN).

			Prediction-1: 16 processors			Prediction-2: 64 processors		
		Empirical Model	simulation	prediction	error%	simulation	prediction	error%
q_multi	References	30609328	30609328	30609328	0.000	30609328	30609328	0.000
	PCM	$9.000\sqrt{P} + 5680.999$	5717	5717	0.000	5753	5753	0.000
	TSM	$68471.312\sqrt{P} - 72730.601$	196336	201181	-2.468	475067	475066	0.000
	FSM	$28492.214\sqrt{P} - 12682.999$	98030	101285	-3.321	216340	215254	0.501
rhs_multi	References	4952992	4952992	4952992	0.000	4952992	4952992	0.000
	PCM	5626	5626	5626	0.000	5626	5626	0.000
	TSM	$2234.143\sqrt{P} - 2652.001$	6084	6284	-3.2967	15288	15221	0.437
	FSM	$2865.714\sqrt{P} + 87.996$	11367	11550	-1.617	23075	23013	0.265
All Other Arrays	References	47204070	47204070	47204070	0.000	47204070	47204070	0.000
	PCM	$255.638\sqrt{P} + 105532.976$	106556	106555	0.000	107578	107578	0.000
	TSM	$28838.513\sqrt{P} - 29126.130$	84883	86227	-1.584	201582	201581	0.000
	FSM	$30071.044\sqrt{P} + 21702.458$	142532	141986	0.382	262089	262270	-0.069
Total	References	82766390	82766390	82766390	0.000	82766390	82766390	0.000
	PCM	$264.638\sqrt{P} + 116839.977$	117899	117898	0.000	117898	118957	-0.898
	TSM	$99543.969\sqrt{P} - 99204.734$	287303	293692	-2.224	691937	691868	0.010
	FSM	$61428.977\sqrt{P} + 9107.454$	251929	254821	-1.148	501504	500537	0.193
	Total Miss	$161237.594\sqrt{P} + 26742.695$	657131	666411	-1.412	1311339	1311362	-0.002
	MissRate(%)	$\frac{161237.594\sqrt{P} + 26742.695}{82766390}$	**0.7940**	**0.8052**	**-1.411**	**1.5844**	**1.5844**	**0.000**

Table 33. Empirical models and prediction results for number of processors (OCEAN).

6.5 FFT

Shared variables: FFT is a one-dimensional version of the radix-$\sqrt{N}$ six-step FFT algorithm. Two $\sqrt{N}$-by-$\sqrt{N}$ data arrays of complex numbers are `x` which contains the data points, and `trans` which contains the roots of unity. Two other data structures, umain and umain2, are only accessed by their home processors. To reflect the fact that the arrays are $\sqrt{N}$-by-$\sqrt{N}$, we choose $n=\sqrt{N}$ to represent the data set size. FFT is not iterative and each element of `x` and `trans` (size $O(n^2)$) is accessed $O(1)$ times. So, the total number of references and the cold miss count are $O(n^2)$.

Data-sharing is observed when processors perform the transpose steps (*three* times). Each time, arrays are partitioned so that contiguous rows are statically assigned to P processors. A processor transposes its contiguous submatrices of $\sqrt{N}$/P-by-$\sqrt{N}$/P from every other processor and transposes one submatrix locally. The PTSM count is $O(n^2)$. The number of misses and miss rate are predicted within very small errors.

The effect of the number of processors on TSM is $O((P-1)/P)$ (see Figures 61, 62 and Table 34). Note that the effects of the number of processors on PC are distinct for two arrays. The reason is as follows. In `x`, many processors make their first accesses to the elements that have not been modified (line 2 of Figure 61). On the other hand, The Home of `trans` array elements modifies their elements before remote processors access them (line 2 in Figure 62). This results in no effect of the number of processors on the number of PCM (Table 34).

Size of an element:	8 bytes
Dimension:	2
Size in dimension 1:	$\sqrt{N}$
Size in dimension 2:	$\sqrt{N}$
Partition in dimension 1:	None
Partition in dimension 2:	Block

(a) Static sharing information

1. { E_L: and E_L: } Read;
2. { E_{LV}: and E_{LV}: } Read;
3. { E_L: and E_L: } Write;
4. { E_L: and E_L: } Write;
5. { E_{LV}: and E_{LV}: } Read;

(b) Behavioral sharing description

Figure 61. Static and behavioral sharing description for array *X* in FFT.

Array	Metrics	Scaling Effects	
		Data Set	Processor
x	References	$O(\sqrt{n}^2)$	-
	PCM	$O(\sqrt{n}^2)$	$O((P-1)/P)$
	TSM / FSM	$O(\sqrt{n}^2)$	$O((P-1)/P)$
trans	References	$O(\sqrt{n}^2)$	-
	PCM	$O(\sqrt{n}^2)$	-
	TSM / FSM	$O(\sqrt{n}^2)$	$O((P-1)/P)$

Table 34. Effects of data set size and number of processors.

```
Size of an element:           8 bytes
Dimension:                    2
Size in dimension 1:          √N
Size in dimension 2:          √N
Partition in dimension 1:     None
Partition in dimension 2:     Block
```

(a) Static sharing information

1. { E_L: and E_L: } Read;
2. { E_L: and E_L: } Write;
3. { E_L: and E_L: } Write;
4. { E_L: and E_L: } Write;
5. { E_{LV}: and E_{LV}: } Read;
6. { E_L: and E_L: } Write;

(b) Behavioral sharing description

Figure 62. Static and behavioral sharing description for array `trans` in FFT.

Simulation results: Two arrays yield TSM but no FSM (Figure 63). Interestingly, CTSM are caused only by `trans` and PTSM by x. In FFT program [6, 82], because there are only three places where data-sharing are observed when transposing two large $\sqrt{N}$-by-$\sqrt{N}$ matrices, the cold misses (PCM, CTSM) prevail the true sharing misses (PTSM).

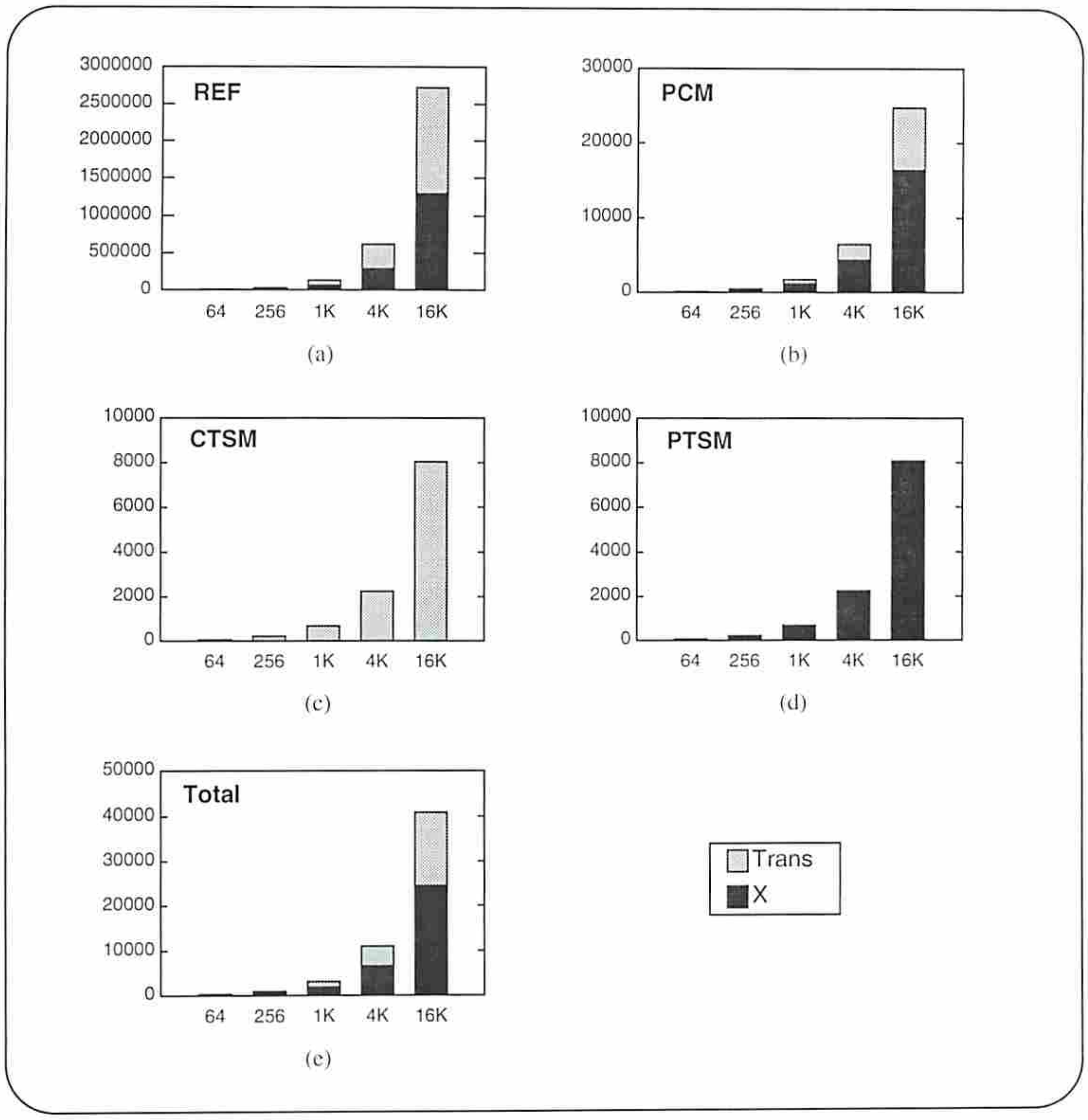


Figure 63. Simulation results in FFT.

Prediction results: The six step FFT algorithm used in this study works on $\sqrt{N}$-by-$\sqrt{N}$ matrices, instead of one-dimensional vectors. Thus, the numbers of references and cache misses seem to depend on $\sqrt{N}$. Table 35 includes the prediction models and their results in terms of $n = \sqrt{N}$ for large data sets and Table 36 is for large number of processors. As expected, the prediction quality is extreme with less than 1% of prediction errors.

			Prediction-1: $n = \sqrt{2^{16}}$			Prediction-2: $n = \sqrt{2^{18}}$		
		Empirical Model	**simulation**	**prediction**	**error%**	**simulation**	**prediction**	**error%**
X	References	$103.10n^2$-4058.9n+74832.6	5734400	5792624	-1.015	25034752	25024166	0.042
	PCM	$0.9373n^2$ + 8.0950n-10.322	63488	63492	-0.006	249852	249855	0.000
	TSM	$0.4373n^2$+7.0953n-10.335	30464	30468	-0.013	118272	118271	0.000
trans	References	$111.10n^2$-4058.9n+74832.6	6258688	6316912	-0.930	27131904	27121318	0.039
	PCM	$0.5040n^2$ +0.9460n-1.6187	33272	33269	0.006	132600	132600	0.000
	TSM	$0.4409n^2$+6.1946n+12.0528	30468	30494	-0.087	118776	118771	0.004
Total	Reference	$214.20n^2$-8117.8n+149665	11993088	12109536	-0.971	52166656	52145484	0.041
	PCM	$1.441n^2$+9.041n-11.941	96760	96761	-0.001	382456	382455	0.000
	TSM	0.8782n2+13.2899n+1.7178	60932	60962	-0.049	237048	237042	0.003
	Total Miss	$2.320n^2$+22.330n-10.223	157692	157723	-0.020	619504	619470	0.001
	MissRate(%)	$\frac{2.320n^2 + 22.330n - 10.223}{214.20n^2 - 8117.8n + 149665}$	**1.3149**	**1.3025**	**0.943**	**1.1875**	**1.1880**	**-0.042**

Table 35. Empirical models and prediction results for data set size (FFT).

			Prediction-1: 32 processors			Prediction-2: 64 processors		
		Empirical Model	**simulation**	**prediction**	**error%**	**simulation**	**prediction**	**error%**
X	References	5734400	5734400	5734400	0.000	5734400	5734400	0.000
	PCM	54636.625 (P-1) / P + 19863.195	72704	72792	-0.121	73815	73646	0.229
	TSM	54636.625 (P-1) / P - 13160.803	39680	39768	-0.222	40837	40622	0.526
trans	References	625868	625868	625868	0.000	625868	625868	0.000
	PCM	33278	33278	33278	0.000	33278	33278	0.000
	TSM	54372.164 (P-1) / P - 12896.730	39680	39776	-0.242	40882	40625	0.629
Total	Reference	6360268	6360268	6360268	0.000	6360268	6360268	0.000
	PCM	54636.625 (P-1) / P + 53141.195	105982	106070	-0.083	107093	106924	0.158
	TSM	109008.789 (P-1) / P - 26057.533	79360	79544	-0.232	81719	81247	0.578
	Total Miss	163645.406 (P-1) / P + 27083.662	185342	185614	-0.147	188812	188171	0.339
	MissRate(%)	$\frac{1636452406 (P-1) / P + 27083.662}{6360268}$	**2.9141**	**2.9183**	**-0.144**	**2.9686**	**2.9585**	**0.340**

Table 36. Empirical models and prediction results for number of processors (FFT).

6.6 BARNES

Shared variables: This application simulates the interaction of a three dimensional system of bodies for a number of time-steps using the Barnes-Hut hierarchical N-body method [7]. A primary one-dimensional shared array `bodytab` of size *N* keeps the particles' information. It is partitioned and allocated to the processors. In each iteration, however, the particles are possibly allocated to another processor according to their position in octree. This also yields data-sharing. Two more arrays form an octree: `ltab` with $O(N)$ elements is an array of bodies at the leaves of the tree and `ctab` with $O(logN)$ elements is an array with the non terminal nodes of the tree.

The number of cold misses is $O(N+logN)$. In each iteration, every element of `bodytab` traverses the octree to visit $O(logN)$ non-terminal cells (elements of `ctab`). Then, an element of `ltab` is accessed. Thus, the number of references to `ctab` and `bodytab` are respectively $O(NlogN)$ and $O(N)$. Total number of references can be expressed as $O(NlogN)+O(N)+O(logN)+O(1)$ and the number of PTSM is estimated as $O(N)+O(logN)+O(1)$.

Static and behavioral sharing information of shared arrays are found in Figures 64 through 66. Note that the *IC* loop for `bodytab` is regarded as an *RC* loop to `ctab`. From them, the effects of data set size and number of processors on performance metrics are summarized in Table 37. The reason for $O((P-1)/P)$ is the random accesses to all array elements during the execution.

Array	Metrics	Scaling Effects	
		Data Set	Processor
bodytab	References	$O(N)$	-
	PCM	$O(N)$	$O((P-1)/P)$
	TSM / FSM	$O(N)$	$O((P-1)/P)$
ctab	References	$O(N)$	-
	PCM	$O(N)$	$O((P-1)/P)$
	TSM / FSM	$O(N)$	$O((P-1)/P)$
ltab	References	$O(N)$	-
	PCM	$O(N)$	$O((P-1)/P)$
	TSM / FSM	$O(N)$	$O((P-1)/P)$

Table 37. Effects of data set size and number of processors.

```
Size of an element:          112 bytes
Dimension:                   1
Size in dimension 1:         N
Partition in dimension 1:    Block
```

(a) Static sharing information

```
1. $E_L$: Read;
2. RC-loop ( $O(1)$ times) BEGIN
3.     $E_L$: Write;
4.     $E_L$: Read;
5.     $E_L$: Write;
6. RC-loop END
7. RC-loop ( $O(1)$ times) BEGIN
8.     $E_{LV-S}$: p, Read;
9.     $E_{LV-S}$: r, Read;
10.RC-loop END
11.$E_L$: Read;
12.$E_L$: Write;
13.RC-loop ( $O(1)$ times) BEGIN
14.    $E_{LV-S}$: p1, Read;
15.    RC-loop ( $O(1)$ times) BEGIN
16.        $E_{LV-S}$: p2, Write;
17.        $E_{LV-S}$: p3, Read;
18.        $E_{LV-S}$: p4, Write;
19.    RC-loop END
20.    RC-loop ( $O(1)$ times) BEGIN
21.        $E_{LV-S}$: p5, Read;
22.        $E_{LV-S}$: p6, Read;
23.    RC-loop END
24.    $E_{LV-S}$: p7, Read;
25.    $E_{LV-S}$: p8, Write;
26.RC-loop END
```

(b) Behavioral sharing description

Figure 64. Static and behavioral sharing description for array *bodytab* in BARNES.

Size of an element:	96 bytes
Dimension:	1
Size in dimension 1:	log(N)
Partition in dimension 1:	Block

(a) Static sharing information

```
1. RC-loop ( $O(1)$ times) BEGIN
2.     RC-loop ( $N$ times) BEGIN
3.         RC-loop ( $O(1)$ times) BEGIN
4.             $E_L$: Read;
5.             $E_L$: Write;
6.         RC-loop END
7.         $E_L$: Write;
8.         RC-loop ( $O(1)$ times) BEGIN
9.             $E_{LV-S}$: q1, Read;
10.        RC-loop END
12.        $E_{LV-S}$: q2, Read;
13.        $E_{LV-S}$: q3, Read;
14.        $E_{LV-S}$: q4, Read;
15.    RC-loop END
16.RC-loop END
```

(b) Behavioral sharing description

Figure 65. Static and behavioral sharing description for array *ctab* in BARNES.

Simulation results: Since there are only a few iterations and small amount of data-sharing on *ctab*, *ltab* and *bodytab*, the cold misses (PCM and CTSM) in BARNES are dominant over the invalidation misses (PTSM and PFSM). So are the true sharing misses (CTSM and PTSM) over false sharing misses (CFSM and PFSM). Especially, *ctab* and *ltab* seldom present false sharing misses (Figure 67 (d), (f)). While *bodytab* produces the most PCM due to the largest size, *ctab* contributes to PTSM, the most. It is because, like *Cells* in MP3D which consists of a few elements and accessed once by every particle, the number of elements of *ctab* is quite small and the octree is traversed once per every particle.

```
Size of an element:          120 bytes
Dimension:                   1
Size in dimension 1:         N
Partition in dimension 1:    Block
```

(a) Static sharing information

```
RC-loop ( $O(1)$ times) BEGIN
    RC-loop ( $O(1)$ times) BEGIN
        $E_L$: Read;
        $E_L$: Write;
    RC-loop END
    $E_L$: Write;
    $E_L$: Write;
    $E_L$: Read;
    $E_{LV-S}$: r1, Read;
    $E_{LV-S}$: r2, Write;
    $E_{LV-S}$: r3, Read;
RC-loop END
```

(b) Behavioral sharing description

Figure 66. Static and behavioral sharing description for array `ltab` in BARNES.

Prediction results: Overall, in Tables 38 and 39, prediction quality in BARNES is not collectively as good as that of other applications. It is because the sharing patterns are quite irregular due to the dynamically changing nature of the particle distribution, unstructured long-distance communication and the fact that different phases of computations in program prefer different data partitioning. By the way, the prediction quality of the overall miss rate is better than that of individual predictions. The reason is that, for the programs whose runtime behavior is decided in a somewhat random manner, the different possible sequences of accesses of distinct processors to the same cache line result in a redistribution of the misses in different categories.

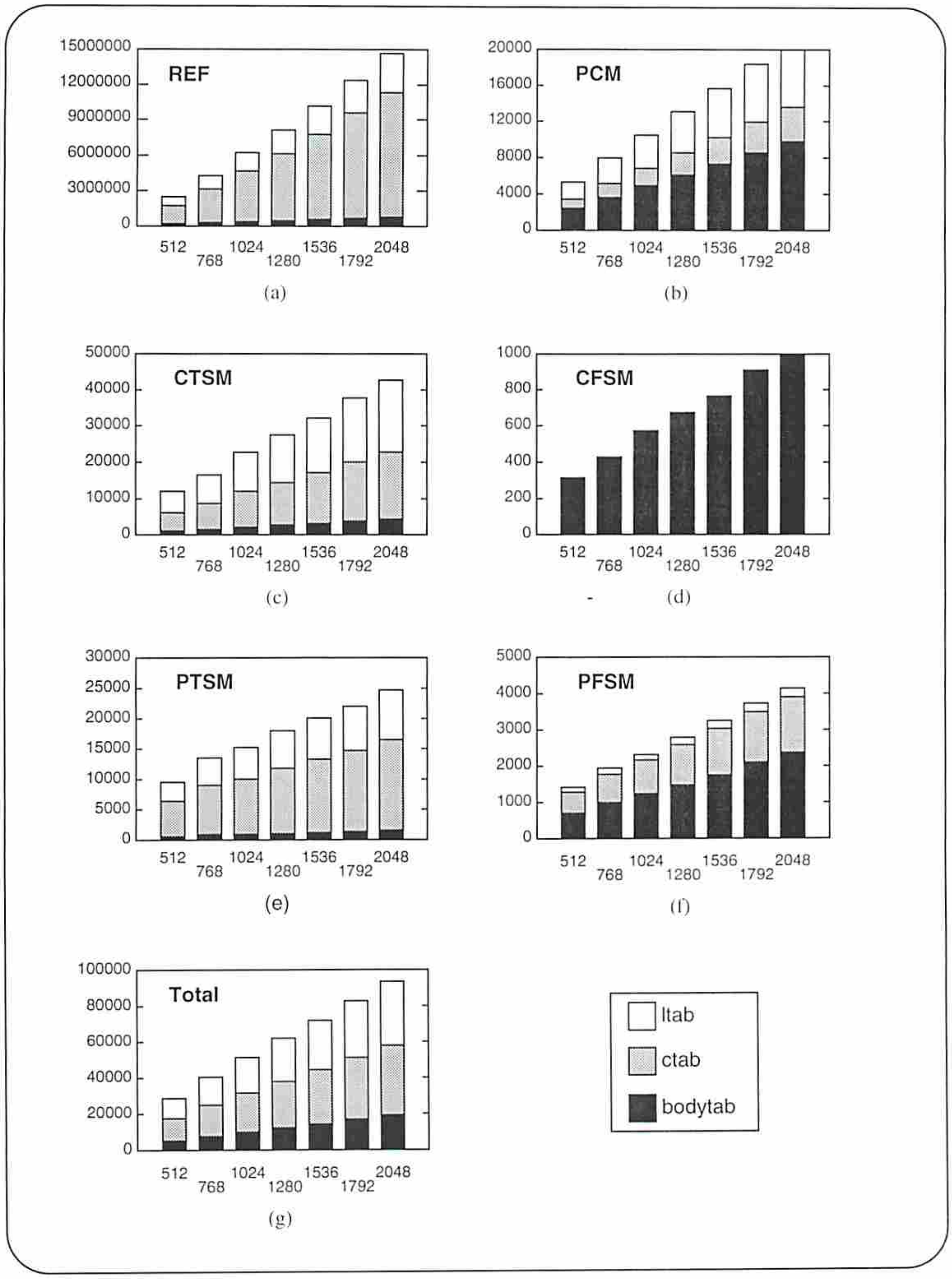


Figure 67. Simulation results in BARNES.

			Prediction-1: 4K			Prediction-2: 8K		
		Empirical Model	simulation	prediction	error%	simulation	prediction	error%
bodytab	References	364.774N+30855.0	1525892	1524970	0.060	3016576	3019086	-0.083
	PCM	364.774N-73.709	19251	19607	-1.852	39438	39289	0.377
	TSM	365.064N+483.034	11025	10828	1.787	21393	21173	1.028
	FSM	1.375N+398.981	5997	6050	-0.884	11670	11687	-0.146
ctab	References	1937.16NlogN-4315971logN+11821739	25855492	26296320	-1.704	62030980	61885876	0.233
	PCM	0.3677NlogN+908.10logN-2290.24	7378	7366	0.160	14690	14696	-0.046
	TSM	2.163NlogN+13366.05logN-24127	55599	56226	-1.128	102165	102026	0.136
	FSM	0.076NlogN+838.346logN-2089.8	2591	2514	2.969	4229	4252	-0.552
ltab	References	1180.61N+541097	6351969	5376861	15.351	9678841	10212625	-5.514
	PCM	3.575N+4.963	14443	14649	-1.426	29373	29293	0.271
	TSM	10.525N+5067.44	46717	48177	-3.125	91429	91288	0.154
	FSM	0.071N+108.698	491	399	18.667	653	689	-5.663
Total	Reference	1937.16NlogN+1545.381N-4315971logN+12393692	33733353	33198151	1.587	74726397	75117587	-0.523
	PCM	0.368NlogN+8.380N+908.10logN-2358.9	41072	41622	-1.339	83501	83278	0.267
	TSM	2.163NlogN+13.051N+13366.05logN-27123.9	112512	115231	-2.417	214987	214487	0.233
	FSM	0.076NlogN+1.446N+838.346logN-17019.699	9079	8963	1.278	16552	16628	0.459
	Total Miss	2.607NlogN+22.877N+15112.5logN-31065	652663	655816	-1.938	315040	314393	0.205
	MissRate(%)	$\frac{2.607N\log N + 22.877N + 15112.5\log N - 31065}{1937.16N\log N + 1545.381N - 431597\log N + 12393692}$	**0.4822**	**0.4995**	**-3.581**	**0.4216**	**0.4185**	**0.725**

Table 38. Empirical models and prediction results for data set size (BARNES).

			Prediction-1: 32 processors			Prediction-2: 64 processors		
		Empirical Model	simulation	prediction	error%	simulation	prediction	error%
bodytab	References	405830	405830	405830	0.000	405830	405830	0.000
	PCM	2330.514 (P-1) / P + 2962.669	5256	5220	0.678	5401	5256	2.670
	TSM	10888.731 (P-1) / P - 5020.632	5749	5527	3.847	5960	5697	4.413
	FSM	4974.394 (P-1) / P - 2039.115	2858	2779	2.735	3014	2857	5.209
ctab	References	4687310	4687310	4687310	0.000	4687310	4687310	0.000
	PCM	-685.457 (P-1) / P + 2999.743	2244	2335	-4.086	2325	2324	0.000
	TSM	134459.781 (P-1) / P - 75390.804	55727	54867	1.543	58162	56986	2.022
	FSM	6522.676 (P-1) / P - 3533.383	2802	2785	0.590	2953	2887	2.235
ltab	References	1517418	1517418	1517418	0.000	1517418	1517418	0.000
	PCM	-4875.343 (P-1) / P + 8071.645	3362	3348	0.396	3319	3272	1.401
	TSM	75196.531 (P-1) / P - 39864.765	34249	32981	3.699	36754	34156	7.069
	FSM	101.390 (P-1) / P + 134.744	177	232	-31.619	276	234	15.081
Total	Reference	6610558	6610558	6610558	0.000	6610558	6610558	0.000
	PCM	-3230.286 (P-1) / P + 14034.057	10862	10903	-0.377	11045	10852	1.747
	TSM	220545.047 (P-1) / P - 120276.203	95725	93375	2.455	100876	96839	4.002
	FSM	11598.460 (P-1) / P - 5437.754	5837	5796	0.702	6243	5978	4.245
	Total Miss	228913.219 (P-1) / P - 111679.906	112424	110074	2.090	118164	113669	3.804
	MissRate(%)	$\frac{229813.219\ (P-1) / P - 111679.906}{6610558}$	**1.7007**	**1.6651**	**2.093**	**1.7875**	**1.7195**	**3.804**

Table 39. Empirical models and prediction results for number of processors (BARNES).

6.7 RADIX

Shared variables: RADIX performs integer radix sort. In each iteration, an *r*-bit digit field of the keys are sorted, *r* being the radix. Two principal array, `key[0]` and `key[1]`, are one dimensional. Most data accesses are made to `key[0]` and `key[1]`. Hence, the number of cold misses is $O(N)$. The number of iterations depends on the largest possible key values and on the radix, both of which are run-time parameters. Thus the number of iterations is $O(1)$. In each iteration, every $O(N)$ element of one of the two arrays `key[0]` or `key[1]` is accessed to compute a local histogram and partially sorted results are stored into the other array. The numbers of references and of PTSM are both $O(N)$.

```
Size of an element:          4 bytes
Dimension:                   1
Size in dimension 1:         N
Partition in dimension 1:    Block
```

(a) Static sharing information

```
1. RC-loop ( $O(1)$ times) BEGIN
2.     $E_L$: Write;
3.     $E_L$: Read;
4.     $E_{LV-S}$: a, Write;
5. RC-loop END
```

(b) Behavioral sharing description

Figure 68. Static and behavioral sharing description for array `key[0]` in RADIX.

At the beginning stage of each iteration, the data to be sorted are stored in one of two arrays and block-partitioned manner to the processors. Using the allocated data, processors build local histograms which will be accumulated to form a global histogram. Then, each processor uses the global histogram to permute its keys into another array for the next iteration. Data-sharing is observed in subsequent iterations when the array which keeps partially sorted data is partitioned and assigned, again.

```
Size of an element:          4 bytes
Dimension:                   1
Size in dimension 1:         N
Partition in dimension 1:    Block
```

(a) Static sharing information

```
1. RC-loop ( $O(1)$ times) BEGIN
2.     $E_{LV-S}$: b, Write;
3.     $E_L$: Read;
4. RC-loop END
```

(b) Behavioral sharing description

Figure 69. Static and behavioral sharing description for array *key[1]* in RADIX.

As shown in the static and behavioral information of both arrays in Figures 68 and 69, the first accesses to them are write operations. This results in no effects of the number of processors on the number of PCM (Table 40). Due to the random selection of memory locations when the processors are building the global histogram, the effects of the number of processors on TSM is denoted by $O((p-1)/(p))$.

Array	Metrics	Scaling Effects	
		Data Set	Processor
key[0]	References	$O(N)$	-
	PCM	$O(N)$	-
	TSM / FSM	$O(N)$	$O((P-1)/P)$
key[1]	References	$O(N)$	-
	PCM	$O(N)$	-
	TSM / FSM	$O(N)$	$O((P-1)/P)$

Table 40. Effects of data set size and number of processors.

Two more shared array of `densities` and `ranks` show invariant sharing patterns. That is, as the data set size varies, the numbers of their cache misses remain the same. As shown in Table 41, they do not contribute much to cache misses. So, when we consider the effects of the number of processors, they are not counted.

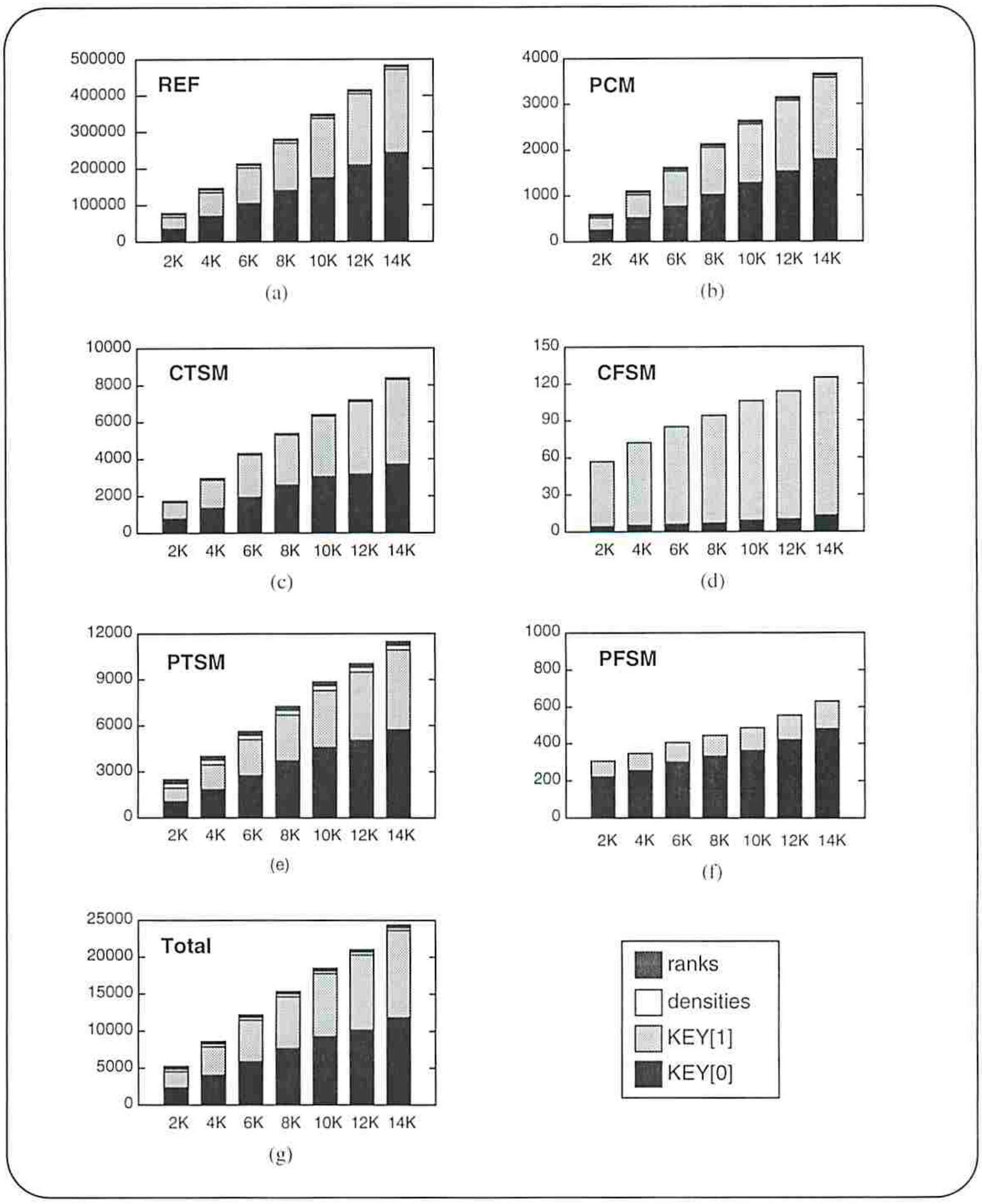


Figure 70. Effect of the data set size on the number of misses in RADIX.

Simulation results: In RADIX, true sharing misses are dominant (Figure 70 (c), (e)) over only a few false sharing misses. This is because of all-to-all communication in permutation step. That is, before each step, all array elements in one of `key[0]` or `key[1]` were updated, and all read accesses to them by other processors result in either CTSM or PTSM. Additionally, the constant number of negligible misses are presented by `densities` and `ranks`.

			Prediction-1: 512K			Prediction-2: 1024K		
		Empirical Model	simulation	prediction	error%	simulation	prediction	error%
KEY[0]	References	17.000N+0.004	8912896	8912895	0.000	17825792	17825790	0.000
	PCM	0.125N+1.000	65537	65537	0.000	131073	131073	0.000
	TSM	0.554N+1379.67	291118	292255	-0.391	583680	583132	0.094
	FSM	0.003N+444.242	2235	1815	18.792	2974	3186	-7.128
KEY[1]	References	16.000N-0.034	8388608	8388609	0.000	16777216	16777218	0.000
	PCM	0.125N+3.013	65539	65538	0.000	131075	131074	0.000
	TSM	0.626N+742.67	329710	329262	0.136	657556	657893	-0.051
	FSM	0.002N+190.083	1691	1682	0.532	3173	3177	-0.126
densities	References	64646464	64646464	64646464	0.000	64646464	64646464	0.000
	PCM	45	45	45	0.000	45	45	0.000
	TSM	336	336	336	0.000	336	336	0.000
ranks	References	4096	4096	4096	0.000	4096	4096	0.000
	PCM	36	36	36	0.000	36	36	0.000
	TSM	288	288	288	0.000	288	288	0.000
Total	Reference	33.000N+10560.0	17301504	17301504	0.000	34603008	34603008	0.000
	PCM	0.250N+85.013	131076	131075	0.001	262148	262147	0.000
	TSM	1.181N+2746.3	620828	621517	-0.111	1241236	1240914	0.026
	FSM	0.006N+634.326	3926	3497	10.927	6147	6363	-3.514
	Total Miss	1.437N+3465.7	755830	756089	-0.089	1509531	1509425	0.007
	MissRate(%)	$\frac{1.437N + 2913.7}{33.000N + 10560.0}$	**4.3686**	**4.3701**	**-0.034**	**4.3624**	**4.3621**	**0.006**

Table 41. Empirical models and prediction results for data set size (RADIX).

Prediction results: Usually, according to the experience in predicting the number of misses, the false sharing misses are not as easy or precise as true sharing misses to predict. The variable of `key[0]` in RADIX exhibits a huge prediction error in PFSM. But the absolute amount of error is too small to influence overall prediction quality. The shared data structures of `densities` and `ranks` have performance parameters which are independent of the data set size. On the whole we have extremely good prediction results for RADIX, too.

The number of FSM is much more difficult to predict than the number of true sharing misses. The number of false sharing misses to `key`'s is poorly predicted. However, the number of these misses is very small and does not affect overall prediction quality.

			Prediction-1: 32 processors			Prediction-2: 64 processors		
		Empirical Model	simulation	prediction	error%	simulation	prediction	error%
KEY[0]	References	2228240	2228240	2228240	0.000	2228240	2228240	0.000
	PCM	16385	16385	16385	0.000	16385	16385	0.000
	TSM	139000.375 (P-1) / P - 34093.289	101839	100563	1.252	110313	102735	6.870
	FSM	18156.074 (P-1) / P - 11356.676	6903	6232	9.720	8124	6515	19.806
KEY[1]	References	2097137	2097137	2097137	0.000	2097137	2097137	0.000
	PCM	16384	16384	16384	0.000	16384	16384	0.000
	TSM	141441.875 (P-1) / P - 34635.703	104618	102386	2.1333	112515	104596	7.038
	FSM	2348.941 (P-1) / P - 1419.567	949	855	9.803	967	892	7.756
Total	Reference	4325377	4325377	4325377	0.000	4325377	4325377	0.000
	PCM	32769	32769	32769	0.000	32769	32769	0.000
	TSM	280442.250 (P-1) / P -68728.992	206457	202949	1.699	222828	207331	6.955
	FSM	20505.016 (P-1) / P -12776.243	7852	7087	9.743	9091	7407	18.524
	Total Miss	300947.250 (P-1) / P - 81505.234	247078	242805	1.729	264688	247507	6.491
	MissRate(%)	$\frac{300947.250\ (P-1)\ /\ P-\ 81505.234}{4325377}$	**5.7123**	**5.6135**	**1.730**	**6.1194**	**5.7222**	**6.491**

Table 42. Empirical models and prediction results for number of processors (RADIX).

6.8 Summary

Table 43 summarizes the prediction results for data set size of all applications. In addition, Table 44 is for the number of processors. Finally, Table 45 shows the prediction errors when all variables are treated together as in Section 3 and when they are treated individually as in this section. In most cases, we have achieved marked improvements in prediction errors.

Benchmark	Empirical Model for Miss Rate	Prediction - 1			Prediction - 2		
		simulation	prediction	error%	simulation	prediction	error%
LU	$\frac{1.250N^2 + 12.967N - 61.928}{1.999N^3 + 7.327N^2 - 23.5566N + 803.06}$	0.2220	0.2220	0.000	0.1237	0.1237	0.000
MP3D	$\frac{21.453N + 4111.13}{348.843N + 41419.6}$	6.1494	6.151	-0.035	6.1332	6.1507	-0.285
WATER	$\frac{4.490N^2 + 151.164N - 221.48}{540.016N^2 - 2938.3N + 644.47}$	0.8964	0.8953	0.122	0.8646	0.8636	0.115
OCEAN	$\frac{6.914N^2 + 2136.77N + 9717.01}{4713.52N^2 + 69573.5N - 3339941}$	0.3121	0.3110	0.352	0.2285	0.2297	-0.525
FFT	$\frac{2.320n^2 + 22.330n - 10.223}{214.20n^2 - 8117.8n + 149665}$	1.3149	1.3025	0.943	1.1875	1.1880	-0.042
BARNES	$\frac{2.607N\log N + 22.877N + 15112.5\log N - 31065}{1937.16N\log N + 1545.381N - 431597\log N + 12393692}$	0.4822	0.4995	-3.581	0.4216	0.4185	0.725
RADIX	$\frac{1.437N + 2913.7}{33.000N + 10560.0}$	4.3686	4.3701	-0.034	4.3624	4.3621	0.006

Table 43. Empirical models for cache miss rates and prediction results for data set size.

Benchmark	Empirical Model for Miss Rate	Prediction - 1			Prediction - 2		
		simulation	prediction	error%	simulation	prediction	error%
LU	$\frac{8288.700P - 6766.746}{34014695}$	0.8432	0.8324	1.280	1.6121	1.6121	0.000
MP3D	$\frac{823901.438 \times (P-1)/P + 797203.062}{11242601}$	6.9761	7.0180	-0.601	13.0672	12.8420	1.723
WATER	$\frac{4102.024P + 300212.562}{36138850}$	1.1831	1.1939	-0.913	1.5572	1.5572	0.000
OCEAN	$\frac{161237.594\sqrt{P} + 26742.695}{82766390}$	0.7940	0.8052	-1.411	1.5844	1.5844	0.000
FFT	$\frac{1636452406 (P-1) / P + 27083.662}{6360268}$	2.9141	2.9183	-0.144	2.9686	2.9585	0.340
BARNES	$\frac{229813.219 (P-1) / P - 111679.906}{6610558}$	1.7007	1.6651	2.093	1.7875	1.7195	3.804
RADIX	$\frac{300947.250 (P-1) / P - 81505.234}{4325377}$	5.7123	5.6135	1.730	6.1194	5.7222	6.491

Table 44. Empirical models for cache miss rates and prediction results for number of processors.

Benchmarks		LU	MP3D	WATER	OCEAN	FFT	BARNES	RADIX
Typical Variable Size		$O(N^2)$	$O(N)$	$O(N)$	$O(N^2)$	$O(n^2)$	$O(N)$&$O(\log N)$	$O(N)$
Prediction Error (%)	All variables	-0.080	-0.280	0.115	-3.938	-3.023	0.071	2.924
	Individual	0.000	-0.285	0.115	-0.525	-0.042	0.725	0.006

Table 45. Comparison of prediction errors for data set szie.

Chapter 7

CONCLUSION

An empirical modeling technique for primary performance metrics such as the numbers of references and of various types of cache misses was introduced in this thesis. To overcome the huge resource requirements and excessively long execution time of software simulations, many researchers proposed diverse analytical modeling techniques. Traditional analytical models are referred to as structural models which are abstracted from the expected behavior of the workload and formalized with a set of statistically defined parameters. It is in general difficult to estimate the effects of variables such as the data set size or the number of processors. Thus a model is only valid for a particular application with a given data set size and a given number of processors.

Owing to the wide availability of general-purpose hardware and existing mathematical curve fitting technique, empirical models can be developed. In our empirical performance model, a few samples of performance metrics are collected by simulating a few small-size problems running on small systems. On the other side, the magnitude orders of selected performance metrics are found from static workload analysis. Then, statistical extrapolation techniques produce an analytical model which can be applied to estimate the values of the performance metrics for realistically large data set size and number of processors.

With the simulation environments and mathematical technique, the only remaining problem is to obtain the magnitude orders of the performance metrics in big-oh notation along which the sample points tend to be located. Since our target is the cache miss rate model whose primary source is the data-sharing, the research was started with the data-sharing analysis.

The data-sharing analysis can be carried on using the traces collected from the actual execution of parallel programs, or on the parallel application codes. In general, the traces are valid only for the hardware and software configurations from which they are collected. For this reason, the level of accuracy in data-sharing analysis is higher when the application codes are used.

However, in directly using the program codes for data-sharing, there is one manifest obstacle. For example, consider a program segment given in Figure 71. Since the index of `Particles` array in line 8 depends on the run time results of preceding computations, we cannot statically identify which memory location is accessed by a processor. Therefore, it is practically impossible to define the sharing information from the source code of general scientific SPMD model applications.

```
1. Part = &Particles[pn];
2. u = Part->u;
3. v = Part->v;
4. Part->x += u;
5. Part->y += v;
6. Acell = &Cells[CELL(Part-x][CELL[Part->y];
7. space_val = ACell->space;
8. Collide( &Particles[space_val], Part );
```

Figure 71. Shared array and index variables.

As an alternative, the development of a formal descriptor called an access pattern descriptor (APD) was proposed. The program statements in an application will be substituted by appropriate data-sharing representation of APDS (APD statement). Then, algorithmic procedures made direct use of the APDS to produce the models for the miss rates. The validity of this methodology is supported by the fact that the researchers involved in HPF and ALPSTONE projects take the similar direction as ours.

The data objects are normally represented by array data structures and the computations on them are performed within iterative loops. The information regarding the memory locations of the data objects can be found in the program. The index expressions of array variable bear the information regarding possible data-sharing in the program. Therefore, the data-sharing analysis started with the array index expression analysis.

Our research goal is to establish a method to translate the source code into the APDS by means of index expression analysis. To this goal, we tried to characterize the data-sharing patterns for various types of array index expressions that are found in general scientific parallel applications. First of all, we investigated which types of factors are associated with data-sharing. Once we understood the sharing factors and their ways of making unique contributions on data-sharing, the components used in array index expressions and the composite index expressions composed of the index expression components were enumerated. At this moment, the effects of the data set size and the number of processors on the amount of data-sharing were included. The models for the number of data references and the number of cache misses were built based on the knowledge we obtained by applying the algorithms to count the number of data references and the number of misses.

Generally, overall quality of the modeling technique is determined by comparing the resulting models against experimental results. In addition, the prediction accuracy for very large data sets and processor numbers is also important in judging the usefulness of analytical models. Once the model is proved to be accurate, the whole technique including the definition of sharing factors, grouping their values into categories, capturing the values of sharing factors from applications and formularizing the factors into a numeric expression is appraised to be accurate, as well. In this regard, the extreme accuracy of our empirical models for miss rates based on sharing analysis evidences the high quality of modeling technique introduced in this thesis.

Followings are the contributions of the empirical performance modeling technique introduced in this thesis.

In structural modeling, the parameters are statistically evaluated from execution histories or traces. The traces are usually so large that dealing with the traces is very time-consuming. Additionally, since a structural model is valid only for the configurations in which it is built, the parameters must be reevaluated whenever different data set size or number of processors is considered. By contrast, in the empirical technique, performance models are obtained very quickly by robust estimation method. Furthermore, having the data set size and the number of processors as variables, an empirical model does not require repetitive parameter computation for various problem and system sizes.

The usefulness of our empirical model is substantiated by its correct representation and accurate prediction of expected performance for varying data set size and number of processors. Therefore, researchers can quickly foresee the potential performance of the target system running realistically heavy workloads without suffering from large resource requirements and long simulation times.

Simple syntax of APDS and easy transformation of SPMD applications into APDS allow easy representation of rich sharing characteristics. Fast interpretation procedure of APDS codes reduces the estimation time for the performance model parameters. Finally, we can locate the places in APDS descriptions where the performance bottlenecks are possible due to the huge amount of data-sharing. This is possible because the data access information in the given SPMD applications are maintained in APDS.

Chapter 8

RELATED WORK

Scientific applications simulate physical phenomena by discretizing continuous problems in both space and time into numerically approximate algorithms. In order to simulate new aspects of physical phenomena or to improve the simulation accuracy, multiprocessor systems are used to run much larger problems than can be run on a uniprocessor. The issues arising when scaling scientific problems are addressed in [69]. Besides the data set size, other parameters, such as the time step or the total simulated time, which affect simulation errors should be scaled as well. The effects of these time parameters on the miss rates are easy to evaluate. Thus, the work in this proposal has been started with the study of the effects of the data set size on the miss rates.

Cache misses play a key role in the performance of shared-memory systems. They are customarily classified into cold, coherence and replacement misses [27]. Cold misses are due to the first access to a memory block by a processor. Coherence misses are the misses following invalidations in a system with infinite cache sizes. Replacement misses are all the other misses caused by the finiteness of the cache size [82]. Thus the numbers of cold and coherence misses are independent of the cache size and can be counted in a system with infinite caches. The general effects of cache sizes on replacement misses have been shown in [64]. Each application has a hierarchy of a few, well defined per-processor working sets. For a given problem size, as the cache size is increased, the (replacement) miss rate drops abruptly every time a new working set fits in the cache. The conclusion is that most of the miss rate reduction is obtained for very small caches. The classification of cold misses and coherence misses can be further refined [26, 29, 76]. Some coherence misses are essential in the sense that they communicate a new word value (true sharing misses) [26]. Other misses, often called false sharing misses, are useless misses.

Because the performance of shared-memory multiprocessors is known to depend strongly on the way concurrent processes communicate by writing and reading the shared data, the data-sharing characteristics of programs have been the subject of many previous research papers. Data-sharing patterns result from the algorithmic movements of data among the processors and they represent the necessary data communications among the processors for the correct execution of an application. A good understanding of data-sharing patterns helps in designing better memory hierarchies and cache coherence protocols. Thus, several models were developed in the past to estimate the number of misses in a shared-memory system based on the structural characteristics of applications with respect to data-sharing.

In [23], Dubois and Briggs defined a synthetic program model with data-sharing. Different access patterns are defined for S-blocks; the shared writable blocks, and P-blocks, the other shared blocks plus the private blocks. The effects of cache coherency on the misses and on the traffic between the caches are then evaluated analytically. A drawback of this model is that the locality of accesses to shared data was not modeled. In [24] the model was extended to reflect the fact that shared writable data are typically accessed in critical or semi-critical sections.

Eggers and Katz [27] characterized the sharing pattern by the write run obtained from an interleaved reference trace. The average write run is an indicator of invalidation misses and traffic in snooping protocols. It is a very parsimonious model, but the information contained in the write run is not sufficient to characterize all possible coherence transactions in complex protocols. Another drawback is that the write run relates to the composite behavior of all shared data in a whole application. In practice an application manipulates several data structures with distinct sharing patterns and, as the data set size changes, it is possible that the coherence overhead will be dominated by different shared data structures as shown in current research results.

Recognizing the need to characterize the behavior of each separate shared data structure, Gupta and Weber [36] classified shared objects as read-only, migratory, mostly read, or frequently read/written data. This classification is still vague and does not lead to any quantification of the coherence overheads. Bennett *et al.* [9] expanded this classification of shared objects by defining additional classes of shared-data access patterns.

Tsai and Agarwal [77] constructed an analytical expression for cache miss rates by first identifying the types of shared data blocks in the parallel program and by capturing their individual caching behavior through a few parameters such as problem size, number of processors and cache line size. The exact number of accesses to the shared data in the program must be known in order to apply the model, which makes the analysis of complex or irregular applications difficult.

In [72], a set of analytical models for predicting the performance of parallel applications were presented. The authors adopted the traditional shared data classification where the access types are read or write and the number of processors are defined by singular or multiple. The parameters used in forming the analytical models were obtained by investigating the traces of previous execution of an application even though the authors claimed that compile-time analysis is possible. These parameters were used to express the number of system events for each type of shared data and the correctness of their models depend on the duration of interval during which the performance parameters were collected. Their models are applicable only when the run time parameters are exactly the same as the ones obtained when collecting traces.

Brorsson and Stenstrom [13, 14, 15] presented simple classification methods of shared data blocks based on the accesses made to the blocks within a finite time slot. They take the access mode and degree of sharing into account. These classification techniques were finally used in a performance analysis tool called SM-prof [13] which, as a compliment to other proposed tools [4, 33, 51, 52, 83], visualizes the shared-data access pattern in a diagram with links to the source code lines of parallel applications causing performance degrading accesses.

As shown above, data-sharing behavior has been characterized in many different ways. While classifications deal with the sharing behavior of data in the whole execution or in finite time slots, they all combine the data access types and the number of processors that make the accesses. In our context, the structural models built by some of the above authors are problematic. First, they are hard to fit to a particular application. Second, the model parameters depend on block size, data set size and number of processors; thus their predictive value is limited. Third, they are not refined enough to model all classes of misses.

In our research, the refinements of shared data classification were made by, first, managing the individual shared data structures to reflect their unique sharing behaviors. Second, to provide more specific and useful sharing information, the notations for the number of processors are more finely defined to incorporate not only the number of processors but also the logical relationships between accessed data and processors. Finally, the data access patterns are created into APDS whenever distinct sharing behaviors are observed in the execution time phases of an application. The descriptive power of APD is such that, based on the sharing description, we can express the value of the diverse performance metrics for each shared data structure in terms of a parametric polynomial functions of the data set size. The accuracy of our method is verified by the extreme prediction quality.

Because the ultimate goal in a performance study is the execution time, our goal of research is the development of an automated prediction tool for the execution time. Fundamentally, many efforts have been made to achieve the performance prediction through compile-time analysis [32, 38, 54, 59]. Even though the compile-time prediction technique may not fall in the scope of our methodology, relevant literature could offer helpful hints in designing and developing the description language.

Driscoll and Daasch [22] focused on the execution time analysis of the traditional Amdahl's Law [3] and Gustafson's Law [39]. Like other researchers [41, 66], authors in [22] generalized Amdahl's Law and Gustafson's addition. The sequential and parallel portions of applications are considered separately and thus the execution time model consists of two empirical models built based on the experimental results and curve fitting methods. In finding the best fit curves, they merely assumed the execution time to be increased or decreased linearly, which may not be true in most applications as seen in our results in previous sections.

A simple performance prediction toolset has been introduced [80] whose inputs are the description of a parallel program execution in terms of a directed task graph, the specification of the parallel system, and a correspondence between processors and each task graph node is mapped. This model did not include the amount of computations so that the execution time prediction model is too simple to consider all associated factors impacting the execution time.

There is an on-going project, ALPSTONE [16, 45], which is composed of another three sub-projects. The first one is for the development of an algorithm classification scheme called BACS [18] which is actually the vocabulary and methodology for describing parallel algorithms. Another project is for developing the parallel programming language suite called ALWAN [17] and BALI [46] which are the languages used for writing algorithmic skeletons. The last one for developing the ALWAN compiler called TIANA.

In addition to ALPSTONE, more performance prediction systems called PPPE [30] and PAMELA [31] are under development. The core of the work in all these projects is the design and development of the programming languages. Since not much has been published even though the systems have been developed for five or more years in large research groups, we cannot make decisive comments, at this point. Many other literatures suggested the issues to be considered in designing the sharing pattern descriptors [5, 8, 35, 57]. Anyway, continuous efforts will be made to understand the frameworks being carried on in [30] and [31]. In the following, we briefly review the work in APLSTONE.

The researchers in each sub-project of ALPSTONE are currently working on their own benchmarks whose execution time modeling and prediction is done in three steps. At first, a parallel program is modeled by abstract notations defined in BACS including the network topologies. Then, the execution time of each line written in BACS is estimated. If the estimated execution time of the given benchmark is satisfactory, the programmers start programming using ALWAN. Finally the compiled ALWAN program is run on the target machine and the performance is measured to verify the accuracy of the predicted execution time.

The notations and the syntax of BACS is too specific for general users to use or even to understand. Additionally, even though the architecture parameters are independent of the application programs, their inclusion in BACS and ALWAN programs will increase the complexity of the programs and will decrease the freedom of the programmers. Especially, the execution times of all possible combinations of network topologies, data placements, data communications among processors and every kind of operations have to be collected in advance by running on a set of existing physical machines such as SP2. These times are then stored in the database. This type of database is defined for each of their own benchmarks.

Therefore, besides the fact that the prediction results have not been posted since the projects are still under development, the amount of the work in collecting the performance data of every combination of BACS statements is prohibitively large. Furthermore, the application area is limited and is not flexible since the database has been built for the finite set of existing network topologies and physical systems.The syntactic and semantic definitions of BACS will be troublesome for the general users to learn.

Compared to BACS or ALWAN, APDS is a lot easier to understand and to use since the syntactic and semantic definitions are much simpler than that of not only BACS and ALWAN but also the C language. Finally, instead of building a huge database that maintains the execution time model of all possible combinations of every programming object, we collected the simulation results by running problems with small data sets. This requires much less time and efforts. Furthermore, many possible hardware configurations of system under study can be tried for better prototyping of target systems.

Understanding the operational procedure in ALPSTONE project, one may first ask why they use the formal description of underlying algorithm described in BALI instead of directly using the source code written in high level languages such as the C or FORTRAN. In fact, the source of scientific applications are too complicated to statically analyze. For example, data partitioning portion of a parallel program includes some numeric computations using the variables whose values are known only at run time. Furthermore, the sequence of program statements executing very complex computations are almost impossible to be understood by compilers.

Many researchers formed a forum to design a syntax and associated language definitions for High Performance Fortran (HPF) [42]. Since they know that today's compilers cannot capture all useful information to improve the computational efficiency of a program running on parallel machines, they decided to develop additive directives. The primary goals in HPF development are to support data parallel programming and to achieve high performance on parallel computers. To these goals, the directives provide some fundamental factors affecting the performance of a parallel program such as the degree of available parallelism, exploitation of data locality and choice of appropriate task granularity. HPF supplies mechanisms for the programmer to guide the compiler with respect to these factors. HPF directives appear as structured comments that suggest implementation strategies or assert facts about a program to the compiler. When properly used, they affect only the efficiency of the computation performed, but do not change the value computed by the program.

In addition, HPF also features new library routines. The HPF library of computational functions defines a standard interface to routines that have proven valuable for high performance computing. These additional functions include those for mapping inquiry, bit manipulation, array reduction, array combining scatter, prefix and suffix, and array data sorting.

The Stanford University Intermediate Format (SUIF) compiler system [40, 73] is for researching compiler techniques for high performance architectures. One of the goals of SUIF is to automatically translate sequential dense matrix computations into efficient parallel code for parallel machines. Target parallel machines include distributed address space machines as well as shared address space machines. The inputs to SUIF compiler are sequential FORTRAN-77 or C programs. The source programs are first translated into the SUIF compiler's intermediate representation. After passing through the optimization and parallelization, the resulting parallelized SUIF program is then converted to C and is compiled by the native compiler on the target machine. The C program contain subroutine calls to invoke a portable run time library which is linked in by the native compiler.

The key in optimization and parallelization in SUIF is the locality analysis. Given a matrix computation application, the program must make effective use of the computer's memory hierarchy as well as its ability to perform computations in parallel. As a result, reducing interprocessor communication by increasing the locality of data references is important. In this regard, the SUIF restructures the array so that the major regions of data accessed by a processor are contiguous in memory.

The SUIF makes the analysis by investigating the index expressions of array elements in locality analysis phase. The researchers participating in the SUIF project limit themselves to the dense matrix computation applications in which the index expressions are considerably simpler than other general scientific applications such as those in SPLASH [67] and SPLASH-2 [82] benchmark suites.

Appendix A

PREDICTION RESULTS WITHOUT DATA-SHARING ANALYSIS

In this appendix, brief descriptions of benchmark applications and their algorithmic behavior are provided. The tables include emprical models and prediction results for all kinds of performance metrics of every shared data structure in each application.

A.1 LU

LU		Prediction-1: N = 288			Prediction-2: N = 512		
Total	**Empirical Model**	**simulation**	**prediction**	**error%**	**simulation**	**prediction**	**error%**
Reference	$1.994\ N^3 + 8.764\ N^2 - 178.70\ N + 3849.08$	48358061	48321296	0.076	270273517	269893536	0.140
CM	$1.250\ N^2 + 7.030\ N - 36.233$	105669	105662	0.005	331243	331225	0.005
PTSM	$5.396\ N - 14.143$	1532	1539	-0.457	2736	2748	-0.439
PFSM	$0.562\ N - 12.932$	148	148	0.000	276	274	0.725
Essen. Miss	$1.250\ N^2 + 12.426\ N - 50.376$	107201	107201	0.000	333979	333973	0.001
Total Miss	$1.250\ N^2 + 12.988\ N - 63.308$	107349	107349	0.000	334255	334247	0.002
MissRate(%)	$\frac{1.250N^2 + 12.426N - 63.308}{1.994N^3 + 8.764N^2 - 178.70N + 3849.08}$	**0.2220**	**0.2220**	**0.000**	**0.1237**	**0.1238**	**-0.080**

Table 46. Empirical models and prediction results for LU.

LU performs the LU-decomposition of a dense square matrix. The problem size, *N*, is number of rows of the matrix as well as the number of iterations. The data structures are two large matrices, `A` and `L`, of size *N*-by-*N* and one vector, `thisPivot`, of size *N*. Thus, we expect $O(N^2)$ cold misses. In i^{th} iteration (*i*=0 to *N*-1), the rightmost *N*-*i* columns of `A` ($O(N^2)$) are accessed once by the processors to which they are allocated, the ith column of `L` ($O(N)$) N-i times ($O(N^2)$), and the ith element of `thisPivot` ($O(1)$) N-i times ($O(N)$). Thus, the number of shared data accesses is $O(N^3)$. `A` and `L` only presents only cold misses and $O(N)$ sharing misses are on `thisPivot`.

As shown in Table 46, the prediction errors are extremely small for large problem sizes. This is expected because the patterns of accesses to shared data and of data communications among processors are very regular and predictable in LU.

A.2 MP3D

MP3D repeatedly calculates the positions and velocities of particles during a preset number of time steps. The data set size, N, is the number of molecules in the one dimensional array `Particles`. The size of another array, `Ares`, is also proportional to N and `Cells` is a three-dimensional array of size independent of N. Therefore, we estimate the cold misses in MP3D as $O(N)$.

The number of iterations is ($O(1)$). In each iteration, every one of the $O(N)$ `Particles` is accessed and moved in cell space. Each time a particle is moved, an element of `Cells` is updated to reflect the changes of the three-dimensional coordinates of the particle so that the elements of `Cells` are accessed $O(N)$ times, in each iteration. Each element of `Ares` is also moved once and experiences exactly one collision ($O(N)$ accesses). Thus, the number of references in MP3D is $O(N)$ and the PTSM count is also $O(N)$.

The shared memory locations accessed by a processor are largely unpredictable. When a processor moves one particle in cell space, a randomly generated number decides if the processor should simulate a collision. These collisions are the only source of coherence misses for `Particles`. Moreover, the memory location of an element of `Cells` is selected by the space coordinates of the particle and is independent of the processor or of the index of the particle. This unpredictable behavior might cause serious prediction errors. However, robust estimation is rooted in statistics and the basic algorithmic step does not change as the data set size increases. The prediction results shown in Table 47 are very accurate and the prediction error for each type of misses and for the miss rate is way below 1%.

MP3D		Prediction-1: 256K molecules			Prediction-2: 512K molecules		
Total	**Empirical Model**	**simulation**	**prediction**	**error%**	**simulation**	**prediction**	**error%**
Reference	348.845 N + 41097.4	91485470	91488736	-0.003	182937948	182936384	0.000
CM	1.522 N - 23.046	398842	399127	-0.071	798626	798277	0.043
PTSM	17.321 N - 1507.18	4542127	4539559	0.056	9102790	9080176	0.248
PFSM	2.609 N + 5180.57	684831	689093	-0.622	1318448	1373006	-4.138
Essen. Miss	18.843 N - 1530.2	4940969	4938686	0.046	9901416	9878453	0.231
Total Miss	21.452 N - 3650.3	5625800	5627779	-0.035	11219864	11251459	-0.281
MissRate(%)	$\frac{21.452N - 3650.3}{348.845N + 41097.4}$	**6.1494**	**6.1513**	**-0.030**	**6.1332**	**6.1504**	**-0.280**

Table 47. Empirical models and prediction results for MP3D.

A.3 WATER

WATER		Prediction-1: 360 molecules			Prediction-2: 720 molecules		
VAR/Total	**Empirical Model**	**simulation**	**prediction**	**error%**	**simulation**	**prediction**	**error%**
Reference	$540.016\ N^2 - 2938.3\ N + 644.47$	71035088	71044592	-0.013	282029536	282060862	-0.011
CM	48.000 N - 28.015	17252	17252	0.000	34533	34532	0.000
PTSM	$4.490\ N^2 + 103.164\ N - 193.48$	619504	618789	0.115	2403970	2401458	0.104
Essen. Miss	$4.490\ N^2 + 151.164\ N - 221.48$	636756	636040	0.112	2438503	2435989	0.103
Total Miss	$4.490\ N^2 + 151.164\ N - 221.48$	636756	636040	0.112	2438503	2435989	0.103
MissRate(%)	$\frac{4.490N^2 + 151.164N - 221.48}{540.016N^2 - 2938.3N + 644.47}$	**0.8964**	**0.8953**	**0.122**	**0.8646**	**0.8636**	**0.115**

Table 48. Empirical models and prediction results for WATER.

WATER performs an *N*-body molecular dynamics simulation of the forces and potentials in a system of water molecules. The data set size parameter *N* is the number of molecules in array `VAR`. In each of $O(1)$ iterations every $O(N)$ element of `VAR` is accessed to compute the intra-molecular and inter-molecular interactions with *N*/2 elements ($O(N)$)

ahead of it. Thus the number of references is $O(N^2)$ and the CM count is $O(N)$. In addition, for each element of `VAR`, a processor must read $N/2$ components, which were mostly modified by other processors. Thus, the PTSM count is $O(N^2)$. From Table 48, we see that the prediction is extremely accurate for WATER.

A.4 OCEAN

This program performs the simulation to predict large-scale ocean movements. The computation is composed of many time steps. In every time step several spatial partial differential equations are solved on sets of two-dimensional fixed-sized grids, each set corresponding to an horizontal cross-section of the ocean basin. The solution method is a multi-grid Gauss-Seidel iteration with Successive Over Relaxation (SOR). There are 25 two-dimensional arrays that store discretized functions associated with the equations of the model. `q_multi` and `rhs_multi` are multi-level grids; their size is $O(N^2)$. Thus the number of cold misses is $O(N^2)$. In each iteration, the two $O(N^2)$ grids are accessed repetitively and the number of repetitions is independent of the data set size. Hence, the number of references is $O(N^2)$. Data sharing is observed only at the partition boundaries, and thus the PTSM count is $O(N)$.

OCAEN		Prediction-1: 258			Prediction-2: 514		
Total	**Empirical Model**	**simulation**	**prediction**	**error%**	**simulation**	**prediction**	**error%**
Reference	$4576.93\,N^2 - 53728.1\,N - 1562387$	329996291	316955936	3.951	1277460884	1235256448	3.303
CM	$6.911\,N^2 + 79.348\,N + 229.352$	482821	480757	0.013	1867445	1867001	0.023
PTSM	$1021.86\,N + 6299.61$	275316	269940	1.952	519512	531538	-2.315
PFSM	$1036.65\,N + 3052.20$	273927	270479	1.258	532095	535860	-0.707
Essen. Miss	$6.911\,N^2 + 1101.21\,N + 6528.96$	756137	750697	0.719	2386957	2398539	-0.485
Total Miss	$6.911\,N^2 + 2137.86\,N + 9554.16$	1030064	1021176	0.862	2919052	2934399	-0.525
MissRate(%)	$\frac{6.911N^2 + 2137.86N + 9554.16}{4596.93N^2 - 53728.1N - 1562387}$	**0.3121**	**0.3221**	**-3.204**	**0.2285**	**0.2375**	**-3.938**

Table 49. Empirical models and prediction results for OCEAN.

OCEAN has been the most challenging program analytical model in two aspects. First, because of the huge memory requirement we could only collect sample for three small data sets. This is the minimum needed to estimate the three unknown coefficients of a polynomial of order N^2. Furthermore, the execution time of OCEAN is strongly dependent on the values of the data since it runs until the changes in computed values are smaller than a preset threshold. Nevertheless, Table 49 shows that the prediction errors are well below 5% in all cases.

A.5 FFT

FFT is a one dimensional version of the radix-$\sqrt{N}$ six-step FFT algorithm. The two $\sqrt{N}$-by-$\sqrt{N}$ data arrays of complex numbers are `x`, which contains the data points, and `trans`, which contains the roots of unity. Two other data structures, umain and umain2, are only accessed by their home processors. To reflect the fact that the arrays are $\sqrt{N}$-by-$\sqrt{N}$, we choose n=$\sqrt{N}$ to characterize the data set size. FFT is not iterative and each element of `x` and `trans` (size $O(n^2)$) is accessed $O(1)$ times. So, the total number of references and the cold miss count are $O(n^2)$. The PTSM count is $O(n^2)$. As can be seen from Table 50 the number of misses and the miss rate are predicted with an error below 5%.

FFT		Prediction-1: 2^{16}			Prediction-2: 2^{18}		
Total	**Empirical Model**	**simulation**	**prediction**	**error%**	**simulation**	**prediction**	**error%**
Reference	$197.952\, n^2 - 2614.8\, n + 24184.5$	11993088	12327778	-2.790	52166656	50577336	3.046
CM	$1.878\, n^2 + 15.903\, n - 0.474$	127228	127169	0.045	501232	500538	0.138
PTSM	$0.437\, n^2 + 7.005\, n - 0.198$	30464	30463	0.001	118272	118267	0.003
Essen. Miss	$2.315\, n^2 + 22.908\, n - 0.672$	157692	157632	0.038	619504	618805	0.112
Total Miss	$2.315\, n^2 + 22.908\, n - 0.672$	157692	157632	0.038	619504	618805	0.112
MissRate(%)	$\frac{2.315n^2 + 22.908n - 0.672}{197.952n^2 - 2614.8n + 24184.5}$	**1.3149**	**1.2787**	**2.753**	**1.1875**	**1.2234**	**-3.023**

Table 50. Empirical models and prediction results for FFT.

A.6 BARNES

This application simulates the interaction of a three dimensional system of bodies for a number of time-steps using the Barnes-Hut hierarchical *N*-body method [7]. A primary one dimensional shared array `bodytab` of size *N* keeps the particles' information. Two more arrays form an octree: `ltab` with *O(N)* elements is an array of bodies at the leaves of the tree and `ctab` with *O(logN)* elements is an array with the non terminal nodes of the tree. Given the size of the three shared arrays, the number of cold misses is *O(N+logN)*. In each of the *O*(1) iterations, every *O(N)* particle element of `bodytab` partially traverses the octree to visit *O(logN)* non-terminal cells (elements of `ctab`). Then, an element of `ltab` is accessed. Thus, the number of references to `ctab` and `bodytab` are respectively *O(NlogN)* and *O(N)*. The total number of shared data references is *O(NlogN)*+*O(N)*+*O(logN)*+*O*(1) and the number of PTSM is *O(N)*+*O(logN)*+*O*(1).

BARNES		Prediction-1: 512K			Prediction-2: 1024K		
Total	Empirical Model	simulation	prediction	error%	simulation	prediction	error%
Reference	1491.8 NlogN+3278.6 N-2741459 logN+6796905	33733353	33701928	0.093	74726397	74732568	-0.008
CM	28.532 N - 108.061 logN + 4471.05	115856	120907	-4.360	239227	237740	0.062
PTSM	5.088 N + 10919.2 logN - 25920.3	39415	38597	2.073	62819	63078	-0.413
PFSM	1.323 N + 1154.58 log N - 2800.35	7392	7239	2.062	12994	13045	-0.400
Essen. Miss	33.620 N + 10811.1 logN - 21449.3	155271	159504	-2.725	302046	301852	0.064
Total Miss	34.943 N + 11965.7 log N - 24249.6	162663	166743	-2.508	315040	314845	0.061
MissRate(%)	$\frac{34.943N + 11965.7\log N - 24249.6}{1491.8N\log N + 3278.6N - 2741459\log N + 6796905}$	**0.4822**	**0.4947**	**-2.592**	**0.4216**	**0.4213**	**0.071**

Table 51. Empirical models and prediction results for BARNES.

A.7 RADIX

RADIX performs integer radix sort. In each iteration, an *r*-bit digit field of the keys are sorted, *r* being the radix. Most data accesses are made to two one-dimensional arrays `key[0]` and `key[1]`. Hence, the number of cold misses is *O(N)*. The number of iterations depends on the largest possible key values and on the radix, both of which are runtime parameters. Thus the number of iterations is (*O*(1)). In each iteration, every *O(N)* element of one of the two arrays `key[0]` or `key[1]` is accessed to compute a local histogram and partially sorted results are stored into the other array. Thus the numbers of references and of PTSM are both $O(N)$.

The number of false sharing misses is much more difficult to predict than the number of true sharing misses. The number of false sharing misses to `key`'s is poorly predicted. However, the number of these misses is very small and does not affect overall prediction quality.

RADIX		**Prediction-1: 512K**			**Prediction-2: 1024K**		
Total	**Empirical Model**	**simulation**	**prediction**	**error%**	**simulation**	**prediction**	**error%**
Reference	34.009 N + 219.030	17301504	17831184	-3.061	34603008	35662148	-3.060
CM	0.749 N + 1416.06	393235	394264	-0.261	786957	787113	-0.019
PTSM	0.684 N + 1640.14	359634	360205	-0.158	718224	718769	-0.076
PFSM	0.004 N + 519.106	2961	2435	17.752	4350	4351	-0.037
Essen. Miss	1.433 N + 3506.20	752869	754469	-0.212	1505181	1505882	-0.046
Total Miss	1.437 N + 3575.31	755830	756904	-0.142	1509531	1510233	-0.046
MissRate(%)	$\frac{34009N + 219.030}{1.437N + 3575.31}$	**4.3686**	**4.2448**	**2.833**	**4.3624**	**4.2348**	**2.924**

Table 52. Empirical models and prediction results for RADIX.

Bibliography

[1] Agarwal, A. and Gupta, A., "Memory Reference Characteristics of Multiprocessor Applications under MACH," *ACM SIGMETRICS*, 1988.

[2] Aho, V. A., Hopcroft, J. E. and Ullman, J. D., "Data Structures and Algorithms," *Addison-Wesley Publishing Company*, 1983.

[3] Amdahl, G. M., "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," *Proc. AFIPS Conf.* 31, pp. 483-485, 1967.

[4] Anderson, T. E. and Lazowska, E. D., "Quartz: A Tool for Tuning Parallel Program Performance," *Proc. of the 1990 Conf. on Measurement & Modeling of Computer Systems (SIGMETRICS)*, pp. 115-125, May 1990.

[5] Bagrodia, R., Chandy, M. and Dhagat, M., "UC: A Set-Based Language for Data-Parallel Programming," *Journal of Parallel and Distributed Computing*, 28, pp. 186-201, 1995.

[6] Bailey, D. H., "FFT's in External or Hierarchical Memory," *Journal of Supercomputing*, 4(1): pp. 23-35, Mar. 1990.

[7] Barnes, J. E. and Hut, P., "A Hierarchical O(NlogN) Force Calculation Algorithm," *Nature*, Vol. 324, No. 4, p.p 446-449, Dec. 1986.

[8] Beguelin, A. and Nutt, G., "Visual Parallel Programming and Determinacy: A Language Specification, an Analysis Technique, and a Programming Tool," *Journal of Parallel Processing and Distributed Computing*, 22, pp. 235-250, 1994.

[9] Bennett, J., Carter, J., and Zwaenepoel, W., "Adaptive Software Cache Management for Distributed Shared Memory Architectures," *Proc. of 17th Int. Symp. on Comp. Arch*, May 1990.

[10] Blelloch, G. E., Leiserson, C. E., Maggs, B. M., Plaxton, C. G., Smith, S. J. and Zagha, Macro, "A Comparison of Sorting Algorithms for the Connection Machine CM-2," *Proc. of the Symp. on Parallel Algorithms and Architectures*, pp. 3-16, Jul. 1991.

[11] Boyle, J., et al. "Portable Programs for Parallel Processors". *Holt, Rinehart, and Winston Inc.* 1987.

[12] Brorsson, M., Dahlgren, F., Nilsson, H., and Stenstrom, P., "The CacheMire Test Bench--A Flexible and Effective Approach for Simulation of Multiprocessors," *Proc. of 26th Ann. IEEE International Simulation Symposium*, pp. 41-49 Apr. 1993.

[13] Brorsson, M., "SM-prof: A Tool to Visualise and Find Cache Coherence Performance Bottlenecks in Multiprocessor Programs," *Proc. of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pp.178-187, May 1995.

[14] Borosson, M. and Stenstrom, P., "Visualizing Sharing Behavior in Relation to Shared Memory Management," *Proc. of the 1992 International Conf. on Parallel and Distributed Systems*, pp. 528-536, Dec. 1992.

[15] Borosson, M. and Stenstrom, P., "Visualization of Cache Coherence Bottlenecks in Shared Memory Multiprocessor Applications," *IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, pp. 32-36, Fall 1993.

[16] Burkhart, H., Frank, R. and Hachler, G., "Structured Parallel Programming: How Informatics Can Help Overcome The Software Dilemma," *Scientific Programming*, 1996.

[17] Burkhart, H, Frank, R., Hachler, G, Ohnacker, P. and Pretot, G., "ALWAN Programmer's Manual," *Tech. Report 94-4, Institut fur Informatik, University of Basel, Switzerland*, Nov. 1994.

[18] Burkhart, H., Korn, C. F, Gutzwiller, S., Ohnacker, P. and Waser, S., "BACS: Basel Algorithm Classification Scheme, Version 1.1," *Tech. Report 93-3, Institut fur Informatik, University of Basel, Switzerland*, March 1993.

[19] Censier, L. M., and Feautrier, P., "A New Solution to Coherence Problems in Multicache Systems," *IEEE Trans. on Computers*, Vol. C-27, No. 12, pp. 1112-1118, Dec. 1978.

[20] Chandy, K. M., and Misra, J., "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communication of ACM*, 24:11, pp. 198-206, Apr. 1981.

[21] Cormen, T. H., Leiserson, C. E. and Rivest, R. L., "Introduction to Algorithms," *McGrow-Hill Book Co.,* 1989

[22] Driscall, M. and Daasch, W. R., "Accurate Predictions of Parallel Program Execution Time," *Journal of Parallel and Distributed Computing*, 25, pp.16-30, 1995.

[23] Dubois, M. and Briggs, F. A., "Effects of Cache Coherency in Multiprocessors," *Proc. of 9th Ann. Int. Symp. on Computer Architecture*, pp.299-308, May 1982.

[24] Dubois, M. and Wang, J. C., "Shared Block Contention in a Cache Coherence Protocol," *IEEE Transactions on Computers*, Vol. 40 No. 5, May 1991.

[25] Dubois, M., Skeppstedt, J., Ricciulli, L., Ramamurthy, K., and Stenstrom, P., "Detection and Elimination of Useless Misses in Multiprocessors," *Proc. of 20th Ann. Int. Symp. on Computer Architecture*, pp.88-97, May 1993.

[26] Dubois, M., Skeppstedt, J., and Stentrom, P., "Essential Misses and Data Traffic in Coherence Protocols," *Journal of Parallel and Distributed Computing*, October 1995.

[27] Eggers, S. J. and Katz, R. H., "A Characterization of Sharing in Parallel Programs and Its Application to Coherency Protocol Evaluation," *Proc. of the 15th Annual Int. Symposium on Computer Architecture,"* pp. 257-270, May 1988.

[28] Eggers, S.J. and Katz, R. H., "The effect of the Sharing on the Cache and Bus Performance of Parallel Programs," *Proc. of the 3rd Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 257-270, 1989.

[29] Eggers, S. J., and Jeremiassen, T. E., "Eliminating False Sharing," *Proc. of the 1991 Int. Conf. on Par. Processing*, pp. I-377-I-381, Aug.1991.

[30] Fahringer, T., "Toward Symbolic Performance Prediction of Parallel Programs," *Proc. of Int'l parallel Processing Symposium*, pp. 474-478, Apr. 1996.

[31] Gemund, A. J. C., "Performance Modeling with PAMELA: An Introduction," *Tech. Report 1-68340-44(1992)01, Delft University of technology*, Dec.1992.

[32] Gemund, A. J. C., "Compile-time Performance Prediction of Parallel Systems," *Proc. Computer Performance Evaluation Modeling techniques and Tools*, Sept.1995.

[33] Goldberg, S. J. and Hennessy, "Mtool: An Intergrated System for Performance Debugging Shared Memory Multiprocessors Applications," *IEEE Transactions on Parallel and Distributed Systems*, 4(1), pp. 28-40, Jan. 1993.

[34] Golub, G. H. and Van Loan, C. F., "Matrix Computations," *2nd Ed. The Johns Hopkins University Press*, ch. 10, 1989

[35] Gorton, I., Gray, J. P. and Jelly, I., "Object-Based Modeling of Parallel Programs," *IEEE Parallel and Distributed Technology*, pp. 52-63, 1995.

[36] Gupta, A., and Weber, W-D., "Cache Invalidation Patterns in Shared-Memory Multiprocessors," *Proc. of IEEE Trans. on Computers* 41(7): pp. 794-810, July 1992.

[37] Gupta, A. and Kumar, V., "On the Scalability of FFT on Parallel Computers," *Proc. Frontiers 90 Conf. Massively Parallel Computation,* 1990.

[38] Gupta, S. K. S., Kaushik, S. D., Huang, C.-H. and Sadayappan, P., "Compiling Array Expressions for Efficient Execution on Distributed-Memory Machines," *Journal of Parallel and Distributed Computing,* 32, pp. 152-172, 1996.

[39] Gustafson, J. L., "Reevaluating Amdahl's Law," *Communication AC*M, 31, 5, pp. 532-533, May 1988.

[40] Hall, M., *et al.*, "Maximizing Multiprocessor Performance with the SUIF Compiler," *IEEE Computer*, pp. 84 - 89. Dec. 1996.

[41] Heath, M. and Worley, P., "Once Again Amdahl's Law," *Communication ACM*, 32, 2, pp. 262-263, Feb. 1989.

[42] High Performance Fortran Forum, "High Performance Fortran Language Specification," Version 2.0, Jan., 1997.

[43] Hogg, Robert V., "Adaptive Robust Estimation," *Journal of American Statistics Association*, 69, pp. 909-927, 1974.

[44] Johnson, L. and Riess, R. D., "Numerical Analysis, 2nd Ed.," *Addison-Wesley,* 1982.

[45] Kuhn, W., "The ALPSTONE Project: An Overview of a Performance Modeling Environment," *Institut fur Informatik, University of Basel, Switzerland*, March 1995.

[46] Kuhn, W., Ohnacker, P. and Burkhart, H., "Support for Software Reuse: The Basel Algorithm Library BALI," *Parallel Computing Conference*, Sept. 1995.

[47] Launer, R. L., and Wilkinson, G. N., "Robustness in Statistics," *Academy Press*, 1978.

[48] Lavenberg, S. S., "Computer Performance Modeling Handbook," *Academy Press*, 1983.

[49] Lawson, Charles L., and Hanson, Richard J., "Solving Least Squares Problems," *Englewood Cliffs, N.J., Prentice Hall,* 1974.

[50] Lazowska, E. D., Zahorjan, G. S. and Sevcik, K. C., "Quantitative System Performance, Computer System Analysis Using Queueing Network Models," *Prentice Hall,* 1984.

[51] Martonosi, M., "Analyzing and Tuning Memory Performance in Seuqential and Parallel programs," *Ph.D thesis, Dept. of Electrical Engineering, Stanford University,* Jan. 1994.

[52] Martonosi, M., Gupta, A. and Anderson, T., "MemSpy: Analyzing Memory System Bottlenecks in Programs," *Performance Evaluation Review*, 20(1), pp. 1-12, Jun. 1992.

[53] McMillan, K. L. and Schwalbe, J., "Formal Verification of the Gigamax Cache Consistency Protocol", *Proc. of the ISSM Int'l Conf. on Parallel and Distributed Computing*, Oct. 1991.

[54] Mounes-Toussi, F. and Lilja, D. J., "The Potential of Compile-Time Analysis to Adapt the Cache Coherence Enforcement Strategy to the Data Sharing Characteristics," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 6 No. 5, pp. 470-481, May 1995.

[55] Nanda, A. K. and Bhuyan, L. N., "A Formal Specification and Verification Technique for Cache Coherence Protocols", *Proc. of the 992 Int'l Conf. on Parallel Processing*, pp. I-22-I-26, 1992.

[56] Nussbaum, D. and Agarwal, A., "Scalability of Parallel Machines," *Comm. ACM, Vol. 34, No. 3,* pp. 57-61, Mar. 1991.

[57] Panwar, R. B., Kim, W. and Agha, G. A., "Parallel Implementations of Irregular Problems Using High-Level Actor Language," *Proc. of Int'l Parallel Processing Symposium*, pp. 857-862, 1996.

[58] Papoulis, A, "Probability and Statistics," *Prentice Hall,* 1990.

[59] Parashar, M. and Hariri, S., "Compile-Time Performance Prediction of HPF/Fortran 90D," *IEEE Performance Evaluation*, pp. 57-73, Spring 1996.

[60] Pong, F., "Symbolic State Model: A New Approach for the Verification of Cache Coherence Protocols,", *Ph.D thesis, Dept. Electrical Engineering, University of Southern California*, Aug. 1995.

[61] Press, W. I., Flannery, B. P., Teukolsky, S. A. and Vetterling, W. T., "Numerical Recipes," *Cambridge University Press*, 1986.

[62] Rose, J. S., "Parallel Global Routing for Standard Cells," *IEEE Trans. on Computer-Aided Design of Circuits and Systems*, Sept. 1990.

[63] Rothberg, E. and Gupta, A., "Techniques for Improving the Performance of Sparse Factorization on Multiprocessor Workstations," *Proc. of Supercomputing '90*, Nov. 1990.

[64] Rothberg, E., Singh, J. P. and Gupta, A., "Working Sets, Cache Sizes and Node Granularity Issues for Large-Scale Multiprocessors," *Proc. of the 20th Annual Int. Symposium on Computer Architecture,"* pp. 14-25, May 1993.

[65] Rudolf, L. and Segall, Z., "Dynamic Decentralized Cache Schemes for MIMD Parallel Processors", *Proc. of the 11th Int'l Symposium on Computer Architecture*, pp. 340-347, June 1984.

[66] Scherson, I. D. and Corbett, P. F., "Communications Overhead and the Expected Parallel Speedup of Multidimensional Mesh-Connected Parallel Processors," *Journal of Parallel and Distributed Computing*, 11, 1, pp. 86-96, Jan. 1991.

[67] Singh, J. P., Weber, W-D, and Gupta, A., "SPLASH: Stanford Parallel Applications for Shared-Memory," *Computer Architecture News,* 20(1):5-44 March 1992.

[68] Singh, J. P. and Hennessy, J. L., "Finding and Exploiting Parallelism in an Ocean Simulation Program: Experience, Results and Implications," *Journal of Parallel and Distributed Computing* Vol 15, No. 1, pp. 27-48 May 1992.

[69] Singh, J. P., Hennessy, J. L., and Gupta, A., "Scaling Parallel Programs for Multiprocessors: Methodology and Examples," *IEEE Computer* pp. 42-50 July 1993.

[70] Singh, J. P., Gupta, A. and Levoy, M., "Parallel Visualization Algorithms: Performance and Architectural Implications," *IEEE Computer* 27(7). pp. 45-55 July 1994.

[71] Soule, L. and Gupta, A., "Analysis of Parallelism and Deadlocks in Distributed-time Logic Simulation," *Tech. Rep. CSL-TR-89-378, Stanford University*, Mar. 1989.

[72] Srbljic, S., Vranesic, Z. G., Stumm, M. and Budin, L., "Models for Performance Prediction of Cache Coherence Protocols", *Tech. Report CSRI-332, Computer Systems research Institue, University of Toronto*, July, 1995.

[73] Stanford SUIF Compiler Group, "SUIF: A Parallelizing & Optimizing Research Compiler", tech. Rep. CSL-TR-94-620, Computer Systems Lab, Stanford University, May 1994.

[74] Stenstrom, P., "A Survey of Cache Coherence Scheme for Multiprocessors," *IEEE Computer*, Vol. 23, No. 6, pp. 12-24, Jun 1990.

[75] Tawbi, N., "Estimation of Nested Loop Execution Time by Integer Arithmetic in Convex Polyhedra," *Proc. of Int'l parallel Processing Symposium*, Apr. 1994.

[76] Torrelas, J., Lam, M. S., and Hennessy, J. L., "Shared Data Placement Optimization to Reduce Multiprocessor Cache Misses," *Proc. of the 1990 Int. Conf. on Par. Processing*, pp. 266-270, Aug. 1990.

[77] Tsai, J. and Agarwal, A., "Analyzing Multiprocessor Cache Behavior Through Data Reference Modeling," *Proc. of ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems,* pp. 236-247, May 1990.

[78] Vernon, M. K., and Holliday, M. A., "Performance Analysis of Multiprocessor Cache Consistency Protocols Using Generalized Timed Petri Nets," *Proc. of Performance 86 and ACM SIGMETRICS*, pp. 9-17, May 1986.

[79] von Mises, Richard, "Mathematical Theory of Probability and Statistics," *Academy Press, New York*. 1964.

[80] Wabnig, H. and haring, G., "Performance Prediction of parallel Systems with Scalable Specifications - Methodology and Case Study," *ACM Performance Evaluation Review*, Vol. 22 #2-4, pp. 46-62, Apr. 1995.

[81] Weber, W-D. and Gupta, A., "Analysis of Cache Invalidation Patterns in Multiprocessors," *Proc. of the 3rd International conference on Architectural Support of Programming languages and Operating Systems*, pp. 243-256, April 1989.

[82] Woo, S. C., Ohara, M., Torrie, E. Singh, J. P. and Gupta, A., "The SPLASH-2 Programs: Characterization and Methodological Consideration," *Proc. of 22nd Ann. Int. Symp. on Computer Architecture*, pp.24-36, May 1995.

[83] Wood, D. A. and Lebeck, A. R., "Cache Profiling and the SPEC Benchmarks: A Case Study," *IEEE Computer Magazine*, pp. 15-26, Oct. 1994.