This is to certify that the

dissertation entitled

A Framework for Multiprocessor Performance Characterization and Calibration

presented by

Arun K. Nanda

has been accepted towards fulfillment of the requirements for

Ph.D. degree in Computer Science

Lionel M. Ni
Major professor

Date 10/12/92

A Framework for Multiprocessor Performance Characterization and Calibration

By

Arun K. Nanda

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science

1992

ABSTRACT

A Framework for Multiprocessor Performance Characterization and Calibration

By

Arun K. Nanda

In parallel programs using the shared-variable paradigm, run-time communication overhead manifests itself along three principal dimensions, namely, shared data accesses (including memory contention, cache misses and non-local memory access latencies), inter-process synchronization operations, and global barrier synchronizations. Performance measurements to quantify the rate at which communication costs for an algorithm increases as more processors are used is integral to the study of an algorithm's efficiency and scalability. In this thesis, we explore the problem of performance characterization of a multiprocessor in the context of the shared-variable programming model with emphasis on characterizing the dynamic run-time behavior.

We have developed a hierarchical model to characterize multiprocessor system performance using a multi-phase computation structure with concurrent asynchronous execution within a phase. Two sets of system characterization parameters have been proposed that completely describe the static and dynamic behavior of a given input workload on a target multiprocessor system. The characterization parameters are calibrated by experimental measurements on the input workload. A series of *loss* functions are formulated to describe the performance degradation resulting from static and dynamic overheads, thus providing realistic estimates of performance loss.

Since the characterization of performance is tied inextricably with the input workload, we have presented a flexible technique for benchmark workload generation, that can be tailored to fit a user's preference for selective workload characteristics. A family of workload emulation kernels, namely, the MAD, SAD and BAD kernels, have been designed to isolate and measure the incremental impact of memory contention, critical sections and barrier synchronization on performance, repectively, to calibrate the hierarchical performance model. We have demonstrated the applicability of the system characterization methodology and the effectiveness of the workload emulation kernels by evaluating the performance of several synthetic workloads on the Sequent Symmetry and BBN TC2000 commercial multiprocessors.

The proposed methodology is independent of any particular architecture or application. We believe that our approach to performance characterization will serve to model performance with greater fidelity than exists in the current state of art, since it incorporates the effect of both static and dynamic influences in a workload execution. Since a shared-variable programming paradigm is only assumed with no assumptions made about the organization of the shared address space, our framework can be used equally effectively to evaluate multiprocessors that provide a physical shared memory or highly-parallel systems that support a shared virtual memory.

To my parents

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my appreciation to those who have contributed to the completion of this dissertation. I will always be indebted to my advisor, Lionel Ni. He has been my mentor, my colleague, and my friend. His guidance has helped me mature as a researcher, and his respect for my ideas has made working with him very rewarding. I look forward to many fruitful interactions with him in the future.

I am very grateful to the other members of my dissertation committee: Richard Enbody, for his invaluable discussions on numerous occasions and comments to improve the readability of this thesis, his perpetual willingness to listen to whatever I had to say, be it research related or otherwise, and offer friendly advice; Abdol Esfahanian, for being my faculty advisor for two years, his critical suggestions on some aspects of this thesis, and for his time and support; V. Mandrekar, for his continuous encouragement and always accommodating me in his schedule at short notice.

I would like to thank the members of the Advanced Computing Research Facility at Argonne National Laboratory, especially Dave Levine, for providing me access to their computer systems and their help in arranging my special job scheduling requests.

My thanks to Honda Shing and Ten-Hwan Tzen for many enlightening discussions on research issues.

A person cannot accomplish anything without the help and understanding of family members. My mother's constant encouragement, in spite of her personal hardships, inspired me to do my best. My brother and sister always stood behind all my decisions. My father- and mother-in-law offered their patient understanding throughout the course of my doctorate work. I proudly share this accomplishment with them all.

Finally, my very special thanks to my wife Susmita, for sustaining me with her continuous love and understanding, and spending many a sleepless nights with me during my work to keep me company.

TABLE OF CONTENTS

LIST OF TABLES x

LIST OF FIGURES xi

1 INTRODUCTION 1

1.1 Multiprocessor Performance Evaluation 2
1.2 Survey of Benchmarks 8
1.2.1 Synthetic Benchmarks 8
1.2.2 Kernel Benchmarks 9
1.2.3 Application Benchmarks 10
1.3 Motivation and Problem Definition 10
1.4 Objective and Scope of Research 14
1.5 Thesis Outline 20

2 BACKGROUND 21

2.1 Multiprocessor Memory Organization 21
2.2 Limitations to Parallelism 24
2.2.1 Memory Access Contention 25
2.2.2 Spin Locks and Mutual Exclusion 31
2.2.3 Synchronization Barriers 38
2.3 Target System Architectures 40
2.4 Summary 45

3 PERFORMANCE CHARACTERIZATION METHODOLOGY 46

3.1 The Parallel Computation Model 47
3.2 Workload Characterization 50
3.2.1 The Unit Grain 51
3.2.2 Workload Classification 53
3.3 Experimental Framework 55
3.3.1 Measurement Structure 55
3.3.2 Workload Generation 59

3.4 Performance Characterization Parameters 60
3.4.1 Static Parameters 61
3.4.2 Dynamic Parameters 67
3.4.3 Performance Metrics 72
3.4.4 Aggregate Multiphase Performance 74
3.5 The Workload Emulation Kernels 75
3.5.1 Measurement of Incremental Overheads 76
3.5.2 Kernel Structure 78
3.5.3 Minimization of Experimental Errors 80
3.6 Summary 83

4 MAD KERNELS AND MEMORY ACCESS PERFORMANCE 85
4.1 Preliminary Studies 86
4.1.1 Workload Parameters 86
4.1.2 Quantities Measured 88
4.1.3 Memory Access Overhead Factors 88
4.1.4 Experimental Results 94
4.2 MAD Workload Parameters 97
4.2.1 Unit Grain Characterization 98
4.2.2 Output Metrics 102
4.3 Concurrent-Access Workloads 103
4.3.1 Homogenous Workloads 103
4.3.2 Heterogenous Workloads 111
4.4 Dual-Mode Access Workloads 114
4.5 Summary 115

5 SAD KERNELS AND SYNCHRONIZATION PERFORMANCE 117
5.1 Preliminary Studies 118
5.1.1 Synchronization Overhead Factors 119
5.1.2 Experimental Results 121
5.2 SAD Workload Parameters 130
5.2.1 Unit Grain Characterization 130
5.2.2 Output Metrics 132
5.2.3 Lock Implementations Studied 133
5.3 Exclusive-Access Workloads 137
5.4 Dual-Mode Access Workloads 144
5.4.1 Homogenous Workloads 144
5.4.2 Heterogenous Workloads 147

5.5 Summary 150

6 BAD KERNELS AND BARRIER PERFORMANCE 152

6.1 BAD Workload Parameters 153

6.1.1 Phase Characterization 153

6.1.2 Output Metrics 154

6.1.3 Barrier Implementations Studied 155

6.2 Embarrassing Workloads 158

6.2.1 Scalability of Barrier Implementations 162

6.2.2 Balanced Load and Simultaneous Arrivals 162

6.2.3 Unbalanced Load and Staggered Arrivals 164

6.3 Dual-Mode Access Workloads 166

6.4 Summary 169

7 CONCLUSIONS 171

7.1 Research Contributions 171

7.2 Directions for Future Research 174

BIBLIOGRAPHY 176

LIST OF TABLES

1.1 Performance level comparisons for three classes of multiprocessors . . 5

2.1 Summary of target system architectures 44

3.1 An example of weights assigned to different types of floating-point operations to normalize their execution time to floating-point addition time 53
3.2 Summary of average shared data access time t_m 63
3.3 System characterization parameters 72
3.4 Application parameters used in the performance model 72
3.5 Summary of access degradation kernel measurements 79

4.1 Basic time measurements for the overhead factors model 90
4.2 Parameter settings for different workload types used in the preliminary studies 95
4.3 Unit grain attributes for studying memory access behavior 99
4.4 Static characterization parameters for a homogenous workload with $M = 128K, G_t = G_c = (g_m = (0/1, 0, \vec{s}, 1), g_c = \phi, g_s = \phi)$. 106

5.1 Actual execution times ($M = N + 1, \omega = 500, x = 50\mu s$) 128
5.2 Actual overhead times ($M = N + 1, \omega = 500, x = 50\mu s$) 128
5.3 Unit grain attributes for studying synchronization behavior 131
5.4 Native lock support on each machine 134
5.5 Pseudo-code for the TAS lock 134
5.6 Latency of locks used in the SAD experiments 138
5.7 Half-performance lock factor $c_{1/2}$ for different lock implementations . 139
5.8 Static characterization parameters for workloads used in incremental overhead measurements 146

6.1 Workload parameters for studying barrier performance 153

LIST OF FIGURES

1.1 Performance measurement levels 4
1.2 Steps in the experimental performance characterization method . . . 17

2.1 Organization of memory hierarchy in shared-memory multiprocessors 23
2.2 Tree saturation as a result of *hot spot* accesses over a multistage interconnection network 27
2.3 Memory address interleaving techniques: (a) Fine interleaving with sequential assignment across modules (one bank per module); (b) Coarse interleaving with sequential assignment within module (one bank per module); (c) Mixed scheme with fine interleaving among banks of a module and coarse interleaving among modules (multiple banks per module) 29
2.4 Sequent Symmetry system architecture 41
2.5 BBN TC2000 system architecture 42

3.1 Structure of parallel program execution 49
3.2 Structure of a unit grain 52
3.3 Structure of a single computational phase 54
3.4 Structure of the measurement framework 56
3.5 Incremental measurement of dynamic overheads 77
3.6 The concurrent loop structure of the kernels 80
3.7 Normalized 90 percent confidence intervals for three workload measurements on the Sequent Symmetry for $N_{repeat} = 5, 10, 20$ 82

4.1 Efficiency vs. N $(M = 1, \omega = 100, x = 0)$ 96
4.2 Efficiency vs. N $(M = N + 1, \omega = 100, x = 0)$ 97
4.3 Creation of memory access patterns using attributes d and s 100
4.4 Effect of spatial distribution of memory access stream on performance 105
4.5 Effect of temporal distribution of memory access stream on performance 107
4.6 Effect of contention for a memory location (*hot-spot*) on performance 108
4.7 Effect of length of computation on *hot-spot write* performance 109

4.8 Effect of shared-data size on *read* performance 111
4.9 Random access performance expressed in MegaWARPS 112
4.10 Interaction between read and write memory-access streams 113
4.11 Effect of length of computation on interference between read and write streams 114

5.1 Generic structure of program executed by every processor 119
5.2 Efficiency vs. N $(M = N + 1, \omega = 100, \rho = 0)$ 121
5.3 Efficiency vs. N $(M = N + 1, \rho = 0.1, x = 30\mu s)$ 122
5.4 Efficiency vs. N $(M = N + 1, \omega = 100, x = 100\mu s)$ 123
5.5 Efficiency vs. N $(M = N + 1, \omega = 100, \rho = 0.3)$ 124
5.6 Overhead components vs. N $(M = N + 1, \omega = 500, \rho = 0.1, x = 30\mu s)$ 126
5.7 Overhead components vs. x $(M = N + 1, \omega = 500, \rho = 0.1)$ 127
5.8 Overhead components vs. ρ $(M = N + 1, \omega = 500, x = 50\mu s)$ 129
5.9 Critical section structure 132
5.10 Pseudo-code for the MCS list-based queuing lock 136
5.11 Working of the MCS list-based queuing lock 137
5.12 Effect of frequency of CS on performance 141
5.13 Effect of non-CS to CS computation ratio on performance 143
5.14 Effect of non-CS to CS shared data access ratio on performance . . . 145
5.15 Incremental interference measured with stride of access $s = 1$ 147
5.16 Incremental interference measured with stride of access $s = 23$ 148
5.17 Impact of non-CS memory accesses on CS execution performance . . 149
5.18 Impact of CS spin-lock on non-CS memory accesses 150

6.1 Pseudo-code for a sense reversing centralized barrier 155
6.2 Pseudo-code for a distributed dissemination barrier 157
6.3 Time to achieve barrier vs. N 161
6.4 Time to achieve DSM barrier on the TC2000 163
6.5 Barrier performance of a perfectly balanced load 164
6.6 Barrier performance of an unbalanced load 165
6.7 Performance of staggered arrivals at the barrier 167
6.8 Cumulative interferences unit stride workload on the Symmetry . . . 168
6.9 Cumulative interferences unit stride workload on the TC2000 169

CHAPTER 1

INTRODUCTION

The ever increasing need for faster and more powerful computers, coupled with the advent of fairly cheap microprocessors, has prompted considerable interest in massively parallel processor systems. Computational power has reached a plateau at the current state of technology for single processor systems [23], due to certain fundamental limits (*i.e.*, the speed of light and the width of the atom) being approached. In an effort to sustain increases in the peak speed of new computer systems so as to bridge the discrepancy between computational needs and available computing power, designers have turned to multiple processors, vector arithmetic units, and other architectural innovations. Using a large number of low-cost processors for achieving supercomputing performance is attractive indeed. Unfortunately, it is much more difficult for a programmer or a compiler to take advantage of multiple processors than of a faster clock speed. As a result, many machines with complex architectures are able to deliver only a small fraction of their theoretical peak performance on all but the most ideal problems.

The purpose of this dissertation research is to develop a flexible approach to characterize multiprocessor systems for general purpose parallel programming that can measure and quantify the expected losses in parallel execution performance and determine performance bottlenecks for any selected workload. The proposed methodology provides a framework for customized benchmark workload generation and yields a set of parameters which characterize the target system. These parameters spotlight the strong and weak points of a machine and, hence, aid in the design of efficient

algorithms for it. It should be emphasized that it is not the intent of this thesis to address the issue of performance prediction of application programs. We have chosen the *shared-memory programming model* as the focus of our study. In this model, processes communicate with each other through *shared-variables* residing in globally accessible memory. The shared-memory programming model is widely believed to be easier to use than the message-passing model. The conceptual simplicity of the shared-memory model derives from similarities with sequential programming. Evidence in favor of the shared-memory model is the overwhelming dominance of shared-memory multiprocessors for general purpose parallel programming, and the considerable effort in software development designed to provide the illusion of shared memory on multicomputers.

In this introductory chapter, we elaborate some of the pertinent issues in multiprocessor performance evaluation, provide a brief survey of the commonly used multiprocessor benchmarks, and describe the objective and scope of this research.

1.1 Multiprocessor Performance Evaluation

The goal of computer performance evaluation is to identify opportunities for specific performance improvements throughout the life of a computer system and to guide the design of more effective architectures. The requirements of target applications motivate the development of new systems; the development of novel systems creates the need and the basis for performance evaluation research. Effective performance evaluation of highly-parallel systems is essential because these systems must function at the limits of their computing potential in order to meet the overwhelming demands of large scientific applications. However, analyzing the performance of multiple-processor systems is a very complex task since many factors jointly determine system performance, and the modification of some factors affects others. Since many different tradeoffs are involved, it is crucial to carefully tune various parameters such that a system achieves its peak performance.

Traditionally, three common approaches are used to evaluate multiprocessor per-

formance: analytical, simulation and experimental [56]. All three approaches are necessary because each has its own advantages and limitations. *Analytical* models are extremely powerful in the sense that they allow the analytical correlation of performance with organizational parameters. However, their applicability is not universal. In order to be tractable, they typically have to make many simplifying assumptions about the architecture and application characteristics that may not reflect an accurate representation of reality. For example, memory interference models for multiprocessors based on queueing theory often assume a randomly distributed (both in time and space) memory request stream. This assumption fails for many scientific and engineering applications that exhibit very regular data access patterns. If vector instructions are used to implement these codes they must exploit, and hence emphasize, this regularity in the temporal and spatial distribution of requests. *Simulations* can generally approximate reality more closely, but they are expensive to run and still do not replace real measurements. Moreover, interactions may be present on a real system that affect performance and are difficult to capture in a model.

The advantage of *experimental* performance analysis is, of course, that the performance of the real system is obtained as opposed to the performance of a model of the system. The drawback of such a solution is its experimental nature which limits the number of codes analyzed and generally does not provide any methodology for extrapolating the performance of an arbitrary code from the performance of the benchmark codes. Furthermore, even when using very simple benchmarks, there is no general method of correlating code characteristics with the performance observed.

Analytical and simulation modeling techniques find maximum applicability at the system design phase where they facilitate prediction of system behavior long before the actual hardware implementation. This helps in making judicious design decisions that can avoid considerable investment of resources in an inefficient design. For example, analytical models of processor–memory interconnection have been studied in [86, 11, 18, 19]. Analytical models of application (or algorithm) execution on a given architecture can also aid in asymptotic scalability studies [47, 42]. However, hardware related parameters in such models need to be calibrated by experimental

measurements.

Owing to the diversity of architectural approaches of a multiprocessor, the development of working models that can provide a true measure of the "actual" performance of these machines under workloads of interest can be an extremely complex, if not impossible, problem. Since a multitude of architectural and application parameters jointly determine system performance and the modification of some factors affects others, it is not feasible to construct an elegant yet tractable analytical model that encompasses all performance effects. Nondeterminism present in parallel program execution on multiprocessors introduces an additional degree of complexity into the performance measurement phenomenon. The dynamic run-time behavior of multiprocessor programs is impossible to capture accurately in analytical models.

In the face of the above difficulties, empirical results are the only reliable performance measures [29]. This has led to the use of benchmark programs to characterize and evaluate parallel computer performance (*benchmarking*). Although benchmarking is widely acknowledged to be a difficult and often controversial process [87, 97], it also provides one of the few recognized means of acquiring useful performance information about complex systems running complex tasks. The methodologies commonly used in computer benchmarking and the associated pitfalls encountered are described in [35].

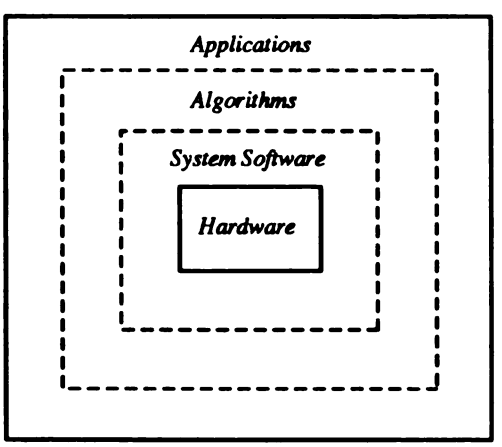


Figure 1.1. Performance measurement levels

There are four levels in the hierarchy of performance measurements [85] as illustrated in Figure 1.1. The answer to the oft-asked question, "How fast is it?" depends on the intended use of the performance data. At the lowest level lies the performance of the hardware design. Determining this performance provides both a validation of and directives for system software design. Only by understanding the strengths and weaknesses of the hardware can system software designers develop an implementation and user interface that maximizes the raw hardware potential available to the end user. Given some characteristics of the available processing resources and the services provided by the system software, users can develop algorithms that are best suited to the computer system's capabilities. Finally, the best mix of key algorithms will maximize the performance of user applications.

A complete performance characterization requires not only an analysis of the system's constituent levels, it also requires both *static* and *dynamic* characterizations. Static or average behavior analysis may mask transients that dramatically alter system performance. A combination of static and dynamic characterizations is also needed to understand the interactions between performance levels. Table 1.1 shows a subset of the important performance measurements at three levels for three classes parallel processing architectures.

Table 1.1. Performance level comparisons for three classes of multiprocessors

Level	**Vector processors**	**Shared-memory multiprocessors**	**Message-passing multicomputers**
Hardware	Vector startup Memory conflicts	Memory contention Network contention Memory-cache interaction	Processor speed Communication latency and bandwidth
System software	Compiler	Compiler	OS support
Algorithm	Vectorization	Shared-memory access Inter-processor synchronization	Communication pattern

Historically, benchmarking has been employed for system procurements. It will certainly maintain its value in that arena as it expands to become the experimental basis for a developing theory of supercomputer and multiprocessor performance evaluation. The number of benchmarks currently used is growing day by day. Every new benchmark is created with the expectation that it will become the standard of the industry and that manufacturers and customers will use it as the definitive test to evaluate the performance of computer systems with similar architectures. A survey of the common benchmarks in use today is provided in Section 1.2.

One of the key questions in benchmarking has to do with what kind of unit constitutes the benchmark set. A number of general benchmarks such as Livermore Fortran Kernels, NAS Kernels and the Linpack Benchmark have emerged during the past two decades that are based on a collection of computation-intensive kernels extracted from a range of real application domains. Another benchmark called Whetstones, on the other hand, is based on a collection of synthetic kernels. All these benchmarks perform measurements at the "algorithms" level of Figure 1.1 and have one thing in common—each component kernel in the benchmark is designed to stress a particular aspect of system performance.

Discussions of benchmarking [35, 60, 117, 125] have lead to a growing recognition that the most accurate information on a system's aggregate performance is obtained by making measurements on complete applications (*applications-based benchmarking*). The underlying assumption here is that real engineering and scientific codes stress and evaluate machines in a way that kernels and algorithms cannot. Efforts in this direction include the Perfect, SPLASH and SLALOM benchmarks. Performance measurements at the applications level capture and reflect the interactions that occur within and between all the lower levels (Figure 1.1). Although this is indeed true, these benchmarks provide useful measures of performance only to the particular set of users that are represented by the benchmark applications. Because of the complexity of designing a complete application program, when tests are done at this level rather than on simpler units, the skill of the programmer may be a significant factor in the performance. Some of the limitations with this approach are:

- Complete applications are difficult to port to a new architecture. Unless the existing applications are modified and tuned to the new architecture, they may not yield optimal performance.

- The software technology for writing parallel programs is immature. It is unclear how well programs written with today's constructs will represent those that might be written in the future, and what the implications of this are for the effectiveness of evaluation studies performed today.

- The available programs might not represent the best parallelization of the problem they solve, but only one that is reasonable and convenient to implement. More significantly, large-scale parallel processing may call for very different algorithms than those implemented on smaller machines today.

- The relationships between applications and architectures take on new dimensions with parallelism. The number of architectural variables is much larger, making careful correlation of performance with code characteristics more difficult. ·

Empirical studies based on carefully defined benchmark experiments at all the levels in Figure 1.1 can provide a hierarchical path to a complete definition of system performance by extending our understanding of the incremental contributions made by architecture, technology, compilers, operating systems, algorithms, and programming implementations of physical problems.

Finally, there is the question of appropriate metrics to represent multiprocessor performance. A single figure of merit such as MIPS (Millions of Instructions Per Second) is meaningless in the context of the diverse CPU architectures available today. The single number metric MFLOPS (Millions of Floating point Operations Per Second) is more appropriate for scientific computations, but yet insufficient. From the end user's standpoint, perhaps the desirable metric would be MRPS (Millions of Results Per Second), although this metric would have no universal meaning. Usually, different benchmark program measurements are summarized in order to find the "average" performance of a computer. How to calculate these averages has been one of

the most confusing issues in performance evaluation [41, 117]. Siegel *et al.* provide a detailed discussion of other metrics used for multiprocessor performance in [115].

1.2 Survey of Benchmarks

Benchmarks are standard programs used to evaluate the performance of a wide range of computer systems. What distinguishes a benchmark from an ordinary program is a general consensus of opinion within the industry and research circles that the benchmark exercises a computer well. Common benchmarks fall into one of several categories. *Synthetic benchmarks* are small programs especially constructed for benchmarking purposes with the underlying assumption that the average characteristics of real programs can be statistically approximated by a small program. They do not perform any useful computation. *Kernel benchmarks* are code fragments extracted from real programs in which the code fragment is believed to be responsible for most of the execution time. *Application benchmarks* go with the assumption that complete real applications stress and evaluate machines in a way that kernels and code fragments cannot. The most important advantage of reducing benchmarks to kernels is that they may be rapidly ported to new computer architectures, whereas porting a mature application would need a lot more effort. However, complete applications provide the most accurate indication of performance.

The field of multiprocessor benchmarking is still evolving and not yet mature. The methodologies commonly used in supercomputer benchmarks and some of the pitfalls encountered are examined by Dongarra *et al.* in [35]. Although there are a wide variety of benchmarks available, some very site-specific, there is no consensus yet on the most effective and acceptable multiprocessor benchmarks. We summarize some of the more commonly used benchmarks in this section.

1.2.1 Synthetic Benchmarks

Whetstone. The Whetstone benchmark [27] was the first program in the literature explicitly designed for benchmarking. It is a synthetic program constructed with

nine small loops each containing statements of a particular type (integer arithmetic, floating-point arithmetic, "if" statements, calls, *etc.*). It uses mostly global data and has a high percentage of floating-point operations. Most of its execution time is spent in mathematical library functions. The benchmark results are reported as MWIPS (mega Whetstone instructions per second).

Dhrystone. This is another synthetic benchmark [123] that consists of 12 procedures included in one measurement loop with 94 statements. It contains no floating-point operations and a considerable percentage of its execution time is spent in string functions. Unlike Whetstone, it uses very little global data and emphasizes data locality. The benchmark results are given in Dhrystones per second.

1.2.2 Kernel Benchmarks

Linpack. This is a numeric benchmark [33] with a high percentage of floating-point operations and no mathematical functions at all. More than 75 percent of its execution time is spent in a 15-line subroutine (called **saxpy** in the single-precision version and **daxpy** in the double-precision version). The results of this benchmark are reported in MFLOPS.

Livermore Fortran Kernels. Also called the Lawrence Livermore Loops, this benchmark [88] consists of 24 kernels (inner loops) of numeric computations from different areas of physical sciences. The individual loops range from a few lines to about one page of source code. They contain many floating-point computations and a high percentage of array accesses. The program computes MFLOPS rate for each kernel, for three different vector lengths.

NAS Kernels. This benchmark program [10] consists of approximately 1000 lines of Fortran code, organized into seven separate tests each containing a loop that iteratively calls a subroutine. The subroutines have been extracted from a variety of computational fluid dynamics problems currently being worked on the NASA Ames supercomputers. They all emphasize the vector performance of a computer system. The performance is measured in MFLOPS.

1.2.3 Application Benchmarks

Perfect Benchmarks. Prompted by Kuck and Sameh's proposal [69] and initiated by a group of academic and industrial collaborators, the goals of this effort were to define an applications-based methodology for supercomputer performance evaluation. The Perfect Benchmarks [29, 17] consist of 13 programs drawn from a variety of scientific and engineering fields with over 60,000 lines of Fortran source listing. The methodology requires a set of *baseline* measurements followed by any number of optimized measurements of each code.

SPLASH. Similar to the Perfect benchmarks, the Stanford Parallel Applications for Shared-Memory (SPLASH) [116] is a suite of seven applications drawn from several scientific and engineering problem domains. The applications are intended as a design aid for architects and software people working in the area of shared-memory multiprocessing.

SLALOM. The SLALOM benchmark [5] solves a complete problem dealing with "optical radiosity on the interior of a box". It times input, problem setup, solution, and output, not just the solution. It is the first benchmark based on fixed time rather than fixed problem comparison.

SPEC Benchmarks. Probably the most important current benchmarking effort is SPEC [120] — the systems performance evaluation cooperative effort. Its goal is to collect, standardize, and distribute large application programs that can be used as benchmarks. The SPEC suite consists of 10 benchmark programs. The results are given as performance relative to a VAX 11/780 using VMS compilers. A comprehensive number, the "SPECmark", is defined as the geometric mean of the relative performance of the 10 programs.

1.3 Motivation and Problem Definition

There are two distinct activities [110] in evaluating any computer that are often not distinguished in practice: system characterization and performance evaluation. The goal of *system characterization* is to obtain a set of parameters that fully describes

the system behavior at some level of abstraction. The characterization parameters spotlight the strong and weak points of the system they represent. *Performance **evaluation***, on the other hand, is the measurement of some number of properties during the execution of a given workload. The properties measured may be the total execution time to complete some job steps, the utilization of system resources, the amount of parallel execution overhead, *etc.* It is important to note that the results depend on, and are only valid for, the workload used in the evaluation.

Accurate performance characterization of a computer is crucial to the design of effective algorithms for the system as it offers information on the sensitivity of the system to various workload attributes. By providing a validation suite for performance trends, it can guide the selection of appropriate values and tuning of important algorithmic parameters. Characterization of uniprocessor systems have been undertaken in [103] using a low-level machine architecture model and in [110] using a higher-level Abstract Fortran Machine model.

The performance characterization of a multiprocessor system introduces a number of new considerations due to the presence of interactions between concurrently executing processes. Inter-process communication, synchronization and contention for shared resources are the primary sources of interference that influence a concurrently executing process. Therefore, in addition to describing the static behavior of a single processor in isolation, multiprocessor performance characterization must also incorporate some mechanism to represent the dynamic execution behavior of multiple processors in the presence of these interactions. Further, the magnitude of the interference encountered is a function of not only the number of processors but also the parallel program structure and behavioral characteristics.

The well-known *Amdahl's Law* [4] is one of the earliest attempts to address the fundamental issue of parallel program performance. He qualitatively described the gross features of a typical performance spectrum arising in supercomputers. He considered the overall performance of a machine that has two modes of computing (one relatively slow, the other relatively fast) as a function of the time spent in each mode.

Ware [122] quantified the idea in the following model of multiprocessor performance:

$$\text{Speedup} = \frac{t_s + t_p}{t_s + t_p/p} \tag{1.1}$$

where t_s is the amount of time spent on serial parts of a program, t_p is the amount of time spent on parts of the program that can be executed in parallel, and p is the number of processors used. The numerator in Eq. 1.1 denotes the execution time on a single processor whereas the denominator denotes the execution time on p processors. Buzbee [25] has pointed out that this model neglects the effect of multiprocessor synchronization overhead. To correct this inadequacy, he proposed the additional term $\sigma(p)$ in the parallel execution time, which is usually a monotonically increasing function. However, he did not suggest any method for quantifying $\sigma(p)$. Gustafson [54] has recently demonstrated that the assumptions underlying Amdahl's Law are inappropriate for the current approach to ensemble parallelism and has reformulated the law. Gelenbe [48, 49] has given a set of formulae that provide insight into the effective speedup of parallel programs by taking into account the capacity of a program to use its parallel structure effectively.

A three parameter $(r_\infty, n_{1/2}, s_{1/2})$ description, introduced by Hockney [61, 63], characterizes the performance of vector multiprocessors in terms of its vector startup overhead and multiple instruction stream synchronization overhead. The parameter r_∞ is the asymptotic rate of the vector operation for large vectors, $n_{1/2}$ is the vector length at which half the asymptotic rate is achieved, and $s_{1/2}$ is the amount of useful arithmetic that could have been done during the time taken for synchronization. These three parameters were measured experimentally on a 2-CPU CRAY X-MP machine in [62].

All the above models ignore the dynamic effects of communication and synchronization on parallel program execution. More recently, Zhang [127] has presented a timing model based on a modified Ware model that incorporates the various shared-memory multiprocessor program execution effects into the sequential time component t_s of Eq. 1.1. He calibrated t_s and t_p using experimental measurements on some matrix

computations. Although this study demonstrates the various multiprocessor effects, it does not offer any method to deduce system behavior under other workloads. Analytical models for predicting multiprocessor performance on iterative algorithms in terms of the speed of the processor, memory and the interconnection network have been developed in [121, 28]. Statistical models for synchronous parallel algorithms have also been proposed in [84]. But these models do not include the effect of memory contention as a result of access patterns and mutual exclusion synchronization effects.

An experimental characterization technique for multiprocessor memory system behavior was developed by Gallivan *et al.* [45] using a set of "load/store" kernels to define memory access patterns. This method was used to study the relation of the Alliant FX/8 vector instruction set to its memory hierarchy. Although this technique is very effective for observing the dynamic behavior of concurrent memory access streams, it is limited in scope and does not address the other sources of performance degradation on a multiprocessor. Experimental study of memory access contention has also been reported in [24]. Numerous comparative studies of multicomputer/supercomputer performance on specific application programs exist in the literature [34, 82, 57, 32]. These studies, although interesting to read, frequently provide only anecdotal information.

Using standard benchmarks to evaluate machine performance is a widely used practice. Considerable effort has been expended to develop benchmark suites, as described in Section1.2, that are considered to reflect real workloads [69]. Although benchmarking is an excellent vehicle for "performance evaluation" (as defined earlier), there are a number of limitations to using it as an approach to "performance characterization":

- Each benchmark is itself a mixture of characteristics, and doesn't relate to a specific aspect of machine performance.
- They provide no insight as to which components of a given program workload have the potential of being the bottlenecks and to what extent.

From the standpoint of the person engaged in the performance measurement activ-

ity, the use of a standard benchmark program suffers from one significant limitation—the lack of control over the benchmark characteristics. Selecting any standard benchmark as the basis for performance evaluation automatically establishes an associated program workload that is built into the benchmark structure. Hence, it is not possible to experiment with changing individual parameters in the workload that affect performance so as to determine optimal settings for such parameters for a given architecture/application combination. Such selective characterization of performance along controlled performance dimensions is integral to the design and implementation of better algorithms. Upon identifying the most important parameters that have significant influence on system performance, we need to develop a simple model to understand and a method to quantify the incremental effect of each of the parameters on performance when they are observed separately. The method should also provide means for observing how different parameters interact. Based on these results, we can identify critical parameters and recognize performance bottlenecks.

Essentially, what is needed is a performance evaluation and characterization methodology that includes the following functional components:

- A flexible benchmark workload generator that can be tailored to highlight the performance of a system along selected dimensions.
- A measurement framework that can incrementally capture and quantify both the static and dynamic aspects of program behavior along the selected performance dimensions.
- A system characterization method that uses the measured quantities in a global timing model to help predict performance trends.

In this research, we address the above problem and present a new approach to selective performance evaluation and characterization for multiprocessor systems.

1.4 Objective and Scope of Research

The goal of this research is to explore the use of algorithm characteristics as an abstraction that can help in designing benchmark sets that measure the effect of those parameters which most significantly influence multiprocessor performance. The final objective of such an exercise is to evolve a "system characterization" of the system under test that can effectively guide the design of efficient algorithms. The impact of changing algorithmic parameters on algorithm performance can be predicted and validated using the characterization data suite. Knowledge of expected performance degradation of a multiprocessor program in advance, before actually writing it, helps support an efficient design and implementation methodology. The insight thus gained helps users (and eventually compilers) understand why a given computation runs slowly and how to redesign the algorithms to optimize performance.

We have focused on evaluations at the algorithm level, which means that the types of conclusions that may be drawn relates to how well the structure of an algorithm matches the capabilities of an architecture. Thus, the evaluations at this level do not address the question of how the algorithm fits into a complete task. However, algorithms are more often readily available than complete tasks, and solutions to complete applications are often constructed from a library of key algorithms. It will therefore be of interest to understand what is being learned from architecture evaluations performed at the algorithm level. Our approach will be to propose abstractions by which this sort of evaluation can be facilitated. The objective is to make more systematic the way in which benchmark sets are selected. The approach proposed in this research is intended to complement applications-based benchmarking as a method for performance evaluation.

We have restricted the scope of our studies to multiprocessors supporting a *shared address space*. The hardware architecture of the machine need not furnish a common shared-memory. The underlying programming model is assumed to be one using *shared-variables*. This programming model is widely used and is evident from the overwhelming dominance of shared-memory multiprocessors for general purpose parallel

programming both in the commercial and academic sectors. Examples of commercial multiprocessors include the Encore Multimax, the Sequent Balance and Symmetry, and the BBN GP1000 and TC2000 systems; among research prototypes are the NYU Ultracomputer [51], the IBM RP3 [104], and the Illinois Cedar [44] machines. Furthermore, a considerable effort in software development is designed to provide the illusion of shared memory on multicomputers [26, 20, 79, 78, 108, 40, 22]. By restricting our attention to a given class, we filter out some of the strong differences, allowing ourselves to understand the performance within a class more precisely.

The execution time of a task on a multiprocessor may be nondeterministic on account of queueing delays due to contention for shared resources such as memory or communication channels, or to data-dependent computation times. Variations in execution times generally result in synchronization delays where one task has to await the completion of other tasks. These synchronization delays are inherent in the structure of the algorithm and limit the potential speedup of the parallel algorithm over a serial algorithm. We distinguish between *implicit* and *explicit* synchronizations. Implicit synchronization is caused by the contention for shared resources (shared memory, critical sections). Algorithms exhibiting only implicit synchronizations have been called asynchronous [71]. Explicit synchronization mechanisms are normally used to enforce precedence relations in synchronized algorithms. This thesis specifically addresses the effect of implicit synchronizations in parallel algorithm execution.

Communication cost, synchronization overhead and the contention for shared resources are recognized as the main sources of overhead present in multiple-processor systems. The performance of a parallel program using shared-variables and exhibiting only implicit synchronizations is strongly influenced along three major dimensions: the distribution of shared-data over the memory hierarchy and the concurrent *memory reference patterns* to access them, *mutually exclusive access* to shared-data to preserve consistency, and the presence of global *synchronization barriers.* Along each performance dimension, the behavior of a given program is a complex function of a number of architectural as well as application parameters. It is important to be able to isolate and determine the effects of each of these components on overall system per-

formance. By increasing our ability to measure the pieces, combine their effects, and relate their contributions to architectural and algorithmic characteristics, we enhance our ability to model and predict performance in complex systems.

As discussed earlier, standard benchmark programs are not suitable for performing the task of system characterization since we cannot isolate the effects of each of the three performance factors when executing the benchmark workload. Although they provide good indication of the overall system performance, a user does not have any control on the benchmark characteristics. We need a flexible benchmark workload generator and a systematic measurement methodology to capture the incremental contribution of each performance factor to the total parallel execution overhead. We

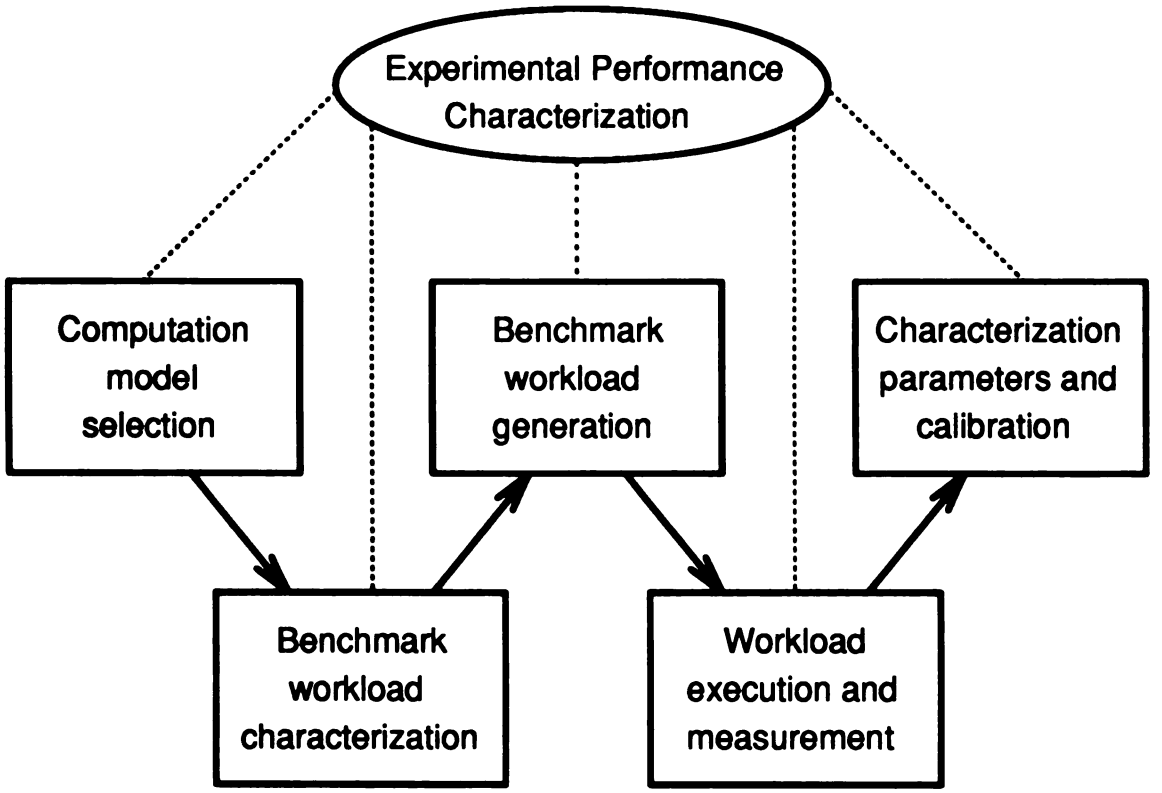


Figure 1.2. Steps in the experimental performance characterization method

have developed an experimental performance characterization method based on the

construction of synthetic executable workloads. These workloads have the advantage that they can be made parametric and hence flexible in representing workload characteristics. Although they have the disadvantage of possible lack of realism at the applications level, they can be made to reflect the algorithm characteristics quite accurately [121]. Our characterization technique consists of five distinct steps (Figure 1.2):

1. **Parallel computation model selection.**

   To be universally applicable, the system characterization measurements must be based on a uniform model of execution so that the results of an experiment can be related to previous and future experiments. We consider a class of structured multi-phase [91] iterative algorithms as our basis for characterizing multiprocessor performance. Many engineering and scientific applications are most frequently characterized as being highly iterative and adhere to this *phase-and-transition* model.

2. **Benchmark workload characterization.**

   The benchmark workload characterizer uses a hierarchical approach to construct a variety of artificial workloads of interest using the parameters that most influence the behavior of concurrent program execution. At the lowest level, it uses a single grain of computation, called a *unit grain*, as the unit of parallel workload specification. The unit grains are assembled into the multi-phase parallel computation structure at a higher level thus incorporating the algorithmic component into the workload.

3. **Benchmark workload generation.**

   Assigning appropriate values to the attributes used to characterize a unit grain creates synthetic workloads that are used as benchmarks for the characterization process. Values assigned to the attributes may be constant quantities thus creating invariant deterministic unit grain characteristics, or the attributes may be treated as random variables of known probabilistic distributions thereby producing stochastic unit grain behavior. The unit grain attributes are varied

in a controlled fashion to create *parameter families* that systematically traverse the input parameter space.

4. **Workload execution and performance measurement.**
   A family of workload emulation programs has been developed that use the workload specification to mimic the execution behavior of an actual program that would demonstrate the same workload characteristics. Three sets of such emulation programs have been designed corresponding to the three major performance dimensions described earlier; each measures and quantifies the performance degradation resulting from overheads along its associated dimension.

   - *Memory Access Degradation (MAD) kernels* measure the overheads resulting from memory contention while accessing shared-data.
   - *Synchronization Access Degradation (SAD) kernels* measure the overheads resulting from synchronization operations and mutually exclusive access to shared-data.
   - *Barrier Access Degradation (BAD) kernels* measure the overheads resulting from the presence of synchronization barriers in parallel program execution.

   The measurement framework allows for observation of interference between both *homogenous* and *heterogenous* concurrent processes.

5. **Performance characterization parameters.**
   Two performance metrics, *unit grain efficiency* and *interference*, are introduced to measure the *relative* performance of a workload as the number of parallel processes increases. The performance of a given workload as the number of processors vary is completely described by a set of six parameters — three constants $(R_\infty, c_{1/2}, f_{1/2})$ and three functions $(\psi_m(N), \psi_s(N), \psi_b(N))$.

The usefulness of this methodology lies in its ability to selectively assess and characterize a shared-memory multiprocessor using synthetic benchmarks whose characteristics can be controlled by the person performing the evaluation. This is of

great practical importance to computer manufacturers as well as system and application programmers alike. For researchers, it is an important exercise if lessons are to be learned, particularly in the area of scalability. From a computer manufacturer's viewpoint, its use lies in evaluating a new system as soon as a prototype is running, using the measured values to determine performance bottlenecks and making architectural refinements. The measurements also provide performance data for competitive bidding. The goal for system and application programmers, on the other hand, is understanding how the characteristics of an algorithm relate to the constraints of an architecture. Further, most compilers for multiprocessor systems available today which feature automatic vectorization and/or parallelization incorporate explicitly or implicitly an *econometric model* of the processor for which they are targeted [112]. This model is used to evaluate when particular optimization choices should be invoked. The performance data obtained can be used to calibrate such models accurately.

1.5 Thesis Outline

The rest of this thesis is organized as follows. In Chapter 2, an overview of the organization of shared-memory multiprocessors is presented. The factors that limit parallelism on these machines along the three performance dimensions discussed in the previous section are examined in detail. A summary of the architectural features of the multiprocessor systems used for our experiments is also provided. The performance characterization framework and its components are described in Chapter 3. In Chapter 4, the use of the MAD kernels to evaluate the performance of shared-memory accesses and quantify the losses in parallelism due to memory contention is addressed. In Chapter 6, the study of performance losses due to inter-process synchronization using the SAD kernels is presented. The measurement of the impact of synchronization barriers on parallel execution performance using the BAD kernels is described in Chapter 6. Finally, Chapter 7 summarizes the major contributions of this research and provides directions for future research.

CHAPTER 2

BACKGROUND

Although multiprocessor systems do hold the potential for solving problems with vast computational requirements, it is by no means obvious that a particular algorithm will perform well on a given machine. Access to common memory is one of the key factors in the performance of shared-memory multiprocessors. Large-scale multiprocessors can encounter significant performance degradation due to a number of factors related to memory sharing. Contention for shared resources such as interconnection networks, memory modules and shared-variable locations, serialization of execution due to mutually exclusive access to shared-writable data, and synchronization barriers are all factors that limit the performance of parallel program execution on shared-memory multiprocessors. It is important to understand how these performance penalties depend on the various architecture and algorithm design parameters.

In this chapter we review the shared memory organization, the primary factors limiting parallel execution performance and the techniques used to reduce the impact of contention in shared-memory multiprocessors. A summary of the architectural features of the multiprocessor systems used in our experiments is also given.

2.1 Multiprocessor Memory Organization

In multiprocessors with global shared memory, parallel memory modules must be used to provide sufficient bandwidth for the processors. Furthermore, a suitable interconnection network must establish the effective sharing of the memory modules

between the processors. Memory access *latency* can become a critical problem in large systems when the distance between parts of the system is such that the time required for data transfer is excessive. In small-scale multiprocessors such as the Alliant FX/8 [102] and the Sequent Symmetry [80], all processors are attached to a single bus which connects them to a global memory. Memory latency is reduced by associating private caches with each processor. Cache coherence is enforced by protocols relying on a fast broadcast mechanism.

For large-scale multiprocessors, a single bus fails as an effective interconnect as its fixed bandwidth limits its scalability. Technology limitations make it too expensive to provide full hardware connectivity between all processors and memory modules. Therefore, large-scale multiprocessors are built with intermediate connectivity using interconnects such as multi-stage interconnection networks as in the BBN TC2000 [15] and the IBM RP3 [104] systems, point-to-point connections as in the Intel Touchstone DELTA [64], and hierarchical interconnects as in the Kendall Square Systems KSR1 [66] and the DASH multiprocessor [77]. Since broadcasting for cache coherence on these interconnects is cumbersome, larger systems either provide cache consistency using a directory-based protocol (as in the DASH project) or provide caching in a restricted fashion under software control (as in the BBN TC2000 and IBM RP3 systems).

One solution to the memory latency problem on large multiprocessors is to build a system in which not all memories are equally distant from all processors, thus allowing data of special interest to a particular processor to be profitably located near it. Distributing a variety of memories around the system (*hierarchical organization*) can minimize the average data access time and thereby improve system performance. Other approaches to reducing memory latency where the interconnection network itself is a component of the memory hierarchy have been explored in [90]. The number of shared memory modules has a great impact on memory contention. If the number of memory modules is less than the number of processors, memory contention will occur if all processors issue a shared memory request at the same time.

Multiprocessor systems differ in their design as to how the shared memory mod-

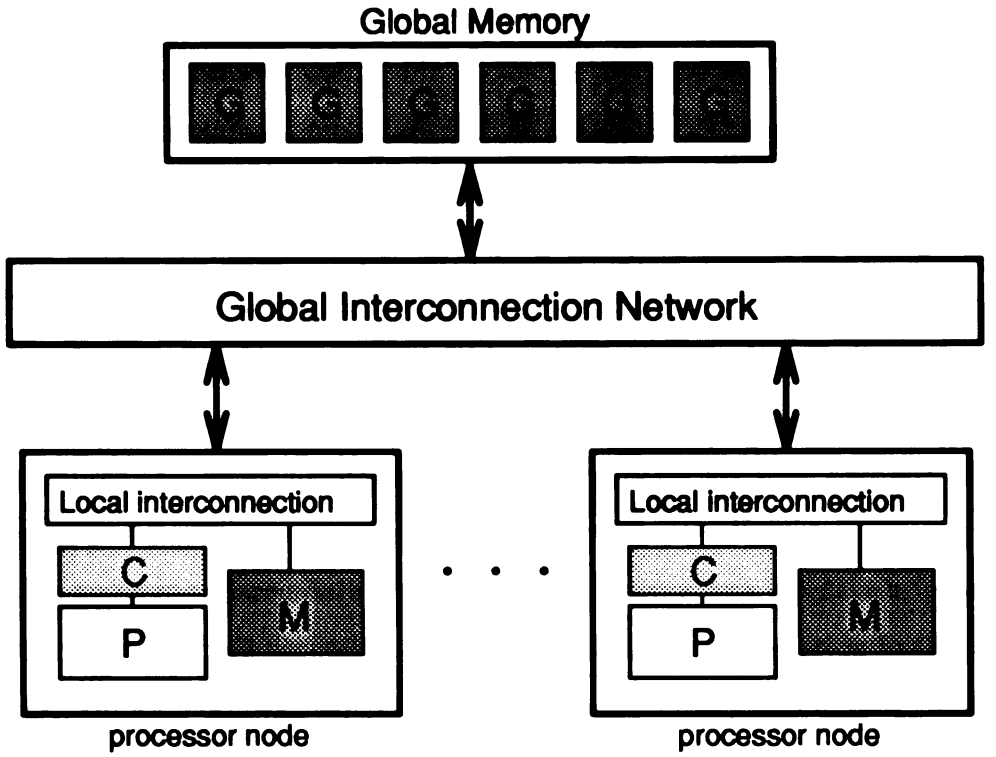


Figure 2.1. Organization of memory hierarchy in shared-memory multiprocessors

ules are distributed over the memory hierarchy and how they provide hardware and software so that each processor sees a single address space in this hierarchy. Typically, a memory module is either *local*, meaning that it attaches to one processor, or *global*, meaning that it is only accessible by sending requests through the interconnection network. A request from a processor to its local memory does not cause any network traffic. This kind of memory organization is depicted in Figure 2.1. Such a memory organization is motivated by price/performance reasons similar to the cache/main memory hierarchy prevalent in uniprocessors. Note that in small-scale multiprocessors, the memory local to each processor includes only its cache. Each processor node in Figure 2.1 could also be a cluster of nodes with each node having access to some local memory and cluster-global memory in addition to the system-global memory modules represented by G. The Illinois Cedar [44] system, for instance, has such a cluster organization.

Local memory modules (such as M in Fig. 2.1) can be divided further into *shared*

and *private* modules. Shared memory modules are accessible by all processors, whereas a private memory module is accessible only by the processor to which it is attached. Global memory modules (such as G in Fig. 2.1) are implicitly shared and private memory modules are implicitly local. Consequently, there are three types of memory modules: local/private, local/shared, and global/shared. For example, the BBN TC2000 has only local/shared memory modules, and the IBM RP3 can be set up to have both local/private and local/shared memory modules.

Private memory provides a means for reducing network traffic. Allocating private data structures to private memory means that requests for such data structures do not cause network traffic and occur with minimum latency. However, memory latency incurred in accessing shared data structures depends on where the data is located with respect to the requesting processor. The location-dependent variation in the latencies of shared-memory modules results in a non-uniform memory access time thus making the issue of data distribution over the memory hierarchy a critical consideration for performance. As an example, a remote memory access takes four times longer than a local memory access on the BBN TC2000. Architectures such as the KSR1 support dynamic migration of data to the point of demand.

2.2 Limitations to Parallelism

Communication, synchronization and contention for shared resources are recognized as the three primary sources of overhead in parallel program execution on multiple-processor systems. We consider only multi-phase asynchronous parallel algorithms constructed using the shared-memory programming model in this research. Since all communication between concurrent processes in such algorithms occurs through globally shared variables, the memory conflicts encountered in accessing the shared variables is critical to overall performance. The amount of memory contention, and the consequent performance degradation, depends not only upon the characteristics of the memory hierarchy and the distribution of shared-data over the hierarchy, but also on the characteristics of the data reference patterns and the interaction between

the two.

To ensure the consistency of concurrent updates to shared data, conflicting accesses must be protected within *critical sections*. In other words, a fundamental form of synchronization necessary for asynchronous parallel algorithms is *mutual exclusion*. Another form of synchronization commonly used by multi-phase algorithms to demarcate the individual phases is *barrier synchronization*. Barriers enforce the arrival of all participating processes at a point before any one of them can proceed further. Both critical sections and barriers induce sequential components into the execution profile of an asynchronous parallel algorithm thus resulting in loss of parallelism. Moreover, inefficient implementations of the mutual exclusion and barrier operations (in hardware or software) could also lead to performance degradation. In the following paragraphs, we discuss how each of these potential sources of loss in parallelism is affected by design choices and what techniques have been developed to minimize their impact.

2.2.1 Memory Access Contention

Distance is one reason for memory reference delays. A second reason is contention, which consists of both network contention and memory contention. Multiprocessor applications usually require shared data areas appropriately distributed over the memory modules. Memory conflicts may occur when two or more processors attempt to gain access to a shared resource along the processor–to–memory path simultaneously. The effect of memory conflicts, referred to as *memory interference*, may decrease the execution rate of the processors. We describe below the factors that cause memory access conflicts and contribute to performance penalties.

Contention for processor–to–memory path

Processors executing concurrently contend not only for memory, but also for the path to memory. There are three principal ways of interconnecting processors and memory modules: bus, crossbar and multistage network. The *bus*, by its very nature, provides a common route shared between all processors to gain access to a global

memory space, thus enforcing sequential-access to the shared memory. The high-performance bus systems of today (*e.g.*, the Sequent System Bus [114], the Encore Nanobus [38]) employ a *split-transaction protocol* whereby multiple memory access requests are pipelined onto the bus before a single memory transaction proceeds to completion. As a result, the bus capacity can be fully utilized if the memory reference pattern can constantly keep the bus busy. The data transfer capacity between processors and memories is determined by the bandwidth of the bus, and is therefore constant. This limits the number of processors that can be usefully incorporated into such a system, and hence fixes an upper limit on performance. *Crossbars* scale up linearly in terms of performance, but their major shortcoming is the cost and size which is proportional to the square of the number of interconnected components.

Multistage networks provide multiple parallel paths to memory, but processors may contend for paths through the network. Such paths consist of switches at each stage of the network and links between switches in different stages. The switches of a multistage network may be blocking or nonblocking. *Blocking switches* have buffers to hold messages waiting while some other message is using the switch. *Nonblocking switches* reject all but one of the conflicting requests so that no queues are formed. This distinction has important implications for system performance as shown by simulation studies conducted as part of the IBM RP3 project. These studies [105] show that small nonuniformities in memory reference patterns can lead to severe degradation of overall system performance due to some memory modules becoming hot. Such nonuniform patterns resulted in a phenomenon called *tree saturation*, where traffic to the hot memories queued up in the switches and interfered with all other traffic. This saturation effect propagates back through the network, as shown in Figure 2.2, fanning outward from the hot memory module in a tree-like fashion. This problem can be partially resolved by combining networks [105, 75]. On the other hand, nonblocking switches, by rejecting all but one of the conflicting memory requests, avoid the phenomenon of tree saturation [119] so that degraded performance is experienced only by the processors that access the hot memories. Thus, the design and implementation of the interconnection network have a profound effect on the processor–to–memory–path

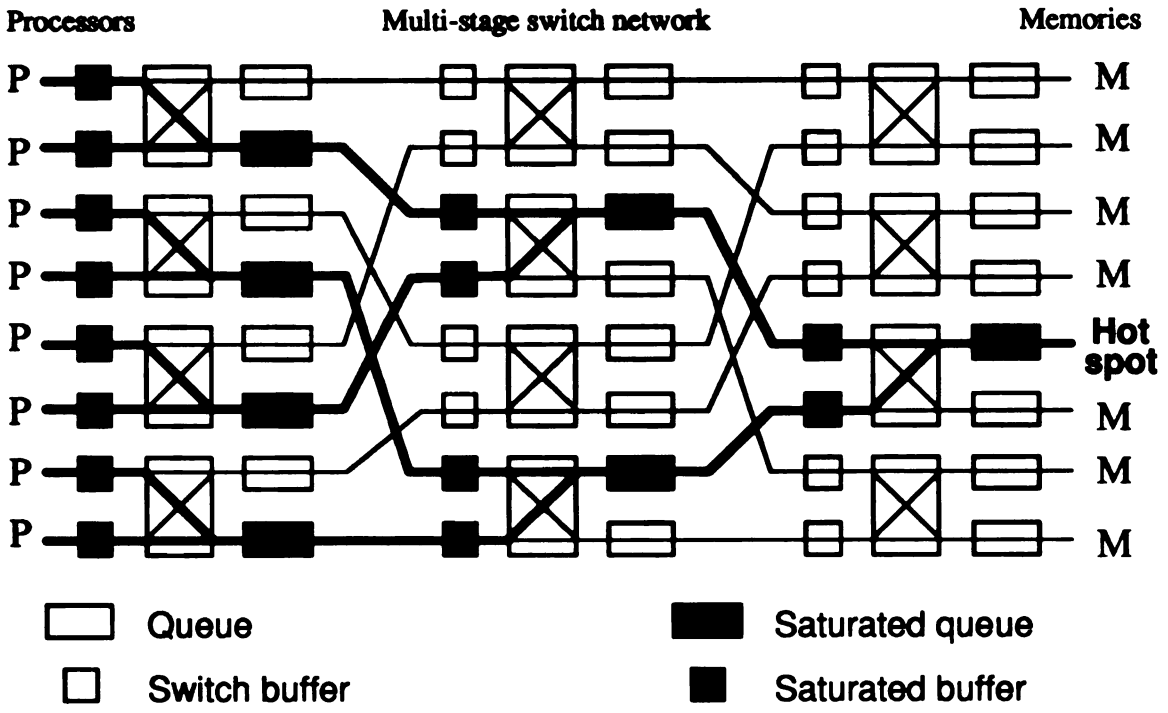


Figure 2.2. Tree saturation as a result of *hot spot* accesses over a multistage interconnection network

delay experienced by memory accesses in large-scale multiprocessor systems.

Contention for memory module

Even if the interconnection network meets the bandwidth requirements of the processor-memory traffic, memory contention can still cause a problem if the processor-memory traffic concentrates on a small number of memory modules. Therefore, it is essential to consider how data structures are allocated to the shared memory modules. A memory module can service only one request at a time (assuming multiport memories are not used). This causes multiple simultaneous requests to the same memory module to be serialized resulting in loss of parallelism.

Memory address interleaving is a technique [73, 99] used to reduce the effective memory access time and, hence, increase memory bandwidth by attempting to distribute the concurrent memory request streams from multiple processors evenly across multiple memory banks. Two broad classes of interleaving schemes used are *modulo-*

interleaving and *random*-interleaving. In the former scheme, a word with physical address β is mapped to the bank address β (modulo M), where M is the number of memory modules (assuming a single bank per memory module) and is called the *degree* of interleaving. The address format and address distribution for such fine interleaving is shown in Figure 2.3(a).

Usually processors access memory in the form of blocks (or cache lines if processor cache is present). With fine interleaving, the transfer of each word requires the establishment of its own path from the processor to each memory module. In order to maximize the amount of data transferred from a memory module during an access, many of the multiprocessors today increase the *granularity* of interleaving from a single word to several consecutive words, say g (equal to the cache line size). Thus, every successive block of g words are now interleaved across the memory modules instead of a single word, as shown in Figure 2.3(b). If multiple banks are used per memory module, then addresses can be finely interleaved across the banks of a memory module and consecutively among modules (shown in Figure 2.3(c)). Each module can now transfer a block at a time thus increasing memory bandwidth. This kind of coarse interleaving works quite well when most reference sequences address successively numbered memory modules.

If the ratio between the time required to issue a request and the time required to service a request is r, then a factor of $f = min(M, r)$ increase in memory performance is obtained by allowing all the memory modules to operate in parallel. However, when the sequence of addresses does not access successive memory modules, as is the case in many scientific applications, then the gain in performance can be significantly less. The *random interleaving techniques* [107, 124, 74, 98] attempt to overcome this drawback by employing various methods to randomize the consecutive memory addresses issued by a processor. Most of these approaches involve logical operations on carefully selected address bits to effect the randomization.

Address skewing is yet another technique [55] that has been used in improving the memory bandwidth in applications involving arrays. In these methods, the starting positions in memory of successive rows of an array are displaced relative to one another

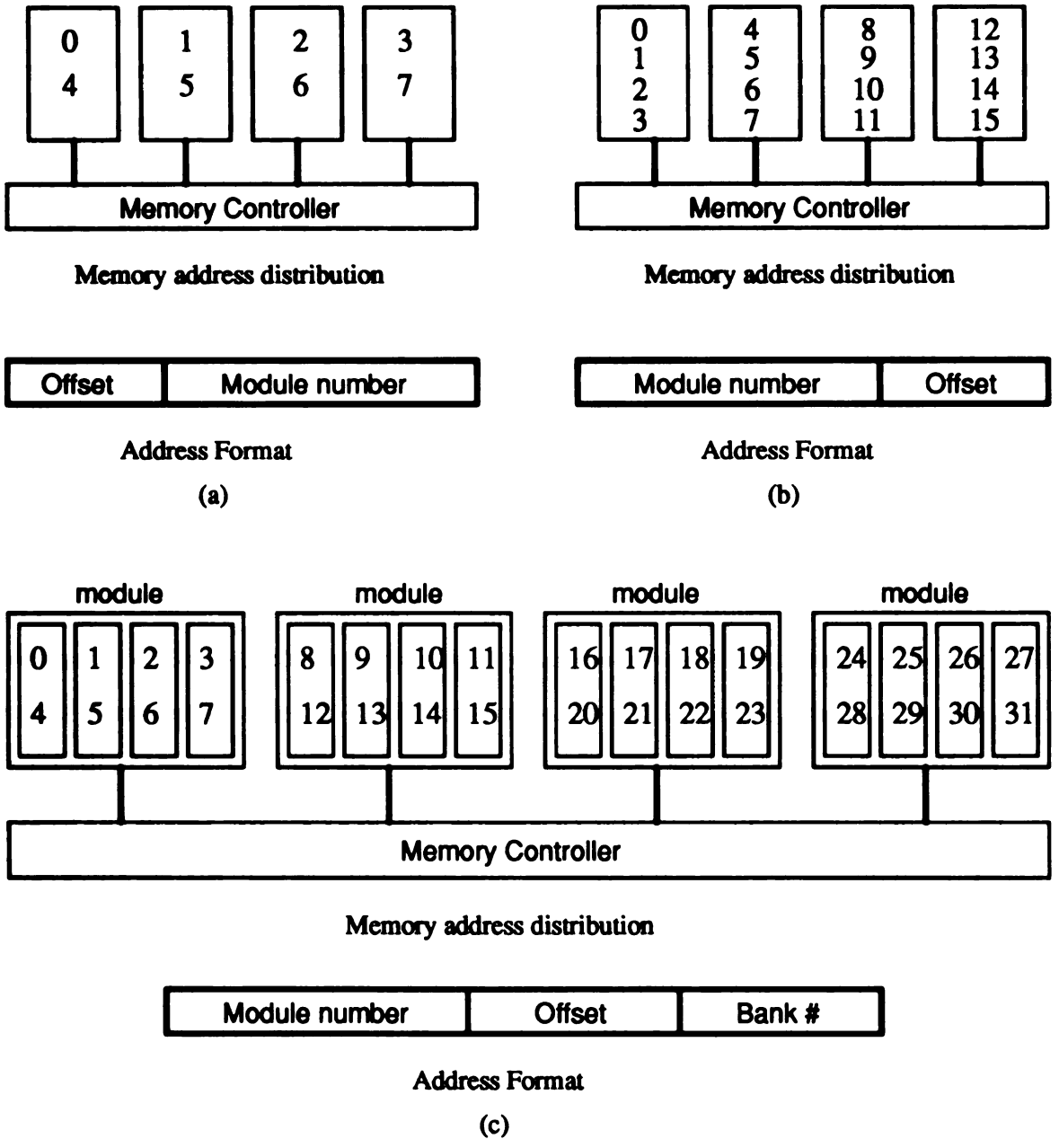


Figure 2.3. Memory address interleaving techniques: (a) Fine interleaving with sequential assignment across modules (one bank per module); (b) Coarse interleaving with sequential assignment within module (one bank per module); (c) Mixed scheme with fine interleaving among banks of a module and coarse interleaving among modules (multiple banks per module)

by a fixed distance (skewed) such that several subvectors of the array can be accessed without conflict.

The ratio of the memory cycle time to the interconnection network cycle time is a critical factor in the service demand placed on the memory modules. Address and data buffers are, sometimes, used locally in each memory module to hold pending memory requests thus eliminating them from the contention for the interconnection network. Buffering is also used so that transient nonuniformities which occur in some access patterns do not degrade performance [55]. The depth of buffering provided at each module determines the extent to which memory access performance suffers.

Contention for memory location

In multiprocessor with a single address space, there are situations in which many processors must access a single memory location. One typical example is the case of memory locations holding *synchronization variables* like mutual exclusion locks which are used to ensure exclusive access to a shared data or to a critical section of code. If many processors need to access the resource controlled by the lock at about the same time, there is a high degree of contention for the memory location of the lock due to the highly repetitive access to the lock caused by busy-wait spinning [6, 52]. Depending on the implementations used for the busy-waiting mechanism, differing degrees of memory and interconnection network contention may result introducing performance bottlenecks that become markedly more pronounced as architectures scale.

Both hardware and software techniques have been explored to reduce the impact of such contentions for a shared memory location. Proposals for multistage interconnection networks that combine concurrent accesses to the same memory location [109, 104, 51], software combining techniques [126], multistage networks that have special synchronization variables embedded in each stage of the network [65], and special-purpose cache hardware to maintain a queue of processors waiting for the same lock [76, 50] are among the many hardware solutions suggested for this problem. Software solutions for scalable synchronization in shared-memory multiprocessors us-

ing carefully designed data structures and their appropriate placement in the memory hierarchy have also been implemented and tested [6, 52, 89].

Maintenance of data coherence

Memory incoherence (inconsistent copies of data) is another serious problem in multiprocessors with global memory and memory (or cache) that is local to each cluster or processor. The coherence problem is caused by the existence of replicated copies of a shared memory block at different levels of the shared memory hierarchy. This can introduce inconsistency if special arrangements are not provided to detect when one copy is modified. Note that inconsistency can occur only for shared, writable memory blocks. Read-only or nonshared data can always be safely cached or replicated without precautions. Many multiprocessor systems (such as the Encore Multimax, the Sequent Symmetry) provide additional hardware to automatically enforce data coherency among multiple shared copies of a datum.

Stenstrõm [118] has surveyed a number of proposed cache coherence schemes for maintaining data consistency in shared-memory multiprocessors. The two most popular approaches are the *snoopy cache protocols* that rely on a broadcast interconnection medium such as a bus, and *directory-based protocols* [3] used on other general interconnection networks. The data coherency mechanism may add an overhead component to the access time of shared-writable data thus degrading performance.

2.2.2 Spin Locks and Mutual Exclusion

Synchronization is a fundamental concept in parallel programming because it provides the basis for cooperation of tasks in a program and controls access to shared resources. In the shared-variable programming model, processors communicate by sharing data structures. Since each processor has equal access to the shared memory, some method for ensuring mutual exclusion—the logically atomic execution of operations (*critical section*) on a shared data structure—is required. Consistency of the data structure is guaranteed by serializing the operations done on it. Synchronization constructs can be divided into two classes: *blocking* constructs that deschedule wait-

ing processes, relinquishing the processor to do other work, and *busy-wait* constructs in which processors repeatedly test shared variables to determine when they may proceed. Busy-wait synchronization is preferred over scheduler-based blocking when scheduling overhead exceeds expected wait time, when processor resources are not needed for other tasks, or when blocking is inappropriate or impossible (for example in the kernel of an operating system).

One of the most widely used busy-wait synchronization constructs is a *spin lock*. Spin locks provide a means for achieving mutual exclusion and are a basic building block for synchronization constructs with richer semantics, such as semaphores and monitors. Spin locks are ubiquitously used in the implementation of parallel operating systems and application programs. Since pure software mutual exclusion is expensive [72], virtually all shared-memory multiprocessors provide some form of hardware support for making mutually exclusive accesses to shared data structures. This support usually consists of instructions that atomically read and then write a single memory location. Atomic instructions serve two purposes. First, if the operations on the shared data are simple enough, they can be encapsulated into single atomic instructions [59]. Mutual exclusion is directly guaranteed in hardware. If a number of processors simultaneously attempt to update the same location, each waits its turn without returning control back to software. Second, if the critical section requires more than one instruction, then a spin lock is used to guard the critical section and atomic instructions are used to arbitrate between simultaneous attempts to acquire the lock. If the lock is found busy, then waiting is done in software.

Spin locks are generally employed to protect very small critical sections, and may be executed an enormous number of times in the course of a computation. Unfortunately, simple approaches to busy-waiting tend to produce large amounts of memory and interconnection network contention thus exhibiting very poor performance. With an ill-designed spin lock, spinning processors can slow other processors doing useful work including the one holding the lock by consuming communication bandwidth. As a consequence, the overhead of busy-waiting synchronization, referred to as *lock interference*, is widely regarded as a serious performance problem.

When many processors busy-wait on a single synchronization variable, they create a *hot spot* that is the target of a disproportionate share of the network traffic. Pfister and Norton [105] showed that the presence of hot spots can severely degrade performance for all traffic in multistage interconnection networks, not just traffic due to synchronizing processors. Agarwal and Cherian [2] have investigated the impact of synchronization on overall program performance by simulations of benchmarks on a cache coherent multiprocessor. Their study indicates that memory references due to synchronization cause cache line invalidations much more often than non-synchronization references. In order to alleviate these performance concerns, modern multiprocessors generally incorporate sophisticated atomic operations into their architectures, permitting faster and more efficient implementation of synchronization primitives. Particularly common are various **Fetch-And-Φ** operations [67] which atomically read, modify, and write a memory location. **Fetch-And-Φ** operations include **Test-And-Set**, **Fetch-And-Store** (swap), **Fetch-And-Add**, and **Compare-And-Swap**.

More recently, there have been proposals for multistage interconnection networks that combine concurrent accesses to the same memory location [51, 104, 109], multistage networks that have special synchronization variables embedded in each stage of the network [65], and special-purpose cache hardware to maintain a queue of processes waiting for the same lock [50, 76]. The principal purpose of these hardware primitives is to reduce the impact of busy waiting. Several software techniques developed of late have also achieved a similar result. By distributing the synchronization data structures over the shared-memory hierarchy appropriately, it can be ensured that each processor spins only on locally accessible locations, locations that are not the target of spinning references by any other processor. All software approaches to efficient spin lock implementation have adopted this philosophy in one form or the other. The implication of these software techniques is that efficient synchronization algorithms can be constructed in *software* for shared-memory multiprocessors of arbitrary size. Special-purpose synchronization hardware can offer only a small constant factor of additional performance for mutual exclusion [89].

We describe briefly several spin lock implementations that have been proposed.

Each lock implementation uses a hardware supported atomic operation to invoke mutual exclusive access to the shared lock variable. However, they differ in the frequency with which the shared lock variable is polled, and the amount of network traffic generated as a result of busy-waiting.

Simple Locks

The simplest mutual exclusion lock employs a polling loop to access a shared variable that indicates whether the lock is held. Based on what operation is used to poll the shared lock variable there are two possible implementations:

- **Spin on Test-And-Set:** Each processor to repeatedly executes a `Test-And-Set` instruction until it succeeds at acquiring the lock. The principal shortcoming of the test-and-set lock is contention for the shared lock variable. Each waiting processor accesses the single shared flag as frequently as possible, using relatively expensive read-modify-write (`Test-And-Set`) instructions. The result is degraded performance not only of the memory bank in which the lock resides but also of the processor-memory interconnection network.

- **Spin on Read (Test-And-Test-And-Set):** `Fetch-And-`Φ instructions can be particularly expensive on cache-coherent multiprocessors since each execution of such an instruction may cause many remote invalidations. To reduce this overhead, waiting processors poll with read requests during the time that the lock is held. As a result, spinning is done in the cache without consuming bus or network cycles. Once the lock becomes available, some fraction of the waiting processors detect that the lock is free and perform a test-and-set operation of which exactly one attempt succeeds.

Collision Avoidance Locks

The primary factor responsible for the poor performance of the simple lock approaches is the high degree of collisions among concurrent lock acquisition attempts. Thus, if each waiting process delays an amount of time before rechecking and attempting to

obtain the lock, then the number of unsuccessful **Test-And-Set** instructions and the resulting reads by other waiting processes can be reduced. There are two possible variations:

- **Delay-after-release:** This variation waits for the lock to be released before delaying. If some other processor acquires the lock during this delay, then the processor can resume spinning; if not, then the processor can try the test-and-set, with a greater likelihood that the lock will be acquired. Polling for the lock release is only practical for systems with per-processor coherent caches. On other systems, processors would consume communication bandwidth if they were to spin reading memory while waiting for the lock to be released.

- **Delay-between-reference:** An alternative approach is to insert a delay between successive polls of the lock. This can be used on architectures without coherent caches or with invalidation-based coherence to limit the communication bandwidth consumed by the spinning processors.

The mean delay can be set statically or dynamically using exponential backoff techniques (similar to the Ethernet exponential backoff for CSMA networks) to adapt to varying conditions.

Ticket Locks

Ticket locks reduce the number of **Fetch-And-Φ** operations to exactly one per lock acquisition. They ensure FIFO service by granting the lock to processors in the same order in which they first requested it. The lock consists of two counters, one containing the number of requests to acquire the lock, and the other the number of times the lock has been released. A processor acquires the lock by performing a **Fetch-And-Increment** operation on the request counter and waiting until the result (its *ticket*) is equal to the value of the release counter. Contention due to polls of the release counter can be reduced by introducing a delay on each processor between consecutive probes of the counter. In this case, however, exponential backoff is clearly a bad idea. Since processors acquire the lock in FIFO order, overshoot in backoff by

the first processor in line will delay all others as well, causing them to backoff even farther. Experiments conducted by Mellor-Crummey and Scott [89] suggest that a reasonable delay can be set proportional to the difference between a newly-obtained ticket and the current value of the release counter (proportional backoff).

Tournament Locks

Another approach to reducing contention for a single shared lock variable is to have a tree of locks of radix b and height h. The tree forms a tournament wherein winners of leaf lock contests become contestants at the next level. The winner of the root lock has permission to enter the critical section protected by the tree of locks. Each process uses its process identity to choose a random path from the root to a leaf lock. The process may contend only for locks along that path. While every process may contend for the root lock, the number of processes eligible to contend for a lock decreases by the radix of the tree at each level (b^h) as we proceed towards the leaves. Thus, contention at the leaf locks can be made arbitrarily small as the number of leaves approaches the number of processes.

Queuing Locks

In a queue lock, each arriving processor enqueues itself and then spins on a separate flag. When the processor finishes with the critical section, it dequeues itself and nudges the next processor in the queue. This permits the hand-off of the lock to be free of contention. The trick is for each processor to use an atomic operation to obtain the address of a location on which to spin. This class of locks is characterized by FIFO ordering of lock acquisitions and, if the spin location of each processor is selected properly, then, a constant bound on the number of network transactions per lock acquisition.

The best implementation varies somewhat among architectures. With distributed-write cache coherence, processors can all spin on a single counter. To release the lock, a processor simply writes its sequence number into the counter. Each processor's cache is updated, directly notifying the next processor in line with a single network

transaction. With invalidation-based coherence, each processor should wait on a flag in a different cache block. Only two bus or network transactions (an invalidation and a read miss) are needed to signal the next processor in line. Similarly, on a multistage network without coherent caches, each flag should be placed in a separate memory module.

Based on the data structure chosen for the queue of spinning processors, the queuing locks can be classified as array-based or list-based.

- Array-based queuing locks
- List-based queuing locks

Anderson [6] has developed an array-based method of queuing busy-waiting processors in shared memory that requires only a single atomic operation per execution of the critical section. The queue is implemented as a circular array of flags on which busy-waiting processors can spin. Each arriving processor does an atomic **Fetch-And-Increment** to obtain a unique sequence number, which determines a location in the array (**flags**) on which it can spin thus enqueuing itself. When a processor finishes with the lock, it taps the processor with the next highest sequence number; that processor now owns the lock. Since processors are sequenced, no atomic read-modify-write instruction is needed to pass control of the lock. A similar array-based queuing lock has also been proposed by Granuke and Thakkar [52].

A queuing lock wherein the queue of spinning processors is structured as a linked-list was devised by Mellor-Crummey and Scott [89]. Their technique works equally well, requiring a constant number of network transactions per lock acquisition, on machines with and without coherent caches. It requires an atomic **Fetch-And-Store** (swap) instruction and benefits from the availability of the **Compare-And-Swap** instruction. Without **Compare-And-Swap**, the guarantee of FIFO ordering of lock acquisitions is lost introducing the theoretical possibility of starvation, although the lock acquisitions are likely to remain very nearly FIFO in practice.

2.2.3 Synchronization Barriers

In addition to the spin-lock, barrier synchronization is the other most important mechanism for coordinating parallel processes. A barrier defines a logical point in the control flow of an algorithm at which all processes must arrive before any is allowed to proceed further. Barriers are commonly employed when an algorithm consists of several distinct stages, each of which has internal parallelism but which must be performed in strict sequence without overlap. A barrier is clearly one of the most deleterious forms of synchronization, since it requires in effect that every process communicate with every other process. Additionally, since all processes must wait at the barrier until the last arrives, the effects of fluctuations in process execution time or imperfect load balancing are maximized.

A processor typically performs the following three steps at a barrier:

1. Marks itself as present at the barrier (*entry phase*).

2. Waits for all other participating processors to arrive at the barrier.

3. After all participating processors have arrived, it proceeds past the barrier (*exit phase*).

Many algorithms exist for performing barrier synchronization in software [81, 21, 58]. Careful implementation of some of these algorithms are found to scale well to large-scale multiprocessors without the contention for synchronization operations, referred to as *barrier interference*, becoming a significant problem [89]. Barrier algorithms can be distinguished [8] by three features: the *depth* of the barrier (linear or logarithmic), the barrier *scheduling mechanism* (static or dynamic), and the type of *exit phase* (symmetric entry and exit phases, or broadcast exit).

Linear barriers are most commonly implemented using centralized counters to keep track of the number of processors that have arrived at the barrier. Each processor incurs a fixed amount of overhead accessing the shared counter, so the total overhead of such barriers is linear in the number of processors. *Logarithmic* barriers include the software combining tree barrier [126], the butterfly barrier [21] and the

dissemination barrier [58]. In the butterfly and dissemination barriers, synchronizing P processors is accomplished in $\lceil \log_2 P \rceil$ stages of $\lceil P/2 \rceil$ two-processor synchronizations each. In a tree barrier, groups of processors synchronize with each other, and one processor out of each such group goes on to synchronize with the next higher level group, and so on. Although in a logarithmic barrier each processor performs $O(\log P)$ synchronization operations (versus $O(1)$ for the linear barrier), these synchronizations can be overlapped in machines with parallel processor–memory networks, resulting in total barrier overhead that is only logarithmic in the number of processors. In *bus-based* machines, linear barriers are more efficient than logarithmic barriers because fewer total bus accesses need to be performed (assuming the bus is the limiting factor on performance).

In *statically* scheduled barriers, processors update synchronization variables in an order predefined at compile or load time, whereas in *dynamically* scheduled barriers, processors proceed in the order that they arrive at the barrier. Therefore, dynamically scheduled barriers require either explicit software locks (such as `Test-And-Set`), or more complex atomic read-modify-write operations such as `Fetch-And-Add`. Statically scheduled barriers do not incur the overhead of software locks, but also cannot take advantage of the "skew" in processor arrival times where some processors can start synchronizing early.

In the entry phase of a barrier, processors report their arrival by updating some shared state information. In the exit phase, processors exit the barrier after determining that all other processors have arrived. Separate entry and exit phases are required if the barrier is to be reused, in order to properly reinitialize the synchronization variables. In barriers with symmetric entry and exit phases, similar operations are used in both phases. In barriers with broadcast exit, the last processor to complete the entry phase broadcasts this information to all other processors. Barriers with broadcast exit are more efficient than symmetric barriers because they require fewer memory operations on shared variables. However, they also require more local storage at each processor.

Many research efforts have also focused on hardware barrier synchronization tech-

niques on the premises that a $O(\log P)$ growth in synchronization delay of software approaches prevents the exploitation of fine-grain parallelism. The Burroughs Corp. proposal for the Flow Model Processor (FMP) [83] included the first detailed description of a hardware implementation of barrier using the equivalent of a massive "AND" gate. Another scheme developed in [106] consists of a hardware module with bit-addressable registers $R(i)$, $(i = 1, 2, ..., P)$, one associated with each of P processors, an enable switch, logic to test for all zeroes (all processors have reached the barrier), and a barrier register BR. The "fuzzy" barrier scheme of [53], also supported in hardware, is basically a delayed barrier firing mechanism where the actual wait may occur several instructions after a processor indicates that it has encountered a barrier. In all these schemes, all physical processors in the machine were considered to be involved in each barrier synchronization. More recently, the "barrier MIMD architectures" proposed in [100] support an arbitrary subset of the processors to be barrier synchronized.

2.3 Target System Architectures

We have used two shared-memory multiprocessors with very different shared memory organizations, namely a 26-node Sequent Symmetry S81 and a 45-node BBN TC2000, to illustrate our experimental characterization methodology. Two older generation systems, a 24-node Sequent Balance 21000 and a 96-node BBN GP1000, were also used in some of our early experiments. These systems were selected more because of the convenience of access than anything else. Of these, the BBN GP1000 system is installed at Michigan State University whereas the remaining systems are installed at the Advanced Computing Research Facility of the Argonne National Laboratory. In this section, we briefly describe and compare the salient features of these system architectures that are relevant to the interpretation of the experimental results obtained.

The Sequent Symmetry S81 [114] is a bus-based shared-memory multiprocessor, belonging to the Uniform Memory Access (UMA) class, and, containing from 2 to 30

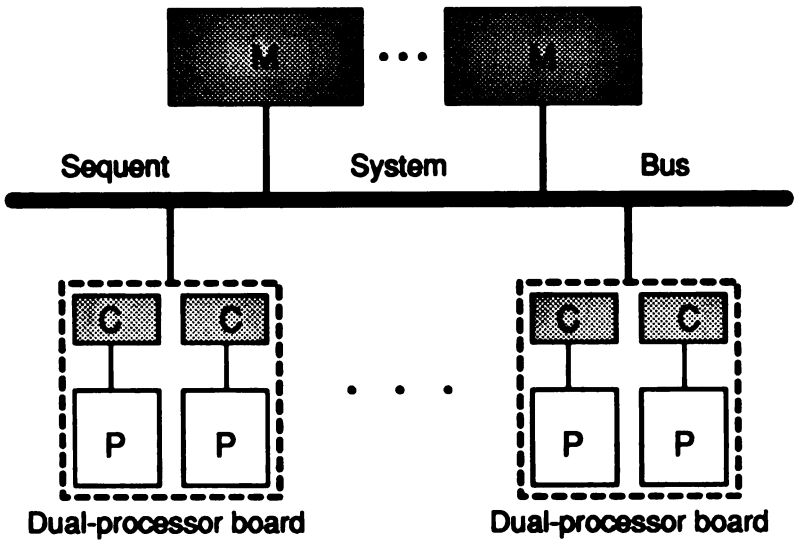


Figure 2.4. Sequent Symmetry system architecture

processors packaged on dual-processor boards and upto 240 Mbytes of main memory. Each processor subsystem consists of an Intel 80386/80387 CPU/FPU combination and a 64-Kbyte 2-way set-associative cache. Cache coherence is enforced by using a *write-invalidate copy-back* caching policy on a cache line that is 16 bytes long. It can contain upto six memory modules, each consisting of a memory controller board and 8 or 16 Mbytes of memory. It can also, optionally, contain a memory expansion board with 24 Mbytes of memory on it. When the system contains a pair of equal-sized memory subsystems, alternate 32-byte address blocks are interleaved between the two modules. The Sequent System Bus (SSB) forms the heart of the system's global interconnection network. All the processor and memory subsystems along with other device interfaces are directly connected to the bus. The system bus operates at 10 MHz (1 cycle = 100 ns). It can carry 64 bits of data with address and data being time multiplexed on the bus. Multiple bus transactions are pipelined so that the bus throughput can be maximized. The bus is rated at a peak data transfer rate of 53.3 Mbytes/second. The Symmetry provides an atomic `Fetch-And-Store` operation but no `Compare-And-Swap` operation.

The BBN TC2000 [14] is a large shared-memory multiprocessor that belongs to

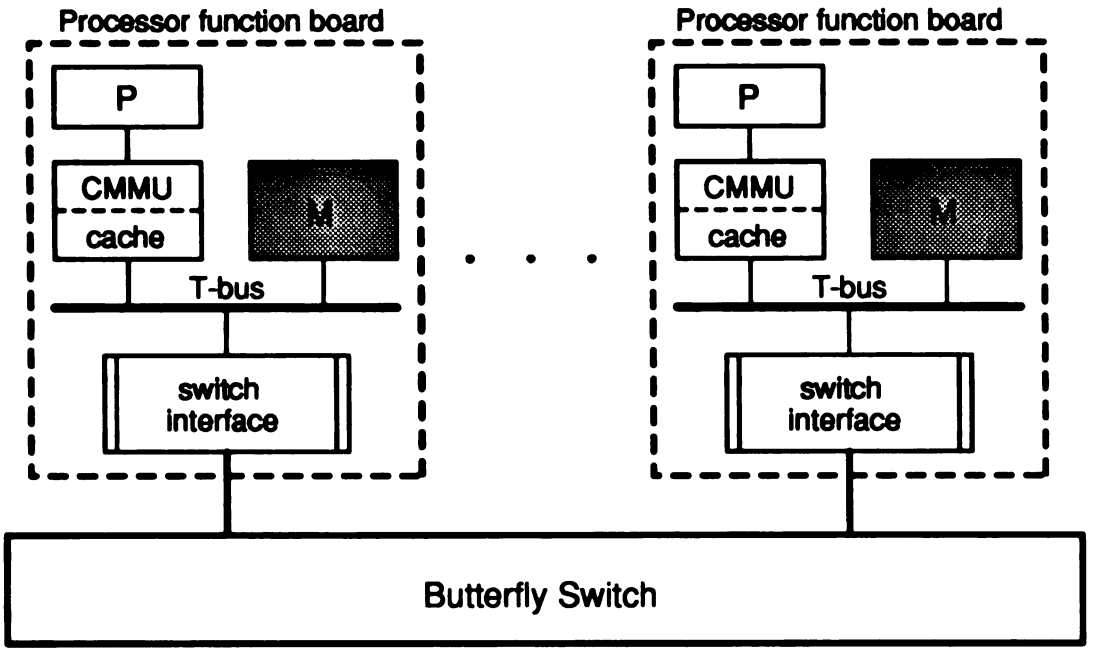


Figure 2.5. BBN TC2000 system architecture

the Non-Uniform Memory Access (NUMA) class due to the distributed nature of its shared memory modules. It is built using Motorola 88100 RISC processors. These processors reside on a function board that also has a MC88200 Cache and Memory Management Unit (CMMU) 16 Kbytes each of instruction and data cache, 4 or 16 Mbytes of memory, and a switch interface. The function boards are interconnected by a multistage switching network so that they can access each other's address space. The network consists of 8x8 crossbar bidirectional switches arranged in a $\log_8 N$-column butterfly interconnection pattern, where N is the number of processor nodes. Every remote memory reference is sent out over the switching network, but local memory access is performed over a direct path bypassing the network. This causes a remote memory access to incur a higher latency in comparison to a local access.

A route specifies a complete and exact path through the switch. A reply to a given request is also returned along the same path. If a conflict occurs at any stage in the network, due to the access paths of two or more concurrent requests crossing each other, then the switch selects exactly one request at random to proceed and

rejects all others, which must be retried at a later time. Thus, the switches are non-blocking in nature. Alternate paths between function boards may exist depending on configuration size. Use of these alternate paths helps reduce congestion within the switch. However, on the TC2000, the switch interface selects a given route for an initial message before its first transmission into the switch, and does not change that route during any retries of the message. Different paths may be used by separate initial messages, but not by separate retries. There were two alternate paths available on the system to which we had access. All shared-data, by default, are not cached on the TC2000. A user can choose to selectively cache shared-data and manage its coherency explicitly. The TC2000 provides a **Fetch-And-Store** operation via the **xmem** instruction.

The earlier generation Sequent Balance 21000 system [113] is also a bus-based global memory multiprocessor, much as the Symmetry, based on the NS32000 series microprocessor. The bus supports multiple pipelined memory requests. Cache consistency is maintained by *write-through with invalidation* scheme. An additional feature present on the Balance, that was later removed from the Symmetry, is a dedicated lock memory (called Atomic Lock Memory or ALM) connected to the bus that supports process synchronization primitives. However, the overhead of accessing the ALM is sufficiently high that applications on the Balance may use spin-locking based on **xchg**, the exchange-with-memory instruction supplied by the processor [7].

The BBN GP1000, a generation older than the TC2000, is also a NUMA multiprocessor [13] based on the Butterfly switch multistage interconnection network. It incorporates upto 256 processor nodes each containing a Motorola 68020 CPU, 4 Mbytes of memory and a MC68851 paged memory management unit for virtual memory processing. The network is composed of 4 stages of 4x4-switches. Memory accesses over these switches is handled much the same way as on the TC2000.

The architectural features of the systems on which our experiments were conducted are summarized in Table 2.1.

Table 2.1. Summary of target system architectures

Feature	**Sequent Symmetry**	**BBN TC2000**
No. of Processors	26	45
Processor Type	Intel 80386	Motorola 88100
Clock Cycle Time	62.5 ns	50 ns
Memory Size	32 MB	720 MB (16 MB/proc)
Data Cache Size	64 KB/proc	N/A
Cache Line Size	16 bytes	N/A
Cache Coherence	copy back	N/A
IN Network	Bus	2-col 8x8-switch MIN
Peak IN Bandwidth	53.3 MB/sec	38 MB/sec/channel
Operating System	DYNIX B3.1.2	nX OS release 2.0.6
Timer Resolution	1 μs	1 μs
Feature	**Sequent Balance**	**BBN GP1000**
No. of Processors	24	96
Processor Type	NS32000	Motorola 68020
Memory Size	16 MB	384 MB (4 MB/proc)
Data Cache Size	8 KB/proc	N/A
Cache Line Size	8 bytes	N/A
Cache Coherence	write-through	N/A
IN Network	Bus	4-col 4x4-switch MIN
Peak IN Bandwidth	26.7 MB/sec	32 Mbits/sec/channel
Operating System	DYNIX	Mach 1000
Timer Resolution	1 μs	62.5 μs

2.4 Summary

Efficient access of shared data is the single most important factor in the performance of parallel program execution on shared-memory multiprocessors. The effective memory access latency is determined by the hierarchical organization of the shared memory modules and the distribution of data over this hierarchy. Contention for the network, memory modules and memory locations can all increase the memory reference delay. Data coherence mechanisms for replicated data can also contribute to increased latency due to additional network traffic generated. The performance of asynchronous parallel algorithms on multiprocessors is also influenced by the use of spin-locks for enforcing mutual exclusion and barriers. Not only do these forms of synchronization introduce a sequential bottleneck, but an inefficient implementation of these primitives can have a significantly detrimental effect on other shared memory accesses.

In this chapter, we have enunciated the various factors that contribute to the performance degradation of asynchronous parallel algorithms on multiprocessors using the shared-variable programming model. The observation and quantification of these overheads is the object of our performance characterization study.

CHAPTER 3

PERFORMANCE CHARACTERIZATION METHODOLOGY

The execution performance of a parallel program using shared-variables depends on *static* characteristics of the underlying algorithm such as computation granularity, computation-to-communication ratio, data reference patterns and fixed synchronization costs. In addition, performance is also influenced by run-time overheads incurred during parallel execution from three primary activities, namely, resource contention during concurrent accesses to shared data, mutually-exclusive access to shared data, and synchronization barriers. This overhead is a function of the *dynamic* run-time behavior of the system. It is added to the execution time in the form of processor latencies and busy waits. As overhead increases, the amount of parallelism that can be exploited decreases. An accurate and complete performance characterization of multiprocessor program execution must take into account not only the static system behavior, but its dynamic behavior as well. Furthermore, it is important to be able to isolate and measure the effect of each component on overall system performance. By increasing our ability to measure the pieces, combine their effects, and relate their contributions to architectural and algorithmic characteristics, we enhance our ability to model and predict performance in complex systems.

In this dissertation, we have developed a *hierarchical* performance characterization technique that relies on experimental calibration. The method is based on the construction of synthetic executable workloads. These workloads have the advantage that they can be made parametric and hence flexible in representing workload characteristics. Our technique consists of five distinct steps as shown in Figure 1.2 of Chapter 1:

- parallel computation model selection,
- benchmark workload characterization,
- benchmark workload generation,
- workload execution and performance measurement, and
- performance characterization.

In this chapter, we describe each of these activities leading to the system characterization objective. The characterization parameters obtained represent the static performance of a machine as well as different aspects of dynamic interaction between the machine architecture and the application structure.

3.1 The Parallel Computation Model

The objective of this thesis is to develop a set of parameters that characterize the static and dynamic performance of a shared-memory multiprocessor, and obtain quantitative measures for these system characterizers in the context of a certain class of algorithms based on the shared-variable computational paradigm. The quantification of the characterization parameters is performed through experimental measurements on the target machine for a selected set of workloads.

To be universally applicable, the system characterization measurements must be based on a uniform model of execution for parallel computations so that the results of an experiment can be related to previous and future experiments. Besides, in the development of a parallel program on a shared memory system, it is natural to

first deal with the software structure of the program and then with the algorithmic parameters that determine computational efficiency (for example, task granularity, distribution of shared data and their access patterns, frequency of synchronization, length of critical sections, *etc.*). We use a hierarchical model to characterize and measure the incremental impact of software structure, hardware resource contention, lock contention and synchronization barriers on the absolute rate of computation as well as the relative computational efficiency.

Parallel algorithms can be classified based on the structure of their task graphs [91]. Experience shows that most parallel algorithms belong to one of only a small number of classes [46]. Examples of classes of task graphs are those representing asynchronous, multilevel partitioned, multiphase, and pipelined parallel algorithms. We use a *phase and transition* model of program execution with a multi-phase task structure as the basis of our system characterization methodology. A parallel computation is viewed as a sequence of computational phases separated by global synchronization points (or *barriers*) (as shown in Figure 3.1). The computation and communication patterns and, hence, the program behavior are well defined within a phase, but may change from one phase to the next. Many scientific and engineering problems adhere to this model in practice. Application examples represented by this computational structure include the parallel PDE solver using the synchronous Jacobi method, parallel FFT, molecular-motion computations, weather prediction models, *etc.*

Each phase is comprised of a collection of asynchronous tasks without any explicit synchronization constraints among them. They may, however, synchronize implicitly as a result of hardware resource contention during shared-data accesses and software resource (such as locking semaphores) contention during mutually exclusive access to critical regions of code. Computations developed according to the popular SPMD (Single Program Multiple Data) parallel programming paradigm fit this task structure well. At a lower level, a task may correspond to one more iterations of a parallel **DOALL** loop construct [70] executing concurrently on a single processor. The iterations of a **DOALL** loop are data independent and, therefore, can be assigned to different processors and executed in any order. Parallelism at a higher level can be exploited

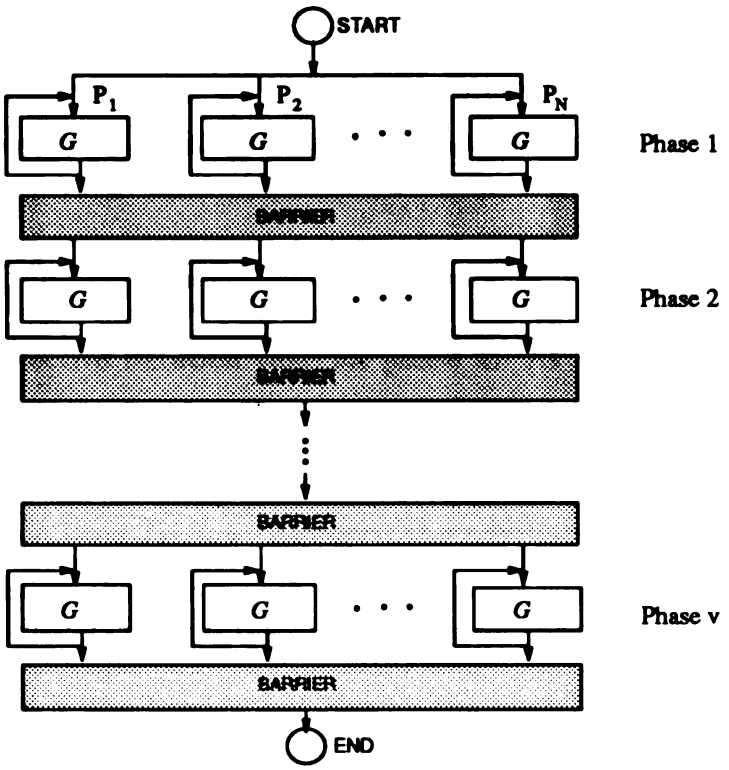


Figure 3.1. Structure of parallel program execution

by *high level spreading* of large-grain tasks.

We focus attention on the class of structured iterative algorithms with multiple phases. Within a phase the computation, shared data access and synchronization patterns are very regular for each iteration. Frequently in these applications, the computation can be uniformly distributed among processors thus assigning equal amount of work with identical characteristics to each processor. Therefore, if we assume that each process performs a series of identical iterations within a phase, then the overall multi-phase performance of the complete application can be extrapolated from measurements performed at the iteration level [92]. Since all iterations are identical, we will measure the performance behavior of a single loop iteration when executing concurrently with other identical iterations.

In Figure 3.1, assume that there are v computational phases. Assume that phase k is comprised of w_k identical iterations on each processor and the number of processors employed (*degree of parallelism*) is N_k. If t_{k_i} is the time it takes to complete one

iteration on processor i during phase k, then the total time to complete w_k iterations on processor i is given by $w_k t_{k_i}$ since all iterations are identical. Hence, the time T_k required to complete phase k of computation is a function of N_k and is given by

$$T_k(N_k) = \max_{1 \le i \le N_k} \{w_k t_{k_i}\} = w_k \cdot \max_{1 \le i \le N_k} \{t_{k_i}\} = w_k t_k$$

where t_k is the effective iteration time for phase k. At the end of each phase, all processes must wait for the slowest process among them before they can continue. The time spent in waiting for the last process to arrive is already accounted for in the phase execution time $T_k(N_k)$. However, the additional time penalty needed to broadcast the event of the arrival of the last process at the barrier contributes to the total execution time. This time, $T_{barr}(N_k)$, depends not only on the number N_k of processors involved in the barrier, but also on the implementation and the method used to busy-wait for the arrival of the last process. If all the sequential components of the parallel program execution, such as creation of parallel processes *etc.*, can be represented by the single time component T_{seq}, then the total execution time T of the computation is given by

$$T = T_{seq} + \sum_{k=1}^{v} (T_k(N_k) + T_{barr}(N_k)) = T_{seq} + \sum_{k=1}^{v} (w_k t_k + T_{barr}(N_k))$$

Using this model, if the per-iteration execution time t_k and the barrier performance T_{barr} can be accurately characterized for a given workload for varying degrees of parallelism N_k, then the overall performance of the computational workload can be estimated.

3.2 Workload Characterization

System characterization (to distinguish it from benchmarking) is a set of experiments that isolate and measure the performance response of a system to controlled workload inputs. These responses describe the system and determine its performance. The accuracy of the system characterization depends closely on the type of work-

loads chosen for selective assessment and how well they represent the measurement objective. Having chosen a multi-phase program structure at the algorithm level, we next concentrate on defining the program characteristics within a phase.

3.2.1 The Unit Grain

Measurement data about the behavior of real workloads on shared memory multiprocessors are scarce (examples are [1], [30], [37] and [12]). Hence a broad but abstract model of workload specification is adopted for system characterization. It allows the exploration of performance over a wide spectrum of assumptions about data sharing, locality of reference, and inter-process synchronizations.

It has been shown [121] that the performance of a parallel system in the short term—during one iteration—for example, can also be used to model long term performance. We model the computation in a single process (or thread of activity), which is part of the parallel workload, as a sequence of loop iterations that may be random or deterministic. Each such loop iteration represents a single grain of computation, called a *unit grain* and denoted as G in Figure 3.1. The sequence of iterations, therefore, represents a string of grains constituting the execution profile of a single processor in a parallel program. The unit grain is the fundamental level at which all performance measurements are taken.

Each unit grain G is further assumed to be composed of exactly three granules: shared-memory access, local computation and synchronization (Figure 3.2). A *shared-memory granule*, denoted as g_m, is concerned with accessing globally shared data needed for the computation. Most often, access to globally shared data within this granule would be in concurrent–read mode, since writes to shared data must be properly guarded within critical sections in order to preserve memory access consistency. In situations where concurrent writes are legitimate and consistency preserving, however, g_m could include writes to shared data. A *local computation granule*, denoted as g_c, represents the portion of the execution grain that performs CPU bound computation using only process private data. We assume that any shared data needed for the computation is first retrieved into a process private area (possibly internal reg-

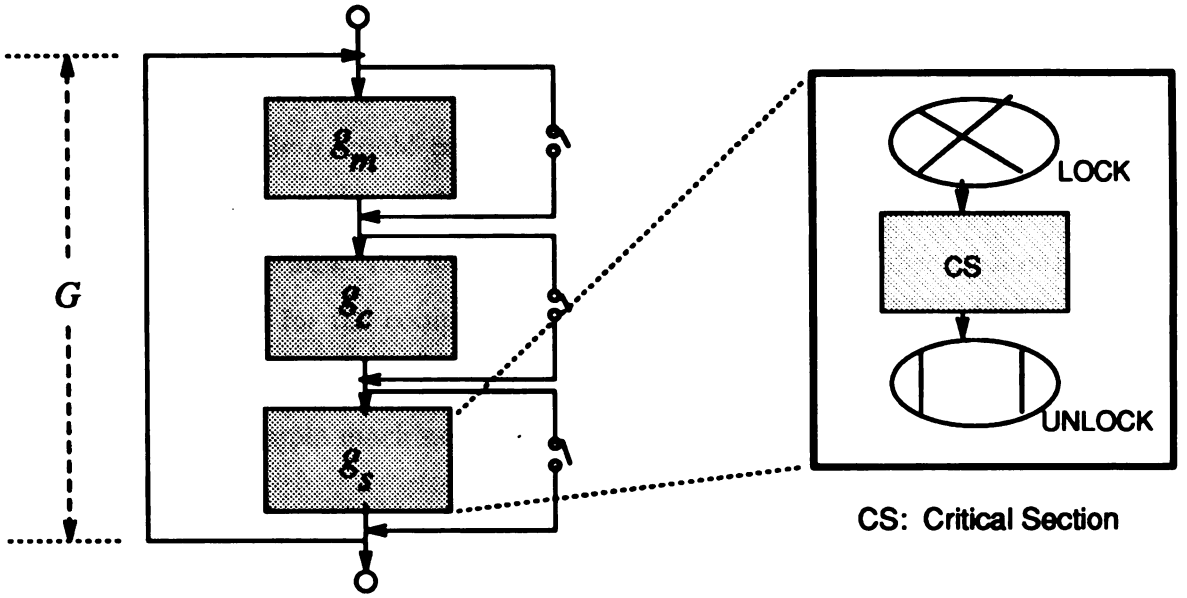


Figure 3.2. Structure of a unit grain

isters or processor cache) before being used. A *synchronization granule*, denoted as g_s, represents inter-processor synchronization in the form of *mutual exclusion* (using locking semaphores) to access critical sections of code wherein updates to write-shared data are performed. It could also represent synchronization operations such as *event post/wait* for synchronous algorithms. This granule imposes an ordering restraint on the otherwise concurrent execution of a multiprocessor application. Using this decomposition, the unit grain G is defined to be a *3-tuple* of granules.

$$G = (g_m, g_c, g_s)$$

A special characterization called *null characterization*, and denoted by $g_i = \phi$, is reserved to indicate that granule g_i is absent from the unit grain. Any component granule in the definition of G can be null, reflected by the alternate bypass paths shown around each granule in Figure 3.2.

We will characterize the unit grain G by choosing an appropriate characterization for each of its component granules. The choice of *attributes* needed to characterize each granule depends upon what aspect of the multiprocessor system performance

is under study and the level of abstraction at which the analysis is to be carried out. For example, if the speed of floating-point operations were of interest, then the computation granule g_c could consist of an appropriate floating-point expression(s), whereas the granule g_m could simply be specified as a memory access frequency, and the granule g_s made null. The hardware execution times of the different floating-point operations selected for the computation in g_c can be normalized to addition time by assigning suitable weights to each type of floating-point operation. An example set of weights for a sample machine are shown in Table 3.1.

Table 3.1. An example of weights assigned to different types of floating-point operations to normalize their execution time to floating-point addition time

Floating-point operation	Normalizing weight
+, −, *	1
/, SQRT	4
EXP, SIN, *etc.*	8
IF (X .REL. Y)	1

Similarly, the absolute performance of synchronization primitive could be measured by using null characterizations for g_m and g_c, while characterizing g_s with the relevant details of the implementation of the synchronization primitive.

3.2.2 Workload Classification

Using the 3–granule decomposition of the unit grain, a single phase of computation in our multi-phase program structure can now be represented as shown in Figure 3.3. Each task (assigned to a separate processor) processes a string of ℓ unit grains before synchronizing at a global barrier. Granule g_c contains the meaningful computations performed by a task and hence represents the operations whose overall rate should be maximized. Based on whether the granules g_m and g_s are present in the unit grain

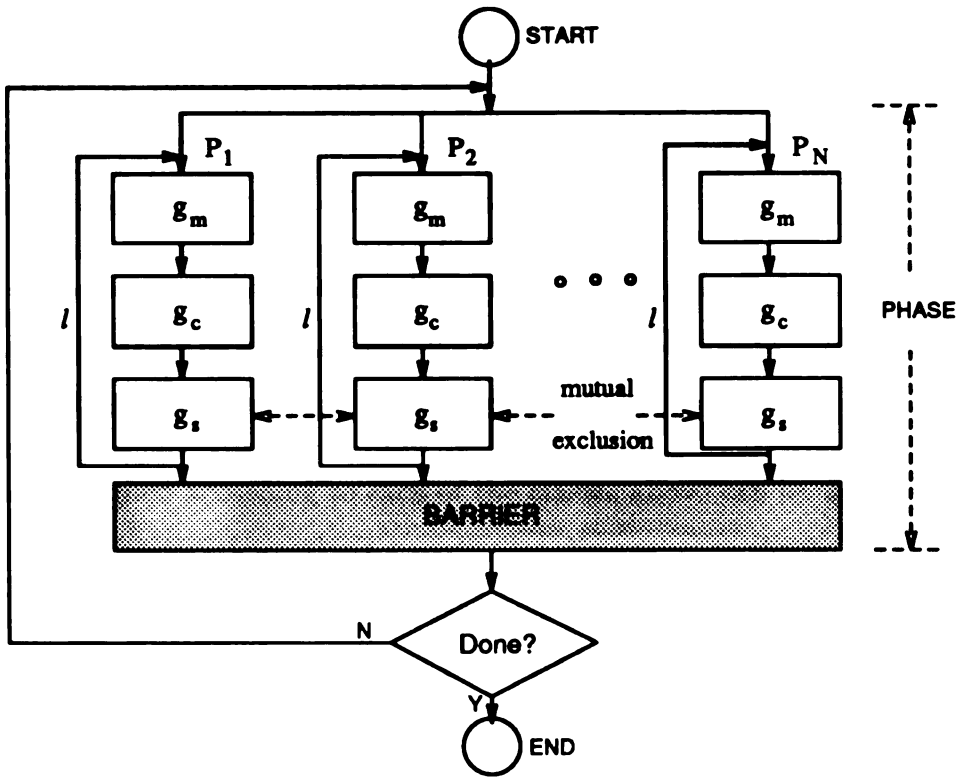


Figure 3.3. Structure of a single computational phase

definition, the range of workloads represented by this technique can be categorized into four broad classes based on the mode of concurrent accesses to shared data.

A. *Embarrassing workloads.* All computation in these workloads is performed within granule g_c with no shared-data access or inter-process synchronizations ($g_m = \phi, g_s = \phi$).

B. *Concurrent-access workloads.* In addition to computation performed within g_c, processes also access shared data concurrently in g_m ($g_m \neq \phi, g_s = \phi$). As an example, processes may perform local computations while accessing shared data in concurrent read-only mode. Concurrent write-sharing is also permissible as long as the write operations performed on the shared data are consistency preserving.

C. *Exclusive-access workloads.* In workloads belonging to this class, processes access shared data only in exclusive mode, *i.e.*, in a mutually exclusive fashion,

inside g_s in addition to performing local computation in g_c ($g_m = \phi, g_s \neq \phi$). There is no concurrent sharing of any global data in this class. Write-sharing of data between processes that requires mutually exclusive updates to ensure data integrity belongs to this workload class.

D. *Dual-mode access workloads.* This is the most general class in that both concurrent sharing as well as exclusive sharing of global data is allowed in addition to local computation ($g_m \neq \phi, g_s \neq \phi$).

Workloads designed according to each of the classes above can be used to either measure a system's performance along a particular dimension or the interactions between different performance dimensions. This provides a means of observing how different factors affecting performance interact. Based on these results, one can identify critical parameters and recognize performance bottlenecks.

3.3 Experimental Framework

The definition of the unit grain provides a unit of workload specification for the computational activity in a single process (or a single thread of control). Our objective is to measure not only the static characteristics of the execution of a specified workload but also the dynamic characteristics that result from the run-time interactions between concurrent processes. In other words, we would like to be able to observe and quantify the loss in performance that results from the interference between concurrently executing grains. The program characteristics of the interfering grains may be identical (homogenous) or non-identical (heterogenous). With this objective in mind, the measurement structure selected for the experimental study of the interference behavior is now described.

3.3.1 Measurement Structure

In order to fulfill our goal of observing the interference between a set of simultaneously executing homogenous or heterogenous grains under varying degrees of parallelism,

we have established an experimental structure for our measurements (see Figure 3.4) that consists of:

- one *test* processor (called P_0),
- a variable number, N, of *competitor* processors (called $P_1, P_2, ..., P_N$), and
- a number, M, of data elements that are shared by the test and competitor processors.

The test processor P_0 executes a unit grain called the *test grain* and denoted by $G_t = (g_m^t, g_c^t, g_s^t)$. Each competitor processor executes a unit grain called the *competitor grain* and denoted by $G_c = (g_m^c, g_c^c, g_s^c)$. Every competitor processor, $P_1, ..., P_N$, executes an identical copy of the competitor grain G_c simultaneously. The number of competitor processors, N, can be varied to control the degree of parallelism and, hence, the extent of interference among the concurrent grains.

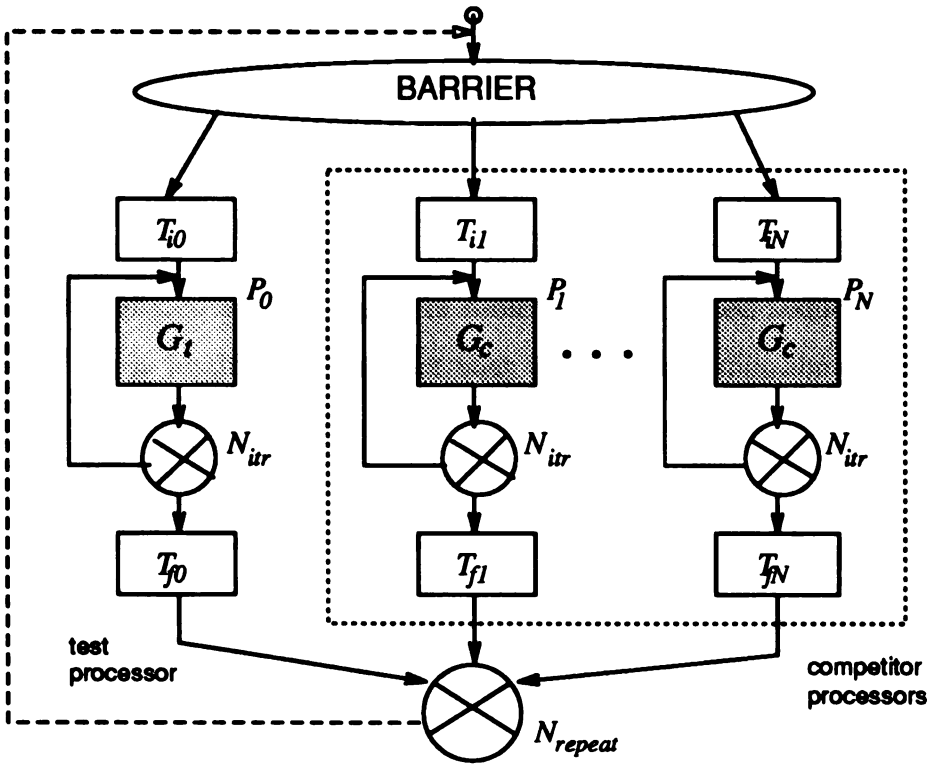


Figure 3.4. Structure of the measurement framework

We also make the following assumptions for all our experimental measurements:

- The number of concurrently competing processes in our framework is less than or equal to the maximum number of available processors N_{max}, *i.e.*, $N+1 \leq N_{max}$.
- A process once created and attached to a processor remains stationary, *i.e.*, process migration is not allowed.
- The execution of a process is *nonpreemptive.*

The first assumption ensures that all the processes in a given workload are simultaneously active on different processors thus participating in shared resource contention resulting in the worst-case runtime overheads. Throughout this thesis, therefore, the terms process and processor are used interchangeably. The second assumption helps eliminate the context-switch overhead that would entail from process migration. The third assumption precludes any unexpected program behavior due to unpredictable process preemptions. Further, all measurements are performed on a quiescent system thus enabling us to ascribe reasons for the observed losses with greater confidence.

The second aspect of the grain interactions that needs to be controlled is the size of the shared-data space within which all the grains interact. This is accomplished by assigning a suitable value to M. The structure of the shared data is assumed to be a one-dimensional array consisting of M elements and distributed over the memory modules in the shared address space in some pre-determined fashion. This view of the shared data is justified by the fact that any higher dimensional data structure will ultimately be translated into a one-dimensional sequence of memory addresses for the purpose of storage. A hot-spot scenario results from setting $M = 1$.

The set of input parameters to the experiment, $\mathcal{I}$, can now be consolidated and written as

$$\mathcal{I} = \{N, M, G_t, G_c\}$$

Note that by setting $G_t = G_c$, we can create a *homogenous* workload; or by using different characterizations for G_t and G_c ($G_t \neq G_c$), we can create a *heterogenous*

workload. Homogenous workloads are used to characterize the loss in processing efficiency ensuing from runtime overheads when multiple identical processes cooperate to achieve a common goal (as in SPMD style computations). On the other hand, heterogenous workloads are used to characterize the interaction between unrelated processes (the test and competitor grains in this case). The interference in the execution of a process of interest (the test grain) due to the simultaneous execution of multiple "non-related" processes (the competitor grains) can be observed. By varying N, the performance degradation under varying degrees of parallelism can be measured.

For a given set $\mathcal{I} = \{N, M, G_t, G_c\}$ of input parameters, the average execution time per unit grain for processor P_k (denoted by $\mathcal{T}_k$) is given by (refer to Figure 3.4)

$$\mathcal{T}_k = \frac{T_{f_k} - T_{i_k}}{N_{itr}}, \quad k = 0, 1, ..., N.$$

The effective unit grain execution time $T_G(N)$ for a concurrent workload with N competitor processes active is recorded for each experiment performed. The value recorded for $T_G(N)$ is different for homogenous and heterogenous workloads given that the purpose behind the two types of workloads is different.

For *homogenous workloads*:

$$T_G(N) = \max_{0 \le k \le N} \{\mathcal{T}_k\}$$

For *heterogenous workloads*:

$$T_G(N) = \mathcal{T}_0$$

With these definitions, it is obvious that a null characterization of the test grain (*i.e.*, $G_t = (\phi, \phi, \phi)$) is meaningless for both types of workloads. However, a null characterization may be used for G_c for heterogenous workloads.

We also define the *uncontested execution time* of a granule g_i, denoted by $T_{g_i}(0)$, as the time required for the unit grain G with g_i as the only non-null component granule in it to complete its execution when executing *alone* on a multiprocessor (no interference from other grains). The uncontested execution time of a unit grain is

the sum of the uncontested execution times of its component granules. Using this definition, and the fact that $T_{g_i=\phi}(0) = 0$, we can write

$$\begin{aligned}
\tau_m &= T_{g_m}(0) \quad \text{when} \quad G = (g_m, \phi, \phi) \\
\tau_c &= T_{g_c}(0) \quad \text{when} \quad G = (\phi, g_c, \phi) \\
\tau_s &= T_{g_s}(0) \quad \text{when} \quad G = (\phi, \phi, g_s) \\
\tau &= T_G(0) \quad \text{when} \quad G = (g_m, g_c, g_s) \\
&= \tau_m + \tau_c + \tau_s
\end{aligned}$$

where τ_m, τ_c, τ_s and τ are the uncontested execution times of g_m, g_c, g_s and G, respectively.

3.3.2 Workload Generation

With a suitable selection of attributes characterizing the unit grain, the workload model parameters contained in $\mathcal{I}$ allow a wide range of workload behaviors to be represented. If all the points in the parameter space of $\mathcal{I}$ were to be tested, it would result in an overwhelming number of experiments. This would not only be extremely time consuming to be practically feasible, but also make it impossible to draw conclusions. Hence, a systematic method is adopted for traversing the input parameter space by the creation of *parameter families* wherein a family of related behaviors is obtained by fixing all but one parameter. The parameters in $\mathcal{I}$ that remain fixed within a parameter family are said to be *anchored.* The changing parameter, say X, within a family is denoted by $\vec{X}$. If attribute y of grain G_i is varied, then the changing parameter is denoted by $\vec{G}_i(y)$.

Using this convention, $\mathcal{I}_1 = \{\vec{N}, M, G_t, G_c\}$, for example, denotes a parameter family wherein M, G_t and G_c are anchored while the number of competitor processes N is varied. Similarly, $\mathcal{I}_2 = \{N, M, \vec{G}_t(s), G_c\}$ denotes a parameter family wherein N, M and G_c are anchored while the attribute s of G_t is varied.

Assigning constant values to each attribute in the characterization of G creates an

instance of the unit grain. The resultant tuple is called a *characterization instance* of G. If a study of the *deterministic* execution behavior of the workload represented by G is desired, then the value assigned to each of the characterization attributes may be interpreted as invariant quantities, resulting in an invariant G from one iteration to another. On the other hand, treating each attribute value to be the mean of a known probabilistic distribution transforms the corresponding attribute into a random variable thus permitting a study of the stochastic execution behavior of the workload. For the probabilistic characterization of input workloads, any input parameter X can be associated with a *spread factor* f (denoted by $X[f]$), $0 \leq f \leq 1$, causing X to become uniformly distributed in the interval $[(1-f)X, (1+f)X]$. In other words, an input parameter specification of the form $X[f]$ is equivalent to X being selected from a uniform distribution over the interval $[(1-f)X, (1+f)X]$.

$$X[f] \equiv U[(1-f)X, (1+f)X]$$

3.4 Performance Characterization Parameters

It has been recognized for years that the single parameter Mflop/s (megaflops) is inadequate to measure the performance of a multiprocessor system, because it takes no account of the communication, synchronization and resource contention overhead inherent in the parallel execution of multiple processes. More recently, a two-parameter $(r_\infty, s_{1/2})$ description has been used [61] to characterize the floating-point performance in MIMD computing that is based on measuring the importance of the overhead of synchronizing multiple instruction streams. The parameter r_∞ denotes the asymptotic floating-point performance as Mflop/s whereas $s_{1/2}$ indicates the amount of useful arithmetic that could have been done during the time taken for synchronization. In a similar spirit, a three-parameter $(r_\infty, n_{1/2}, s_{1/2})$ description of MIMD vector computers [62] has also been used that incorporates, in addition to the synchronization overhead $s_{1/2}$, the vector startup overhead in terms of $n_{1/2}$. However, the parameters used in these characterizations assume that the overheads are constant quantities

thus accounting for only the static overheads encountered. The variation of program performance with the number of processors and the associated dynamic overheads caused by run-time interactions between processes cannot be accurately captured by such static parameters only.

In this dissertation, we develop a hierarchical performance model to describe the performance of the multiphase program structure used as the basis of our studies. Each level in the hierarchy provides a measure of the fraction of total processing power that is lost due to inefficiencies at that level. In doing so, each level furnishes a set of parameters that characterizes the importance of overhead factors that limit performance at that level. The hierarchical performance model integrates the characterization parameters from each level into a composite framework that describes the net performance of a system as its ideal potential performance degraded successively by overheads encountered at each level of the hierarchy.

The lowest level in the hierarchy, the *granule level*, focuses attention on each component granule of the unit grain. The effect of the static distribution of work among the granules on computational performance is captured by the three *static* parameters $(R_\infty, f_{1/2}, c_{1/2})$ measured at this level. Measurements at the next higher level, the *grain level*, quantify the overheads that result from run-time interactions between concurrent instruction streams as a function of the number of interfering processes. The influence of these overheads on overall performance is described by the two *dynamic* parameters $(\psi_m(N), \psi_s(N))$. At the highest level, the *phase level*, the loss in performance due to global synchronization at the end of each phase is observed and quantified using the *dynamic* parameter $\psi_b(N)$.

3.4.1 Static Parameters

The decomposition of the unit grain G into the three component granules (g_m, g_c, g_s) signifies the division of the total work performed within a unit grain into communication, computation and synchronization components. The granule g_c performs all the meaningful computation, whereas the time spent within the granules g_m and g_s represents communication (through shared variables) and synchronization (mutual

exclusion) overheads, respectively. The relative proportion of time spent in each of these granules during execution determines the maximum rate at which the computation in g_c can progress. The static parameters characterize the dependence of a multiprocessor performance on the static overheads inherent in the algorithm design resulting from communication and synchronization.

Assume that the computation performed within a unit grain can be expressed in terms of a number of *basic computation units* (BCUs). A BCU may simply represent a single floating-point operation at one extreme, or it may represent a very large computational block involving many number of floating-point operations at the other extreme. In other words, the amount of computation that a BCU is chosen to represent is a matter of the level of abstraction at which the computational performance is of interest. Stated another way, a single BCU produces a single result of interest and the rate of BCU execution determines the rate at which results are generated. Let the unit grain G contain a total of c BCUs distributed between g_c and g_s. Also, let the unit grain G contain a total of m shared data references distributed between g_m and g_s. The synchronization in g_s is assumed to be mutually-exclusive access to a *critical section* guarded by a pair of *lock/unlock* operations.

For a given workload (*i.e.*, a given characterization instance of G), define

$$
\begin{aligned}
t_m &= \text{the average time per shared data access,} \\
t_c &= \text{the average time per BCU,} \\
t_s &= \text{the average time per synchronization operation.}
\end{aligned}
$$

The value of t_m depends not only on the hardware characteristics of the shared memory, but also on the distribution of shared data over the memory hierarchy and their access patterns imposed by the application algorithm. In the case of UMA (Uniform Memory Access) multiprocessors with no caches, where all memory is global and equidistant from all processors, the shared data access time t_m is equal to the time t_{global} to access a data item in global memory. If per-processor caches are present on a UMA multiprocessor (*e.g.*, Sequent Symmetry), then the shared data access time

is governed by the proportion h of cache hits exhibited by the shared data access pattern. If t_{cache} denotes the time to fetch a data item from the cache and t_{global} to fetch it from the global memory, then t_m is given by

$$t_m = ht_{cache} + (1 - h)t_{global}.$$

In the case of NUMA (Non-uniform Memory Access) multiprocessors, all memory is not equidistant from all processors thus exhibiting different access latencies for different levels in the memory hierarchy. Let t_{local} and t_{remote} respectively denote the times to access a data item from processor local and remote memory modules, and r denote the proportion of shared data accesses that go out to a remote memory module. Assuming that no per-processor cache is present (*e.g.*, IBM RP3), the average access time t_m is given by

$$t_m = rt_{remote} + (1 - r)t_{local}.$$

If per-processor caches are present (*e.g.*, BBN TC2000) and h denotes the proportion of cache hits, then t_m is given by

$$t_m = ht_{cache} + (1 - h)[rt_{remote} + (1 - r)t_{local}].$$

The average shared data access times for the different memory organizations are summarized in Table 3.2.

Table 3.2. Summary of average shared data access time t_m

Memory	*no per-processor cache*	*with per-processor cache*
UMA	t_{global}	$ht_{cache} + (1 - h)t_{global}$
NUMA	$rt_{remote} + (1 - r)t_{local}$	$ht_{cache} + (1 - h)[rt_{remote} + (1 - r)t_{local}]$

The value of t_c depends upon the composition of the BCU. For instance, suppose that the rate of floating-point operations were of interest. Let a single BCU consist of

a total of n arithmetic operations each involving a different number of floating-point operations. If the n operations can be classified into t types such that there are n_i arithmetic operations of type i that require w_i floating-point operations, then the BCU time t_c can be expressed in terms of the time t_{fp} to perform a single floating-point operation as

$$t_c = \sum_{i=1}^{t} n_i w_i t_{fp} \quad \text{where} \quad \sum_{i=1}^{t} n_i = n.$$

The value of t_s is determined by the particular implementation chosen for the locking primitives. If t_{lock} and t_{unlock} represent the latencies of the locking primitives, then

$$t_s = t_{lock} + t_{unlock}.$$

Using the characteristic times t_m, t_c and t_s of a given workload, and the unit grain parameters c and m, the single processor (no interfering processors) execution time $T_G(0)$ of the unit grain G can be expressed as follows.

$$T_G(0) = \tau = ct_c + mt_m + t_s. \tag{3.1}$$

Since a total of c BCUs are computed, we find the average BCU computation rate per processor, $R(0)$, as a function of the grain parameters to be

$$R(0) = \frac{c}{\tau} = \frac{c}{ct_c + mt_m + t_s}. \tag{3.2}$$

With a little rearrangement of the above expression, the average computation rate $R(0)$ can also written as

$$R(0) = \frac{R_\infty}{1 + \frac{f_{1/2}}{f} + \frac{c_{1/2}}{c}} \tag{3.3}$$

where:

$$R_\infty = \frac{1}{t_c}, \quad f_{1/2} = \frac{t_m}{t_c}, \quad c_{1/2} = \frac{t_s}{t_c}, \quad \text{and} \quad f = \frac{c}{m}.$$

The value R_∞ provides a measure of the *asymptotic (i.e., maximum) performance* in BCUs/second per processor. The degradation of performance from this peak is determined by the amount of computational work performed per shared data reference,

here measured by f, the computation granularity c, and the static parameters $f_{1/2}$ and $c_{1/2}$. The *half-performance memory factor*, $f_{1/2}$, measures the memory bottleneck in terms the amount of *lost work* that could have been done during the time of the shared data access, whereas the *half-performance lock factor*, $c_{1/2}$, measures the lost work due to synchronization. Hence, they signify the cost of shared data access and synchronization in a currency that has a known value to the programmer.

The significance of the half-performance factors become apparent if we consider a unit grain G with only one kind of overhead in it. For instance, if the synchronization granule g_s is absent from G, then $\tau_s = 0 \Rightarrow t_s = 0 \Rightarrow c_{1/2} = 0$. This results in the average computation rate $R(0)$ given by

$$R(0) = \frac{R_\infty}{1 + f_{1/2}/f}.$$

It can be seen from the above expression that for $f = f_{1/2}$, half the asymptotic performance R_∞ is obtained. Thus $f_{1/2}$ is the minimum computation–to–communication ratio necessary to achieve half the asymptotic performance.

Similarly, if the shared-memory access granule g_m is absent from G, then $\tau_m = 0 \Rightarrow t_m = 0 \Rightarrow f_{1/2} = 0$. This results in the average computation rate $R(0)$ given by

$$R(0) = \frac{R_\infty}{1 + c_{1/2}/c}.$$

Once again, as before, it can be seen that $c_{1/2}$ is the amount of work in a unit grain that is necessary to achieve half of the asymptotic performance.

We characterize the static performance of a multiprocessor system in terms of the 3-parameter description $(R_\infty, f_{1/2}, c_{1/2})$. The values of R_∞, $f_{1/2}$ and $c_{1/2}$ are likely to depend on hardware and application characteristics as the discussion on t_m, t_c and t_s earlier in this section illustrated. The parameters $(R_\infty, f_{1/2}, c_{1/2})$ have been chosen for system characterization rather than the original timing parameters t_m, t_c and t_s, because they are more directly related to facts about a problem that are known to a computer user. The parameters $f_{1/2}$ and $c_{1/2}$ provide a user with a yardstick with which to compare the computation–to–communication ratios and computation

granularities that occur in his problem, and R_∞ provides a target with which to compare the actual performance of his program.

Eq. 3.3 gives the functional form of the approach of the average computation rate to the maximum R_∞ as the computation granularity c and the computation–to–communication ratio f change. The functional form of this approach to the asymptotic will occur repeatedly in the subsequent discussion of performance, and we define it as the *loss* function

$$loss(x) = \frac{1}{1+x}. \tag{3.4}$$

The expression for the average computation rate in Eq. 3.3 can then be written as

$$R(0) = R_\infty loss(f/f_{1/2} + c/c_{1/2}) \tag{3.5}$$

which shows how the peak performance is degraded by memory bottleneck (first term) and inadequate granularity (second term).

We can now express the uncontested single-processor execution time of the unit grain in terms of the static characterization parameters as

$$T_G(0) = \tau = R_\infty^{-1}(c + mf_{1/2} + c_{1/2}), \tag{3.6}$$

and the individual timing parameters as

$$t_c = R_\infty^{-1}, \quad t_m = R_\infty^{-1} f_{1/2}, \quad t_s = R_\infty^{-1} c_{1/2}.$$

Values of $(R_\infty, f_{1/2}, c_{1/2})$ can be obtained by fitting a set of measurements of τ for different combinations of c and m to Eq. 3.6. As Eq. 3.6 represents an equation in three unknowns, a set of three measurements of τ with linearly-independent combinations of c and m should, in theory, be sufficient to solve for the unknown parameters.

3.4.2 Dynamic Parameters

If the concurrent execution of processes, represented by unit grains, on different processors were ideal (*i.e.*, no mutual interference), then the net computational rate achieved with N competitor processors would be $(N+1)R_\infty$. However, in practice, parallel execution of cooperating processes involves contention for shared resources in hardware (memory modules, interconnection network, *etc.*) and software (shared lock variables). The result is runtime overheads that are dynamic in nature which degrade the asymptotic performance further beyond the inefficiencies introduced by the static parameters ($R_\infty, f_{1/2}, c_{1/2}$). It is important to know the computation cost of these dynamic overheads, because this will influence the way in which a particular program is organized for parallel execution (*i.e.*, how it is parallelized).

The multiphase algorithm structure chosen for our studies is assumed to exhibit asynchronous behavior (*i.e.*, only implicit synchronizations) of parallel processes within a phase and global barrier synchronizations between phases. As discussed earlier, there are three overhead dimensions that exert a critical influence on the performance of such application structures, namely, overhead due to contention for shared data and memory, overhead due to access of mutually-exclusive critical sections, and overhead due to synchronization barriers. The measurement of the *incremental* contribution of each of these factors to the total overhead helps identify critical parameters in the workload and recognize potential performance bottlenecks.

The incremental overheads resulting from memory contention and shared lock contention are characterized by measuring the interference among concurrent grains within a phase. In other words, the performance degradation is observed at the grain level. The incremental overhead due to synchronization barriers is obtained from experimental measurements at the phase level. Each overhead component for a given workload is characterized by an *interference factor* expressed as a function of N, the number of competing processes.

Grain level characterization

Barring the loss in efficiency due to the relative proportions of the granule lengths, the ideal parallel execution performance of a unit grain G in the absence of any external interference is given by its uncontested execution time $T_G(0) = \tau$. If the asynchronous execution of concurrent unit grains within a phase were free of mutual interference, then the execution time per unit grain would still remain as τ. However, this ideal performance is hampered by two factors: memory interference and lock interference. Memory interference results from contention for shared hardware resources along the processor–to–memory path, contention for memory modules and the overheads of maintaining data coherence across the memory hierarchy (*e.g.*, cache coherence). Lock interference results from contention for a shared lock variable and the queuing delay ensuing from enforcing the mutual exclusion semantics.

The total execution time of ℓ unit grains (Figure 3.3) within a phase with N other interfering grains present, $T(N)$, is given by its ideal execution time $T(0) = \ell\tau$ augmented by memory and lock interference overheads. In other words,

$$T(N) = \ell\tau + O_m(N) + O_s(N), \tag{3.7}$$

where $O_m(N)$ and $O_s(N)$ are, respectively, the extra overheads due to memory and lock interference. If the corresponding average overheads per unit grain are denoted by $\hat{O}_m(N) = O_m(N)/\ell$ and $\hat{O}_s(N) = O_s(N)/\ell$, then Eq. 3.7 can be rewritten as

$$T(N) = \ell\tau\left(1 + \frac{\hat{O}_m(N)}{\tau} + \frac{\hat{O}_s(N)}{\tau}\right). \tag{3.8}$$

We define two grain-level dynamic characterization parameters *incremental memory interference* (ψ_m) and *incremental lock interference* (ψ_s) as follows:

$$\psi_m(N) = \frac{\hat{O}_m(N)}{\tau} \quad \text{and} \quad \psi_s(N) = \frac{\hat{O}_s(N)}{\tau} \tag{3.9}$$

The memory interference $\psi_m(N)$ for a given workload varies with N, and depends

upon the distribution of shared data objects over the memory hierarchy and the memory reference patterns. Similarly, the lock interference $\psi_s(N)$ also varies with N, and depends on the implementation of the locking primitives, the frequency of critical section access and the amount of computation performed in between consecutive critical section operations. The total execution time from Eq. 3.8 can be expressed in terms of the dynamic characterization parameters as

$$T(N) = \ell\tau(1 + \psi_m(N) + \psi_s(N)). \tag{3.10}$$

Given that there are c BCUs computed per unit grain and ℓ unit grains executed per processor within a phase, the total number of BCUs computed within a phase is $(N+1)\ell c$ as there are $(N+1)$ processors executing concurrently. Hence, the effective BCU rate with N competitor processes active, $R(N)$, is given by (using Eq. 3.10)

$$R(N) = \frac{(N+1)\ell c}{T(N)} = \frac{(N+1)c/\tau}{1 + \psi_m(N) + \psi_s(N)}. \tag{3.11}$$

Substituting the rate c/τ from the granule level expression in Eq. 3.3, we get

$$R(N) = \frac{(N+1)R_\infty}{(1 + f_{1/2}/f + c_{1/2}/c) \cdot (1 + \psi_m(N) + \psi_s(N))}. \tag{3.12}$$

The computation rate $R(N)$ can also be expressed in the functional form of the *loss* function defined earlier as

$$R(N) = (N+1)R_\infty \; loss(f/f_{1/2} + c/c_{1/2}) \; loss(\psi_m(N) + \psi_s(N)) \tag{3.13}$$

which shows how the peak performance is degraded by the static (first *loss* term) and the dynamic (second *loss* term) overheads.

The average unit grain execution time, $T_G(N)$, with N competitor processes present can be expressed in terms the system characterization parameters defined

so far as follows.

$$T_G(N) = R_\infty^{-1}(c + mf_{1/2} + c_{1/2}) \cdot (1 + \psi_m(N) + \psi_s(N)) \tag{3.14}$$

The dynamic parameters $\psi_m(N)$ and $\psi_s(N)$ for a given workload can be obtained by experimental measurements at the grain level to determine the increase in the average execution time per unit grain G.

Phase level characterization

In addition to the increase in unit grain latencies caused by memory and lock interference, the effective BCU computation rate per phase is further decreased due to additional overhead of barrier synchronization at the end of a phase. If the additional latency due to the barrier with N competitor processes is given by $O_b(N)$, then the total time to complete a phase, $T(N)$, with ℓ unit grains per processor is obtained by augmenting Eq. 3.10.

$$\begin{aligned} T(N) &= \ell\tau(1 + \psi_m(N) + \psi_s(N)) + O_b(N) \\ &= \ell\tau(1 + \psi_m(N) + \psi_s(N)) \left(1 + \frac{O_b(N)}{\ell\tau(1 + \psi_m(N) + \psi_s(N))}\right) \end{aligned} \tag{3.15}$$

We define the phase-level dynamic characterization parameter *incremental barrier interference* (ψ_b) as follows:

$$\psi_b(N) = \frac{O_b(N)}{\tau}. \tag{3.16}$$

The barrier interference $\psi_b(N)$ for a given workload varies with N, and depends upon the implementation of the barrier and the degree of load imbalance within the phase preceding the barrier. Using this definition of barrier interference, the execution time of a single phase can then be expressed as

$$T(N) = \ell\tau(1 + \psi_m(N) + \psi_s(N))(1 + \tilde{\psi}_b(N)/\ell) \tag{3.17}$$

where

$$\tilde{\psi}_b(N) = \frac{\psi_b(N)}{1 + \psi_m(N) + \psi_s(N)}.$$

The modified parameter $\tilde{\psi}_b(N)$ can be interpreted as the incremental barrier overhead normalized with respect to the actual unit grain execution time $T_G(N)$ under contention conditions, as opposed to being normalized with respect to the uncontested unit grain time $T_G(0)$.

The effective BCU rate per phase including the barrier and with N competitor processes active, $R(N)$, can then be computed from Eq. 3.17 as

$$R(N) = \frac{(N+1)\ell c}{T(N)} = \frac{(N+1)R_\infty}{(1 + f_{1/2}/f + c_{1/2}/c) \cdot (1 + \psi_m(N) + \psi_s(N)) \cdot (1 + \tilde{\psi}_b(N)/\ell)}.$$

Expressing the net per-phase computation rate $R(N)$ in the *loss* functional form, we get

$$\begin{aligned} R(N) &= (N+1)R_\infty \; loss(f/f_{1/2} + c/c_{1/2}) \\ &\qquad loss(\psi_m(N) + \psi_s(N)) \; loss(\tilde{\psi}_b(N)/\ell). \end{aligned} \tag{3.18}$$

which shows the net performance as the peak performance degraded by all the characterization (both static and dynamic) parameters.

The total execution time per phase, $T(N)$, with N competitor processes active then becomes (in terms the system characterization parameters)

$$T(N) = R_\infty^{-1}(c + mf_{1/2} + c_{1/2}) \cdot (1 + \psi_m(N) + \psi_s(N)) \cdot (1 + \tilde{\psi}_b(N)/\ell). \tag{3.19}$$

The dynamic parameter $\psi_b(N)$ for a given workload is obtained by experimental measurements at the phase level to determine the increase in execution time of the phase on account of the barrier being present.

The system characterization parameters described in the previous paragraphs quantify the losses in performance that result from the static characteristics of an algorithm and the dynamic overheads overheads encountered at run-time. For a

Table 3.3. System characterization parameters

Type	Parameter	Description
Static parameters	R_∞	*Asymptotic computation rate (BCUs/s)*
	$f_{1/2}$	*Half-performance memory factor*
	$c_{1/2}$	*Half-performance lock factor*
Dynamic parameters	$\psi_m(N)$	*Incremental memory interference*
	$\psi_s(N)$	*Incremental lock interference*
	$\psi_b(N)$	*Incremental barrier interference*

given workload, the system characterization parameters (summarized in Table 3.3) help relate the expected performance of the workload to the application parameters (summarized in Table 3.4) as a function of the employed parallelism N (or degree of interference).

Table 3.4. Application parameters used in the performance model

Parameter	Description
c	Number of BCUs per unit grain
m	Number of shared-data accesses per unit grain
ℓ	Number of unit grains per processor per phase
N	Degree of interference (#of processors $= N + 1$)

3.4.3 Performance Metrics

The performance measurements taken at either the grain or phase level in our experimental framework are quantified using the fundamental metric called *cumulative interference* and denoted by $\Psi(N)$. This measure answers the question: how much longer is the expected execution time $T(N)$ of the given workload in a conflicting situation compared to its expected conflict-free execution time $T(0)$. This results in

the following definition of the cumulative interference measure.

$$\Psi(N) = \frac{T(N) - T(0)}{T(0)} = \frac{T(N) - \ell\tau}{\ell\tau} \tag{3.20}$$

For measurements performed at the grain level, $T(N) = \ell T_G(N)$ where $T_G(N)$ is the average unit grain execution time. Substituting this in Eq. 3.20, one can see that

$$\Psi(N) = \frac{T_G(N) - \tau}{\tau}. \tag{3.21}$$

In other words, the cumulative interference $\Psi(N)$ for measurements performed at the grain level can be inferred from the average execution times per unit grain. It is apparent that $\Psi(N) \geq 0$ always.

For grain level experimental measurements, we also define a second metric called *unit grain efficiency*, denoted as $\xi(N)$, that measures the relative performance of the unit grain in the presence of contention with respect to its uncontested execution time. It is given by the following ratio.

$$\xi(N) = \frac{\tau}{T_G(N)} \tag{3.22}$$

Combining equations 3.21 and 3.22 it can be seen that

$$\xi(N) = \frac{1}{1 + \Psi(N)} \tag{3.23}$$

The value of $\xi(N)$, $0 < \xi(N) \leq 1$, for a given point in the performance space expresses the extent of deterioration of a unit grain performance as a result of conflicts. A value of $\xi(N) = 1$ indicates no degradation at all implying that the concurrently executing unit grains in the workload do not suffer any mutual interference. This is, obviously, the ideal situation for achieving the best possible utilization of the processing resources for a group of concurrent tasks. The cumulative interference for this ideal case is $\Psi(N) = 0$.

3.4.4 Aggregate Multiphase Performance

The usual parameter that is used to compare the performance of algorithms is the speedup, which is defined as $S(N) = T(1)/T(N)$ where $T(1)$ and $T(N)$ are respectively the times for the algorithm to run on one and N processors. Using the *rate of work* notation used in our study, speedup can be written as

$$S(N) = \frac{R(N)}{R(1)}.$$

However, the value of the speedup alone cannot be used to compare the execution time of two algorithms unless the value of $T(1)$ is the same in both cases. Put another way, speedup is the execution speed measured in arbitrary units which change from algorithm to algorithm if $T(1)$ changes. It is quite possible in the comparison of two algorithms, for the algorithm with the worst speedup to execute in the least time, if the $T(1)$ for the worst algorithm is the greater. We prefer therefore to measure performance in absolute units (for example, BCUs per second), which is the variable $R(N)$. It should always be remembered that the objective of algorithm development is to reduce the execution time $T(N)$ (*i.e.*, increase $R(N)$) which is not necessarily the same as increasing the speedup. An algorithm with the greater speedup in some sense uses the parallel hardware more intensely (*e.g.*, there are fewer idle processors), but it does not necessarily execute in the least time.

Since a program, in our study, is an ensemble of multiple phases (Figure 3.1), the aggregate performance of the program may be characterized by the performance of its component phases. The performance of each phase, in turn, is characterized by the static parameters $(R_\infty, f_{1/2}, c_{1/2})$ and the dynamic parameters $(\psi_m(N), \psi_s(N), \psi_b(N))$ for the workload within that phase and follows the performance model elaborated earlier in this section.

The net computation rate of a program is simply the total number of BCUs computed, W, divided by the total computation time, $T(N)$. Note that $T(N)$ depends on the multiprocessor used, but W is constant for a given problem. Similarly, the computation rate of an individual phase k is $R_k = w_k/T_k$ where w_k is the total number

of BCUs computed by phase k and T_k is the total time required by phase k. The net rate of the program containing v phases is

$$R_{net} = \frac{\sum w_k}{\sum T_k} \quad 1 \leq k \leq v$$

or, it can also be written as

$$R_{net} = \frac{\sum w_k}{\sum \frac{w_k}{R_k}} \quad 1 \leq k \leq v.$$

Thus, the net computation rate of a program is the *weighted harmonic mean* of the computation rates of the component phases (not the arithmetic average of the rates). Note that the weights are the total computation work of each phase.

3.5 The Workload Emulation Kernels

Once an appropriate characterization for the unit grain has been selected, we have a method of specifying different workloads of interest by assigning suitable values to the grain attributes and the input parameters. What is needed is an emulation program that uses the workload specification to mimic the execution behavior of an asynchronous program that would demonstrate the same characteristics, namely, memory reference and synchronization patterns. The Memory Access Degradation (MAD), Synchronization Access Degradation (SAD), and Barrier Access Degradation (BAD) kernels are a family of such emulation programs. As we are only interested in measuring the concurrent execution conflicts of the given workload, no real computation need be performed by the emulation programs. Their only purpose is to mimic the usage of shared resources of the specified workload keeping intact the timing relationships between the different components of the computational structure.

Each kernel is written to use a set $\mathcal{I}$ of input parameters and generate a set of performance measures, $\Phi(\mathcal{I})$, of interest by executing the the emulated workload in a controlled experiment. Each experiment represents a point in the performance space

of the system.

Access Degradation Kernel: $\mathcal{I} \longrightarrow \Phi(\mathcal{I})$

It should be emphasized that these kernels are different from standard benchmarks. They are not parts of "real" computations like the Livermore loop kernels. The key attribute of these kernels is that they are programs that do not perform any useful computation, but rather, they are programs that model the computation, memory access and synchronization *structure* of a class of workloads of interest. They generate synthetic loads that are designed to stress a particular aspect of the target system. The usefulness of this approach lies in the fact that:

- The measured performance is not tied to any specific application. The user can design selective workloads, using the workload characterization technique provided, to generate a system characterization of interest.
- A collection of such kernels can be used to quantify and compare the performance of existing, new, or experimental architectures.
- They are simple and, hence, the interpretation of the observed behavior in terms of the kernel structure is easy.

3.5.1 Measurement of Incremental Overheads

The static system characterization parameters $(R_\infty, f_{1/2}, c_{1/2})$ can be measured by timing the single-processor execution of a unit grain defined by a given input workload, and fitting the measured data to the timing model for uncontested execution time dictated by Eq. 3.6. The key purpose of the workload emulation kernels (MAD, SAD and BAD) is to facilitate the measurement of the incremental contribution of dynamic overheads along the three focal performance dimensions—memory contention, lock contention and barrier synchronization—for a given input workload. In other words, the kernels help calibrate the dynamic system characterization parameters (ψ_m, ψ_s, ψ_b) as functions of N and hence characterize the dynamic behavior of a given

workload. The incremental measurement relationship between the three kernels is shown in Figure 3.5.

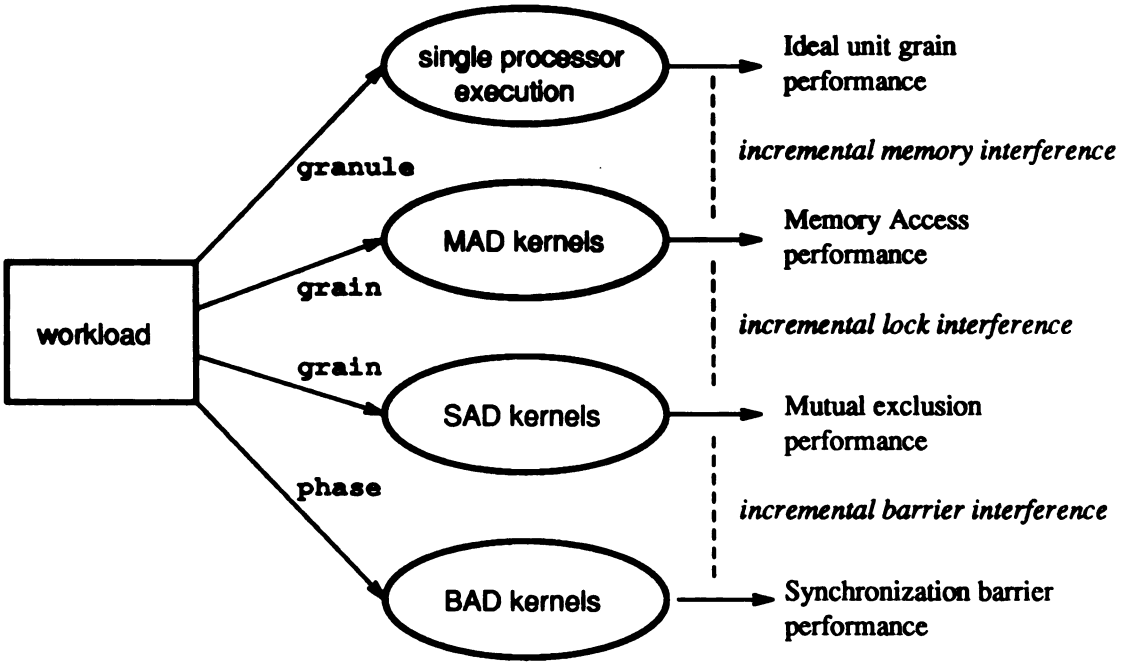


Figure 3.5. Incremental measurement of dynamic overheads

The kernels are executed in the order MAD → SAD → BAD for a given workload. The MAD kernels measure the run-time overheads arising only due to contention for shared memory; the SAD kernels measure the cumulative overheads arising due to memory as well as lock contention; and the BAD kernels capture the total cumulative overheads. Each kernel is coded so as to eliminate from its own measurements the incremental contention overhead measured by its successor kernel.

Each kernel computes the fundamental metric *cumulative interference* $\Psi(N)$, as defined in Eq. 3.20, by timing the execution of a given workload with varying number of competitors N. Let us denote the cumulative interference measured by the MAD, SAD and BAD kernels as Ψ_m, Ψ_s and Ψ_b, respectively; and the workload execution time measured with N competitors as $T^{mad}(N)$, $T^{sad}(N)$ and $T^{bad}(N)$, respectively.

Then from the definition of Ψ (Eq. 3.20), we can derive the expression for incremental memory interference ψ_m from the MAD kernels as

$$\Psi_m(N) = \frac{T^{mad}(N) - T(0)}{T(0)} = \frac{O_m(N)}{\ell\tau} = \frac{\hat{O}_m(N)}{\tau} = \psi_m(N). \tag{3.24}$$

Similarly, for the SAD kernels we have

$$\begin{aligned}\Psi_s(N) &= \frac{T^{sad}(N) - T(0)}{T(0)} \equiv \frac{(T^{sad}(N) - T^{mad}(N)) + (T^{mad}(N) - T(0))}{T(0)} \\ &= \frac{O_s(N) + O_m(N)}{\ell\tau} \equiv \frac{\hat{O}_s(N) + \hat{O}_m(N)}{\tau} \equiv \psi_m(N) + \psi_s(N).\end{aligned}$$

Therefore, the incremental lock interference ψ_s can be computed from the following expression.

$$\psi_s(N) = \Psi_s(N) - \psi_m(N) = \Psi_s(N) - \Psi_m(N) \tag{3.25}$$

The cumulative interference measured by the BAD kernels is given by

$$\begin{aligned}\Psi_b(N) &= \frac{T^{bad}(N) - T(0)}{T(0)} \equiv \frac{(T^{bad}(N) - T^{sad}(N)) + (T^{sad}(N) - T(0))}{T(0)} \\ &= \frac{O_b(N)}{\ell\tau} + \Psi_s(N) \equiv \frac{\psi_b(N)}{\ell} + \Psi_s(N).\end{aligned}$$

Therefore, the incremental barrier interference ψ_b can be computed as

$$\psi_b(N) = \ell(\Psi_b(N) - \Psi_s(N)). \tag{3.26}$$

The workload level at which experimental evaluation is performed and the metrics computed by each of the kernels is summarized in Table 3.5.

3.5.2 Kernel Structure

This section describes the program structure of the access-degradation kernels and their relationship with the experiment control parameters. As seen from Figure 3.4, every participating processor executes a unit grain (test or competitor), specified by

Table 3.5. Summary of access degradation kernel measurements

Workload processed by	Measurement level	Barrier present?	Metrics computed
1-proc execution	granule	no	$R_\infty, f_{1/2}, c_{1/2}$
MAD kernels	unit grain	no	Ψ_m, ψ_m
SAD kernels	unit grain	no	Ψ_s, ψ_s
BAD kernels	phase	yes	Ψ_b, ψ_b

the input parameters $\mathcal{I}$, repetitively. Each processor executes a concurrent loop as shown in Figure 3.6. All processors are synchronized at a barrier at the beginning to ensure that they start executing their assigned grains at the same time. Two distinct iterative regions can be identified in this concurrent loop. The code to emulate the unit grains specified by $\mathcal{I}$ is enclosed within the inner loop with `i` as its loop control variable, and is repetitively executed N_{itr} number of times. In reality, we unrolled this loop to reduce the loop overhead per iteration. The additional code delimited by the two invocations to the `read_clock()` function is what we call an *observation.*

The outer loop with `k` as its loop control variable constitutes an *experiment.* Thus an experiment consists of a set of observations (controlled by the variable N_{repeat}). All the observations in an experiment are assumed to be statistically independent. The final step in an experiment consists of computing the arithmetic mean and variance of the sample of recorded observations. The sample mean is used as the observed measure of performance, $\Phi(\mathcal{I})$, for the input parameter set $\mathcal{I}$. Confidence intervals are computed for each set of observations to ensure that the variation between extremes is within reasonable limits.

The length of each observation run, N_{itr}, and the size of an experiment sample, N_{repeat}, are selected based on the resolution of the clock available on the target system, the overhead of the timing function and the overhead of the loop control statements. The choice of suitable values for these two control parameters is crucial to the minimization of experimental error and the confidence interval of the measured quantities [111]. A more detailed discussion of the dependence of experimental errors on these

```
Concurrent Loop
{
    for (k = 0; k < Nrepeat; k++)
    {
        kernel_specific_initialization();
        barrier();
        t1 = read_clock();
        for (i = 0; i < Nitr; i++)
        {
            body of test/competitor grain
        }
        t2 = read_clock();
        runtime[k] = (t2 - t1) / Nitr;
    }
    compute_sample_stats (Nrepeat, runtime);
}
```

Figure 3.6. The concurrent loop structure of the kernels

control parameters is provided in the next section.

3.5.3 Minimization of Experimental Errors

One of the important considerations of the experimental system characterizer is to control the accuracy and exactitude of the measurements. In this section, we discuss the sources of variability in the measurements and illustrate the importance of the control variables N_{itr} and N_{repeat} in minimizing experimental errors and hence confidence intervals.

Referring to Figure 3.6 we see that the time recorded in each observation O_j also includes the execution time of the loop control code that controls the test ($FOR_{overhead}$) and the overhead incurred by the timer routine ($C_{overhead}$). These have to be subtracted from each observation O_j. These measurements have their own variance and the subtraction of these overheads increases the variance of our measurements. The

mean value $\hat{O}$ of a sample of observations is

$$\hat{O} = \frac{1}{N_{repeat}} \sum_{j=1}^{N_{repeat}} O_j$$

and its variance

$$\sigma^2 O = \frac{1}{N_{repeat} - 1} \sum_{j=1}^{N_{repeat}} (O_j - \hat{O})^2$$

Now the mean value of each experiment is the time it takes to execute the body of the test/competitor grain N_{itr} times, plus the overhead of the timing function

$$\hat{O} = N_{itr}(\hat{T} + FOR_{overhead}) + C_{overhead}$$

where $\hat{T}$ is the mean time it takes to execute once the body of the test/competitor grain. We can compute this value and the variance with the equations

$$\hat{T} = \frac{\hat{O} - C_{overhead}}{N_{itr}} - FOR_{overhead}$$

and

$$\sigma^2 T = \frac{\sigma^2 O + \sigma^2 C_{overhead}}{N_{itr}^2} + \sigma^2 FOR_{overhead}$$

Looking at the above equations we can see that there are four factors affecting the magnitude of variance: the resolution of the timing function; the variance of our observations; the variance of the execution time of the timing function; and the variance of the *FOR* control statements. If the execution time of each observation is such that we have

$$O_j \gg C_{resolution} + C_{overhead} + FOR_{overhead}$$

then the only factors that affect our measurements are the dispersion of our observations.

We have two ways of reducing the variance of our results and therefore the size of the confidence intervals—increasing the length N_{itr} of an observation and increasing

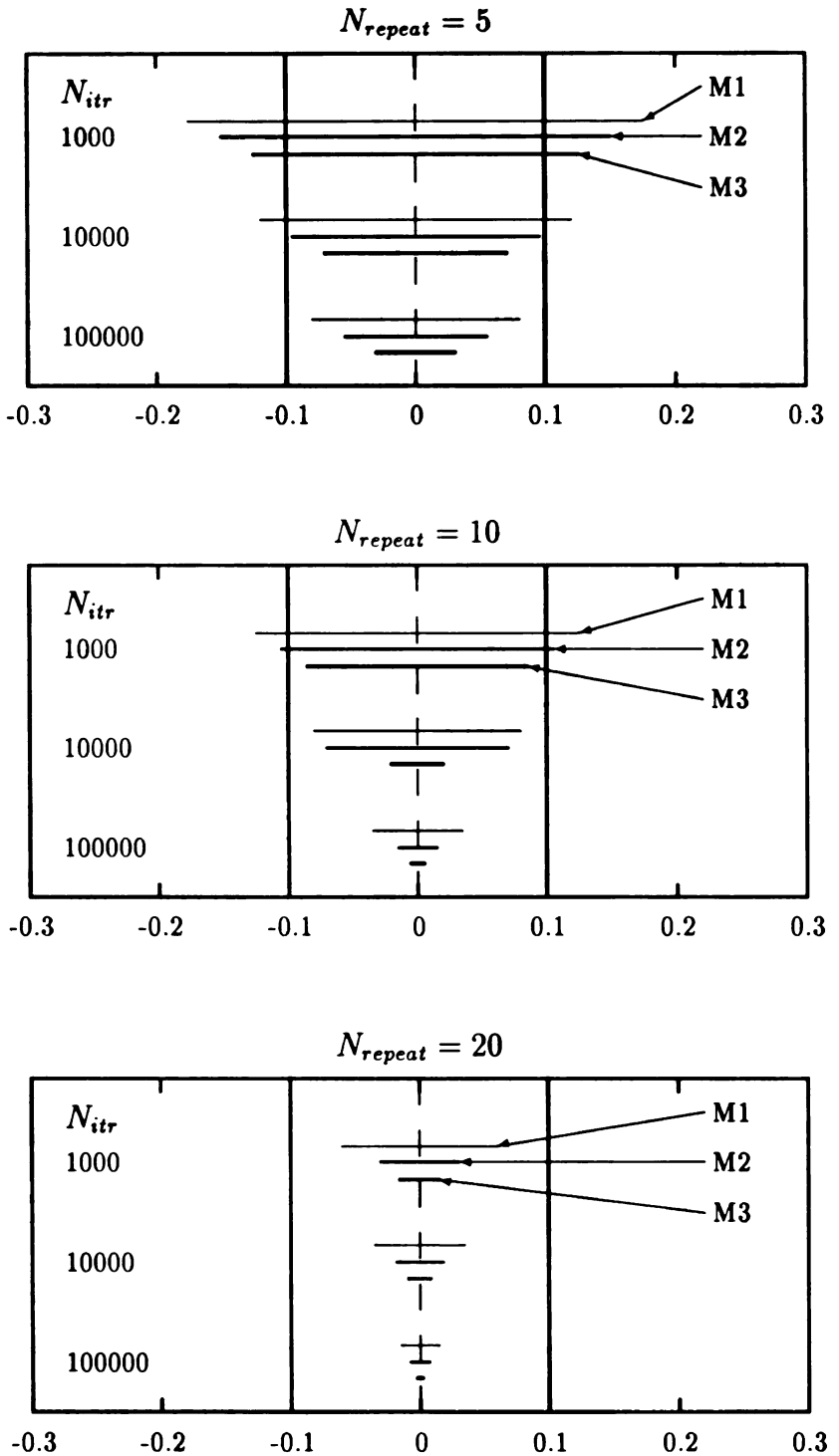


Figure 3.7. Normalized 90 percent confidence intervals for three workload measurements on the Sequent Symmetry for $N_{repeat} = 5, 10, 20$

the sample size N_{repeat}. It is important to know the values for N_{itr} and N_{repeat} that will give a small standard deviation in our measurements. These values are system dependent. In Figure 3.7 we show the normalized 90 percent confidence interval of three workload measurements (indicated as M1, M2 and M3) on the Sequent Symmetry S81 multiprocessor system. The workload measurements were performed for values of $N_{itr} = 1000, 10000$ and 100000. We also obtained measurements for $N_{repeat} = 5, 10$ and 20. The confidence intervals for $\hat{T}$ are obtained using the Student's t distribution and the standard error of $\hat{T}$ as follows:

$$\left[\hat{T} - \frac{t_{.95}}{\hat{T}}\left(\frac{\sigma^2 T}{N_{repeat}}\right)^{1/2},\ \hat{T} + \frac{t_{.95}}{\hat{T}}\left(\frac{\sigma^2 T}{N_{repeat}}\right)^{1/2}\right]$$

and the normalized confidence intervals are

$$\left[-\frac{t_{.95}}{\hat{T}}\left(\frac{\sigma^2 T}{N_{repeat}}\right)^{1/2},\ \frac{t_{.95}}{\hat{T}}\left(\frac{\sigma^2 T}{N_{repeat}}\right)^{1/2}\right]$$

We can see that for a fixed value of N_{repeat} the confidence interval of our measurements decrease as the time of the measurement (controlled by N_{itr}) increases. We obtained acceptable results on the Sequent Symmetry for $N_{repeat} = 10$ and $N_{itr} = 100000$.

3.6 Summary

In this chapter, we developed a comprehensive experimental performance characterization methodology for shared memory multiprocessors based on measurement of the static and dynamic overheads that arise during program execution. The run-time interference along three principal performance dimensions have been considered, namely, memory contention, lock contention and synchronization barriers. A parallel computation structure with multiple phases separated by global synchronization barriers and asynchronous balanced task execution within each phase has been selected as the basis of the performance characterization study in this dissertation. A hierarchical workload characterization technique using the abstraction of a *unit*

grain has been proposed for the flexible and parametric specification of workloads of interest. Three *static* parameters $(R_\infty, f_{1/2}, c_{1/2})$ and three *dynamic* parameters $(\psi_m(N), \psi_s(N), \psi_b(N))$ were defined to describe the static and dynamic behavior of a given input workload as a function of the number N of processes competing for shared resources. The structure and semantics of of three kernel families — MAD, SAD and BAD — was presented to facilitate the measurement of the static and the dynamic parameters. Finally, the primary sources of experimental errors and means to minimize them were also discussed.

CHAPTER 4

MAD KERNELS AND MEMORY ACCESS PERFORMANCE

On large-scale multiprocessors, access to common memory is one of the key performance limiting factors due to the significant overheads that may be encountered related to contention for access to shared memory modules. The shared memory performance depends not only on the characteristics of the memory hierarchy itself, but also upon the characteristics of the memory address streams and the interaction between the two. The factors that cause memory access conflicts and the architectural solutions adopted to minimize contention were discussed in Chapter 2. Quantitative assessment of the contention overheads for different types of memory access workloads promotes a better understanding of the performance of systems as they scale in size and use newer memory technologies.

The MAD kernels and the related experimental framework described in this chapter provides an effective testbed for characterizing the shared memory performance for a variety of memory access workloads. Experimental measurements are performed at the unit grain level with multiple unit grains executing the specified workload in parallel without a global synchronization barrier. The performance metrics are computed on a per-unit-grain basis. The MAD kernels can be used in isolation to perform

a detailed evaluation of the sensitivity of a shared memory organization to various memory access parameters; or they can be used in conjunction with the SAD and BAD kernels, within the hierarchical framework described in Chapter 3, to characterize the incremental loss in performance for a given workload resulting from memory access conflicts.

4.1 Preliminary Studies

The performance studies described in this section were designed and aimed at a preliminary investigation of the performance degradation experienced by multiprocessors as a result of contention for shared memory resources. Three commercial multiprocessors were used as the target systems in the study: a 96-node BBN GP1000 (called BBN-1), a 32-node BBN TC2000 (called BBN-2) and a 24-node Sequent Balance 21000 (called Balance). The architectural features of these systems were described in Chapter 2. The performance measurements taken were used to quantify two major sources of overhead in shared memory accesses, namely, non-local access latency and waits due to access conflicts. An analytical model for these overhead factors was formulated to explain and corroborate the observed behavior [96].

Parallel execution performance degradation in the presence of synchronization *locks* were also a subject of these preliminary investigations. The presence of lock-based mutual exclusion operations introduces two additional sources of runtime overheads, namely, locking latency and waits due to lock conflicts. The experimental results from input workloads containing lock-based mutual exclusion operations are reported in Chapter 5 (Section 5.1). The observed performance losses solely due to memory access conflicts using workloads with no lock accesses are presented in this section.

4.1.1 Workload Parameters

The unit grain abstraction is used as the fundamental unit of input workload specification. A very simple parametric workload model is used to create a variety of

program behaviors. A unit grain G is characterized by three attributes; $G = (c, m, x)$. The attribute c defines the number of local computational operations, including local memory access, performed by a process between consecutive accesses to a critical section (defined to be a unit grain). This parameter controls the computational load of each processor. Similarly, the attribute m defines the number of shared data references, not mutually-exclusive, made by a process between successive accesses to critical sections. The attribute x specifies the amount of time (in μ *secs*) spent by a process within each critical section. This attribute is specified as an absolute time duration to highlight the influence of critical section length on performance. Since memory contention overheads are the focus of the study described in this section, a value of $x = 0$ is used.

In addition to the unit grain attributes two more parameters, N and M, are used to specify global characteristics of the workload. N specifies the number of competitor processes interfering with the execution of any grain whereas M specifies the number of shared data objects used by the concurrent processes. The M objects are assumed to be evenly distributed over the available shared memory modules. Thus, the complete input workload specification includes the six parameters (N, M, c, m, x).

For notational convenience we define two derived parameters in terms of the basic input parameters described above. First, the *granularity* $\omega = c + m$ of a program is defined to be the total number of operations performed between synchronization points (*e.g.*, critical sections). Second, the *shared-access fraction* $\rho = m/(c + m)$ is the fraction of total operations devoted to shared data accesses.

In each execution of a unit grain, a processor performs ω operations, each operation being a local computation or shared data access in the proportion dictated by ρ. Each shared data reference consists of a *read* followed by a *write* to the shared data location. This is done to force the reference to actually go out to shared memory even in the presence of data caching. If $x \neq 0$, then the processor acquires a lock and enters the critical section for a duration of $x\mu$ *secs*. Only homogenous workloads, with every participating processor executing an identical copy of the unit grain G, are used in these preliminary studies. In other words, the workload unit G is "replicated" on all

the $N + 1$ processors involved.

4.1.2 Quantities Measured

For each workload specified by a set of input parameters, a corresponding set of timing data that essentially consists of the effective execution time per unit grain, $T_G(N)$, is generated. The two performance metrics computed for each workload are *unit grain efficiency* (ξ) and *overhead factor* (Θ) defined as follows:

$$\xi(N) = \frac{T_G(0)}{T_G(N)} \quad \text{and} \quad \Theta = \frac{T_G(N) - T_G(0)}{T_G(0)}.$$

Because of the replicated workload used, this definition of efficiency ξ of running a program on a parallel architecture can also be interpreted as the ratio of the actual speedup achieved to the ideal speedup achievable on that architecture.

The loss in efficiency is attributable to two key overheads arising from shared data accesses — non-local access latency and waiting time due to access conflicts. The first kind of overhead is an important factor for a non-uniform organization of memory hierarchy (NUMA multiprocessors). The second type of overhead is a result of contention for hardware resources during shared data access. If we denote the overhead time due to non-local memory latency as O_l, and the overhead due to hardware contention as O_c, then we can rewrite the expression for overhead factor Θ as

$$\Theta = \frac{T_G(N) - T_G(0)}{T_G(0)} = \frac{O_l + O_c}{T_G(0)} = \theta_l + \theta_c$$

which gives the normalized overhead components θ_l (*latency factor*) and θ_c (*contention factor*).

4.1.3 Memory Access Overhead Factors

In this section, we formulate a mathematical model to describe the behavior of concurrent unit grain execution and the resulting overheads. To facilitate the brevity of expression, we define some basic cycle times that characterize the program execution

on each system. All subsequent execution and overhead times will be expressed in terms of these fundamental time units. Define

t_c = avg. time per local computation operation,

t_a = avg. time per local memory access,

t_l = avg. latency per remote memory access,

t_w = avg. waiting time per remote memory access due to contention,

t_{lk} = avg. time to execute the ***lock*** primitive without contention,

t_{ul} = avg. time to execute the ***unlock*** primitive without contention.

The time t_a denotes the basic time required to access a local data object. The time t_l denotes the additional latency component incurred in accessing a remote data object. In the BBNs, the t_l component is non-zero since a remote memory reference goes out on the interconnection network whereas a local reference does not. Thus, in the absence of contention, the time for a remote memory access on the BBNs is given by $t_a + t_l$. However, in the Balance, the bus latency is subsumed in the basic memory access time t_a since it is an integral component of the memory access time. There is no additional delay incurred by "remote" references, since local and remote memories are indistinguishable, thus giving $t_l = 0$. The time t_w denotes the additional delay over and above the components t_a and t_l caused as a result of contention among concurrent memory accesses. Note that all the times defined above (except t_w) are constant being the characteristic of the underlying hardware/operating system and do not depend on the workload. The values of t_{lk} and t_{ul} include the overhead of function call. A comparison of these fundamental unit times for the three systems under consideration is shown in Table 4.1.

The remaining term t_w, however, is dependent on the memory reference pattern and the communication bandwidth of the interconnection medium. It embodies the queueing delay experienced by a memory reference that must traverse the interconnection medium to be serviced. This delay arises from the interference between concurrent memory references at the destination memory module as well as on the

Table 4.1. Basic time measurements for the overhead factors model

System	$t_c(\mu s)$	$t_a(\mu s)$	$t_l(\mu s)$	$t_{lk} + t_{ul}(\mu s)$
BBN-1	10.12	2.18	3.42	71.83
BBN-2	1.49	0.71	1.43	28.62
Balance	37.22	10.85	0.00	83.18

network. We need to obtain an expression for t_w that reflects its dependence on the workload. Several earlier works have modeled memory interference for MINs using Markovian models [18] and probabilistic analysis [19, 101]. Similar work done for analyzing contention in bus-based systems include [86, 31, 43].

Contention Time on the BBNs

We use the result derived by Patel [101] using probabilistic analysis for Delta networks. The derived results apply to a p-stage MIN using $k \times k$ switching elements. A *memory-access-cycle* (MAC) is defined to be the time interval from the initiation of a memory request to the completion of the request. No distinction is made between read and write cycles in the analysis. The primary assumptions on which this analysis is based are as follows.

(i) The memory references generated by each processor are independent of each other.

(ii) The memory references are uniformly distributed over all the memory modules.

(iii) All the k^p potential processors (since the system consists of a p-stage MIN with $k \times k$ switches) in the system participate in the memory workload creation.

If each processor generates memory requests at the rate of r requests per MAC, then for any input line of a switch in stage-1 of the MIN, the probability

$$\Pr[\text{a request arrives during a MAC}] = r$$

The first two assumptions are satisfied by our performance measurement framework, where r is determined by the processor workload. However, since only $n = N+1$ processors (out of the total capacity k^p) participate in generating memory requests, the effective request rate at each stage-1 switch must be changed in assumption *(iii)*. Assume that any processor in the system could be selected to participate with equal probability. Now, for any switch input at stage-1, we have

$$\begin{aligned} \Pr[\text{input is active}] &= n/k^p \\ \Pr[\text{a request arrives during a MAC} \mid \text{input is active}] &= r \\ \Pr[\text{a request arrives during a MAC} \mid \text{input is not active}] &= 0 \end{aligned}$$

By using Bayes' formula [39], we obtain

$$\Pr[\text{a request arrives during a MAC}] = \hat{r} = \left(\frac{n}{k^p}\right) \cdot r$$

Thus, $\hat{r}$ becomes the effective request rate at each input line of stage-1 switches. Using the new effective request rate, the probability P_A that an arbitrary memory request is accepted by the MIN (from [101]) is given by

$$P_A = \frac{r_p}{\hat{r}} = \frac{k^p}{n} \cdot \frac{r_p}{r}$$

where

$$r_i = 1 - \left(1 - \frac{r_{i-1}}{k}\right)^k \text{ and } r_0 = \hat{r} \tag{4.1}$$

We do not have a closed form solution for P_A, but plots [101] of P_A *vs. network size* (k^p) indicate that P_A decreases logarithmically as the *network size* increases.

On the BBN system, a memory conflict is essentially a conflict at the output line of a switch in the last stage of the MIN. Hence, it is accounted for by the switch contention analysis. The average number of wasted memory cycles per request, w, is easily computed by noting that a request that is rejected i times consecutively before

being accepted waits for i cycles.

$$w = \sum_{i=0}^{\infty} i(1-P_A)^i P_A = \frac{1-P_A}{P_A} = \frac{n}{k^p} \cdot \frac{r}{r_p} - 1$$

Hence, the average waiting time per request due to contention is

$$t_w = w(t_a + t_l) = \left(\frac{n}{k^p} \cdot \frac{r}{r_p} - 1\right)(t_a + t_l) \tag{4.2}$$

Independent, uniformly distributed references are not, however, an accurate model in the presence of global locks, even if all non-lock references are uniformly distributed. Hence, the expression above will not apply accurately in situations with a single "spike" or *hot-spot* in the memory reference pattern. The hot spot case is analyzed later in this chapter in Section 4.3.

Contention Time on the Balance

We use the result derived by Das and Bhuyan [31] using probabilistic analysis for multiple-bus multiprocessors. The derived results hold for a system with n processors, M memory modules and B buses. We have adapted the expressions for the special case of a single bus system ($B = 1$) such as the Balance. Again, no distinction is made between read and write cycles as for the BBNs, and the analysis is based on the following assumptions.

(i) The bus operation is synchronous, *i.e.*, all requests are issued at the beginning of the bus cycle.

(ii) The bus is circuit-switched, *i.e.*, the bus is held for the entire duration of a memory access.

(iii) The requests generated during a bus cycle are random and are independent of each other.

(iv) The requests issued in successive cycles are independent of each other.

The fourth assumption is unrealistic because a rejected request will indeed be resubmitted in the next cycle. However, this assumption leads to simpler analysis, and it does not result in a substantial difference in the actual results [18]. Let r be the probability with which a processor generates a request in every bus cycle. The probability that there is at least one request for a memory module M_i, when n processors participate, is given (from [31]) by

$$X = 1 - \left(1 - \frac{r}{M}\right)^n$$

The number of memory services requested in a cycle is a Binomial random variable with parameters M and X. Hence, the expected number of memory requests received per cycle is MX. Since only one of these requests can be accepted by the bus, the probability P_A that an arbitrary request is accepted can be written as $P_A = 1/MX$. Now, using an argument similar to that for the BBNs, the average number of wasted bus cycles per request, w, can be computed as

$$w = \frac{1 - P_A}{P_A} = (M - 1) - M\left(1 - \frac{r}{M}\right)^n$$

Therefore, the average waiting time per request due to contention is

$$t_w = wt_{bus} = \left((M - 1) - M\left(1 - \frac{r}{M}\right)^n\right) t_{bus} \tag{4.3}$$

where t_{bus} is the bus cycle time (100 ns for the Balance). Note that all the terms in Eq. 4.3, except n, are constant for a given system and workload. The value of $n = N + 1$ varies according to the input parameters specified. It should also be noted that since successive memory requests on the Balance are pipelined onto the bus, the above equation only provides an upper-bound for the contention time.

Overhead Factors

Recall that $T_G(0)$ was used as the unit of normalization in the definition of Θ. We can express $T_G(0) = \tau$ in terms of the workload parameters and the basic time units

as

$$T_G(0) = \tau = ct_c + mt_a + x = \omega(1-\rho)t_c + \omega\rho t_a + x$$

Note that we have not added the terms $(t_{lk} + t_{ul})$ in the above expression since an application with a single process does not need the service of a lock to exclusively access a shared resource. We can now express the overhead factors in terms of the time units defined earlier.

Latency Factor.

For the Balance, since there is no distinction between local and remote memory, there is no additional overhead incurred by remote accesses. The bus latency constituent of the memory access time is subsumed in the basic memory access time t_a. Hence, there is no additional latency overhead, resulting in $\theta_l = 0$.

In the case of the BBNs, the latency overhead is contributed by those shared-data references that are sent out on the interconnection network. Every iteration contains m references to shared-data, each involving two accesses (read/write), and one shared access each for acquiring and releasing the lock. Converting the shared-memory reference count in terms of time, we obtain

$$\theta_l = \frac{O_l}{\tau} = \frac{(2m+2)t_l}{\tau} = \frac{2(\omega\rho+1)t_l}{\omega(1-\rho)t_c + \omega\rho t_a + x} \tag{4.4}$$

Contention Factor.

The contention overhead is contributed by all shared-data references (in both the BBNs and the Balance). Using the same arguments as for latency factor above, we obtain the following result for *contention factor* for the BBNs as well as the Balance.

$$\theta_c = \frac{O_c}{\tau} = \frac{(2m+2)t_c}{\tau} = \frac{2(\omega\rho+1)t_w}{\omega(1-\rho)t_c + \omega\rho t_a + x} \tag{4.5}$$

4.1.4 Experimental Results

During the course of our experimentation, more than two hundred input parameter sets were tested. Each parameter set was constructed by varying the input parameters according to one of the workload forms shown in Table 4.2. The range of parameters

was selected with the goal of observing the sensitivity of ξ and Θ to different types of workloads. The data presented in this section are only a few excerpts from the workloads created to measure pure memory contention characteristics [95] with no synchronization. Data corresponding to workloads (C and D) with synchronization in the unit grain are reported in Section 5.1 of Chapter 5.

Table 4.2. Parameter settings for different workload types used in the preliminary studies

Workload	N	M	ω	ρ	$x(\mu s)$
A	0 to *max*	1	100	0.0 to 0.4	0
B	0 to *max*	$M = N + 1$	100	0.0 to 0.4	0
C	0 to *max*	$M = N + 1$	100,500	0.0 to 0.4	0 to 100
D	*max*	$M = N + 1$	500	0.1 to 0.4	0 to 150

Value of *max* chosen based on the number of processors available.

Workload A

This workload represents an extreme case in that it creates a "hot-spot" memory access pattern by forcing all processes to continually access a single memory module. The effect of a hot-spot on the efficiency ξ is shown in Figure 4.1 for two different values of shared-access fraction ρ.

As can be seen from Figure 4.1, the efficiency drops by more than 50% at $N = 20$, $\rho = 0.1$ for the BBN systems. The performance degradation becomes even more pronounced as the values of N and ρ increase. On the other hand, the Balance is not affected as much by the hot-spot, since the entire shared-memory of Balance forms one indistinguishable unit. As long as the mean time between requests is greater than or equal to the memory-access-time t_a, there is no contention at the memory module and the Balance is able to service the requests efficiently.

Obviously, the deterioration in execution speed in the case of BBN-1 and BBN-2 is primarily due to the increasing contention overhead θ_c. The expression for t_c (Eq. 4.2),

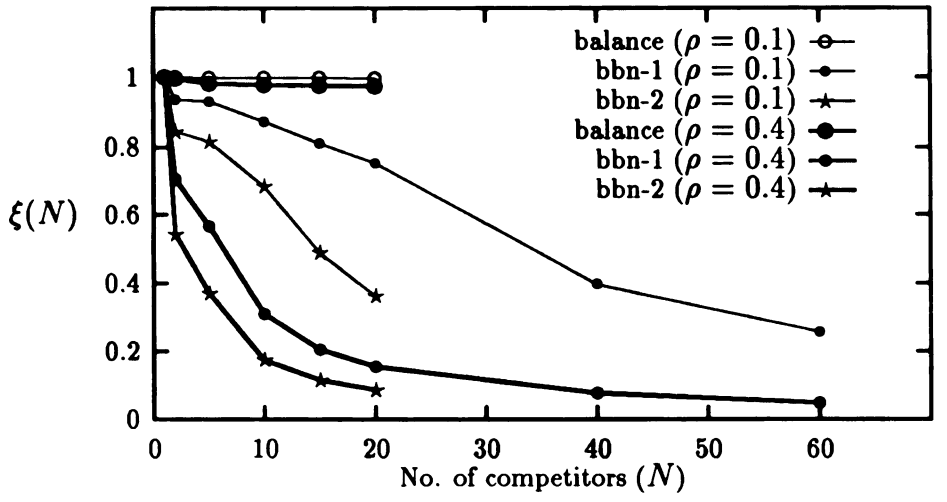


Figure 4.1. Efficiency vs. N ($M = 1, \omega = 100, x = 0$)

however, fails to explain the phenomenon as it is based on the premise that shared references are uniformly distributed over the memory modules. An explanation for the observed behavior is found from [105], a communication bandwidth analysis done for the RP3 system. It shows that the effective bandwidth of the network reduces drastically in the presence of a memory "hot-spot". This is true even when the fraction of total memory references directed at the hot-spot is as low as 1%. The severe degradation in bandwidth occurs due to the *Tree Saturation Effect* described in Chapter 2, which not only deteriorates the access time for the hot-spot references, but penalizes other references as well.

Workload B

This workload highlights the hardware overhead characteristics of each architecture. Figure 4.2 shows the trend in efficiency as the number of processors executing concurrently is varied. Since $x = 0$ in this case, all the overhead is due to communication latency and contention in hardware. Clearly, both θ_l and θ_c depend on the shared-access fraction ρ and increase linearly with ρ as indicated by Eqs. 4.4 and 4.5. For

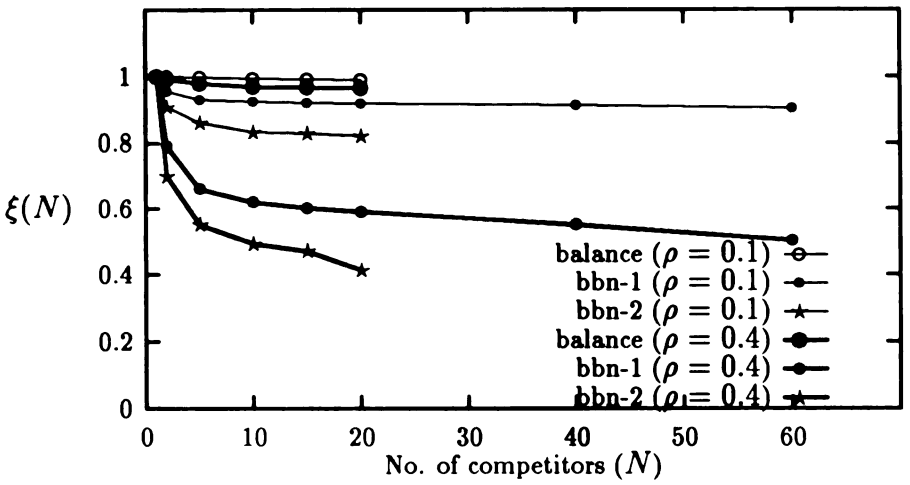


Figure 4.2. Efficiency vs. N ($M = N + 1, \omega = 100, x = 0$)

the Balance, $\theta_l = 0$ and, again, the loss in efficiency is little. Notice that for the BBNs, ξ drops initially but remains relatively flat for higher values of N. This is due to the fact that as N increases, the number of memory modules also increase and the data references get redistributed uniformly over the memory modules. The efficiency curve for BBN-2 drops off faster than the than the corresponding curve for BBN-1. This can be inferred by examining the expression for t_c (Eq. 4.2). The factor (n/k_p) in this equation signifies the fraction of the network capacity that is occupied. For a given value of N, this factor is larger for the BBN-2 ($k = 8, p = 2$) than for BBN-1 ($k = 4, p = 4$), thus yielding a larger value for t_c for BBN-2. However, the relatively flat shape of the curves for higher values of N points to the fact that the systems can be utilized better by using a larger number of processors to compensate for the loss in efficiency due to latency and contention.

4.2 MAD Workload Parameters

The major consideration in memory system design for multiprocessors is that the memory bandwidth must match the memory demand of the processors. The effec-

tiveness of the memory design in meeting this goal depends not only on the organization of the memory hierarchy, but also on the distribution of the shared data in the hierarchy, the memory reference pattern of the program, and the locality of memory references. In addition to *temporal locality* and *spatial locality* of references, parallel computing also makes a new type of locality, called *processor locality*, desirable. To keep high processor locality, unnecessary interleaving of references by more than one processor to the same memory data should be avoided.

It is clear that the workload used to evaluate the memory performance can have a strong influence on the results. For example, a (perhaps artificial) workload exhibiting little or no locality of reference will tend to favor a very simple processor–memory interconnection network built out of fast, dumb switches over a network with smarter, slower switches. Hence, the selection of appropriate workloads of interest is of prime importance to the success of the experimental study.

4.2.1 Unit Grain Characterization

The domain of the parameter space for investigating the shared-memory performance is prohibitively large. Unfortunately, measurement data about the behavior of real workloads are scarce. So, it is not possible to make performance comparisons using "a typical, real workload". Therefore, we adopt a flexible parametric model of unit grain characterization that facilitates the exploration of performance over a wide spectrum of memory access workloads. The attributes selected for the unit grain should help probe the memory system systematically by creating diverse sets of memory address streams to determine its sensitivity to the different workload characteristics. These workloads not only measure the sustained memory bandwidth under different memory demands, but also highlight potential bottlenecks. The unit grain characterization selected for this purpose is summarized in Table 4.3.

Characterization of g_m:

The shared-memory access granule g_m is characterized by a *4-tuple* of attributes: $g_m = (p, d, s, m)$. The first attribute, p, simply indicates the probability of a shared

Table 4.3. Unit grain attributes for studying memory access behavior

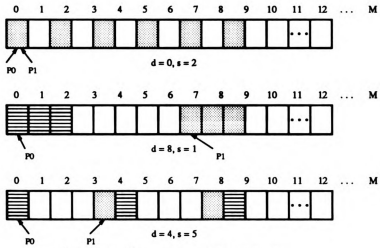
Granule	Attribute	Meaning
common	N	number of competitor processors
	M	number of shared data elements
g_m	p	probability of write access to shared memory
	d	initial distance of concurrent address streams
	s	stride of memory access
	m	number of shared memory accesses per granule
g_c	c	number of basic computation units (BCUs)
g_s	ϕ	non-existent

data reference being a write access. In other words, $p = 0$ implies that all accesses are *reads*, and $p = 1$ implies all accesses to be *writes*. As mentioned earlier, writes to shared data by multiple processors are typically performed within critical sections in a mutually exclusive fashion unless the concurrent writes are guaranteed to be consistency preserving.

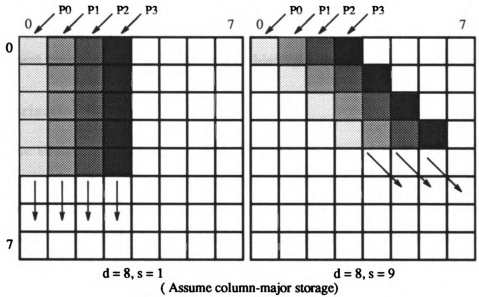
The next attribute, d, determines the initial disposition of the concurrent memory reference streams emanating from the processes executing in parallel. It denotes the distance between the starting addresses of shared data access of each processor expressed as number of shared data elements. In other words, if there are M shared data elements in all, then the processor P_i begins its string of memory accesses with element $i \times d$ (modulo M). Thus, if the shared data elements are accessed with regular stride, then the attribute d can be used to stagger the starting addresses of multiple processors in any desired fashion. For instance, a value of $d = 0$ causes all participating processors to begin their shared data access with the 0^{th} element.

The attribute s represents the stride of shared data access from one memory access to the next, thus defining the *spatial distribution* of the memory request streams. By manipulating the access stride, the effect on performance of the mapping strategies used to assign elements of an array to the memory banks at a given hierarchy can be evaluated. Depending on how the shared data elements are distributed over the memory hierarchy, using different access strides will cause the memory request

transactions to traverse over different components of the processor–to–memory interconnection. Figure 4.3 illustrates the use of the attributes d and s together to create a variety of memory access patterns for both one-dimensional and two-dimensional shared data structures.



(a) One-dimensional data access



(b) Two-dimensional data access

Figure 4.3. Creation of memory access patterns using attributes d and s

Finally, the attribute m denotes the number of memory accesses to be performed within a single memory-access granule. The value of m determines the granularity of shared data access within a grain. The main purpose of changing this attribute is to control the density of memory requests, thus highlighting the interaction between request bursts and idle periods.

Characterization of g_c:

Since all the computation within granule g_c operates purely on processor private data out of a private memory space (assumed to be available locally), by our definition, the computation granule does not alter the memory interference behavior of the shared data access stream as it is external to the processor. Its only influence is setting the memory access rate and, hence, the temporal distribution of the shared data references. So we have characterized the computation granule g_c by simply a *1–tuple* consisting of a delay count: $g_c = (c)$. The attribute c represents the number of computational steps performed within a unit grain, and is expressed in terms of a "basic computation unit" (BCU). The basic unit of computation chosen for granule g_c is a simple delay loop with a loop count of 1. Alternate BCUs such as a single floating-point computation could be used to highlight the floating-point performance.

Characterization of g_s:

As only the shared memory access performance is of interest here, the null characterization was chosen for the synchronization granule, *i.e.*, $g_s = \phi$. When the MAD kernels are used in the hierarchical performance framework of Figure 3.5 to measure the incremental overheads due to memory contention, a non-null characterization of g_s could be used. The handling of a workload with $g_s \neq \phi$ by the MAD kernels is described in Section 4.4.

Using the individual granule characterizations, the definition for the unit grain G can be written as the *3–tuple* of tuples.

$$G = ((p, d, s, m),\ (c),\ \phi)$$

Both homogenous and heterogenous workloads can be created by selecting different attribute values for G_t and G_c.

4.2.2 Output Metrics

The metric used to observe the trends in the memory contention performance of an input workload, as a function of the degree of interference N, is the *unit grain efficiency* $\xi_m(N)$ as defined by Eq. 3.22. A value of $\xi_m(N) = 1$ would seem to indicate that the concurrent memory access streams are independent of each other and do not encounter any conflicts at all. A value of $\xi_m(N) \ll 1$ reflects significant conflicts with the competitor processes leading to extremely high access latencies.

The *cumulative memory interference* $\Psi_m(N)$ can be computed from $\xi_m(N)$ using Eq. 3.23. Also, from Eq. 3.24, it is known that the *incremental memory interference* $\psi_m(N)$ is equal to $\Psi_m(N)$ in the case of the MAD kernels. Therefore, we have the following relationship between the efficiency and interference measures.

$$\Psi_m(N) = \frac{1 - \xi_m(N)}{\xi_m(N)} = \psi_m(N).$$

It should be emphasized that the efficiency metric is a measure of the *relative* performance of a workload with N competitors as compared to its performance with no competitors. Similarly, the interference metric is also a relative measure in that it presents the net contention overhead as a fraction of the uncontested unit grain execution time, *i.e.*, the number of unit grains that could have been processed during the time lost due to overheads. Thus both the measures are scaled in terms of the uncontested unit grain time τ. The implication of this for two workloads with identical absolute contention performance (*i.e.*, same net overheads) is that the one with the larger amount of work per unit grain (*i.e.*, larger τ) will be adjudged as the more efficient of the two.

4.3 Concurrent-Access Workloads

Concurrent-access workloads, with no lock-based synchronization within the unit grain (*i.e.*, $g_s = \phi$), were designed and used [93] to characterize the impact of concurrent memory reference patterns on the shared memory performance. The increased access latencies observed in this case are purely due to access conflicts in hardware and the overhead of maintaining the consistency of replicated data over the memory hierarchy. The workloads have been employed to measure and compare the performance of the Sequent Symmetry and the BBN TC2000 systems.

The shared data, with M elements, were allocated using the `shmalloc()` call on each machine. On the Symmetry, the data elements are interleaved across the memory modules with a interleaving granularity of 32-bytes. On the TC2000, the shared data use shared, uncached memory. If the system is configured with interleaved memory, then the shared data is interleaved. However, since the current version of the *nX* operating system does not support interleaving, the shared data is scattered across the allocated cluster instead.

We conducted experiments using a number of parameter families. Each family was designed to measure the effect of a particular grain attribute on the resultant contention and, hence, unit grain efficiency. The spectrum of input parameters included both homogenous and heterogenous settings. The heterogenous parameter families were particularly useful in revealing the interactions between concurrent read and write streams, especially on cache-based systems such as the Symmetry.

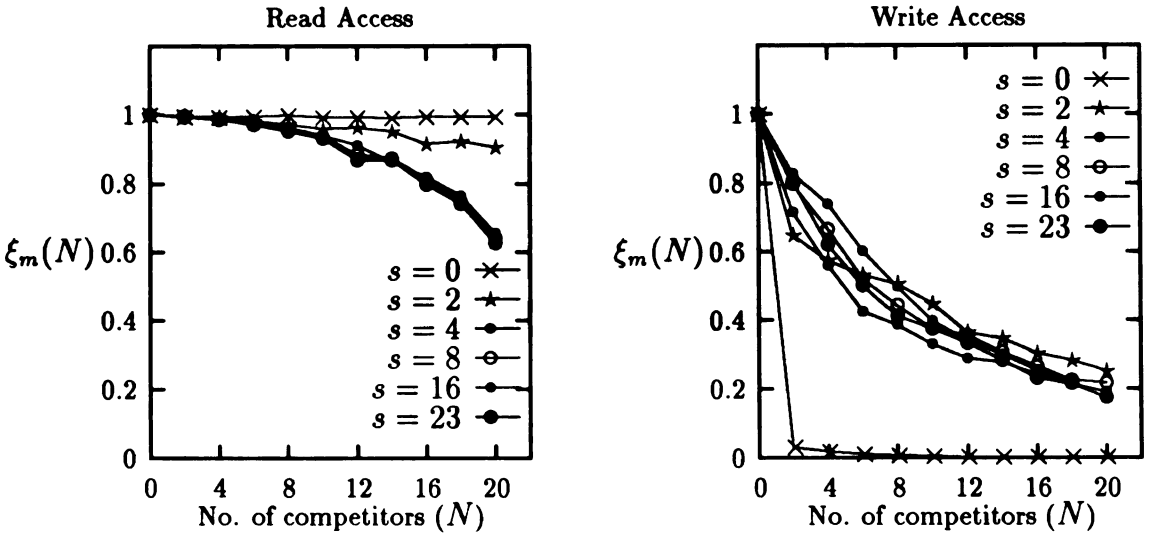
4.3.1 Homogenous Workloads

In these experiments, the attributes for the test and competitor grains were set to be identical, *i.e.*, $G_t = G_c$. Thus, the resultant performance degradation when concurrent grains with identical execution behavior compete was measured.

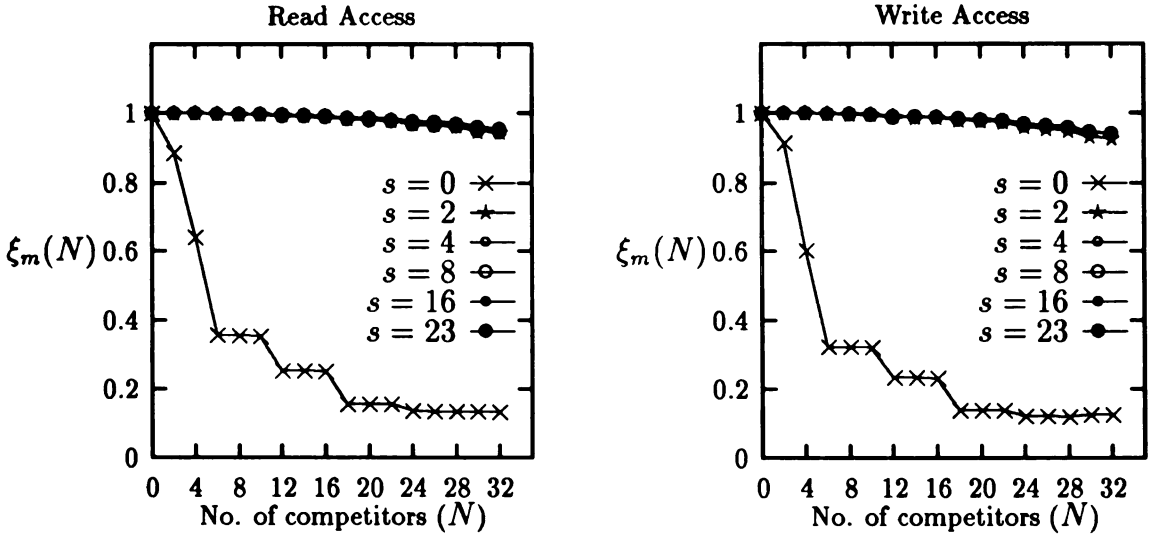
Spatial Distribution

By manipulating the stride s of shared-data access, and by choosing a value of M large enough to cause a complete sweep of all the memory modules, the effectiveness of the interleaving of the main memory system is probed. Changing the value of s, in effect, creates different spatial distributions of the memory access stream generated by each process. In Figure 4.4, the efficiency of both read and write accesses is shown. The observed efficiency $\xi_m(N)$ of a given workload provides a measure of the potential increase in the memory bandwidth for that workload by a factor of $(N+1)\xi_m(N)$. By examining the input parameters, it can be seen that all processors start their access from the shared-data element 0 (since $d = 0$) and perform subsequent accesses with identical strides. For read access, the Symmetry scales fairly for $s = 0, 2$. However, for $s \geq 4$, every access to a shared-data element results in a cache-miss (since the cache line length is 16) forcing a memory read transaction over the bus. The bus, therefore, begins to saturate at $N = 14$. For write access, a stride of 0 causes repeated writes to the same location by all processes. This results in heavy cache-invalidation traffic on the bus in addition to severe memory module contention. This is reflected by a steep drop in the efficiency of the grain as early as for $N = 2$. For other stride values, there is still cache-invalidation traffic on the bus, although not as severe as the $s = 0$ case, since every process writes to the same data locations in sequence. Hence, the memory bandwidth saturates (reflected by the extremely low value of $\xi_m(N)$) right at the outset with $N = 6$. However, by distributing the writes so that all processes do not trace the same sequence of addresses, the write performance could be much improved. The TC2000 scales well for both reads and writes for all strides except $s = 0$, which is effectively a *hot-spot* scenario [119].

The static characterization $(R_\infty, f_{1/2})$ of the memory access performance for various stride values appears in Table 4.4. The parameter $c_{1/2} = 0$ since there is no synchronization granule present in the concurrent-access workloads. The access patterns selected have the attribute $d = 0$ implying that all processors start with the first shared data element and trace the exact same sequence of elements in their respective memory reference streams (except in the *random* stride case). The standard unit of



(a) Sequent Symmetry



(b) BBN TC2000

$\vec{N}$, $M = 128K$, $G_t = G_c(p = 0/1, d = 0, \vec{s}, m = 1, c = 0)$

Figure 4.4. Effect of spatial distribution of memory access stream on performance

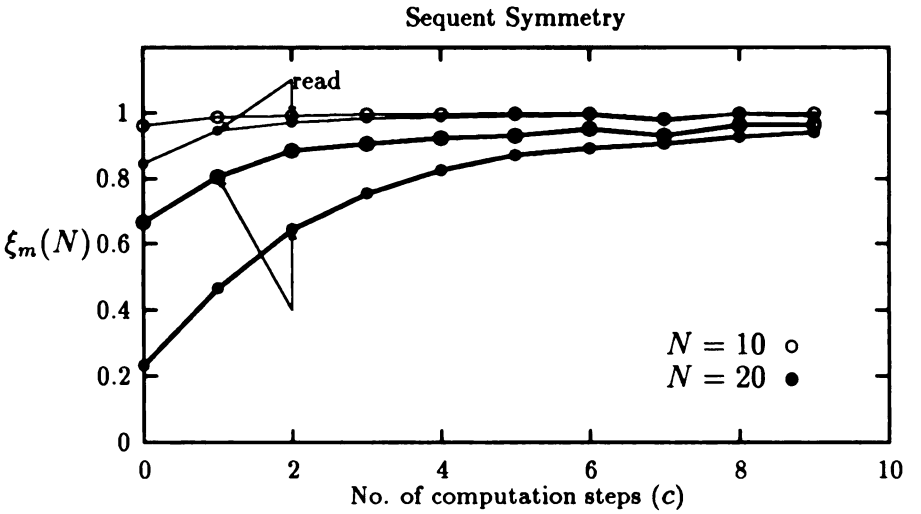
computation chosen for granule g_c is a simple delay loop with a loop count of 1. The much higher value of the parameter R_∞ for the BBN TC2000 is a consequence of its RISC instructions compared to the CISC instructions of the Sequent Symmetry as well as its faster clock rate.

Table 4.4. Static characterization parameters for a homogenous workload with $M = 128K, G_t = G_c = (g_m = (0/1, 0, \vec{s}, 1), g_c = \phi, g_s = \phi)$.

Stride of Access	Sequent Symmetry $R_\infty = 0.6 \times 10^6/second$		BBN TC2000 $R_\infty = 4.9 \times 10^6/second$	
	Memory *Read*	Memory *Write*	Memory *Read*	Memory *Write*
(s)	$f_{1/2}$	$f_{1/2}$	$f_{1/2}$	$f_{1/2}$
0	0.052	0.066	11.45	11.71
1	0.288	0.150	11.47	10.75
2	0.520	0.432	11.48	10.75
3	0.777	0.753	11.48	10.76
4	1.002	1.060	11.49	10.76
6	1.012	1.053	11.50	10.76
8	1.037	1.076	11.50	10.77
16	1.032	1.083	11.53	10.80
23	1.030	1.089	11.56	10.83
random	1.267	1.295	13.31	12.68

The $f_{1/2}$ parameter for all access strides is much higher for the BBN TC2000 pointing to the fact that there is a large disparity between the computation and shared memory access speeds on that system. Another interpretation of this fact is that for a given target rate of computation, a much larger computational granularity per shared data access is necessary on the TC2000 as compared to the Symmetry. Also noticeable in Table 4.4 is the fact that $f_{1/2}$ is relatively insensitive to the stride of access s on the TC2000. This is a consequence of the absence of data caching thus necessitating a majority of the data accesses to go out over the network incurring the worst-case latency. On the other hand, the parameter $f_{1/2}$ on the Symmetry is relatively lower for $s = 0, 1, 2, 3$ than for higher values of s. This is as a result of some

of the data accesses being satisfied by the cache for $s < 4$. For $s \geq 4$, every access results in a cache-miss as the cache line size is 16 bytes on the Symmetry.



$\vec{N}$, $M = 128K$, $G_t = G_c(p = 0/1, d = 2, s = 16, m = 1, c = 0)$

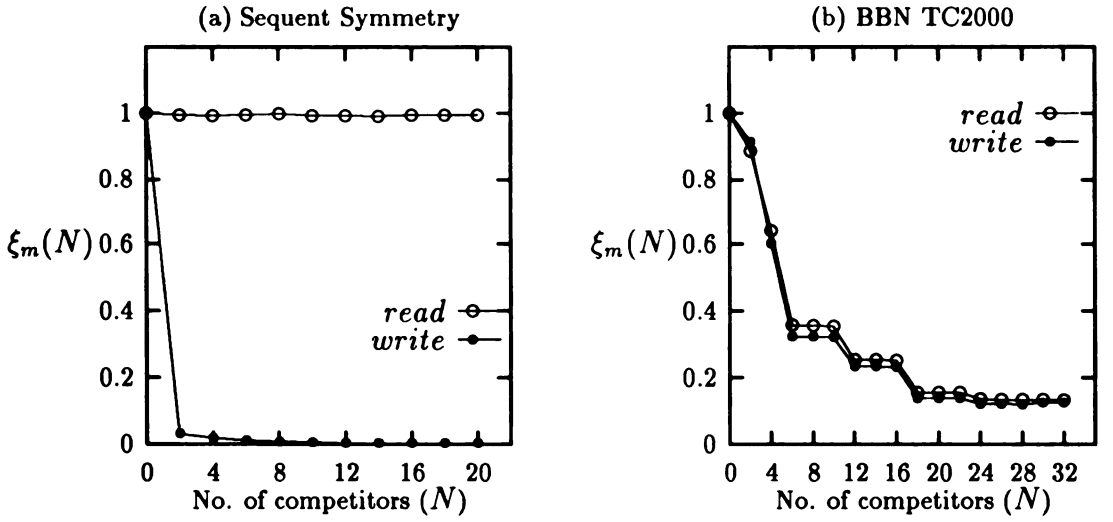
Figure 4.5. Effect of temporal distribution of memory access stream on performance

Temporal Distribution

The variation of the density of memory requests of each processor is accomplished by altering the number of computation steps performed within the computation granule g_c. This corresponds to a shared memory access followed by a subsequent interval of c units of delay with no memory access. Figure 4.5 shows the improvement in unit grain efficiency that is achieved as a consequence of increasing the length of g_c on the Symmetry. The effect is particularly striking for write operations, since the intervening computational delay without any bus accesses provides sufficient time for the cache-invalidation traffic on the bus to reach quiescence.

Memory Hot Spot

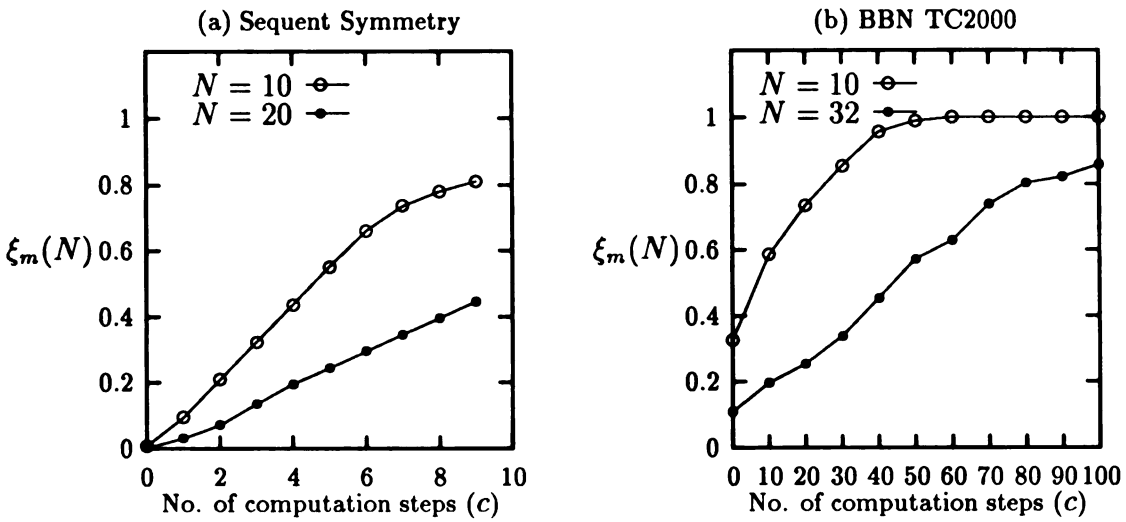
The interference profiles generated by setting $M = 1$ is indicative of the performance of the execution grain under severe contention (hot spot) conditions. In these experiments, the processors not only contend for the global interconnection network, but also for a single shared-data item. This performance is depicted in Figure 4.6. The write performance on the Symmetry degrades severely. The reads on Symmetry cache the shared-data item on the first access, and operate out of the cache on subsequent accesses, thus exhibiting no degradation. However, writes to a single location by multiple processors cause the shared location to bounce between the processor caches (*ping-pong effect*) thus generating an overwhelming amount of cache-invalidation traffic causing bus saturation. This is apparent from the extremely low value of $\xi_m = 0.025$ with just 3 processors executing concurrently, *i.e.*, $N = 2$.



$\vec{N}$, $M = 1$, $G_t = G_c(p = 0/1, d = 0, s = 0, m = 1, c = 0)$

Figure 4.6. Effect of contention for a memory location (*hot-spot*) on performance

On the TC2000 both reads and writes exhibit a severe bottleneck. To analyze the performance of hot-spot accesses, we resort to the expression for the maximum



$\vec{N}$, $M = 1$, $G_t = G_c(p = 1, d = 0, s = 0, m = 1, \vec{c})$

Figure 4.7. Effect of length of computation on *hot-spot write* performance

network throughput per processor as derived in [105]. The asymptotic maximum value r_{max} of per-processor network throughput as determined by the hot spot access request rate is given by

$$r_{max} = \frac{1}{1 + h(P - 1)} \tag{4.6}$$

where P is the number of processors (it is assumed that there are an equal number of memories), r is the number of network packets emitted per processor per switch cycle ($0 \leq r \leq 1$), and h is the fraction of memory references directed at the hot spot (*i.e.*, each processor emits packets directed at the hot spot at a total rate of rh).

Using the unit grain attributes, the net memory request rate per processor is given by $r' = m/\tau$. If t_{sw} denotes the network switch cycle time, then the memory request rate per processor per switch cycle becomes $r = mt_{sw}/\tau$. For the workload shown in Figure 4.6(b), $P = N + 1$ and all accesses are to the hot spot making $h = 1$. The maximum per-processor request rate, therefore, is limited to $r_{max} = 1/(N + 1)$ using Eq. 4.6. In other words, the following constraint should be satisfied to prevent

network saturation.

$$r = \frac{mt_{sw}}{\tau} \leq \frac{1}{N+1} \implies N+1 \leq \frac{\tau}{mt_{sw}} \tag{4.7}$$

Since $m = 1$ and $\tau = t_m$ (t_m being the average memory access time) for the workload of Figure 4.6(b), the limiting value of N is given by $N + 1 \leq t_m/t_{sw}$. As can be seen from the figure, the network begins to be saturated at $N = 18$.

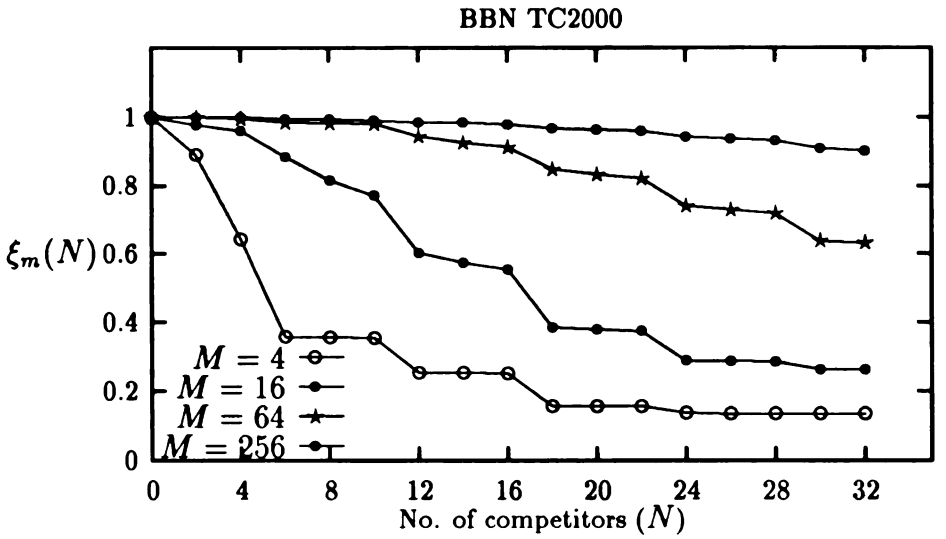
Figure 4.7 shows the improvement in the efficiency of writes to a hot spot resulting from an increase in the length of computation c within a unit grain. The increased computation time on the Symmetry allows the cache-invalidation traffic to subside between consecutive hot spot writes. On the TC2000, increasing c results in a larger value of $\tau = ct_c + mt_m$ in Eq. 4.7 thus increasing the limiting value of N at which network saturation sets in.

Size of Shared Data

By manipulating the size M of shared data, all memory references on the Symmetry can be kept in the cache, or made to flush cache on each pass through the shared-data. The TC2000, on the other hand, does not cache shared data. However, varying the shared-data size on the TC2000 revealed some interesting facts. The efficiency ξ_m was observed to behave identically for values of M from 1 through 4. Progressive improvement in ξ_m was observed for each increment of 4 in the value of M (Figure 4.8). This would imply that the scattering of shared-data by the system across cluster memory modules was done in chunks of 4 elements (*i.e.*, 16 bytes). Thus, going from $M = 4$ to $M = 16$ (and so forth) increases the number of memory modules, for which the processors contend, from 1 to 4 (and so forth) leading to a decrease in contention.

Random Memory Access

Most multiprocessor memory organizations are designed to use special techniques (such as memory interleaving, skewing) to maximize the performance of uniform memory-access patterns. But the performance of the memory hierarchy under condi-



$\vec{N}$, $\vec{M}$, $G_t = G_c(p = 0, d = 1, s = 1, m = 1, c = 0)$

Figure 4.8. Effect of shared-data size on *read* performance

tions that do not display such uniformities in memory access is also of interest. So, we measured the memory bandwidth under random access conditions, expressed as Words Accessed Randomly Per Second (WARPS), to quantify this performance. This is done using a homogenous workload consisting of only memory-access granules g_m and varying its stride attribute s randomly. The results of these tests are presented in Figure 4.9. The read and write performance on the TC2000 are comparable and appear to scale reasonably with the number of processors. The read performance on the Symmetry scales (for the number of processors used in the experiment), but the writes begin to show saturation at around 13 processors. This difference is, again, due to the extra cache-invalidation traffic injected into the bus as a result of writes to shared-data.

4.3.2 Heterogenous Workloads

Using a heterogenous workload, we have investigated the interactions that occur between concurrent read and write memory access streams. In particular, we demonstrate using the following two scenarios:

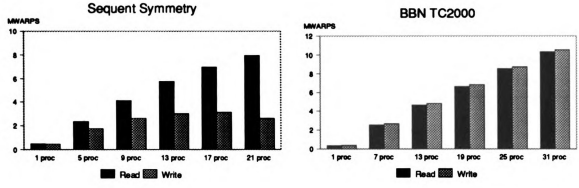


Figure 4.9. Random access performance expressed in MegaWARPS

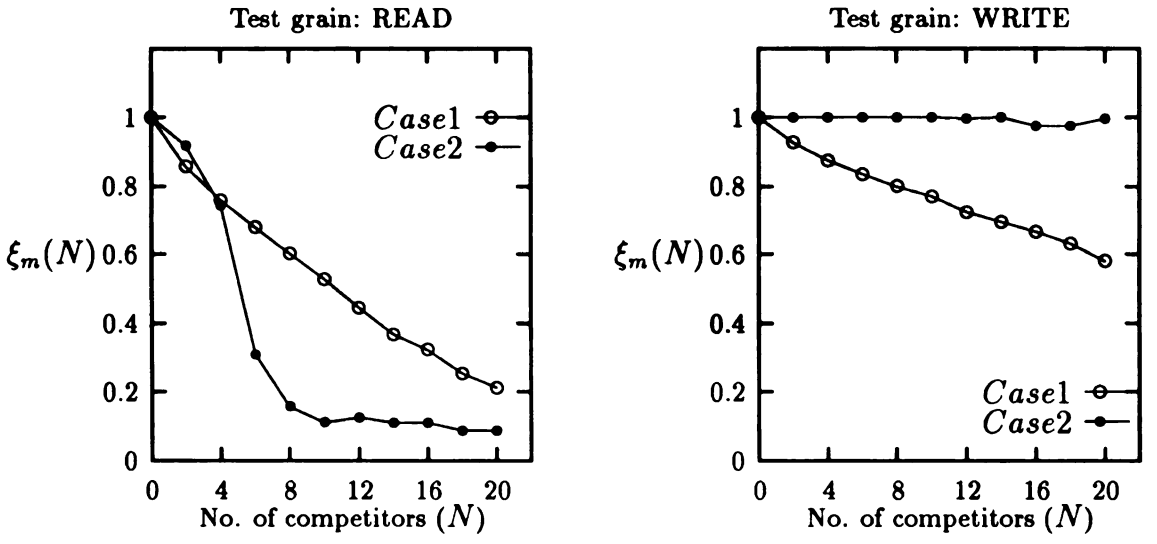
(a) *Case 1*: the test grain performs read (write) accesses to shared data with uniform stride, while the competitor grain performs write (read) accesses with random stride,

(b) *Case 2*: the test grain performs read (write) accesses to shared data with uniform stride, while the competitor grain performs write (read) accesses to a single shared (hot spot) location.

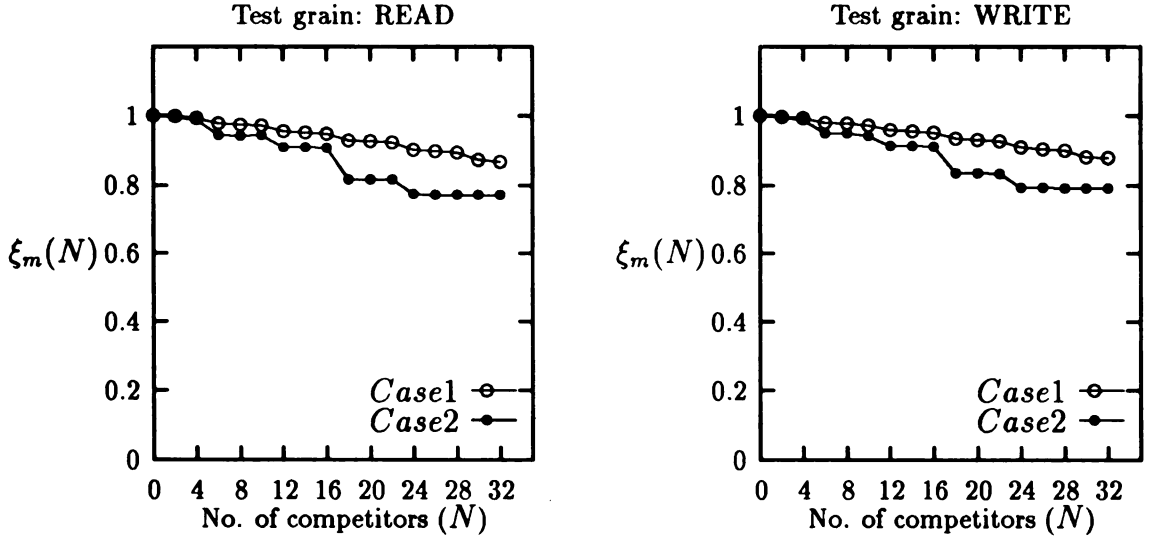
The grain efficiencies for both these cases is shown in Figure 4.10.

Performance on the Symmetry, when G_t performs read accesses, steadily deteriorates. It is markedly worse for *Case 2* due to the heavy invalidation traffic generated by the competitor grains while repeatedly writing to one shared location. When G_t executes write accesses on the Symmetry, *Case 2* corresponds to the competitor processors operating out of their private caches thus causing no bus traffic and memory contention. Hence, virtually no degradation is experienced by the test grain. The interference from the competitor grains on the TC2000 is fairly small in both cases, owing to the much higher bandwidth of the multistage network and the non-blocking switches used.

The improvement in execution efficiency of G_t on the Symmetry, for *Case 2* above, as a result of introducing computational delay is shown in Figure 4.11. Again, the cache-invalidation traffic on the bus reaches quiescence during the computational



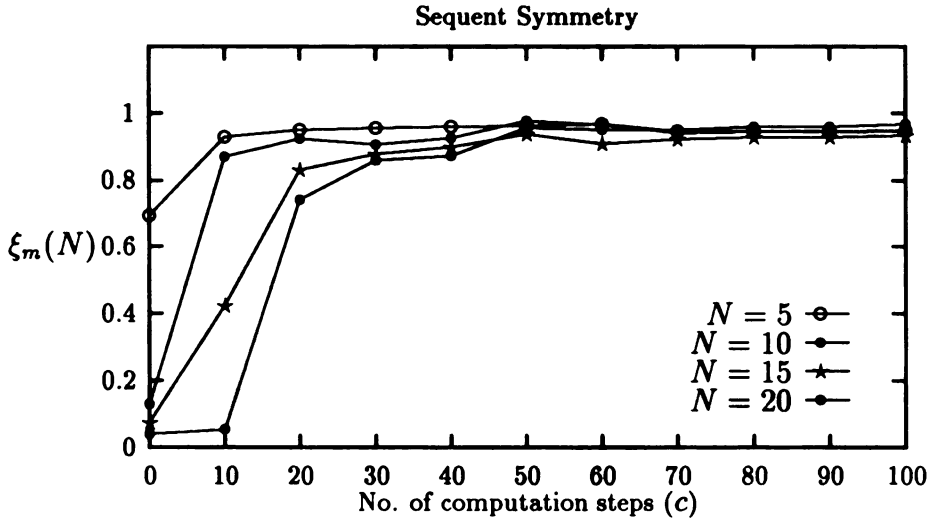
(a) Sequent Symmetry



(b) BBN TC2000

$\vec{N}$, $M = 128K$, $G_t \neq G_c$ (described in text)

Figure 4.10. Interaction between read and write memory-access streams



$\vec{N}$, $M = 128K$, $G_t(p = 0, d = 0, s = 4, m = 1, c = 0), G_c(p = 1, d = 0, s = 0, m = 1, \vec{c})$

Figure 4.11. Effect of length of computation on interference between read and write streams

delay, resulting in faster execution time. The amount of computation necessary for a given N to restore the execution efficiency to a requisite level can be determined from this characterization graph. For example, a value of $c = 20$ is needed with 10 competitors to reach an efficiency of 0.9, whereas a value of $c = 50$ is needed with 20 competitors to reach the same level of efficiency.

4.4 Dual-Mode Access Workloads

Workloads consisting of concurrent accesses to shared data (granule $g_m \neq \phi$) as well as exclusive access to shared data within critical sections (granule $g_s \neq \phi$) can be used to characterize the combined degradation of performance resulting from memory and lock contention. The MAD kernels, for such dual-mode access workloads, measure the incremental overhead (and therefore incremental interference) resulting from the dynamic nature of pure memory access conflicts. The overheads arising from the locking semantics of the critical section access is precluded from the measured per-

formance degradation by transforming the *shared lock* variable in g_s into *private lock* variable and replicating it into each processor's local memory during the execution of the MAD kernels. This leaves the memory contention behavior for shared data accesses intact, but eliminates the performance losses due to lock contention (which depends upon the implementation of the locking primitives) and queuing delay for mutually-exclusive critical section access. The lock contention and queuing delay characteristics are measured by the SAD kernels.

The incremental interference charcterization studies, including both memory and lock interference, for dual-mode access workloads are presented in Chapter 5.

4.5 Summary

The performance of the shared memory organization of a multiprocessor depends not only on the characteristics of the memory hierarchy itself, but also upon the characteristics of the memory address streams and the interaction between the two. The MAD kernels described in this chapter provide an effective testbed for characterizing the shared memory performance for a variety of memory access workloads. These kernels were employed to measure and compare the performance of the Sequent Symmetry and the BBN TC2000 multiprocessors.

The static characterization parameter R_∞ for the TC2000 was much higher than the Symmetery on account of its simpler RISC instruction set and faster clock rate. With the shared data uniformly distributed over the available memory modules, the static parameter $f_{1/2}$ was insensitive to the stride of data access on the TC2000 in the absence of caching. However, on the Symmetry, $f_{1/2}$ was related to the proportion of the data references satisfied by the cache for a given stride of access. The Symmetry, being a bus-based machine, displayed limited scalability in memory performance due to the bandwidth saturation of the bus. The onset of saturation was much faster when writes to shared-data were performed due to the additional cache-invalidation and write-back traffic on the bus. The degradation in performance was most severe when continuous writes to a single shared location were performed. On the other

hand, the TC2000 with a multistage network interconnection, was more tolerant to increasing bandwidth demands from the concurrent grains and displayed better scalability as long as the shared-data was distributed relatively evenly across the available memory modules. Performance degradation in the presence of memory hot-spots was quite severe for reads and writes alike. The read and write performance were always comparable on the TC2000.

The MAD kernels can be used either independently to perform a detailed evaluation of the sensitivity of a shared memory organization to various memory access parameters; or they can be used in conjunction with the SAD and BAD kernels to isolate the incremental overhead contribution of memory access conflicts from the total performance loss experienced by an input workload. The MAD kernels have also been used at Oak Ridge National Laboratory to perform a preliminary investigation [36] of the memory access performance of the new KSR1 multiprocessor from Kendall Square Research.

CHAPTER 5

SAD KERNELS AND SYNCHRONIZATION PERFORMANCE

On shared-memory machines, processors communicate by sharing data structures. To ensure the consistency of shared data structures, processors perform simple operations by using hardware-supported atomic primitives, and coordinate complex operations by using synchronization constructs and conventions to protect against overlap of conflicting operations. Inter-processor synchronization can become a significant performance limiting factor on large-scale multiprocessors. For the class of asynchronous multi-phase algorithms considered in this dissertation, the most prevalent form of synchronization construct used within a phase is the *critical section* that must be accessed in a mutually-exclusive manner. Entry into critical sections is usually guarded by *spin locks* and may be executed an enormous number of times in the course of a computation. Quantitative assessment of the synchronization performance of a combination of given workload and spin-lock implementation provides valuable insight into the scalability of the synchronization technique to large-scale multiprocessors.

The critical factors affecting spin lock performance and the various design implementations commonly used have been discussed in Chapter 2. The impact of critical section synchronization and the spin lock implementation used on the overall perfor-

mance of a workload is our focus in this chapter. The SAD kernels and the related framework are presented as an effective testbed to characterize the synchronization performance of a multiprocessor for a variety of workloads and spin lock implementations. The SAD kernels can be used in isolation to evaluate the sensitivity of a chosen synchronization method to various workload parameters; or they can be used in conjunction with the MAD and BAD kernels, as per the hierarchical model presented in Chapter 3, to characterize the incremental loss in performance for a given workload resulting from synchronization overheads.

5.1 Preliminary Studies

The performance studies described in this section are a part of the same suite of preliminary studies described in Section 4.1. The results presented here describe the parallel execution performance degradation in the presence of synchronization *locks*. Besides the latency and contention overhead factors arising due to memory contention described in Section 4.1, the presence of lock-based mutual exclusion operations introduces two additional sources of runtime overheads, namely, locking latency and waits due to lock conflicts. Developing a model for the lock related overheads and measuring them for an input workload is the subject of this investigation.

The parameters used to specify input workloads are (N, M, c, m, x), which have the same semantics as described in Section 4.1. However, $x \neq 0$ for the workloads used in these studies. An identical copy of the generic program based on these parameters, whose structure is illustrated in Figure 5.1, is executed by each processor. The LOCK and UNLOCK routines were implemented by us using the low-level locking primitives provided on each system. Furthermore, the LOCK routine was instrumented to count the amount of delay incurred by the invoking processor before acquiring the lock. This data was used to compile the total queuing delay encountered by a workload due to lock contention. The two performance metrics computed for each workload are *unit grain efficiency* (ξ) and *overhead factor* (Θ) as before.

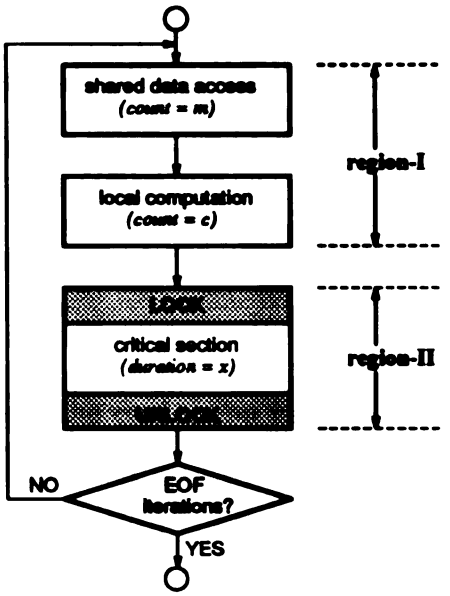


Figure 5.1. Generic structure of program executed by every processor

5.1.1 Synchronization Overhead Factors

In addition to the memory access overhead factors defined in Section 4.1, the loss in workload efficiency also includes the lock related factors, namely, the software overhead of executing the LOCK/UNLOCK routines and the queuing delay due to lock contention. If we denote the software execution overhead time O_s, and the queuing delay due to lock contention as O_q, then the expression for the total overhead factor Θ can now be written as

$$\Theta = \frac{T_G(N) - T_G(0)}{T_G(0)} = \frac{O_l + O_c + O_s + O_q}{T_G(0)} = \theta_l + \theta_c + \theta_s + \theta_q$$

which gives the two new normalized overhead components θ_s (*software factor*) and θ_q (*lock factor*). Using the definitions of ξ and Θ, it can easily be verified that

$$\xi = \frac{1}{1+\Theta} = \frac{1}{1+(\theta_l + \theta_c + \theta_q + \theta_s)} \tag{5.1}$$

which provides an indication of the trend in efficiency ξ as the overhead factor Θ varies.

Software Factor.

The pure software overhead arising out of a call to the LOCK and UNLOCK routines is a constant for a given system and a given implementation of these routines.

$$\theta_s = \frac{O_s}{\tau} = \frac{t_{lk} + t_{ul}}{\omega(1-\rho)t_c + \omega\rho t_a + x} \tag{5.2}$$

Lock Factor.

This overhead arises from the contention for a global shared lock and the consequent queueing delay to acquire the lock. Let q denote the probability that at any instant of time, a process P_i is executing in region-II of Figure 5.1 (in the absence of any lock contention). Note that a process in region-II could be in one of three possible states: waiting to acquire the lock, executing in the critical section or trying to release the lock. We can express the probability q as the proportion of the iteration time spent by process P_i in region-II.

$$q = \frac{x + 2t_w + (t_{lk} + t_{ul})}{\omega(1-\rho)t_c + 2(\omega\rho + 1)(t_a + t_l + t_w) + x + (t_{lk} + t_{ul})} \tag{5.3}$$

Since the workload of all the concurrent processes in our model is identical, the probability q is the same for all of them. Now, let W be the number of processes already in region-II when process P_i arrives at region-II. It is clear that W is a Binomial random variable with parameters N and q, *i.e.*, $W \sim B(N, q)$, since there are N other processors contending for the critical section. Hence, the expected number of processes in region-II when P_i arrives is given by

$$E[W] = Nq$$

As the implementation of our locking protocol assigns the lock to processes in a FCFS fashion, the process P_i must wait for $E[W]$ processes before it can acquire the lock. Thus, the average waiting time for the lock is given by

$$O_q = E[W] \cdot (x + t_w + t_{ul})$$

We can now express the *lock factor* as

$$\theta_q = \frac{O_q}{T(1)} = \frac{Nq(x + t_w + t_{ul})}{\omega(1-\rho)t_c + \omega\rho t_a + x} \tag{5.4}$$

5.1.2 Experimental Results

Once again, the workloads were created as per the parameter variations shown in Table 4.2. The performance data presented in this section correspond to the workload types B, C and D that include a non-empty critical section ($x \neq 0$) in the unit grain.

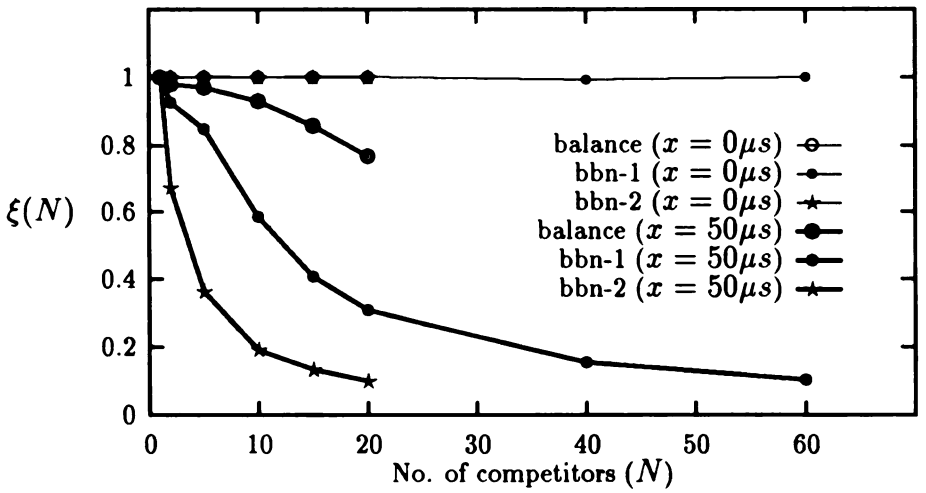


Figure 5.2. Efficiency vs. N ($M = N + 1, \omega = 100, \rho = 0$)

Workload B

Figure 5.2 illustrates the effect of introducing synchronization points into the program workload, where the synchronization occurs through a globally shared lock. Notice that even in the absence of any other shared-data reference ($\rho = 0$), the efficiency drops by more than 50% in the BBN-1 and the BBN-2. This, once again, vindicates

the existence of the hot-spot problem on the BBNs — the shared lock being the hot-spot site in this case. The dominant contributon to total overhead was found to be from θ_q. From Eq. 5.4, it can be seen that θ_q increases linearly with N, but the bulk of the delay in the expression emanates from t_w for a hot-spot lock reference. The Balance, once again, does not suffer a significant loss in efficiency from the the globally shared lock.

Workload C

The size of critical sections in parallel programs is usually kept small to alleviate the queueing delays at the critical section entry points. Since critical sections introduce serialization bottlenecks into an otherwise parallel program, the granularity of the computation performed in parallel between these synchronization points must be appropriately selected to compensate for the synchronization overhead. Otherwise, the effective speedup gained from parallelization is sacrificed.

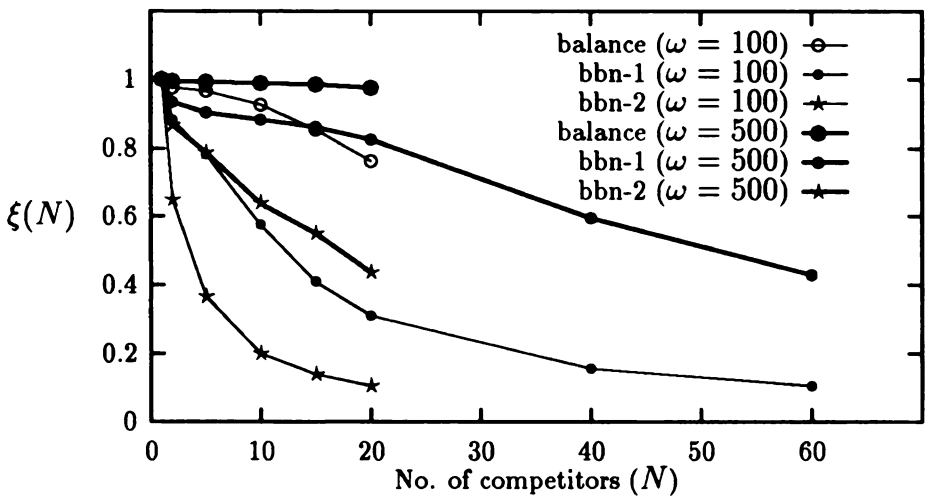


Figure 5.3. Efficiency vs. N ($M = N + 1, \rho = 0.1, x = 30\mu s$)

Figure 5.3 shows how efficiency is affected when the program granularity is changed

from $\omega = 100$ to $\omega = 500$. As can be seen from the graph, the efficiency improves for all the three systems when granularity is increased, keeping other parameters fixed. At $N = 20$, the increase in efficiency is approximately 24% for the Balance, 48% for the BBN-1 and 36% for the BBN-2. A key reason for this improvement can be ascribed to the fact that process executions get staggered in region-I (Figure 5.1), thus reducing the probability that the arrival of two processes at the critical section coincide. Examining Eq. 5.3 for this probability q, it can be seen that an increase in ω increases the denominator thus yielding a smaller value of q. That, in turn, produces a smaller θ_q in Eq. 5.4.

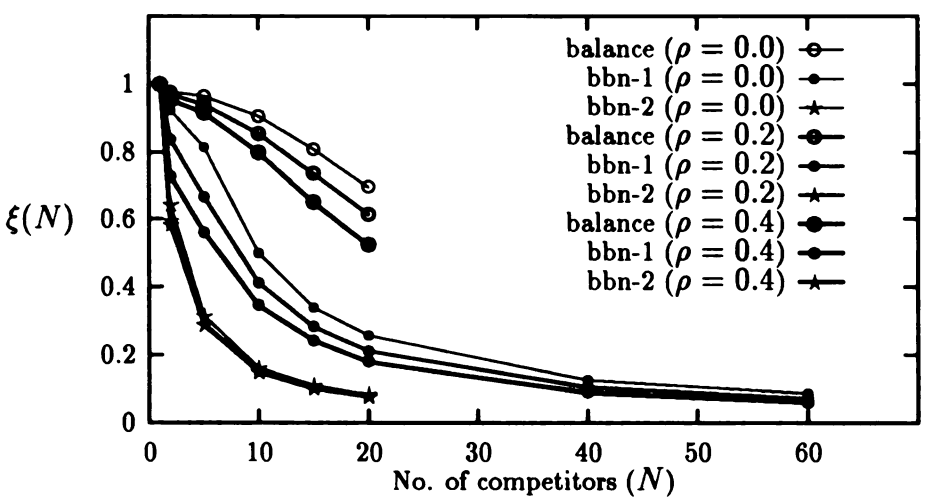


Figure 5.4. Efficiency vs. N ($M = N + 1, \omega = 100, x = 100\mu s$)

Figure 5.4 illustrates the loss in efficiency due to increased contention as N increases for three different values of shared-access fraction ρ. In the case of the Balance, increasing ρ leads to greater contention for the bus bandwidth, thus yielding a higher value of the contention factor θ_c. Hence, a steady decrease in efficiency is observed as ρ is increased. In the BBNs, however, the additional deterioration in efficiency by increasing the value of ρ from zero to a positive quantity is not so striking. This,

once again, points to the fact that the performance degradation due to the shared lock hot-spot when $\rho = 0.0$ still remains the dominant cause for overhead at $\rho = 0.2$ and $\rho = 0.4$.

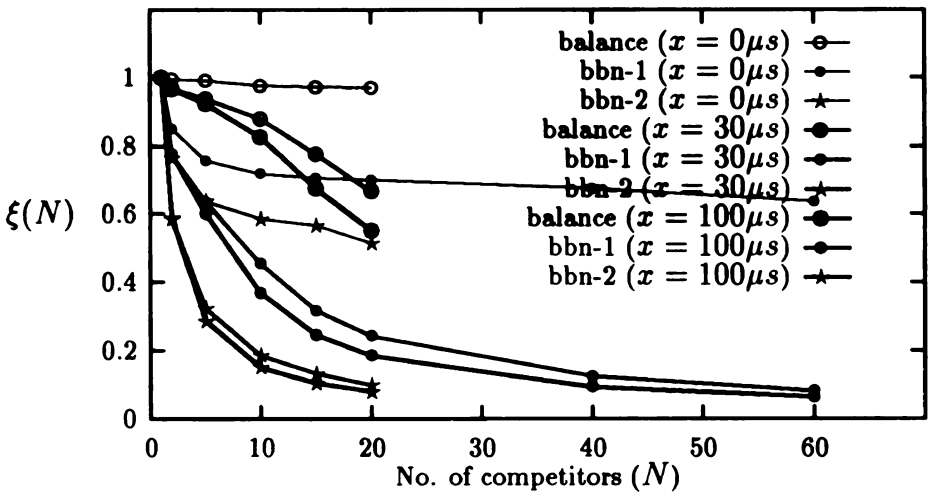


Figure 5.5. Efficiency vs. N ($M = N + 1, \omega = 100, \rho = 0.3$)

The influence of critical section length on the overall efficiency of a program workload is plotted in Figure 5.5. The efficiency suffers on all the three systems as the critical section length x is increased. Increasing the value of x results in a process having to wait for the shared lock for a longer time on the average, as indicated by Eq. 5.4. However, in the BBNs, the extent of loss in efficiency in going from $x = 0$ to $x = 30$ is far more significant than that from $x = 30$ to $x = 100$. An increase in θ_q proportional to x, as predicted by Eq. 5.4, does not explain this non-uniformity. The additional overhead, that causes this non-uniform behavior, is due to the introduction of a memory hot-spot at the site of the shared lock for the critical section.

Workload D

This workload was designed to study the effect of the input parameters on the individual overhead components. Figure 5.6 plots the individual overhead factors for the three systems as the degree of concurrency (N) is varied under a fixed shared-access fraction (ρ) and critical section length (x). As explained earlier, the software overhead is a fixed and constant quantity. The latency factor θ_l also remains fixed here as it depends only on the proportion of shared accesses ρ. The θ_c and θ_q components increase steadily with n for all the three systems, as predicted by Eqs. 4.2 and 5.4.

For small critical section lengths, a process spends a greater proportion of its time in region-I of Figure 5.1 and, hence, the execution profile of the concurrent processes gets evenly distributed in region-I. However, as x increases, the lock factor θ_q begins to dominate as shown in Figure 5.7. This is an outcome of the two-fold effect that the critical section duration has on θ_q in Eq. 5.4. An increase in the length of the critical section not only increases the x term, but also leads to an increase in the probability q. In fact, as the hardware technology gets faster (*i.e.*, t_c, t_a and t_l become smaller), the value of q increases even more for a given computational granularity ω (Eq. 5.3), further accentuating the θ_q component. This fact is apparent from the θ_q curve for BBN-2 which uses a faster technology. To compensate for the decrease in t_c, t_a and t_l, the computation granularity ω must be increased to prevent an increase in the value of q. On the Balance, the unit times t_c and t_a are very large causing the term $\omega(1-\rho)t_c + 2(\omega\rho+1)(t_a+t_w)$ in the denominator of Eq. 5.3 to overwhelm x in the range under consideration. This results in an extremely small value of q thus making θ_q negligible. The influence of x on θ_c is only in as much as the creation of a hot-spot effect at the global lock on the BBNs.

Figure 5.8 shows the individual overhead components on the three systems as a function of the shared-access fraction ρ. Observe that the lock factor, θ_q, is the largest overhead component on the BBNs, whereas the contention factor, θ_c, is the largest on the Balance. The presence of a separate dedicated bus for shared lock access in the Balance segregates the contention for lock access from those for other shared-memory access. Increased number of shared-memory accesses, as dictated by increasing ρ,

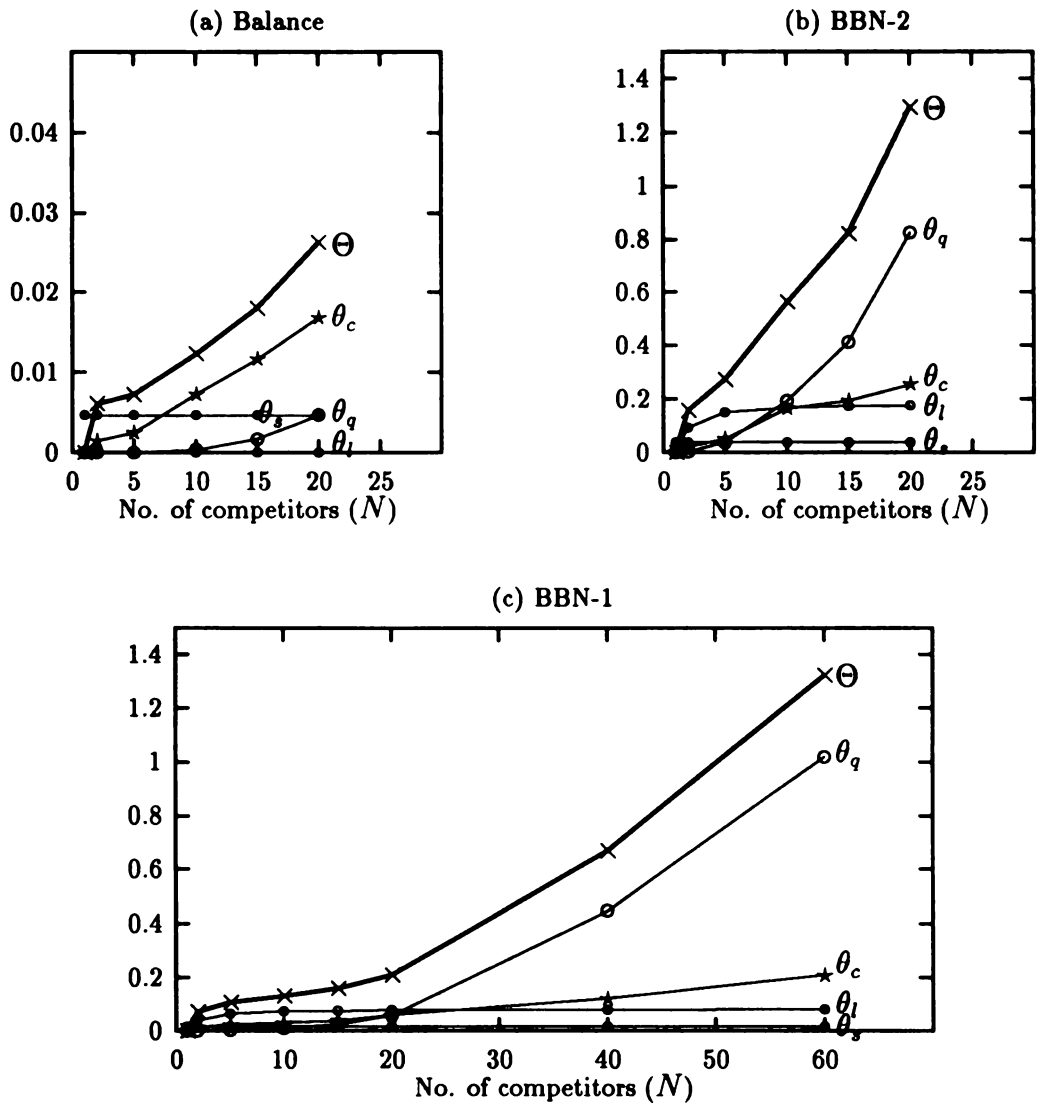


Figure 5.6. Overhead components vs. N ($M = N + 1, \omega = 500, \rho = 0.1, x = 30\mu s$)

leads to greater contention for the system bus bandwidth and a consequent increase in θ_c. Changing the value of ρ also changes the fraction of time spent by a process in region-I (Figure 5.1). The exact nature of this change on any system is governed by the relative measures of t_c and t_a+t_l on that system. Also, note that the normalization factor $T_G(0)$, too, depends on the value of ρ. If $t_c > t_a+t_l$, then increasing ρ results in a smaller proportion of time in region-I and a smaller value of $T_G(0)$. The results are just the opposite if $t_c < t_a + t_l$. Hence, the interpretation of the plots in Figure 5.8 is closely related to the ratio of the computational to memory-access speeds of the individual systems.

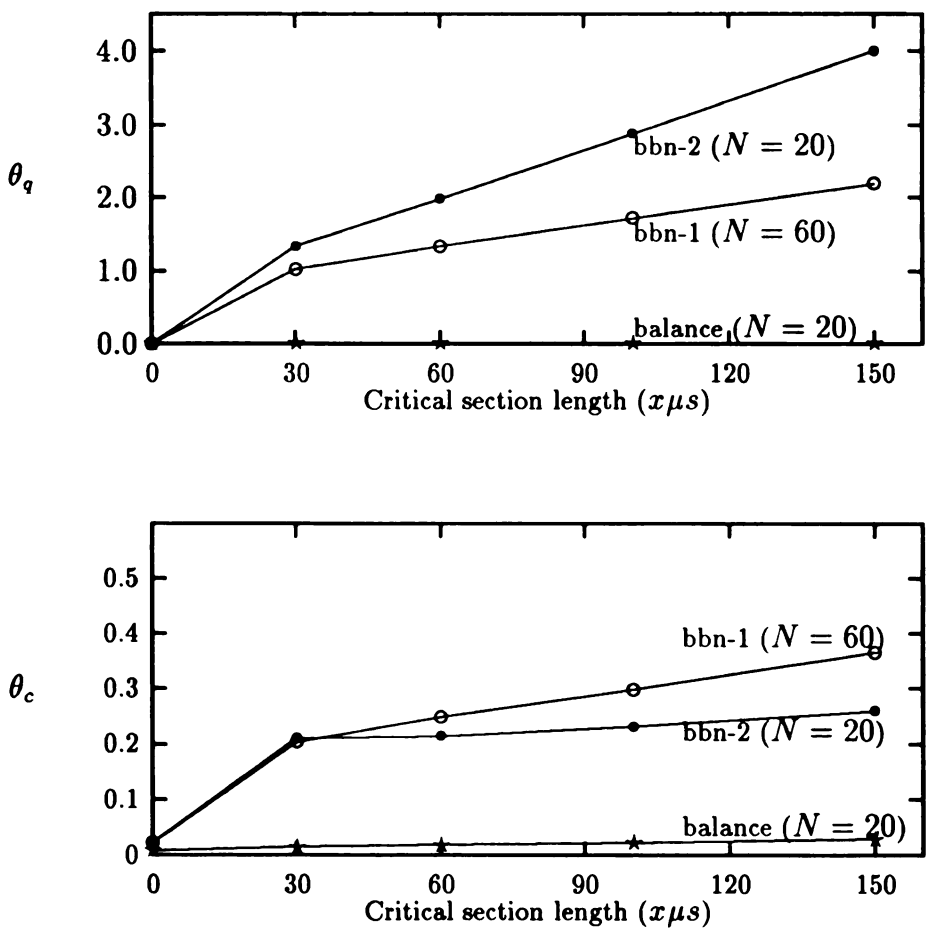


Figure 5.7. Overhead components vs. x ($M = N + 1, \omega = 500, \rho = 0.1$)

All the performance figures presented so far have been normalized quantities. However, in order to provide a feel for the absolute speed of each system, we also enumerate some real execution times. Table 5.1 shows the unnormalized values for execution times as ρ varies with a fixed parameter setting of $M = N + 1 = 20$, $\omega = 500$ and $x = 50\mu s$. It immediately reveals that the BBN-2 has the fastest and the Balance has the slowest execution times of the three systems. Table 5.2 documents the unnormalized overhead times corresponding to the same workload as represented in Table 5.1. The software overhead time, $O_s = t_{lk} + t_{ul}$, is not included in this table. It is a fixed quantity for a given system and can be found from Table 4.1.

Table 5.1. Actual execution times ($M = N + 1, \omega = 500, x = 50\mu s$)

ρ	BBN-1 ($N = 60$)			BBN-2 ($N = 20$)			Balance ($N = 20$)		
	$T_G(0)$ (μs)	$T_G(N)$ (μs)	ξ	$T_G(0)$ (μs)	$T_G(N)$ (μs)	ξ	$T_G(0)$ (μs)	$T_G(N)$ (μs)	ξ
0.1	4627.6	11289.6	0.41	789.7	2115.2	0.37	17748.0	18303.5	0.97
0.2	4208.9	11756.3	0.36	770.8	2199.8	0.35	16086.2	16834.0	0.95
0.3	3815.3	11843.3	0.32	766.1	2256.3	0.34	14494.5	15287.5	0.95
0.4	3455.3	11906.2	0.29	744.9	2273.7	0.33	12984.4	13849.3	0.94

Table 5.2. Actual overhead times ($M = N + 1, \omega = 500, x = 50\mu s$)

ρ	BBN-1 ($N = 60$)			BBN-2 ($N = 20$)			Balance ($N = 20$)		
	O_l (μs)	O_c (μs)	O_q (μs)	O_l (μs)	O_c (μs)	O_q (μs)	O_l (μs)	O_c (μs)	O_q (μs)
0.1	371.2	1040.4	5178.3	138.2	114.5	1043.2	0.0	367.6	104.8
0.2	742.4	1357.2	5375.8	276.4	221.2	901.7	0.0	537.4	127.3
0.3	1113.6	1605.7	5236.6	414.6	170.5	875.5	0.0	567.9	142.3
0.4	1484.8	1888.9	5005.2	552.8	238.7	707.7	0.0	661.9	119.8

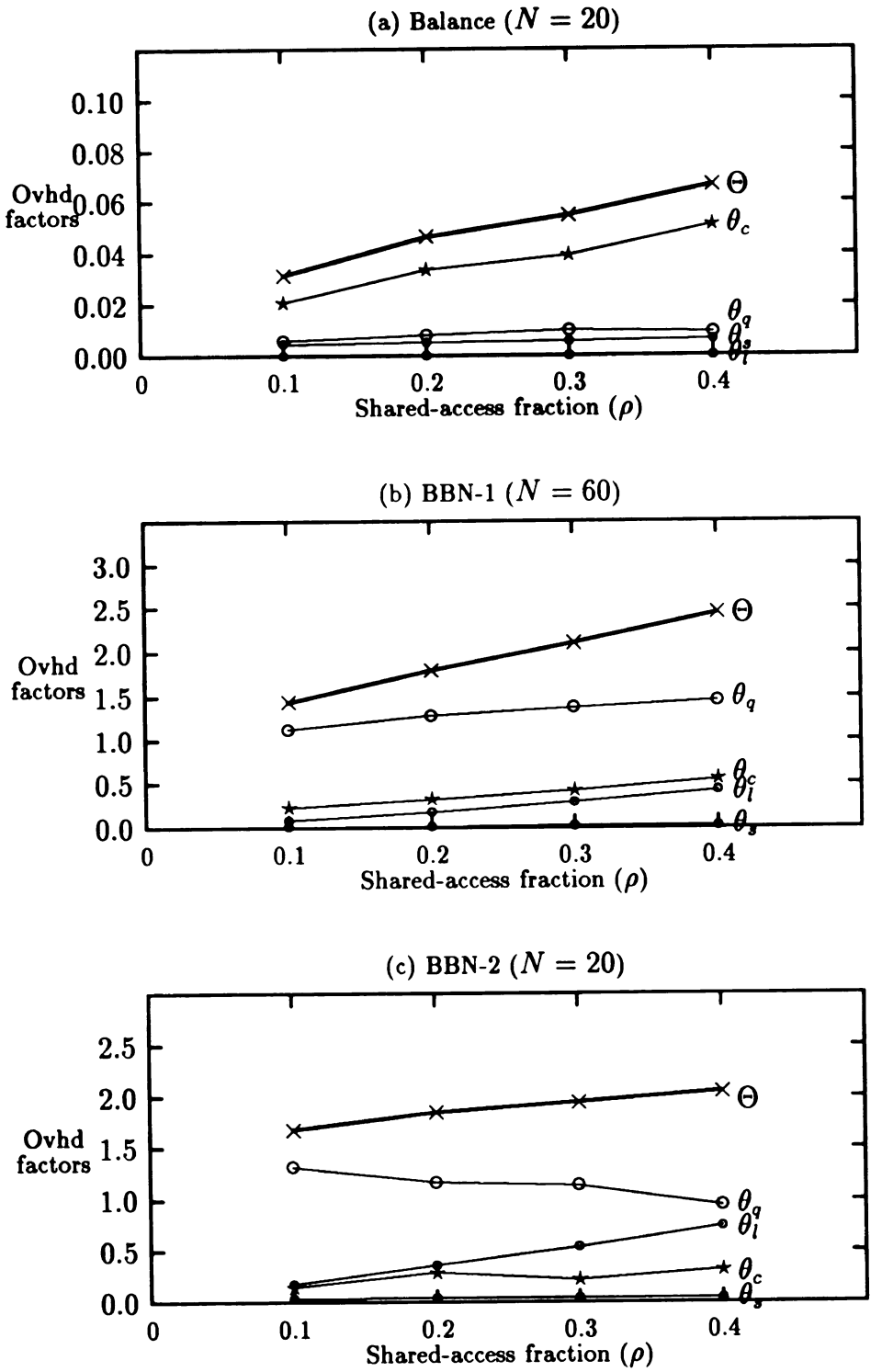


Figure 5.8. Overhead components vs. ρ ($M = N + 1, \omega = 500, x = 50\mu s$)

5.2 SAD Workload Parameters

Although the performance model presented in Chapter 3 can be adapted to evaluate any synchronization mechanism, the form of inter-processor synchronization in granule g_s chosen for the class of algorithms under consideration in this thesis is the *critical section* (CS). The critical section is guarded by a pair of LOCK/UNLOCK operations (Figure 5.9), implemented as spin locks, to ensure mutual exclusion. Besides the performance of a spin lock implementation itself (*i.e.*, latency and throughput), an important criterion for any lock-based synchronization mechanism in the presence of many competing processors is the impact it has on other components of grain execution and *vice versa*. This mutual interference can be acutely detrimental to application performance when execution of the code within a critical section is prolonged as a result of interference from other concurrent operations, which in turn causes the serial bottleneck to become more pronounced leading to a greater number of spinning processors waiting for the lock to be released. The family of SAD kernels are designed to measure this mutual interference as well as the performance of the spin lock implementation itself.

5.2.1 Unit Grain Characterization

As was the case for the MAD kernels, due to the scarcity of data on real workloads, a flexible parametric model of unit grain characterization is again chosen. The attributes selected for the unit grain should help not only in evaluating the selected spin lock implementation, but also in measuring the waiting time on account of lock contention and the interference between code executed within and outside of the critical section. The unit grain characterization selected for this purpose is summarized in Table 5.3.

Characterization of g_m:

The same four-attribute characterization of the shared-data access granule g_m as used for the MAD kernels is chosen for the study of the synchronization behavior

Table 5.3. Unit grain attributes for studying synchronization behavior

Granule	Attribute	Meaning
common	N	number of competitor processors
	M	number of shared data elements
g_m	p	probability of write access to shared memory
	d	initial distance of concurrent address streams
	s	stride of memory access
	m	number of shared memory accesses per granule
g_c	c	number of basic computation units (BCUs)
g_s	c_s	number of computation steps in CS
	m_s	number of memory accesses in CS
	p_s	probability of a write access in CS

with the SAD kernels. Therefore, $g_m = (p, d, s, m)$ where the attributes have the same semantics as discussed for the MAD kernels.

Characterization of g_c:

The single-attribute characterization of the computation granule g_c as used for the MAD kernels is also chosen here. Therefore, $g_c = c$ where the attribute c has the same meaning as in the case of the MAD kernels.

Characterization of g_s:

Two factors related to the synchronization operation that have a significant influence on the speed of execution of a unit grain are the frequency and length of the critical section. Since the durations of the granules g_m and g_c indirectly determine the frequency of occurrence of the synchronization granule, we characterize g_m with a *3-tuple* of additional attributes necessary to control the duration of the critical section and the shared-data access pattern within it.

$$g_s = (c_s, m_s, p_s)$$

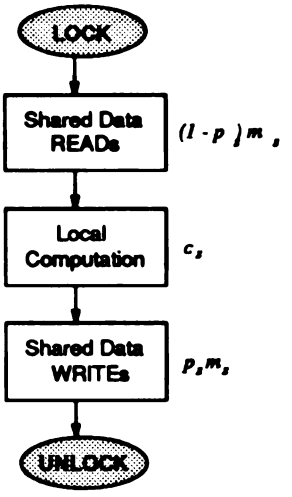


Figure 5.9. Critical section structure

The value of the attribute c_s indicates the number of computational steps performed within the critical section, using processor private data, expressed in exactly the same delay unit as in g_c. This time interval is marked by the fact that there is no access to the shared-memory, and thus no contribution to the global interconnection network traffic, by the processor executing the granule. The attributes m_s and p_s together define the nature of memory accesses performed from within the critical section. The total number of shared memory references within the critical section is given by m_s, while p_s indicates the fraction of these references to shared data that are *write* operations. All the shared data accesses within this granule are assumed to go out over the global interconnect thus contributing to network traffic.

Using the individual granule characterizations, the definition for the unit grain G can be written as the *3-tuple* of tuples.

$$G = ((p, d, s, m),\ (c),\ (c_s, m_s, p_s))$$

5.2.2 Output Metrics

The metric used to observe the trends in lock contention and the serialization loss due to synchronization for an input workload, as a function of the degree of interference

N, is the *unit grain efficiency* $\xi_s(N)$ as defined by Eq. 3.22. A value of $\xi_s(N) = 1$ would seem to indicate that the concurrent processes do not mutually interfere at all. Furthermore, the relative disposition of the computation performed within and outside the CS is such that mutually-exclusive accesses to the CS do not result in any waits. A value of $\xi_s(N) \ll 1$ reflects considerable lock contention and execution serialization to access the CS.

The *cumulative lock interference* $\Psi_s(N)$ can be computed from $\xi_s(N)$ using Eq. 3.23. Also, from Eq. 3.25, it is known that the *incremental lock interference* $\psi_s(N)$ is equal to the difference of $\Psi_m(N)$ and $\Psi_s(N)$ for a given workload. Therefore, we have the following relationship between the efficiency and interference measures.

$$\Psi_s(N) = \frac{1 - \xi_s(N)}{\xi_s(N)} = \psi_m(N) + \psi_s(N).$$

In the case of exclusive-access workloads, since concurrent shared memory accesses are non-existent ($g_m = \phi$), there is no incremental memory interference, *i.e.*, $\psi_m(N) = 0$, thus making $\psi_s(N) = \Psi_s(N)$. For dual-mode access workloads, both the incremental overhead components would be present.

It should be emphasized that the efficiency metric is a measure of the *relative* performance of a combination of workload and spin lock implementation with N competitors as compared to its performance with no competitors. Therefore, although suitable for characterizing the behavior of a given spin lock implementation with respect to the different workload parameters, it does not facilitate an effective comparison between two different implementations. The absolute unit grain execution times, $T_G(N)$, should be used instead for this purpose.

5.2.3 Lock Implementations Studied

We have chosen three spin lock implementations on each of the target systems studied. The first one is the *native* LOCK/UNLOCK operations provided on each system (referred to as the NAT lock) to support parallel programming. This support is in the form of function calls in a parallel programming library as shown in Table 5.4.

Table 5.4. Native lock support on each machine

Procedure	Sequent Symmetry	BBN TC2000
InitLock	`s_lock_init (lock)`	`lock =: CLEAR`
GetLock	`s_lock (lock)`	`UsLock (lock,delay)`
ReleaseLock	`s_unlock (lock)`	`UsUnlock (lock)`

The other two implementations selected represent somewhat two extremes of busy-waiting efficiency. The *test-and-test-and-set lock* (referred to as the TAS lock) spins by reading the shared lock variable until it becomes free, and then attempts a test-and-set operation to acquire the lock. The simple pseudo-code for it is listed in Table 5.5. On machines with coherent caches, the spin on read eliminates interconnection network traffic. But upon release of the lock, several spinning processors rush to grab the lock simultaneously thus inundating the interconnect with test-and-set requests. This problem is especially acute on systems with invalidation-based cache coherence where the flood of invalidations as a result of the test-and-set operations cause the shared-lock location to bounce from one processor cache to another before quiescence sets in. This effect has also been called the *ping-pong* effect. On architectures without coherent caches, even the spin on read generates heavy network traffic in addition to creating a memory hot-spot. The TAS and NAT implementations are almost identical on the Symmetry. But on the TC2000, NAT incorporates a fixed delay between consecutive polls of the shared lock variable by a processor unlike TAS.

Table 5.5. Pseudo-code for the TAS lock

Procedure	Implementation
InitLock	`lock := CLEAR`
GetLock	`while (lock = BUSY or test_and_set (lock) = BUSY)`
Releaselock	`lock := CLEAR`

The last spin lock implementation chosen is a *list-based* queueing lock devised by Mellor-Crummey and Scott [89] (referred to as the MCS lock) with the following characteristics:

- guarantees FIFO ordering of lock acquisitions;
- spins on locally-accessible flag variable only; and
- works equally well (requiring only $O(1)$ network transactions per lock acquisition) on machines with and without coherent caches.

Figure 5.10 shows the algorithm for this lock. Each processor using the lock allocates a **Qnode** record containing a queue link and a boolean flag. Each processor employs one additional temporary variable during the **GetLock** operation. Processors holding or waiting for the lock are chained together by the links. Each processor spins on its own locally-accessible flag. The lock itself contains a pointer to the **Qnode** record for the processor at the tail of the queue (or the value **nil** if the lock is not held). Each processor in the queue holds the address of the record for the processor behind it — the processor it should resume after releasing the lock. **Compare-And-Swap** enables a processor to determine if it is the only processor in the queue, and if so remove itself correctly, as a single atomic action. The spin in **GetLock** waits for the lock to become free. The spin in **ReleaseLock** compensates for the timing window between the **Fetch-And-Store** and the assignment to **predecessor↑.next** in **GetLock**. Both spins are local to the processor.

Figure 5.11, parts (a) through (e), illustrates a series of **GetLock** and **ReleaseLock** operations. The lock itself is represented by a box containing an 'L' in it. The other rectangles are **Qnode** records. A box with a slash through it represents a **nil** pointer, and non-nil pointers are shown as directed arcs. The state of each processor in the queue (R: running, B: blocked, E: exiting from critical section) is indicated along with its identification within each **Qnode** record. In (a), the lock is free. In (b), processor 1 has acquired the lock and is running. In (c), two more processors have entered the queue and are blocked spinning on their **locked** flags. In (d), processor 1 has completed and has changed the **locked** flag of processor 2 so that it is now

```
type Qnode = record
    next : ↑Qnode;
    locked : Boolean;
  end;
type Lock = ↑Qnode;

{ Parameter "Q" below points to a Qnode record allocated in shared memory
locally accessible to the invoking processor }

procedure GetLock (L : ↑Lock; Q : ↑Qnode)
    Q↑.next := nil;
    predecessor : ↑Qnode := Fetch-And-Store (L, Q);
    if predecessor ≠ nil then  {queue was non-empty}
      Q↑.locked := TRUE;
      predecessor↑.next := Q;
      while Q↑.locked = TRUE do;   {spin}

procedure ReleaseLock (L : ↑Lock; Q : ↑Qnode)
    if Q↑.next = nil then  {no known successor}
      if Compare-And-Swap (L, Q, nil) then
        return;   {returns if and only if it swapped}
      while Q↑.next = nil do;   {spin}
    Q↑.next↑.locked := FALSE;
```

Figure 5.10. Pseudo-code for the MCS list-based queuing lock

running. In (e), processor 2 has completed and has unblocked processor 3. If no more processors enter the queue in the immediate future, the lock will return to the situation in (a) when processor 3 completes its critical section.

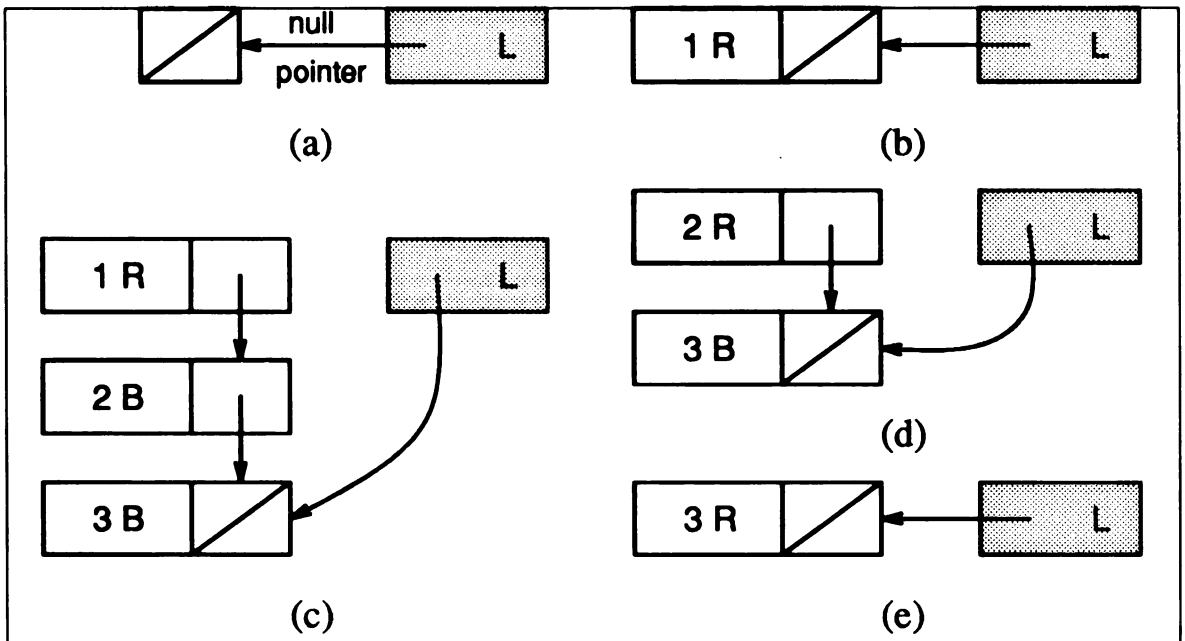


Figure 5.11. Working of the MCS list-based queuing lock

We emphasize that we have deliberately chosen only a few spin lock implementations for the purpose of demonstrating the effectiveness of the evaluation methodology. There are a number of other spin-lock implementations available in the literature [6, 89]. Our selection should not be construed as a definitive indication of their relative merits.

5.3 Exclusive-Access Workloads

Exclusive-access workloads, with synchronization in the form of lock-based mutually-exclusive access to a critical section within the synchronization granule g_s, were designed and used [94] to characterize the impact of serialization of execution, and

lock latency and contention on a shared memory multiprocessor performance. The increased unit grain execution times observed in this case are purely due to the software overhead of executing the locking primitives, serialization of access to the CS, and lock contention. The workloads have been employed to measure and compare the performance of three lock implementations on the Sequent Symmetry and the BBN TC2000 systems.

Table 5.6. Latency of locks used in the SAD experiments

Spin Lock	Sequent Symmetry	BBN TC2000	
		Lock Local	Lock Remote
NAT	7.4 μs	4.3 μs	12.1 μs
TAS	6.1 μs	1.8 μs	8.0 μs
MCS	10.1 μs	8.6 μs	15.8 μs

An important fundamental criterion for any lock implementation is its *latency*—the time it takes to acquire and release it in the absence of competition. Table 5.6 shows this measure for the locks used in our study. On the TC2000, since a dichotomy in the memory hierarchy exists, the latency of the lock depends on its location with respect to the processor invoking it. Thus, the latency when the lock is situated in a processor's local memory and a remote memory are shown under the columns "Local" and "Remote", respectively. The results presented in this section pertain to the case of the shared-lock being remote to all processors. The half-performance lock factor $c_{1/2}$ for the various lock implementations is given in Table 5.7. Once again, the large disparity between processor speed and lock access latency on the TC2000 is reflected by its high values of $c_{1/2}$.

A critical section synchronization enforces a serialization of execution on the participating processors, thus causing a loss of parallelism. Since only one processor can execute in the CS at any time, all other processors waiting for mutually exclusive access to the CS spend time idling, wasting potentially productive computational cy-

Table 5.7. Half-performance lock factor $c_{1/2}$ for different lock implementations

Spin Lock Type	Sequent Symmetry ($R_\infty = 0.6 \times 10^6/second$) $c_{1/2}$	BBN TC2000 ($R_\infty = 4.9 \times 10^6/second$)	
		Lock Local $c_{1/2}$	Lock Remote $c_{1/2}$
NAT	4.45	21.07	59.31
TAS	3.67	8.82	39.22
MCS	6.08	42.16	77.45

cles. Further, the implementation technique used for the spin lock guarding the CS can also adversely impact performance beyond what is dictated by serialization due to excessive lock contention and interconnection network traffic generated [6]. The net execution efficiency $\xi_s(N)$ observed for a combination of input workload and spin lock implementation is a result of both of the above factors. Let us suppose that the net observed efficiency can be decomposed into two factors as follows:

$$\xi_s(N) = \alpha(N) \cdot \beta(N)$$

where $\alpha(N)$ represents the loss in parallel work due to serialization of CS access, and $\beta(N)$ represents the loss in performance due to lock implementation considerations. The factor $\alpha(N)$, called *structural efficiency*, signifies the influence of the unit grain structure on the overall synchronization performance. The factor $\beta(N)$, called *spin lock efficiency*, on the other hand, signifies the impact of the spin lock implementation methodology on the overall synchronization performance.

The efficiency component $\beta(N)$ is difficult to quantify analytically since it is a complex function of the runtime interactions occuring between concurrent processes. However, we can derive an approximate relation for $\alpha(N)$ for the case of deterministic homogenous workloads. For now, let us assume that spin lock implementation is 100% efficient, *i.e.*, $\beta(N) = 1$. This implies that $\xi_s(N) = \alpha(N)$. For a homogenous workload, since all the $N + 1$ processors are executing identical unit grains, they will soon become "skewed" so that they attempt their CS access at different times. Thus,

there will be no CS contention if N processors have time to complete their g_s granule while the $(N+1)^{th}$ is processing granules g_m and g_c, that is, if $\tau_m + \tau_c \geq N\tau_s$. Otherwise, contention occurs and the waiting time for each unit grain is $N\tau_s - (\tau_m + \tau_c)$. Hence, the unit grain execution time is given by

$$T_G(N) = \tau_m + \tau_c + \tau_s + t_{queue}$$

where

$$t_{queue} = \begin{cases} N\tau_s - (\tau_m + \tau_c) & \text{if } \tau_m + \tau_c < N\tau_s \\ 1 & \text{otherwise} \end{cases} \tag{5.5}$$

It should be noted that if concurrent memory accesses are present in the granule g_m, then the overhead due to memory access contention is not included in the total unit grain time. In other words, memory accesses in g_m are assumed to be conflict-free for this derivation. The MAD kernels measure the extent of performance degradation due to memory access contention. If we define the *serialization ratio* λ as

$$\lambda = \frac{\tau_m + \tau_c}{\tau_s} \tag{5.6}$$

then the structural efficiency can be expressed using Eqs. 5.5 and 5.6 as

$$\alpha(N) = \frac{\tau_m + \tau_c + \tau_s}{T_G(N)} = \begin{cases} \frac{1+\lambda}{N+1} & \text{if } \lambda < N \\ 1 & \text{if } \lambda \geq N \end{cases} \tag{5.7}$$

In reality, the spin lock implementation will not be 100% efficient. Therefore, if we remove this restriction, then the net unit grain execution efficiency can be expressed as

$$\xi_s(N) = \begin{cases} \left(\frac{1+\lambda}{N+1}\right)\beta(N) & \text{if } \lambda < N \\ \beta(N) & \text{if } \lambda \geq N \end{cases} \tag{5.8}$$

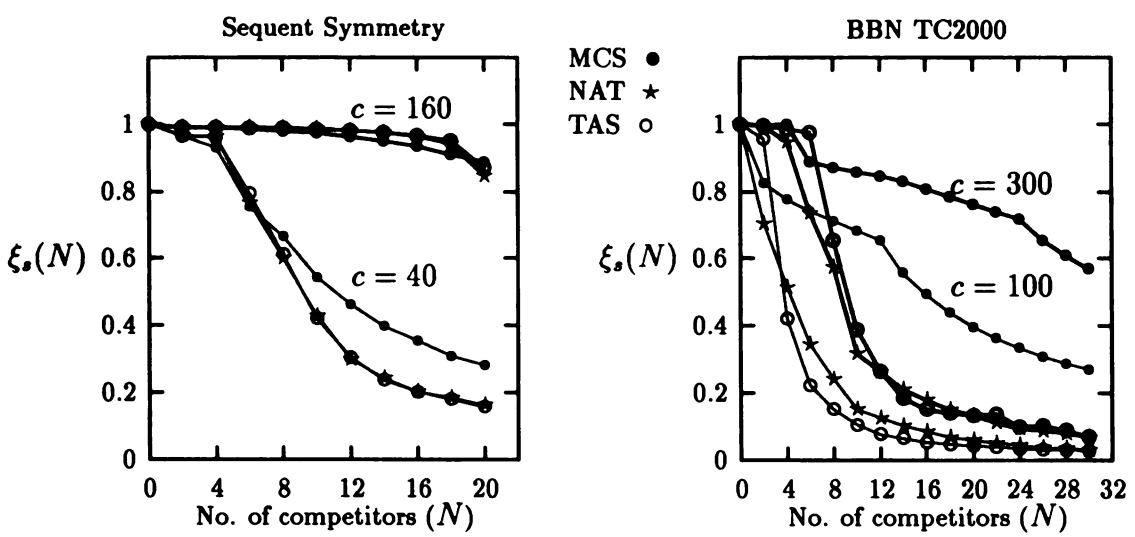
Expressing $\xi_s(N)$ in this form highlights the synchronization losses as being a consequence of two distinct effects: the serialization loss given by $\alpha(N)$ and determined by the characteristics of the algorithm and hardware speed; and the lock contention

loss $\beta(N)$ determined by the spin lock implementation characteristics.

Experiments were conducted using a number of homogenous parameter families, each family designed to measure the effect of a particular grain characteristic on the resultant contention and the consequent unit grain efficiency. In particular, the impact of the frequency of synchronization and the serialization ratio were evaluated. All processors contend for a common lock and the CS guarded by it. In the case of the TC2000, the lock is remote to all processors.

Frequency of synchronization

The probability that a processor arriving at the CS finds it busy, thus incurring a queueing delay, is proportional to the frequency, $1/(\tau_c + \tau_s)$, with which the CS is accessed [96]. The computation granularity τ_c between synchronization points required to restore the loss in efficiency due to synchronization is measured for varying degrees of parallelism N. The result is plotted in Figure 5.12.



$\vec{N}, M = 128K, G_t = G_c(g_m = \phi, g_c = (\vec{c}), g_s = (0,0,0))$

Figure 5.12. Effect of frequency of CS on performance

Examining the workload used in Figure 5.12, we see that the serialization ratio is given by $\lambda = ct_c/t_s = c/c_{1/2}$. The computation granularity c required to obtain 100% structural efficiency, using Eq. 5.7, is given by

$$c \geq Nc_{1/2} \text{ is required for } \alpha(N) = 1.$$

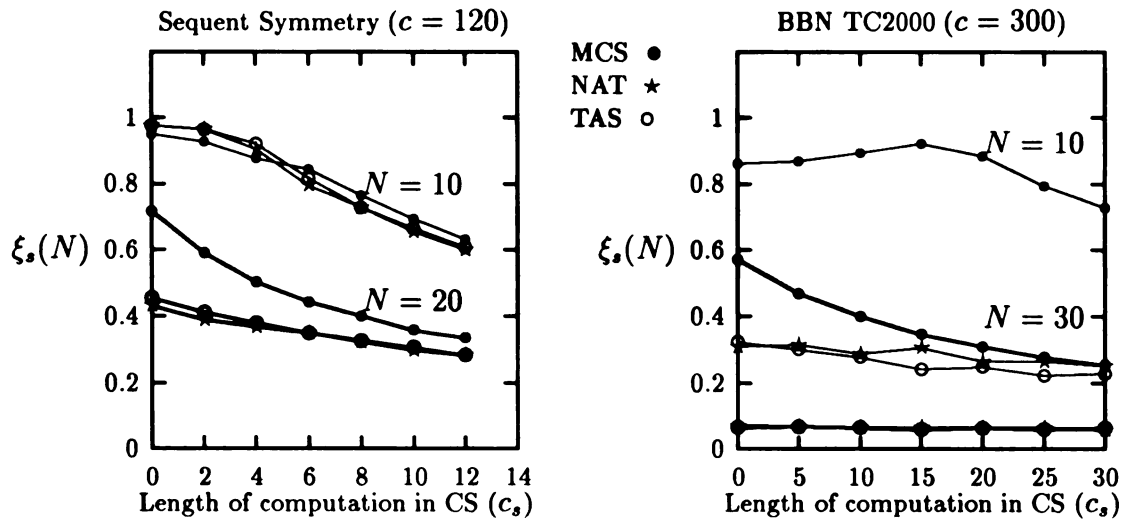
The value of $c_{1/2}$ for the three locks used is given in Table 5.7. A further increase in granularity is required to compensate for the spin lock efficiency $\beta(N)$. As is evident on the Symmetry, the grain efficiency improves by increasing the granularity from $c = 40$ to $c = 160$. The performance of NAT and TAS are almost identical due to their identical implementation. For values of $N \leq 4$, a granularity of $c = 40$ is sufficient to maintain close to perfect efficiency. However, a much higher granularity of $c = 160$ is needed for greater values of N. The additional cache-invalidation traffic on the bus in the case of NAT and TAS account for their lower efficiency compared to MCS.

The $c_{1/2}$ values for all the three locks on the TC2000 are very high necessitating a very large computation granularity to make up for the structural efficiency factor $\alpha(N)$. For the range of c considered in Figure 5.12, the $\alpha(N)$ factor dominates the synchronization performance on the TC2000 thus yielding efficiency curves proportional to $1/(N+1)$ for all three lock types. A granularity of $c = 100$ is not sufficient compensation for any of the locks even with $N = 2$. However, $c = 300$ restores the efficiency to 1 for MCS and NAT with $N \leq 4$, and for TAS with $N \leq 6$.

Serialization ratio

Another important factor governing the performance of an application on a multiprocessor is its serialization ratio λ as given by Eq. 5.6. The length of the CS, τ_s, denotes the amount of time for which the shared lock is held thus affecting the number of spinning processors waiting to access the CS, and λ determines the amount of wait before the CS can be accessed. The serialization ratio λ can be varied by changing the relative amount of computation and shared data accesses performed within and

outside the CS. In Figure 5.13, the variation in grain efficiency is shown as a function of the length of communication in CS (c_s) for a fixed length of computation (c).



$$\vec{N}, M = 128K, G_t = G_c(g_m = \phi, g_c, g_s = (\vec{c_s}, 0, 0))$$

Figure 5.13. Effect of non-CS to CS computation ratio on performance

For the workload used in Figure 5.13, the serialization ratio is given by $\lambda = ct_c/(t_s + c_s t_c) = c/(c_s + c_{1/2})$. The computation granularity c required to obtain 100% structural efficiency, using Eq. 5.7, is given by

$$c \geq N(c_s + c_{1/2}) \text{ is required for } \alpha(N) = 1.$$

A further increase in granularity is required to compensate for the spin lock efficiency $\beta(N)$.

On the Symmetry, the grain efficiency decreases with an increase in the CS length. For $N = 10$, the MCS lock is a little less efficient for low CS lengths. This can be attributed to its higher latency compared to the other two spin-locks. However, for higher values of N, lock contention becomes the dominant factor and MCS outperforms the others. On the TC2000, MCS consistently performs better than the other

two locks for a given N due to its constant number of network accesses per lock acquisition. For the value of c and the range of c_s considered for the TC2000 in Figure 5.13, the $\alpha(N)$ factor dominates the synchronization performance. The NAT and TAS locks exhibit almost identical performance.

5.4 Dual-Mode Access Workloads

Dual-mode access workloads, with both g_m and g_s granules present, are used for characterizing the relative contributions of memory contention and synchronization to the total performance degradation. Measurements were performed on the Sequent Symmetry and the BBN TC2000 systems. The shared data, with M elements, were allocated using the `shmalloc()` call on each machine. On the Symmetry, the data elements were interleaved across the memory modules with a interleaving granularity of 32-bytes; on the TC2000, they were scattered across the allocated cluster.

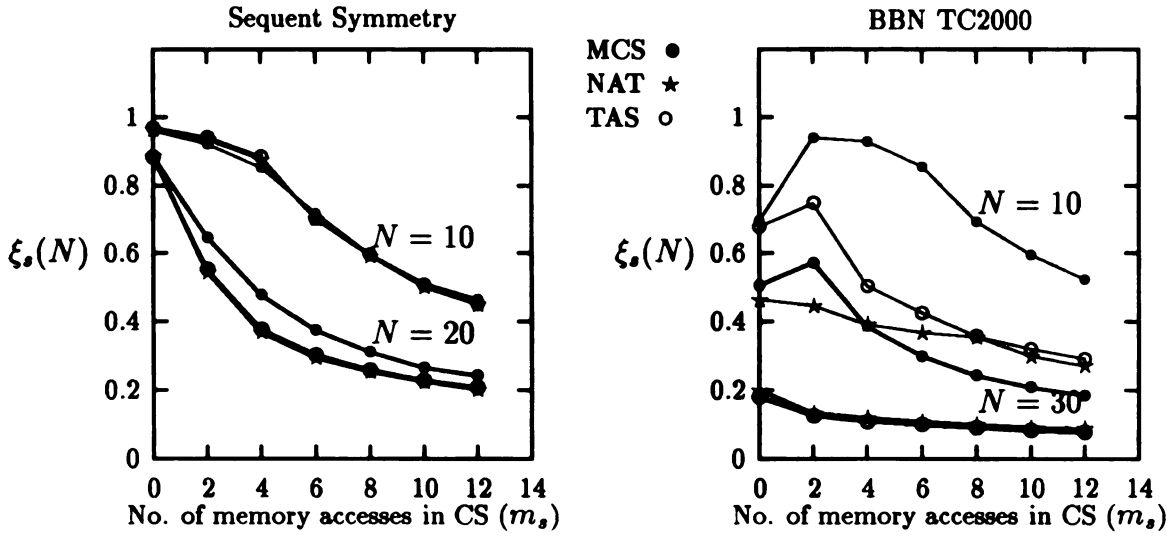
5.4.1 Homogenous Workloads

In these experiments, identical attributes were used for the test and competitor grains. All processes contend for a common lock and the CS guarded by it. On the BBN TC2000 system, the lock is located remote to all processors.

Serialization ratio

In Figure 5.13 the serialization ratio λ was varied by changing the relative number of computation units processed within and outside the CS. A similar effect is also accomplished by varying the number of shared data accesses within the CS (m_s) with respect to the number of accesses outside (m). However, in the latter case, the shared data accesses within the CS may also encounter additional memory access conflicts resulting in longer CS duration. This case is shown in Figure 5.14.

On any machine, all the lock types should experience the same structural efficiency $\alpha(N)$ for a given number of competitors N, since it is proportional to $\lambda = mt_m/(t_s + m_s t_m) = m/(m_s + t_s/t_m)$, and depends only on the unit grain char-



$\vec{N}, M = 128K, G_t = G_c(g_m = (0, 64K[1.0], 4, 48), g_c = \phi, g_s = (0, \vec{m_s}, 1))$

Figure 5.14. Effect of non-CS to CS shared data access ratio on performance

acteristics. Hence, the observed differences in the unit grain efficiency $\xi_s(N)$ of the different spin locks can be attributed to the lock efficiency factor $\beta(N)$ and interference between the shared data accesses performed within and outside the CS.

On both systems, the grain efficiency decreases with an increase in the number of shared data accesses within the CS. At $m_s = 0$, the performance difference is entirely due to the efficiency of the spin lock implementations. As m_s increases, the memory access contention encountered by each access within the CS effectively increases the CS length. On the Symmetry, the MCS lock performs better than the other two for high degree of contention ($N = 20$) because of its higher lock efficiency factor $\beta(N)$. On the TC2000, the effect of the higher efficiency $\beta(N)$ for the MCS lock is more pronounced at $m_s = 0$ for high degree of contention ($N = 30$). The increase in the CS length with an increase in m_s causes the structural efficiency $\alpha(N)$ to reduce thus causing the MCS lock performance to approach that of the other implementations.

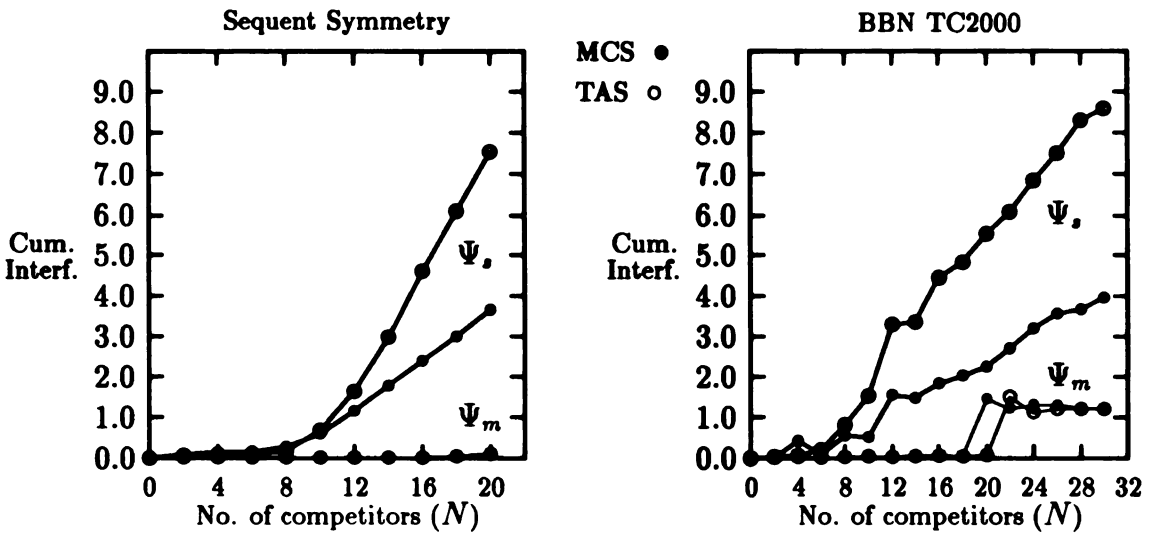
Incremental Overheads

The gross unit grain execution time in the presence of competitor grains includes in it an overhead component due purely to memory contention during shared data accesses, and another due to critical section synchronization. As discussed in Chapter 3, the incremental magnitude of the shared data access and synchronization overhead components for a given workload can be measured and characterized in a hierarchical fashion with the help of the MAD and SAD kernels, respectively. The isolation of the memory contention overhead was described in Section 4.4. Figures 5.15 and 5.16 show the cumulative interferences $\Psi_m(N)$ and $\Psi_s(N)$ measured for two different workloads differing only in the stride of shared data access (*i.e.*, the s attribute). Each interference is shown for two different spin lock implementations, namely, the TAS and MCS locks described earlier. The static characterization parameters for the two workloads are tabulated in Table 5.8.

Table 5.8. Static characterization parameters for workloads used in incremental overhead measurements

Multiprocessor System	R_∞	$f_{1/2}$ ($s = 1$)	$f_{1/2}$ ($s = 23$)	$c_{1/2}$ (TAS lock)	$c_{1/2}$ (MCS lock)
Symmetry	$0.6 \times 10^6/second$	0.288	1.030	3.67	6.08
TC2000	$4.9 \times 10^6/second$	10.75	10.83	39.22	77.45

For the workload shown in Figure 5.15, a stride of 1 is used for data accesses. On the Symmetry, this results in 3 out of 4 accesses being satisfied by the cache thus placing a very low demand on the system bus. This is reflected by an extremely low value of Ψ_m even for large N. On the TC2000, the butterfly switch is able to sustain the bandwidth demand for $N \leq 18$. For higher N, switch contention results in a larger memory interference Ψ_m. On both machines, the MCS lock exhibits a much slower growth rate in the synchronization overhead owing to its lower interconnection network demand.



$\vec{N}, M = 128K, G_t = (g_m = (0, 64K[1.0], 1, 32), g_c = (16), g_s = (1, 2, 0.5))$

Figure 5.15. Incremental interference measured with stride of access $s = 1$

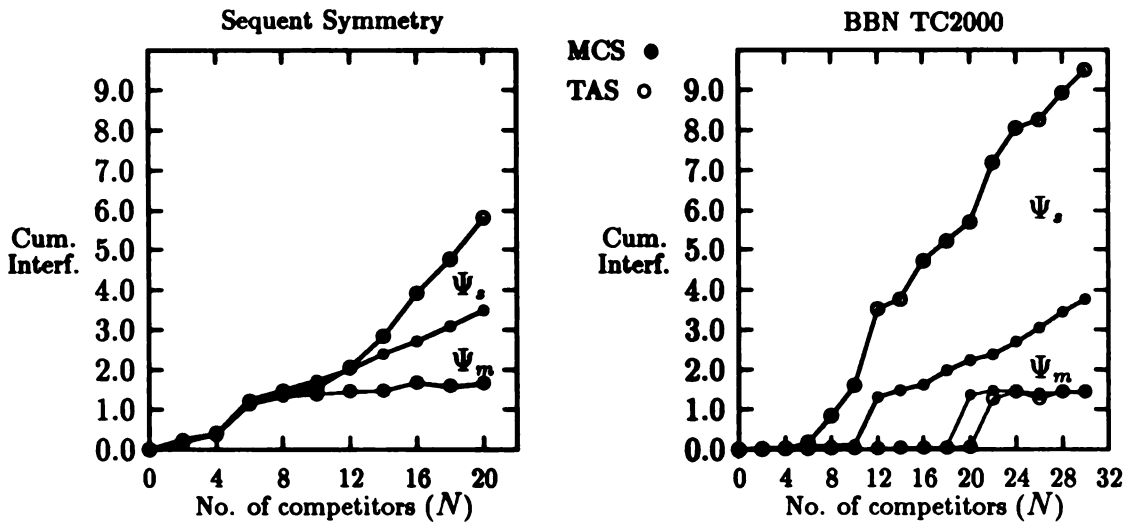
The workload in Figure 5.16 uses a stride of 23 to access shared data. This causes a cache miss for every access on the Symmetry leading to heavier bus contention and consequently a higher value of Ψ_m. On the TC2000, stride 23 accesses do not behave any differently than stride 1 accesses as all memory references go out over the butterfly switch for either strides. The MCS lock again exhibits a slower growth rate of overhead compared to the TAS lock.

The incremental interferences $\psi_m(N)$ and $\psi_s(N)$ can be computed from the measured values of Ψ_m and Ψ_s using the relations 3.24 and 3.25 respectively.

5.4.2 Heterogenous Workloads

Using a heterogenous workload, we have explored the interactions that occur between concurrent execution of code within and outside the critical section. In particular, we investigated the following two situations:

(a) Impact of memory accesses done outside the CS in prolonging the length of the CS by interfering with shared data accesses within the CS, and



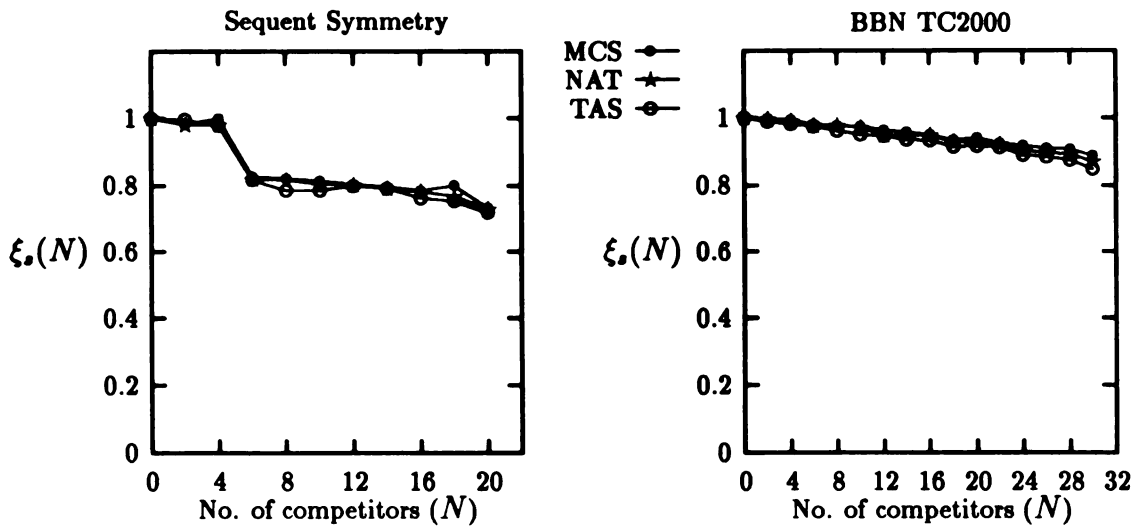
$\vec{N}, M = 128K, G_t = (g_m = (0, 64K[1.0], 23, 32), g_c = (16), g_s = (1, 2, 0.5))$

Figure 5.16. Incremental interference measured with stride of access $s = 23$

(b) Impact of the spin lock accesses to enter the CS on memory references external to the CS.

Figure 5.17 depicts the results of the test described in case (a) above. In this experiment, the test processor P_0 is the only one executing the CS (granule g_s) whereas all competitor processors perform only memory accesses (granule g_m). Since the shared lock location itself encounters no contention, the observed degradation in performance can be ascribed purely to the memory access conflicts that occur between shared-data access within and outside the CS. Hence, all the spin-lock implementations exhibit comparable performance. The efficiency remains close to 1 on the TC2000 due to the much higher bandwidth of its switching network.

An important measure of synchronization performance is the additional amount of interconnection network traffic caused by multiple processors attempting to synchronize, and the impact of this traffic on the execution of the other components of a unit grain. This measure was obtained by recording the performance of a test grain composed of only shared memory accesses (granule g_m) when competing with grains comprised of only critical section accesses (granule g_s). The results are shown

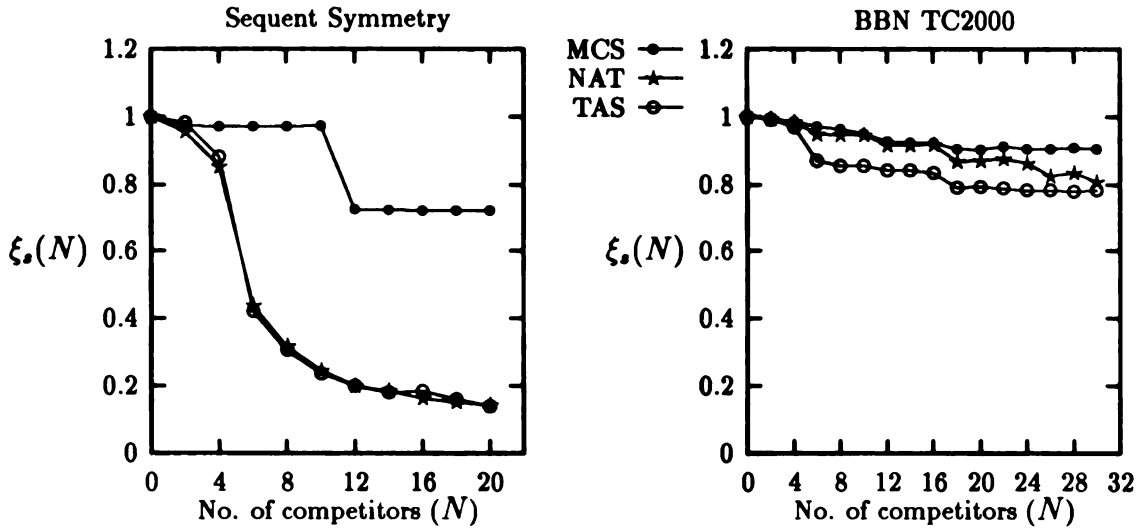


$$\vec{N}, M = 128K, G_t = (g_m = \phi, g_c = \phi, g_s = (0, 8, 0.5)),$$
$$G_c = (g_m = (0, 64K[1.0], 4, 48), g_c = \phi, g_s = \phi)$$

Figure 5.17. Impact of non-CS memory accesses on CS execution performance

in Figure 5.18.

As expected, the MCS spin-lock outperforms the other two by a significant margin on the Symmetry for values of $N \geq 2$ due to its constant number of network accesses per lock acquisition, thus contributing minimal additional bus traffic. Each competitor grain competes with the test grain for the use of the bus three times (refer to Figure 5.10) during each CS access: to fetch the lock queue header, to attach itself at the end of the queue, and to release the lock to the next processor in line. The critical point at which the bus usage by G_c interferes with G_t occurs at $N = 12$ as evident from the sudden drop in test grain efficiency at that point. On the TC2000, the NAT and TAS spin lock traffic interferes with other shared-data access causing a decrease in the grain efficiency. The extent of degradation is not as marked as for the Symmetry due to the higher bandwidth of its interconnect.



$\vec{N}, M = 128K, G_t = (g_m = (0, 64K[1.0], 4, 20), g_c = \phi, g_s = \phi),$
$G_c = (g_m = \phi, g_c = \phi, g_s = (0, 0, 0))$

Figure 5.18. Impact of CS spin-lock on non-CS memory accesses

5.5 Summary

Synchronization among concurrent processes to access critical sections of code in a mutually-exclusive manner leads to loss in parallel performance as a result of serialization of execution. The synchronization overhead incurred is not only a function of the ratio of the amount computation performed outside and inside the CS (serialization ratio), but also of the implementation characteristics of the spin lock used to guard the critical section. The SAD kernels described in this chapter provide an effective means of characterizing the synchronization performance under varying workload conditions. These kernels were employed to measure and compare the performance of three spin lock implementations (TAS, NAT, MCS) on the Sequent Symmetry and the BBN TC2000 multiprocessors.

The MCS lock has the highest uncontested latency. However, it induces the least amount of interconnect contention. For cases in which competition is expected, the MCS lock is the best implementation of choice. The TAS lock has the least latency, but its performance deteriorates rapidly with contention. The performance of the

TAS lock can be improved by exponential backoff between successive poll of the lock variable [2], although this case has not been evaluated here. The NAT lock on the TC2000 does delay between polls for a fixed amount of time (*i.e.*, it is not adaptive). This could result in the release of a busy lock going unnoticed for some time because of the waiting processors being in the middle of the polling delay.

The SAD kernels can be used either independently to evaluate the efficiency and scalability of the implementation of synchronization primitives; or they can be used in conjunction with the MAD and BAD kernels to isolate the incremental overheads resulting from inter-process synchronization and lock access contention from the total performance loss experienced by an input workload.

CHAPTER 6

BAD KERNELS AND BARRIER PERFORMANCE

A barrier defines a logical point in the control flow of an algorithm at which all processes must arrive before any is allowed to proceed further. They are commonly employed when an algorithm consists of several distinct stages, each of which has internal parallelism but which must be performed in strict sequence without overlap. A barrier is clearly one of the most deleterious forms of synchronization, since it requires in effect that every process communicate with every other process. Additionally, since all processes must wait at the barrier until the last arrives, the effects of fluctuations in process execution time or imperfect load balancing are maximized.

The key factors in the performance of a barrier implementation were discussed in Chapter 2. Quantification of the overheads arising from barrier synchronization for a variety of workloads helps assess not only the scalability of a particular barrier implementation to large multiprocessors, but also the loss in execution parallelism. The BAD kernels presented in this chapter can be used in isolation to evaluate and compare the performance of different barrier implementations; or they can be used in conjunction with the MAD and SAD kernels, as per the hierarchical model presented in Chapter 3, to characterize the incremental loss in performance for a given workload resulting from barrier synchronization.

6.1 BAD Workload Parameters

The synchronization barrier separates adjacent phases of the multi-phase computation structure selected as the basis of our performance studies. The presence of the barrier increases the completion time of a phase by adding the time to execute the barrier. Moreover, it also forces all processors to wait for the slowest among them thus accentuating the worst-case performance. To incorporate the effects of barriers into our characterization of the aggregate performance of a workload, measurements are performed at the level of a phase of computation.

6.1.1 Phase Characterization

A single phase consists of a number of concurrent processes executing a string of unit grains and terminating at a global synchronization barrier. Hence, the parameters necessary to characterize a phase of computation must include: (1) a set of attributes to describe the behavior of the concurrent unit grains within the phase, (2) the number of unit grains executed by each processor in the phase, and (3) the choice of a particular type of barrier implementation. The parameters used in the barrier performance studies are summarized in Table 6.1.

Table 6.1. Workload parameters for studying barrier performance

Granule	Attribute	Meaning
common	N	number of competitor processors in a phase
	M	number of shared data elements
	ℓ	number of unit grains per processor per phase
g_m	p	probability of write access to shared memory
	d	initial distance of concurrent address streams
	s	stride of memory access
	m	number of shared memory accesses per granule
g_c	c	number of basic computation units (BCUs)
g_s	c_s	number of computation steps in CS
	m_s	number of memory accesses in CS
	p_s	probability of a write access in CS

As is evident from Table 6.1, the attributes describing the unit grain behavior are the same as used for the SAD kernels with the additional parameter ℓ representing the the length of the task executed by each processor. If the total work to be performed within a phase, say W unit grains, is perfectly parallelizable among P processors in a homogenous setting, then the amount of work performed by each individual processor is given by $\ell = \lceil W/P \rceil$. The consolidated set of input parameters $\mathcal{I}$ to the experimental framework (described in Section 3.3), therefore, now becomes

$$\mathcal{I} = \{N, M, \ell, G_t, G_c\},$$

where the test and competitor unit grains are characterized by the *3–tuple* of tuples $G = ((p, d, s, m), (c), (c_s, m_s, p_s))$.

6.1.2 Output Metrics

The metric used to quantify the overhead of barrier synchronization and the consequent increase in the phase execution time for an input workload, as a function of the degree of interference N, is the *cumulative barrier interference* $\Psi_b(N)$ as defined by Eq. 3.20. In other words, if $T(N)$ denotes the total time to complete executing a phase with N competitor processes contending for resources, then

$$\Psi_b(N) = \frac{T(N) - T(0)}{T(0)} = \frac{T(N) - \ell\tau}{\ell\tau}.$$

When $N = 0$, there is only one processor operating thus making the barrier synchronization redundant. Therefore, the measured execution time for a workload with $N = 0$, *i.e.*, $T(0)$, does not include the barrier overhead thus yielding $T(0) = \ell\tau$ used in the expression above. The *incremental barrier interference* $\psi_b(N)$ can be computed from the measured values of $\Psi_b(N)$ and $\Psi_s(N)$ for a given workload as dictated by Eq. 3.26.

A value of $\Psi_b(N) = 0$ would indicate the barrier as an idealized entity which consumes no resources and induces no interference with the processes executing within

a phase. In reality, however, a barrier *does* consume resources, and this will have a major effect on performance. Although $\Psi_b(N) > 0$ for a non-ideal barrier, judicious design choices can help minimize this interference. It should be noted that since $\Psi_b(N)$ expresses the barrier overhead encountered in terms of an abstract normalization unit, namely $T(0)$, it is a suitable metric for comparing performance only when the same reference workload is used as the basis. For performance comparisons across workloads, the absolute time measure $T(N)$ should be used instead.

```
shared count : integer := P; { number of processors synchronizing }
shared sense : Boolean := True;
processor private local_sense : Boolean := True;

procedure CentralBarrier()
    local_sense := not local_sense; { Each processor toggles its own sense }
    if Fetch-And-Decrement (&count) = 1 then
      count := P;
      sense := local_sense; { Last processor toggles global sense }
    else
      repeat until sense = local_sense;
```

Figure 6.1. Pseudo-code for a sense reversing centralized barrier

6.1.3 Barrier Implementations Studied

We have chosen two barrier implementations on each of the target systems studied to demonstrate the utility of the BAD kernels. The first is a centralized implementation of the barrier (referred to as the CNT barrier), where each processor updates a small amount of shared state to indicate its arrival and then polls that state to determine

when all of the processors have arrived. Most barriers are designed to be used repeatedly (to separate the phases of an algorithm). In the most obvious formulation, each instance of a centralized barrier begins and ends with identical values for the shared state variables. Each processor must spin twice per instance; once to ensure that all processors have left the previous barrier and again to ensure that all processors have arrived at the current barrier.

The number of references to the shared state variables can be reduced and one of the two spinning episodes can be eliminated by "reversing the sense" of the variables (and leaving them with different values) between consecutive barriers [58]. The resulting code is shown in Figure 6.1. Arriving processors decrement **count** and wait until **sense** has a different value than it did in the previous barrier. The last arriving processor resets **count** and reverses **sense**. Consecutive barriers cannot interfere with each other because all operations on **count** occur before **sense** is toggled to release the waiting processors.

The potential drawback of centralized barriers is the spinning that occurs on a single, shared location. Because processors do not in practice arrive at a barrier simultaneously, the number of busy-wait accesses will in general be far above the minimum. On broadcast-based cache-coherent multiprocessors, these accesses may not be a problem. The shared flag (or **sense** variable) is replicated into the cache of every waiting processor thus causing local spinning without any network traffic. This shared variable is written only when the barrier is achieved, causing a single broadcast invalidation of all cached copies.

On machines without coherent caches, however, or on machines with directory-based caches without broadcast, busy-wait references to a shared location may generate unacceptable levels of memory and interconnection contention. For such classes of machines, Hengsen, Finkel, and Manber [58] have proposed a "dissemination barrier" (referred to here as the DSM barrier) that yields a much more efficient pattern of synchronization. In round k (counting from 0) with P processors participating, processor i signals processor $(i + 2^k)$ mod P. Synchronization is not necessarily pairwise and requires only $\lceil \log_2 P \rceil$ synchronization operations on its critical path regardless

of P. The flags on which each processor spins are statically determined, and no two processors spin on the same flag. Each flag can therefore be located near the processor that reads it leading to local-only spinning.

```
type Flags = record
     myflags : array [0..1] of array [0..LogP] of Boolean;
     partnerflags : array [0..1] of array [0..LogP] of ↑Boolean;
   end;
processor private parity : integer := 0;
processor private sense : Boolean := True;
processor private localflags : ↑flags;
shared allnodes : array [0..P-1] of flags;
   { allnodes[i] is allocated in shared memory locally accessible to processor i. }

{ On processor i, localflags points to allnodes[i]. Initially allnodes[i].myflags[r][k]
is False for all i, r, k. If j = (i + 2^k) mod P, then for r = 0,1:
   allnodes[i].partnerflags[r][k] points to allnodes[j].myflags[r][k]. }

procedure DisseminationBarrier()
     for instance : integer := 0 to LogP-1 do
       localflags↑.partnerflags[parity][instance]↑:= sense;
       repeat until localflags↑.myflags[parity][instance] = sense;
     if parity = 1 then
       sense := not sense;
     parity := 1 - parity;
```

Figure 6.2. Pseudo-code for a distributed dissemination barrier

Figure 6.2 presents the dissemination barrier. Alternating sets of variables are used in consecutive barrier episodes for each signaling operation, thus avoiding interference

without needing two separate spins in each operation. Sense reversal is also used to avoid resetting variables after each barrier. The **parity** variable controls the use of alternating sets of flags in successive barrier episodes. The shared **allnodes** array would be scattered statically across the memory banks on a machine with distributed shared memory and no coherent caches.

6.2 Embarrassing Workloads

The class of embarrassing workloads (refer to Section 3.2) are used to measure the performance effects attributable purely to barrier synchronization. Since no shared data accesses nor inter-processor synchronizations are present within the unit grain (*i.e.*, $g_m = \phi, g_s = \phi$), the concurrent processes within a phase execute independently of each other. Any observed losses in performance can be ascribed entirely as the result of global barrier synchronization.

Synchronization barriers impose two kinds of performance penalties on the runtime behavior of an algorithm. The first, which is in some sense irreducible, is due to fluctuations in the time taken by the processors to complete their share of the work within a phase. The second kind of penalty results from the use of resources by the barrier, and in particular the contention for shared resources. The consequences of fluctuations in the execution time or the unbalanced workload distribution are maximized as a result of the wait for the last processor to complete its work.

If the barrier itself is considered as an idealized entity which consumes no resources, the execution time of the phase can be determined analytically, as Kruskal and Weiss [68] have shown. If there are P processors (note that in terms of our workload parameters, $P = N + 1$) that begin their work simultaneously, and the time each takes has the mean μ and standard deviation σ, then the time at which the last processor completes its work, T_P, is given by

$$T_P = \mu + \sigma\sqrt{2 \log P}. \tag{6.1}$$

The approximation is especially good for a Gaussian distribution function but is valid more generally as shown in [68].

In reality, the barrier execution does consume resources and computational cycles. The time to achieve the barrier, T_{barr}, consists of two distinct components [16]: the *entry phase* time, T_{entry}, during which processors report their arrival; and the *exit phase* time, T_{exit}, during which processors exit after determining that all other processors have arrived. There are two cases to consider:

1. A *balanced* load is one for which the variance in arrival times is less than the overhead incurred at the entry phase. An extreme case is the perfectly balanced load for which $\sigma = 0$ in which case all processors arrive at the barrier simultaneously. The barrier overhead in this case is the time for all P simultaneously arriving processors to traverse the entry and exit phases.

$$T_{barr}(P) = T_{entry}(P) + T_{exit}(P) \tag{6.2}$$

2. An *unbalanced load* is one for which the variance in arrival times is greater than the time required for the entry phase. An extreme case occurs when the last processor to arrive at the barrier finds that all other processors have completed the entry phase. In this case, the barrier overhead is given by the last processor to complete its entry and all P processors to exit.

$$T_{barr}(P) = T_{entry}(1) + T_{exit}(P) \tag{6.3}$$

The total time to complete a phase of execution, $T(P)$, can be expressed as the sum of the effects of unbalanced load (Eq. 6.1) and barrier overhead.

$$T(P) = \mu + \sigma\sqrt{2\log P} + T_{barr}(P)$$

If the total performance penalty resulting from barrier synchronization is denoted as

$O_b(P)$, then

$$O_b(P) = \sigma\sqrt{2\log P} + T_{barr}(P).$$

It is clear that the cumulative barrier interference $\Psi_b(N)$ is proportional to $O_b(P)$. Using Eqs. 6.1, 6.2 and 6.3, and the fact that $\sigma = 0$ for a balanced load, the overhead function can be written, for a balanced load, as

$$O_b(P) = T_{entry}(P) + T_{exit}(P), \tag{6.4}$$

and for an unbalanced load as

$$O_b(P) = \sigma\sqrt{2\log P} + T_{entry}(1) + T_{exit}(P). \tag{6.5}$$

The time to complete the entry and exit phases for the two barrier implementations selected (CNT and DSM) can be expressed in terms of the timing of the basic operations involved. For the CNT barrier, the entry phase entails that each of the P processors atomically decrement the **count** variable, each decrement operation requiring a time of t_{atomic}. The entry phase of the DSM barrier, on the other hand, requires each arriving processor to signal its arrival only to its first round synchronization partner, the pairwise synchronization round needing a time of t_{signal}. Therefore, the time for the entry phase can be expressed as follows.

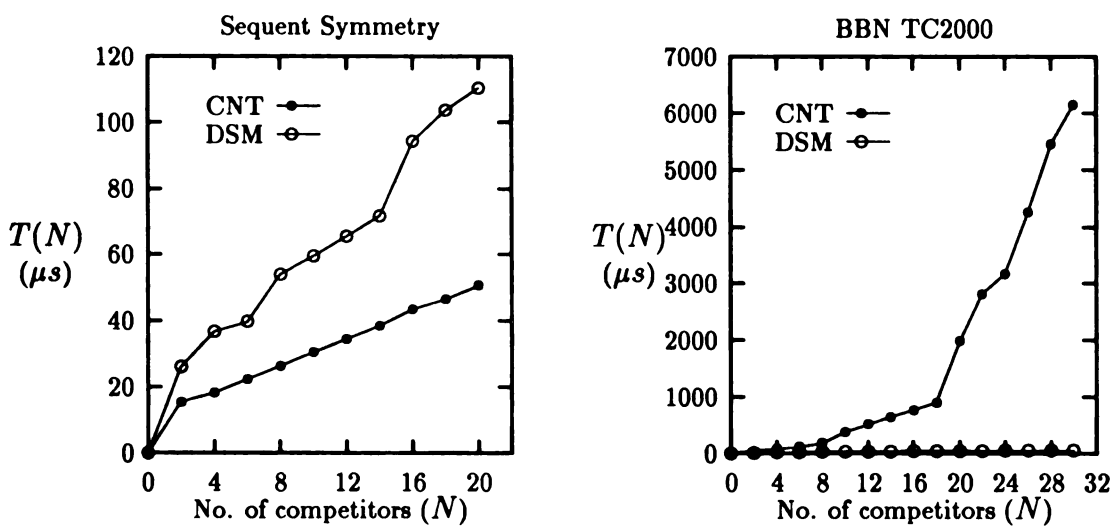
$$T_{entry}(P) = \begin{cases} Pt_{atomic} & \text{for the CNT barrier} \\ t_{signal} & \text{for the DSM barrier} \end{cases} \tag{6.6}$$

Similarly, the exit phase of the CNT barrier consists of the last arriving processor writing to the **sense** flag to toggle its status (requiring time t_{write}), and the changed status of **sense** being read by the $P-1$ waiting processors (each read requiring time t_{read}). The exit phase of the DSM barrier goes through the remaining $(\log P - 1)$ rounds, the first round having been performed in the entry phase, of pairwise signaling

to complete the barrier. Thus, the exit phase time can be expressed as follows.

$$T_{exit}(P) = \begin{cases} t_{write} + (P-1)t_{read} & \text{for the CNT barrier} \\ (\log P - 1)t_{signal} & \text{for the DSM barrier} \end{cases} \quad (6.7)$$

The equality in the Eqs. 6.6 and 6.7, in reality, should read "proportional to" for accuracy. However, the constant of proportionality is not important for the discussion at hand and, hence, has been treated as unity. It should also be noted that the values of t_{write} and t_{read} used in the expressions for the CNT barrier are not constants for machines without coherent caches, and are determined by the hot spot access latency for that system with the variable **sense** being the hot spot site. Similarly, t_{signal} used for the DSM barrier may involve $O(1)$ network transactions if parallel accesses over the interconnection are possible (such as on a MIN), or $O(P)$ network transactions on serial interconnections (such as on a bus).



$$\vec{N}, M = 0, \ell = 0, G_t = G_c(g_m = \phi, g_c = \phi, g_s = \phi)$$

Figure 6.3. Time to achieve barrier vs. N

6.2.1 Scalability of Barrier Implementations

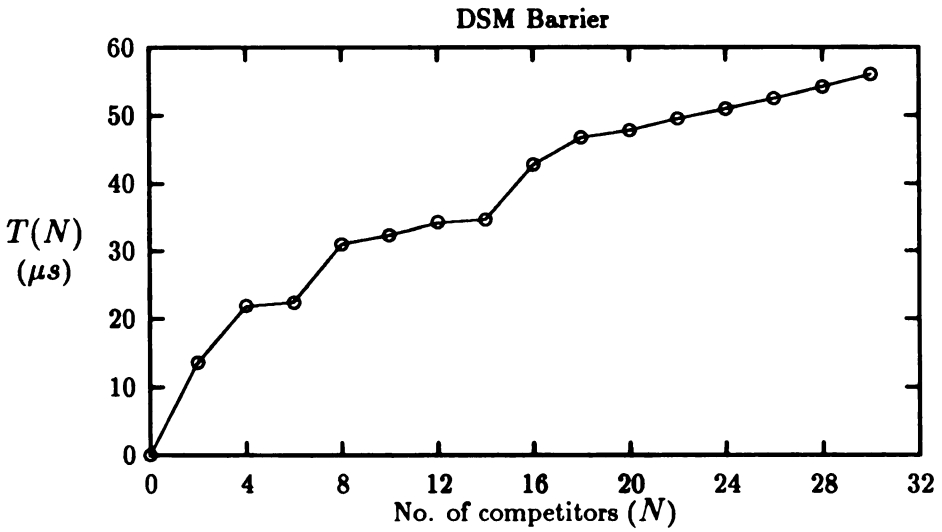
In large-scale multiprocessors, the number of interconnection network accesses per processor to achieve the barrier increases sharply as collisions in the network cause processors to repeat accesses. This observation is especially true for centralized barrier algorithms, like CNT, implying that they will not scale well to large numbers of processors. Algorithms that restrict spinning to locally-accessible memory, like the DSM, are much more amenable to scaling for large numbers of processors. Our measurements confirm this conclusion. Figure 6.3 shows the time $T(N)$ to achieve barrier, with no computation at all in the unit grains, for the two barrier implementations chosen.

On the TC2000, the time to achieve a CNT barrier increases more than linearly in the number of participants. Since the Butterfly switch does not provide hardware combining, at least $2P - 1$ accesses to the barrier state are required (P to signal arrivals, and $P - 1$ to discover that all have arrived). The DSM barrier, on the other hand, proceeds through only $\lceil \log_2 P \rceil$ rounds of synchronization that leads to a stair-step curve (shown in Figure 6.4 for clarity). The time to achieve a barrier with this algorithm scales logarithmically with the number of processors participating.

The performance on the Symmetry differs sharply from that on the TC2000 for two principal reasons. First, it is acceptable on the Symmetry for more than one processor to spin on the same location; each obtains a copy in its cache. Second, no significant advantage arises from distributing writes across the memory modules; the shared bus enforces an overall serialization. The DSM barrier requires $O(P \log P)$ bus transactions to achieve a P-processor barrier, whereas the CNT barrier requires only $O(P)$ transactions. Consequently, the CNT barrier scales better than the DSM barrier on the Symmetry.

6.2.2 Balanced Load and Simultaneous Arrivals

A balanced workload exhibits a variance in processor arrival times at the barrier that is much less than the overhead incurred at the barrier. A perfectly balanced



$$\vec{N}, M = 0, \ell = 0, G_t = G_c(g_m = \phi, g_c = \phi, g_s = \phi)$$

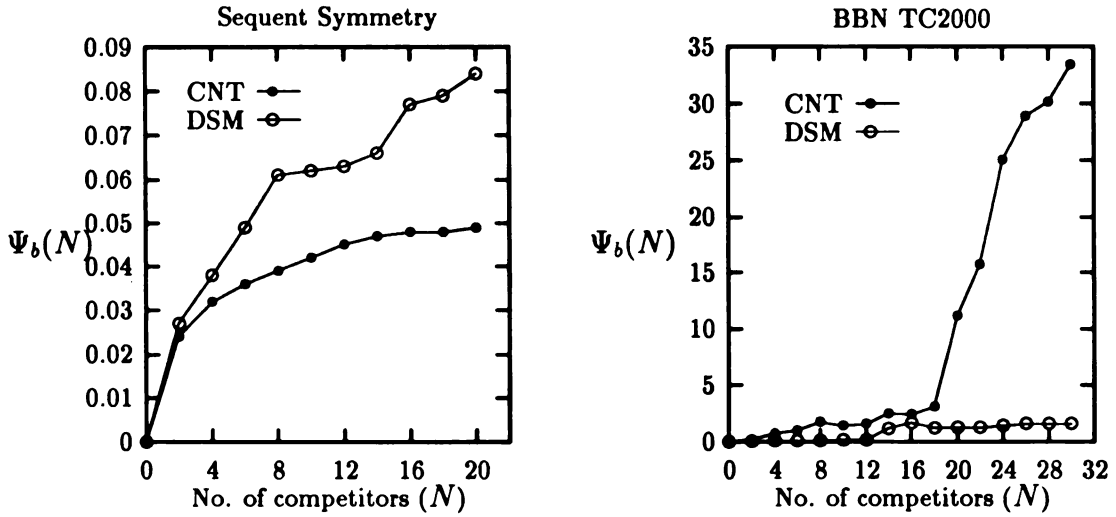
Figure 6.4. Time to achieve DSM barrier on the TC2000

load with a constant execution time on each processor (*i.e.*, $\sigma = 0$) induces the maximum overhead at the entry phase of a linear barrier, since simultaneously arriving processors contend for access to the shared barrier state and must be serialized. A slightly increased fluctuation level can, indeed, benefit performance [9]. This occurs because the presence of fluctuations can reduce the queue length at the barrier entry critical section by spreading out arrival times and causing some processors to start synchronizing early.

Figure 6.5 shows the barrier performance of a perfectly balanced workload with a constant number of computation steps ($c = 1000$) executed per processor between barriers. The overhead curves observed for this workload are a result of the dominant effect of the barrier overhead on performance as given by Eq. 6.4. The overhead $O_b(P)$ incurred is obtained by combining Eqs. 6.4, 6.6 and 6.7.

$$O_b(P) = \begin{cases} P t_{atomic} + t_{write} + (P-1) t_{read} & \text{for CNT} \\ (\log P) t_{signal} & \text{for DSM} \end{cases}$$

The higher overhead for DSM on the Symmetry is a direct consequence of the



$\vec{N}, M = 0, \ell = 1, G_t = G_c(g_m = \phi, g_c = (1000), g_s = \phi)$

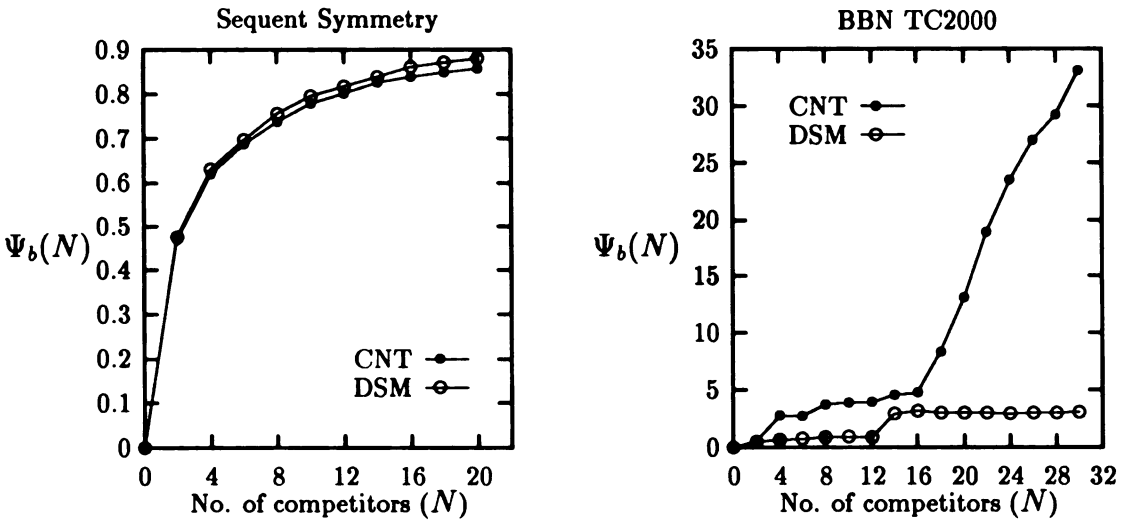
Figure 6.5. Barrier performance of a perfectly balanced load

P bus transactions required in each t_{signal}, due to the serial nature of the bus, thus needing a total of $P \log_2 P$ bus accesses. In comparison, CNT requires only $2P$ bus accesses. Thus, the performance of DSM and CNT remain comparable for up to $\log_2 P \leq 2$ (*i.e.*, $P \leq 4$) beyond which the higher $\log P$ factor for DSM causes the performance curves to diverge. In contrast, DSM on the TC2000 requires a total of only $\log P$ network transactions as the Butterfly switch permits parallel accesses, whereas CNT must perform P atomic operations serially followed by P accesses to a hot spot location. The hot spot accesses result in extremely high latencies for t_{write} and t_{read} for $N \geq 18$. This is evident from the significant difference in overheads for DSM and CNT for $N \geq 18$.

6.2.3 Unbalanced Load and Staggered Arrivals

In an unbalanced workload, processors arrive at the barrier in a staggered fashion. The variance in processor arrival times is greater than the time required to synchronize at the barrier. This results in the variance in arrival times to dominate observed performance. Figure 6.6 shows the barrier effects on an unbalanced workload in which

each processor performs c computation steps randomly selected from an Uniform distribution over the interval $(0, 2000]$.



$$\vec{N}, M = 0, \ell = 1, G_t = G_c(g_m = \phi, g_c = (1000[1]), g_s = \phi)$$

Figure 6.6. Barrier performance of an unbalanced load

The total performance penalty $O_b(P)$ for an unbalanced load is given using Eqs. 6.5, 6.6 and 6.7.

$$O_b(P) = \begin{cases} \sigma\sqrt{2\log P} + t_{atomic} + t_{write} + (P-1)t_{read} & \text{for CNT} \\ \sigma\sqrt{2\log P} + (\log P)t_{signal} & \text{for DSM} \end{cases}$$

The standard deviation σ of a random variable uniformly distributed over the interval $[a, b]$ is given by $(b-a)^2/12$. For the workload used in Figure 6.6, the standard deviation of the computation times is thus given by

$$\sigma = \frac{2000}{\sqrt{12}} \cdot t_c = \frac{2000}{\sqrt{12}} \cdot R_\infty^{-1}$$

where t_c is the time per computation step. Since $t_c = 1/R_{infty}$ is much larger on the

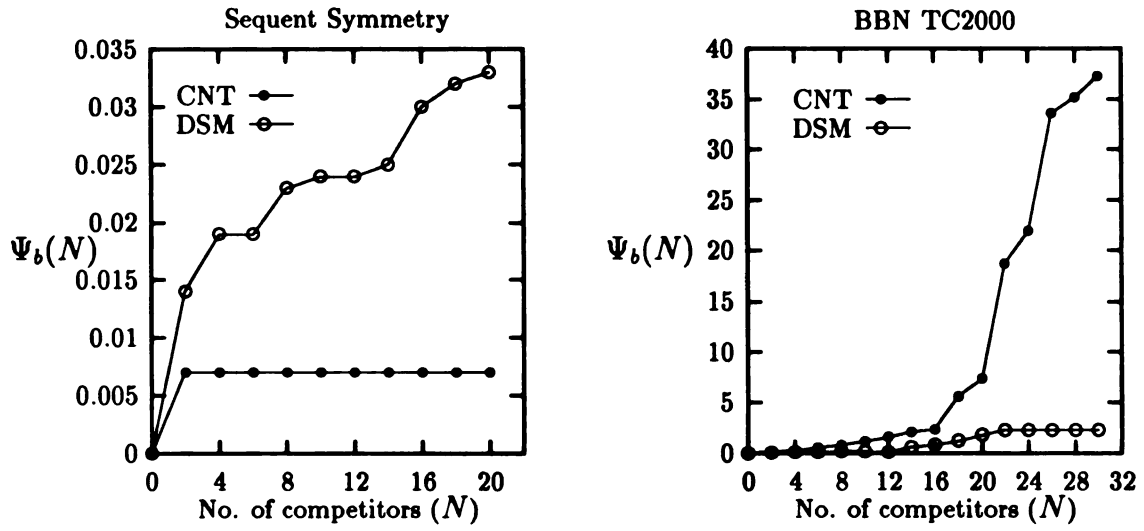
Symmetry (refer to Table 4.4), the effect of the variance in arrival times predominates thus rendering the difference in the barrier overheads as insignificant. The DSM and CNT, therefore, exhibit almost identical behavior of O_b (and hence Ψ_b) on the Symmetry. The dominance of the arrival fluctuations is evident by observing that $\Psi_b(20) = 0.049$ for the CNT barrier for a balanced workload with $c = 1000$ (Figure 6.5); but it makes a quantum jump to $\Psi_b(20) = 0.86$ for an unbalanced workload with the same mean (Figure 6.6). A similar increase can also be noted for the DSM barrier. On the TC2000, however, the value of σ is not large enough to overshadow the difference in the DSM and CNT barrier overheads. The effect of σ, therefore, is observed as a slight increase in the value of $\Psi_b(N)$.

To isolate and highlight solely the effect of the barrier overhead (the terms $T_{entry}(1) + T_{exit}(P)$ in Eq. 6.5) in an unbalanced load, we measured the barrier performance of a heterogenous workload in which the test processor executes a number of computation steps far in excess of those executed by the competitor processors, *i.e.*, $G_t(c) \gg G_c(c)$. Also, $G_c(c)$ is made to vary randomly over an uniformly distributed interval to emulate the staggered arrivals. By the time the test processor P_0 arrives at the barrier, all other processors have already completed their entry phase and are waiting. The performance for this case is shown in Figure 6.7.

The broadcast-based cache-invalidate operation in T_{entry} of the CNT barrier on the Symmetry results in a constant overhead ($t_{atomic} + t_{write}$) incurred by the test processor P_0. For CNT on the TC2000, P_0 has to compete with the N processors already spinning on the sense variable to toggle it thus incurring a high t_{write} latency. The DSM barrier on both machines, however, requires P_0 still has go through the $\log P$ rounds of synchronizations thus exhibiting essentially the same overhead as for a balanced load.

6.3 Dual-Mode Access Workloads

In accounting for the effect of variance in the arrival times of processors at a barrier in Eq. 6.1, it was assumed that the the fluctuations in the execution time of each



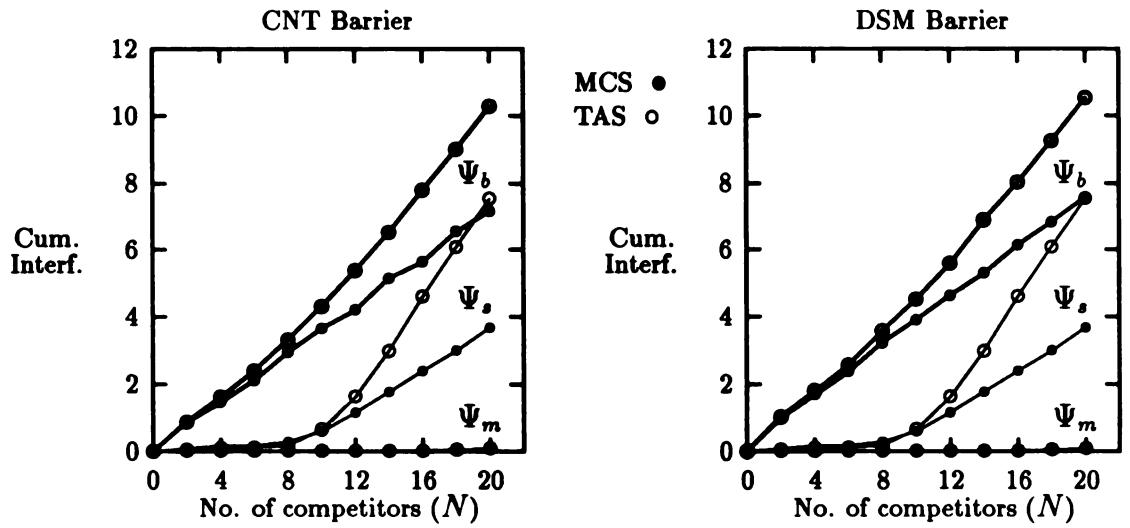
$\vec{N}, M = 0, \ell = 1, G_t = (g_m = \phi, g_c = (1000), g_s = \phi)$,
$G_c = (g_m = \phi, g_c = (300[1]), g_s = \phi)$

Figure 6.7. Performance of staggered arrivals at the barrier

processor were incidental in the computation itself. The effects of memory conflicts, contention for other hardware resources or other interprocessor interactions were ignored due to the nature of the embarrassing workloads. Therefore, the computation times of all processors could be treated as independent identically distributed random variables (i.i.d's) with mean μ and variance σ.

However, if fluctuations in the barrier arrival times are present as a result of planned interactions between processors during the phase, such as contention in regular reference patterns to shared data and mutual exclusion at critical sections, then the assumption of independence between processors no longer holds. The situation thus becomes more complex and the effect of fluctuations is best characterized experimentally. The dual-mode access workloads are used for this purpose.

The same workloads as used in Section 5.4 to measure the incremental overhead components associated with memory access contention and CS synchronization are used here again to observe the incremental overhead resulting from barriers. The cumulative interference values Ψ_m, Ψ_s and Ψ_b as measured by the MAD, SAD and



$\vec{N}, M = 128K, \ell = 1, G_t = (g_m = (0, 64K[1.0], 1, 32), g_c = (16), g_s = (1, 2, 0.5))$

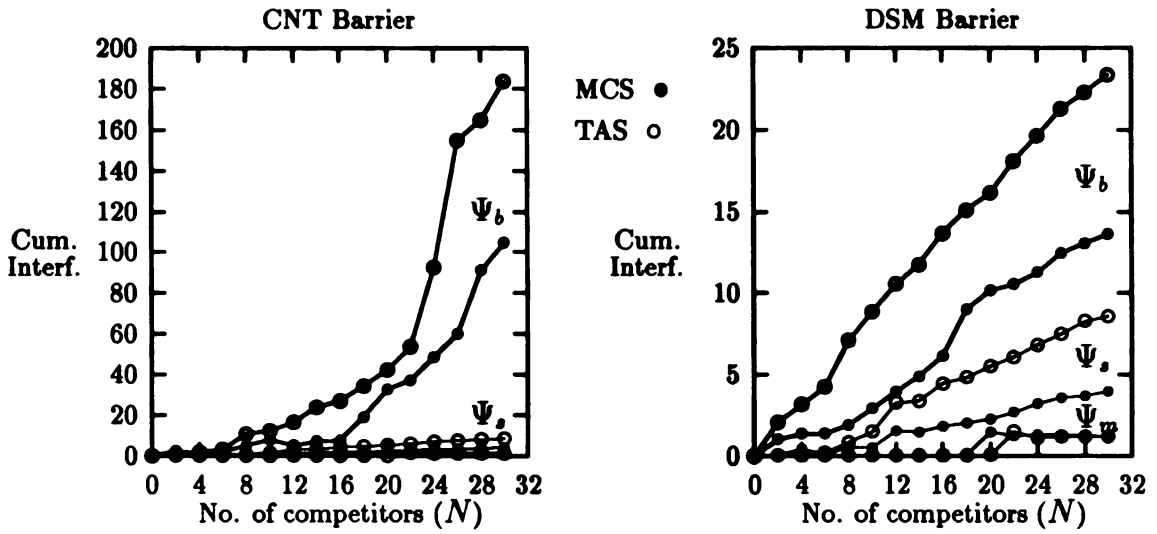
Figure 6.8. Cumulative interferences unit stride workload on the Symmetry

BAD kernels respectively for the same workload are plotted in Figure 6.8 (for the Symmetry) and Figure 6.9 for the TC2000. The workload with unit stride ($s = 1$) is used. For each barrier implementation, the cumulative barrier interference $\Psi_b(N)$ is measured with the TAS and MCS locks used, in turn, for the critical section.

Since $\ell = 1$ for the workload used, the difference between the Ψ_b and Ψ_s curves directly measure the incremental barrier interference ψ_b. In other words,

$$\psi_b(N) = \Psi_b(N) - \Psi_s(N)$$

in Figures 6.8 and 6.9. On the Symmetry, both CNT and DSM barriers display comparable values for the cumulative and hence incremental barrier interference. This is as a result of the predominance of the effect of barrier arrival fluctuations discussed in the previous section. Both TAS and MCS lock workloads experience similar increases in the total overhead on account of the barrier. It is also noteworthy that for low values of $N < 12$ the incremental barrier interference $\psi_b(N)$ is the single largest source of runtime overheads.



$\vec{N}, M = 128K, \ell = 1, G_t = (g_m = (0, 64K[1.0], 1, 32), g_c = (16), g_s = (1, 2, 0.5))$

Figure 6.9. Cumulative interferences unit stride workload on the TC2000

On the TC2000, the incremental barrier interference $\psi_b(N)$ is far worse for the CNT barrier than for DSM. The primary reason behind this dismal performance is two-fold: first, the last arriving processor at the CNT barrier must contend with the N processors already present for access to the "sense" flag to toggle its state; second, the continuous spins on the barrier sense flag flood the interconnection network with busy-wait traffic thus interfering with the memory accesses performed within the unit grain. The effect of the busy-waits is further accentuated in the performance of the TAS lock workload for large N due to the combination of two spinning instances, namely, within the TAS lock and within the CNT barrier. With the DSM barrier, however, the incremental barrier penalties experienced by both TAS and MCS lock workloads are comparable.

6.4 Summary

Synchronization barriers impose two kinds of performance penalties on parallel algorithm performance: overhead of barrier execution, and maximization of load im-

balance losses. The overhead of barrier execution includes the contention for shared resources by the barrier code. Two barrier implementations were studied on the Sequent Symmetry and TC2000 multiprocessors — a centralized sense-reversing barrier (CNT) and a tree-like dissemination barrier (DSM). If independent network transactions can proceed in parallel on a machine, then the critical path length is $O(\log P)$ for the DSM, but $O(P)$ for the CNT. On an interconnection that serializes network transactions, the logarithmic factor will be dominated asymptotically by the linear (or more) total number of network transactions.

The DSM barrier was observed to be more suitable on the distributed-memory TC2000 system, whereas CNT performed better on the cache-coherent Symmetry system. In the DSM barrier, no network transactions are due to spinning, so interconnect contention is not a problem. The CNT, on the contrary, maximizes memory contention and hot spots caused by synchronization on machines without coherent caches. The performance of CNT on distributed-memory machines without coherent caches can be improved by adaptive backoff strategies between polls of the sense flag. However, their scalability is limited on large-scale systems, as the number of network accesses per processor increases sharply as collisions in the network cause processors to repeat accesses [2].

The CNT barrier enjoys one additional advantage over DSM: it adapts easily to differing numbers of processors. If the number of processors participating a barrier changes from one barrier episode to another, the log-depth DSM barrier would require internal reorganization, possibly swamping any performance advantage obtained in the barrier itself. Changing the number of processors in the CNT entails no more than changing a single constant.

The BAD kernels can be used either independently to evaluate the efficiency and scalability of the implementation of a barrier mechanism; or they can be used in conjunction with the MAD and SAD kernels to isolate the incremental overheads incurred as a result of synchronization barriers from the total performance loss experienced by an input workload.

CHAPTER 7

CONCLUSIONS

The increasing complexity of multiprocessor systems necessitates the development of accurate techniques to characterize their behavior under a variety of workload conditions so that efficient algorithms can be designed to effectively utilize the machine and reasonable performance expectations established. This thesis proposes a hierarchical model to characterize multiprocessor system performance and develops techniques to measure and calibrate the parameters of the model. In this chapter, we summarize the salient contributions made by this research and present interesting avenues for possible future research.

7.1 Research Contributions

The run-time overhead of communication on multiprocessors can significantly limit the amount of program parallelism that can be exploited. In programs using the shared-variable paradigm, communication manifests itself along three principal dimensions, namely, shared data accesses (including memory contention, cache misses in cache-coherent machines and non-local memory accesses in hierarchical or distributed memory machines), inter-process synchronization operations, and global barrier synchronizations. As more processors are added, the communication costs of algorithms increase. It is the rate at which these costs increase that determines an algorithm's efficiency and scalability. Measurements must be made to quantify the impact of such run-time overheads on the overall performance of a system for specific algo-

rithms/applications.

We have developed a system characterization methodology based on a hierarchical approach using a multi-phase computation structure to describe the static and dynamic behavior of program execution on a multiprocessor. We maintain that the characterization of performance is tied inextricably to the input workload used and, therefore, should depend significantly on the user's needs and preference for selective workload characteristics. We have presented a flexible technique for benchmark workload generation that can be tailored to highlight specific aspects of performance. The workload generator is based on the definition of a unit grain that allows a user to identify the most significant factors influencing performance and use them as the characterization attributes for the unit grain.

Two sets of system characterization parameters have been proposed to completely describe the behavior of a given input workload on a target multiprocessor system. The first set, involving the three *static* parameters $(R_\infty, f_{1/2}, c_{1/2})$, describes the maximum asymptotic performance possible and the expected performance degradation as a result of static overheads in the input workload. The second set, consisting of the three *dynamic* parameters $(\psi_m(N), \psi_s(N), \psi_b(N))$, describes the run-time overheads resulting from dynamic interactions between concurrent processes along the three performance dimensions mentioned earlier as a function of execution parallelism. We have also presented a series of parameterized formulae that relate the quantitative characteristics of a workload to a realistic estimation of its performance via the system characterization parameters.

We have developed a family of workload emulation kernels that allow one to study the interaction of the different factors identified in an input workload and measure the incremental influence of each factor on performance. The measured data is used to calibrate the system characterization parameters described above. The MAD kernels, designed to calibrate the memory contention parameter $\psi_m(N)$, provide a testbed for the investigation of multiprocessor memory system performance under a variety of memory reference patterns. The SAD kernels, used to calibrate the synchronization parameter $\psi_s(N)$, facilitate the evaluation of the implementation

efficiency of synchronization operations based on spin locks and their sensitivity to algorithm characteristics. The BAD kernels, used to calibrate the barrier parameter $\psi_b(N)$, allow us to explore the efficiency of a barrier implementation and the losses accruing from barrier synchronization. We demonstrated the applicability of the system characterization methodology and the effectiveness of the workload emulation kernels on the Sequent Symmetry and BBN TC2000 commercial multiprocessors in studying the performance of several synthetic workloads.

We believe that our approach to performance characterization will serve to model performance with greater fidelity than exists in the current state of art, since it incorporates the effect of both static and dynamic influences in a workload execution. Further, the proposed methodology is independent of any particular multiprocessor architecture or application. Only a shared-variable programming paradigm is assumed, but no assumptions are made about the organization of the shared address space. Hence, our framework can not only be used to evaluate multiprocessors that provide physical shared memory, but also possible highly-parallel designs in the future supporting shared virtual memory over scalable interconnection networks.

Limitations of the Approach

Although the approach presented in this thesis can be successfully applied to characterize the performance of a wide variety of multiprocessor workloads, it has several limitations.

- The parallel processes in the workload are assumed to be statically assigned to processors with no run-time migration. Hence, the overhead of dynamic load balancing strategies, adopted on many multiprocessors, is not modeled.

- It is assumed that processes are assigned only one per processor with the total number of processes being less than the number of physical processors available. Although this is an accurate reflection of the structure of parallel programs on systems on which process creation and destruction are too expensive to be done frequently, many parallel machines have begun to support the implementation

of "light-weight processes" (or *threads*) that may time-share a single processor. If such parallel threads are used, then our model does nor account for the context-switch overheads associated with managing the threads.

- The model has limited applicability to heavily data-dependent parallel applications. For algorithms with data-dependent branches, probabilistic models are more appropriate. Although our workload generator allows the use of stochastic parameter values, the reliability of the measured performance will depend on the accuracy with which the the probability distributions chosen for workload parameters represent the real algorithm characteristics.

7.2 Directions for Future Research

The performance models and experimental results presented in this thesis establish a foundation for future study, but need to be extended in several ways.

Algorithms that exhibit essentially asynchronous execution of concurrent processes within a phase (only implicit synchronization in the form of mutually-exclusive access to a critical section are present) are considered in our performance studies. The unit grain based workload models should be expanded to include a larger variety of workload characterizations. For example, other forms of shared memory inter-process synchronizations such as those with explicit *event post/wait* or *message send/receive* semantics should be investigated. Also, the performance of alternate abstractions of the basic computation unit (BCU), such as a complex floating-point expression or a fundamental matrix operation, should provide interesting insight into the computing performance of a machine. In the same light, program models other than the multi-phase structured iterative algorithms studied here can be selected as the basis of system characterization.

Only a single memory access stream emanating from each processor was considered, since most available general-purpose multiprocessor systems provide only a single physical path from processor to memory. However, to include vector processors with multiple processor–memory paths in the scope of the proposed methodology, the

workload generation techniques can be adapted to provide multiple memory access streams and the performance model augmented to reflect the corresponding change.

The other most popular programming model, besides the shared-variable paradigm, uses message passing for inter-process communication and is normally used on distributed-memory multicomputers. The extension of our proposed framework to address the performance issues in the message-passing programming paradigm and characterize the behavior of message-passing workloads would, in some sense, impart a degree of completeness to the performance characterization methodology.

Finally, a particularly challenging proposition, in this respect, is the building of an integrated system characterization and application performance estimation environment. It would allow common performance experiments to be performed on different multiprocessor systems to characterize them and use the repository of data gathered, in conjunction with an application analyzer, to enable accurate estimation of application execution performance on a target architecture.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] A. Agarwal and A. Gupta. Memory reference characteristics of multiprocessor applications under Mach. In *Proceedings of the 1988 ACM SIGMETRICS Conference*, pages 422 - 433, 1988.

[2] Anant Agarwal and Mathews Cherian. Adaptive backoff synchronization techniques. In *Proceedings of the International Symposium on Computer Architecture*, pages 396 - 406, May 1989.

[3] A. Agarwal et al. An evaluation of directory schemes for cache coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 280 - 289, 1988.

[4] G.A. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483 - 485, 1967.

[5] Ames Research Laboratory. *The SLALOM Benchmark*, 1992.

[6] Thomas E. Anderson. The performance of spin lock alternatives for shared memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6 - 16, 1990.

[7] Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Transactions on Computers*, 38(12):1631 - 1644, December 1989.

[8] N.S. Arenstorf and H.F. Jordan. Comparing barrier algorithms. Technical Report 87-65, ICASE, NASA Langley Research Center, Hampton, VA, September 1987.

[9] T.S. Axelrod. Effects of synchronization barriers on multiprocessor performance. In *Parallel Computing 3*, pages 129 - 140. North-Holland, 1986.

[10] D.H. Bailey and J.T. Barton. The NAS kernel benchmark program. Technical report, NASA Technical Memorandum 86711, August 1985.

[11] F. Baskett and A. J. Smith. Interference in multiprocessor computer systems and interleaved memory. *Communications of the ACM*, 19:327 - 334, June 1976.

[12] S.J. Baylor and B.D. Rathi. A study of the memory reference behavior of engineering/scientific applications in parallel processors. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume 1, pages 78 - 82, 1989.

[13] BBN Advanced Computers Inc., Cambridge, Massachusetts. *Overview of the Butterfly GP1000*, November 1988.

[14] BBN Advanced Computers Inc., Cambridge, Massachusetts. *TC2000 Technical Product Summary*, November 1989.

[15] BBN Advanced Computers Inc., Cambridge, Massachusetts. *Inside the TC2000 Computer*, 1990.

[16] C.J. Beckmann and C. Polychronopolous. The effect of barrier synchronization and scheduling overhead on parallel loops. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume 2, pages 200 - 204, 1989.

[17] M. Berry. The Perfect Club benchmarks: Effective performance evaluation of supercomputers. *The International Journal of Supercomputer Applications*, 3:5 - 40, 1989.

[18] D.P. Bhandarkar. Analysis of memory interference in multiprocessors. *IEEE Transactions on Computers*, C-24:897 - 908, September 1975.

[19] Laxmi N. Bhuyan. An analysis of processor-memory interconnection networks. *IEEE Transactions on Computers*, C-34:279 - 283, March 1985.

[20] R. Bisiani and M. Ravishankar. PLUS: A distributed shared-memory system. In *Proc. 17th Intl. Symp. on Computer Architecture*, pages 115–124, 1990.

[21] E.D. Brooks. The butterfly barrier. *Int. Jour. of Parallel Programming*, 15(4):295 - 307, 1986.

[22] R. Bryant, P. Carini, H. Chang, and B. Rosenburg. Supporting structured shared virtual memory under Mach. In *Proc. USENIX Mach Symposium*, November 1991.

[23] W.H. Burkhardt. Aspects of multiprocessor systems. In *Proceedings of the COMPEURO '87 Conference*, pages 99 – 105, 1987.

[24] Ingrid Y. Butcher and Margaret L. Simmons. Measurement of memory access contentions in multiple vector processor systems. In *Proceedings of the Supercomputing '91 Conference*, pages 806 – 817, November 1991.

[25] B.L. Buzbee. The efficiency of parallel processing. *Computer Design*, June 1984.

[26] A. Cox and R. Fowler. The implementation of a coherent memory abstraction on a NUMA multiprocessor: Experiences with PLATINUM. In *Proc. 12th ACM Symp. on Operating System Principles*, pages 32–44, Dec. 1989.

[27] H.J. Curnow and B.A. Wichmann. A synthetic benchmark. *The Computer Journal*, 19(1):43 – 49, 1976.

[28] Zarka Cvetanovic. *Performance Analysis of Multiple-Processor Systems.* PhD thesis, University of Massachusetts, Amherst, Department of Computer Science, May 1986.

[29] George Cybenko, Lyle Kipp, Lynn Pointer, and David Kuck. Supercomputer performance evaluation and the Perfect benchmarks. Technical Report 965, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, Urbana, IL, March 1990.

[30] F. Darema-Rogers, G.F. Pfister, and K. So. Memory access patterns of parallel scientific programs. In *Proceedings of the 1987 ACM SIGMETRICS Conference*, pages 46 – 58, 1987.

[31] Chita R. Das and Laxmi N. Bhuyan. Bandwidth availability of multiple-bus multiprocessors. *IEEE Transactions on Computers*, C-34:918 – 926, October 1985.

[32] U. Detert and G. Hofemann. CRAY X-MP and Y-MP memory performance. In *Parallel Computing 17*, pages 579 – 590. North-Holland, 1991.

[33] J.J. Dongarra. The Linpack benchmark: An explanation. In *Supercomputing First International Conference Proceedings, Athens, Lecture Notes in Computer Science 297*, pages 456 – 473, 1987.

[34] J.J. Dongarra and A. Hinds. Comparison of the Cray X-MP/4, Fujitsu VP-200 and Hitachi S-810/20: An Argonne perspective. Technical Report ANL-85-19, MCS Division, Argonne National Laboratory, Argonne, IL, October 1985.

[35] J.J. Dongarra, J. Martin, and J. Worlton. Computer benchmarking: Paths and pitfalls. *IEEE Spectrum*, 24(7):38 - 43, July 1987.

[36] Thomas H. Dunigan. Kendall Square multiprocessor: Early experiences and performance. Technical Report ORNL/TM-12065, Oak Ridge National Laboratory, Oak Ridge, March 1992.

[37] J. Eggers and R.H. Katz. A characterization of sharing in parallel programs and its applicability to coherency protocol evaluation. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 373 - 382, 1988.

[38] Encore Computer Corporation. *Multimax Technical Summary*, 1986.

[39] W. Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. New York: Wiley, 1957.

[40] B. Fleisch and G. Popek. Mirage: A coherent distributed shared memory design. In *Proc. 12th ACM Symp. on Operating System Principles*, pages 211–223, December 1989.

[41] Philip J. Fleming and John J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *ACM Computing Practices*, 29:218 - 221, March 1986.

[42] Ian Foster, William Gropp, and Rick Stevens. The parallel scalability of the spectral transform method. Technical report, MCS Division, Argonne National Laboratory, Argonne, IL, January 1991.

[43] K.T. Fung and H.C. Torng. On the analysis of memory conflicts and bus contentions in a multiple microprocessor system. *IEEE Transactions on Computers*, C-27:28 - 37, January 1979.

[44] D. Gajski et al. Cedar construction of a large scale multiprocessor. Technical Report UIUCDCS-R-83-1123, University of Illinois, Department of Computer Science, February 1983.

[45] K. Gallivan, D. Gannon, W. Jalby, A. Malony, and H. Wijshoff. Experimentally characterizing the behavior of multiprocessor memory systems: A case study. *IEEE Transactions on Software Engineering*, 16(2):216 - 223, February 1990.

[46] E.F. Gehringer, D.P. Siewiorek, and Z. Segall. *Parallel Processing: The Cm* Experience.* MA: Digital, Bedford, 1987.

[47] E. Gelenbe. Asymptotic processing time of a model of parallel computation. In *Proc. of Nat. Comp. Conf., Las Vegas, NV*, November 1986.

[48] E. Gelenbe. *Multiprocessor Performance.* New York: Wiley, 1989.

[49] E. Gelenbe. Multiprocessor performance and the activity set model of program behavior. In J.L. Delhaye and E. Gelenbe, editors, *High Performance Computing*, pages 121 - 132. Amsterdam, The Netherlands: North-Holland, 1989.

[50] J.R. Goodman, M.K. Vernon, and P.J. Woest. Efficient synchronization primitives for large-scale cache coherent multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Language and Operating Systems*, pages 64 - 75, April 1989.

[51] A. Gottlieb, R. Grishman, C.P. Kruskal, K.M. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer - designing an MIMD shared memory parallel computer. *IEEE Transactions on Computers*, C-32(21):175 - 189, February 1983.

[52] Gary Graunke and Shreekant Thakkar. Synchronization algorithms for shared memory multiprocessors. *IEEE Computer*, pages 62 - 69, June 1990.

[53] R. Gupta. The fuzzy barrier: A mechanism for the high speed synchronization of processors. In *Third Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 54 - 63, April 1989.

[54] J.L. Gustafson. Amdahl's law re-evaluated. *Communications of the ACM*, 31:532 - 533, 1988.

[55] D.T. Harper III and J.R. Jump. Vector access performance in parallel memories using a skewed access scheme. *IEEE Transactions on Computers*, C-36(12):1440 - 1449, December 1987.

[56] P. Heidelberger and S. Lavenberg. Computer performance evaluation methodology. *IEEE Transactions on Computers*, C-33:1195 - 1220, December 1984.

[57] J. Helin and K. Kaski. Performance analysis of high-speed computers. In *Proceedings of the 1989 Supercomputing Conference*, pages 797 - 808, 1989.

[58] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *Int. J. Parallel Program*, 17(1):1 - 17, 1988.

[59] M. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 276 – 291, 1988.

[60] R.H. Hill. The art of benchmarking. *The Spang Robinson Report on Supercomputing and Parallel Processing*, (3), 1989.

[61] R.W. Hockney. Performance characterization of the HEP. In J.S. Kowalik, editor, *MIMD Computation: HEP Supercomputer and its Applications*. Cambridge: MIT Press, 1985.

[62] R.W. Hockney. $(r_\infty, n_{1/2}, s_{1/2})$ measurements on the 2-CPU CRAY X-MP. In *Parallel Computing 2*, pages 1 – 14. North-Holland, 1985.

[63] R.W. Hockney. Parameterization of computer performance. In *Parallel Computing 5*, pages 97 – 103. North-Holland, 1987.

[64] Intel Corporation. *A Touchstone DELTA System Description*, 1991.

[65] D.N. Jayasimha. Distributed synchronizers. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 23 – 27, 1988.

[66] Kendall Square Research. *KSR1*, 1992.

[67] C.P. Kruskal, L. Rudolph, and M. Snir. Efficient synchronization on multiprocessors with shared memory. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 218 – 228, 1986.

[68] C.P. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. In *Proceedings of the 1984 International Conference on Parallel Processing*, pages 236 – 240, 1984.

[69] David J. Kuck and Ahmed H. Sameh. A supercomputing performance evaluation plan. Technical Report 692, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, Urbana, IL, June 1987.

[70] D.J. Kuck et al. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM Symposium on Principles of Programming Languages*, pages 207 – 218, January 1981.

[71] H.T. Kung. Synchronized and synchronous parallel algorithms for multiprocessors. In J.F. Traub, editor, *Algorithms and Complexity: New Directions and Recent Results*. New York: Academic, 1976.

[72] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1), 1987.

[73] D.H. Lawrie and C.R. Vora. The prime memory system for array access. *IEEE Transactions on Computers*, 31(5):435 - 442, May 1982.

[74] D. Lee. Scrambled storage for parallel memory systems. In *Proceedings of the International Symposium on Computer Architecture*, pages 232 - 239, 1988.

[75] G. Lee, C.P. Kruskal, and D.J. Kuck. The effectiveness of combining in shared-memory parallel computers in the presence of "hot-spots". In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 11 - 12, August 1986.

[76] J. Lee and U. Ramachandran. Synchronization with multiprocessor cache. In *Proceedings of the International Symposium on Computer Architecture*, pages 27 - 37, May 1990.

[77] D. Lenowski et al. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148 - 159, May 1990.

[78] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. In *ACM Transactions on Computer Systems*, pages 321–359, November 1989.

[79] K. Li and R. Schaefer. A hypercube shared virtual memory system. In *Proc. Intl. Conf. on Parallel Processing*, pages 125–131, 1989.

[80] T. Lovett and S. Thakkar. The Symmetry multiprocessor system. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 303 - 310, August 1988.

[81] B.D. Lubachevsky. Synchronization barrier and related tools for shared memory parallel programming. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume 2, pages 175 - 179, August 1989.

[82] O. Lubeck, J. Moore, and R. Mendez. A benchmark comparison of three supercomputers: Fujitsu VP-200, Hitachi S-810/20 and Cray X-MP/2. *IEEE Computer*, 18, December 1985.

[83] S.F. Lundstrom. Applications cosiderations in the system design of highly concurrent multiprocessors. *IEEE Transactions on Computers*, C-36(11):1292 - 1309, November 1987.

[84] S. Madala and J.B. Sinclair. Performance of synchronous parallel algorithms with regular structures. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):105 - 116, January 1991.

[85] Allen D. Malony. *Performance Observability.* PhD thesis, University of Illinois at Urbana–Champaign, Department of Computer Science, October 1990.

[86] M.A. Marsan and M. Gerla. Markov models for multiple-bus multiprocessors. *IEEE Transactions on Computers*, C-31:239 - 248, March 1982.

[87] J.L. Martin. Performance evaluation: Applications and architectures. In *Second International Conference on Supercomputing*, pages 369 - 373, May 1987.

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

[89] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21 - 65, February 1991.

[90] H.E. Mizrahi, J.L. Baer, D. Lazowska, and J. Zahorjan. Extending the memory hierarchy into multiprocessor interconnection networks: A performance analysis. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume 1, pages 41 - 50, August 1989.

[91] J. Mohan. *Performance of Parallel Programs: model and analyses.* PhD thesis, Carnegie-Mellon University, Pittsburg, Department of Computer Science, July 1984.

[92] Arun Nanda and Lionel M. Ni. Benchmark workload generation and performance characterization of multiprocessors. In *Proceedings of the Supercomputing '92 Conference*, November 1992.

[93] Arun Nanda and Lionel M. Ni. MAD kernels: An experimental testbed to study multiprocessor memory system behavior. In *Proceedings of the 1992 International Conference on Parallel Processing*, August 1992.

[94] Arun Nanda and Lionel M. Ni. SAD kernels: A software tool to evaluate synchronization behavior of multiprocessors. In *Proceedings of the 1992 Computer Science and Applications Conference*, September 1992.

[95] Arun Nanda, Honda Shing, Ten-Hwan Tzen, and Lionel M. Ni. A replicate workload framework to study performance degradation in shared-memory multiprocessors. Technical Report MSU-CPS-ACS-18, Michigan State University, Department of Computer Science, January 1990.

[96] Arun Nanda, Honda Shing, Ten-Hwan Tzen, and Lionel M. Ni. Resource contention in shared-memory multiprocessors: A parameterized performance degradation model. *Journal of Parallel and Distributed Computing*, 12:313 – 328, July 1991.

[97] K.W. Neves and H.D. Simon. Supercomputer performance evaluation: Benchmarking applications on supercomputers. In *Second International Conference on Supercomputing*, pages 374 – 379, May 1987.

[98] A. Norton and E. Melton. A class of boolean linear transformations for conflict-free power-of-two stride access. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 247 – 254, August 1987.

[99] W. Oed and O. Lange. On the effective bandwidth of interleaved memories in vector processing systems. *IEEE Transactions on Computers*, C-34(10):949 – 957, October 1985.

[100] M.T. O'Keefe and H.G. Dietz. Hardware barrier synchronization: Static Barrier MIMD (SBM) and Dynamic Barrier MIMD (DBM). In *Proceedings of the 1990 International Conference on Parallel Processing*, volume 1, pages 35 – 46, 1990.

[101] J.H. Patel. Performance of processor-memory interconnections for multiprocessors. *IEEE Transactions on Computers*, C-30:771 – 780, October 1981.

[102] R. Perron and C. Mundie. The architecture of the Alliant FX/8 computer. In *Spring COMPCON '86*, pages 390 – 393, March 1986.

[103] B.L. Peuto and L.J. Shustek. An instruction timing model of CPU performance. In *Proc. Fourth Annual Symp. Comput. Architecture*, volume 5, pages 165 – 178, March 1977.

[104] G. Pfister, W.C. Brantley, D.A. George, S.L. Harvey, W.J. Kleinfelder, K.P. McAvliffe, T.A. Melton, V.A. Norton, and J. Weiss. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764 – 771, August 1985.

[105] G.F. Pfister and V.A. Norton. Hot-spot contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, C-34:943 - 948, October 1985.

[106] C. Polychronopolous. Compiler optimizations for enhancing parallelism and their impact on architecture design. *IEEE Transactions on Computers*, C-37(8):991 - 1004, August 1989.

[107] Ram Raghavan and John P. Hayes. On randomly interleaved memories. In *Proceedings of the Supercomputing '90 Conference*, pages 1 - 10, November 1990.

[108] U. Ramachandran, M. Ahamad, and M.Y.A. Khalil. Coherence of distributed shared memory: Unifying synchronization and transfer of data. In *Proc. Intl. Conf. on Parallel Processing*, volume II, pages 160–169, August 1989.

[109] R.D. Rettberg, W.R. Crowther, P.P. Carvey, and R.S. Tomlinson. The Monarch parallel processor hardware design. *IEEE Computer*, pages 18 - 30, April 1990.

[110] Rafael H. Saavedra-Barrera, Alan J. Smith, and Eugene Miya. Machine characterization based on an abstract high-level language machine. *IEEE Transactions on Computers*, 38:1659 - 1679, December 1989.

[111] R.H. Saavedra-Barrera. Machine characterization and benchmark performance prediction. Technical Report UCB/CSD 88/437, University of California, Berkeley, June 1989.

[112] R.G. Scarbourough and H.G. Kolsky. A vectorizing FORTRAN compiler. *IBM Journal of Research and Development*, 30(2), March 1986.

[113] Sequent Computer Systems Inc. *Balance 8000 System Technical Summary*, 1984.

[114] Sequent Computer Systems Inc. *Symmetry Technical Summary*, 1987.

[115] Leah J. Siegel, Howard J. Siegel, and Philip H. Swain. Performance measures for evaluating algorithms for SIMD machines. *IEEE Transactions on Software Engineering*, SE-8(4):319 - 330, July 1982.

[116] J.P. Singh, W. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. Technical report, Computer Systems Laboratory, Stanford University, CA, 1991.

[117] James E. Smith. Characterizing computer performance with a single number. *ACM Computing Practices*, 31:1202 – 1206, October 1988.

[118] Per Stenstrom. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, pages 12 – 24, June 1990.

[119] R. Thomas. Behavior of the Butterfly parallel processor in the presence of memory hot spots. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 46 – 50, 1986.

[120] J. Uniejewski. SPEC benchmark suite: Designed for today's advanced systems. *SPEC Newsletter*, 1, 1989.

[121] Dalibor F. Vrsalovic, Daniel P. Siewiorek, Zary Z. Segall, and Edward F. Gehringer. Performance prediction and calibration for a class of multiprocessors. *IEEE Transactions on Computers*, 37:1353 – 1364, November 1988.

[122] W.H. Ware. The ultimate computer. *IEEE Spectrum*, pages 84 – 91, March 1982.

[123] R.P. Weicker. Dhrystone: A synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013 – 1030, October 1984.

[124] S. Weiss. An aperiodic storage scheme to reduce memory conflicts in vector processors. In *Proceedings of the International Symposium on Computer Architecture*, pages 380 – 385, 1989.

[125] J. Worlton. Understanding supercomputer benchmarks. *Datamation*, pages 121 – 129, 1984.

[126] P.C. Yew, S.N. Tzeng, and D.H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 51 – 58, August 1987.

[127] Xiaodong Zhang. Performance measurement and modeling to evaluate various effects on a shared-memory multiprocessor. *IEEE Transactions on Software Engineering*, 17(1):87 – 93, January 1991.