REDUCING COMMUNICATION COST IN SCALABLE SHARED MEMORY SYSTEMS

by

Gheith Ali Abandah

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1998

Doctoral Committee:

Professor Edward S. Davidson, Chair
Assistant Professor Peter M. Chen
Assistant Professor Emad S. Ebbini
Assistant Professor Steven K. Reinhardt
Professor Kang G. Shin

ABSTRACT

REDUCING COMMUNICATION COST IN SCALABLE SHARED MEMORY SYSTEMS

by
Gheith Ali Abandah

Chair: Edward S. Davidson

Distributed shared-memory systems provide scalable performance and a convenient model for parallel programming. However, their non-uniform memory latency often makes it difficult to develop efficient parallel applications. Future systems should reduce communication cost to achieve better programmability and performance. We have developed a methodology, and implemented a suite of tools, to guide the search for improved codes and systems. As the result of one such search, we recommend a remote data caching technique that significantly reduces communication cost.

We analyze applications by instrumenting their assembly-code sources. During execution, an instrumented application pipes a detailed trace to configuration independent (CIAT) and configuration dependent (CDAT) analysis tools. CIAT characterizes inherent application characteristics that do not change from one configuration to another, including working sets, concurrency, sharing behavior, and communication patterns, variation over time, slack, and locality. CDAT simulates the trace on a particular hardware model, and is easily retargeted to new systems. CIAT is faster than detailed simulation; however, CDAT directly provides more information about a specific configuration. The combination of the two tools constitutes a comprehensive and efficient methodology. We calibrate existing systems using carefully designed microbenchmarks that characterize local and remote memory, producer-consumer communication involving two or more processors, and contention when multiple processors utilize memory and interconnect.

This dissertation describes these tools and illustrates their use by characterizing a wide range of applications and assessing the effects of architectural and technological advances on the performance of HP/Convex Exemplar systems, evaluates strengths and weaknesses of current system approaches, and recommends solutions.

CDAT analysis of three CC-NUMA system approaches shows that current systems reduce communication cost by minimizing either remote latency or remote communication frequency. We describe four architecturally varied systems that are technologically similar to the low remote latency SGI Origin 2000, but incorporate additional techniques for reducing the number of remote communications. Using CCAT, a CDAT-like simulator that models communication contention, we evaluate the worthiness of these techniques. Superior performance is reached when processors supply cached clean data, or when a remote data cache is introduced that participates in the local bus protocol.

To my loving wife Dana, and caring parents Ali and Nawal.

ACKNOWLEDGMENTS

My special thanks to my committee. To Edward Davidson for his guidance and support throughout my program. To Peter Chen, Emad Ebbini, Steven Reinhardt, and Kang Shin for their helpful questions and suggestions.

I would like to thank the University of Jordan, Fulbright Foundation, Ford Motor Company, and the National Science Foundation for their generous financial support. Parallel computer time was provided by the University of Michigan's Center for Parallel Computing (CPC) and Caltech's Center for Advanced Computing Research. CPC is partially funded by NSF grants CDA–92–14296 and ACI–9619020.

Parts of this research was initiated in 1996 while I was a research intern in HP Laboratories in Palo Alto, California. I would like to thank all the people I have worked with there. In particular, Rajiv Gupta and Josep Ferrandiz for their guidance, Tom Rokicki for his assistance in implementing some of the tools, Lucy Cherkasova for her constructive discussions and comments, and Milon Mackey for providing the TPC traces.

I would like to thank Isom Crawford and Herb Rothmund of the HP/Convex Technology Center for providing the Exemplar NPB implementation.

Finally, to Dana for her unlimited encouragement and emotional support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure

CHAPTER 1
INTRODUCTION

This introductory chapter specifies the objectives of this research and provides some background and terminology to enable presenting the following chapters in a smooth way. Section 1.1 overviews some of the main concepts in distributed shared memory multiprocessing and discusses some alternative architectural approaches. Section 1.2 outlines some performance and programmability issues that have motivated this research, Section 1.3 specifies the main dissertation research objectives, Section 1.4 outlines our approach to accomplishing these objectives, and Section 1.5 surveys some related work.

1.1 Distributed Shared Memory Multiprocessors

Distributed-memory systems are parallel processors that use high-bandwidth, low-latency interconnection networks to connect powerful processing nodes which contain processors and memory [CSG98, Hwa93]. The interconnection networks provide the communication channels through which nodes exchange data and coordinate their work in solving parallel applications.

Distributed-memory systems reduce some bottlenecks that limit performance in systems with central memories. Thus, they have the potential for scaling in size and performance. When the distributed memory is node private, as in distributed-memory *multicomputers*, processing nodes communicate and exchange data using explicit message passing. Although message-passing applications can be efficient and portable, they are hard to develop and, because of the typically large setup overhead per message, they have problems in exploiting fine-grain parallelism [For94, CLR94]. On the other hand, distributed shared-memory (DSM) *multiprocessors* provide programmers convenience of memory sharing with one global address space, some portion of which is found in each node [PTM96]. Uniprocessor applications are often easily ported to DSM multiprocessors, and their performance can then be incrementally tuned to exploit the available parallelism. Moreover, tuned shared-memory applications can be as efficient as message-passing applications [CLR94], provided that support is available for high bandwidth block transfers.

Due to the increasing gap between processor speed and memory speed, DSM systems use one or more levels of caches that are often kept consistent by using one of many cache coherence protocols [CSG98, LW95].

Various software and hardware techniques have been proposed for implementing DSM multiprocessors [PTM96]. Commercial implementations usually employ hardware techniques because of their higher performance and easier programmability. Three commercial multiprocessors using hardware techniques, ordered by increasing degree of hardware support for coherent data replication and migration, are Cray Research T3D [KS93], Convex SPP1000 [Bre95], and Kendall Square Research KSR1 [FBR93, BD94]. These three multiprocessors represent three distinct paradigms for implementing DSM systems.

The T3D interconnects dual-processor processing nodes using a 3-dimensional torus. The T3D is a *non-uniform memory access* (NUMA) multiprocessor. A processor can access memory in remote nodes using load and store instructions that optionally update its cache. Coherence is not maintained among the processor caches or with remote memory, e.g., when a processor updates data in its cache, the data copies, if any, in other caches and remote memory are not affected. Hence, T3D shared-memory applications frequently use direct memory instructions to bypass the cache, and use explicit synchronization operations to order stores and loads among the processors. The T3D incorporates special hardware support to achieve fast synchronization.

The SPP1000 interconnects eight-processor processing nodes using four rings. The SPP1000 is a *cache-coherent non-uniform memory access* (CC-NUMA) multiprocessor. It has a cache-coherence controller (CCC) that uses a directory-based cache coherence protocol to enable coherent data migration and replication, e.g., when a cache line is updated, the CCC invalidates other copies and insures that a processor request always gets a copy of the most recent data. In addition to the processor caches, each SPP1000 node reserves a portion of its memory for use as a large interconnect cache (IC) for reducing the number of remote-memory accesses. Referenced remote data is copied into the IC in addition to the processor cache, thus future accesses to the referenced remote data that result in processor cache misses could be served locally by the IC.

The KSR1 interconnects single-processor processing nodes using a hierarchy of rings with up to 32 nodes in each lowest level ring; a higher level ring can connect up to 32 lower level rings. The KSR1 is a *cache-only memory architecture* (COMA). Similar to the SPP1000, the KSR1 uses a directory-based coherence protocol. Additionally, the KSR1 "memory" is implemented in hardware as a set of *attraction memories*, i.e. a large cache in each node. The KSR1 moves and replicates data among the attraction memories dynamically and coherently in response to processor accesses.

Several vendors are adopting the CC-NUMA architecture for building their new high-end servers. CC-NUMA achieves a nice balance between NUMA's programming complexity and COMA's hardware complexity. Examples of new CC-NUMA systems are: HP/Convex SPP2000 [BA97], Sequent

NUMA-Q [LC96], and SGI Origin 2000 [LL97].

1.2 DSM Performance and Programmability

Computer architects increasingly rely on application characteristics for insight in designing cost-effective systems. This is true in the early design stages as well as later stages. In the early design stages, architects face a large and diverse design space. Some of the early design decisions are: node and system organization, number of processors per node, target communication latency and bandwidth, memory caching, and coherence and consistency protocols. They need to select a design that best fits their performance, scalability, availability, security, programmability, portability, and manageability objectives for the target application market.

Additionally, programmers involved in developing and tuning shared-memory applications need tools for analyzing applications to identify performance bottlenecks and to get hints for improving performance. An application analysis tool's utility depends on its ability to provide relevant characteristics in an accurate and timely manner.

Figure 1.1, which shows the effects of machine size and remote communication latency on performance, illustrates one example of a design trade-off. The remote latency is the latency for satisfying a processor cache miss from another node. The figure shows a trend that is typical of many parallel applications. This figure is based on data found by simulating traces of a 256×256 blocked matrix multiplication on a particular CC-NUMA system which has 4 processors per node and supports coherence protocols similar to the Stanford DASH protocols [LLG^{+}92].

As the number of nodes involved in solving the problem increases, the execution time decreases.

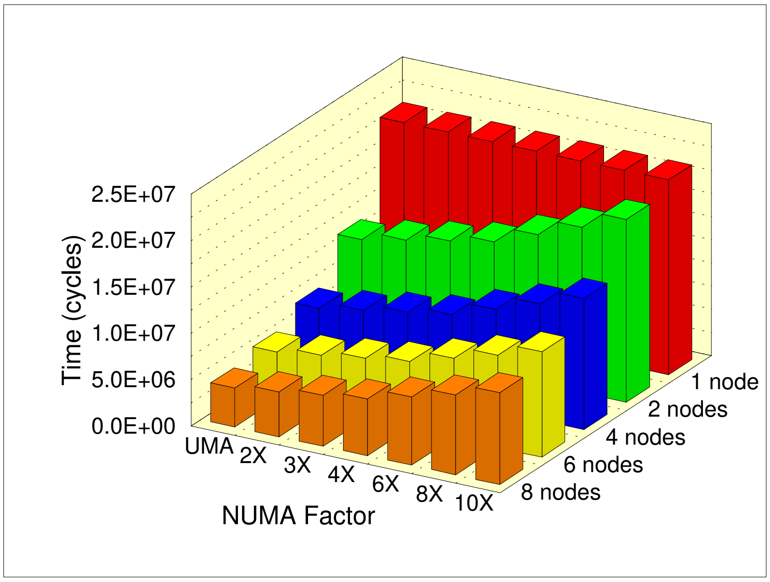


Figure 1.1: Effects of machine size and remote latency.

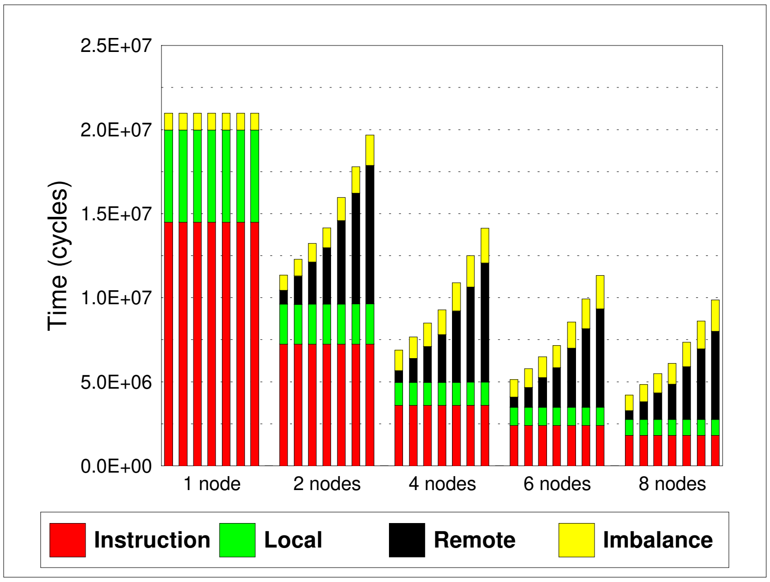


Figure 1.2: Profile of the execution time for a variety of remote latencies.

For any given number of nodes, as the latency of accessing data in remote nodes increases, the execution time increases. The figure shows the effects of increasing the remote latency from the local, i.e. same node access, time (UMA) to 10 times the local time (10X). Note that all the single node accesses are local; thus, the single node time is independent of the remote latency. On the other hand, the system cost increases with larger machine size and with lower remote latency.

Figure 1.2 provides an explanation of the trends illustrated in Figure 1.1 by breaking down the execution time into several distinct components. It shows profiles of the execution time of the 5 machine sizes for the various remote latencies. The average time spent in executing useful instructions and satisfying local misses decreases as the machine size increases. However, the relative contribution of remote misses and the imbalance time, i.e. time spent waiting for other processors to complete some portion of their work, each increases as the machine size increases which severely limits the performance of the larger machine sizes.

For a particular machine size of more than one node, the average time spent in satisfying remote misses and the imbalance time both increase as the remote latency increases. The combined effects result in the possibility of achieving better performance with fewer nodes and lower remote latency than with more nodes and higher remote latency.

Our experience with the Convex SPP1000 and the SPP1600 (which uses a more advanced processor) is that many applications do not scale beyond 8 processors (the size of one node). Many users of this system find the effort of tuning their applications for scalability, by careful localization of memory references, so overwhelming that they abandon tuning these applications. In the University of Michigan's Center for Parallel Computing (CPC), for example, most users run application codes that do not scale beyond 8 processors. Therefore, the SPP1600 is typically partitioned into several subcomplexes, none of which contains more than 8 processors.

For programmers to succeed in developing efficient and scalable applications on DSM systems, they need accurate and relevant information about their applications and the DSM systems on which they run. They need means to analyze their applications in order to identify performance bottlenecks and consequently more easily tune these applications for better performance. The knowledge of an application's characteristics is often insufficient to tune it for a particular system. The programmer also needs to know enough information about the system's performance characteristics in order to exploit the system's strengths and avoid its weaknesses.

In our evaluation of the SPP1000 (presented in Chapter 5), we found that its remote latency is about 4 to 7 times the local latency. This makes the remote communication cost high in applications with a large number of remote misses. In order to build DSM systems that feature good programmability and scalability, it is very important to find effective approaches to reducing the communication cost. Systems with remote communication cost that is close to the local cost relieve the programmer from having to invest significant effort in localizing memory references.

1.3 Dissertation Research Objectives

The main objectives of this research are:

- To develop a methodology for analyzing shared-memory applications and illustrate its use to support development of efficient DSM applications and systems.
- To develop and use a methodology for evaluating and calibrating existing DSM systems.
- To propose and evaluate system-level design techniques for reducing the cost of communication among the processors, with a primary focus on the system architecture and protocols.

Although application tuning, e.g., localizing memory references and software prefetching, can also significantly reduce communication cost, it is not within the scope of this research.

1.4 Dissertation Outline

To support our methodology of analyzing applications and evaluating design options, we have developed a set of tools for collecting and analyzing traces of shared-memory applications. These tools and our analysis methodology are presented in Chapter 2. The tool for collecting traces, SMAIT, has two parts: a `perl` script program for instrumenting the assembly language files, and a run-time library that is compiled with the application object code. During program execution, the instrumented instructions generate one trace per thread by performing calls to the run-time library. These traces can either be dumped to a file or piped to the analysis tools for on-the-fly analysis.

Piping enables full analysis of long execution periods (hundreds of millions of instructions) without excessive storage requirements.

Three main analysis tools have been developed: CIAT, for performing configuration independent analysis, and CDAT and CCAT, for configuration dependent analysis. CIAT helps us understand and quantify an application's inherent behavior with respect to memory instructions, synchronization, communication, and private and shared memory access. CIAT is unique in its generic analysis approach which is not specific to any particular machine configuration or coherence protocol, i.e., it requires no machine description and reports no artifactual behavior which is caused by some aspect of a machine's configuration or its protocols.

CDAT is a system-level simulator that observes the memory and I/O references generated by each processor. It pays special attention to those references that generate second-level cache misses since these misses cause the system traffic. CDAT is a flexible simulation tool that is easily retargeted to a new machine configuration by reading a configuration file. The configuration file allows the system designer to select among a wide range of design options. For a particular system configuration, CDAT generates information that describes the application's cache misses, system traffic, latencies, and amount of data transferred.

CCAT, like CDAT, is also a system-level simulator. However, it is more detailed and models contention on memory and interconnects. CCAT is now targeted to memory-based directory CC-NUMA systems and assumes an advanced processor model.

These analysis tools also generate traces that may be used by TDAT, our time distribution analysis tool, to analyze the variations in an application's behavior over time.

In Chapter 3, we present a thorough characterization of the eight applications that are used in this research with a variety of problem sizes and number of processors. We use CIAT and TDAT to characterize the inherent properties of these applications, and show how configuration independent analysis is used to characterize eight aspects of application behavior: general characteristics, working sets, concurrency, communication patterns, communication variation over time, communication slack, communication locality, and sharing behavior. We demonstrate that our approach provides an efficient, clean, and informative characterization.

In Chapter 4, we move on to present an experiment-based methodology for characterizing the memory, communication, scheduling, and synchronization performance of existing DSM systems. We extend existing microbenchmarking techniques to characterize the important aspects of a DSM system. In particular, we present carefully designed microbenchmarks to characterize the performance of the local and remote memory, producer-consumer communication involving two or more processors, and the effects on performance when multiple processors contend for utilization of the distributed memory and the interconnection network.

Advances in microarchitecture, packaging, and manufacturing processes enable designers to build new systems with higher performance and scalability. In Chapter 5, we use the microbenchmarks of Chapter 4 to contrast the memory and communication performance of two generations of the Convex Exemplar scalable parallel processing system. The SPP1000 and SPP2000 have significant architectural and implementation differences, but maintain upward binary compatibility. The SPP2000 employs manufacturing and packaging advances to obtain shorter system interconnects with wider data paths and improved functionality, thereby reducing the latency and increasing the bandwidth of remote communication. Although the memory latency is not greatly improved, newer out-of-order execution processors coupled with nonblocking caches achieve much higher memory bandwidth. The SPP2000 has a richer system interconnect topology that allows scalability to a larger number of processors. The SPP2000 also employs innovations in its coherence protocols to improve synchronization and communication performance. We characterize the performance effects of these changes and identify some remaining inefficiencies that future systems should address.

In Chapter 6, we present a comparative study of three important CC-NUMA implementations, the Stanford DASH, the Convex SPP1000, and the SGI Origin 2000, to find strengths and weaknesses of current implementations. Although the three systems share many similarities, they have significant differences that translate into large performance differences. For example, they differ in the number of processors per node, processor cache configuration, memory consistency model, location of memory in the node, and cache-coherence protocol.

In this study, we evaluate the effects of these differences on cache misses, local bus traffic, internode traffic, miss time in various component types, and amount of data transferred. We use detailed multiprocessor traces of selected shared-memory applications to drive CDAT. We first use models that represent each of the three systems as closely as possible. Our results indicate large performance differences among the three systems, which are largely due to the use of different component speeds and sizes. We then put the three systems on the same technological level by assigning them components of similar size and speed, but preserve their individual organizations and coherence protocol differences. Although the Origin 2000 still has the least average remote access time, it spends the longest time satisfying its misses because the majority of its misses are satisfied remotely. From this study, we have concluded that the Origin 2000 has a potential for superior performance, provided that the ratio of remote misses is reduced.

In Chapter 7, we recommend a scheme that reduces the remote miss ratio without increasing latency. We use CCAT to evaluate this scheme and show that this scheme always improves the execution time of the parallel applications used. This scheme offers as much as a 50% reduction in the execution time, and is relatively more effective with “problematic” applications.

In Chapter 8, we present our conclusions about the methodology, the application case studies, performance of existing systems, the strengths and weaknesses found in the three major system approaches that we have studied, and the studied techniques for reducing the communication costs of future systems. We also outline some future work.

1.5 Related Work

In this section, we present a survey of some related work and outline some of the similarities and differences with respect to our work.

1.5.1 Performance Collection

Characterizing shared-memory applications involves performance collection and performance analysis. Performance collection often involves collecting detailed information about the application's memory references. There are several classes of techniques for performance collection, each with its own advantages and limitations. Hardware monitors, e.g., the VAX microcode monitor [EC84], Convex CXpa [CXp93], and the IBM POWER performance monitor [WCNSH94] provide low-level information using event counters, but require special hardware support and so tend to be system-specific. ATUM [ASH86] generates a compressed trace file for post analysis, but is also based on hardware support since it enables collecting address traces by modifying the microcode.

Code instrumentation, e.g., ATOM [SE94], RYO [ZK95], and Pixie [Smi91], enables collecting various performance data and traces by instrumenting either the assembly or the object file of a uniprocessor application. This technique often requires source code availability, perturbs the execution, and cannot be used with applications that generate code dynamically, e.g. data base systems. MPTRACE [EKKL90] uses assembly code instrumentation to collect traces of shared-memory multiprocessor applications.

Simulation is also used in performance data collection, for example, Proteus [BDCW91], Tango Lite [GH92], and SimOS [RHWG95]. Similar to code instrumentation, simulation enables collecting performance data of various kinds, but is often slower. SimOS collects traces for activities within the operating system in addition to the user-space activities and does not require source code availability.

Dynamic code translation can be used in performance collection, e.g., DEC FX!32 [FX] which enables executing some x86 programs on an Alpha workstation. In this technique, traces can be collected by instrumenting, during execution time, the basic blocks of an application.

Our trace collection tool was developed to enable collecting traces of some Convex SPP1000 and SPP1600 applications. SMAIT uses code instrumentation techniques. It is similar to MPTRACE in its ability to collect multiprocessor traces by instrumenting the assembly files of an application. However, its technique of replacing the instrumented instructions by subroutine calls, makes it relatively flexible. SMAIT addresses the execution perturbation problem by providing a light partial instrumentation option for collecting timing information. Unlike hardware monitors and SimOS, SMAIT does require source code availability and cannot trace activity within the operating system, unless the operating system is made available for instrumentation.

1.5.2 Performance Analysis

Available parallel performance analysis tools have mainly been developed for analyzing message-passing applications, e.g., Pablo [RAN+93], Medea [CMM+95], and Paradyn [MCC+95b]. There is, however, some work that focuses on characterizing shared-memory applications. Singh et al. demonstrated that it is often difficult to model the communication of parallel algorithms analytically [SRG94]. They suggested developing general-purpose simulation tools to obtain empirical information for supporting the design of parallel algorithms.

The tools developed in this research address the shortage of tools for analyzing shared-memory applications, and the limited scope of the tools that do exist. They enable mechanical characterization of a wider range of shared-memory application properties than any previous tool suite, and they use a cleaner approach to characterizing inherent application properties without being biased toward any particular system configuration. A judicious combination of our configuration independent and configuration dependent analyses constitutes a comprehensive and efficient methodology. A sharper contrast between our approach and other related work is given in Chapter 3.

There are several studies that combine source-code analysis with configuration dependent analysis to characterize shared-memory applications [SWG92, RSG93, WOT+95]. Woo et al. have characterized several aspects of the SPLASH-2 suite of parallel applications [WOT+95]. Their characterization includes load balance, working sets, communication to computation ratio, system traffic, and sharing. They used execution-driven simulation with the Tango Lite [GH92] tracing tool. In order to capture some of the fundamental properties of SPLASH-2, they adjusted model parameters between low and high values. In contrast, our configuration independent analysis characterizes these properties more naturely and efficiently, without resorting to a series of specific configurations. For example, the above studies, unlike CIAT, characterize communication by measuring the coherence traffic, which is a function of the application's inherent communication and the simulated system's cache configuration and coherence protocol. CIAT's characterization of the inherent communication is fast and clean because it tracks only the alternation of accesses on each memory location and does not use a specific cache coherence protocol or model the states of multiple caches. CIAT characterizes an application's working sets in one experiment and is accurate even with applications that do not exhibit good spatial locality. CIAT's concurrency characterization includes the serial fraction, load imbalance, and resource contention in addition to speedup.

Chandra et al. also used simulation to characterize the performance of a collection of applications [CLR94]. Their main objective was to analyze where time is spent in message-passing versus shared-memory programs. Perl and Sites [PS96] have studied some Windows NT applications on Alpha PCs. Their study includes analyzing the application bandwidth requirements, characterizing the memory access patterns, and analyzing application sensitivity to cache size.

To get insight in designing interconnection networks, Chodnekar et al. analyzed the time distribution and locality of communication events in some message-passing and shared-memory applications [CSV+97]. CIAT and TDAT characterize the time distribution of communication events as a function of time in addition to reporting the cumulative distribution function of the event rates. CIAT characterizes communication locality by characterizing the communication between each processor pair, not just characterizing the communication from one particular processor to the other processors. Thus, CIAT and TDAT present more useful characterizations for understanding and tuning shared-memory applications.

Leutenegger and Dias [LD93] analyzed the TPC-C disk accesses to model its disk access patterns and showed that TPC-C can achieve close to linear speedup in a distributed system when some read-only data is replicated.

1.5.3 System Calibration

Microbenchmarking has been used in many studies to characterize low-level performance of uniprocessor and multiprocessor systems [SS95, McC95a, MS96, MSSAD93, GGJ+90, HLK97].

Saavedra and Smith have used microbenchmarking to characterize the configuration and performance of the cache and TLB subsystems in uniprocessors [SS95]. They have shown that microbenchmark characterization can be used to parameterize performance models that predict the performance of simple applications to within 10% of the actual run times. Using his STREAM microbenchmarks, McCalpin measured the memory bandwidth of many high-performance systems and noticed that the ratio of CPU speed to memory speed is growing rapidly [McC95a]. McVoy has observed that special care and attention must be given to designing microbenchmarks that accurately measure the memory latency and bandwidth of modern processors which allow multiple outstanding misses [MS96]. His microbenchmark suite, `lmbench`, measures many aspects of processor and system performance.

Although there are some studies that have used microbenchmarks to characterize the memory performance of shared-memory multiprocessors [GGJ+90, SGC93, HLK97], the microbenchmarks presented here offer wide coverage of the memory and communication performance for DSM systems. These microbenchmarks characterize the latency and bandwidth of local and shared accesses as functions of the access pattern and distance, and characterize the overheads due to preserving cache coherence and contention resulting from concurrent accessing.

1.5.4 Distributed Shared Memory Architecture

In Section 1.1 and Chapter 6, we discuss the implementations of some DSM systems. To conserve space, we do not repeat this discussion here. But it is worthwhile to mention that many of the techniques and approaches used in the DSM systems that we focus on were developed in the Stanford DASH [LLG+92], MIT Alewife [ABC+95], and SCI standard [SCI93] projects. Detailed surveys of distributed shared-memory concepts and systems are found in [LW95, PTM96, CSG98].

Many CC-NUMA systems, existing commercial machines as well as research prototypes, use caching to reduce the number of remote capacity misses [AB97, LC96, NAB+95, LLG+92, FW97, MD98].

Caching remote data in local specialized caches is a popular approach used mainly to reduce the cost of capacity misses. Many approaches have used DRAMs to serve as large interconnect caches (Exemplar [AB97], NUMA-Q [LC96], S3.mp [NAB+95], and NUMA-RC [ZT97]), or SRAMs to serve as fast interconnect caches (DASH [LLG+92]), or both (R-NUMA [FW97] and VC-NUMA [MD98]). In Chapter 7, we evaluate the worthiness of an SRAM interconnect cache in reducing the number of remote communication misses. Unlike DASH and VC-NUMA, our implementation is compatible with the MESI cache coherence protocols supported by modern processors. Additionally, unlike R-NUMA's block cache, our implementation caches remote lines on load misses and does not cache remote lines on store misses, and therefore offers a better reduction of the remote communication cost.

CHAPTER 2

METHODOLOGY AND TOOLS

This chapter describes our methodology for characterizing shared-memory applications and evaluating scalable shared-memory systems, and describes the tool suite used to support it. These tools are then used to carry out the case studies in Chapters 3, 6, and 7. Microbenchmarking is introduced in Chapter 4 and used in Chapter 5. Microbenchmark results are also used to support the case studies in Chapters 6 and 7.

2.1 Overview

This methodology is based on a suite of five flexible tools that enable collecting and analyzing detailed traces of shared memory applications as shown in Figure 2.1. The *Shared-Memory Application Instrumentation Tool* (SMAIT) is used for trace collection. Three tools are used for trace analysis: the *Configuration Independent Analysis Tool* (CIAT), the *Configuration Dependent Analysis Tool* (CDAT), and the *Communication Contention Analysis Tool* (CCAT). The *Time Distribution Analysis Tool* (TDAT) characterizes event time distributions.

In Figure 2.1, a shared-memory multiprocessor (MP) is used to execute and analyze instrumented application codes. However, the analysis tools can also accept trace files generated by other means. SMAIT supports *execution-driven analysis* by (i) piping the traces directly to one of the analysis tools, instead of generating trace files, and (ii) accepting feedback to control the execution timing on the traced multiprocessor according to the simulated system configuration or CIAT's analysis model. Execution-driven analysis enables analyzing longer execution periods by using piping to eliminate the need to store huge trace files and using feedback to avoid the non-deterministic behavior of some applications.

Usually, we first use CIAT to characterize the application's inherent characteristics, as outlined in Section 3.2. Then we use CDAT or CCAT to characterize other aspects and to find the application performance and generated traffic on a particular system configuration. CDAT and CCAT are used to characterize things like cache misses and false sharing that depend on configuration parameters,

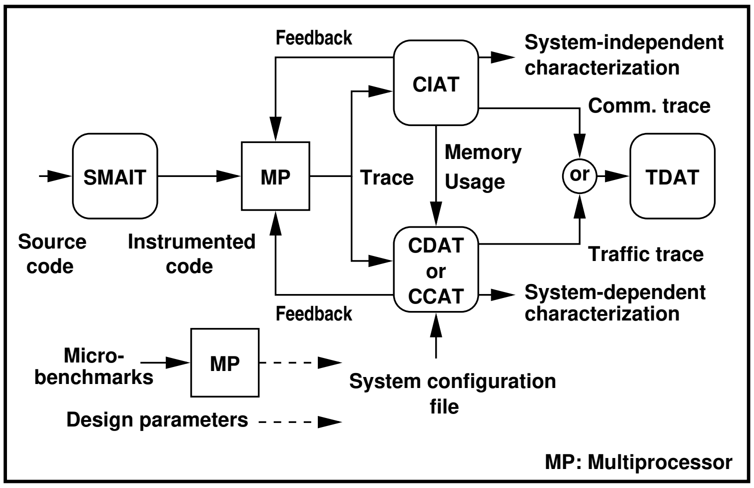


Figure 2.1: A comprehensive shared-memory application analysis methodology.

such as cache size and width.

CDAT and CCAT are system level simulators. CCAT models contention on busses, memory banks, and interconnection links, and has a more detailed processor model. Although CDAT handles each cache miss as an atomic transaction, its relative simplicity gives it a five-fold speed advantage over CCAT, enabling the analysis of longer execution periods.

CIAT analyzes inherent application properties, i.e., those that do not change from one configuration to another, thus relieving CDAT and CCAT from repeating this analysis for every configuration. CDAT and CCAT, which use fairly detailed models of the system coherence protocol and system state, are generally slower than CIAT.

In addition to its system-independent characterization report file, CIAT generates a detailed memory usage file that provides access information for each accessed memory page. CDAT and CCAT may or may not require access to this memory usage file, depending on which policy is specified in the configuration file for mapping memory pages to the simulated memory banks. CIAT optionally generates a trace of the communication events that is analyzed by TDAT to characterize the communication variation over time. TDAT is also used to analyze CDAT's and CCAT's traffic traces.

The system configuration to be analyzed by CDAT or CCAT is specified through a file that selects from the supported architectural options and specifies component sizes and speeds. To enable performance analysis of application runs on an existing system, we use a suite of microbenchmarks to calibrate the system. This calibration is then used to fill in a configuration file for that system. To evaluate runs on a proposed design, the designer fills in a configuration file with the proposed design parameters.

The characterization reported by these tools is used to support application tuning, early design of scalable shared-memory systems, parameterizing synthetic workload generators, comparing alternative design options, and investigating new design approaches. The suite of microbenchmarks enables calibrating existing systems and evaluating their strengths and weaknesses.

SMAIT is described in Section 2.2. Section 2.3 introduces trace analysis techniques that are common to the three analysis tools. Sections 2.4, 2.5, and 2.6 describe the CIAT, CDAT, and CCAT trace analysis tools, respectively. Section 2.7 describes TDAT. Finally, Section 2.8 outlines our assessment of the accuracy of these tools. Further detail on these tools is reported in [Aba96].

2.2 Trace Collection (SMAIT)

SMAIT is based on RYO [ZK95], a tool developed by Zucker and Karp for instrumenting PA-RISC [Hew94] instruction sequences. RYO is a set of `awk` scripts that enable replacing individual machine instructions with calls to user written subroutines.

SMAIT is designed to enable collecting traces of multi-threaded shared-memory parallel applications. SMAIT has two parts: a `perl` script program for instrumenting PA-RISC assembly language files, and a run-time library that is linked with the instrumented program. The `perl` script program replaces some PA-RISC instructions with calls to the run-time library subroutines. During program execution, the run-time library generates one trace file per thread. SMAIT provides three levels of instrumentation:

- At *Level 1*, SMAIT instruments the procedure call instructions for some I/O, thread management, and synchronization subroutines. At this level, SMAIT enables collecting traces for the I/O stream and timing information. The run-time library generates one *call record* whenever an instrumented call instruction is executed. A call record contains the call type, time before and after the procedure call, and four argument fields.
- At *Level 2*, SMAIT additionally instruments all load and store instructions to trace the data stream. The run-time library generates one *memory-access record* whenever an instrumented memory-access instruction is executed. A memory-access record contains the type of the instruction and the virtual address.
- At *Level 3*, SMAIT additionally instruments all branch instructions and the first instruction of every procedure in order to trace the instruction stream. The run-time library generates one *branch record* for each taken branch. A branch record contains the virtual address of the branch instruction plus 4 and the virtual address of the branch target. Four is added to the branch instruction address to account for the instruction in the delay slot after the branch instruction which, in PA-RISC architecture, is fetched and conditionally executed. The run-time library also generates one record whenever the first instruction of an instrumented procedure is executed. This record contains the virtual address of the procedure start. At this

level, the memory-access records have an additional field that specifies the location of the memory instruction relative to the predecessor traced branch or memory instruction, so as to indicate the number of nontraced intervening instructions.

When Level 1 is used, the instrumented code is lightly perturbed and runs near the original uninstrumented speed. This level is mainly used for collecting timing information. When Level 2 or 3 is used, the instrumented code is heavily perturbed and runs about 70 to 100 times slower than the original uninstrumented code.

2.3 Trace Analysis

This is an introductory section that describes some trace analysis techniques that are common among the three trace analysis tools.

Each tool accepts two sets of trace files. Each set is made up of p trace files coming from p threads of execution. The first set, called *call traces*, is optional and contains traces of the I/O and synchronization calls. The second set, called *detailed traces*, is required and contains the call records, data stream records and, if available, the instruction stream records. Although all the records in the call traces are also in the detailed traces, the call traces are used because they usually come from a less perturbed execution and their time stamps are closer to the timing of an uninstrumented execution. The call traces are collected using SMAIT instrumentation Level 1, and the detailed traces are collected using SMAIT instrumentation Level 2 or 3.

The tools assume that the traces come from an application with one or more *execution phases* where each phase has its own properties. Currently, the supported phases are *serial/parallel* phases and *user-defined* phases. In a serial phase, Thread 0 is the only active thread and other threads, for a multi-thread execution, are idle. In a parallel phase, multiple threads are active. The tools recognize transitions between serial and parallel phases from the trace records of the thread-spawn and thread-join calls that activate and deactivate threads between serial and parallel phases. The user-defined phases are recognized when the tools encounter special *marker* records [Aba96]. The marker records can be generated by instrumenting the high-level source code.

The tools perform analysis per phase and report characterization statistics at the end of each phase. They also report the characterization statistics aggregated over all phases at the end.

CIAT and CDAT manage one *pseudo clock* per thread in order to interleave the processing of multiple traces. The clocks are initialized to zero at the start of the first phase. A thread clock is incremented by one whenever an instruction is processed for that thread. Additionally, CDAT on second-level cache misses increments the thread clock by the number of cycles of the miss latency. The clocks are synchronized at the end of each phase, and as they emerge from a synchronization barrier, to the value of the largest clock.

Within a serial phase, CIAT and CDAT process the trace records of Thread 0 until reaching a thread spawn call. Within a parallel phase, both tools process trace records from all the available threads until reaching a thread join call. At each step, the next trace record selected for processing is chosen from the thread with the smallest clock value. In case multiple threads have this smallest clock value, trace records are selected for processing in an ascending order according to the respective thread IDs.

Although CCAT uses the same front-end engine to parse the input traces, it is an event driven simulator that consumes the trace on demand according to the status of the simulated processors. CCAT has an event scheduling core that is adapted from the SimPack toolkit [Fis95].

The tools also support traces that do not contain information about thread management and synchronization, e.g., the TPC traces described in Chapter 3. The tools support analyzing T trace files taken from T different processes by interleaving these traces on p processors.

For these traces, the tools use the time stamps in the trace synchronization records to break the traces into logical *slices*. A slice is a sequence of memory-access, branch, and system call records that are surrounded by two synchronization records which contain the start and end time stamps of the slice, as observed when the trace was collected. The tools sort the slices into a list according to their start times and schedule them on the available processors. If the slice at the list head, A, has a start time that is larger than the end time of an earlier slice, B, that is still active, then A will not be scheduled until B completes.

Although this conservative scheduling correctly captures inter-process communication; however its conservative ordering of the slices lengthens the execution time, and consequently we cannot accurately characterize some configuration independent performance properties (of the TPC benchmarks) like concurrency, communication variation over time, and communication slack (Chapter 3), nor configuration dependent execution time and contention (Chapter 7).

2.4 Configuration Independent Analysis (CIAT)

CIAT is intended to capture the inherent application characteristics that do not change from one multiprocessor configuration to another. A multiprocessor configuration specifies the way that processors are clustered in a hierarchy, the interconnection topology, the coherence protocols, the cache configurations, and the sizes and speeds of the multiprocessor components.

CIAT uses a model similar to the PRAM model [FW78] which assumes that p processors can execute p instructions concurrently and each instruction takes a fixed time. Therefore, CIAT keeps track of time in instruction units. CIAT interleaves the analysis of multiple thread traces on p processors according to the thread spawn and join calls, and obeys the restrictions of the lock and barrier synchronization calls. CIAT additionally maintains internal data structures for the accessed memory locations that are used by its characterization algorithms (more detail is given in Chapter 3).

CIAT uses many counters for counting various events and uses a memory structure to keep track of data and code accesses. The phase statistics are reported in a report file at the end of each phase and the aggregate statistics are reported in the report file at the end of the last phase. Additionally, at the end of the last phase, CIAT scans the memory structure and reports memory usage statistics in the report file and generates a memory usage file that summarizes the memory usage of each touched page.

CIAT provides the option of generating a trace file of the key communication events. This trace file enables conducting time distribution analysis of the application's communication patterns using TDAT, as described in Section 2.7.

2.5 Configuration Dependent Analysis (CDAT)

CDAT is a system-level simulator that is intended to investigate a large design space at the early design stages of new scalable shared-memory systems. CDAT is faster than cycle-by-cycle simulators, e.g. CCAT, due to using the simplification assumptions outlined in the next paragraph. CDAT describes the system traffic that would be generated by an application on a particular multiprocessor configuration. CDAT is a trace-driven simulator that has cache, memory, bus, and internode interconnection models. These models can be arranged in a hierarchical configuration as specified in the configuration file.

CDAT assumes that each instruction takes one cycle to complete as long as it does not generate a miss in the second-level cache. When an instruction generates a miss, CDAT simulates the system-level transactions (transactions among the system components) to satisfy this miss. This is done in accordance with the cache coherence protocol as specified in the configuration file. In this case, CDAT assumes that the instruction takes as many cycles as there are in the critical path to satisfy the miss. To keep the models simple and fast, CDAT assumes that each miss is atomic and that there is no contention on system resources.

CDAT uses many counters for counting various system events to report event counts and flow parameters that describe many aspects of the application behavior on the specified configuration. These abstractions are also used to parameterize workload generators. The phase statistics are reported in the report file at the end of every phase and the aggregate statistics are reported at the end of the last phase.

Additionally, CDAT (and CCAT) can generate two types of traces; *general* and *detailed*. The general trace contains the request and return transactions generated by the processors and I/O devices. Processor requests are generated on cache misses, and processor returns are generated on cache replacements. I/O requests are generated on DMA activities. The detailed trace contains records of all the signals that are needed to satisfy processor requests and returns. More detail on the format of the two trace types is reported in [Aba96]. The general trace is used in time distribution analysis (TDAT) and to drive other system-level simulators (developed and used in Hewlett-Packard Laboratories, Palo Alto). The detailed trace is used for debugging CDAT.

CDAT modeling of shared-memory systems is sufficiently flexible to enable experimenting with a wide variety of design options. For example, through the configuration file options, users may select a desired coherence protocol variant, specify system component sizes and speeds, and specify how processors and memory banks are clustered and interconnected. Additionally, other system-level simulators that use CDAT's general trace have the freedom to use various hardware implementations.

CDAT and CCAT have several policies for mapping virtual memory pages to the distributed physical memory banks. Some of these policies rely on the information gathered by CIAT about memory usage. They read the memory usage file and map the virtual pages used into the physical memory banks according to the policy specified in the configuration file. CDAT and CCAT have the ability to allocate private pages in the local node, interleave shared pages across nodes, and replicate code pages in multiple nodes.

2.6 Communication Contention Analysis (CCAT)

CCAT is a system-level simulator that has an aggressive processor model for simulating the high traffic conditions that are common with modern processors. CCAT models contention on shared resources; it models individual buses, memory banks, and interconnection network links as facilities. A system signal utilizes a facility by occupying it for a specified number of cycles. If the facility is busy, the signal is queued on the facility queue and served later in a FIFO style [Fis95].

The processor executes a specified number of instructions per cycle (IPC) when it is not stalled. The processor can have up to some specified number of outstanding misses, and thus it is able to overlap the latencies of multiple cache misses and generate high system traffic. The processor supports sequential memory consistency [Lam79] and stalls on the following events:

1. It attempts to fetch an instruction and misses in the secondary cache. The processor remains stalled until this code miss is satisfied.

2. It has a data miss and its outstanding request buffer is full. Each processor has an architected number of outstanding request buffer entries. This buffer keeps track of the secondary cache misses and replacements. The processor stays stalled until some outstanding requests are satisfied and there are enough free entries. Note that a miss that forces the replacement of a dirty line needs two entries; one for the miss request and a second for the replacement request.

3. The window of instructions between the oldest unsatisfied memory instruction[1] and the current instruction equals the number of entries in the instruction reorder buffer (ROB). Since the processor graduates performed instructions in program order, an unsatisfied memory instruction causes all successor instructions to be held in the ROB. Thus when the ROB is full, the processor cannot issue new instructions and consequently stalls.

4. In some modern processors, the ROB has dedicated entries for memory instructions. Hence, when the window of instructions in the ROB includes a number of memory instructions that equals the number of entries dedicated for this type of instruction, the processor stalls until some instructions graduate and there is a free memory ROB entry.

Since the processor model does not stall due to data dependencies, the rate of system traffic caused by this processor model is an upper bound on the actual traffic, and thus is appropriate for evaluating systems under heavy traffic conditions. However, the CCAT processor model can be parameterized to cause lighter traffic by, as an extreme example, selecting 1 IPC and a one-entry ROB, which forces a serialization of the instruction stream as if every instruction is dependent on the preceding instruction and no dependence is tolerated by the processor. This selection can give a lower bound on the traffic rates with normal application workloads. Alternatively, some intermediate selection can be chosen according to some estimate of how well the processor is able to tolerate dependence.

2.7 Time Distribution Analysis (TDAT)

TDAT is a separate time distribution analysis tool. TDAT is used to analyze event traces generated by CIAT, CDAT, or CCAT. TDAT accepts event trace files stored on disk, or alternatively accepts event trace pipes from any of these three analysis tools for on-the-fly analysis. The event trace is an ASCII file that has one record per event. Each record starts with a clock field and may have other optional fields.

TDAT puts the events into bins according to the clock field. The bin width can be specified by the user. TDAT reports the number of events in each bin, the number of bins, the average, minimum, and maximum event rates, and the event density and distribution functions.

2.8 Tool Validation

We have used a combination of techniques to verify the correctness of the tools described in this chapter.

SMAIT was validated by comparing the generated traces against the source assembly code.

[1] A memory instruction is unsatisfied if it has an outstanding miss request.

The three trace analysis tools were written in C using a cautious programming style with frequent use of assertion statements. An assertion statement acts as a diagnostic point; it verifies that the program state at that point is as expected [KR92]. This frequent use of assertion statements helped us to expose and fix many programming bugs and algorithmic errors throughout the development process.

We have also used simple synthetic traces to validate various aspects of these tools. By running these tools on synthetic traces that have predetermined analysis results, we were able to validate the output report files.

The CIAT communication trace and the CDAT and CCAT detailed trace were used to validate the respective tools. In particular, the CCAT detailed trace was very useful in verifying that CCAT correctly simulates the target cache coherence protocols.

CIAT analysis of the SPLASH2 suite of benchmarks (presented in Chapter 3) was verified against the configuration dependent analysis presented in [WOT+95]. While the results of the two approaches generally agree, there are some justifiable differences due to using two different hardware platforms and the fundamental limitations and differences of the two approaches.

We have also verified CDAT's results against a report generated by Convex CXpa [CXp93]. CXpa collects performance statistics using hardware monitors on the SPP1600, which uses the PA7200 processor [CHK+96]. For a single node configuration running a 256×256 blocked matrix multiplication program, CDAT reported 14% more data misses. We believe that CDAT reported more misses because it does not model the PA7200's assist cache that eliminates some of the conflict misses.

As presented in Chapter 7, CCAT simulates systems similar to the SGI Origin 2000 [LL97]. CCAT's local and remote miss latencies agree with the latencies measured using microbenchmarks and reported in [HLK97].

CHAPTER 3

APPLICATION CHARACTERIZATION CASE STUDIES WITH CIAT

In this chapter, we present the collection of shared-memory applications that are used in this research. We characterize these applications using our configuration independent analysis approach to get a general and thorough understanding of their inherent properties. We also use these case studies to illustrate the advantages of this approach.

3.1 Introduction

As described in Chapter 2, configuration dependent analysis runs an application trace on a model of a target system configuration to predict its performance on the target system. Configuration independent analysis, on the other hand, uses only a parallel execution trace and extracts the inherent configuration independent application characteristics.

While configuration dependent analysis is repeated for every target configuration, configuration independent analysis is only performed once per combination of problem size and number of processors. Configuration independent analysis provides a general understanding of an application's inherent properties and, as demonstrated in following chapters, enables explaining the results of configuration dependent analysis.

This chapter demonstrates that configuration independent analysis is a useful approach to characterizing several important aspects of shared-memory applications, it is more efficient in characterizing certain properties than configuration dependent analysis, and it can capture some application properties that are easily missed by configuration dependent analysis on a fixed configuration. Using a judicious combination of both configuration dependent and configuration independent analysis provides an efficient and comprehensive methodology for characterizing shared-memory applications.

In this chapter, we use our configuration independent analysis tool, CIAT, to characterize eight shared-memory application benchmarks drawn from the Stanford SPLASH-2, NAS Parallel Benchmarks (NPB), and Transaction Processing Performance Council (TPC) application suites.

In the rest of this chapter, Section 3.2 describes some important shared-memory application characteristics and why knowledge of them is useful to architects and software engineers. Section 3.3 describes the eight case-study benchmarks. Section 3.4 contains eight subsections, each describing how we used configuration independent analysis to characterize a particular aspect of the benchmarks, interprets the results of the characterizations, and discusses the advantages and disadvantages of this approach. Section 3.5 presents some conclusions.

3.2 Shared-Memory Application Characteristics

This chapter addresses eight characteristics of shared-memory applications:

- *General characteristics* of the application include dynamic instruction count, number of distinct touched instructions, a parallel execution profile (serial and parallel phases), number of synchronization barriers and locks, I/O traffic, and percentage of memory instructions (by type).

- The *working set* of an application in an execution interval is the number of distinct memory locations accessed in this interval [Den68]. The working set often changes over time and may be hierarchical, e.g., multiple working sets may be accessed iteratively and collectively constitute a larger working set. The working set size is a measure of the application's temporal locality, which affects its cache performance. A large working set indicates a low degree of temporal locality; when the working set size is larger than the cache size, capacity misses occur. Characterizing the working sets of target applications is important in selecting the cache size of a new system. This characterization is also useful to programmers; for example, when the working set size is larger than the cache size, the programmer can improve the application performance by reducing its working set, e.g., by segmenting a matrix computation into blocks [Wol96].

- The amount of *concurrency* available in an application influences how well the application performance scales as more processors are used. An application with high concurrency has the potential to efficiently utilize a large number of processors. Section 3.4.3 discusses factors that affect the concurrency of shared-memory applications. The amount of available concurrency in the target applications provides the system designer with insight in selecting the machine size and the number of processors to be clustered in each node. Characterizing and reducing factors that adversely affect an application's concurrency is valuable for improving application scalability.

- *Communication* in a shared-memory multiprocessor occurs implicitly when multiple processors access shared memory locations. Communication occurs in several patterns, depending on the type and order of the accesses and the number of processors involved. One example is the producer-consumer pattern where one processor stores to a memory location and another then loads from this location. Section 3.4.4 presents a classification of the memory access patterns in shared-memory applications that cause communication traffic. For a system designer, since coherence misses and traffic are a function of the communication patterns and the system configuration, characterizing the volume of the various communication patterns is particularly important. A successful system designer designs a system that efficiently supports the common communication patterns of the target applications. Additionally, characterizing the communication patterns in an application can help the programmer to avoid expensive patterns.

- *Communication variation over time* is as important as characterizing overall communication volume. Communication can exhibit uniform, bursty, random, periodic, or other complex behavior. Expressing how the application behaves over time enables identifying and characterizing the program segments most responsible for the application's communication. For a system designer, the knowledge of the distribution function of the communication rates is important in specifying appropriate bandwidth for the system interconnects.

- *Communication slack* is the temporal distance between producing a new value and referencing it by other processors. Characterizing the communication slack is useful to predict the potential utility of prefetching. For an application with large slack, prefetching can be scheduled early to reduce processor stalls due to cache miss.

- *Communication locality* is a measure of the distance between the communicating processors. For example, some applications have local communication where a processor tends to communicate with its near neighbors, and others have uniform communication where a processor communicates with all other processors. In application tuning, communication locality is useful for assigning threads to physical processors. In system design, it is useful in selecting the system organization and the interconnection topology. As an example, consider an application where one thread produces shared data that is consumed by all other threads. In a system with mesh-style interconnect, better performance can be achieved when the producing thread is run on a centrally located processor. For an architect designing for this application, supporting a broadcast capability may be a viable design choice.

- The *sharing behavior* of an application refers to which memory locations are shared and how. A *real shared* memory location is a location in the shared space that is accessed by multiple processors during the program execution. A *private* location is accessed by only one processor. A *shared access* is an access to a real shared location and a *private access* is an access to a private location. Notice that not all shared accesses cause communication. For example, consider a processor performing multiple loads of one shared location after a store performed by another processor. Using perfect caches, only the first load is a communication event that requires coherence traffic to copy the data from the producer's cache to the local cache. The subsequent loads are not considered communication events because they are satisfied from the local cache. Characterizing the sharing behavior of an application helps the programmer to localize data, i.e. to map data to memory in a way that minimizes access time. For example, mapping private data to local memory reduces the private access time, and mapping shared data to the node where it is most referenced generally reduces the average shared access time.

3.3 Applications

We have analyzed Radix, FFT, LU, and Cholesky from SPLASH-2 [WOT+95], CG and SP from NPB [B+94], and TPC benchmarks C and D [TPC92, TPC95]. SPLASH-2 consists of 8 applications and 4 computational kernels drawn from scientific, engineering, and graphics computing. NPB consists of 5 kernels and 3 pseudo-applications that mimic the computation and data movement characteristics of large-scale computational fluid dynamic applications (an earlier report characterizes 5 benchmarks of NPB using CIAT and CDAT [Aba97]). The TPC benchmarks are intended to compare commercial database platforms. The following is a short description of the eight benchmarks. These particular applications were selected because they represent a wide range of applications.

Radix is an integer sort kernel that iterates on radix r digits of the keys. In each iteration, a processor partially sorts its assigned keys by creating a local histogram. The local histograms are then accumulated into a global histogram that is used to permute the keys into a new array for the next iteration (two arrays are used alternately). Our experiments used a radix of 1024.

FFT is a one-dimensional n-point complex Fast Fourier Transform kernel optimized to minimize interprocessor communication. The data is organized as $\sqrt{n} \times \sqrt{n}$ matrices and each processor is responsible of $\sqrt{n}/p$ contiguous rows. The kernel's all-to-all communication occurs in three matrix transpose steps. Every processor transposes a contiguous submatrix of $\sqrt{n}/p \times \sqrt{n}/p$ from every processor. If every processor starts by transposing from Processor 0's rows, high contention occurs. Hence, to minimize contention, each processor begins by transposing a submatrix from the next processor's set of rows.

LU is a kernel that factors a dense $n \times n$ matrix into the product of a lower triangular and an upper triangular matrix. The matrix is divided into $B \times B$ blocks. The blocks are partitioned among the processors, where each processor updates its blocks. To reduce false-sharing and conflict

Benchmark	*Problem Size I*		*Problem Size II*	
	Problem Size	Total Instructions (M)	Problem Size	Total Instructions (G)
Radix	256K integers	88	2M integers	0.63
FFT	64K points	30	1M points	0.51
LU	256×256	80	512×512	0.57
Cholesky	`tk15.O` file	860	`tk29.O` file	2.13
CG	$1,400$/15	147	$14,000$/15	2.43 (0.75)
SP	16^3/100	611	64^3/400	189 (1.61)
TPC-C	NA	NA	16 users	(1.0)
TPC-D	NA	NA	1 GB data base	(2.8)

Table 3.1: Sizes of the two sets of problems analyzed. The two numbers specifying the problem size of CG and SP refer to the problem size and the number of iterations, respectively. The total instructions is the total number of instructions executed using 32 processors. Values in parenthesis show the number of actually analyzed instructions for the partially analyzed problems.

misses, elements within a block are allocated contiguously using 2-D scatter decomposition. Our experiments used $B = 16$.

Cholesky is a kernel that uses blocked Cholesky factorization to factor a sparse matrix into the product of a lower triangular matrix and its transpose.

CG is an iterative kernel that uses the *conjugate gradient* method to compute an approximation to the smallest eigenvalue of a sparse, symmetric positive definite matrix. Table 3.1 shows the order of the matrix and the number of iterations of the two problem sizes analyzed.

SP is a simulated application that solves systems of equations resulting from an approximately factored implicit finite-difference discretization of the Navier-Stokes equations. SP solves *scalar pentadiagonal* systems resulting from full diagonalization of the approximately factored scheme.

TPC-C is an on-line transaction processing benchmark that simulates an environment where multiple operators execute transactions against a database. The TPC-C analysis presented in this chapter is based on a 1 Giga instruction trace that represents the benchmark execution.

TPC-D is a decision support application benchmark that performs complex and long-running queries against large databases. TPC-D is comprised of 17 queries that differ in complexity and run time. Each query often undergoes multiple phases with varying disk I/O rates. The TPC-D analysis presented in this chapter is for a 2.8 Giga instruction trace representing the third phase of Query 3 where most of the query time is spent. Compared with other queries, although Query 3 takes a moderate run time, it has a high disk I/O and communication rates [TPC]. However, other queries can be characterized similarly.

Table 3.1 shows the problem sizes analyzed in this study. The scientific benchmarks were analyzed using two problem sizes on a range of Processors from 1 to 32. Problem size II has about

Feature	SPP1600 Data
Number of processors	32 in 4 nodes
Processor	PA 7200 @ 120 MHz
Main memory	1024 MB per node
OS version	SPP-UX 4.2
Fortran compiler	Convex FC 9.5
C compiler	Convex CC 6.5

Table 3.2: The SPP1600 host configuration.

one order of magnitude more instructions than problem size I.

The SPLASH-2 benchmarks were developed at Stanford University to facilitate shared-memory multiprocessor research and are written in C. The NPB are specified algorithmically so that computer vendors can implement them on a wide range of parallel machines. We analyzed the Convex Exemplar [Bre95] implementation of NPB, which is written in Fortran. The performance of an earlier version of this implementation is reported in [SB95]. However, to get a general characterization of these benchmarks, we undid some of the Exemplar-specific optimizations, e.g., we removed the extra code that localizes shared data into the node-private memory of the 8-processor nodes. The six scientific benchmarks were instrumented, compiled, and analyzed on a 4-node Exemplar SPP1600 multiprocessor. Table 3.2 shows the configuration of this system.

The six scientific benchmarks are multi-threaded. Each starts with a serial *initialization phase* where only Thread 0 is active to setup the problem. After the initialization phase, p threads are spawned to run on the p available processors in the main *parallel phase*. The problem is partitioned among the available processors and each processor is responsible for its part. The threads coordinate their work by using synchronization barriers and mutual-exclusion regions controlled by locks. At the end of the parallel phase, the multiple threads join and only Thread 0 remains active in the *wrap-up phase* to do validation and reporting.

Unless otherwise specified, the reported scientific application characteristics are for the parallel phase using 32 processors. For CG and SP, their characteristics do not change from one iteration to another in the parallel phase. Hence, to save analysis time, we performed our analysis of problem size II only from the program start to the end of the second iteration in the parallel phase. We report the characteristics of the second iteration as representative of the whole parallel phase.

The TPC traces were collected at HP Labs on a 4-CPU HP server running a commercial database environment. In this configuration, parallelism is exploited using multiple processes that communicate using shared memory and semaphore operations. The TPC-C trace is composed of trace files for 45 processes, and the TPC-D trace is composed of trace files for 23 processes. The operating system serves the active processes by performing context switching on the limited number of CPUs. To capture the characteristics of each process, CIAT analyzes the TPC traces by running each process trace on a dedicated processor. Consequently, 45 processors are used to analyze TPC-C and 23 processors are used for TPC-D.

These TPC traces include records for the user-space memory instructions, taken branches, system calls, and synchronization instructions, e.g. load-and-clear-word. However, they do not contain information about context switching. Thus, it is impossible to analyze these traces in the exact occurrence order. For such cases, CIAT uses the conservative trace scheduling algorithm described in Section 2.3. And consequently we do not characterize the concurrency, communication variation over time, and communication slack of the TPC benchmarks.

3.4 Characterization Results

The following eight subsections are devoted to the eight targeted characteristics. Each subsection describes the configuration independent analysis used, states the advantages and disadvantages of the approach, and presents and interprets the results of characterizing the eight benchmarks.

3.4.1 General Characteristics

Each memory instruction accesses one or more consecutive locations, where the size of each location is one byte, e.g., the load-word instruction accesses four locations. CIAT maintains a *hash table* that has an entry for each accessed location. Each entry holds the location's status bits and access information. One status bit, the code bit, is set when the location is accessed by an instruction fetch. At the end of a trace, CIAT sums the set code bits to find the touched code size and sums the clear code bits to find the touched data size. The code and data sizes of the eight benchmarks are shown in Table 3.3. While Cholesky and SP have about 85 KB of touched code, the other four scientific kernels have less. However, the TPC benchmarks touch hundreds of code kilobytes in the traced period.

The data size is one to three orders of magnitude larger than the code size, where LU has the smallest data size and CG has the largest. TPC-C's data size is larger than TPC-D's data size mainly due to the differences in their disk access patterns. Since TPC-C processes random transactions that generate short random-access disk reads and writes, it uses a large disk cache in memory to improve disk access time. However, TPC-D queries generate long sequential disk reads with little data reuse, consequently it uses a limited number of memory buffers to temporarily hold and process read disk chunks.

Table 3.3 also shows five aggregate characteristics of the parallel phase: the number of executed instructions, percentage of memory instructions, average instructions executed per taken branch, the number of barriers, and the number of locks. CG has the highest percentage of memory instructions and the largest data size. Consequently, it is a benchmark that can potentially stress the

	Radix	FFT	LU	Cholesky	CG	SP	TPC-C	TPC-D
Code size (KB)	9	18	13	88	23	83	820	200
Data size (MB)	17	49	2.0	46	90	30	47	3.5
	(2.9)	(3.2)	(0.52)	(21)	(9.0)	(0.57)		
No. of instrs. in (M)	110	480	540	2,000	2,000	190,000	1,000	2,900
	(22)	(27)	(69)	(770)	(130)	(600)		
Memory Instrs.	29%	29%	40%	26%	51%	35%	36%	48%
Instructions/ taken branches	33	16	25	24	21	68	10	10
	(15)	(16)	(24)	(24)	(21)	(70)		
No. of barriers	11	7	67	4	1,185	1,600	0	0
	(11)	(7)	(35)	(4)	(735)	(400)		
No. of locks	442	32	32	72,026	0	0	7.9×10^5	5.3×10^5
	(442)	(32)	(32)	(54,419)	(0)	(0)		

Table 3.3: General characteristics of problem size II using 32 processors. Characteristics of problem size I are given in parenthesis if significantly different.

processor cache. This high memory instruction percentage is due to simple reduction operations on long vectors. The six scientific benchmarks have more instructions between taken branches than the TPC benchmarks. Infrequent branching is typical of scientific applications in which the application spends most of its time in loops with large loop bodies and one backward branch. Table 3.3 indicates that the scientific benchmarks use little synchronization; among them, CG has the fewest instructions per barrier and Cholesky has by far the most locks. The TPC traces have relatively many synchronization events, mainly load-and-clear-word instructions and `semop` system calls. However, the trace collection perturbs execution and the traces have elevated synchronization rates relative to unperturbed execution. In order to minimize the effect of this behavior on our analysis, we ignore all accesses to the synchronization variables when characterizing communication and sharing. Thus, our characterization of the TPC communication only includes communication on normal shared memory and does not include communication on synchronization variables.

Table 3.4 shows some statistics for the disk I/O activity in the TPC traces. TPC-C accesses relatively little data per disk read and write access; most TPC-D disk accesses are 64 KB reads. Although TPC-D has more disk I/O bytes per instruction, its disk accesses are predictable which enables hiding their latency by prefetching.

Figure 3.1 shows the percentage of the byte, half-word (2 bytes), word (4 bytes), float (single-precision floating-point), and double (double-precision floating-point) load and store instructions. The left graph is for problem size I, and the right graph is for problem size II. While the percentage of byte and half-word memory instructions is negligible in the scientific benchmarks, it is 23% in TPC-C and 32% in TPC-D. CG has the largest percentage of load instructions due to its reduction operations. The average for the remaining benchmarks is about 2 loads per store, i.e. typically two

	TPC-C	TPC-D
No. of disk read calls	18,000	2,800
Average read chunk	1.7 KB	63 KB
No. of disk write calls	4,000	81
Average write chunk	1.5 KB	33 B
Disk I/O bytes per instruction	0.037	0.061

Table 3.4: Disk I/O in TPC-C and TPC-D.

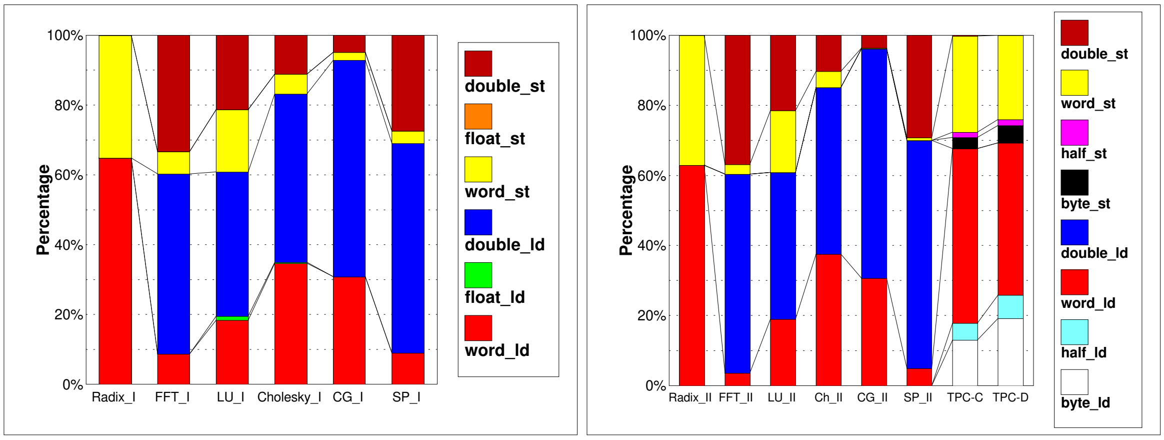


Figure 3.1: Percentage of the memory instructions according to the instruction type (load or store) and type of data accessed (byte, half-word, word, float, or double). Left chart is for problem size I and right chart is for problem size II.

operands to one result.

Except for Radix, an integer kernel, more than 58% of the memory instructions in the scientific benchmarks manipulate double values and almost all the rest manipulate word objects. Cholesky uses a large percentage of load-word instructions to find the indices of the sparse matrix non-zero elements.

When the problem is scaled up, some instruction sequences are executed more frequently, e.g. the body of some loops, while other instruction sequences are not affected. Consequently, there are some differences in the percentage of memory instructions between size I and II. One obvious change is the lower percentage of store-word instructions in size II. Store-word instructions are often associated with instruction sequences that are executed a fixed number of times, e.g. saving the previous state at the procedure entry for a procedure that is called a fixed number of times.

3.4.2 Working Sets

The size of a shared-memory application's working set is sometimes characterized by conducting multiple simulation experiments using a fully-associative cache with LRU replacement policy [RSG93, WOT+95]. Each experiment uses one cache size and measures the cache miss ratio.

A graph of the cache miss ratio versus cache size is used to deduce the working set sizes from the graph knees. A knee at cache size C indicates that there is a working set of size $\leq C$. This is a time consuming procedure since it involves multiple simulation experiments. Although there are efficient techniques for simulating multiple cache sizes in one experiment [KHW91], these techniques are not used with shared-memory multiprocessor simulations because of the interaction among the processors due to coherence traffic which is some function of the cache size. Additionally, this procedure does not differentiate between coherence and capacity misses, and may over-estimate the working set size when using cache lines that are larger than the size of the individually accessed data elements, especially with applications that have poor spatial locality.

CIAT characterizes the inherent working sets of an application in one experiment using the *access age* of the load and store accesses. The access age of an instruction accessing location x is the total size of the distinct locations accessed between this access and the previous access of x, inclusively. By definition, the access age is set to ∞ for the first access of x. For example, the sequence of word accesses (A, B, C, A, A, B, B) has access ages $(\infty, \infty, \infty, 12, 4, 12, 4)$.

The access age predicts the performance of a fully-associative cache with LRU replacement policy and a line size that equals the size of the smallest accessed element. For an S-byte cache, every access with age $\leq S$ is a hit, every access with age $= \infty$ generates a compulsory miss, and every access with age $> S$ generates a capacity miss.

To find the access age, CIAT uses a counter for the accessed bytes. For each access, the counter is incremented by the number of accessed bytes and the incremented value is stored in the hash table entry corresponding to the accessed location. When a location is reaccessed, the *access reach* is found as the current counter value (i) minus the stored value (j) at the location's hash table entry. The access age is then calculated as the access reach minus the number of repeated bytes within this reach. The number of repeated bytes within reach k is stored in element k of the vector **Rep**, therefore

$$\text{Age} = (i - j) - \mathbf{Rep}[i - j].$$

Rep is maintained as follows: For each access to a previously accessed location, its size s is added to the $(i - j)$th element of vector **New**. Then **Rep** is updated by accumulating the new repeated bytes and aging previous repetitions (a shift by s) as shown in the two loops below. The first loop accumulates new repetitions (after initializing *sum* to zero), and the second loop shifts by s.

$$\left.\begin{array}{lcl} \mathit{sum} & += & \mathbf{New}[k], \\ \mathbf{Rep}[k] & += & \mathit{sum}, \\ \mathbf{New}[k] & = & 0 \end{array}\right\} \forall k : s, s+1, \ldots, i.$$

$$\begin{array}{lcll} \mathbf{Rep}[k] & = & \mathbf{Rep}[k-s] & ; \forall k : i, i-1, \ldots, s, \\ \mathbf{Rep}[k] & = & 0 & ; \forall k : s-1, s-2, \ldots, 1. \end{array}$$

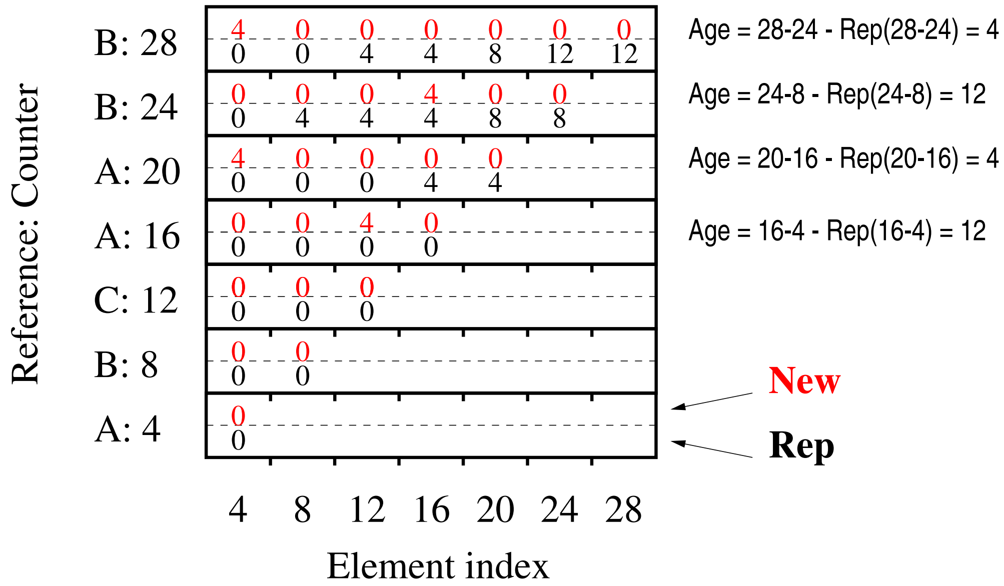


Figure 3.2: An example illustrating how the repeated bytes vectors Rep and New are updated to find the access ages of the sequence of word accesses (A, B, C, A, A, B, B). Each row shows the contents of vector elements $4, 8, 12, \ldots$**.**

Although, in this basic algorithm, vector **New** seems redundant because it carries only one value between two updates, it is useful in the optimized algorithm described below. Figure 3.2 shows how the two vectors are updated to find the access ages of the sequence of word accesses (A, B, C, A, A, B, B).

While shifting by incrementing a pointer can be inexpensive, accumulating the new repetitions takes $O(i)$ operations per access. Consequently, the overhead of maintaining the repetition vectors is $O(N^2)$ complexity, where N is the trace length. We use two optimizations to reduce this overhead:

1. Coarse repetition vectors are used where a vector element represents a region of K reaches and the vectors are updated each time K additional bytes have been accessed. Thus, the complexity is reduced to $O(N^2/K^2)$ at the expense of up to a $\pm K$ error in a reported access age. With a reasonable choice of K, such an error is quite acceptable when using this characterization to judge the working set size relative to particular cache sizes that are larger than K.

2. To exploit temporal locality, CIAT updates the repetition vectors in a lazy fashion. Although, aging the repetition vectors is performed by incrementing a pointer after every K bytes of access, accumulation of **New** into **Rep** is only done when needed and only up to the needed element. For example, when analyzing an access i of reach $i-j$, all unaccumulated elements in the repetition vectors between elements 1 and $(i-j)/K$ are accumulated. This lazy algorithm eliminates most of the accumulations, e.g. for Radix, the lazy algorithm does only

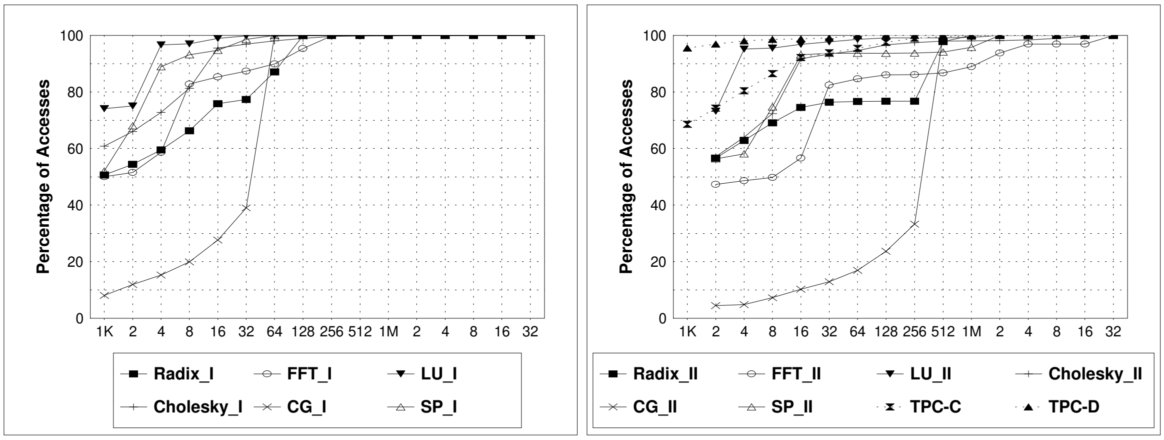


Figure 3.3: The cumulative distribution function of the access age. A point (x, y) indicates that y% of the accesses have access age of x bytes or less.

5% of the accumulations done when only the first optimization is used.

With these two optimizations, the time for characterizing the working sets is about 30% of CIAT's total analysis time. When it is sufficient to characterize the working set with respect to a limited number of cache sizes M, the stack simulation algorithm described in [KHW91], which has $O(N \times M)$ time complexity, can be used instead of our algorithm. Note that the stack block size should be selected to match the size of the smallest accessed element so as to avoid over-estimating the size of the working set.

Figure 3.3 shows the cumulative distribution function of the access age using 32 processors, ignoring infinite ages with $K = 1$ KB. A point (x, y) indicates that y% of the accesses have access age $\leq x$ bytes.

When a curve has a distinguishable rise to a plateau, this is an indication that the respective benchmark has an important working set of size $\leq$ the x value at the beginning of the plateau. CG has one important working set of size $\leq$64 KB for problem size I, and $\leq$512 KB for problem size II. CG is expected to have frequent capacity misses when the cache size is smaller than its working set size. Unlike other scientific benchmarks, the working set size of LU and Cholesky, at 4 KB and 16 KB respectively, does not change from one problem size to another.

Some applications have multiple important working sets. FFT with problem size II has two important working sets; one at 32 KB and another at 4 MB. Figure 3.3 indicates that TPC-D has better temporal locality than TPC-C and TPC-C performance could be improved by increasing the cache size beyond 64 KB. Since the access age shown is only for one process, TPC-C's performance may also be improved by larger caches because the operating system interleaves multiple processes per processor.

3.4.3 Concurrency

Concurrency is often characterized by measuring the execution time for a number of machine sizes and calculating the speedup. Good speedup indicates that the application has high concurrency. CIAT characterizes concurrency by measuring the time, in instructions, that processors spend executing instructions or waiting at synchronization points. Figure 3.4 shows a 2-processor execution profile of an application running on a perfect system (with fixed memory access time and zero synchronization overhead). The concurrency is reflected in the busy time relative to the total time of both processors.

Figure 3.4 demonstrates three general factors that adversely affect concurrency: *serial fraction*, *load imbalance*, and *resource contention*. At the application start (`T0`), Processor 0 is busy in a serial phase and Processor 1 is idle. At `T1`, the application enters a parallel phase by spawning an execution thread on Processor 1. Processor 1 joins at the end of the parallel phase (`T9`) where Processor 0 starts a final serial phase. The parallel phase has some load imbalance which is visible as Processor 0's wait on the synchronization barrier at `T2` and its join wait at `T8`. The first wait time is due to Processor 1 having more busy cycles (work) than Processor 0 and the second wait time is due to Processor 0 reaching the join point earlier than Processor 1. Processor 1 fails to acquire a lock that protects a shared resource at `T5`, so it waits until Processor 0 releases this lock at `T6` before it enters the critical region between `T6` and `T7` and accesses the shared resource.

Based on this model, the speedup of an application can be found by

$$\text{Speedup} = \frac{\text{Busy}(1)}{\{\text{Busy}(p) + \text{Idle}(p) + \text{Imbalance}(p) + \text{Contention}(p)\}/p}$$

where p is the number of processors, Busy(1) is the busy time for the basic work when using

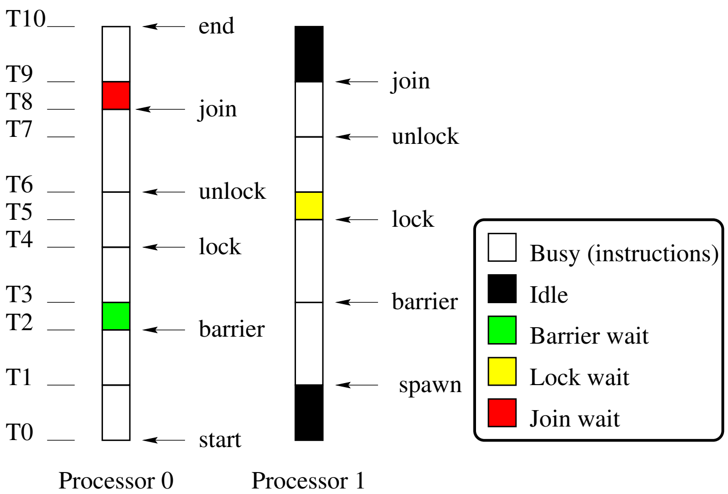


Figure 3.4: Execution profile of a parallel application running on a perfect system using 2 processors.

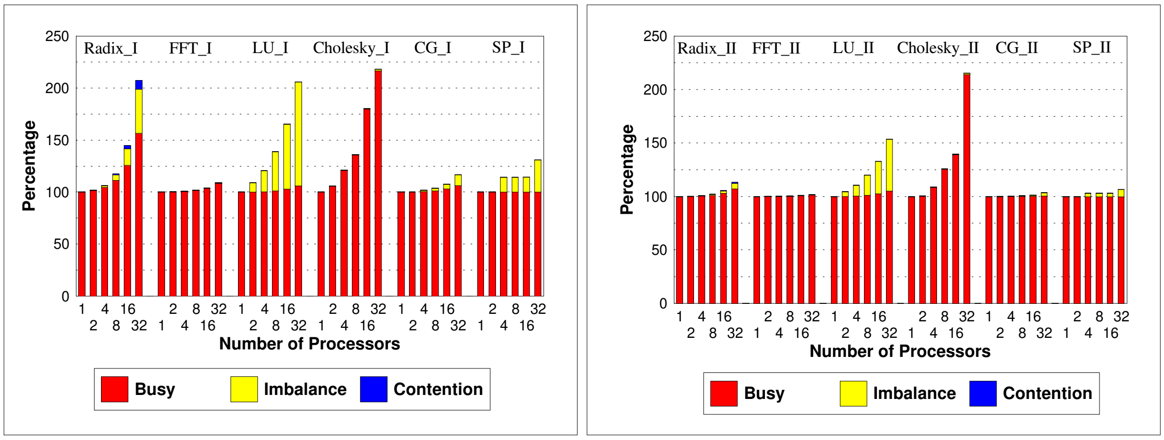


Figure 3.5: Concurrency and load balance in the parallel phase.

one processor, Busy(p) is the total busy time summed over the p processors (Busy(p) − Busy(1) is the parallel overhead busy work including redundant and added computations), Idle(p) is the total idle time during serial phases, Imbalance(p) is the total wait time on barriers and joins, and Contention(p) is the total wait time on locks. Amdahl's serial fraction [Amd67], ignoring the three parallel overheads, is

$$\text{Serial Fraction} = \frac{\text{Idle}(p)/(p-1)}{\text{Busy}(1)}$$

Perfect speedup is only possible when the serial fraction, parallel overhead busy work, imbalance, and contention are zero.

For the scientific benchmarks, Figure 3.5 shows the total processor time spent executing instructions, waiting on barriers and thread joins due to load imbalance, and waiting on locks due to resource contention (normalized to the one-processor time). Idle time is zero because this data is based on the parallel phase only.

Perfect speedup within the parallel phase occurs when the total busy and wait time does not increase as the processors increase. LU and Cholesky are thus expected to have worse speedup than the other four benchmarks, since LU's load imbalance and Cholesky's busy time increase as the number of processors increases. For problem size I, Radix also has a bad speedup due to increases in busy time, load imbalance, and contention as the number of processors increases.

Although Cholesky has the most lock attempts, unlike Radix, it shows negligible contention time because it uses flag-based synchronization (a processor first checks the status of the flag attached to the lock; if the flag indicates that the lock is free, the lock is acquired; otherwise, the processor busy-waits, which increases the busy time as the processors increase). In Cholesky, a processor attempting to acquire a lock usually finds it free. However, CIAT does not model the overhead in acquiring and releasing locks. In a machine with high synchronization overheads, the contention time can become more significant than what is reported by CIAT.

3.4.4 Communication Patterns

In configuration dependent analysis, communication is characterized from the traffic that a processor generates to access data that is not allocated in its local memory [SRG94]. This traffic includes traffic due to inherent coherence communication, cold-start misses, finite cache capacity, limited cache associativity, and false sharing. Inherent communication is the communication that must occur in order to get the current data of a location that is accessed by multiple processors, assuming unlimited replication is allowed and that a memory location's status is not affected by accesses to other locations.

CIAT characterizes the inherent communication by tracking, for each memory location, the type and the processor ID of the last access. When there are consecutive load accesses by multiple readers, their IDs are stored in a sharing vector. CIAT then captures the inherent communication for a shared location from changes in the accessing processor's ID. CIAT classifies communication accesses into four main patterns and 8 variants to provide a thorough characterization. CIAT reports the volumes of the following access patterns, and the sharing and invalidation degrees:

1. *Read-after-write* (RAW) access occurs when one or more processors load from a memory location that was stored into by a processor. This pattern does not include the case where only one processor stores into and loads from a memory location. Moreover, when a processor performs multiple loads from the same memory location, only its first load is counted a RAW access. RAW is a common communication pattern; it is the second part of a producer-consumer(s) situation where one processor produces data and one or more processors consume it. RAW is reported in two variants: (i) RAW accesses where the reader is different than the writer, and (ii) RAW accesses where the reader is the same as the writer and there are other readers. The second variant generates coherence traffic with caches that invalidate the dirty line on supply.

2. *Sharing degree* for RAW. This is a vector **S**, where $\mathbf{S}[k]$ is the number of times that k processors loaded from a memory location after the store into this location.

3. *Write-after-read* (WAR) access occurs when a processor stores into a memory location that one or more processors have loaded. This pattern does not include the case where only one processor loads from and stores into a memory location. WAR is also a common pattern; it occurs when a processor updates a location that was loaded by other processors. WAR is reported in four variants according to the identity of the writer: (i) a new writer that is not one of the readers, (ii) same writer as the previous writer which is not one of the readers, (iii) a new writer that is one of the readers, and (iv) same writer as the previous writer which is one of the readers. A WAR access generates a miss when the accessed location is not cached.

Additionally, with an update cache coherence protocol, a WAR access generates coherence traffic to update the copies in the readers' caches; while with an invalidation protocol, it generates invalidation traffic.

4. *Invalidation degree* for WAR. This is a vector **I**, where $\mathbf{I}[k]$ is the number of times that a memory location was stored into after previously being loaded by k processors.

5. *Write-after-write* (WAW) access occurs when a processor stores into a memory location that was stored into by another processor. This pattern occurs when multiple processors store without intervening loads, or when processors take turns accessing a memory location where in each turn a processor stores and loads, and its first access is a store. WAW is reported in two variants: (i) the previous processor's last access was a load, and (ii) the previous processor's last access was a store.

6. *Read-after-read* (RAR) access occurs when a processor loads from a memory location that was loaded by another processor and the first visible access to this location is a load. This is an uncommon pattern; it occurs in bad programs that read uninitialized data. Nevertheless, CIAT sometimes encounters this pattern when the data is initialized in untraced routines. For simplicity, these accesses could be added to the RAW accesses.

When using a particular coherence protocol in configuration dependent analysis, not all the above access patterns generate coherence traffic. Consider, for example, a store-update cache coherence protocol. In iterative WAR and RAW, a RAW access is satisfied from the updated local cache. However, with a store-invalidate protocol, a RAW access generates coherence traffic to get the data from the producer's cache. Additionally, sequential locality hides some of the communication events since each miss operates on a cache line that often contains multiple shared elements.

Configuration dependent analysis, when used to characterize inherent communication, can miss some communication events. For example, with finite write-back cache, a RAW access of a replaced line does not generate coherence traffic since the miss is satisfied from memory. However, this access may generate coherence traffic with a larger cache in which the line is not replaced. Another example is a RAW access performed by the writer. When the store is followed by a load from a different processor, the dirty cache usually supplies the data and keeps a shared copy, thus a RAW access by the writer is satisfied from the local cache. However, the RAW does generate coherence traffic if the cache protocol invalidates on supply.

Figure 3.6 shows the distribution of the four classes of communication accesses. Most of the communication in these benchmarks is RAW and WAR. Only Radix has a significant number of WAW accesses due to permuting the sorted keys between two arrays. TPC-C also has some RAR and WAW accesses which may actually become RAW and WAR accesses if the trace is longer

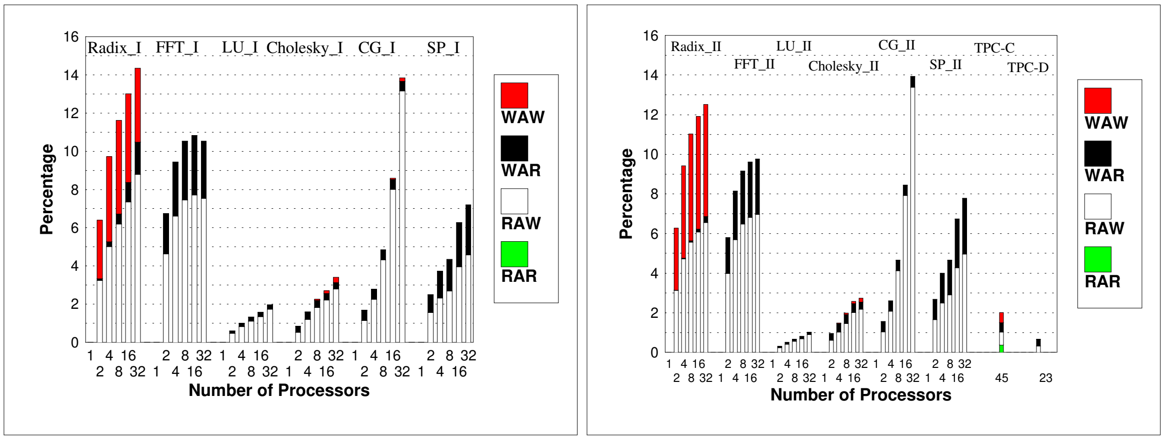


Figure 3.6: Percentage of the four classes of communication accesses among all data accesses, as a function of the number of processors for the two problem sizes.

and includes the operating system activity. However, generally the TPC benchmarks have less communication and are expected to have a lower coherence miss ratio, especially when we consider that in practice multiple processes run on one processor.

The communication percentage generally increases as the number of processors increases due to (i) the increase in RAW accesses of widely shared data, and (ii) the increase in the boundary to the body of data when partitioned among more processors. Often, most of the communication occurs when accessing the boundary elements. For FFT with problem size I, the communication percentage drops when the number of processors increases from 16 to 32 because the increase in total accesses is larger than the increase in communication events.

Figure 3.7 shows the distribution of the 32 possible sharing degrees for RAW accesses when using 32 processors ($(\mathbf{S}[k] \times 100/\sum_{i=1}^{32} \mathbf{S}[i]); k = 1, \ldots, 32$). TPC-C has 45 possible sharing degrees and TPC-D has 23, only the first 32 are shown due to graphing limitations. However, TPC-C has negligible sharing with degrees higher than 32. Radix, FFT, SP, TPC-C, and TPC-D have small sharing degree, LU and Cholesky have medium sharing degree, and CG has large sharing degree which explains its fast increase in communication percentage as the number of processors increases.

Figure 3.8 shows the percentage of the 32 possible invalidation degrees of the WAR access when using 32 processors ($(\mathbf{I}[k] \times 100/\sum_{i=1}^{32} \mathbf{I}[i]); k = 1, \ldots, 32$). TPC-C has 45 possible invalidation degrees and TPC-D has only 23. While Radix, FFT, CG, SP, TPC-C, and TPC-D have invalidation degree similar to their sharing degree, LU and Cholesky's invalidation degree drops to 2 and 1, respectively. CG's large invalidation degree implies that for each WAR access, a cache coherence protocol will generate many update or invalidate signals. The TPC benchmarks' singular sharing and invalidation degrees indicates that most of the communication occurs in a producer-consumer

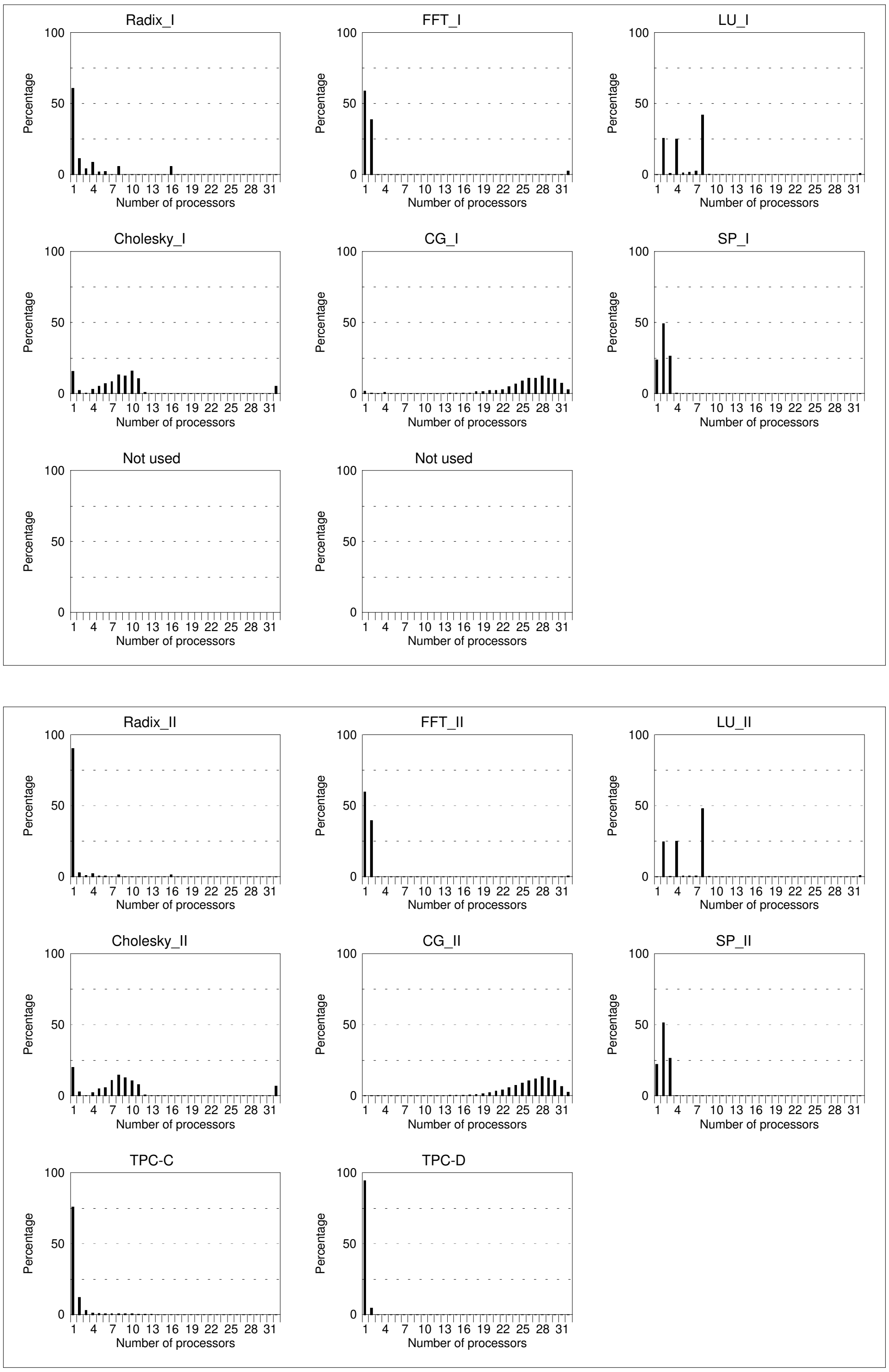


Figure 3.7: RAW sharing degree for 32 processors.

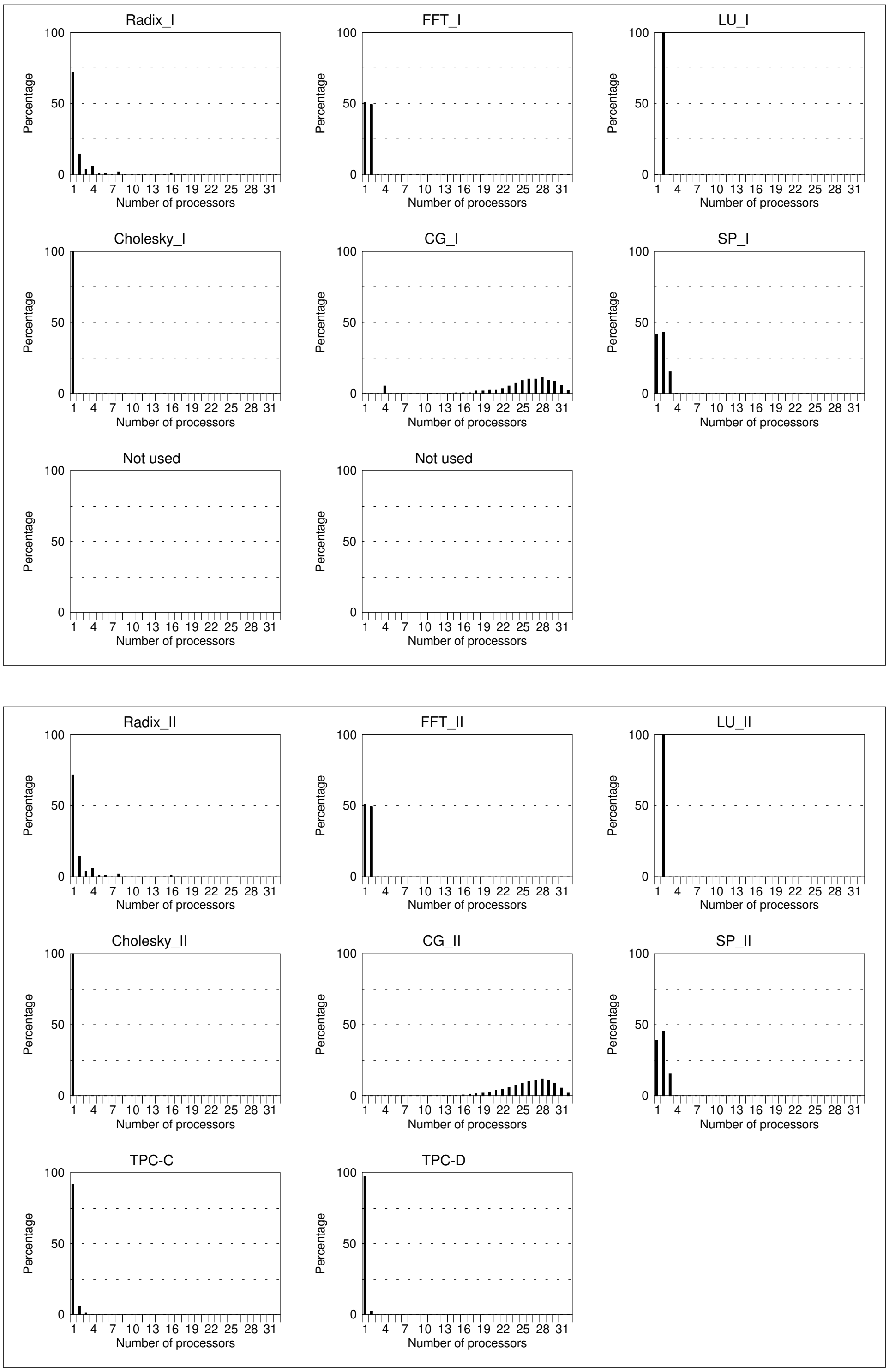


Figure 3.8: WAR invalidation degree for 32 processors.

pattern. However, in TPC-C the consumer generally updates the communicated value, while in TPC-D only one producer generally updates each location.

The sharing degree is identical to the invalidation degree whenever a data location consumed by one or more RAW accesses is updated by a WAR access. However, some applications have sharing degree that is different from the invalidation degree because some shared data is initialized once and never updated.

3.4.5 Communication Variation Over Time

TDAT is used to analyze CIAT's communication event trace that has one record for every communication event. Each record specifies the event's type and time (in instructions). The time distribution analysis is summarized as follows:

1. CIAT is used to analyze the benchmark and to generate the communication event trace.
2. The execution period is divided into 1000-instruction intervals.
3. The number of communication events in each interval is counted.
4. The communication rate in each interval is calculated as the number of communication events divided by the product of the interval size and the number of processors.
5. The average, minimum, and maximum communication rates over all intervals are calculated.
6. The communication rate density function is calculated, not including rate=0. The rate zero is excluded to minimize the effect of the serial initialization phase which does not have any communication.
7. The density function is integrated to find the distribution function.

Figure 3.9 shows the number of communication events over time for 32 processors and size I. The graphs do not show the initial execution periods that have no communication. LU, Cholesky, and CG have a high burst of communication at the parallel phase start when Processors 1 through 31 start to access the shared data initialized by Processor 0. Radix has two phases of communication to build the global histogram and to permute the keys between the two arrays. FFT's communication occurs in three phases when matrix transposition is performed. LU's communication is relatively less intense than the other benchmarks and is periodic with a decreasing cycle. Cholesky's communication is not uniform. CG and SP have periodic communication with fixed cycle. CG has a simple periodic behavior with a cycle of about 20,000 instructions; SP has a more complex behavior with a cycle of about 200,000.

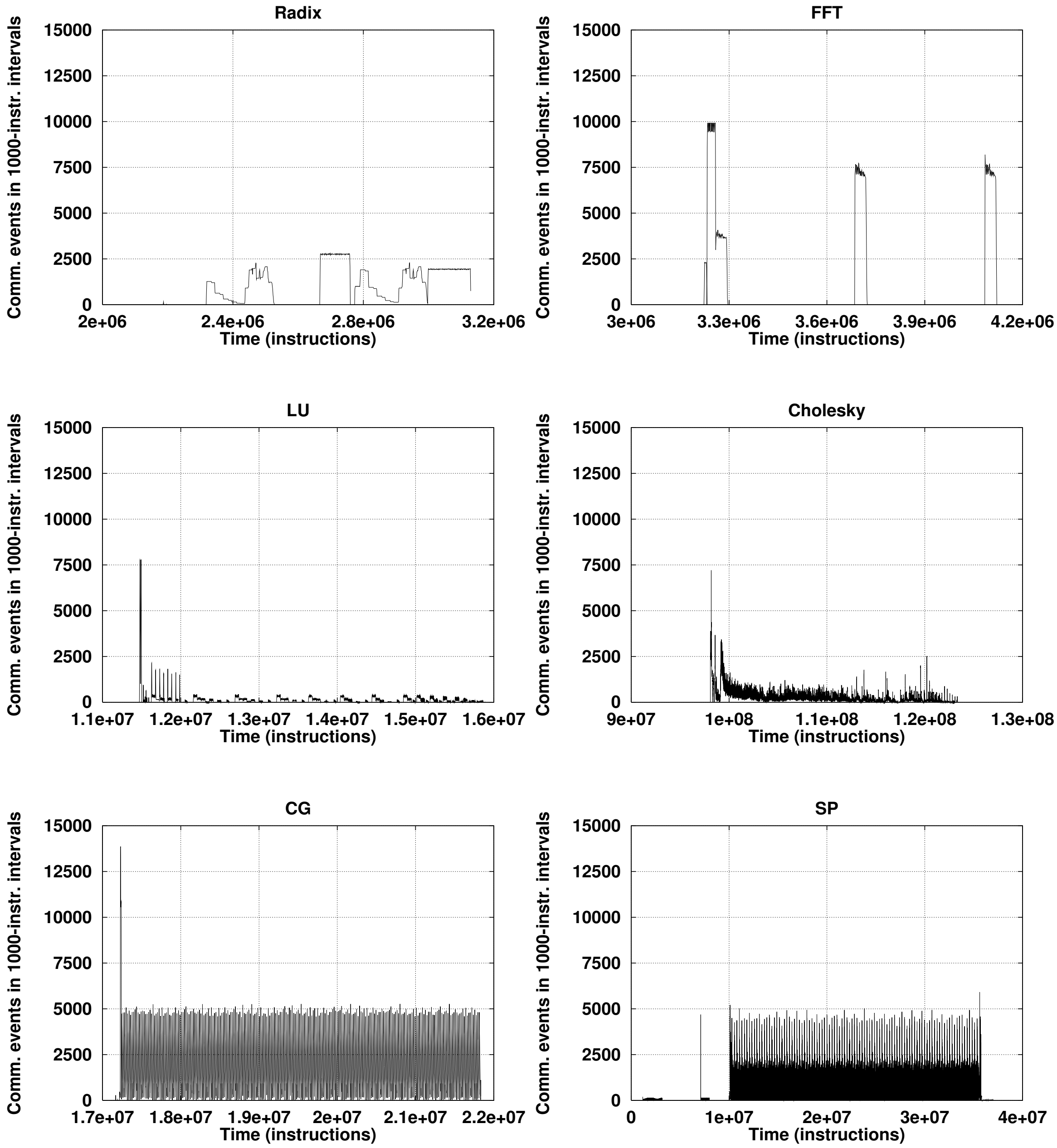


Figure 3.9: Number of communication events over time (32 processors, problem size I).

Figure 3.10 shows the communication rate distribution function with 32 processors and problem size I. While all Radix's communication occurs with a communication rate < 0.1 events/instruction, only 0.09 of FFT's communication occurs at a rate < 0.1 events/instruction. CG also has some high communication rates, suggesting that FFT and CG may suffer most from contention in systems with limited communication bandwidth.

3.4.6 Communication Slack

CIAT measures the communication slack as the time in instructions between generating a new value and referencing it by a RAW or a WAW access. Figure 3.11 is based on the communication slack histogram reported by CIAT for the scientific benchmarks when using 32 processors. The figure shows the percentage of the communication events binned in eight slack ranges, e.g. for problem size II, more than 96% of SP's communication has slack in the range of millions of instructions. For all the benchmarks, most of the communication has a slack of thousands of instructions or more. However, CG's more frequent use of barriers is reflected in its smallest slack. Each of the six benchmarks have large enough slack to make prefetching rewarding.

3.4.7 Communication Locality

CIAT characterizes the communication locality by reporting the number of communication events for each processor pair. CIAT reports this information in the matrix COMM_MAT. The element COMM_MAT[i][j] is the number of communication events from Processor i to Processor j which is incremented by one in the following cases:

1. Each of Processor j's RAW accesses to a location stored into by Processor i.
2. Each of Processor j's WAW accesses to a location previously stored into by Processor i.
3. Each of Processor j invalidation that is caused by a WAR access of Processor i.

Figures 3.12 and 3.13 show the reported communication matrix for the scientific benchmarks using 32 processors for problem size I and II, respectively. Figure 3.14 shows the communication matrix for the TPC benchmarks. The eight benchmarks have clearly different communication localities.

In addition to Radix's uniform communication component, processors have additional communication with their neighbors. This additional communication depends on the processor ID. A processor with an even ID, i, communicates to Processor $i + 1$; however, a processor with an odd ID communicates to three or more processors. The middle processor, Processor 15, communicates to 20 processors.

In FFT, Processor 0 communicates a constant number of values to every other processor, otherwise the processors communicate in a uniform pattern. In LU, the communication is mainly

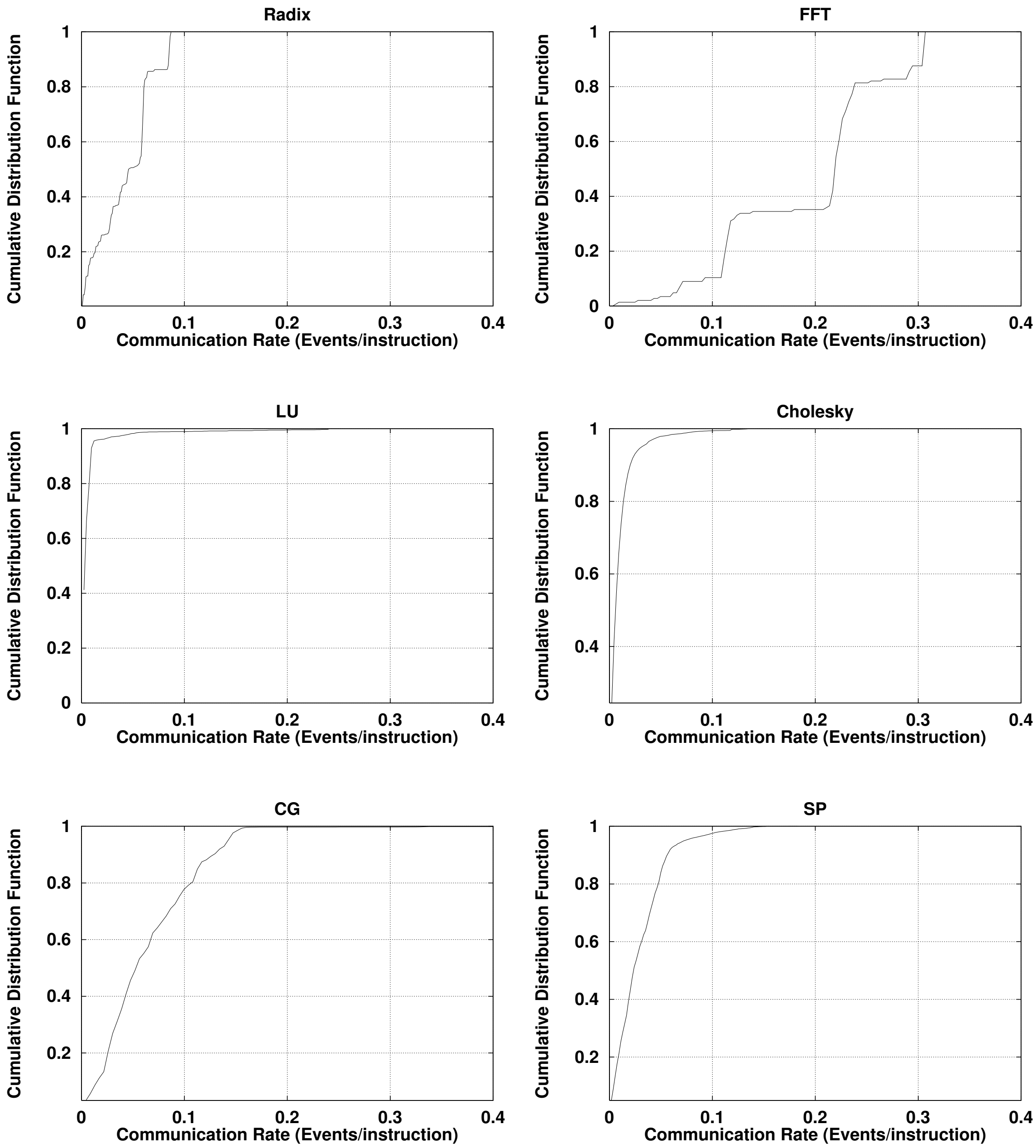


Figure 3.10: Communication rate per processor (32 processors, problem size I).

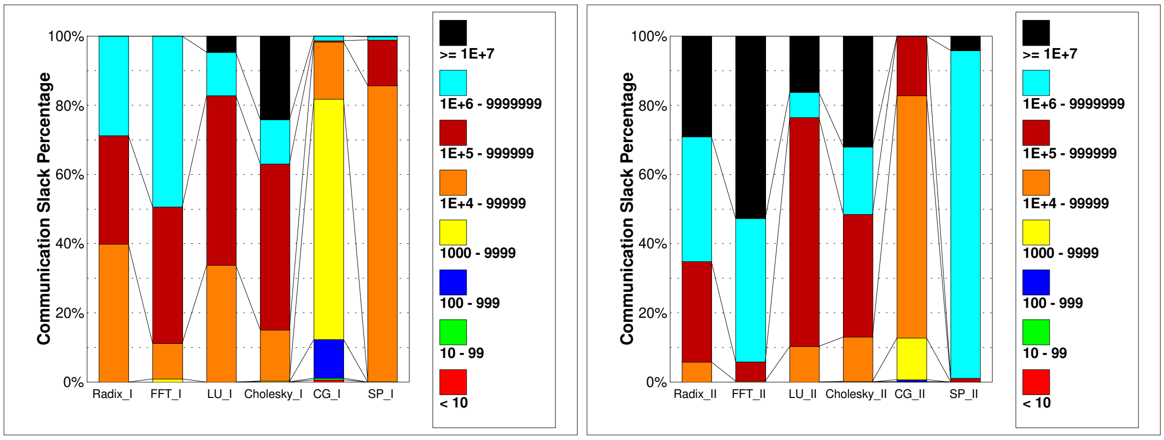


Figure 3.11: Communication slack distribution for 32 processors.

clustered within groups of 8, with some far communication: Processor i communicates to each processor j where $j = i \pm 8k$ for some integer k. Moreover, Processor 0 communicates to and from all other processors.

In Cholesky, Processor 0 communicates to all other processors, otherwise the processors communicate in a non-uniform pattern. While CG's communication is relatively uniform, SP's communication is clustered: Processor i communicates from each processor j where $j = i \pm 4k$ for some integer k, and from the mth group of 8 processors, where $m = i (\bmod 4)$.

In TPC-C, there is some communication among the first 16 processes (on Processors 0 through 15). Apparently, each is responsible for one user. Additionally, each of these processes produces more than 100,000 elements for the process on Processor 42 which in turn produces some data for the first 16 processes and more than 150,000 elements for the process on Processor 41.

In TPC-D, most of the communication is from the first 8 processes, which do disk reads and preprocessing, to the second 8 processes. This indicates that parallelism is exploited functionally among two 8-process groups, and spatially by partitioning the data into 8 parts.

3.4.8 Sharing Behavior

The data presented in this section is based on analyzing the code and data accesses of the whole execution, including the serial phase. Figure 3.15 shows the size of referenced memory locations classified in three classes: code locations, private data locations, and shared data locations. As the size of the code locations is much smaller than the size of the data locations, the code size is not visible in the chart. Generally, more locations become shared as the number of processors increases; all the scientific benchmarks show this trend. For problem size II, using 32 processors, more than 93% of the data locations of each of Radix, FFT, LU, and SP are shared. Furthermore, each additional thread may require some new private memory locations, causing an increase in the size of private memory and hence the total data memory. This trend is particularly visible in

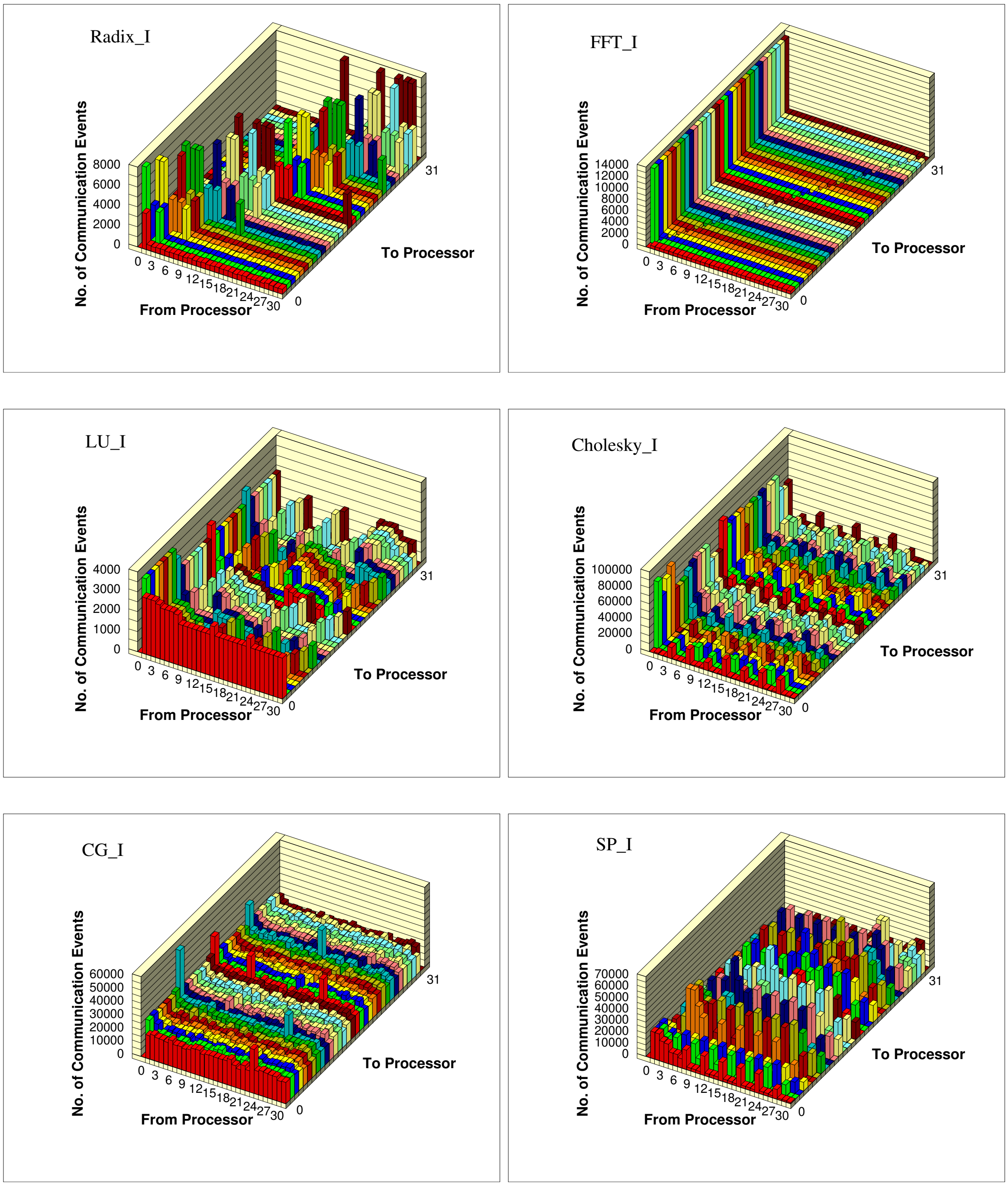


Figure 3.12: Number of communication events per processor pair (32 processors, problem size I).

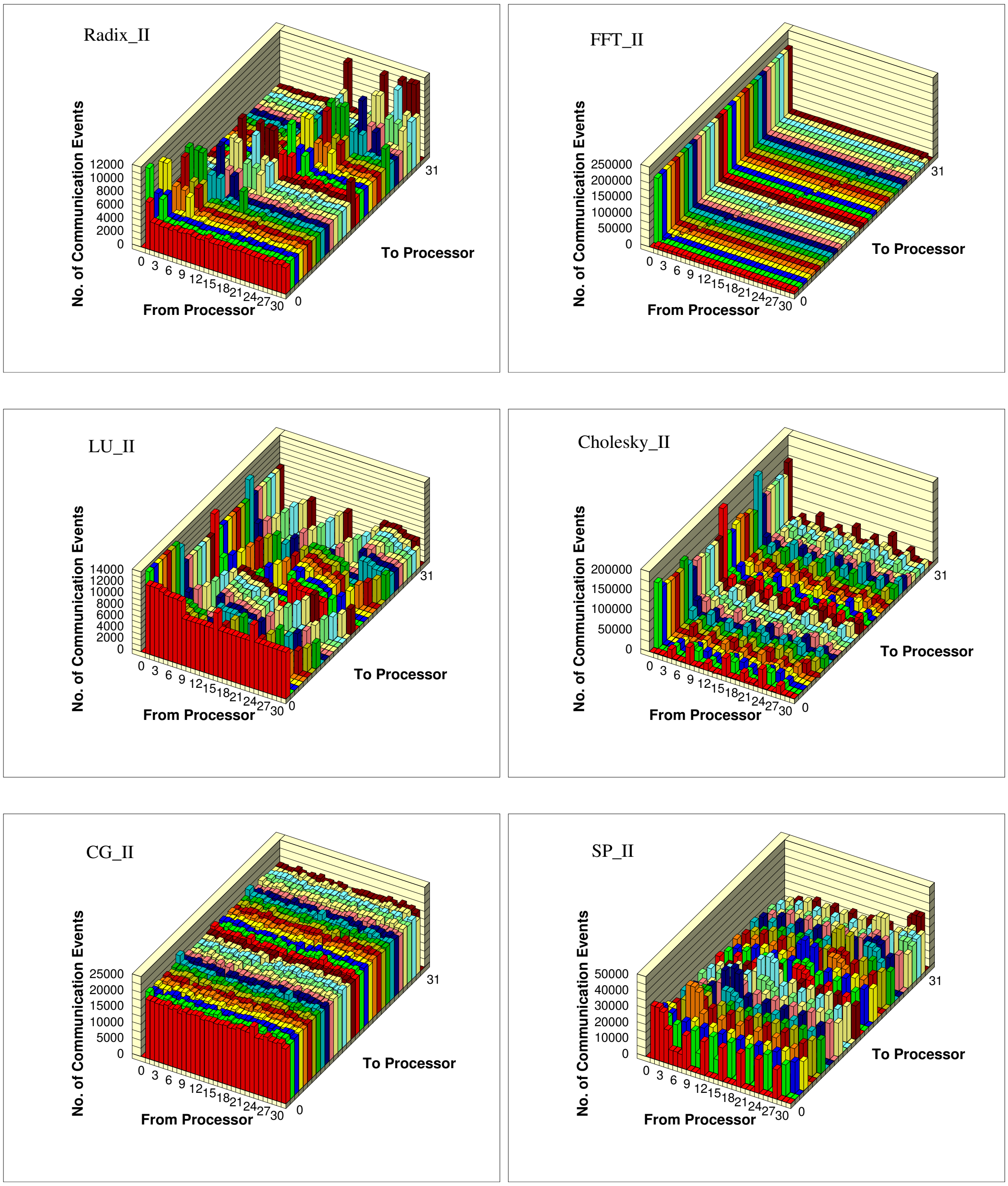


Figure 3.13: Number of communication events per processor pair (32 processors, problem size II).

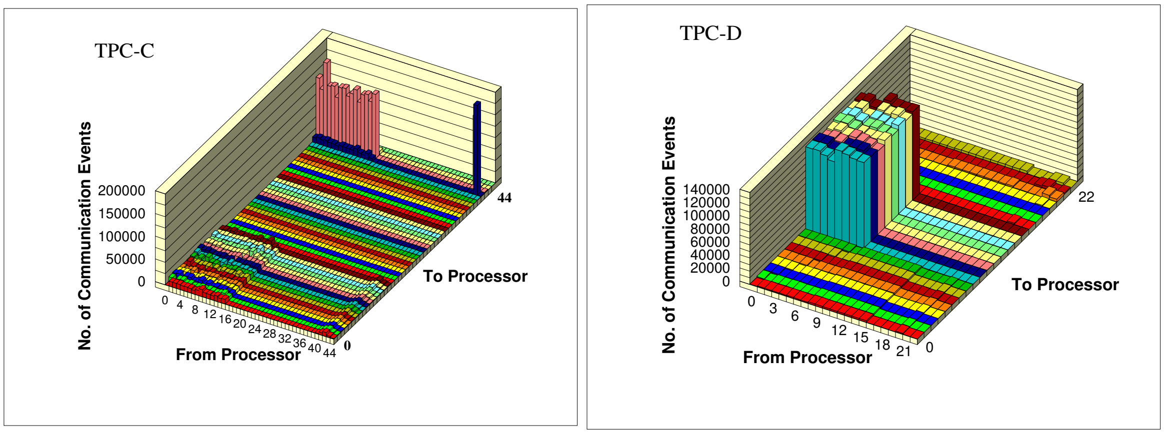


Figure 3.14: Number of communication events per processor pair (TPC-C: 45 processors, TPC-D: 23 processors).

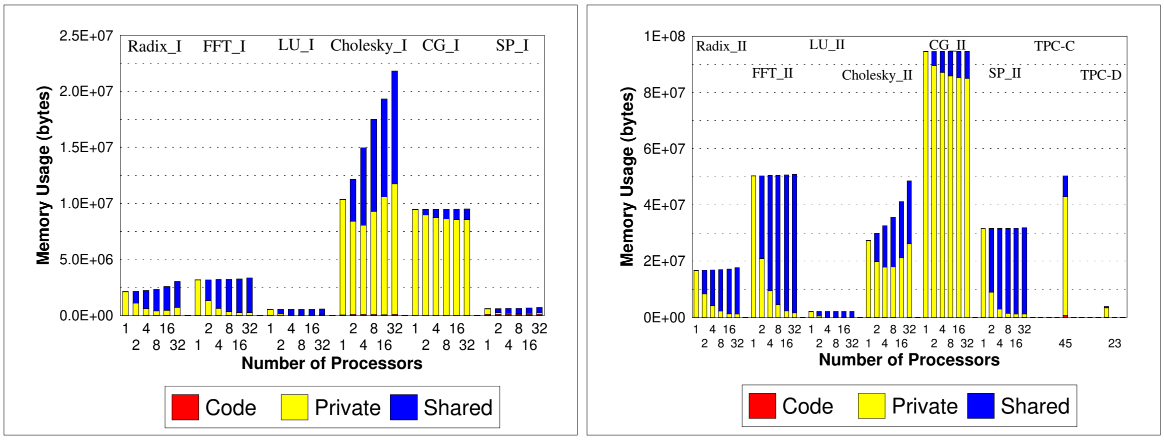


Figure 3.15: The size of code, private data, and shared data locations.

Cholesky, and somewhat in Radix. TPC-C's shared percentage is 15% and TPC-D's is 12%.

Figure 3.16 shows the number of private and shared data accesses normalized to the number of data accesses using one processor. In Cholesky, CG, TPC-C, and TPC-D, the shared locations are more intensely accessed than the private locations. For example, using 32 processors to solve problem size II, 10% of CG's data locations are shared and these are referenced by 25% of all the data accesses. However, in Radix, FFT, LU, and SP, the shared locations are less intensely accessed. For example, using 32 processors to solve Radix's problem size II, only 17% of all accesses are to shared locations. The scientific benchmarks show some increase in the total number of accesses due to the increase in the private accesses as the number of processors increases. However, the large increase in the total number of accesses in Cholesky is due to the increase in both private and shared accesses.

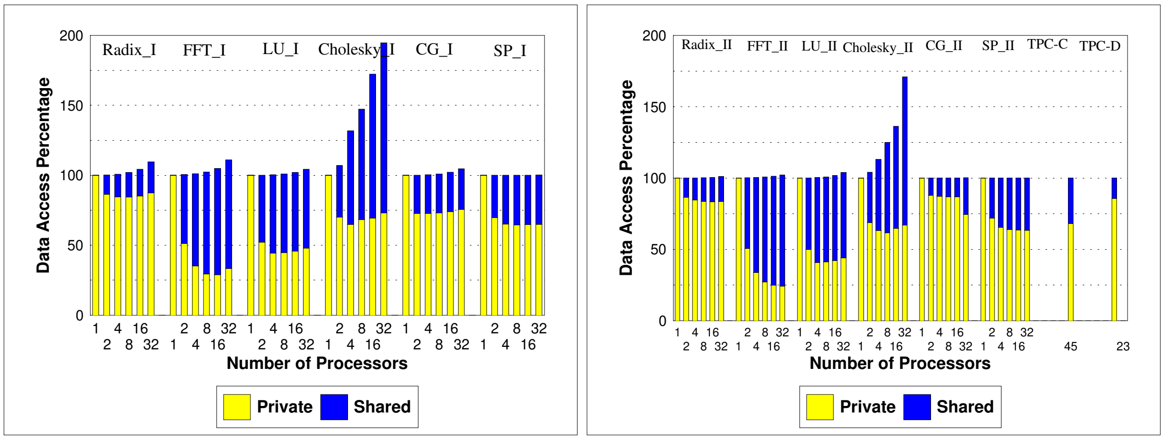


Figure 3.16: Number of private and shared data accesses normalized to the number of data accesses when using one processor.

3.5 Chapter Conclusions

In this chapter, we have demonstrated that configuration independent analysis is a viable approach to characterizing shared-memory applications. CIAT, our configuration-independent tool, efficiently characterizes several important aspects of shared-memory applications. CIAT can be used to mechanically characterize a wide range of shared-memory applications.

CIAT characterization of concurrency is informative since it specifies the application's serial fraction, parallel overhead busy work, load balance, and resource contention. Using an algorithm based on finding the age of the memory accesses, CIAT characterizes the working sets of an application by doing only one experiment and is not confused by coherence misses. CIAT also characterizes the inherent communication which is not affected by capacity, conflict, or false-sharing misses. It reports the volume of the four types of communication patterns and their variants, and it characterizes the communication variation over time, as well as communication slack and locality.

We have demonstrated our analysis approach by analyzing eight benchmarks using two problem sizes and a varying number of processors. This study shows the power of this approach and the insights that can be gained from a configuration independent analysis of targeted benchmarks. The results are presented in a form that can readily be exploited by application and system designers.

CHAPTER 4

MICROBENCHMARKS FOR CALIBRATING EXISTING SYSTEMS

This chapter outlines our methodology for characterizing distributed shared memory performance using microbenchmarks. We describe what these microbenchmarks measure and how.

4.1 Introduction

In distributed shared memory systems, the distributed memory, caches, and coherence protocols make their performance a very complex function of the workload, thus slowing the development of efficient applications. Developing efficient applications and compilers for a DSM multiprocessor requires a good understanding of the performance of its hardware and parallel software environment. Performance models are also useful for conducting quantitative comparisons among different multiprocessors and selecting the best available application implementation techniques.

In this chapter, we present an experimental methodology to characterize the memory, scheduling, and synchronization performance of DSM systems. We extend the microbenchmark techniques used in previous studies [GGJ$^+$90, MSSAD93, SS95, McC95a, MS96] by adapting them to deal with the important attributes of a DSM system. Like previous studies, our benchmarks evaluate the cache, memory, and interconnection network performance. Moreover, special attention is given to characterizing the communication performance, the overheads in maintaining cache coherence, and the effect of contention for local and remote resources. This characterization is useful to programmers to estimate the components of the memory access time seen by their programs. Additionally, it is useful to architects to identify strengths and weaknesses in current systems, as demonstrated by the case study in the next chapter.

The next section outlines our methodology for characterizing DSM performance. Section 4.3 describes the memory kernels used in these microbenchmarks. Section 4.4 presents the microbenchmarks for local-memory characterization. Shared memory and communication performance are treated in Section 4.5. Sections 4.6 and 4.7 treat the scheduling and synchronization overheads, respectively. Section 4.8 offers some conclusions.

4.2 DSM Performance

Figure 4.1 illustrates some performance aspects of the execution of a shared-memory program. It shows two processors executing a two-segment program, where each segment is ended by a synchronization barrier. The execution time depends on the computation, scheduling overhead, load balance, and synchronization overhead. In this chapter, we focus on characterizing the memory access components of the computation time, and briefly consider scheduling and synchronization overheads.

In shared-memory programs, communication is implicit through load and store instructions to the shared memory during the computation time. We present experimental methods for measuring the memory access latency under varying conditions of access distance, sharing, and traffic. The distance is determined by where in the memory hierarchy, relative to the processor, the memory access is satisfied. For example, in the SPP1000 [Bre95], a memory access can be satisfied by a hit in the processor cache, local memory, another processor's cache in the same node, remote shared memory, or a remote processor's cache.

Due to the various overhead factors in the cache coherence protocol, the memory access latency is affected by the way that the referenced data is shared and the sequence of accesses to it. We present experiments for characterizing the performance of basic shared data access patterns for two processors, and then extend these experiments to multiple processors. We then present experiments to characterize the effects on performance when multiple processors compete for utilization of the distributed memory and interconnection network.

The scheduling overhead is time spent in the parallel environment while allocating and freeing processors for parallel tasks. The example in Figure 4.1 shows static scheduling overhead at the start and at the end of task execution. Scheduling overhead may also exist within the execution when dynamic allocation is supported.

The synchronization overhead is the time spent performing synchronization operations, e.g.,

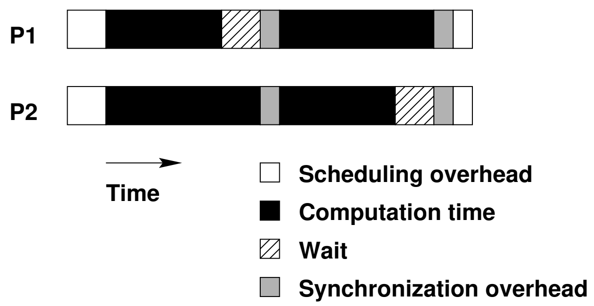


Figure 4.1: Illustration of a shared-memory program execution with two processors.

barrier synchronization, or acquiring a lock for a critical section [Hwa93]. The synchronization overhead does not include the wait time due to load imbalance, which can easily be treated separately [Tom95, Boy95].

We use simple Fortran programs to gather timing data for each of these three aspects of DSM performance. Each program calls and times a kernel 100 times and, after subtracting timer overheads, reports the minimum, average, and standard deviation of its *call times*. The first call time is ignored since it is significantly affected by page faults and other cold start effects. To minimize extraneous effects, the experiments should be carried out during an exclusive reservation period where the one active user process is running one of these timing programs.

4.3 Memory Kernel Design

Memory kernels are the building blocks of the memory and communication microbenchmarks. Each kernel accesses the l-byte elements of an array of size w bytes with stride s bytes. Thus, one call time consists of the w/s accesses in a kernel call.

The memory kernels should be carefully designed to measure latency and bandwidth. A kernel that measures latency exposes the access latency and does not overlap or hide the latencies of individual memory accesses, whereas, a kernel that measures bandwidth maximizes access overlap to achieve the highest possible bandwidth. We use four kernels to measure latency and bandwidth of load and store accesses ($l = 4$ for latency kernels, and $l = 8$ for bandwidth kernels):

Load latency: To measure the load latency, the load kernel should force the processor to have only one outstanding access, which is achievable by the `load-use` Fortran kernel that is shown with an example in Figure 4.2:

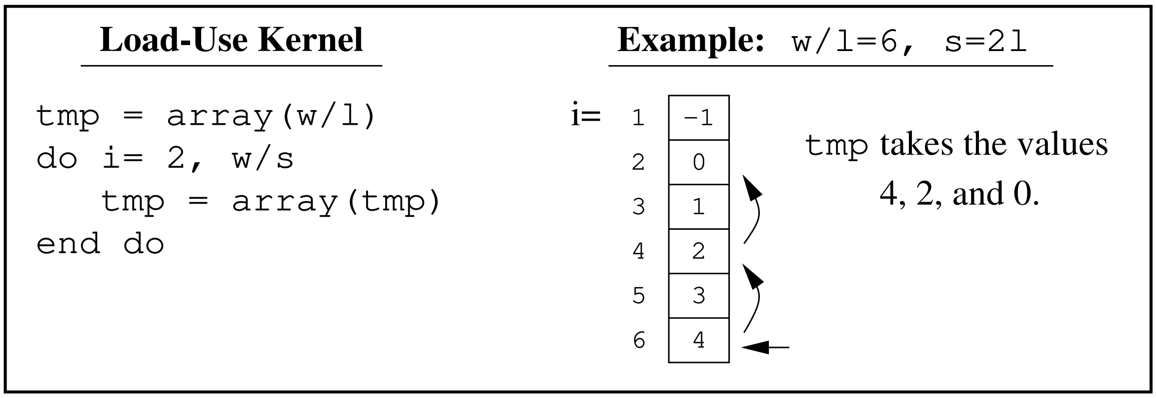


Figure 4.2: `Load-use` kernel.

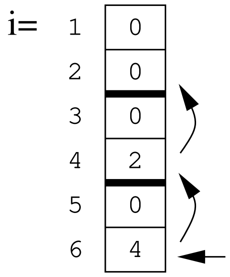
The array used in this kernel is initialized as `array(i) = i -` s/l, for each of its w/l elements. Consequently, this kernel scans the array backwards and has only one outstanding load because the address of each load is a function of the previous loaded value.

Store latency: The store instruction does not return data that can be used to establish a data dependency that serializes the store stream. However, for strides $\geq 2l$ bytes, the `store-load-use` kernel in Figure 4.3 is capable of measuring the store latency. This kernel measures the store miss latency in write-allocate caches as the latency from issuing the store to completing the allocation in the cache of the missed line. The store in this kernel may hit or miss, but the load always hits. The load simply reads the address for the next store, thereby establishing the data dependency needed to serialize the store accesses.

Store-Load-Use Kernel

```
array(w/l-1) = 0
tmp = array(w/l)
do i= 2, w/s
   array(tmp-1) = 0
   tmp = array(tmp)
end do
```

Example: `w/l=6, s=2l`



`tmp` takes the values 4, 2, and 0. Elements 5, 3, and 1 are cleared.

Figure 4.3: `Store-load-use` kernel.

Consider the array used in this kernel as being divided into consecutive groups of s/l elements each. The array is initialized such that the elements in each group have the same value and point to the last element in the previous group. This kernel also scans the array backwards, and the two elements accessed by one iteration's store and load instructions must be allocated in the same cache line. The store instruction in this kernel must precede the load (otherwise, in a multiprocessor system, the load may generate a load miss, and if the memory returns a shared line, a following store to that line would request exclusive ownership).

In modern superscalar processors, the overhead of the load and use is minimized since these processors usually perform the load concurrently with the store and bypass the load's result for quick calculation of the new address. On the PA8000 [Hun95], the `load-use` kernel and the `store-load-use` kernel each takes three cycles per iteration when all accesses hit in the cache, which is the load followed by use latency.

Load bandwidth: The following `load` kernel, which performs simple, forward accessing of the array, can be used to measure bandwidth, possibly with some modifications as described below.

```
do i= s/l, w/l, step s/l
   sum = sum + array(i)
end do
```

Usually, modern optimizing compilers can achieve peak bandwidth with this kernel. The compiler should unroll this loop sufficiently to minimize the effect of the backward branch at the loop's end, and schedule the loop in a way that masks its compute time. In some processors, the existence of the add instruction may reduce the number of outstanding misses that the processor can have. For such processors, the add instructions can be deleted from the assembly file of the compiled kernel.

Store bandwidth is measured by the `store` kernel, which is similarly designed by replacing the reduction, `sum = sum + array(i)`, in the `load` kernel by `array(i)=0`.

Processors that support prefetching complicate measuring the cache miss latencies. Some processors support software prefetching by providing "non-binding" load instructions that can be added by the programmer or the compiler to bring the data to the cache ahead of its use. Non-binding loads do not return data to a register. Software prefetching can be disabled by instructing the compiler not to insert prefetch instructions. Some processors support hardware prefetching where the cache controller, based on the miss history, speculates which uncached lines will be referenced in the near future and automatically generates requests to prefetch these lines. For example, the PA7200 processor [KCZ+94] prefetches the next sequential cache line on a cache miss. Since the above two latency kernels scan the array backwards, the next (forward) sequential line is always in the cache, thus, no sequential prefetching is done.

In addition, the PA7200 prefetches cache lines based on hints from the load/store-with-update instructions. These instructions are usually used in loops with constant strides through arrays. They perform loads or stores that update the base register that is used in calculating the current address. The modified base is typically used to find the address in the next iteration. When a load/store-with-update instruction generates a cache miss, the PA7200 uses the modified address to find the stride and direction for prefetching. The above latency kernels have indirect addressing and do not have load/store-with-update instructions, thus, they avoid this type of prefetching.

In these latency kernels, `array` is initialized in such a way as to cause the array be scanned linearly in the backward direction. However, it can be initialized differently to avoid hardware prefetching when other prefetching algorithms are used.

In this characterization, the *average access time* is calculated as the minimum call time divided by the accesses per call (w/s). The *transfer rate* is found by dividing the number of accessed bytes per call (lw/s) by the minimum call time. Usually the highest observed transfer rate occurs at $s = l$ bytes because all the transferred data are accessed. The *bandwidth* through an interconnecting link is calculated by dividing the number of bytes transferred through this link by the average call time.

The minimum call time is used in calculating the average access time and the transfer rate for a processor in order to minimize extraneous effects. The average call time is used in calculating the bandwidth through a link in order to account for concurrent link utilization when multiple processors are active.

4.4 Local Memory Performance

In the experiments used to characterize the processor data cache and local memory performance under private access conditions, a memory kernel is run on one processor. Each experiment is repeated for strides $l, 2l, 4l, ..., S$, and $2S$ bytes; where $2S$ is the first stride with the same time per access as the previous stride. Thus, one cache line contains S bytes. Figure 4.4 shows typical results of the average access time as a function of array size for a processor with one cache of size C bytes, A-way set-associative, and least-recently-used replacement policy. The figure shows three regions:

1. *Hit region* ($0 < w \leq C$ bytes): The array size is smaller than the cache size, and every access is a hit taking T_{H} sec.
2. *Transition region* ($C < w < C + C/A$ bytes): Some of the accesses are hits and others are misses, resulting in an average access time of $T_{\mathrm{T}}(s, w)$ sec. The width of this region is the cache size divided by the degree of the cache associativity (assuming LRU replacement).
3. *Miss region* ($w \geq C + C/A$ bytes): The array size is so large that every access to a new cache line is a miss taking an average access time of $T_{\mathrm{M}}(s)$ sec. Store miss time is usually larger than load miss time due to the write-back overhead of flushing the dirty replaced line from the cache.

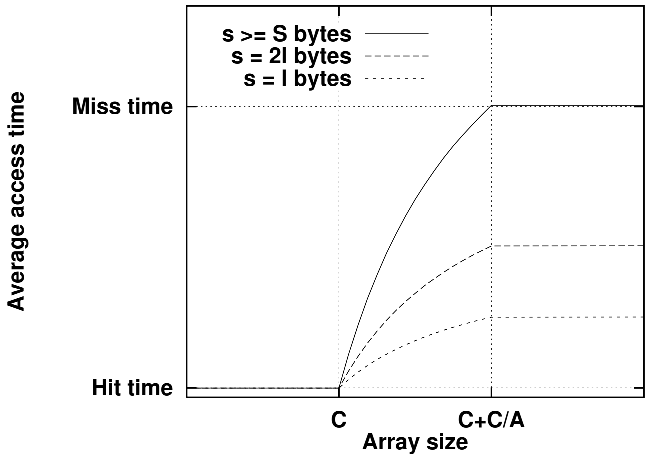


Figure 4.4: Typical average access time as a function of array size for various strides, $s = l, 2l,$ and S bytes.

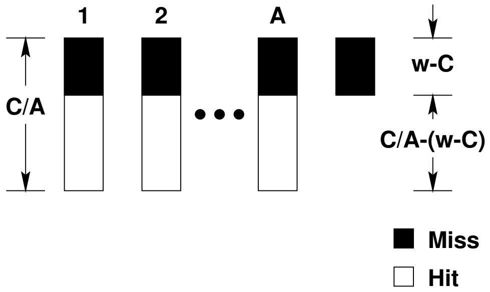


Figure 4.5: Cache misses in the transition region ($C < w < C + C/A$) for array size w, and an A-way associative cache of size C.

Using the latency kernels, T_{H} is the latency of an access that hits in the cache, and $T_{\mathrm{M}}(S)$ is the latency of an access that misses in the cache (plus the latency of finding the next address). Using the bandwidth kernels, the *cache* transfer rate is calculated as the number of accessed bytes (lw/s) divided by the minimum call time in the hit region. The *memory* transfer rate is found as the size of the accessed cache lines (w for $s \leq S$, and Sw/s for $s > S$) divided by minimum call time in the miss region.

The $T_{\mathrm{T}}(s, w)$ curve is a truncated hyperbola, namely

$$T_{\mathrm{T}}(s, w) = \frac{T_{\mathrm{H}} \times (\text{resident\_lines}) + T_{\mathrm{M}}(s) \times (\text{nonresident\_lines})}{\text{total\_lines}}$$

Figure 4.5 illustrates $T_{\mathrm{T}}(s, w)$ for a cache with size C and associativity A. For an array of size w, the segment of size $w - C$ shown to the right is the excess segment. When the array is accessed repetitively, assuming an LRU replacement strategy, resident_lines $= A(C/A - (w - C))$ and nonresident_lines $= (A + 1)(w - C)$, where the total_lines $= w$. Hence, $T_{\mathrm{T}}(s, w)$ is given by the following hyperbolic function of w:

$$T_{\mathrm{T}}(s, w) = T_{\mathrm{H}} \times \frac{C - A(w - C)}{w} + T_{\mathrm{M}}(s) \times \frac{(A + 1)(w - C)}{w}$$

Note that $T_{\mathrm{T}}(s, C) = T_{\mathrm{H}}$ and $T_{\mathrm{T}}(s, C + C/A) = T_{\mathrm{M}}(s)$.

4.5 Shared Memory Performance

In this section, we present the shared-memory performance evaluation methodology. Subsection 4.5.1 evaluates interconnect cache performance, 4.5.2 evaluates shared-memory performance when 2 processors interact in a producer-consumer access pattern, 4.5.3 evaluates the overhead of maintaining coherence when multiple processors are involved in shared-data access, and 4.5.4 evaluates overall performance when multiple active processors contend in accessing local and remote memory.

4.5.1 Interconnect Cache Performance

Some DSM systems use interconnect caches (ICs) to exploit locality of reference to remote shared-memory data (shared data with a home memory location in some other node). When remote shared-memory data is referenced by a processor, if it is not in the processor's data cache or the local node's IC, a cache line is retrieved over the interconnect and a copy is stored in the local IC. Hence, subsequent references to this line that miss in one of the data caches of this node can be satisfied locally from the IC (until this line is replaced in the IC or invalidated due to write by a remote node).

To evaluate the performance of the IC, we use an experiment similar to the one used for evaluating local-memory performance. We use a program that is run on two processors from distinct nodes. The first processor allocates a local shared array and initializes it. The second processor accesses the array repetitively using a load kernel. The experiment is repeated for varying array sizes to characterize the IC size and associativity, and for varying strides to characterize the IC line length.

4.5.2 Producer-Consumer Performance

In this subsection, we evaluate the shared-memory performance when two processors interact in a producer-consumer access pattern on shared data. For this purpose, we use a program that has the following pseudo-code:

```
shared A[w/l]
repeat {
  Proc 0 writes into A[] with stride s (a store kernel)
  wait_barrier()
  Proc 1 reads from A[] with stride s (a load kernel)
  wait_barrier()
}
```

This program, which runs on two processors, simulates the case when one processor *produces* data and another processor *consumes* it. For w/l-element array with stride s, Processor 0 does w/s write accesses and then Processor 1 does w/s read accesses in each iteration. w is selected such that the array fits in the processor data cache. The times spent in the two kernels are measured and the minimum times are divided by the number of accesses to get the average access times. The time of Processor 0 is called write-after-read (WAR) access time. The time of Processor 1 is called read-after-write (RAW) access time.

The synchronization barrier used in these microbenchmarks should be carefully selected. Consider that when Processor 0 is in the store kernel, Processor 1 is waiting on the synchronization barrier. Thus, a processor waiting on a barrier should be waiting on a cached flag so that it does not generate memory traffic that affects the accesses of the other active processor or processors.

If the program is modified so that Processor 1 writes into the array instead of reading, we get a third access pattern with write-after-write (WAW) access time. This is a less common access pattern, but may occur, for example, in a false-sharing situation where two processors write into two different variables that happen to be located in the same cache line.

For the fourth access pattern, read-after-read (RAR), the program would be modified so that both processors read. In this case, each processor acquires a local copy of the data, and we would get access times similar to the local-memory access times. Thus, the local-memory access times could simply be used instead (unless the accessed array is larger than the processor cache and has a remote home, which will be cached in the local IC—if used).

The WAR and RAW access times can be used to find the shared-memory point-to-point producer-consumer communication latency and transfer rate. The latency is the sum of WAR and RAW access times with $s = l$ bytes when using the latency kernels. The transfer rate is l bytes divided by the sum of WAR and RAW access times with $s = l$ when using the bandwidth kernels.

These times depend on the access stride, the distance between the two processors, and whether the data is cached in the other processor's cache. Since the array fits in the data cache, it is cached whenever a processor accesses it. To measure the not-cached case, we add some code to flush the cache after the kernel calls or use a large array ($w \geq C + C/A$).

In systems like the Convex SPP1000, the latency of a remote access depends on the number of nodes and does not depend on the distance of the remote node. Each access has a request path and a reply path, which collectively circle one interconnection ring, therefore, the latency is not affected by the location of the remote node. However, other interconnection topologies, like mesh and hypercube, have latencies that do depend on the remote node location. Consequently, the microbenchmarks that measure their communication performance need to report performance figures that do depend on the relative position of the processors.

To characterize such systems accurately, we need either the ability to control processor allocation or, at least, the ability to find the physical location of an allocated thread. When the system provides the ability to control processor allocation, we suggest characterizing such systems along two dimensions: First the distance dimension, by repeating the producer-consumer communication experiment for various distances. Second the function dimension, by performing the experiments described in this section with the best case allocation, in which the nodes used are as close as possible.

To achieve a similar two-dimensional characterization for systems that provide the ability to find the physical location of an allocated thread, we suggest carrying out experiments that request all the physical processors, but activate only a subset of these processors according to their location. This approach assumes that the system does not oversubscribe processors and does not migrate threads from one processor to another.

4.5.3 Coherence Overhead

In this subsection, we evaluate the shared-memory performance when two or more processors perform read and write accesses to shared data. The objective here is to evaluate the coherence overhead as a function of the number of processors involved in shared-memory access. For this purpose, we use a program that has the following pseudo-code:

```
shared A[w/l]
repeat {
  Proc 0 writes into A[] with stride s (store-load-use)
  wait_barrier()
  foreach proc ≠ Proc 0 {
    begin_critical_section
    read from A[] with stride s (load-use)
    end_critical_section
  }
  wait_barrier()
}
```

This program is run on p processors and simulates the case where one processor *produces* data and $p-1$ processors *consume* it. In this program, Processor 0 does w/s write accesses per iteration, and each of the other processors does w/s read accesses per iteration inside a critical section. Notice that no more than one processor is active in the critical section at any time. The time spent in doing these accesses is measured for each processor and the minimum times are divided by the number of accesses to get the average access time for each processor. The time of Processor 0 is the *invalidation time* which is a function of p. The times for the other processors often depend on the order in which they enter the critical section, e.g., the Convex Exemplar evaluated in Chapter 5.

4.5.4 Concurrent Traffic Effects

In each experiment described in the previous sections, only one processor at a time is actively running a load or store kernel. Hence, the reported results represent a best case where processor accesses are not hindered by any resource contention with other accesses. We have modified some of the previous experiments to have multiple active processors. These modifications allow characterizing the aggregate memory bandwidth through the system interconnects, and characterizing single processor memory performance degradation due to concurrent traffic.

To characterize the aggregate local-memory bandwidth, we use an experiment based on the `load` kernel. In this experiment, p processors from one node use the `load` kernel to access a $2pC$-byte array, simultaneously, where C is the size of the processor cache. Each processor accesses a disjoint $2C$-byte segment with stride s. The segment size is selected to evaluate the local-memory performance in the miss region. The aggregate bandwidth is calculated as

$$p \times \frac{\text{cache line size}}{\text{average line access time over all processors.}}$$

We characterize the aggregate remote-memory bandwidth by the performance of the remote RAW access. In this experiment, p processors from one node use the `store` kernel to update a $2pC$-byte array, simultaneously. Then p processors from another node use the `load` kernel to access the array, simultaneously. The `load` kernel time is used to find the aggregate bandwidth for RAW through the system interconnect.

4.6 Scheduling Overhead

The scheduling time is the time needed by the parallel environment to start and finish parallel tasks on p processors. This time may include the overhead of allocating processors for the parallel task, distributing the task's executable code, starting the task on the processors, and freeing the allocated processors at task completion. In this section, we present a characterization of two aspects of scheduling overheads: static scheduling and parallel-loop scheduling.

Static scheduling overhead (SSO) occurs when scheduling a number of processors that does not change during run time. It is incurred once per task and is significant for short tasks. To evaluate this overhead, we run a simple program on a varying number of processors where each processor simply prints its task ID. Measuring the execution wall clock time is a good approximation for SSO.

Some compilers automatically parallelize loops. In the resulting code, during program execution, Processor 0 is always active. When it reaches a parallel loop, it activates the other available processors to participate in the work of the loop. The other processors go back to idle at the parallel-loop completion. The overhead of processor activation and deactivation for a parallel loop is the *parallel-loop scheduling overhead* (PLSO). To evaluate this overhead, we time a parallel loop that has one call to a subroutine consisting simply of a return statement (Null Subroutine).

4.7 Barrier Synchronization Time

In a shared-memory multiprocessor, a call to an explicit synchronization routine is often needed between code segments that access shared data. When a processor reads shared data that is modified by another processor, a barrier synchronization call before the read ensures that the other processor has completed its update.

We evaluate the time to call a barrier synchronization subroutine when all the processors enter the barrier simultaneously. This is implemented by making every processor call the subroutine for many iterations, and reporting the elapsed time per call.

4.8 Chapter Conclusions

In this chapter, we have presented an experimental methodology for measuring the memory and communication performance of a DSM multiprocessor. This methodology characterizes important performance aspects of local-memory and shared-memory accessing, scheduling, and synchronization.

For local-memory access, these microbenchmarks enable characterizing the processor data cache configuration, the latencies of cache hits and misses, and the achievable cache and memory transfer rates. As illustrated in the next chapter, they also expose memory system bottlenecks.

For shared-memory access, they enable characterizing the shared-memory communication cost as function of distance, access pattern, and number of processors. Moreover, they provide a characterization of the effects on performance due to the concurrent traffic when multiple processors are concurrently utilizing the shared memory and the interconnection network.

Although the static scheduling overhead is incurred once per task and is only significant for short tasks, it usually increases with more processors and gives an indication on the parallel environment performance in scheduling processors and distributing the task's code to the scheduled processors [AD96, AD98b]. The parallel-loop scheduling overhead (PLSO) also usually increases as the number of processors increases. Since the PLSO can be significant for tens of processors, it may not be profitable to parallelize a short loop. For a serial loop that takes T_0, parallelizing it with perfect load balance gives a parallel loop that takes $T_0/p + \text{PLSO}(p)$. Hence, parallelism may be profitable if $\text{PLSO}(p) < (1 - 1/p)T_0$, and the optimum p is $\sqrt{T_0/\frac{d}{dp}\text{PLSO}(p)}$.

We suggest that the experiment-based methodology presented in this chapter can be applied to existing shared-memory multiprocessors and that the resulting characterization is useful for developing and tuning shared-memory applications and compilers. We have also shown that a corresponding communication performance characterization of message-passing multicomputers can be systematically carried out [AD96].

CHAPTER 5

ASSESSING THE EFFECTS OF TECHNOLOGICAL ADVANCES WITH MICROBENCHMARKS

In this chapter, we use the microbenchmarks developed in Chapter 4 to calibrate and evaluate the performance of two generations of the HP/Convex Exemplar. We characterize the performance effects of the changes in microarchitecture and implementation technology. We also identify some remaining inefficiencies in the cache coherence protocol and node configuration that future systems should address.

5.1 Introduction

Distributed Shared Memory is becoming the prevailing approach for building scalable parallel computers. DSM systems use high-bandwidth, low-latency interconnection networks to connect powerful processing nodes that contain processors and memory. The distributed memory is shared through a global address space, thus providing a natural and convenient programming model [LW95, CSG98].

The SPP1000 was introduced in 1994 as Convex's first implementation of its Exemplar DSM architecture [Bre95]. In 1997, HP/Convex started to ship multi-node configurations of its second Exemplar generation, the SPP2000 [BA97]. Although the new system is upwardly binary compatible with older systems, it has significant differences that achieve higher performance and scalability. The SPP2000 uses a modern, superscalar processor that features out-of-order execution and non-blocking caches. Additionally, the SPP2000 has larger nodes, a richer interconnection topology, better packaging and manufacturing technology, and more optimized protocols to improve its memory latency and bandwidth.

The following section describes the architecture of the two systems and outlines their relevant differences. Section 5.3 outlines the performance aspects evaluated in this chapter. Section 5.4 compares the local memory performance of the two systems. Section 5.5 evaluates the various aspects of the communication performance when accessing shared memory. Section 5.6 evaluates the aggregate achievable bandwidth and the effects of contention. Section 5.7 evaluates an aspect of the synchronization overhead. Finally, Section 5.8 draws some conclusions and suggests future directions.

5.2 SPP2000 vs. SPP1000 – System Overview

The SPP1000 and the SPP2000 connect multiple nodes using multiple-ring interconnects. An SPP1000 node has 4 pairs of processors and 4 memory boards (with 2 banks each) connected by a crossbar; an SPP2000 node has 8 processor pairs and 8 memory boards (with 4 banks each) connected by a crossbar. Thus, an SPP2000 node has twice the processors and four times the memory banks of the SPP1000. Each processor pair (in both systems) has an *agent* that connects it to a crossbar port; each memory board has a *memory controller* that does likewise (Figure 5.1). Each SPP2000 node can be configured with up to 16 Gbytes of SDRAM; each SPP1000 node with up to only 2 Gbytes of DRAM. Yet, due to improved packaging and device density, a 2-node SPP2000 tower is physically the same size as a 2-node SPP1000 tower.

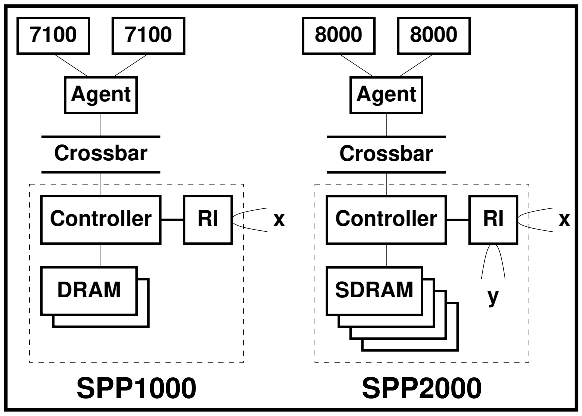


Figure 5.1: Each SPP1000 node has four of the left functional block. Each SPP2000 node has eight of the right functional block. RI is the ring interface.

The SPP1000 uses the Hewlett-Packard PA 7100 [AAD+93], a two-way superscalar processor, running at 100 MHz. The SPP2000 uses the PA 8000 [Hun95], a four-way superscalar processor, running at 180 MHz. In addition to the PA 8000's higher frequency and larger number of functional units, it supports out-of-order instruction execution, and memory latency overlapping and hiding. On HP workstations that use these processors,[1] the PA 8000 achieves 4.7x floating-point performance and 3.6x integer performance on SPEC95, relative to the PA 7100 [SPE].

Each processor has two off-chip data and instruction caches that are virtually addressed. The data caches within one node are kept coherent using directory-based cache coherence protocols.

[1] The HP 9000 Model 735/99 runs a PA 7100 at 99 MHz with 256-KB data and instruction caches. The HP 9000 Model C180-XP runs a PA 8000 at 180 MHz with 1-MB data and instruction caches.

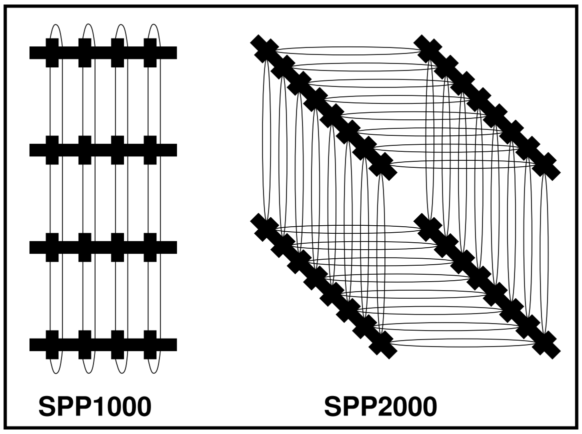


Figure 5.2: Four 8-processor SPP1000 nodes are connected in one dimension using 4 rings. Four 16-processor SPP2000 nodes are connected in two dimensions using 32 rings.

The SPP1000 uses a three-state protocol, and the SPP2000 uses a four-state protocol [SA95]. Each memory line has an associated coherence tag to keep track of which processors have cached copies of this line.

In the SPP1000, each memory controller has a *ring interface* that connects it to one ring. Thus, a multi-node SPP1000 system has 4 rings. Each ring carries the remote memory accesses to the corresponding two memory banks of each other node in the system. The SPP1000 interconnects up to 16 nodes using these 4 rings. Similarly, the SPP2000 has one ring interface per memory board, and two or three node systems simply use 8 rings. However, rather than one port, each ring interface has two ring ports which allow a two-dimensional ring interconnection topology. For example, four SPP2000 nodes can be interconnected in a 2 by 2 configuration using a total of 32 rings (Figure 5.2). The largest SPP2000 configuration is 4 by 8, comprised of 32 nodes interconnected with 96 rings. The two-dimensional topology provides lower latency and higher bisection bandwidth.

The two systems use variants of the IEEE Scalable Coherent Interface [SCI93] to achieve internode cache coherence. The SCI protocol uses a distributed doubly-linked list to keep track of which nodes share each memory line. Each node that has a copy of a memory line, maintains pointers to the next forward and next backward nodes in that line's sharing list.

Most of the glue logic that coherently interconnects the distributed processors and memory banks is in custom-designed gate arrays [AB97]. The SPP1000 uses 250-K gate gallium arsenide technology, which provided the speed to pump a 16-bit flit onto the ring each 3.33 nanoseconds. The SPP2000 uses 0.35-μ CMOS technology, which has evolved to provide competitive speeds in addition to its lower power consumption and higher integration. The SPP2000's ASICs are implemented using Fujitsu's 1.1-M gate arrays clocked at 120 MHz. However, the higher integration of these chips provided wider data paths and richer functionality. Each port of the SPP2000 ring interface can pump a 32-bit flit per 8.33 nanoseconds.

5.3 Evaluation Methodology

Although some applications are latency limited, e.g., those that do frequent pointer chasing, and some others are bandwidth limited, e.g., those that frequently move large data blocks, many applications benefit from both low latency and high bandwidth.

In this evaluation, we have used the microbenchmark introduced in Chapter 4. The building blocks of the memory and communication microbenchmarks are the memory kernels that are called many times to measure the minimum and average *call times*. Each kernel accesses the l-byte elements of an array of size w bytes with stride s bytes. Thus, one call time corresponds to the w/s accesses in a kernel call. In this chapter, $l = 4$ for latency kernels, and $l = 8$ for bandwidth kernels.

Load latency is measured by the `load-use` kernel, which uses the loaded value to find the address of the next load. The data dependency in the `load-use` kernel serializes the load accesses. The `store-load-use` kernel measures the store latency. In each iteration, this kernel performs a store to one element followed by a load from an adjacent element. The loaded value is used to find the address of the next store and establish a data dependency. The runtime of the `store-load-use` kernel is some number of store latencies plus other known times. The `load` kernel and the `store` kernel measure bandwidth by simply generating a sequence of loads or stores, respectively, without any data dependencies.

These four kernels are the building blocks of the SPP2000 experiments. However, since the PA 7100 does not overlap cache misses, the `load` and `store` kernels are used to measure bandwidth as well as latency in the SPP1000 experiments. The benchmarked SPP1000 and SPP2000 systems have 4 nodes each. The SPP2000 nodes are interconnected in a 2 by 2 configuration, while the SPP1000 nodes lie along one dimension.

The following performance aspects of the two systems are evaluated in the following sections:

Local memory: the latency and bandwidth of accessing a private array that is allocated in the local memory. The array size is varied to characterize cases where the array fits in the processor data cache through cases where all accesses are satisfied from the local memory. The access stride is varied to expose processor and system bottlenecks.

Remote memory: the latency of accessing a shared array that is allocated in a remote node. The array size is varied to characterize the performance of the *interconnect cache*, which is described in Section 5.5.1.

Producer-consumer communication: the latencies when two processors access a shared array in a producer-consumer pattern. The locations of the two processors are varied to characterize intranode and internode communication.

Effect of distance: the latency as a function of the distance between two processors engaged in an internode producer-consumer communication.

Effect of home location: the latency as a function of the node location where the shared array is allocated (home node) in the internode producer-consumer communication.

Coherence overhead: the overhead of maintaining cache coherence as a function of the number of processors engaged in producer-consumer communications.

Concurrent traffic effects: the aggregate bandwidth when multiple processors are active in performing memory accesses.

Barrier synchronization time: the time to perform a barrier synchronization as a function of the number of processors when the processors reach the barrier simultaneously.

5.4 Local Memory

This section evaluates the local-memory performance when only one processor is accessing the local memory in its node. This performance is a function of the performance of the processor, processor cache, processor bus, processor agent, crossbar, memory controller, and memory banks.

5.4.1 Local Memory Latency

Figure 5.3 shows the average load latency of the two systems as a function of array size for the access strides $s = 8, 16, 32, \ldots$ bytes. For both systems, the load latency with $s \geq 64$ bytes equals the latency with $s = 32$ bytes, indicating that the cache line size is 32 bytes. The hit region ends at $w = 1024$ KB and the transition region ends at $w = 2048$ KB, indicating that the data cache is a direct-mapped 1024-KB cache [AD98b]. In the hit region, every access hits in the cache, while in the miss region ($w \geq 2048$ KB), every access to a new cache line is a miss.

The SPP2000 load miss latency is 0.52 μsec, which is slightly lower than 0.55 μsec of the SPP1000.[2] However, when considering clock cycles, the SPP2000 latency of 93 cycles is 69% more than the SPP1000 latency. The slight time advantage of the SPP2000 may be attributed to the slight speed advantage of the SDRAM over the DRAM and the lower processor bus and crossbar transfer times.

It is worthwhile to compare this latency to a comparable uniprocessor system. The HP 9000 Model C160 workstation runs a PA 8000 with a 160-MHz clock and has a load miss latency of

[2]The SPP1000 load latency changes slightly from one experiment to another: 0.55 μsec is the smallest latency measured in the miss region.

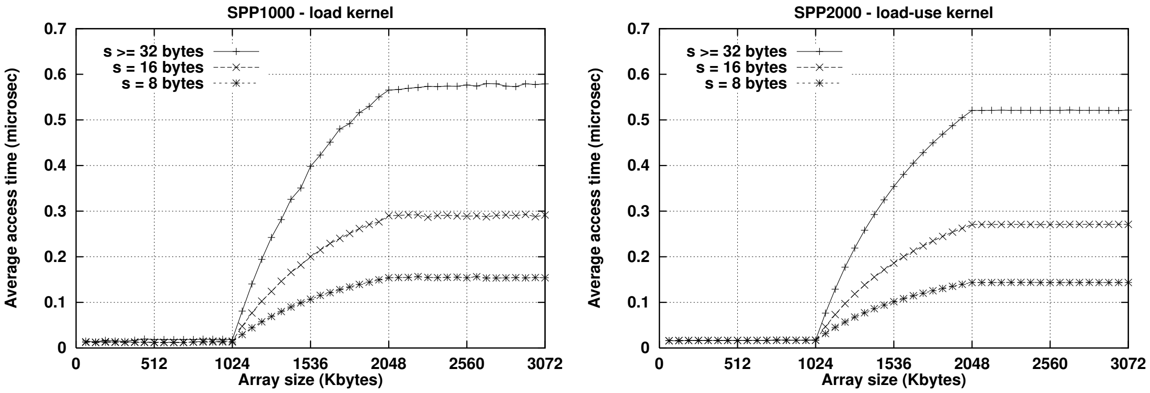


Figure 5.3: Average load latency as a function of array size w. The three regime regions are: hit region ($0 < w \leq 1024$ KB), transition region ($1024 < w < 2048$ KB), and miss region ($w \geq 2048$ KB).

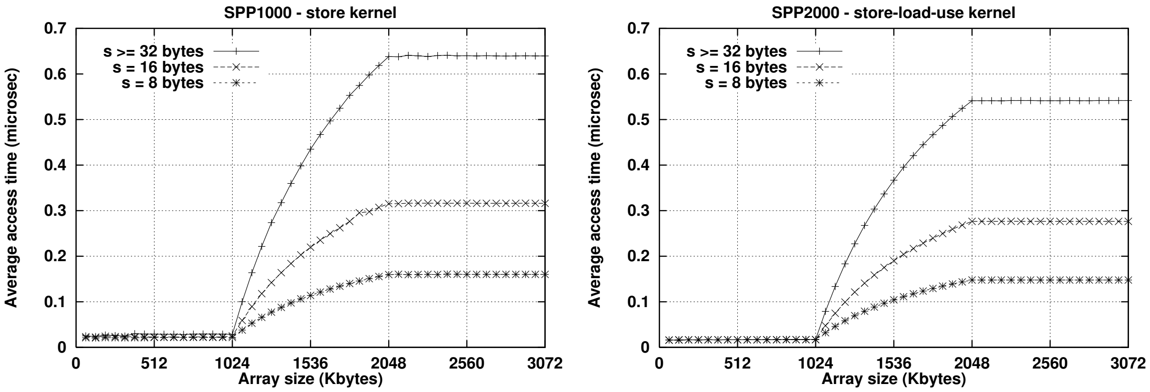


Figure 5.4: Average store latency.

0.32 μsec. It seems that the cost of connecting 16 processors within a coherent SPP2000 node adds an additional 0.20 μsec latency, which penalizes serial applications.

Figure 5.4 shows the average store latency of the two systems. The store miss latency, measured as the latency from issuing the store to completing the allocation in the cache of the missed line, is slightly higher than the load miss latency due to the overhead of flushing the dirty replaced line.

In the SPP2000, the `load-use` and `store-load-use` kernels have 3-cycle access latency in the hit region. This latency is composed of 1 cycle for calculating the address plus 2 cycles cache latency, which is twice the SPP1000 cache latency.

5.4.2 Local Memory Bandwidth

When repeating the above experiments using the bandwidth-measuring kernels, the two systems exhibit very different performance. In the hit region, the SPP1000 can perform 1 load every cycle or 1 store every 2 cycles. For $s = 8$ bytes, the SPP2000 uses the two PA 8000 cache ports to perform

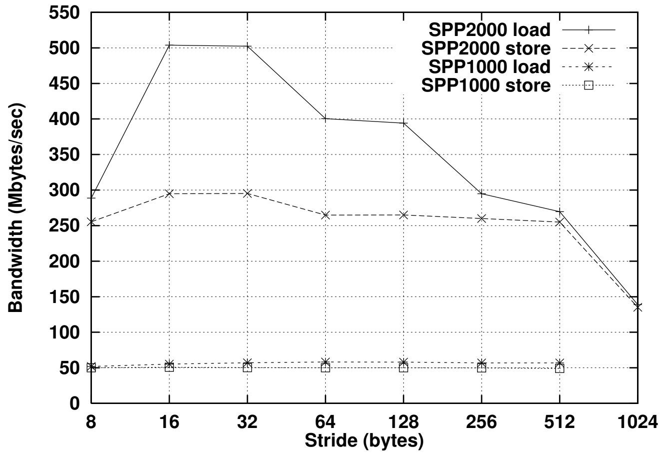


Figure 5.5: Memory transfer rate in the miss region.

2 loads per cycle or 1 store per cycle. Thus, the SPP1000 maximum cache transfer rate is 800 MB/s for loads and 400 MB/s for stores, when accessing double-word elements with a 1-element stride. The SPP2000 maximum transfer rates are $(2 \times 180/100)$ times larger. However, with larger strides, the SPP2000 performs only 1 access per cycle because the cache has one bank for the even double-words and one bank for the odd double-words and each cache port accesses an associated bank. Thus, with $s = 16, 32, \ldots$ bytes, only one port is utilized.

Figure 5.5 shows the memory transfer rates as a function of the access stride for the two systems in the miss region. The transfer rate is found as the cache line size divided by the average access time per line. Note that for strides larger than one element, some of the transferred bytes are not used. The store transfer rate is smaller than the load transfer rate due to the additional traffic generated to write back the replaced dirty lines.

The best-case memory transfer rate in the SPP2000 is about 10 times that of the SPP1000. The SPP1000's transfer rate of about 50 MB/s is limited by the latency of the individual misses because the PA 7100 allows only one outstanding miss. This small transfer rate does not stress the memory system and is independent of the access stride. Whereas, the PA 8000's ability to have up to 10 outstanding misses enables it to achieve much higher transfer rates, and exposes a number of bottlenecks as apparent from the variation in transfer rate as the access stride changes.

The first bottleneck at $s = 8$ bytes is due to the PA 8000's instruction reorder buffer size, which has up to 28 entries for loads, stores, and some other instructions. With this stride, every fourth access is to a new cache line, therefore, the PA 8000 can have at most $28/4$ dispatched instructions that generate cache misses. For $s = 8$ bytes, the degree of miss latency overlapping in the `load` kernel is about 6 ($10 \times 290/500$, assuming that there are 10 outstanding misses at the peak bandwidth).

The other bottlenecks that occur with $s \geq 64$ bytes are external to the PA 8000. In the SPP2000,

s	Accessed banks
8–32	All banks
64	The four banks of boards 0, 2, 4, and 6.
128	The four banks of boards 0, and 4.
256	The four banks of board 0.
512	Banks *i* and *iii* of board 0.
1024	Bank *i* of board 0.

Table 5.1: Accessed SPP2000 memory banks for strides 8 through 1024 bytes. There are 8 memory boards (0 through 7) each has four banks (*i*, *ii*, *iii*, and *iv*).

the memory is 32-way interleaved; consecutive 32-byte memory lines are first interleaved across the 8 memory boards then across the 4 banks of each board. Table 5.1 shows which memory banks are accessed for strides 8 through 1024 bytes. The memory transfer rate declines as fewer memory boards or fewer memory banks are utilized. From $s = 256$ bytes, we see that a single memory board supports up to 295 MB/s to one processor; from $s = 1024$ bytes, we see that a single memory bank supports up to 139 MB/s.

The store transfer rate for strides 16 and 32 bytes is limited by a bus bandwidth bottleneck. Recall that the `store` kernel transfers two lines for each miss, a fetched line and a dirty replaced line. The SPP2000 uses the Runway bus [BCF96], which is an 8-byte wide bus clocked at 120 MHz. This bus multiplexes data and addresses with a 4:2 ratio. Thus, the available bandwidth is 640 MB/s; the `store` kernel achieves 92% of this limit.

5.5 Shared Memory Communication

The memory kernels are also used here to evaluate the performance of accessing shared memory. The accessed array is allocated in shared memory and each experiment uses multiple processors. This section characterizes producer-consumer performance, including the effects of communication distance, home node, and degree of sharing. Since the interconnect cache affects the latency of accessing remote memory, we start with an evaluation of this cache.

5.5.1 Interconnect Cache

The interconnect cache is a dedicated section of each node's memory. The IC in each node exploits locality of reference for the remote shared-memory data (shared data with a home memory location in some other node). When a remote shared-memory access misses in both the processor's data cache and the node's IC, a memory line is retrieved over the ring interconnect through its home node. This line is then stored in the local IC as well as in the processor's data cache. Hence, subsequent references to this line that miss in one of the data caches of this node can be satisfied

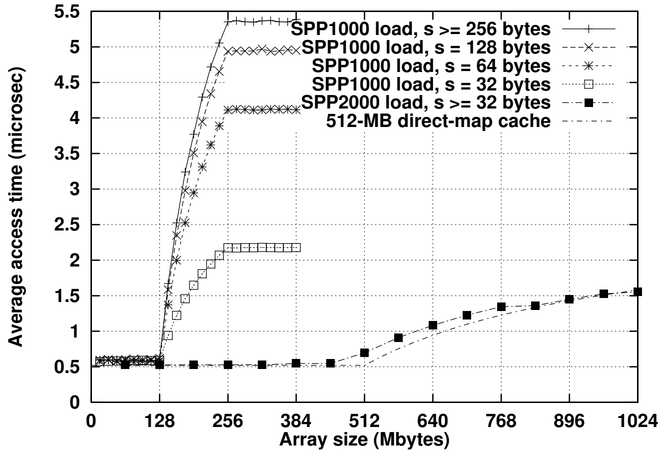


Figure 5.6: Average latency using the `load-use` kernel for an array allocated in a remote node.

locally from the IC, until such time as this line is replaced in the IC or invalidated due to store by a remote node. The IC size is selectable by the system administrator, and is usually selected to achieve the best performance for frequently executed applications.

The microbenchmark that evaluates IC performance uses two processors from distinct nodes. The first processor allocates a local shared array and initializes it. The second processor accesses the array repetitively using the `load-use` kernel (the `load` kernel in the SPP1000 case). Figure 5.6 shows the average load time of the second processor as a function of the array size.

The SPP1000 node's IC is set to 128 MB and the SPP2000 node's is 512 MB. For arrays that fit in the IC, the load time is similar to the local-memory load time shown in Figure 5.3.

In the SPP1000, the transition region is 128 MB wide, which indicates that the IC is direct mapped. For array sizes larger than 256 MB, no part of the array remains in the IC between accesses to it in two successive kernel calls, therefore, the accesses are satisfied from the remote memory. Although the maximum local-memory latency in the SPP1000 is reached with $s = 32$ bytes (1 processor cache line), the remote load latency almost doubles as s increases from 32 to 64 bytes, and continues to increase to reach a maximum at $s = 256$ bytes. In contrast, the SPP2000 remote load latency does not change for $s \geq 32$ bytes.

In the SPP1000, the *memory line size* is 64 bytes, twice the size of the processor cache line. The 64-byte memory line is the base unit for maintaining internode coherence in the SPP1000. Therefore, for $s \geq 64$ bytes in the miss region, every access is an IC miss, while for $s = 32$ bytes, every two processor cache misses generate one IC miss. The miss latency continues to grow as s increases from 64 to 256 bytes due to ring congestion, which increases as fewer rings are used to serve the missed lines that are interleaved across the memory banks; all 4 rings are used with $s \leq 64$ bytes, 2 rings with $s = 128$ bytes, and only one ring with $s = 256$ bytes. A ring apparently remains busy after fetching a line from the remote node (in order to collapse the sharing list of the replaced IC line).

Unlike the general trend toward wider caches, the IC line is shorter in the SPP2000 (32 bytes, as in the processor cache). Since the coherence tags attached to each memory line require 8 bytes in both systems, the shorter line results in higher overhead for coherence tags relative to user-available memory (1:4 in the SPP2000, and 1:8 in the SPP1000), and they provide less prefetching for applications with spatial locality.

However, shorter lines enabled the SPP2000 to achieve lower remote latency because it is faster to extract a 32-byte line from the memory and transfer it over the ring interconnect, the coherence protocol is simpler, and the data transfer is more streamlined. Furthermore, the PA 8000 allows multiple outstanding misses, which reduces the prefetching disadvantage of short lines.

As s increases and fewer rings are used to satisfy remote misses, the SPP2000's latency, unlike the SPP1000, remains constant, implying no ring congestion even when only one ring is utilized with $s \geq 256$ bytes. This improved performance is achieved by supporting up to 32 outstanding requests in each ring interface, rather than only one [AB97]. Therefore, the SPP2000 ring controller can process a line request concurrently with collapsing the sharing list of a replaced line.

Notice that the SPP2000's curve shape differs slightly from the curve of the 512-MB direct-mapped cache model shown [AD98b]. In fact, it changes from one experiment to another according to the OS mapping of the array's virtual pages to the physical memory pages. The curve shown indicates that some of the array's physical pages conflict in the IC for $w = 512$ MB, resulting in an average latency that is higher than the IC hit latency. The SPP2000 data was obtained by running this microbenchmark on a recently booted system. The number of IC conflicts generally increases with time as virtual to physical mapping becomes almost random. The IC addressing scheme in the SPP1000 is different, and never revealed any IC conflicts in our experiments.

5.5.2 Communication Latency

Figures 5.7 and 5.8 show the latencies of the two phases of shared-memory producer-consumer communication as a function of the access stride. Two processors repetitively take turns accessing a shared array, Processor 0 executes the `store-load-use` kernel (produces), then after synchronization, Processor 1 executes the `load-use` kernel (consumes), then after synchronization, this process is iterated. The time that Processor 0 spends in the `store-load-use` kernel is used to find the average write-after-read (WAR) latency, and Processor 1's time is used to find the read-after-write (RAW) latency. Latencies are shown for the two cases of processor allocation: *near*, when both processors are in the same node, and *far* when they are from distinct nodes.

The array size is 1 MB, which fits in the processor data cache. In the near case, this array is allocated in the local shared memory; whereas, it is interleaved across the shared memory of all

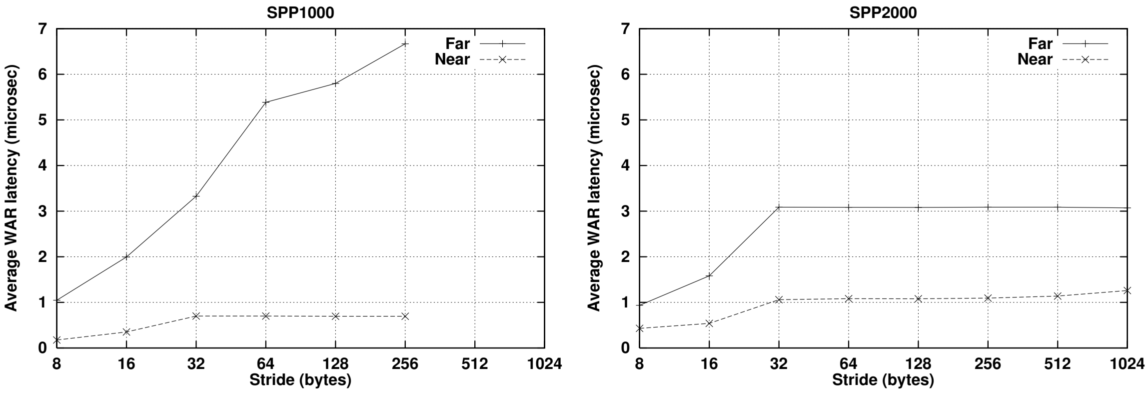


Figure 5.7: Write-after-read access latency as a function of the access stride. Far WAR for the case when the two processors are from distinct nodes, and near WAR when the two processors are from the same node.

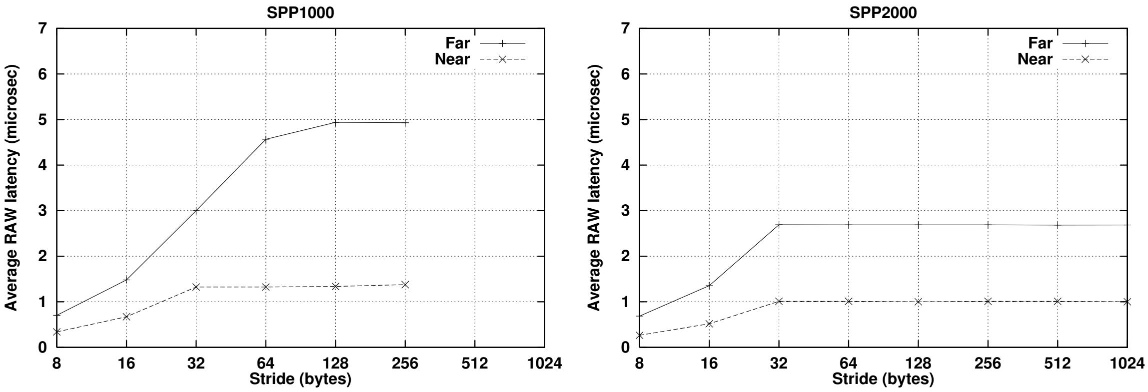


Figure 5.8: Read-after-write access latency.

nodes in the far case.

In the SPP1000, the far latency increases as the stride increases due to the increase in misses per reference (up to $s = 64$ bytes), and then the decrease in the number of rings used. In contrast, as s is increased beyond 32 bytes, the SPP2000 far latency generally does not increase, indicating no ring congestion. WAR generates more ring traffic than RAW because in addition to joining at the head of the line's sharing list, the WAR sends invalidation signals to the other sharing nodes. Thus, SPP1000 far WAR latency increases faster than RAW latency as fewer rings are used.

Far latency is higher than the near latency due to the overheads of the internode coherence protocol and the added latency of the ring interconnect. As with IC miss latency, the SPP1000 far latency at $s \geq 64$ bytes is nearly double the SPP2000 far latency.

However, the SPP2000 near WAR latency is higher than that of the SPP1000. In the SPP2000 intranode coherence protocol, the consumer gets an exclusive copy of the loaded line from the producer's cache, and the producer loses its copy. Therefore, when the producer subsequently updates this line, it must wait to check whether the consumer's copy has been modified. This check lengthens the near WAR latency, and seems to have some residual effect that slightly increases the latency when one bank is utilized at $s = 1024$ bytes. In the SPP1000, the consumer always gets a shared copy, which is invalidated concurrently when the memory controller returns a writable copy to the producer, without waiting to check the consumer's copy status.

In the SPP2000, although giving an exclusive copy of a modified line hurts repetitive near producer-consumer communication, it is profitable for migratory lines (lines that are accessed largely by one processor at a time [ZT97]). When a processor gets exclusive ownership of a migratory line, it does not need to request ownership when it subsequently updates this line.

When the consumer performs a RAW access, the valid copy of the accessed line is in the producer's cache. Although, Section 5.4.1 showed that the two systems have similar local-memory latency, the SPP1000 has about 33% higher latency than the SPP2000 in near RAW when accessing the copy in the producer's cache.

5.5.3 Effect of Distance

In the SPP1000, each remote access has a request path and a reply path, which collectively circle one ring. The remote latency is not affected by where the remote node is positioned on the ring—it depends only on the number of system nodes. However, the SPP2000 remote latency does depend on the remote node location.

Consider a far RAW access to a line that is valid in the memory of the producer's node (the producer's node is the home of this line). This simple access is satisfied by one request and one response. On the SPP2000, this takes 1.58 μsec between two adjacent nodes, and 1.87 μsec when the two nodes are on opposite corners of the 2 by 2 configuration. The additional 0.29 μsec is needed to change dimension and circle a second ring.

For a 4 by 2 SPP2000 system, it still takes 1.58 μsec between two nodes directly connected by 2-node rings, but 1.73 μsec between two nodes directly connected by 4-node rings, and 2.04 μsec otherwise. A simple model that is consistent with these observations is that this RAW access takes 1.43 μsec + 0.075 μsec per node to node link traversal of a ring + 0.15 μsec if two rings are traversed.

For the largest SPP2000 configuration, 4 by 8, the latency of this access is calculated as 1.73 μsec for two nodes directly connected by 4-node rings, 2.03 μsec for two nodes connected by 8-node rings, and $1.43 + 0.075 \times 12 + 0.15 = 2.48$ μsec for nodes not on the same ring. Had the SPP2000 used a one-dimensional topology, the latency of this access on a 32-node system with these parameters would always be $1.43 + 0.075 \times 32 = 3.83$ μsec.

5.5.4 Effect of Home Location

The shared array used in characterizing the far producer-consumer communication in Section 5.5.2 is interleaved among the memory banks of the four nodes. Thus, the measured far WAR (or far RAW) latency is the average of four cases according to the location of the home node. The way that the internode coherence protocol satisfies a cache miss does in fact depend on the home node of the missed line [ABP94, SCI93]; the three main cases are:

Local home: the line's home node is the local node.

Remote home: the line's home node is the remote node.

Third-party home: the line's home node is a third node other than the local and remote nodes (in a 4-node system, there are two third-party nodes).

Table 5.2 shows the WAR and RAW latencies of the three cases of home node locations in the SPP2000 with $s = 32$ bytes. When measuring the latency of the first two cases, the two nodes involved are adjacent. For the third-party home case, the remote node is adjacent to the local node in the x direction and the home node is adjacent to the local node in the y direction. The table shows the latencies of cached and uncached WAR and RAW accesses. For the cached accesses, the shared array size is selected to fit in the processor cache; thus, a WAR or RAW access occurs when the remote processor has a copy of the line (the valid copy in RAW). For the uncached accesses, the array size is twice the processor cache size; thus, a WAR or RAW access occurs when the remote memory or the remote IC has a copy and the remote processor does not.

WAR latency is higher than RAW latency because WAR involves getting exclusive ownership and invalidating other copies. The latency of cached accesses is equal or higher than that of uncached accesses.

WAR latency generally increases as we move from local to remote to third-party homes. Uncached WAR latencies with remote and third-party homes are identical to those of cached WAR because, as the processor does not inform the directory when it replaces a clean line, invalidations are sent to the remote processor in both uncached and cached WAR accesses. However, the latency

Type	Cached	Local	Remote	Third-party
WAR	No	2.08	2.91	3.57
	Yes	3.37	2.91	3.57
RAW	No	2.07	1.58	2.26
	Yes	2.64	2.79	2.83

Table 5.2: The SPP2000's WAR and RAW latencies, in microseconds, according to the location of the home node and the caching status of the remote processor.

of the cached local-home WAR is high due to complications of the SCI coherence protocol. In uncached WAR, the line's sharing list has one entry that points to the remote node, and the WAR is performed by sending an invalidation to the remote node. In cached WAR, the line's sharing list has two nodes; the remote node at the head and the local node at the tail. In order to modify this line, the SCI protocol requires that the local node leaves the sharing list, rejoins at the sharing list head, and invalidates the other node, which sums up to a latency of 3.37 μsec (only the head of the list can invalidate copies in other nodes).

Cached RAW latency is higher than that of uncached RAW because of the additional latency of retrieving the modified copy from the remote producer's cache after checking the coherence tag in the remote node. This retrieval is not needed in uncached RAW where the remote IC or remote memory has the valid copy. SCI complications also increase the gap between cached and uncached remote-home RAW latencies. Uncached remote-home RAW latency is the lowest latency among remote accesses (only 1.58 μsec), as it is simply satisfied through a request to the home node and a reply with data from the home node's memory. However, cached remote-home RAW is more expensive because it comes after the expensive cached local-home WAR described above. When this RAW occurs, the line's sharing list has the home node at the head. The SCI protocol handles this RAW similar to the third-party home case where the requester node first joins the sharing list at the head, then gets the modified line from the old head.

5.5.5 Coherence Overhead

This section evaluates latency as a function of the number of processors, p, involved in shared-memory producer-consumer communications in order to characterize the coherence overhead. In the microbenchmark used, Processor 0 uses the `store-load-use` kernel to access a shared array that is interleaved among all nodes, and the other $p - 1$ processors take turns accessing it using the `load-use` kernel. The WAR time of Processor 0 is the *invalidation time*, shown in Figure 5.9 as a function of $p - 1$. The times for the other processors depend on the order in which they read. Figure 5.10 shows the incremental *read times* for the $p - 1$ reader processors from experiments using 24 and 64 processors on the SPP1000 and SPP2000, respectively. In these experiments, the processors read in the order of their IDs. The accumulative sum of the invalidation time and the $p - 1$ read times is the time of one iteration of this microbenchmark.

The invalidation time increases as the access stride increases, for the same reasons outlined earlier. This time also increases, in steps, as the number of processors increases. When all processors are in the same node, the invalidation time remains constant due to the crossbar's broadcast capability. There is a large step when the new processor is from the second node, and other smaller steps as each subsequent node is added. For the SPP2000, with $p = 2$, the invalidation time is higher than that with $p = 3$ or more within one node. When there is only one reader, the invalidation time is

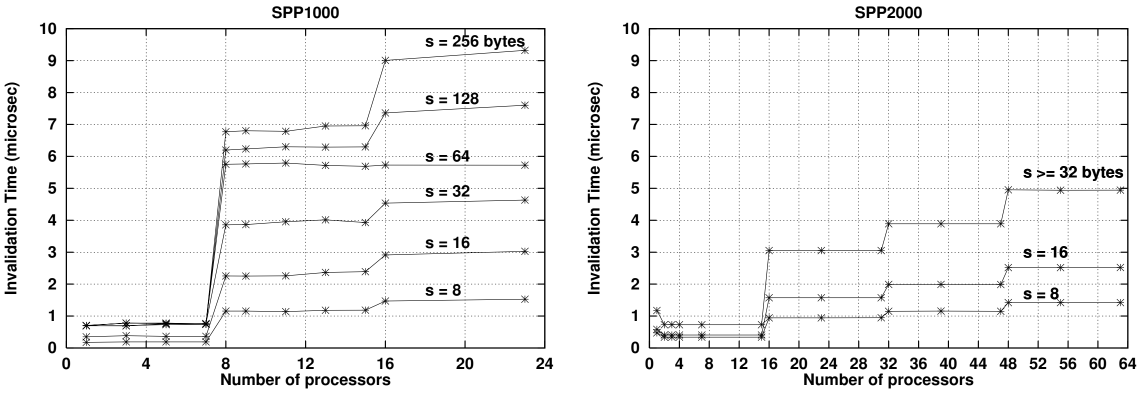


Figure 5.9: Invalidation time as a function of the number of sharing processors.

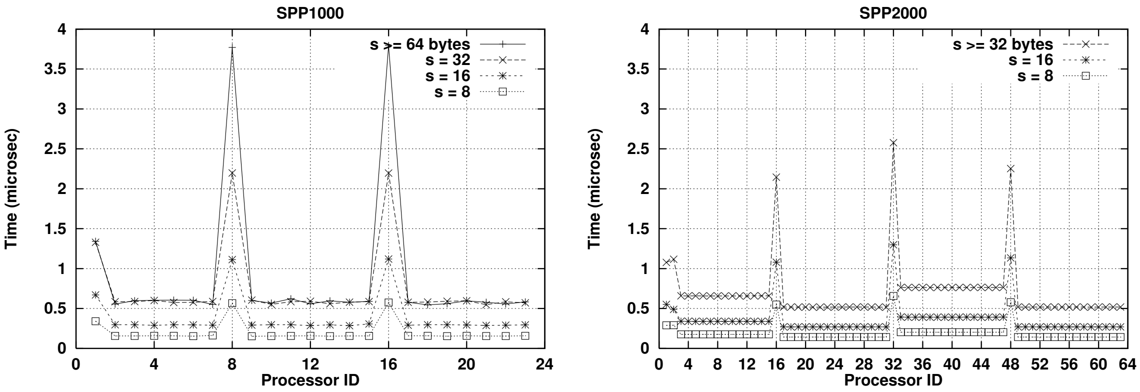


Figure 5.10: Incremental read time for each processor.

longer in order to check the status of the exclusive copy in the reader's cache; with $p > 2$ there is no exclusive copy when invalidation is being done.

In the SPP1000, as suggested by Figure 5.10, Processor 1 reads from the writer's cache causing the memory to be updated as a side effect. The read time of Processors 2 through 7 is thus less since they are satisfied from the local memory that they share. The read time is higher for Processor 8 since the data is not in its node and must be provided remotely. When Processors 9 through 15 read, they find the data in their node, and their read time is similar to Processors 2 through 7. This sequence repeats for each node.

The SPP2000 read time is similar with three differences: (i) the number of processors per node is 16, (ii) the second reader also sees a high latency in order to check the status of the exclusive copy in the first reader's cache, and (iii) the read time changes from one node to another because of varying numbers of IC conflicts.

5.6 Concurrent Traffic Effects

In this section, we use the bandwidth-measuring kernels to evaluate the aggregate bandwidth when multiple processors are active accessing memory.

Figure 5.11 shows the aggregate load bandwidth from local memory when p processors from one node use the `load` kernel to access a $2p$-MB array, simultaneously. Each processor accesses a disjoint 2-MB segment with stride s. The segment size is selected to evaluate the local-memory performance through the crossbar in the miss region.

As the memory bandwidth of one SPP1000 processor is limited by the latency of the individual cache misses, one processor does not stress the memory system, and even the aggregate bandwidth is nearly proportional to p for $s \leq 128$ bytes. The maximum local memory load bandwidth for 8 processors is 420 MB/s when uniformly accessing all the node's memory banks. However, the aggregate bandwidth is significantly lower, 320 MB/s, when accessing one memory board ($s = 256$ bytes), and 200 MB/s when accessing one bank ($s = 512$ bytes).

As discussed in Section 5.4.2, due to the high bandwidth requirements of the PA 8000, the bandwidth to one SPP2000 processor drops when the access stride skips some of the memory banks. Additionally, the local memory cannot provide enough bandwidth to allow linear scaling of the aggregate bandwidth as the number of active processors increases. The aggregate bandwidth falls off even faster when the stride begins to skip memory banks. The maximum local-memory load bandwidth for 16 processors is 4020 MB/s when uniformly accessing all the node's memory banks, 560 MB/s when accessing one memory board, and 300 MB/s when accessing one bank.

Figure 5.12 shows the aggregate remote RAW bandwidth when p processors from one node use the `store` kernel to update disjoint 2-MB segments of a $2p$-MB shared array, simultaneously. Then p processors from another node use the `load` kernel to access these array segments, simultaneously. The `load` kernel time is used to find the aggregate remote RAW bandwidth through the rings.

As the number of processors increases, the remote RAW bandwidth scales worse than the local

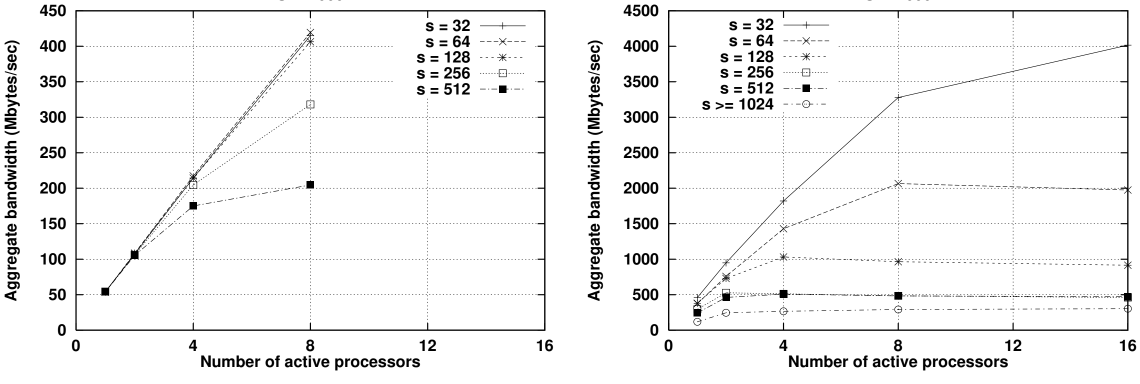


Figure 5.11: Aggregate local load bandwidth (note change of scale).

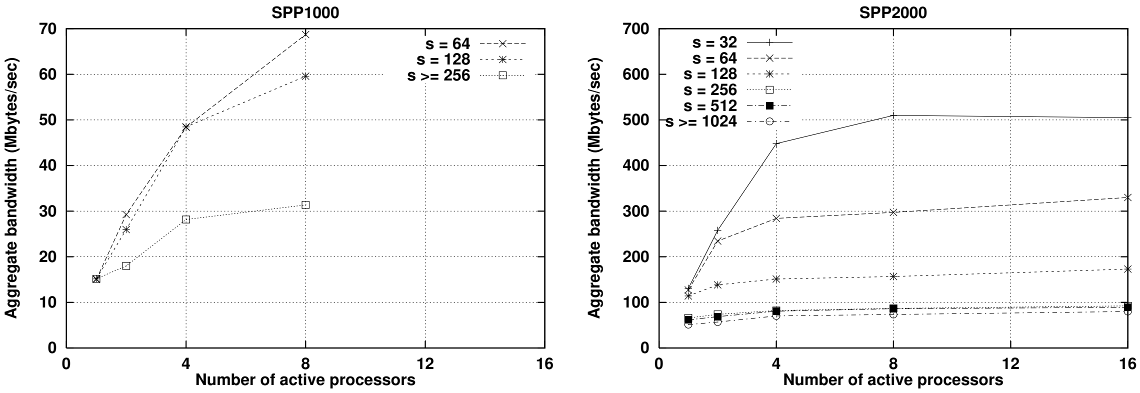


Figure 5.12: Aggregate remote RAW bandwidth (note change of scale).

load bandwidth. The SPP1000 remote bandwidth is adversely affected by the limitation that each ring interface allows only one outstanding request. This limitation makes communication through the rings a bottleneck, which becomes more severe as s increases and fewer rings are utilized.

The SPP2000 remote bandwidth is much higher since its ring interfaces each allow multiple outstanding requests. This high aggregate remote bandwidth can only be achieved by overlapping the remote latencies. The remote bandwidth scalability limitation here is mainly due to the complexity of the internode coherence protocol. The SCI requires at least four memory accesses for each RAW access: one to check the local IC, a second to retrieve the line from the remote memory, a third to update the remote memory coherence tag, and the fourth to put the line in the local IC.

In the SPP1000, the aggregate remote RAW bandwidth with 8 processors is 69 MB/s when all four rings are utilized, and falls to 31 MB/s with one ring. In the SPP2000, with 16 processors, this bandwidth is 510 MB/s when all memory banks are utilized, and falls to 80 MB/s with one memory bank.

5.7 Synchronization Time

The SPP2000 has new primitives to improve synchronization performance. Figure 5.13 shows the average time spent in the `WAIT_BARRIER` library subroutine when all the processors enter the barrier simultaneously.

As visible from the measured data and the curve fits, the SPP2000 has much improved performance. The SPP2000 uses the new `coherent_increment` primitive to reduce ring traffic and to lower barrier times [BA97]. In the SPP1000, when a thread reaches the barrier, it increments a counter and waits on a flag. When the last thread reaches the barrier, it updates the flag to signal to the other threads to go on. This update invalidates the cached flag copies by destroying the sharing list. The next access of each thread acquires the updated flag value and rebuilds a new sharing list, which generates a lot of traffic and contention. In the SPP2000, this traffic and latency is avoided

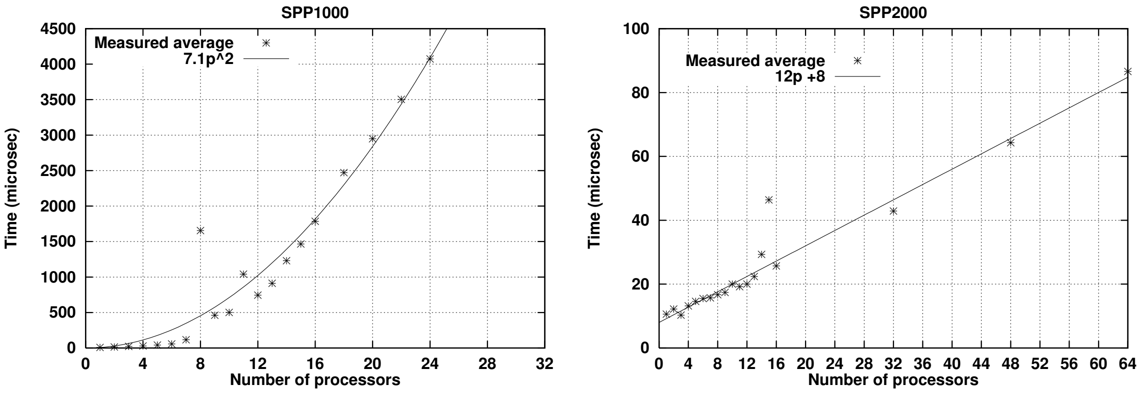


Figure 5.13: Barrier synchronization time (note change of scale).

by the `coherent_increment` primitive which uses the sharing list to update the flag copies in the sharing nodes, without destroying the sharing list.

5.8 Chapter Conclusions

Exploiting advances in microarchitecture, integrated circuit technology, and packaging, the SPP2000 achieves much better performance and scalability than its predecessor, the SPP1000. Although the memory latency, in processor cycles, has not improved, the newer processor overlaps multiple outstanding cache misses to allow up to 10x local memory bandwidth. However, the PA 8000 processor's maximum bandwidth is only achievable when memory load accesses are uniformly distributed over the memory banks and not hindered by accesses from other processors. The 32 memory banks in one SPP2000 node can actually sustain 50% of the maximum bandwidth of all 16 processors in the node, even without intervening cache hits. In the SPP1000, the PA 7100's lower maximum bandwidth allows the 8 memory banks in a node to sustain all 8 node processors at maximum bandwidth.

The latency of local communication through shared memory accesses has changed. The latency of loading a cache line that is dirty in the cache of another local processor has improved. The new four-state intranode coherence protocol reduces communication for private and migratory access patterns, at the expense of repetitive producer-consumer communication.

The two-dimensional ring interconnection topology of the SPP2000 provides much lower internode latency and higher bisection bandwidth than the SPP1000's single-dimension rings. The SPP2000 shrinks the gap between the remote and local latency further by adopting a smaller line size for internode communication, which now matches the line size of the processor. The SPP2000 remote latency is about 0.5x that of the SPP1000, and its remote bandwidth is about 10 times higher for a single active processor, thanks to the ability of the processor and the ring interface to overlap many memory requests. The SPP1000 ring interface, which serves outstanding requests serially, reduces the one processor maximum remote bandwidth by 42% when all 8 processors in a node are actively utilizing the rings. Since it requires multiple memory accesses per remote access, the SPP2000 internode coherence protocol reduces this bandwidth by 76% for 16 processors.

The SPP2000 decision to use custom-designed crossbar and controllers to support the needed aggregate bandwidth allowed it to have 16 processors per node, which greatly helps medium-scale parallel programs. The addition of this glue logic, however, raises the local memory and communication latencies, which penalizes sequential and small-scale parallel programs. Many modern processors are incorporating interfaces for small-scale multiprocessor buses, which can be exploited to build economic small nodes with low local latency [LL97]. However, for a system based on small nodes to succeed with large-scale parallel programs, the remote latency should not increase prohibitively.

Since the Exemplar does not strictly adhere to the SCI standard and does not use second-party SCI-standard components, we do not see a justification for using this protocol in such a high-performance parallel system, other than to avoid developing a new internode coherence protocol. This study has exposed several problems related to SCI efficiency and performance. The interconnect cache, which assists internode accesses, did take 12.5% of the physical memory space of the benchmarked systems, not including the 25% overhead of the coherence tags in the SPP2000. The distributed linked-list nature of the SCI directories unnecessarily increases the latencies of some remote accesses like cached WAR of local-home lines and cached RAW of remote-home lines. Additionally, the invalidation time is linear with the number of sharing nodes. Although some of these issues are addressed in research work such as [KG96], linked-list protocols fundamentally require more directory accesses than other protocols [AD98a, CFKA90, CSG98]. Since the SPP2000 directory is implemented in memory, the high directory access rate adversely affects remote memory latency and bandwidth.

CHAPTER 6

EVALUATION OF THREE MAJOR CC-NUMA APPROACHES USING CDAT

In this chapter, we present the results of our comparative study of three scalable shared-memory systems. The purpose of this study is to evaluate alternative scalable shared-memory system designs, identify strengths and weaknesses in current systems, and identify design aspects that have high positive impact on such systems and areas that need further investigation and improvement.

6.1 Introduction

An important approach in building scalable shared-memory multiprocessors is cache-coherent non-uniform memory access architecture. CC-NUMA systems use high bandwidth, low latency interconnection networks to connect powerful processing nodes that contain processors and memory [LW95]. Each node has a *cache-coherence controller* that enables data replication coherently, e.g., when a cache line is updated, the CCC invalidates other copies, and ensures that a processor request always gets a copy of the most recent data. CC-NUMA multiprocessors provide the convenience of memory sharing with one global address space, some portion of which is found in each node. CC-NUMA has a good balance between programming complexity and hardware complexity. Several vendors are adopting it for building new high-end servers, e.g., HP/Convex SPP2000 [BA97], Sequent NUMA-Q [LC96], and SGI Origin 2000 [LL97].

In an effort to identify strengths and weaknesses in current approaches, we evaluate three important CC-NUMA systems: (i) Stanford DASH, a prototype of the first project to demonstrate a scalable shared-memory system, (ii) Convex SPP1000, the first commercially available CC-NUMA system, and (iii) SGI Origin 2000 which represents today's state-of-the-art technology. Although the three systems share many similarities, they have significant differences that translate into large performance differences. They all cluster processors and memory into nodes that are interconnected by a low latency interconnection network. Each node has multiple processors, memory, and coherence controller. The three systems differ in many respects; for example, in the number of processors per node, processor cache configuration, memory consistency model, location of memory in the node, and cache-coherence protocol. Sections 6.2, 6.3, and 6.4 present overviews of the three systems.

In this study, we use two representative applications from Convex's implementation of the NAS Parallel Benchmarks. For a fair system comparison, we undid the SPP1600-specific optimizations in these benchmarks. The two applications, CG and SP, have interesting differences and are further characterized in Chapter 3. CG performs simple reduction operations on long vectors; its high sharing degree communication provides for updating and reading the partial products. SP performs computations on three-dimensional matrices, with a producer-consumer communication on the boundaries of matrix partitions. The data presented here is from the main parallel phase when solving the "A" problem size on 16 processors.

Section 6.5 presents a raw comparison where CDAT analyzes the performance of the two benchmarks on models that are like the three systems with their original parameters. It shows that the performance of the three systems spans a wide range because their components vary widely in size and speed, as these systems were introduced at different times, with the then prevailing technology. Section 6.6 presents a normalized comparison where the systems are put on the same technological level with similar components. We believe that the normalized comparison better exposes the performance differences due to system organization and cache-coherence protocol, rather than the underlying technology and component sizes. Some conclusions are drawn in Section 6.7.

6.2 Stanford DASH

The Stanford DASH project started in the fall of 1988, and the first prototype was completed in the spring of 1991 [LLG+92]. The DASH is based on the SGI POWER Station 4D/340 which has 4 processor boards connected by a cache-coherent bus. As shown in Figure 6.1, the memory and the I/O interface are also connected to the shared bus. Two boards were added to handle remote memory access and global cache coherence.

Each processor board includes a 33 MHz MIPS R3000 processor with 64-KB instruction cache, 64-KB first-level data cache, and 256-KB secondary data cache, which is a write-back snooping cache that interfaces with the shared node bus. The cache line size is 16 bytes. The shared bus operates at 16 MHz and supports the Illinois MESI cache-coherence protocol [PP84], where the highest priority processor with a valid copy supplies data on bus requests. This is different from the MESI protocols supported by modern processors where a processor only supplies data when it has a modified copy [SS86]. The system interconnection is through two 2-D meshes: one for requests and another for replies. This dual arrangement simplifies protocol deadlock elimination. The network links are 16 bits wide with a maximum transfer rate of 60 MB/s per link.

The CCC functionality is implemented using two boards: the directory controller (DC) and the

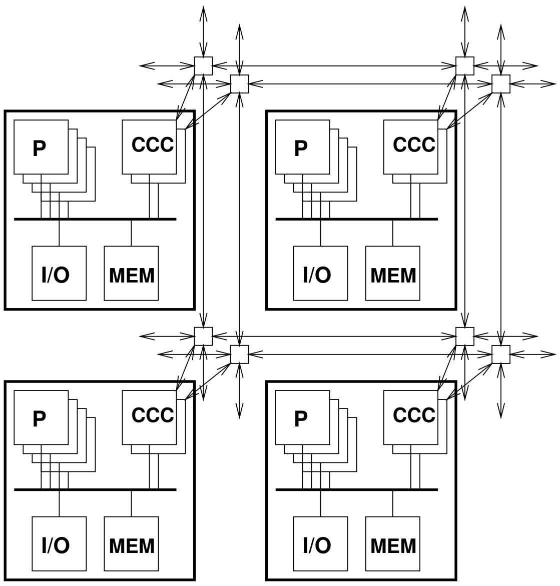


Figure 6.1: Four-node DASH system.

reply controller (RC). The DC generates node requests, and contains the directory of the global sharing state of the local portion of the memory and the x dimension network interface. The RC handles incoming requests and replies, and contains the y dimension network interface and a 128-KB *interconnect cache*. The ICC tracks outstanding requests made by the local processors and caches remote data.

Cache coherence is maintained by a combination of snoopy and directory protocols. The way in which a cache miss is satisfied depends on the home node of the missed line. When a processor generates a bus request for a local line, the other processors snoop this request and check their caches; the highest priority processor that has a valid copy supplies the line. At the same time, the DC accesses the directory to find the global sharing status of the line. If there is a remote node that has a modified copy, the DC blocks the request and generates a recall to the remote node; the processor is unblocked when the line is retrieved. In case the line is not cached locally and not modified in a remote node, the memory supplies the data.

When a processor generates a bus request for a remote line, the line is supplied by a local cache if there is a valid copy locally. Otherwise, the DC generates a request to the home node. When the RC of the home node receives the request, it echoes it on its local bus. A cache or the memory supplies the line unless it is modified in a third node. In the latter case, as shown in Figure 6.2, the home node (H) generates a recall signal (2) to the dirty node (D). When the dirty node receives the recall signal, its RC echos the recall on the local bus, the dirty cache supplies the line, and its DC forwards the line to the local node (3a), where it is inserted in the ICC. The ICC supplies the line when the processor retries its request. Additionally, the dirty node updates the home node (3b).

The *critical miss time* is from the start of the first processor request to the end of the reply that

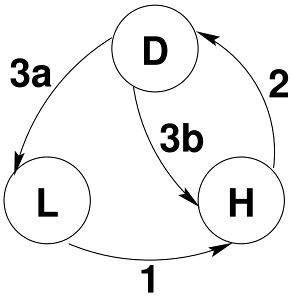


Figure 6.2: Satisfying a cache miss to a line with home node H that is dirty in node D.

satisfies the processor miss. Thus, for this example, only the time of signal (3b) is not critical.

6.3 Convex SPP1000

The SPP1000 was introduced in 1994 and consists of 1 to 16 nodes that communicate via four rings [Bre95]. As shown in Figure 6.3, each node contains four functional blocks interconnected by a crossbar (XBR). Each functional block interfaces with one ring and contains two 100 MHz HP PA 7100 processors and two memory banks. Each processor has 1-MB instruction and data caches. The processor pair share one agent to communicate with the rest of the machine. The memory has a configurable logical section that serves as an *interconnect cache*. The ICC is used for holding copies of shared data that are referenced by the local processors, but have a remote home.

Successive 64-byte lines are interleaved across the memory banks. A processor cache line is 32 bytes, and thus holds half a memory line. Each CCC is responsible for $1/4$ of the memory space that a processor can address. When a processor has a cache miss, its agent generates a request through the crossbar to one of the four local CCCs. The CCC first accesses its memory (the ICC section for remote lines). If the attached memory does not have a valid copy, the CCC either contacts

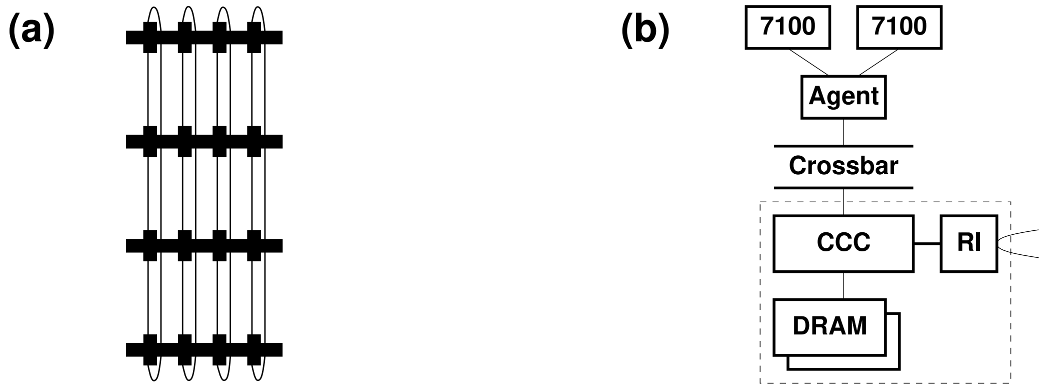


Figure 6.3: (a) Four SPP1000 nodes interconnected by 4 rings; each node has 4 functional blocks. (b) The functional block has 2 processors and 2 DRAM memory banks, and interfaces with the local crossbar and one ring.

a local agent if any of the agent's processors has the valid copy in its cache, or contacts a remote CCC for service through its ring.

Each ring is a pair of unidirectional links with a peak bandwidth of 600 MB/s for each link. The rings support the Scalable Coherent Interface (SCI) standard [SCI93]; which coherently transfer one 64-byte line in response to a global shared memory access.

The coherence data is maintained in tags associated with the 64-byte memory lines. Each tag holds the local and global sharing state of the respective line. The local state part includes the local caching status of each of the two 32-byte halves of the line. The intra-node coherence protocol is a three-state MSI protocol [ABP94]. The global sharing state is arranged in a doubly linked list distributed directory rooted in the home node. The tag in the home node contains the global status and a pointer to the head of the sharing list. Each ICC tag in other sharing nodes contains the caching status of the line and pointers to the previous and next nodes in the list.

6.4 SGI Origin 2000

The SGI Origin 2000 [LL97] was introduced in 1996 and connects dual-processor nodes with an interconnect that is based on the SPIDER router chip [Gal96]. The CCC has multiple paths to interconnect the processor pair, four memory banks with attached directory, the I/O interface, and the interconnect network (Figure 6.4). Each processor runs at 195 MHz and has 32-KB on-chip instruction and data caches and a 4-MB, 2-way associative, combined secondary cache. The secondary cache line size is 128 bytes.

The bandwidths supported by the processor bus, the CCC, and the network are all matched and equal 780 MB/s. This perfect matching reduces latency in no-contention situations by allowing signals to be streamed directly between components; the bandwidth match eliminates the need for buffering.

The processor supports the MESI cache-coherence protocol, but does not snoop bus requests of other processors. Therefore, the Origin 2000 relies purely on a directory-based protocol and point-to-point signals for cache coherence. The directory is implemented in the memory and contains a sharing vector and a status field for each 128-byte line. The directory is accessed in parallel with memory access.

As shown in Figure 6.5, two nodes share one router. Each router has 6 bidirectional ports; two ports interface with the two nodes, and the other 4 ports can be used to interface with other routers. The figure shows eight routers—each using five of its ports—interconnecting 16 nodes in a cube configuration.

The Origin 2000 supports the sequential consistency model [Lam79] which supports a wide range of applications, but with minimal opportunities for hiding the latency of cache misses. For

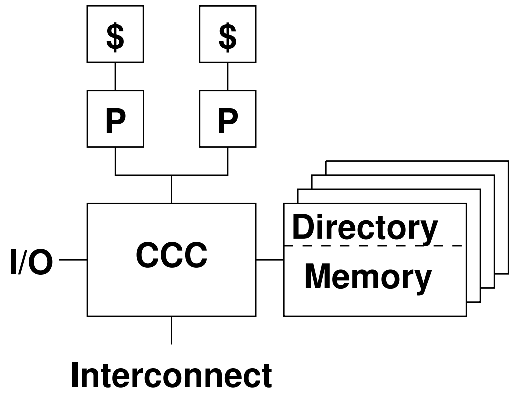


Figure 6.4: The Origin 2000 node contains two processors, 4 to 32 memory banks with attached directory, and cache coherence controller. It interfaces with the I/O devices and the interconnection network.

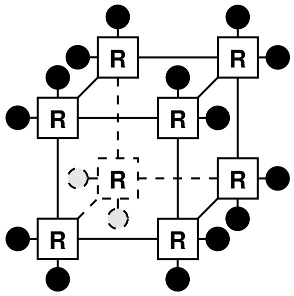


Figure 6.5: Eight routers are used to interconnect sixteen Origin 2000 nodes in a cube configuration.

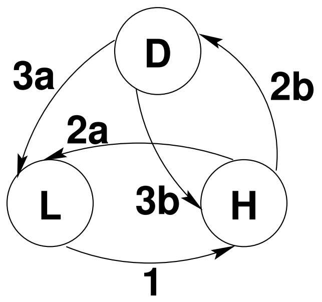


Figure 6.6: The Origin 2000's speculative memory operation (2a) is used in satisfying a cache miss for a line which home node is (H) and is owned by node (O).

this reason, the Origin 2000 uses many protocol optimizations to reduce the local and remote latency. Figure 6.6 shows an example where the local node (L) has a load miss for a line which has a remote home in node (H) and is owned by node (O). When the home node receives the miss request, it accesses its directory and memory. Although the directory indicates that node (O) owns the line, the home node speculatively sends its line to the local node (2a) and generates a recall signal (2b). The local node saves the line until it receives a response from node (O). If node (O) has a modified copy, it is sent to the local node (3a) and the home is updated (3b). In this case, the saved speculative line is discarded. But if node (O) has a clean copy, it sends short negative replies to the local and home nodes, and the speculative line is used by the requesting processor.

6.5 Raw Comparison

This section presents the raw comparison where we use CDAT configuration files that select components of the same size and speed as those used in the three case-study systems. The first touch memory allocation policy is used throughout; a memory page is allocated in the node where the first reference to this page is made. Parameterizing the signal latencies part of the configuration files was relatively difficult because this information is not directly available from the literature that describes these systems. Nevertheless, based on analyzing the available information and some system calibration experiments via microbenchmarking, we believe that the latencies used represent good approximations of the respective systems.

Parameterizing the configuration file of the DASH was the easiest because there are ample detailed DASH publications, e.g. [LW95]. Parameterizing the Origin 2000 configuration file is based on the information available from SGI home page [O2K96, R1097]. As there is little publicly-available information about the SPP1000 latencies from Convex, we used the results of our calibration experiments [AD98b]. Table 6.1 summarizes the latencies used. The CDAT configuration files used in this study are listed in Appendix A.

We refer to the three systems evaluated in this section as DASH, SPP, and Origin. With these signal latencies, the misses satisfied from local and remote memories in the three systems are: 876 and 3,064; 564 and 3,964; and 290 and 610 nanosec, [1] respectively.

Table 6.2 summarizes the main configuration differences among the three systems. The following subsections analyze the cache misses.

6.5.1 Miss Ratio

There are four main reasons for a processor to generate a bus request (other than prefetch operations which are not found in CG and SP): (i) **Fetch** miss when there is an instruction code miss. (ii) **Load** miss when a load instruction misses in the secondary cache. Usually, the processor gets a shared copy (S) of the line. However, the MESI protocol gives the processor an exclusive copy (E) when the line is not cached in other processors. (iii) **Store** miss when a store instruction misses in the secondary cache. (iv) **Store/S** hit on a line in the shared state. The processor requests exclusive ownership of the line, and as for the store miss, the final state is modified (M).

[1] A recent study reports higher Origin 2000 latencies [HLK97].

Signal	DASH	SPP	Origin
From PRO to BUS	375/313	70/120	100
From BUS to CCC	250	-	20
From BUS to MEM	0/188	-	-
From BUS to XBR	-	32/128	-
From XBR to CCC	-	32/128	-
From CCC to MEM	-	0/140	20
From CCC to NET	375	59/85/165/192	150
From MEM to CCC	-	172	80
From MEM to BUS	313	-	-
From NET to CCC	125/250	1432	10
From CCC to XBR	-	32/128	-
From CCC to BUS	125/376	-	20
From CCC to ICC	188	-	-
From XBR to BUS	-	30/90	-
From ICC to BUS	250	-	-
From BUS to PRO	0/188	40	20/50

Table 6.1: Signal latencies in nanoseconds. Multiple values are given when the latency depends on the signal length (see Subsection 6.6.5), latter values are for longer signals. "-" indicates that the signal is not relevant to the corresponding system.

Feature	DASH	SPP	Origin
Processors per node	4	8	2
Secondary cache size	1/4 MB	1 MB	4 MB
Secondary cache associativity	1	1	2
line size (bytes)	16	32/64	128
Interconnect cache size	1/8 MB	16 MB	0
Processor coherence protocol	Illinois	MSI	MESI
Memory consistency model	Relaxed	Weak	Sequential
Exclusive-clean remote state	No	No	Yes

Table 6.2: Summary of differences among the three CC-NUMA systems.

Figure 6.7a shows the data miss ratio in the three systems due to the three reasons for generating bus data requests. The miss ratio is the number of data misses to the number of memory instructions.

The three systems have widely different miss ratios, especially with CG. DASH has the worst miss ratio followed by SPP. The large miss ratios of DASH and SPP are mainly due to capacity misses as the data working set is larger than the processor cache [AD98c]. Although CG's working set does not fit in the processor cache of SPP or DASH, SPP's miss ratio is smaller due to its wider cache. The large, wide, set-associative cache used in Origin succeeds in eliminating most of the capacity and conflict misses; thus it has a miss ratio that is less than 0.5%. The misses in the Origin are mainly coherence misses due to communication.

With CG, most of the misses are load misses due to CG's high percentage of load instructions used to perform the reduction operations. With SP, a large percentage of the misses is due to the producer-consumer communication; store/S miss produces and load miss consumes.

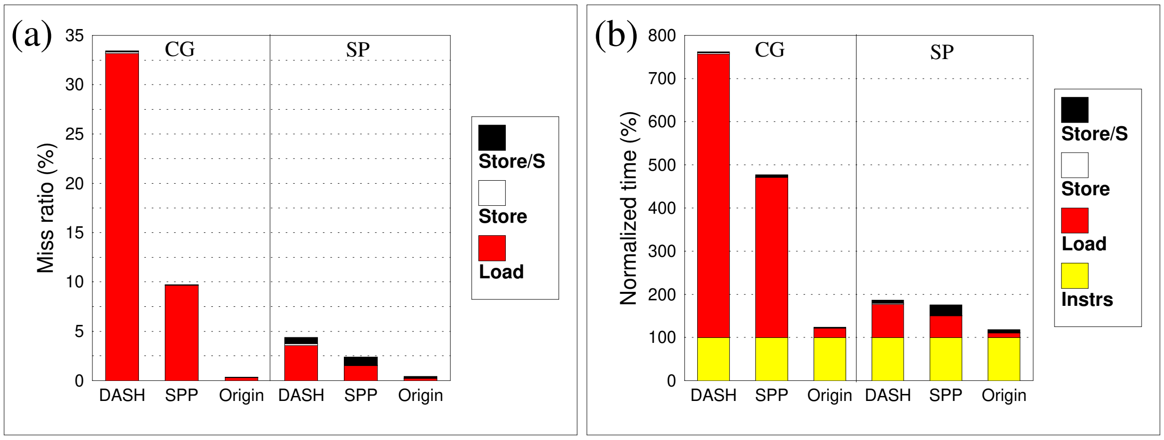


Figure 6.7: (a) The ratio of data misses to memory instructions. (b) Critical miss time relative to the instruction execution time.

To illustrate the effect of the miss ratio on performance, Figure 6.7b shows the critical time spent satisfying misses relative to the instruction execution time, which is normalized to 100%. Notice that DASH with SP spends a relatively small time in satisfying its store/S misses. This is because DASH uses the relaxed memory consistency model [GGH91] that enables it to hide most of the remote time spent to satisfy this type of miss.

6.5.2 Processor Requests and Returns

Before going further, let us look closer at the processor/bus interaction in the three systems. Figure 6.8 shows the percentage of the four processor request signals described in Subsection 6.5.1, processor write-back (w/b) signals, and supply signals.

The percentage of fetch misses is negligible because these applications have small code size that fits in the instruction cache. In Origin, where the misses are mainly coherence misses, the data produced by a processor (visible as store/S) is communicated to other processors through an equivalent number of supply signals.

With CG, DASH has a relatively high supply percentage due to Illinois protocol that allows processors to get shared data from the cache of neighbor processors.

With SP, which has a smaller sharing degree than CG, there is a larger ratio of store/S to load misses. Coherence misses in Origin are visible through supply signals, and capacity misses in DASH and SPP are visible through w/b signals.

6.5.3 Local and Remote Communication

This subsection presents an analysis of the misses according to the place where they are satisfied; a local miss is a miss satisfied from a local cache or memory, and a remote miss is a miss satisfied

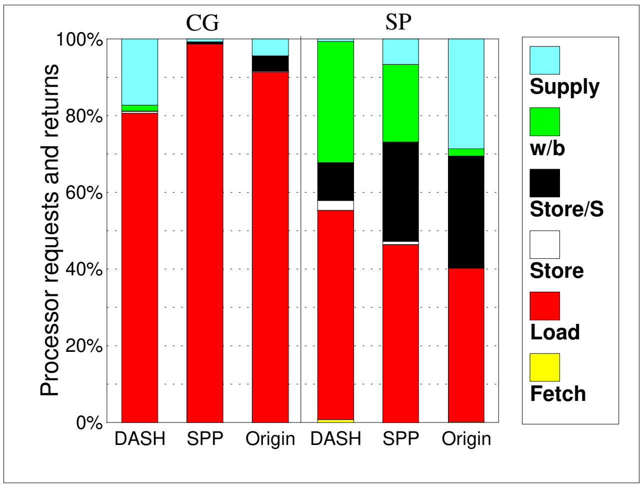


Figure 6.8: Processor/bus interactions, percentage of processor requests and returns.

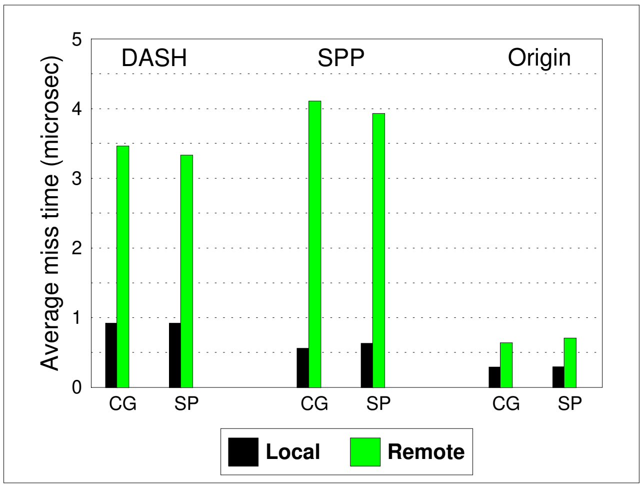


Figure 6.9: The average miss time of local and remote misses.

from a remote cache or memory. Figure 6.9 shows the average miss time for the local and remote misses. In all cases, the average remote time is larger than twice the average local time. The NUMA factor is calculated as the ratio of the average remote time to the average local time. SPP has the highest NUMA factor (6.8), followed by DASH (3.7) and Origin (2.3).

In SPP, the local miss time with SP is larger than with CG because SP has a higher percentage of misses that are satisfied through processor supplies, which is more expensive than satisfying them from the memory. For DASH and SPP, the remote miss time with CG is higher than with SP. The first touch memory allocation policy with SP is successful in reducing some of the more expensive remote accesses like the three-hop communication. Origin shows the opposite trend because the remote read misses with SP are expensive as they are generally satisfied from processor caches, in which case the speculative lines are useless.

Figure 6.10a shows the percentage of local vs. remote misses for each system. Origin has the

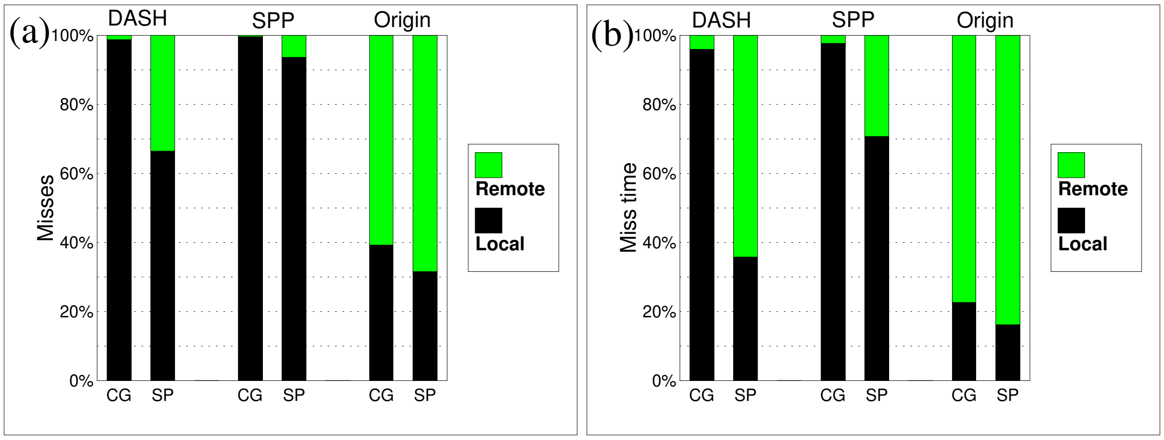


Figure 6.10: (a) Percentage of local vs. remote misses. (b) Breakdown of miss time.

highest percentage of remote misses due to its large number of nodes (8 nodes with 2 processors each). DASH and SPP are more successful than Origin in exploiting the high sharing degree of CG to reduce the remote miss percentage. In DASH, some of the load misses to remote data are satisfied from neighbor caches. In SPP, some of these misses are satisfied from the ICC of the local node. Figure 6.10b shows the percentage of time spent in satisfying local vs. remote misses. The high average remote miss time vs. local miss time makes the impact of remote misses exceed their ratio.

6.6 Normalized Comparison

In this section, we use CDAT configuration files that preserve the architectural and protocol differences, but put the three systems on the same technological level, i.e., same network speeds and same component sizes and speeds. The configuration files configure CDAT to simulate three derived systems: nDASH, nSPP, and nOrigin. They select modern component sizes and speeds similar to those used in Origin 2000 (Table 6.1). The three derived systems, unlike the Origin 2000, have 4 processors per node and use cache lines that are 64 bytes.

As in to the DASH node, the nDASH node has the memory on the local bus and its CCC interfaces with the bus, the network, and the ICC that participates in the local bus coherence protocol. nDASH uses an ICC that has a size and speed identical to the processor secondary cache. nSPP uses a 16-MB ICC that is a section of the node memory as in the SPP1000. However, nSPP node has four processors on one snoopy bus that is directly connected with the CCC, which interconnects the node bus, memory, and network interface. As in the SPP1600, nSPP allows caching local data in the exclusive clean state (MESI protocol).

In the following five subsections, we present analyses of the miss ratio, processor requests and returns, local and remote misses, where miss time is spent, and traffic exchanged between the system components.

6.6.1 Miss Ratio

Figure 6.11a shows the data miss ratio of the three systems. With CG, the three systems do not have noticeable miss differences and most of the misses are load misses. On the other hand, with SP, nSPP has a slightly higher load miss ratio due to conflicts in the ICC. When an ICC line is replaced, all the local copies of this line are invalidated; consequently, future processor accesses of this line generate misses. This behavior is not present in nDASH because its ICC does not maintain the inclusion property with respect to the locally cached remote lines. nOrigin has slightly fewer store/S misses than the other two systems because its global coherence protocol allows caching remote lines in the exclusive clean state.

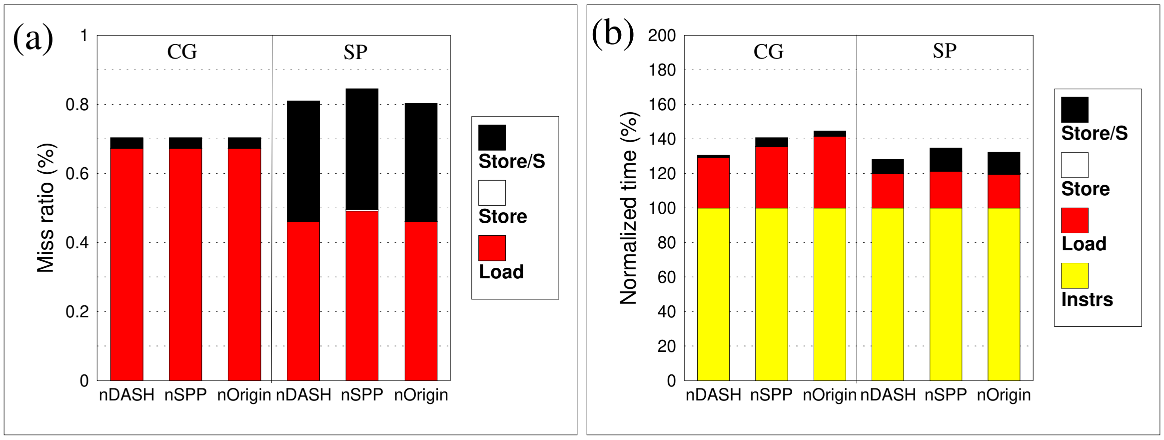


Figure 6.11: (a) The ratio of data misses to memory instructions. (b) Critical miss time relative to the instruction execution time.

Figure 6.11b shows the critical time spent satisfying misses relative to the instruction time. With CG, although the three systems have similar miss ratios, they spend different times satisfying these misses. nDASH manages to reduce load miss time by satisfying more misses through snooping from neighbor processors. nSPP also satisfies some misses locally from its ICCs; however, its load miss time is longer than that of nDASH due to its higher remote latency (see Subsection 6.6.3). Moreover, nDASH spends relatively less time in satisfying store/S misses due to its use of the relaxed memory consistency model. These misses are particularly expensive in nSPP with CG because they invalidate copies in multiple nodes, which the SCI protocol does serially.

6.6.2 Processor Requests and Returns

Figure 6.12 provides a closer look at the processor/bus interaction by showing the percentage of the four processor request signals described in Subsection 6.5.1, processor write-back (w/b) signals, and supply signals. As indicated by the small number of fetch and w/b signals, the three systems have negligible code and capacity misses; the secondary cache is large enough to contain the data

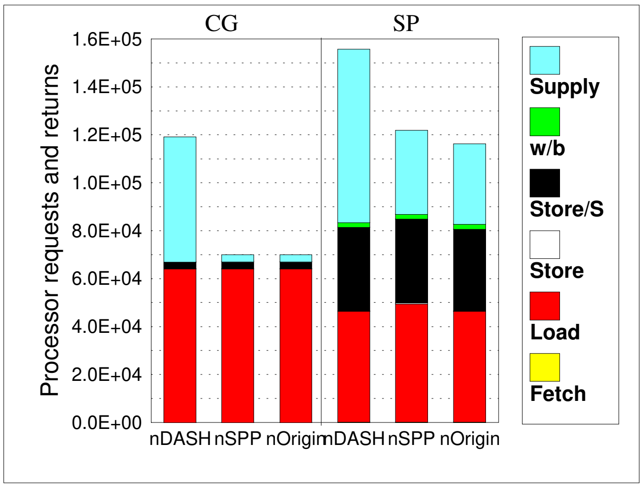


Figure 6.12: Processor/bus interactions; processor requests and returns.

and code working sets. However, as SP has more complex data structures than CG, it has more w/b signals due to conflict misses.

nDASH satisfies many misses from processor caches through supply signals. With SP, which has producer-consumer communication, almost all nDASH's misses are satisfied by supply signals. In nOrigin and nSPP, the data produced by a processor (visible as store/S) is communicated to other processors through an equivalent number of supply signals. With CG, which has larger sharing degree than SP, there is a larger ratio of load to store/S misses.

6.6.3 Local and Remote Communication

Figure 6.13 shows the average miss time of the local and remote misses. The average local miss latency in nDASH is the smallest because its memory is on the local bus, while in the other two systems the latency of traversing the CCC is added to the memory latency. The average local miss latency with SP is higher than with CG because SP has a higher percentage of misses with invalidation. This invalidation cost is more obvious in nSPP and nOrigin because of their use of stricter memory consistency models.

nOrigin's aggressive use of protocol optimizations and streaming of remote requests and responses pay off in achieving the lowest remote miss time. nDASH does not stream remote data to the processor; remote data is first inserted in the ICC, then the requesting processor gets it through snooping from the ICC. nSPP has the highest remote access latency because it checks the local ICC before generating a remote request, and its complicated global coherence protocol requires more network signals, often in a serial pattern, to satisfy remote misses.

In nSPP, the remote miss time with CG is larger than with SP because of CG's high sharing degree; the overhead of maintaining the directory increases with more sharers. In nOrigin, the remote miss time with SP is larger than for CG because SP has a higher percentage of remote load

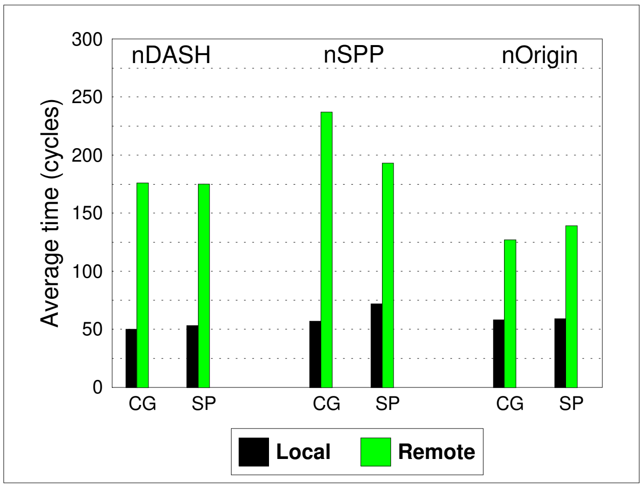


Figure 6.13: The average miss time of local and remote misses.

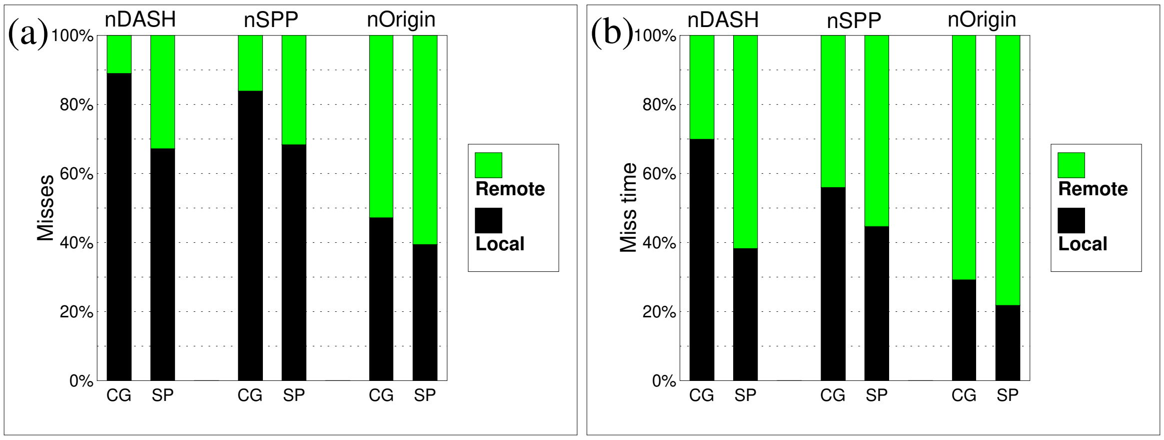


Figure 6.14: (a) Percentage of local vs. remote misses. (b) Breakdown of miss time.

misses that are satisfied from remote caches which is more expensive in nOrigin than satisfying them from remote memory.

Figure 6.14a shows the percentage of local vs. remote misses for each system. nDASH and nSPP have fewer remote misses than nOrigin because they satisfy more misses locally through cache-to-cache transfers or from the ICCs. Figure 6.14b shows the percentage of time spent in satisfying local vs. remote misses. Similar to the raw comparison, the high average remote miss time vs. local time here makes the impact of remote misses exceed their ratio.

6.6.4 Where Time Is Spent

Figure 6.15 shows a breakdown of the time spent in satisfying processor requests and returns in the various system components for CG and SP. The time spent in each component type is split into three parts: critical time spent in local components, critical time spent in remote components, and non-critical time spent in both local and remote components. The figure shows seven compo-

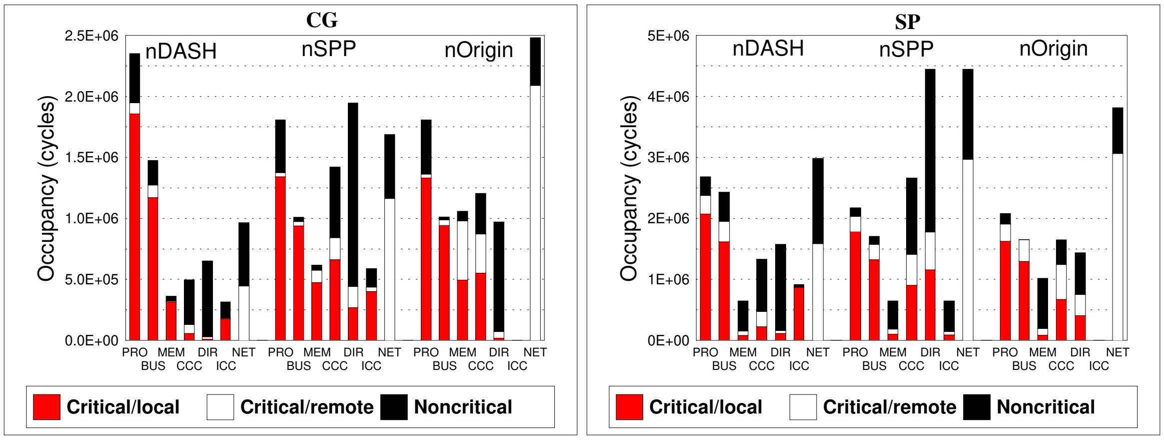


Figure 6.15: A breakdown of the time spent in satisfying processor requests and returns in the various system components.

nent types: processor, bus, memory, cache-coherence controller, directory, interconnect cache, and network.

There are large differences between the three systems. For example, in nDASH the memory is less frequently used than the other systems. The CCC in nDASH is also less involved, and when it is involved, most of the time spent in it is not critical (either because of the relaxed consistency model or because its time is overlapped with other bus activities).

Although most of the time spent accessing the directory in nSPP is not critical, it is the most involved component type in the system. This behavior is due to the SCI protocol which requires several directory accesses to maintain the distributed sharing linked-list. When we take into consideration the fact that nSPP implements both the directory and the ICC in the memory, it becomes clear that it has the most stressed memory subsystem.

nOrigin has a high network utilization because of its high percentage of remote misses. The network utilization in nSPP is also high because the SCI protocol uses more network signals to satisfy one miss.

The differences between CG and SP are due to their different communication patterns. The producer-consumer communication in SP exposes a higher percentage of the directory time as critical time due to the frequent need to find the node that owns the modified copy. SP also has relatively higher CCC and directory times than CG.

6.6.5 Data Transfer

Figure 6.16 shows the number of bytes transferred from one component type to another for CG and SP. When reading this figure, remember that a component only communicates with those components that are directly connected with it. CDAT differentiates between two signal types: short

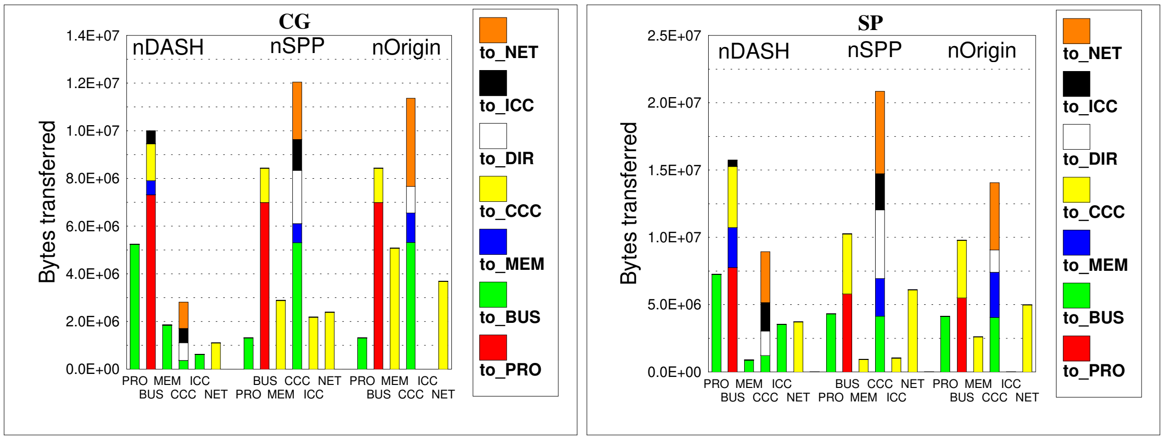


Figure 6.16: The number of bytes transferred from one component type to another.

signals that do not carry data, e.g., processor request; and long signals that carry one cache line, e.g. data response. In the three systems, the short signal is 16 bytes and the long signal is 80 bytes. However, nSPP's global coherence protocol has four signal types (24, 40, 88, and 104 bytes; CDAT does also model these four remote signal types). The shortest nSPP remote signal is 24 bytes. A signal becomes 16 bytes longer when it carries a node pointer, and 64 bytes longer when it carries data.

Relative to the other systems, nDASH's processors transfer more bytes to the bus as a result of the frequent local cache-to-cache transfers, and its busses carry higher traffic as they are the media of communication with the memory and the ICC. nDASH also transfers the least amount of data through the network due to its low percentage of remote misses. nSPP transfers more data over the network than might be expected from its remote miss percentage due to its protocol that uses more network signals per miss and larger remote signal sizes than the other protocols.

With SP, the percentage of bytes transferred to memory is larger than CG due to memory updates when processors supply dirty data.

6.7 Chapter Conclusions

The two benchmarks used in this study have different communication patterns. CG has repetitive producer-consumers communication with a large sharing degree, thus most of its cache misses are load misses. Whereas, SP has generally single consumer, and its load and store/S misses are more balanced.

The raw comparison shows that there are large performance differences among the three systems due to their different component sizes and speeds. Their cache miss ratios are significantly different, and they have different NUMA factors (the average remote miss time to the average local time), with the Origin 2000 at 2.3, the DASH at 3.7, and the SPP1000 at 6.8.

On the other hand, the Origin 2000 has the highest percentage of remote misses. This is due to its higher number of nodes and its lack of an ICC. The Origin 2000's use of the strict sequential consistency model results in the least ability to hide miss latency.

In the normalized study, where we use same processor cache for the three systems, there are only small differences in the cache misses. Among the three normalized systems, the normalized SPP1000, nSPP, has slightly more cache misses due to conflicts in its ICCs. The normalized nOrigin has slightly fewer store/S misses due to allowing the exclusive state for caching remote lines.

nOrigin spends the longest time satisfying its misses because the majority of its misses are satisfied remotely. nDASH's Illinois protocol and nSPP's ICC reduce the percentage of remote misses. However, nDASH has less miss time than nSPP because its average remote time is smaller and its relaxed memory consistency model enables it to hide more of the miss time. nSPP has the highest average remote access time because it uses the SCI coherence protocol which requires more signals to satisfy a miss than the protocols used in the other two systems. The aggressive protocol optimizations used in nOrigin and its ability to stream remote requests and responses give it the least average remote access time.

Furthermore, nDASH has the smallest average local time because the memory, unlike other systems, is closer to the processor. nDASH memory is less involved in satisfying misses because nDASH satisfies more misses through cache-to-cache transfers.

The use of the complex SCI protocol in nSPP results in a large CCC involvement in satisfying misses and in its frequent directory accesses. nSPP also has the most stressed memory. Convex addresses this problem by having 4 CCCs in each SPP1000 node where each CCC has direct access to two memory banks. Although the nOrigin's large remote miss percentage results in heavy network utilization, the complex protocol in nSPP sometimes generates more network traffic for fewer remote misses.

In summary, nDASH outperforms the other normalized systems because of its coherence and consistency protocols, but it uses a bus coherence protocol that is not supported by modern processors. nSPP avoids some of its high remote access time by satisfying some misses from its ICCs. nOrigin, using the sequential consistency model, trades exposing more miss time for supporting a broader range of applications, and trades large remote miss percentage for small average remote time. It might be rewarding to look for new approaches that reduce the remote miss percentage and maintain a small average remote time. One such approach that uses a snoopy interconnect cache is investigated in Chapter 7.

CHAPTER 7

REDUCING COMMUNICATION COSTS OF FUTURE SYSTEMS WITH CCAT EVALUATIONS

This chapter evaluates three techniques to reduce the communication cost in Origin 2000-like systems. It evaluates four systems derived from the Origin 2000 architecture, and shows that one proposed technique, which uses a snoopy cache for remote data, significantly reduces the communication cost, and is most beneficial when the communication cost is high.

7.1 Introduction

In Chapter 6, we have shown that the Origin 2000 is effective in achieving low remote latency. However, it has a high ratio of remote to local misses. Consequently, its average latency and internode traffic are high. In this chapter, we will evaluate three techniques that attempt to reduce the remote misses due to communication without adversely affecting the local and remote latencies. The main causes of the Origin 2000's frequent remote misses are:

1. Each node contains only two processors, thus the shared data is often spread over a large number of nodes.
2. The processor does not snoop the requests of the other local processor, thus remote traffic is generated even when one processor can satisfy its neighbor's request.
3. No interconnect cache is used for caching remote data that is referenced by the local processor, thus remote traffic is generated whenever a processor requests a remote line, even when this line has recently been requested by the other local processor.

However, there are no magic solutions for these weaknesses. Although increasing the number of processors per node would increase the possibility of finding shared data in the local memory, the local bus will be more heavily utilized and contention may increase the memory latency; allowing processors to snoop the requests of other processors decreases remote traffic, but the snooping overhead may affect the processor performance; and incorporating an interconnect cache, as in the SPP1000, may increase the remote latency.

In this chapter, we evaluate four Origin 2000-like systems that address these design trade-offs using CCAT simulations of six applications. Section 7.2 reviews the communication cost in these systems; Section 7.3 expands on these design trade-offs and introduces the four evaluated systems; Section 7.4 describes the experimental setup and the applications used; Section 7.5 provides a preliminary evaluation to investigate the design spaces of cache size, number of processors, number of memory banks, and processor model; Section 7.6 evaluates the execution time and traffic of the four systems; and Section 7.7 presents the conclusions of this study.

7.2 Communication Cost

In this section, we review communication patterns in CC-NUMA systems to identify their costs and explore opportunities for reducing these costs. This review discusses these issues in the context of a coherence protocol similar to the one used in the Origin 2000. The level of detail of the protocol description in this chapter is sufficient for our purposes; a more detailed description of this protocol is presented in [CSG98].

In CC-NUMA systems, processors communicate through accessing shared data. The coherence protocol ensures that when a processor accesses a shared memory location, it will get the most up-to-date data value. In order to exploit spatial locality, the coherence protocol usually transfers one complete line whenever a cache miss occurs. True communication occurs when multiple processors access a shared location; false communication occurs when multiple processors access different locations that happen to be in one line.

Communication occurs in various patterns, and its cost is usually high when it is between nodes. Remote communication is possible for lines that have local home as well as those with remote homes.

The main remote communication patterns when accessing *local* lines are:

RAW: Load miss for a line that is dirty in some remote node. The local directory is checked, which points to the dirty node. The local coherence controller then generates a recall signal to the dirty node. The dirty node then replies with a data response to the requesting processor. The recall and data response signals are remote signals (the arcs in Figure 7.1a).

WAR: Store miss for a line that is shared in one or more remote nodes. The local directory is checked, which points to the reader nodes. The local coherence controller sends a copy to the requester (if the requester does not already have one), and generates invalidation signals to the readers. After receiving acknowledgments for the invalidation signals, the requester performs the store (Figure 7.1b).

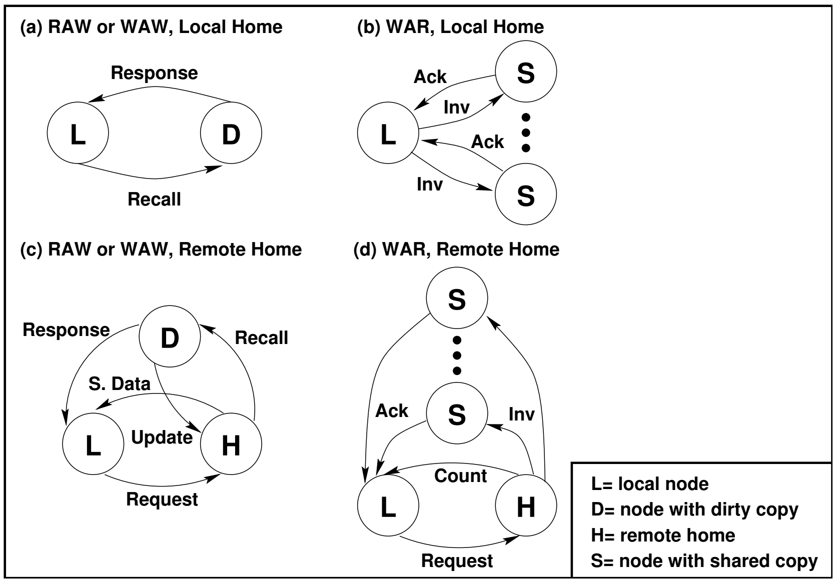


Figure 7.1: Main remote communication patterns.

WAW: Store miss for a line that is dirty in a remote node. This is similar to RAW, but the dirty node invalidates its copy after supplying the data response.

Although these patterns are expensive relative to the respective patterns where all the processors involved are in the same node, these remote signals are unavoidable. The best approach to address their cost is to decrease the directory access and remote latencies. However, weaker consistency models address WAR by allowing the requester to perform its store without waiting for the acknowledgment signals [LLG$^+$92].

The main remote communication patterns with *remote* lines are:

RAW: Load miss for a dirty line. The local coherence controller generates a request signal to the home node. The directory in the home node is then checked, which points to the dirty node. The home node's coherence controller then generates a recall signal to the dirty node, and sends a speculative data reply to the requester. The dirty node then responds to the requester and updates the home node. Each arc in Figure 7.1c is shown as a remote signal. However, when the processor that has the dirty copy is in the local or home node, the response or the recall and update signals, respectively, are local. The speculative data value is used only when the dirty node has the line in the exclusive clean state (modern processors usually do not supply clean lines) and the requester receives a negative response signal from the dirty node. An alternative protocol that does not use this speculative signal has a higher latency; the requester receives the line after four serial signals instead of three: request, recall, negative response from the dirty node to the home node, and data response signal from the home node to the requester.

WAR: Store miss for a shared line. The local coherence controller generates a request to the home node. The directory is then checked, which points to the reader nodes. The home node's coherence controller then sends the readers count to the requester (with the data if the requester does not have a copy), and generates an invalidation signal to each reader. After receiving an acknowledgment from each invalidated node, the requester then performs the store (Figure 7.1d). When a reader processor is in the local or home node, the acknowledgment or invalidation signal, respectively, is local.

WAW: Store miss for a dirty line. This is similar to RAW, but the dirty node invalidates its copy after supplying the data response, and sends a completion signal (rather than an update) to the home node.

The communication patterns with remote lines generate more system traffic and usually have higher latencies than communication with local lines. However, for communication with remote lines, there are many techniques to avoid some of these remote signals and consequently reduce the remote communication cost. The following section describes some of these solution techniques.

7.3 Design Issues and Solutions

In this section, we describe a base system similar to the Origin 2000, and three derived systems that incorporate possible design solutions that attempt to reduce remote communication cost. The following four subsections describe these four systems and qualitatively discuss their strengths and weaknesses.

7.3.1 Base System with Directory-Based Coherence

The base system is the closest of these systems to the Origin 2000 itself. It connects dual-processor nodes with an interconnect that is based on a six-port router. As shown in Figure 7.2, each node has a *cache coherence controller*. The CCC has multiple internal paths to interconnect the processor pair, four memory banks (with attached directory), the I/O devices, and the interconnection network. The processors run at 200 MHz and support the MESI cache-coherence protocol [SS86]. Each processor has a 4-MB, 2-way associative, combined secondary cache. The data cache line size is 128 bytes.

The base system is sequentially consistent [Lam79], and maintains cache coherence by using a directory-based protocol. The directory is implemented in the memory and contains a sharing vector and a status field for each 128-byte line. The directory is accessed in parallel with each memory access.

As shown in Figure 7.3, two nodes share one router. Each router has 6 bidirectional ports; two ports interface with the two nodes, and the other 4 ports can be used to interface with other routers.

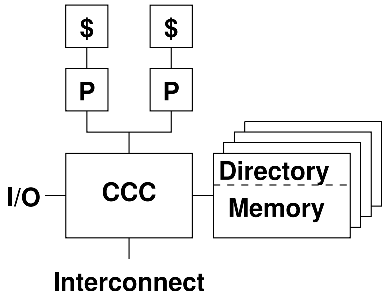


Figure 7.2: A node of the base system contains two processors, four memory banks with attached directory, and a cache coherence controller that connects the processors with the memories and interfaces with the I/O devices and the interconnection network.

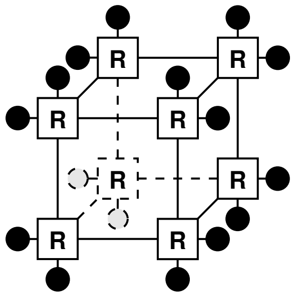


Figure 7.3: Eight routers are used to interconnect sixteen nodes in a cube configuration.

The figure shows eight routers—each utilizing five of its ports—interconnecting 16 nodes in a cube configuration.

The processor pair share one split-transaction bus to communicate with the rest of the system. This bus has two modes of operation, snoopy and point-to-point [R1097]. In the snoopy mode, each processor observes the bus requests of other processors and checks its cache to ensure cache coherence. For example, when the processor snoops a load or store miss request generated by another processor for a line that is dirty in its cache, it supplies a copy of this line; it also invalidates its copy of a cached line when it snoops a store miss request for this line. In the point-to-point mode, a processor does not snoop the bus transactions of other processors. Thus, the memory latency can be reduced by allowing the CCC to forward a processor request to the memory without the need to wait for the snoop result. Additionally, the processors do not lose some of their cache bandwidth to check snooped requests.

The base system uses the bus in the point-to-point mode and relies purely on the directory-based protocol for cache coherence. We refer to this base system as **D2**; the **D** stands for pure directory scheme, and the **2** stands for the number of processors per node.

7.3.2 Intranode Snoopy-Based Coherence

The second system, called **S2**, is identical to **D2**, but uses the node bus in the snoopy mode. **S2** has an advantage over **D2** in the following communication situations:

- In RAW and WAW, when the processor that has the dirty copy is in the same node as the requester, the request is satisfied locally by a supply from the local processor over their shared bus. However, an update data signal is sent to the home memory in RAW.
- In WAR, when there is a local reader, the CCC does not send an invalidation signal to this reader. In the case when the only reader is local, the CCC directly gives the requester the ownership without any invalidation or acknowledgment signals.

Intuitively, we expect **S2** to have better performance than **D2** with applications that have enough local communication to overcome the added overhead of snooping.

7.3.3 Allowing Caches to Supply Clean Data

The Illinois bus cache coherence protocol is similar to the MESI protocol used in **D2**, with one main difference. Whereas, in the standard MESI protocol, a processor only supplies data when the requested line is dirty in its cache, the Illinois protocol allows a processor to supply the line whenever it has a copy. When multiple processors have copies, the Illinois protocol simply selects one of them as the supplier.

Modern processors do not support the Illinois protocol for two reasons: (i) There is no significant latency advantage for satisfying a miss request from a processor cache rather than the memory in modern symmetric multiprocessors. (ii) Modern secondary caches do not have a port that communicates directly with the bus; all cache transactions go through the processor that is situated between the secondary cache and the bus (see Figure 7.2). Thus, the frequent supplies in the Illinois protocol do deprive the processor of some of its cache bandwidth.

However, in a CC-NUMA system, there could still be an advantage to using the Illinois protocol because it avoids some of the expensive remote communications. The third system, **I2**, is similar to **S2**, but uses the Illinois protocol to maintain cache coherence among the processors within a node. Directory-based protocol is still used to maintain coherence among nodes. In addition to the snoopy advantages outlined in Subsection 7.3.2, **I2** reduces communication cost further in the following situations:

- In RAW, when there are multiple readers in the same node and one reader gets a copy, subsequent readers may get the line locally by a supply from the first reader, without generating remote traffic.

- In WAR, when the reader is local and has an exclusive copy, the writer gets a copy and the ownership of the line locally without generating remote traffic.

7.3.4 A Snoopy Interconnect Cache for Remote Data

Caching remote data in local specialized caches is a popular approach used mainly to reduce the cost of capacity misses. The fourth system , **IC2**, is similar to **S2** and uses one SRAM interconnect cache (IC) per node to reduce the cost of remote communication (Figure 7.4). In this chapter, we evaluate two IC implementations:

(**C**) The IC is implemented using standard SRAM modules similar to the modules used to implement each processor's secondary cache. Therefore, the IC can be as large as the processor's secondary cache.

(**c**) The IC is implemented in the CCC chip and is limited in size. This implementation may be the only feasible implementation if the CCC is pin limited.

We assume that both implementations have the same speed, and that the only difference in their effect on system performance is due to their difference in size. In the following discussions, we refer to both implementations as system **IC**. However, when we make a statement that applies to only one of the two implementations, we use references **C** and **c**.

The IC snoops the bus, checks its cached lines, and provides the snoop check result to the CCC with a latency not longer than the latency of the processor's snoop response. Thus, it does not delay the CCC in forwarding a needed request to the memory when there is an IC miss. We assume that, in case of an IC hit, the IC supplies the line after a latency similar to the supply latency from one of the local processors.

Remote lines are inserted in the IC in two cases: (i) when a remote line arrives at the CCC to satisfy a load miss, and (ii) when a local processor supplies a remote line to satisfy a load miss of another local processor. The IC supplies a line to satisfy local requests whenever it has a copy and

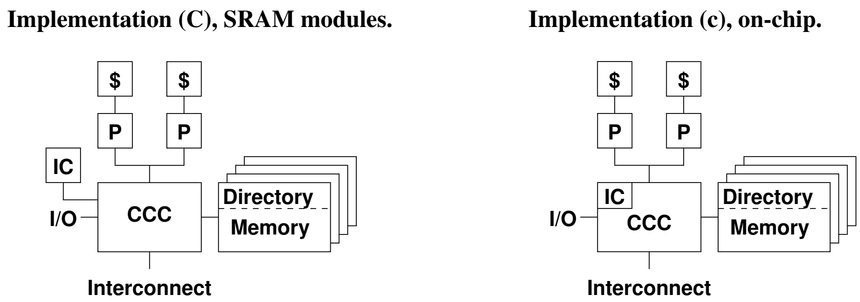


Figure 7.4: Two implementations of cache coherence controller supplemented with a snoopy interconnect cache for remote lines.

no local processor has a dirty copy. It also supplies dirty lines in response to external recall signals whenever no local processor has a dirty copy. A line is marked dirty in the IC when it is inserted as the result of a local supply; in this case, the IC becomes the owner. The node does not lose ownership of this line, and consequently does not update the home memory.

For remote lines, system **IC2**, like **I2**, reduces RAW cost of multiple local readers, and reduces WAR cost when a local processor has the exclusive copy. However, unlike **I2**, system **IC2** is compatible with the cache coherence protocols supported by modern processors. System **IC2** also reduces the system traffic by eliminating the need to update the memory in local RAW because the supplied copy is saved in the local IC. Additionally, since the node does not lose line ownership in RAW, there is no need to generate remote traffic if the reader later stores into the line.

7.4 Experimental Setup

We use CCAT to evaluate the performance of these four systems using six applications. CCAT keeps track of the state of the secondary cache, interconnect cache, and directory, and fully implements the cache coherence protocol to satisfy cache misses. In order to have a realistic experimental setup, the model parameters used in this evaluation, unless otherwise noted, are set to be similar to the Origin 2000 parameters.

As described in Section 2.6, CCAT is a system-level simulator with a processor model that captures some modern processor features that affect system traffic rates. We use two settings of this processor model:

Conservative The processor executes one instruction per cycle (when not stalled), has two buffer entries for outstanding cache requests (one for tracking misses and one for write-backs), and stalls on cache misses. This setting operates as if every instruction is dependent on its predecessor, and this processor model thus generates conservative traffic rates.

Aggressive The processor executes two instructions per cycle (when not stalled), has four entries for outstanding cache requests, a 48-entry instruction *reorder buffer* (only 16 entries of this ROB can hold memory instructions), and tolerates and overlaps misses that are within the window of its ROB size. The processor stalls on code miss, outstanding miss buffer full, and ROB full. This setting assumes no data dependencies among the instructions in the ROB window, and this processor model thus generates higher traffic rates. The parameters used in this setting are identical to the parameters of the MIPS R10000 processor, which is used in the Origin 2000. In Subsection 7.5.4, we also experiment with more aggressive parameters (128-entry ROB, 64-entry memory ROB, and 8-entry outstanding miss buffer).

The node bus is a multiplexed, split-transaction 64-bit bus that runs at half the processor speed with 800-MB/s peak bandwidth. Thus, the bus takes 32 processor cycles to transfer a 128-byte line. Unlike the Origin 2000 bus, which allows only 8 outstanding requests, the bus model used does not limit the number of outstanding requests. Thus, this number is limited only by the number of outstanding misses that the bus processors allow (the aggressive processor model allows 4 outstanding misses per processor, hence with two processors per node there is also a maximum of 8 outstanding bus requests).

We do not model the processor overhead when the bus is used in the snoopy mode. However, we think that this overhead is negligible because the R10000 processor schedules checking its cache for snooping around its own cache accesses, and this checking is rare due to the high ratio of data response bus occupancy to request occupancy (17 to 1). For system **I2**, the processor performance is not reduced, as we assume that the cache bandwidth is increased to handle the extra supplies.

Contention on the buses, memory banks, and network links is modeled according to the occupancies of the various signal types (see Subsection 7.4.1). We assume that a signal arriving at any of these shared resources occupies it directly when the resource is free. When the resource is busy serving another signal, the arriving signal is queued in an unlimited FIFO queue. This is different from the Origin 2000 which has limited queuing and uses negative acknowledgments and retries in queue full situations.

An Origin 2000 feature, not supported in these models, is page migration.[1] In order to reduce remote capacity misses, the Origin 2000 monitors the number of misses per node for each memory page, and migrates the page to the node that exceeds some threshold number of misses. As described in Section 7.5, we evaluate these systems with applications that have few capacity misses relative to coherence misses.

7.4.1 CCAT Models

In order to understand the nature of the performance of the four systems, we have carried out many simulation experiments with a broad range of design parameters. Table 7.1 shows the options and the default values of the parameters used in these experiments. In the following sections, unless otherwise specified, the default values are used. For example, we have used 2 (the default), 4, and 8 processors per node to characterize the effect of node size on performance.

CCAT differentiates between two signal types: a short signal that does not carry data, e.g., a processor request; and a long signal that carries a 128-byte data line, e.g., a data response. Table 7.2 shows the two signal types' occupancies of the shared resources used with the four systems. In the interconnection network, a packet carrying a short signal is 16 bytes, and one carrying a long signal is 144 bytes.

[1]Although the Origin 2000 has hardware support for page migration, the current operating system does not support page migration.

Feature	Values	Default
System	**D**, **S**, **I**, **IC**	**D**
Number of processors	1, 2, 4, 8, 16, 32	32
Processor clock	200 MHz	200 MHz[a]
Processor cache size	64 KB, 256 KB, 1 MB, 4 MB	4 MB
Cache line size	128 bytes	128 bytes
Number of processors per node	2, 4, 8[b]	2
Number of memory banks per node	1, 2, 4, 8, 16, 32	4
Memory page size	4 KB	4 KB[c]
Memory allocation policy	Round Robin, First Touch	First Touch[d]
Interconnect cache size	0, 16 KB (**c**), 4 MB (**C**)	0

[a] The Origin 2000 runs at 195 MHz; 200 MHz clock is more convenient because it has a cycle with an integer number of nanoseconds.

[b] The Origin 2000 bus supports a maximum of 4 processors.

[c] The Origin 2000 page is 16 KB, 4 KB is selected to match the traced system.

[d] Code is replicated in all nodes.

Table 7.1: Experimental parameters.

Type	Short signal	Long signal
Processor request bus occupancy	10	170
Processor response bus occupancy	0	170
CCC recall bus occupancy	10	NA
CCC response bus occupancy	10	160
Memory bank access occupancy	100	100
Interconnection link occupancy	20	180

Table 7.2: Signal occupancies of the shared resources in nanoseconds.

Table 7.3 shows the main latency parameters used with the four systems. Appendix A lists the CCAT configuration file of system **D2**, which specifies all the used system parameters, occupancies, and latencies. The used latency and occupancy values are based on the typical processor and bus values from the R10000 user's manual [R1097], and an Origin 2000 microbenchmark evaluation [HLK97].

Aspect	Latency
Processor request[a]	90
Processor snoop response	80
Processor data response	190
CCC	50
Memory	100
Router	40
Network link	10

[a] latency between detecting the miss and requesting the bus.

Table 7.3: Values of the main latencies in nanoseconds.

When there is no contention, it takes 460 nsec to satisfy a processor miss from the local memory, and 690 nsec from a remote memory through one router.

7.4.2 Applications

We used six applications to evaluate the four systems: Radix, FFT, LU, and Cholesky from SPLASH-2 [WOT+95], and CG and SP from NPB [B+94]. The inherent characteristics of these applications are presented in Chapter 3. Table 7.4 shows the two problem sizes used in this evaluation. These applications are compiled on an SPP1600 and instrumented using SMAIT. The instrumented applications pipe detailed traces to CCAT in an execution-driven simulation setup, as described in Chapter 2. The evaluation presented in this chapter is based on the performance of the main parallel phase.

Application	Problem Size I	Problem Size II
Radix	256K integers	2M integers
FFT	64K points	1M points
LU	256×256	512×512
Cholesky	`tk15.O` file	`tk29.O` file
CG	$1,400$/15	$14,000$/15
SP	16^3/100	64^3/400

Table 7.4: Sizes of the two sets of problems analyzed. The two numbers specifying the problem size of CG and SP refer to the problem size and the number of iterations, respectively.

The SPLASH-2 applications are executed using the default command line options, however FFT and Cholesky executions included advice about the cache parameters according to Table 7.1.

7.5 Preliminary Evaluation

In this section, we investigate the design space to select a suitable design domain for evaluating the four systems. Our main objective here is to set some of the system variables to fixed values to enable us to perform an evaluation of the four systems that is realistic, and relevant for assessing their worthiness in reducing communication cost.

7.5.1 Cache Size

As presented in CIAT's working set analysis (Section 3.4.2), the working set of each of the six benchmarks is smaller than 4 MB. Hence, the Origin 2000 cache size is big enough to avoid most of the capacity misses. This is confirmed by the CCAT results shown in Figure 7.5 that shows the percentage of cache misses relative to the total number of instructions. These results are obtained using the default parameters (except for cache size): system **D2** with the conservative processor

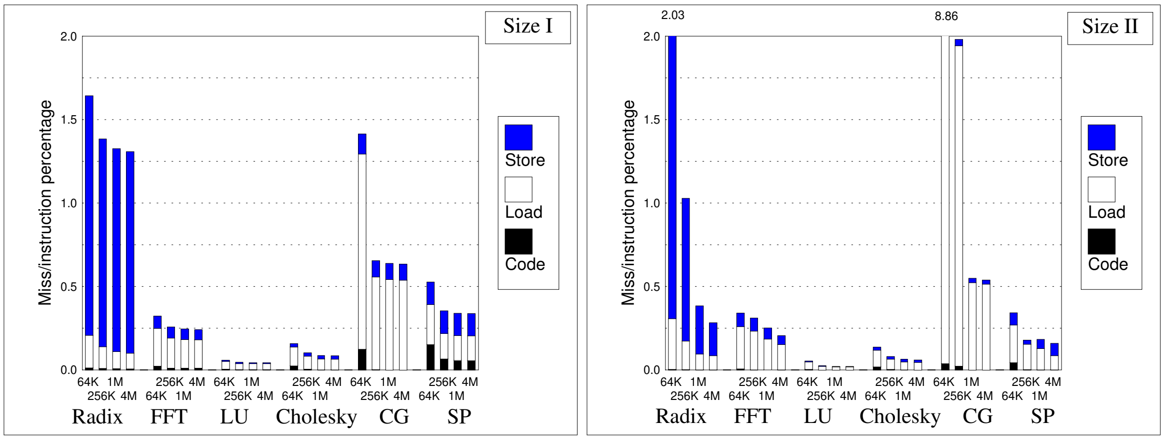


Figure 7.5: The percentage of cache misses to instructions using four cache sizes for the two problem sizes. All caches are two-way associative and 128 bytes wide.

model, 32 processors, and 4 memory banks per node, with four different cache sizes.

Note that Figure 7.5 shows three miss types: code miss, load miss, and store miss. The store misses include storing to a shared line, as well as storing to an uncached line. When storing to a shared line, the processor must request and get the exclusive ownership before performing the store. Only SP with problem size II has a noticeable code miss ratio with big caches; SP has complex data structures that result in some conflict misses in the combined secondary cache. In five of the six applications, the ratio of load misses is higher than the ratio of store misses. This observation agrees with the CIAT analysis presented in Section 3.4.4, which shows more RAW than WAR and WAW. Radix is different; it has more store misses than load misses because it has high WAW accesses and a lot of false sharing. For Radix and CG with problem size II, the large important working sets generate very high miss ratios with small caches.

For the two problem sizes, there is not a large reduction in the miss ratio as cache size is increased from a 1-MB cache to a 4-MB cache. This stabilization (even given the modest continuing reductions in Radix and FFT, and even SP, for problem size II) suggests that the remaining misses are primarily coherence misses, which in fact confirmed in the next subsection. In the following evaluation of the four systems, we fix the cache size at 4 MB. Thus, minimizing the effects of capacity misses and focusing on coherence misses as the primary cause of remote communication.

7.5.2 Number of Processors

Figure 7.6 shows the execution time as a function of the number of processors involved in solving the problem. The five execution time components shown are taken from the processor on the critical path. In serial phases, Processor 0 determines the critical path; and in each parallel phase, the processor that reaches the phase-ending synchronization barrier last is on the critical

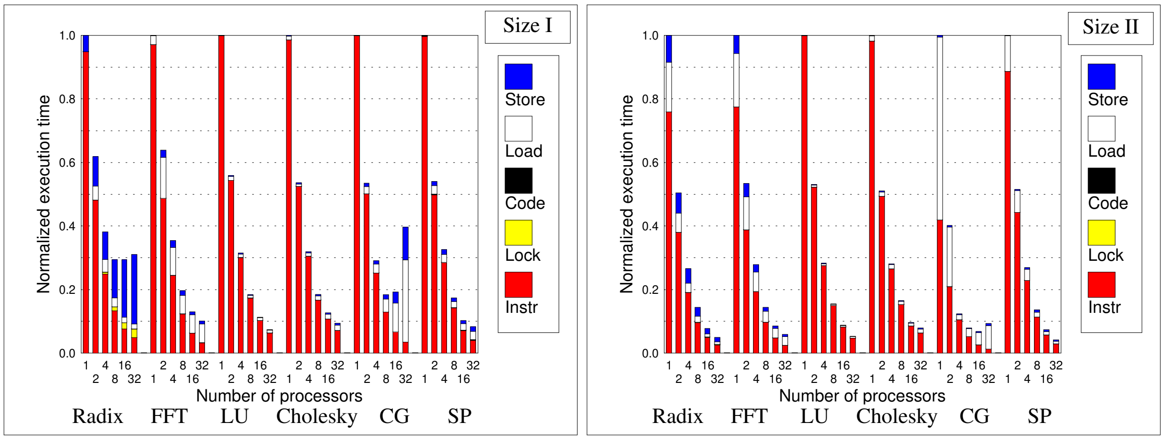


Figure 7.6: Execution time (normalized to single processor execution time) as a function of the number of processors, using 4-MB cache. The components of the execution time are found from the critical path. Speedup is the inverse of this time.

path; thus there is no barrier wait time. Here, we also use the default parameters (except for number of processors), with the conservative processor model.

The five components are: busy time executing instructions (at 1 instruction per clock), lock wait time, and stall time due to code, load, and store misses. The figure shows that these applications are potentially scalable up to 32 processors, as the instruction time drops with more processors. Some other notable observations are:

- For problem size I with one processor, there is negligible stall due to cache miss, indicating that the working set sizes fit in the 4-MB cache. However, the contribution of miss time increases as the processors increase, which confirms that most of the misses with 32 processors are coherence misses. This increase agrees with CIAT's analysis that communication increases with more processors.

- With 32 processors, problem size II has less normalized cache miss stall than size I. Consequently, the execution time with size II scales better than size I.

- For size I, the execution times of LU, Cholesky, and SP decrease as the processors increase, because of their low communication rates.

- Radix scalability is limited by store misses due to WAW and false sharing.

- Although FFT has slightly less communication than SP, it spends more time satisfying its misses because of the bursty nature of its remote communication (as predicted by CIAT's time distribution analysis in Section 3.4.5).

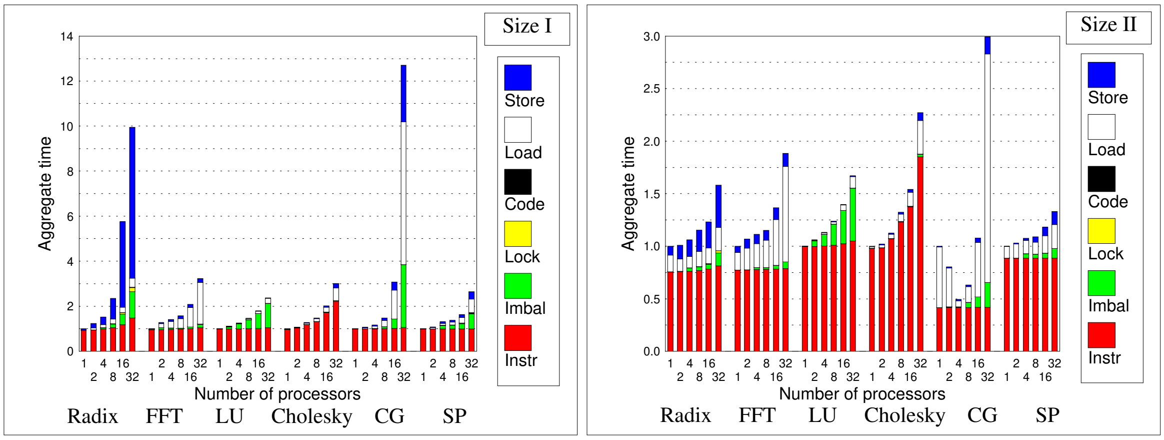


Figure 7.7: Aggregate time as a function of the number of processors. The aggregate time is the execution time components of each processor summed over all processors and normalized to the 1-processor time. Speedup is the ratio of the number of processors to this time. Note the change of scale for problem size II.

- CG also has a bursty behavior that is reflected, with size I, in its high store miss time relative to its store miss ratio.
- For size II with few processors, the working sets of Radix, FFT, and CG are larger than the cache size, resulting in a significant capacity miss contribution to the execution time.

Figure 7.7 shows the aggregate time of all processors, not just those on the critical path. This data is from the same simulations used to get the data presented in Figure 7.6. Here, we see a new component, namely load imbalance, which reflects the wait time at the synchronization barriers. In applications with ideal speedup, the aggregate time remains constant as the processors increase. However, for variety of reasons, the aggregate times of these applications increase as more processors are added.

Generally, problem size II has better scalability than size I (note change of scale in Figure 7.7). CG even shows a superlinear speedup at 2, 4, and 8 processors with size II; when the problem is partitioned among more processors, the working set of each processor gradually becomes small enough to fit in the processor's cache and eliminate most capacity misses—the gains from this effect exceed the losses due to increasing coherence misses in the transitions from 1 to 2 and from 2 to 4 processors.

The aggregate time of Cholesky increases with more processors, mainly due to the added busy-wait instructions at flag-based locks. However, the aggregate times of other applications increase mainly due to load imbalance and miss stall. As the processors increase, the communication time (miss stall) contribution to the execution time becomes significant, especially in Radix, FFT, and

CG.

We decided to evaluate the four systems using 32 processors, because with this number of processors there is enough communication to evaluate the effectiveness of these systems in handling communication.

7.5.3 Number of Memory Banks

Successive cache lines are interleaved among the memory banks of the Origin 2000; each Origin 2000 node has 4 to 32 memory banks. Figure 7.8 shows the execution time as a function of the number of memory banks per node normalized to the time obtained when only one bank per node is provided.

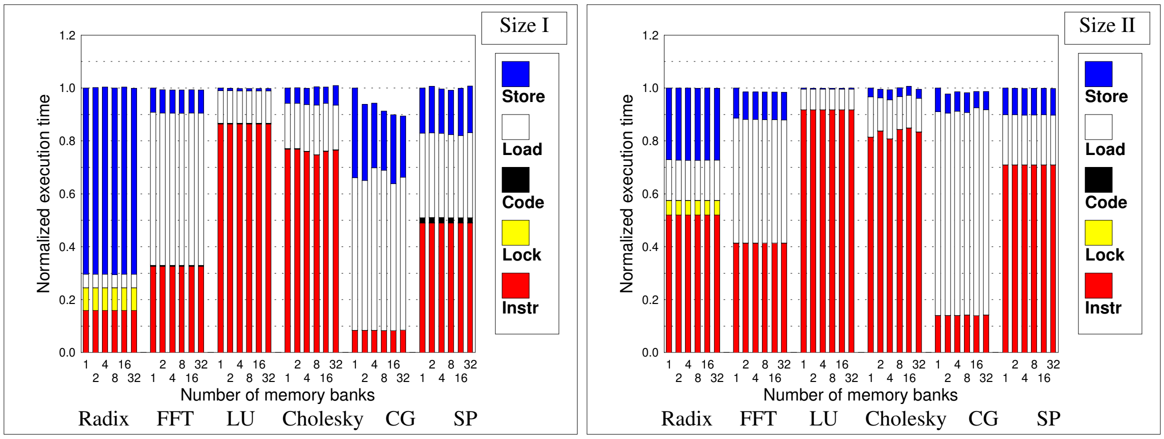


Figure 7.8: Normalized execution time as function of the number of memory banks per node, using 32 processors.

The change in the execution time with more banks is small, and not always in the same direction. Each execution time comes from one execution-driven simulation experiment. In each execution, the operating system gives the application a set of virtual addresses. The different sets generate different address conflicts, which slightly affects the total miss time, and consequently the execution time.

This data shows that the execution time is not very sensitive to the number of memory banks (CG does suffer from memory contention with one bank), and that near maximum performance is achieved with only a few banks. In following system evaluation, we use four banks per node.

7.5.4 Latency Overlapping

In this section, we evaluate three processor models using the default parameters. Figure 7.9 shows the execution times using the conservative and two aggressive processor models normalized to the time of the conservative model. The execution time components, summed over all phases

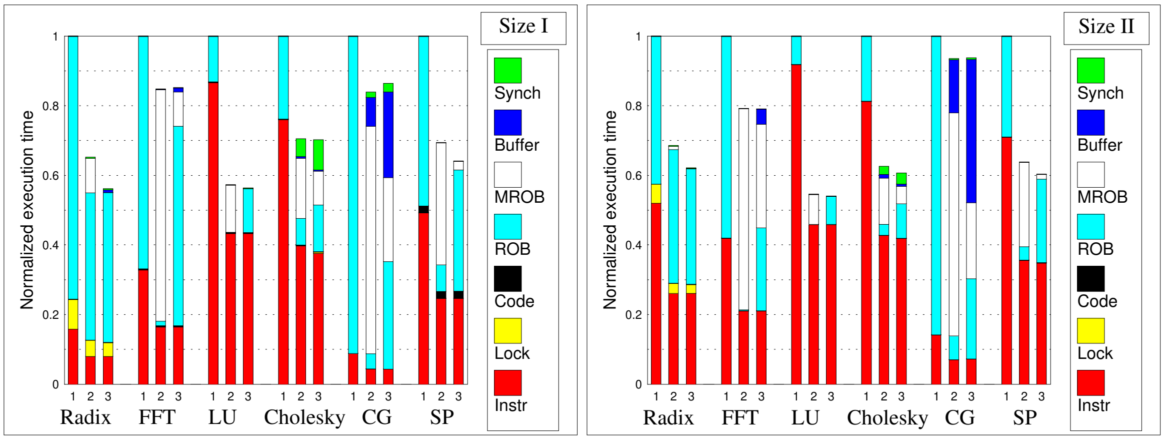


Figure 7.9: Normalized execution times using three processor models: (1) conservative: 1-IPC with 1-entry ROB; (2) aggressive: 2-IPC, 48-entry ROB, 16-entry MROB, and 4-entry miss buffer; (3) more aggressive: 2-IPC, 128-entry ROB, 64-entry MROB, and 8-entry miss buffer.

(using the critical path processor in each phase), are: instruction, lock wait, code miss stall, full ROB stall, full memory ROB stall, full outstanding miss buffer stall, and synchronization stall. The synchronization component is the stall time at the synchronization points while waiting to finish outstanding misses. With the aggressive models, Cholesky and CG have noticeable synchronization times due to Cholesky's many locks and CG's many barriers. Synchronization time is not visible when using the conservative model because there are no outstanding misses when a synchronization point is reached, and in these experiments only negligible synchronization time results from outstanding write-backs. Figure 7.10 is the same comparison, but presents the ROB, MROB, and buffer stall times according to the type of the oldest unsatisfied miss when the stall occurs.

The execution time using either of the two aggressive models is always less than the time of the conservative model, primarily due to the instruction time reduction by about fifty percent. Moreover, the aggressive models slightly reduce execution time further in Radix, CG, and SP by tolerating some of the miss latency. While the more aggressive model (3) reduces the processor stall time more than the aggressive model (2) with Radix, LU, Cholesky, and SP, its stall time with FFT and CG is larger than that of the aggressive model. The more aggressive model is successful with four applications where it overlaps more misses and tolerates higher percentages of the miss time, and is less successful with the two applications that have bursty communication as overlapping more misses increases the system congestion and increases the latency of individual misses.

The reduction in the miss stall time of the aggressive model is not as impressive as it might be because of the limited ROB size (and, in CG, the outstanding miss buffer size as well). Although the aggressive model assumes that instructions are independent and the processor does not stall due to data dependencies, the ROB is not large enough to mask the long communication latency, nor to

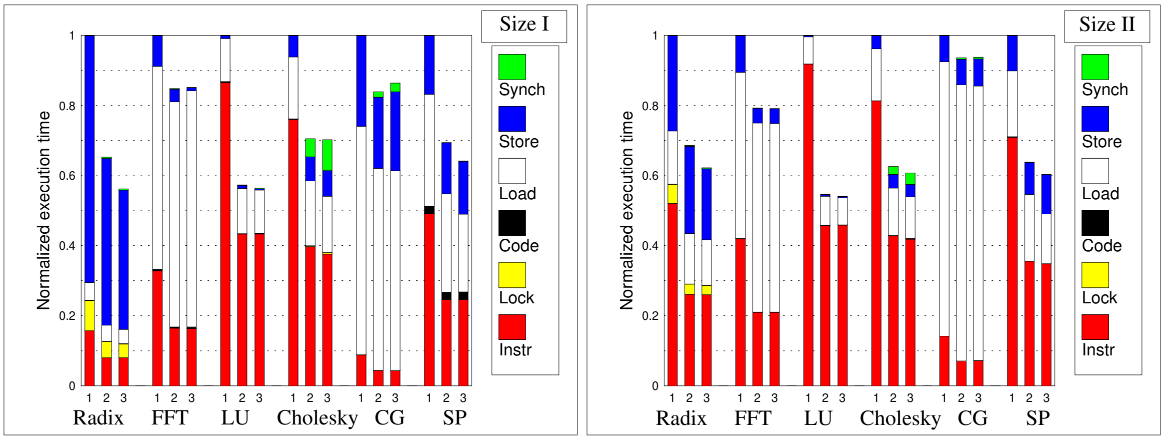


Figure 7.10: Normalized execution times of the conservative and aggressive processor models showing the miss type of the instruction blamed for processor stall.

frequently capture multiple misses in one window so that their misses can be overlapped. This is also true with the more aggressive model. However, as the fraction of ROB entries dedicated for memory instructions grows from 1/3 in the aggressive model to 1/2 in the more aggressive model, the full MROB stall is often replaced by full ROB stall.

The stall time of Cholesky is actually slightly higher when using the aggressive models due to contention effects. The aggressive models reduce the instruction time and consequently the misses become less spread over time, which increases the contention on shared resources and in this case this increase exceeds the benefits of increased latency tolerance.

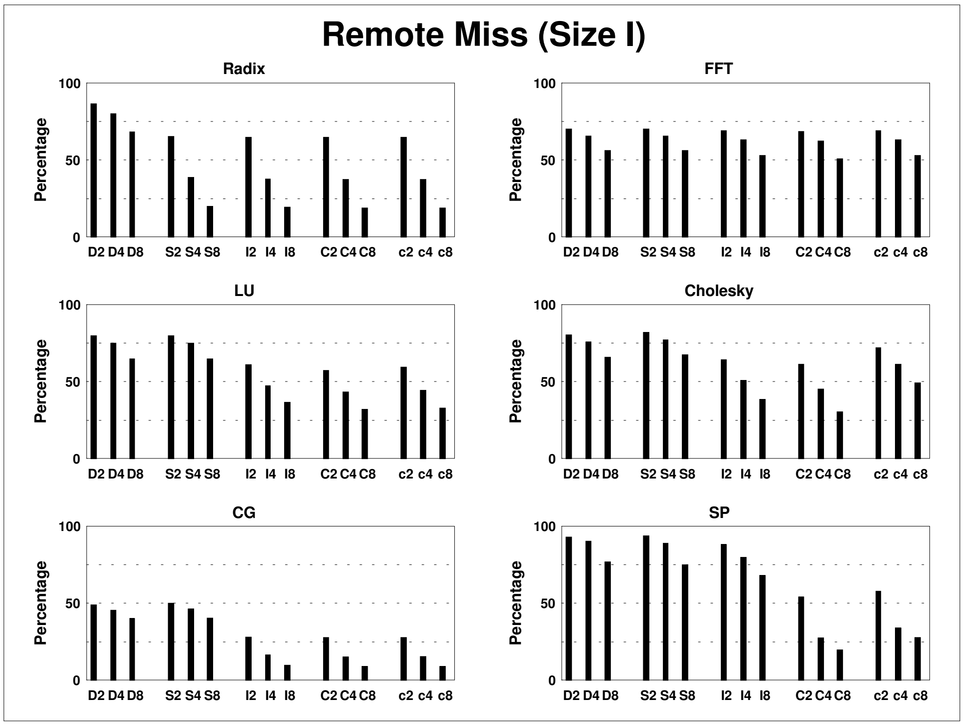
In the following section, we present an evaluation of the four systems using the aggressive model because we believe that its traffic more realistically represents the traffic generated by the R10000 processor. However, when evaluating these systems using the conservative model we get the same qualitative results, and the main conclusions drawn about the four systems in the next section using the aggressive model also hold with the conservative model.

7.6 System Evaluation

This section evaluates the performance of the four systems (**D**, **S**, **I**, and **IC**) using the aggressive processor model with 32 processors, 4-MB caches, and 4 memory banks per node. Each system is evaluated with 2, 4, and 8 processors per node. We evaluate system **IC** using two interconnect cache sizes: 4 MB (**C**), and 16 KB (**c**).

Figure 7.11 shows the percentage of misses that are satisfied from a remote memory or remote cache. Since these remote misses are more expensive than local misses, a small percentage is better.

In all cases, as the number of processors per node increases, the percentage of remote misses decreases. This is expected as more memory locations and more processors become local with fewer



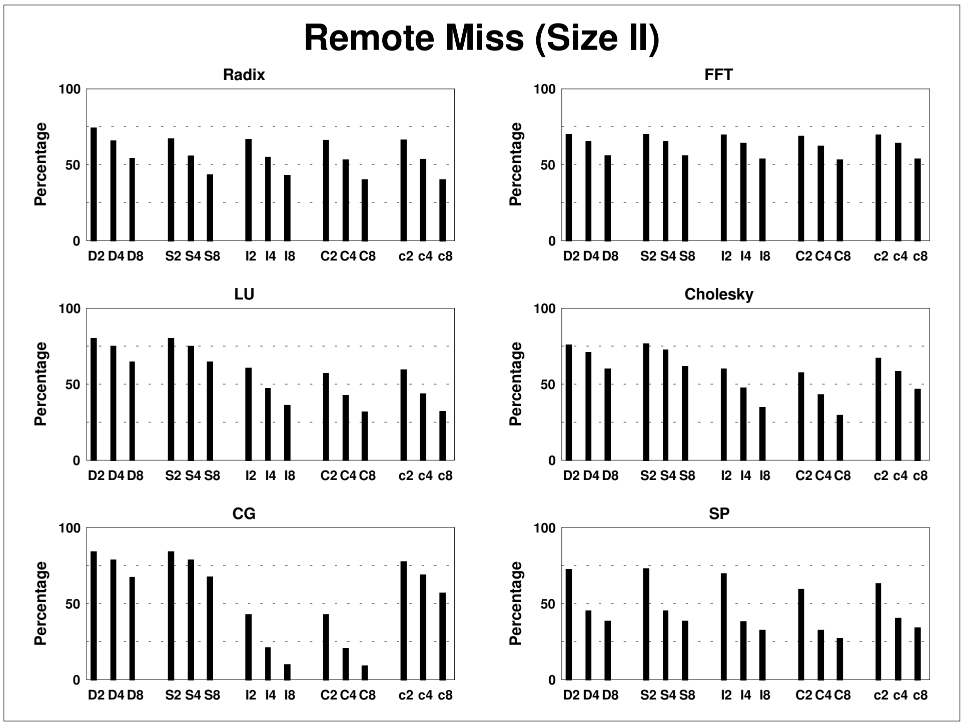


Figure 7.11: Percentage of misses satisfied from a remote node or cache for the four systems: (D) pure directory-based, (S) snoopy within the node, (I) caches supply clean lines, (C) with a 4-MB snoopy IC, and (c) with a 16-KB snoopy IC. Evaluations shown for 2, 4, and 8 processors per node.

nodes. In general, the three snoopy systems, **S**, **I**, and **IC**, have smaller remote miss percentages than system **D**. Systems **I** and **IC** have smaller remote miss percentages than the other two systems with LU, Cholesky, and CG. System **C**, which is the best overall in reducing remote latency, has smaller remote miss percentages than the other systems with SP.

System **c** reduces remote miss as well as system **C** when the communication working set is small enough to fit in the 16-KB IC. This is not the case with Cholesky and with CG when solving problem size II, and **C** shows an improvement over **c** for them.

Figure 7.11 demonstrates that the three snoopy systems are successful to varying degrees in achieving their intended purpose of reducing the remote misses relative to the base system. The following subsections evaluate these systems in more detail by analyzing their execution times, traffic, and miss latency.

7.6.1 Execution Time

Figures 7.12, and 7.13 show the execution time of the six applications for the two problem sizes. In each of the following paragraphs, we discuss the relative performance of each application on the four systems.

Radix has a lot of WAW communication and false sharing, consequently it spends a lot of time satisfying store misses. The snoopy systems gain some advantage over system **D** by allowing bus snooping. This gain is larger when more processors per node are used; however, since execution time actually increases with more processors per node, this is a moot point. Snooping reduces store miss time primarily by locally satisfying store misses to lines that are dirty in some other local processor. Satisfying some misses locally has a positive impact on the whole system because of reduced system traffic and contention. Problem size II benefits less than size I from these advantages because it has a lower miss rate and less contention. For both problem sizes, the smaller IC is nearly as efficient as the larger IC because most of the performance advantages come from snooping.

FFT is dominated by communication and does not benefit much from these techniques. However, system **C** does offer the best performance.

LU has low communication rate and low stall time percentage, therefore, the three techniques cannot significantly reduce the execution time.

Cholesky has significant RAW communication with an average sharing degree of seven processors. Consequently, systems **I** and **C** reduce the load miss time by satisfying multiple local readers with one remote request. System **c** has higher miss time than that of system **C** because its IC is not large enough to capture the communication working set, and thus many lines are

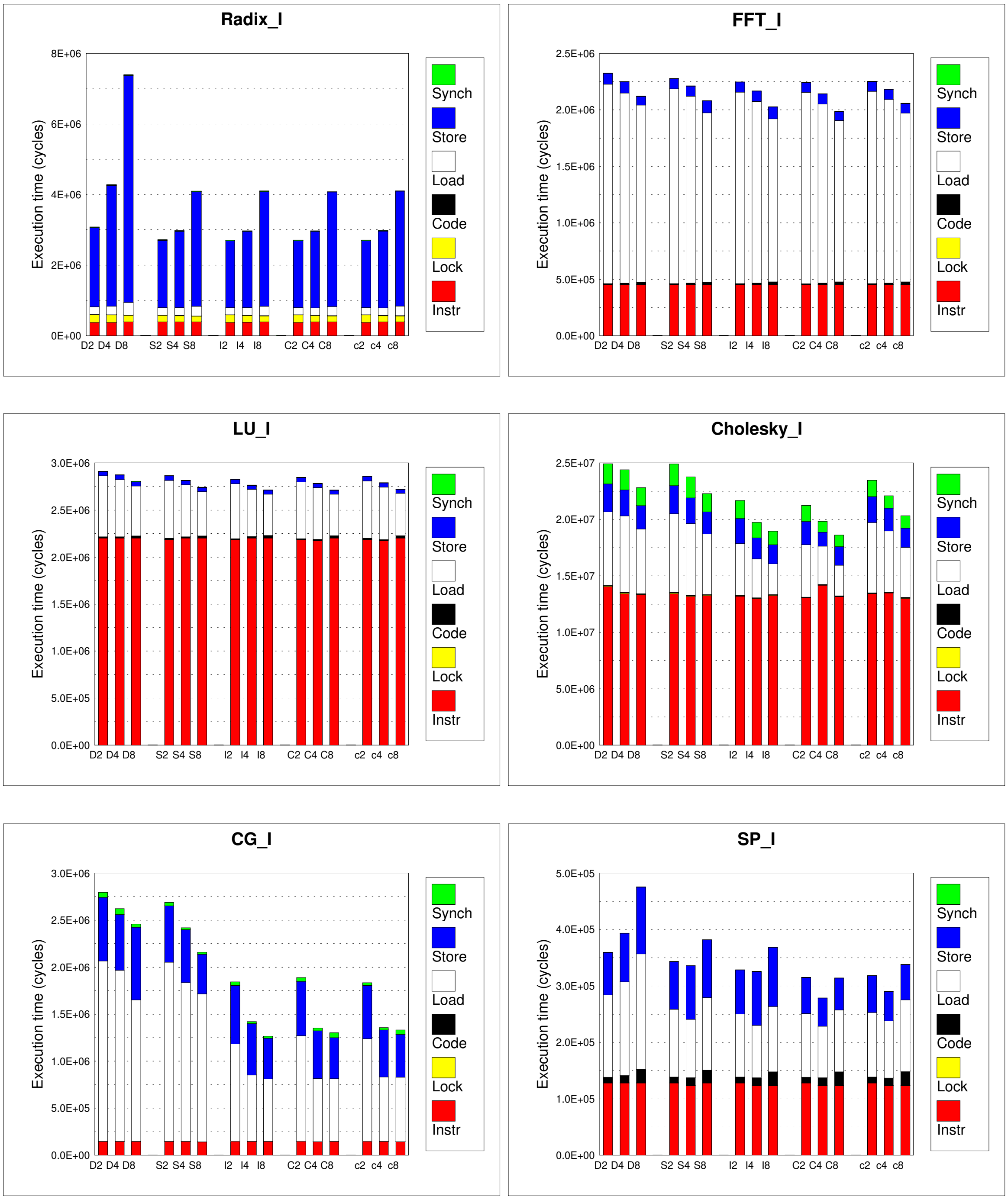


Figure 7.12: Execution times of the four systems (problem size I).

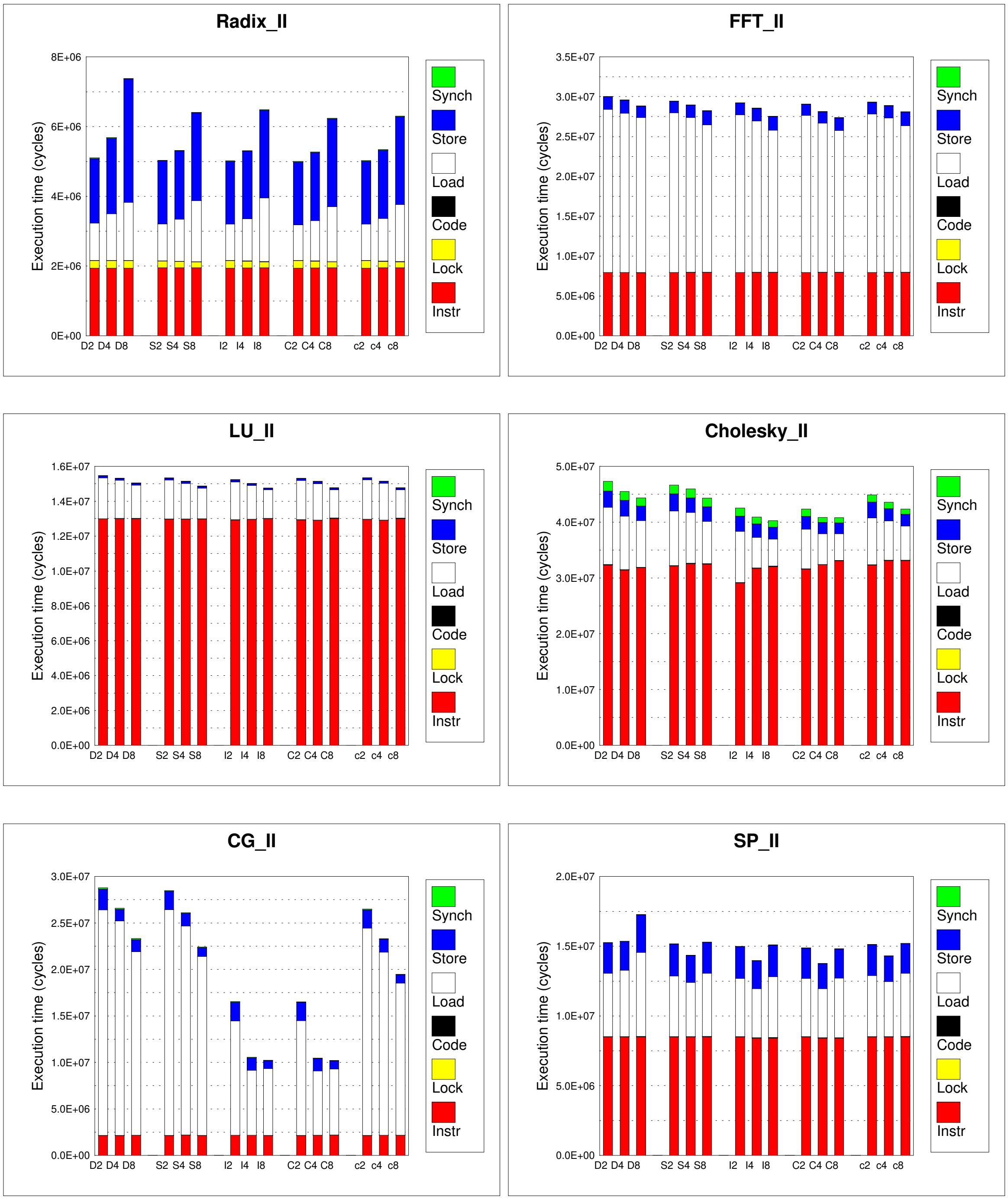


Figure 7.13: Execution times of the four systems (problem size II).

replaced before their presence in the IC is fully exploited.

CG has more RAW communication and higher sharing degree than Cholesky, and therefore, system **C** reduces the load miss time even further. The IC size in system **c** captures the communication working set in problem size I, but not in size II.

SP has significant local producer-consumer communication. System **C** provides the best performance because it satisfies repetitive RAW and WAR locally. Problem size II has a lower communication to computation ratio than size I, and the memory allocation policy is more successful in localizing shared pages with size II. Therefore, size II benefits less from these techniques.

In general, system **IC** performance is as good as system **I**, and better than **I** when the IC is large enough to capture the communication working set. System **IC** performs the best when there is significant local producer-consumer communication on remote lines.

For all these applications, **C2** out-performs **D2**. However, **D2** does perform better than the three snoopy systems in those serial regions where there is little or no communication and the cache misses are satisfied within the local node. For example, CG has a serial initialization phase where only Processor 0 is active. For problem size I, **C2**'s execution time is 2.5% higher than that of **D2** in this phase due to the overhead of snooping which increases the miss latency by 15%.

When the processors per node increases, the number of nodes decreases and the remote latency decreases. This effect reduces the execution time of system **D** with FFT, LU, Cholesky, and CG. However, with more processors per node, the bus contention increases, which in Radix and SP dominates over the increasing benefits cited above. Consequently, 2-processor nodes produce the highest performance for Radix and 4-processor nodes are best for SP. The performance of the three snoopy systems relative to system **D** consistently increases as processors per node grows.

7.6.2 Traffic

Figures 7.14, and 7.15 show the average number of *active signals* per cycle in the network links, memory banks, and buses over the entire system for the two problem sizes. These figures represent the average traffic rate generated in these systems and the relative contention for the available resources of each type. Active signals are either signals utilizing these shared resources or waiting (queued due to contention) to use these resources. Note that one remote access may result in many signals. To provide another perspective on the significance of these numbers, note that the network can serve a maximum of 56, 24, or 10 signals in one cycle for systems with 2, 4, or 8 processors per node, respectively; similarly, the memories can serve 64, 32, or 16, and the buses can serve 16, 8, or 4.

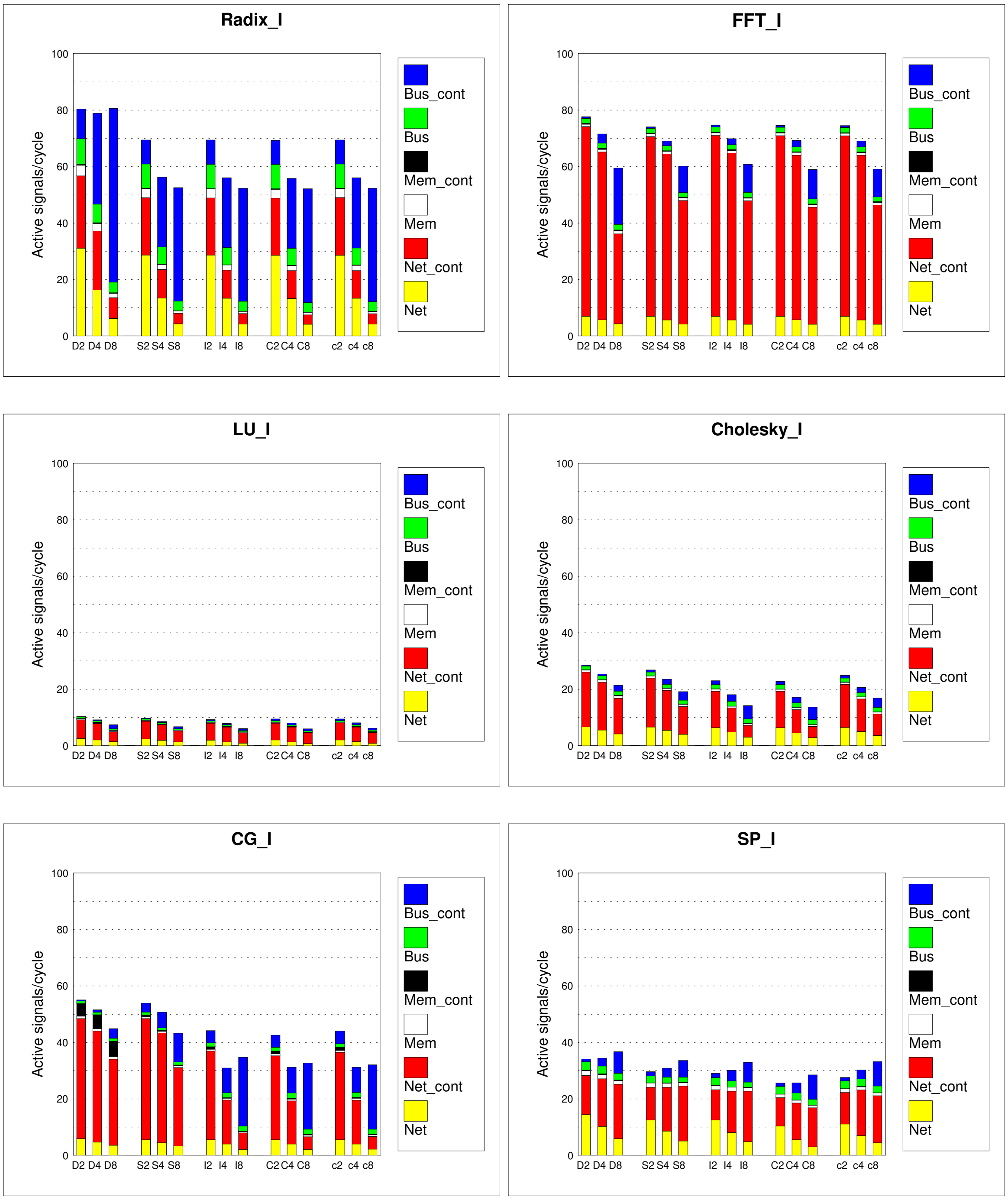


Figure 7.14: Average active signals per cycle (problem size I). Net is signals utilizing network links, Net_cont is signals waiting to utilize links, Mem is signals accessing memory banks, Mem_cont is signals waiting to access banks, Bus is signals utilizing buses, and Bus_cont is signals waiting to utilize buses.

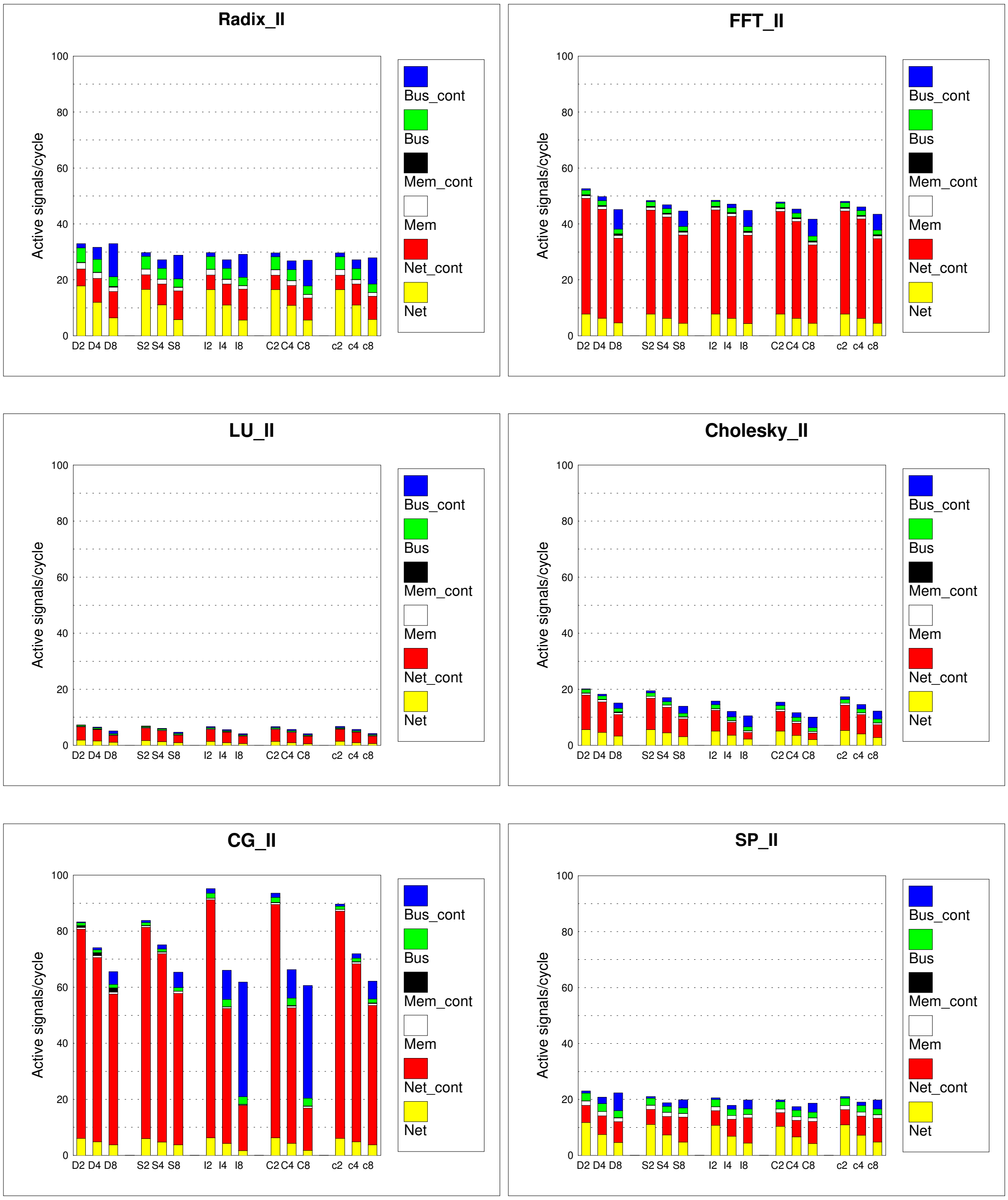


Figure 7.15: Average active signals per cycle (problem size II).

Problem size II generally has lower traffic rate and contention than size I in five of these applications due to lower miss ratios; CG is the exception because its miss ratio with size II is not much less than with size I, and its load balance with size II is better than size I. Note also that in all cases as processors per node increases, the network utilization and contention decrease while the bus utilization and contention increase.

With problem size I, Radix has high network and bus contention and utilization. The three snoopy systems reduce this behavior by 10% to 20%. The two figures expose FFT's bursty remote communication as they show that most of the active signals are either traveling or waiting for service on the network links. As LU and Cholesky have low miss ratios, they have low traffic rates.

CG with system **D** has noticeable memory contention (particularly for problem size I) in addition to its high network contention (for both problem sizes, but more so for problem size II), which occurs due to its bursty behavior when updating the partial sums. The snoopy systems reduce this memory contention by not allowing more than one simultaneously outstanding request for a particular line on each bus. Moreover, systems **I** and **IC** reduce the network traffic, particularly for 4 or 8 processors per node. With CG and problem size II, systems **I2** and **C2** have more network contention than system **D2** because their execution times are significantly lower than that of system **D2**.

SP has an average number of signals utilizing the network links that is larger than that of FFT or CG. However, SP has less network contention than FFT or CG because its communication is less bursty.

7.6.3 Average Miss Latency

Figure 7.16 shows the latency of an average miss. For Radix and SP, the average miss latency increases as the number of processors per node increases due to bus contention. This trend is also present in CG with systems **I** and **C** when the number of processors per node increases from 4 to 8. However, for the other cases, the latency decreases as the number of processors per node increases due to the reductions in network time and remote miss percentage.

This average latency is much higher than the memory latencies calculated from the signal latencies with no contention (see Subsection 7.4.1) due to the following reasons:

1. As most of the misses are due to communication, each is often satisfied after a series of hops that involve multiple processors, which is more expensive than satisfying them directly from memory.

2. When multiple processors are active and each can have multiple outstanding misses, there is contention for utilizing shared resources. Thus, the miss latency becomes higher as the signal wait time for busy resources is encountered.

3. The Origin 2000 has "memory-less" cache coherence control, i.e. it uses NACK-and-retry

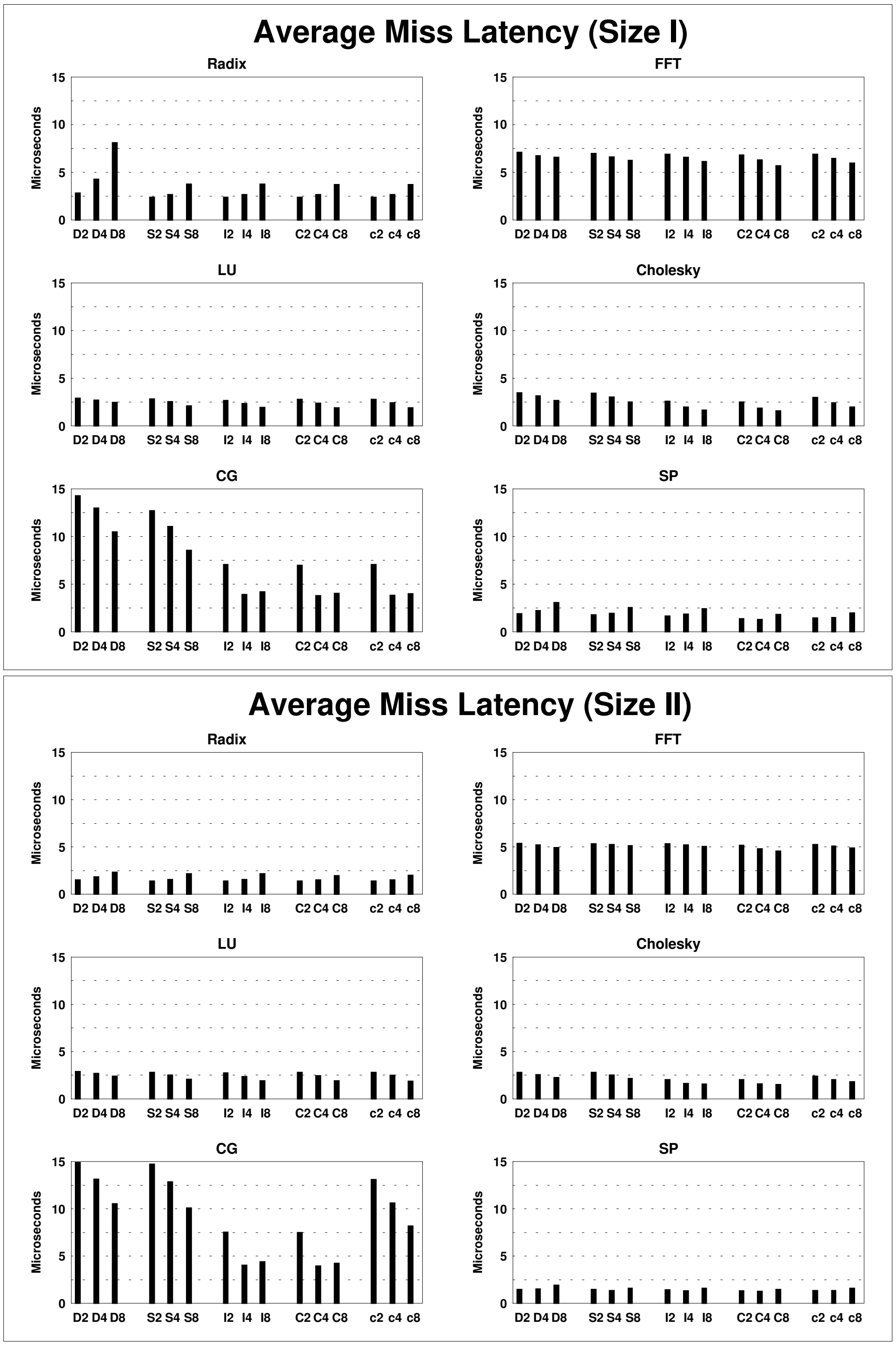


Figure 7.16: Average miss latency in microseconds.

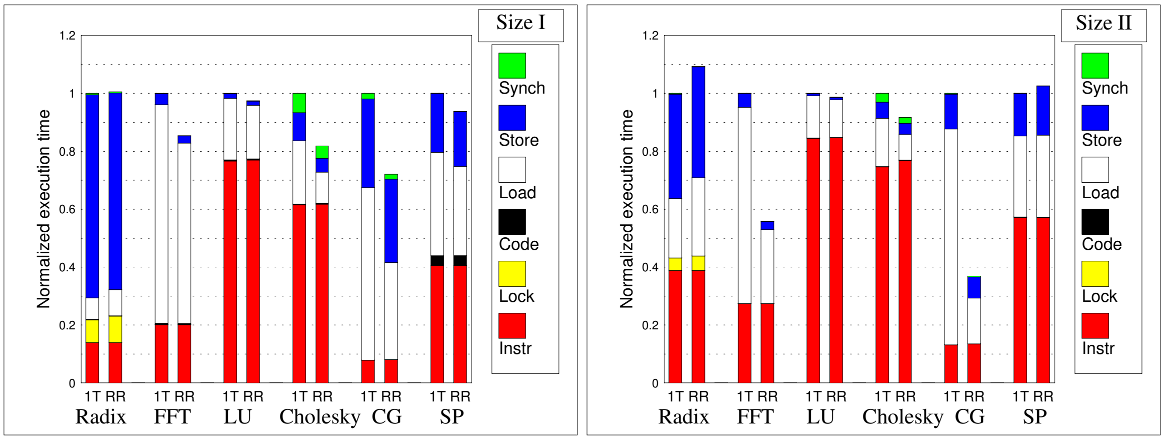


Figure 7.17: Normalized execution time using system C2 with two memory allocation policies: first touch (1T), and round robin (RR).

when the memory cannot immediately satisfy a request. When the CCC receives a request from a processor for a line in a busy state (the directory tag associated with the line is marked busy while the CCC is recalling the exclusive copy of this line for another processor), it does not buffer the new request, but replies with a negative acknowledgment. Thus, the requester retries until it succeeds, which increases the system contention and miss latency further.

4. Some applications, particularly FFT and CG, have bursty communication that result in high contention. Additionally, CG has high invalidation degree, and therefore each WAR miss generates many invalidation and acknowledgment signals.

5. As these applications have a lot of producer-consumer communications, the speculative memory operations are often useless as the producer has a dirty copy. Therefore, the signals that carry speculative data increase the system traffic, contention, and miss latency.

6. For some applications, the first touch memory allocation policy often allocates most of the memory pages in Node 0 when Thread 0 initializes the shared data structures in the serial initialization phase. Consequently, the system traffic becomes unevenly distributed and Node 0 is the bottleneck.

Figure 7.17 shows the execution times using system **C2** with first touch and round robin memory allocation policies, normalized to the time with first touch. FFT, LU, Cholesky, and CG have significantly lower miss time with round robin than with first touch due to the reduced contention on the network links that connect Node 0. However, first touch with Radix and SP when solving problem size II has less miss time than round robin as they initialize most of their shared data in parallel phases and first touch is thus more successful in localizing data.

7.7 Chapter Conclusions

In this chapter, we have studied the communication cost in CC-NUMA systems and evaluated some techniques that have the potential for reducing it. System **D**, which relies purely on the directory for ensuring cache coherence, favors sequential programs in multiprogrammed parallel environments over shared-memory applications with high communication percentages. For our six applications, the three snoopy systems handle communication better than system **D**. System **I**, which allows processors to supply clean cached lines, and system **IC**, which uses an interconnect cache for reducing the communication cost of remote line accesses, both have superior performance.

System **IC** handles local producer-consumer communication better than system **I**. Moreover, unlike system **I**, system **IC** is compatible with the cache coherence protocols supported by modern processors. These two systems, however, do have increased system cost: system **I** because it requires higher processor cache bandwidth, and system **IC** because it requires the addition of an interconnect cache in each node.

Although system **IC** increases the cost of the node, it has less traffic and contention than system **D** and permits more processors per node. When the node becomes larger, the overall system cost decreases as fewer network links, routers, and coherence controllers are needed. The cost reduction with larger nodes may exceed the IC cost.

The IC can be implemented in the local memory to reduce its cost. In this case, a dedicated section of memory is used to hold the data, while the cache coherence controller still stores the IC tags in order to keep up with the demand for snooping. Consequently, the IC can continue to participate in the bus coherence protocol without causing more delaying than a typical processor cache. As we assume that the IC supplies lines with a 190 nanoseconds latency, getting an IC data line from the local memory would not take much more time.

Finally, we have noticed that the effectiveness of system **IC** is best with applications that have high communication costs. Furthermore, it reduces the execution time of all six case study applications which have been carefully designed to achieve low communication rates. Thus, we expect that other less-tuned shared-memory applications would have at least this much benefit. This technique reduces the communication cost and consequently lowers the NUMA factor of DSM systems which is a step to enable CC-NUMA systems to efficiently support more applications with less tuning efforts.

CHAPTER 8

CONCLUSIONS

Here, we present our main conclusions about this dissertation's methodology and tools, the case-study applications, the existing systems and approaches that have been assessed in this research, and the techniques that we have introduced to reduce the communication cost in DSM systems.

8.1 Putting It All Together

In this dissertation, we have demonstrated a methodology and tools for analyzing shared memory applications to support designing scalable shared-memory systems. The demonstration went through four design stages: characterizing a wide range of target applications to get a basic understanding of application properties and performance, measuring the performance of some existing systems to prepare accurate configuration files for them, evaluating current approaches to identify strengths and weaknesses, and evaluating and proposing solutions for some of the identified weaknesses.

Characterizing the inherent application properties gave us better command in later stages to select appropriate experimental setup, and to explain and rationalize the achieved application performance on specific system configurations. For example, CIAT's characterization of working sets gave us the insight to select cache sizes that have a small ratio of capacity misses. CIAT's characterization of the communication patterns and their variation over time also played a vital role in explaining the application performance on the systems evaluated in Chapters 6 and 7. Moreover, as CIAT revealed that the case-study scientific applications have a lot of WAR and RAW communication, we were motivated to experiment with the techniques that reduced the cost of these communication patterns, as presented in Chapter 7.

CDAT's simplicity and flexibility enabled us to evaluate a wide range of current CC-NUMA system approaches and identify one promising approach for further investigation and improvement. Similarly, system designers can use CDAT to investigate much larger design space in the early design stages of their projects. However, in order to evaluate optimization techniques for the selected system approach, we used the more detailed simulator, CCAT which is targeted to CC-NUMA systems with memory-based coherence. The CCAT analysis presented in Chapter 7 shows that the performance of the case-study applications is heavily affected by contention on shared resources; evaluating these techniques with CDAT would be inconclusive as it does not model contention. System designers do need detailed simulators like CCAT at later design stages to fine tune their designs.

8.2 Methodology and Tools

This research has developed a suite of tools that mechanically analyzes DSM applications and provides application characterizations in a form that is directly useful to programmers and designers.

This suite is increasingly gaining acceptance and use. Graduate students in the parallel architecture course in the University of Michigan have used it to characterize and tune shared-memory applications. Several researchers in the Hewlett-Packard Laboratories use it to characterize applications, parameterize workload generators, and evaluate alternative design options for the next generation of DSM systems. Moreover, we have used it to characterize several important aspects of a variety of scientific and commercial shared-memory applications [AD98c], calibrate and evaluate the performance of two generations of the Convex Exemplar [AD98b, AD98d], and evaluate three CC-NUMA approaches [AD98a].

Our methodology relies on instrumenting shared-memory applications to trace the data and code streams. The trace is analyzed using a combination of configuration independent and configuration dependent techniques.

SMAIT is one of few tools that enable collecting traces of multi-threaded shared-memory applications. Its support of execution-driven analysis enables analyzing a wide range of applications including large and dynamic applications. However, it requires source code availability and does not trace the operating system. These two limitations did not, however, adversely affect our case studies of scientific applications since their source code is available and they do not spend significant time in the operating system. Although we did encounter these problems with the commercial applications, our analysis tools can be used to analyze application traces collected by other tracing tools.

Splitting the application analysis into configuration independent and configuration dependent analysis provides a clean and efficient characterization of application performance. Configuration independent analysis gives a basic understanding of the inherent properties of an application, while configuration dependent analysis enables evaluating the application performance on a particular system configuration.

The algorithms that were developed and incorporated in CIAT, the configuration independent analysis tool, provide fast and informative characterizations of several aspects of the application's inherent properties including general characteristics, working sets, concurrency, communication patterns, communication variation over time, communication slack, communication locality, and sharing behavior. These characterizations aid the programmers in understanding and tuning DSM applications, and provide the DSM designers with a basic understanding of their target applications' properties.

CDAT and CCAT were developed to evaluate an application's performance on a particular system configuration. They offer two levels of detail and coverage of DSM system approaches. CDAT simulates a wider range of DSM approaches than CCAT; CCAT simulates the application on more detailed models of the processor, bus, memory, and interconnection network than CDAT. CDAT is faster than CCAT, but is less accurate. CDAT is useful for performing high level comparisons among a wide range of alternatives at the early design stages, and CCAT is useful for performing accurate evaluations of alternative design optimizations at later design stages.

CCAT incorporates a more detailed processor model than CDAT, and captures some properties in modern sequentially consistent processors that affect system traffic. However, CCAT's processor model does not track data dependencies, nor the state of the processor's registers and functional units.[1] A model that tracks these aspects would generate a more accurate memory access stream than CCAT's processor model, but would be much slower. We have decided to keep CCAT's processor model simple and fast in order to be able to simulate large applications on systems with large number of processors in a reasonable time. Standard processor simulators do exist and can be used for this purpose when needed, but we have found that the system level design issues targeted in this research are not very sensitive to the detail of the processor model. Specifically, the conclusions about the four systems evaluated in Chapter 7 hold for the conservative processor model as well as the aggressive processor model.

8.3 Case-Study Applications

In this research, we have used a collection of applications drawn from the scientific, engineering, and commercial domains. We have given special attention to characterizing these applications in order to understand their inherent properties. The application knowledge gained by analyzing these applications was of material benefit in conducting the later stages of this research.

These applications are well tuned for DSM machines and can generally exploit more processors. However, they do exhibit frequent communication accesses that often increase with more processors. On systems that have high communication costs, the coherence misses due to communication limit the scalability of some applications, particularly Radix and CG.

[1] Note that this is not a problem for CDAT due to its serial execution model.

The important working sets of these applications, with the problem sizes analyzed here, do fit within the large caches of modern processors. However, the working sets of Radix, FFT, CG, and SP get larger with larger problem sizes, which would generate a lot of capacity misses. Additional steps would then have to be taken to reduce the cache miss cost due to limited capacity, as well as coherence traffic.

We have noticed significant differences between the scientific applications and the commercial applications that were analyzed in this research. While the scientific applications access word and double-word memory objects in large loops with infrequent branching, the commercial applications access byte, half-word, and word objects with larger code sizes that contain frequent branching.

8.4 CC-NUMA Systems

We have evaluated four CC-NUMA systems using microbenchmarking and simulation. The Convex Exemplar SPP1000 and SPP2000 were evaluated using microbenchmarking; and the Convex SPP1000, the Stanford DASH, and the SGI Origin 2000 were evaluated using CDAT simulation.

The present generation of the Exemplar systems, the SPP2000, utilizes architectural and technological advances to achieve higher memory bandwidth than the SPP1000, and to overlap the latencies of multiple outstanding misses. However, local memory latency is not improving in proportion to increases in the processor clock rate. The SPP2000 also uses a richer interconnection network that transfers its smaller packets faster than the SPP1000. Consequently, the gap between the local and remote latencies in the SPP2000 is smaller than that in the SPP1000, which gives better programmability.

The SPP1000 and SPP2000 use custom-designed crossbars and controllers to support several processors per node, which greatly helps medium-scale parallel programs. The addition of this glue logic, however, raises the local memory latency, which penalizes sequential and small-scale parallel programs. The SCI protocol that is used in both systems to maintain global cache coherence has some inefficiencies that limit the remote aggregate bandwidth and lengthen the remote latency.

The CDAT evaluation of the CC-NUMA systems shows that when the three systems are put on the same technological level, they have analogous performance. The Origin 2000 has the lowest memory latency, but the DASH has the best performance because it uses the relaxed memory consistency model (which hides some of the miss time) and uses the Illinois protocol (which results in fewer remote misses than the Origin 2000). The SPP1000 has the highest remote latency due to its complicated SCI protocol that manages a distributed linked-list directory.

8.5 Reducing Communication Costs

Previous research has focused on reducing the remote communication cost by decreasing remote capacity misses. In this research we focused on reducing this latency by decreasing the cost of remote coherence misses.

We have shown that there are opportunities for reducing remote communication cost that are not taken advantage of by the coherence protocols supported by modern processors. These protocols favor small-scale multiprogrammed systems over scalable DSM systems. We have shown that when the processor cache coherence protocol is changed to allow caches to supply clean data, better performance is achieved for parallel shared-memory applications.

We have also shown that modern systems that focus on reducing latency can incorporate techniques for reducing the remote misses without significant latency increase. By supplementing the node with an interconnect cache that participates in the local bus coherence protocol for remote data, the resulting system achieves superior performance. Using this cache, the case-study applications benefit from reduced remote communication cost in spite of a slight latency increase; overall communication cost is reduced by up to 63% in some cases.

8.6 Future Work

In this dissertation, we have developed configuration independent analysis techniques for characterizing several aspects of shared-memory applications. However, there are other aspects that are now only characterized using configuration dependent analysis. For example, false sharing is characterized by simulating the application on a multiprocessor configuration with caches of some line length. Measuring the number of cache misses of lines invalidated by other processors due to accessing different memory locations that happen to be allocated in the same line, is a characterization of the false sharing. Since this characterization depends on the line length, configuration independent analysis, if an appropriate algorithm can be found, might give a more general characterization of false sharing. Other aspects that are now only characterized using configuration dependent analysis are spatial locality and conflict misses.

In our evaluation of the three techniques for reducing the communication cost in Chapter 7, we deliberately minimized capacity misses and focused on coherence misses. However, communication in DSM systems is due to both capacity and coherence misses. Although our suspicion is that the techniques using the Illinois protocol and the interconnect cache also reduce the cost of capacity misses, it is important to evaluate these techniques with applications that exhibit more capacity misses than were exhibited in our evaluation.

In Chapter 7, we have observed that the average miss latency is much higher than the latency of a simple miss that is satisfied from the memory without any contention. Although the average miss latency is this high mainly due to high rates of bursty communication and inefficiencies in memory allocation, the coherence protocol that was used may be blamed for a portion of this latency. For example, the coherence protocol uses speculative memory operations which increases the system traffic and may cause contention, particularly when the speculative data is not used. Another example is the "memory-less" coherence control that may increase the system traffic and miss latency when busy lines are frequently requested. We think that it is important to evaluate the effectiveness of these protocol features and investigate better approaches when they adversely affect system performance.

Many researchers are investigating approaches to exploit the increasing transistor budgets of a single chip. Some researchers are advocating using this budget to build a single-chip multiprocessor [HNO97]. Current research evaluates single-chip multiprocessor designs in a system that has only one processor chip. We think that this approach will get more momentum in the near future and that we should seriously consider how to incorporate multiple single-chip multiprocessors in a system. We think that there is now an opportunity to affect the design of such multiprocessors so that they can be efficiently interconnected in a scalable system.

APPENDICES

APPENDIX A

SYSTEM CONFIGURATION FILES

Sections A.1, A.2, and A.3 list the CDAT configuration files used in the raw comparison of Chapter 6. Sections A.4, A.5, and A.6 list the CDAT configuration files used in the normalized comparison of Chapter 6. Section A.7 lists the CCAT configuration file used to model the base system (**D2**) in Chapter 7.

A.1 DASH

```
number_of_nodes          4
processors_per_node      4
buses_per_node           1
bus_interconnection      na
mem_banks_per_node       1
mem_line_size            16      (bytes)
page_size                4       (Kbytes)
memory_on_bus            yes
snoopy_icc               yes
data_cache:
used                     yes
size                     256     (Kbytes)
lsize                    16      (bytes)
assoc                    1
instruction_cache:
used                     yes
size                     64      (Kbytes)
lsize                    16      (bytes)
assoc                    1
interconnect_cache:
used                     yes
size                     128     (Kbytes)
lsize                    16      (bytes)
assoc                    1
number_of_threads        16
thread_mapping:
thread 0 node 0 proc 0
thread 1 node 0 proc 1
thread 2 node 0 proc 2
thread 3 node 0 proc 3
thread 4 node 1 proc 0
thread 5 node 1 proc 1
thread 6 node 1 proc 2
thread 7 node 1 proc 3
```

```
thread 8 node 2 proc 0
thread 9 node 2 proc 1
thread 10 node 2 proc 2
thread 11 node 2 proc 3
thread 12 node 3 proc 0
thread 13 node 3 proc 1
thread 14 node 3 proc 2
thread 15 node 3 proc 3
cache_coherence_protocol    1      (memory-based directory)
allow_exclusive_remote      0
allow_update_tag            0
allow_cache_to_cache        1
shared_local_load           0
inv_on_supply               0
local_collects_acks         1
remote_updates_home         1
speculative_mem_ops         0
ack_wb                      1
illinois                    1
memory_allocation_policy    6      (First touch with code replication)
generate_miss_trace         0
generate_transaction_trace  0
clock_in_MHz                33
IPC                         1
FROM  TO    TYPE    TX(ns) RX(ns) BYTES
==============================================
NET   CCC   SHORT   0      125    12
NET   CCC   LONG    0      250    28
PRO   BUS   SHORT   250    125    8
PRO   BUS   LONG    188    125    16
MEM   BUS   LONG    188    125    16
BUS   PRO   SHORT   0      0      8
BUS   PRO   LONG    0      188    16
BUS   MEM   SHORT   0      0      8
BUS   MEM   LONG    0      188    16
BUS   CCC   SHORT   0      250    8
BUS   CCC   LONG    0      250    16
BUS   ICC   SHORT   0      0      8
BUS   ICC   LONG    0      188    16
CCC   NET   SHORT   0      375    12
CCC   NET   LONG    0      375    28
CCC   BUS   SHORT   0      125    8
CCC   BUS   LONG    188    188    16
CCC   DIR   SHORT   125    188    8
CCC   ICC   LONG    0      188    16
ICC   BUS   LONG    0      250    16
```

A.2 SPP

```
number_of_nodes             2
processors_per_node         8
buses_per_node              4
bus_interconnection         crossbar
mem_banks_per_node          4
mem_line_size               64
page_size                   4
memory_on_bus               no
```

```
snoopy_icc                   no
data_cache:
used                         yes
size                         1024
lsize                        32
assoc                        1
instruction_cache:
used                         yes
size                         1024
lsize                        32
assoc                        1
interconnect_cache:
used                         yes
size                         16384
lsize                        64
assoc                        1
number_of_threads            16
thread_mapping:
thread 0 node 0 proc 0
thread 1 node 0 proc 1
thread 2 node 0 proc 2
thread 3 node 0 proc 3
thread 4 node 0 proc 4
thread 5 node 0 proc 5
thread 6 node 0 proc 6
thread 7 node 0 proc 7
thread 8 node 1 proc 0
thread 9 node 1 proc 1
thread 10 node 1 proc 2
thread 11 node 1 proc 3
thread 12 node 1 proc 4
thread 13 node 1 proc 5
thread 14 node 1 proc 6
thread 15 node 1 proc 7
cache_coherence_protocol     2       (cache-based directory)
allow_exclusive_remote       0
allow_update_tag             0
allow_cache_to_cache         1
shared_local_load            1
inv_on_supply                0
local_collects_acks          1
remote_updates_home          1
speculative_mem_ops          0
ack_wb                       0
illinois                     0
memory_allocation_policy     6
generate_miss_trace          0
generate_transaction_trace   0
clock_in_MHz                 100
IPC                          1
FROM  TO    TYPE    TX    RX    BYTES
==============================================
NET   CCC   SHORT   1400  32    24
NET   CCC   LONG    1400  32    88
NET   CCC   MED     1400  32    40
NET   CCC   ELONG   1400  32    104
PRO   BUS   SHORT   40    30    8
PRO   BUS   LONG    40    80    32
MEM   CCC   LONG    140   32    32
```

BUS	PRO	SHORT	0	40	8
BUS	PRO	LONG	0	40	32
BUS	XBR	SHORT	0	32	8
BUS	XBR	LONG	0	128	32
CCC	NET	SHORT	32	27	24
CCC	NET	LONG	32	133	88
CCC	NET	MED	32	53	40
CCC	NET	ELONG	32	160	104
CCC	MEM	SHORT	0	0	4
CCC	MEM	LONG	0	140	32
CCC	XBR	SHORT	0	32	8
CCC	XBR	LONG	0	128	32
CCC	DIR	SHORT	32	140	8
CCC	ICC	SHORT	0	0	4
CCC	ICC	LONG	0	140	32
ICC	CCC	LONG	140	32	32
XBR	BUS	SHORT	0	30	8
XBR	BUS	LONG	0	90	32
XBR	CCC	SHORT	0	32	8
XBR	CCC	LONG	0	128	32

A.3 Origin

```
number_of_nodes            8
processors_per_node        2
buses_per_node             1
bus_interconnection        na
mem_banks_per_node         4
mem_line_size              128
page_size                  16
memory_on_bus              no
snoopy_icc                 no
data_cache:
used                       yes
size                       4096
lsize                      128
assoc                      2
instruction_cache:
used                       no
size                       1024
lsize                      64
assoc                      1
interconnect_cache:
used                       no
size                       128
lsize                      64
assoc                      1
number_of_threads          16
thread_mapping:
thread 0 node 0 proc 0
thread 1 node 0 proc 1
thread 2 node 1 proc 0
thread 3 node 1 proc 1
thread 4 node 2 proc 0
thread 5 node 2 proc 1
thread 6 node 3 proc 0
```

```
thread 7 node 3 proc 1
thread 8 node 4 proc 0
thread 9 node 4 proc 1
thread 10 node 5 proc 0
thread 11 node 5 proc 1
thread 12 node 6 proc 0
thread 13 node 6 proc 1
thread 14 node 7 proc 0
thread 15 node 7 proc 1
cache_coherence_protocol     1
allow_exclusive_remote       1
allow_update_tag             0
allow_cache_to_cache         1
shared_local_load            0
inv_on_supply                0
local_collects_acks          1
remote_updates_home          1
speculative_mem_ops          1
ack_wb                       1
illinois                     0
memory_allocation_policy     6
generate_miss_trace          0
generate_transaction_trace   0
clock_in_MHz                 195
IPC                          1
FROM  TO    TYPE    TX    RX    BYTES
==============================================
NET   CCC   SHORT   0     10    16
NET   CCC   LONG    0     10    144
PRO   BUS   SHORT   50    50    16
PRO   BUS   LONG    50    50    144
MEM   CCC   LONG    70    10    144
BUS   PRO   SHORT   0     20    16
BUS   PRO   LONG    0     50    144
BUS   CCC   SHORT   10    10    16
BUS   CCC   LONG    10    10    144
CCC   NET   SHORT   10    140   16
CCC   NET   LONG    10    140   144
CCC   MEM   SHORT   10    10    16
CCC   MEM   LONG    10    10    144
CCC   BUS   SHORT   10    10    16
CCC   BUS   LONG    10    10    144
CCC   DIR   SHORT   20    70    16
```

A.4 nDASH

```
number_of_nodes              4
processors_per_node          4
buses_per_node               1
bus_interconnection          na
mem_banks_per_node           4
mem_line_size                64
page_size                    4
memory_on_bus                yes
snoopy_icc                   yes
data_cache:
used                         yes
```

```
size                        4096
lsize                       64
assoc                       2
instruction_cache:
used                        no
size                        1024
lsize                       64
assoc                       1
interconnect_cache:
used                        yes
size                        4096
lsize                       64
assoc                       2
number_of_threads           16
thread_mapping:
thread 0 node 0 proc 0
thread 1 node 0 proc 1
thread 2 node 0 proc 2
thread 3 node 0 proc 3
thread 4 node 1 proc 0
thread 5 node 1 proc 1
thread 6 node 1 proc 2
thread 7 node 1 proc 3
thread 8 node 2 proc 0
thread 9 node 2 proc 1
thread 10 node 2 proc 2
thread 11 node 2 proc 3
thread 12 node 3 proc 0
thread 13 node 3 proc 1
thread 14 node 3 proc 2
thread 15 node 3 proc 3
cache_coherence_protocol    1
allow_exclusive_remote      0
allow_update_tag            0
allow_cache_to_cache        1
shared_local_load           0
inv_on_supply               0
local_collects_acks         1
remote_updates_home         1
speculative_mem_ops         0
ack_wb                      0
illinois                    1
memory_allocation_policy    6
generate_miss_trace         0
generate_transaction_trace  0
clock_in_MHz                200
IPC                         1
FROM  TO    TYPE    TX    RX    BYTES
==============================================
NET   CCC   SHORT   0     10    16
NET   CCC   LONG    0     10    80
PRO   BUS   SHORT   50    50    16
PRO   BUS   LONG    50    50    80
MEM   BUS   LONG    70    10    80
BUS   PRO   SHORT   0     20    16
BUS   PRO   LONG    0     50    80
BUS   MEM   SHORT   0     0     16
BUS   MEM   LONG    0     70    80
BUS   CCC   SHORT   10    10    16
```

```
BUS   CCC   LONG    10    10    80
BUS   ICC   SHORT   0     20    16
BUS   ICC   LONG    0     50    80
CCC   NET   SHORT   10    140   16
CCC   NET   LONG    10    140   80
CCC   BUS   SHORT   10    10    16
CCC   BUS   LONG    10    10    80
CCC   DIR   SHORT   20    70    16
CCC   ICC   LONG    10    70    80
ICC   BUS   LONG    50    50    80
```

A.5 nSPP

```
number_of_nodes             4
processors_per_node         4
buses_per_node              1
bus_interconnection         na
mem_banks_per_node          4
mem_line_size               64
page_size                   4
memory_on_bus               no
snoopy_icc                  no
data_cache:
used                        yes
size                        4096
lsize                       64
assoc                       2
instruction_cache:
used                        no
size                        1024
lsize                       64
assoc                       1
interconnect_cache:
used                        yes
size                        16384
lsize                       64
assoc                       1
number_of_threads           16
thread_mapping:
thread 0 node 0 proc 0
thread 1 node 0 proc 1
thread 2 node 0 proc 2
thread 3 node 0 proc 3
thread 4 node 1 proc 0
thread 5 node 1 proc 1
thread 6 node 1 proc 2
thread 7 node 1 proc 3
thread 8 node 2 proc 0
thread 9 node 2 proc 1
thread 10 node 2 proc 2
thread 11 node 2 proc 3
thread 12 node 3 proc 0
thread 13 node 3 proc 1
thread 14 node 3 proc 2
thread 15 node 3 proc 3
cache_coherence_protocol    2
allow_exclusive_remote      0
```

```
allow_update_tag            0
allow_cache_to_cache        1
shared_local_load           0
inv_on_supply               0
local_collects_acks         1
remote_updates_home         1
speculative_mem_ops         0
ack_wb                      0
illinois                    0
memory_allocation_policy    6
generate_miss_trace         0
generate_transaction_trace  0
clock_in_MHz                200
IPC                         1
FROM  TO    TYPE    TX    RX    BYTES
==============================================
NET   CCC   SHORT   0     10    24
NET   CCC   LONG    0     10    88
NET   CCC   MED     0     20    40
NET   CCC   ELONG   0     20    104
PRO   BUS   SHORT   50    50    16
PRO   BUS   LONG    50    50    80
MEM   CCC   LONG    70    10    80
BUS   PRO   SHORT   0     20    16
BUS   PRO   LONG    0     50    80
BUS   CCC   SHORT   10    10    16
BUS   CCC   LONG    10    10    80
CCC   NET   SHORT   10    140   24
CCC   NET   LONG    10    140   88
CCC   NET   MED     10    140   40
CCC   NET   ELONG   10    140   104
CCC   MEM   SHORT   10    10    16
CCC   MEM   LONG    10    70    80
CCC   BUS   SHORT   10    10    16
CCC   BUS   LONG    10    10    80
CCC   DIR   SHORT   20    70    16
CCC   ICC   SHORT   10    10    16
CCC   ICC   LONG    10    70    80
ICC   CCC   LONG    70    10    80
```

A.6 nOrigin

```
number_of_nodes             4
processors_per_node         4
buses_per_node              1
bus_interconnection         na
mem_banks_per_node          4
mem_line_size               64
page_size                   4
memory_on_bus               no
snoopy_icc                  no
data_cache:
used                        yes
size                        4096
lsize                       64
assoc                       2
```

```
instruction_cache:
used                         no
size                         1024
lsize                        64
assoc                        1
interconnect_cache:
used                         no
size                         4096
lsize                        64
assoc                        2
number_of_threads            16
thread_mapping:
thread 0 node 0 proc 0
thread 1 node 0 proc 1
thread 2 node 0 proc 2
thread 3 node 0 proc 3
thread 4 node 1 proc 0
thread 5 node 1 proc 1
thread 6 node 1 proc 2
thread 7 node 1 proc 3
thread 8 node 2 proc 0
thread 9 node 2 proc 1
thread 10 node 2 proc 2
thread 11 node 2 proc 3
thread 12 node 3 proc 0
thread 13 node 3 proc 1
thread 14 node 3 proc 2
thread 15 node 3 proc 3
cache_coherence_protocol     1
allow_exclusive_remote       1
allow_update_tag             0
allow_cache_to_cache         1
shared_local_load            0
inv_on_supply                0
local_collects_acks          1
remote_updates_home          1
speculative_mem_ops          1
ack_wb                       1
illinois                     0
memory_allocation_policy     6
generate_miss_trace          0
generate_transaction_trace   0
clock_in_MHz                 200
IPC                          1
FROM  TO    TYPE    TX    RX    BYTES
==============================================
NET   CCC   SHORT   0     10    16
NET   CCC   LONG    0     10    80
PRO   BUS   SHORT   50    50    16
PRO   BUS   LONG    50    50    80
MEM   CCC   LONG    70    10    80
BUS   PRO   SHORT   0     20    16
BUS   PRO   LONG    0     50    80
BUS   CCC   SHORT   10    10    16
BUS   CCC   LONG    10    10    80
CCC   NET   SHORT   10    140   16
CCC   NET   LONG    10    140   80
CCC   MEM   SHORT   10    10    16
CCC   MEM   LONG    10    70    80
```

```
CCC   BUS   SHORT   10    10    16
CCC   BUS   LONG    10    10    80
CCC   DIR   SHORT   20    70    16
```

A.7 D2

```
number_of_nodes                  16
processors_per_node              2
buses_per_node                   1
bus_interconnection              na
mem_banks_per_node               4
mem_line_size                    128
page_size                        4
memory_on_bus                    no
snoopy_icc                       no
data_cache:
used                             yes
size                             4096
lsize                            128
assoc                            2
instruction_cache:
used                             no
size                             1024
lsize                            128
assoc                            2
interconnect_cache:
used                             no
size                             4096
lsize                            128
assoc                            2
number_of_threads                32
thread_mapping:
thread 0 node 0 proc 0
thread 1 node 0 proc 1
thread 2 node 1 proc 0
thread 3 node 1 proc 1
thread 4 node 2 proc 0
thread 5 node 2 proc 1
thread 6 node 3 proc 0
thread 7 node 3 proc 1
thread 8 node 4 proc 0
thread 9 node 4 proc 1
thread 10 node 5 proc 0
thread 11 node 5 proc 1
thread 12 node 6 proc 0
thread 13 node 6 proc 1
thread 14 node 7 proc 0
thread 15 node 7 proc 1
thread 16 node 8 proc 0
thread 17 node 8 proc 1
thread 18 node 9 proc 0
thread 19 node 9 proc 1
thread 20 node 10 proc 0
thread 21 node 10 proc 1
thread 22 node 11 proc 0
thread 23 node 11 proc 1
thread 24 node 12 proc 0
```

```
thread 25 node 12 proc 1
thread 26 node 13 proc 0
thread 27 node 13 proc 1
thread 28 node 14 proc 0
thread 29 node 14 proc 1
thread 30 node 15 proc 0
thread 31 node 15 proc 1
cache_coherence_protocol        1
local_cache_coherence_protocol  2       (Directory)
allow_exclusive_remote          1
allow_update_tag                0
allow_cache_to_cache            1
exclusive_state_supported       1
upgrade_signal_supported        1
inv_on_supply                   0
local_collects_acks             1
remote_updates_home             1
speculative_mem_ops             1
ack_wb                          1
illinois                        0
memory_allocation_policy        6
generate_miss_trace             0
generate_transaction_trace      0
clock_in_MHz                    200
IPC                             2
ROB                             48
MEM_ROB                         16
OUTSTANDING_MISSES              4
```

FROM	TO	TYPE	TX (cycles)	RX (cycles)	Occupancy (cycles)	BYTES
PRO	BUS	S_REQ	18	0	2	16
PRO	BUS	L_REQ	18	0	34	144
PRO	BUS	S_RES	16	0	0	16
PRO	BUS	L_RES	38	0	34	144
MEM	CCC	L_RES	0	5	0	144
MEM	CCC	S_RES	0	5	0	16
BUS	PRO	S_RES	0	18	0	16
BUS	PRO	L_RES	0	0	0	144
BUS	PRO	S_REQ	0	0	0	16
CCC	MEM	S_REQ	5	0	20	16
CCC	MEM	L_REQ	5	0	20	144
CCC	MEM	S_RES	5	0	20	16
CCC	MEM	L_RES	5	0	20	144
BUS	CCC	S_REQ	0	5	0	16
BUS	CCC	L_REQ	0	5	0	144
BUS	CCC	S_RES	0	5	0	16
BUS	CCC	L_RES	0	5	0	144
CCC	BUS	S_REQ	5	0	2	16
CCC	BUS	L_REQ	5	0	34	144
CCC	BUS	S_RES	5	0	2	16
CCC	BUS	L_RES	5	0	32	144
CCC	DIR	S_REQ	1	1	0	16
CCC	NET	S_REQ	6	2	4	16
CCC	NET	L_REQ	6	2	36	144
CCC	NET	S_RES	6	2	4	16
CCC	NET	L_RES	6	2	36	144
DIR	CCC	S_RES	1	1	0	16
NET	CCC	S_REQ	8	7	0	16

```
NET   CCC   L_REQ   8        7        0          144
NET   CCC   S_RES   8        7        0          16
NET   CCC   L_RES   8        7        0          144
NET   NET   S_REQ   8        2        4          16
NET   NET   L_REQ   8        2        36         144
NET   NET   S_RES   8        2        4          16
NET   NET   L_RES   8        2        36         144
END
number_of_routers 8
Routing matrix:
To
from    0   1   2   3   4   5   6   7
==========================================
0       0   0   0   2   0   4   4   6
1       1   1   3   1   5   1   7   5
2       2   0   2   2   6   4   2   6
3       1   3   3   3   5   7   7   3
4       4   0   0   2   4   4   4   6
5       1   5   3   1   5   5   7   5
6       2   0   6   2   6   4   6   6
7       1   3   3   7   5   7   7   7
END
Router of each node:
node  router
===============
0     0
1     0
2     1
3     1
4     2
5     2
6     3
7     3
8     4
9     4
10    5
11    5
12    6
13    6
14    7
15    7
END
```

BIBLIOGRAPHY

BIBLIOGRAPHY

[AAD+93] T. Asprey, G. Averill, E. DeLano, R. Mason, B. Weiner, and J. Yetter. Performance Features of the PA7100 Microprocessor. *IEEE Micro*, pages 22–34, June 1993.

[AB97] G. Astfalk and T. Brewer. An Overview of the HP/Convex Exemplar Hardware. Tech. paper, Hewlett-Packard Co., June 1997. http://www.hp.com/wsg/tech/technical.html.

[Aba96] G. Abandah. Tools for Characterizing Distributed Shared Memory Applications. Technical Report HPL–96–157, HP Laboratories, December 1996.

[Aba97] G. Abandah. Characterizing Shared-Memory Applications: A Case Study of the NAS Parallel Benchmarks. Technical Report HPL–97–24, HP Laboratories, January 1997.

[ABC+95] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B-J. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proc. 22th ISCA*, pages 2–13, June 1995.

[ABP94] G. Astfalk, T. Brewer, and G. Palmer. Cache Coherence in the Convex MPP. Tech. paper, Hewlett-Packard Co., February 1994. http://www.hp.com/wsg/tech/technical.html.

[AD96] G. Abandah and E. Davidson. Modeling the Communication Performance of the IBM SP2. In *Proc. 10th IPPS*, pages 249–257, April 1996.

[AD98a] G. Abandah and E. Davidson. A Comparative Study of Cache-Coherent Nonuniform Memory Access Systems. In *High Performance Computing Systems and Applications*. Kluwer Academic Publishers, May 1998. 12th Ann. Int'l Symp. High Performance Computing Systems and Applications (HPCS'98).

[AD98b] G. Abandah and E. Davidson. Characterizing Distributed Shared Memory Performance: A Case Study of the Convex SPP1000. *IEEE Trans. Parallel and Distributed Systems*, 9(2):206–216, February 1998.

[AD98c] G. Abandah and E. Davidson. Configuration Independent Analysis for Characterizing Shared-Memory Applications. In *Proc. 12th IPPS*, March 1998.

[AD98d] G. Abandah and E. Davidson. Effects of Architectural and Technological Advances on the HP/Convex Exemplar's Memory and Communication Performance. In *Proc. 25th ISCA*, June 1998.

[Amd67] G. Amdahl. Validity of Single-Processor Approach to Achieving Large-Scale Computing Capability. In *Proc. AFIPS*, pages 483–485, 1967.

[ASH86] A. Agarwal, R. Sites, and M. Horwitz. ATUM: A New Technique for Capturing Address Traces Using Microcode. In *Proc. 13th ISCA*, pages 119–127, June 1986.

[B+94] D. Bailey et al. The NAS Parallel Benchmarks. Technical Report RNR-94-07, NASA Ames Research Center, March 1994.

[BA97] T. Brewer and G. Astfalk. The Evolution of the HP/Convex Exemplar. In *Digest of papers, COMPCON'97*, pages 81–86, February 1997.

[BCF96] W. Bryg, K. Chan, and N. Fiduccia. A High-Performance, Low-Cost Multiprocessor Bus for Workstations and Midrange Servers. *Hewlett-Packard J.*, 47(1):18–24, February 1996.

[BD94] E. Boyd and E. Davidson. Communication in the KSR1 MPP: Performance Evaluation Using Synthetic Workload Experiments. In *Int'l Conf. on Supercomputing*, pages 166–175, 1994.

[BDCW91] E. Brewer, C. Dellarocas, A. Colbrook, and W. Weihl. PROTEUS: A High-Performance Parallel-Architecture Simulator. Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, September 1991.

[Boy95] E. Boyd. *Performance Evaluation and Improvement of Parallel Application on High Performance Architectures*. PhD thesis, University of Michigan, 1995.

[Bre95] T. Brewer. A Highly Scalable System Utilizing up to 128 PA-RISC Processors. In *Digest of papers, COMPCON'95*, pages 133–140, March 1995.

[CFKA90] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. Directory-Based Cache Coherence in Larg-Scale Multiprocessors. *IEEE Computer*, pages 12–24, June 1990.

[CHK+96] K. Chan, C. Hay, J. Keller, G. Kurpanek, F. Schumacher, and J. Zheng. Design of the HP PA 7200 CPU. *Hewlett-Packard J.*, 47(1):25–33, February 1996.

[CLR94] S. Chandra, J. Larus, and A. Rogers. Where is Time Spent in Message-Passing and Shared-Memory Programs? In *ASPLOS-VI*, pages 61–73, October 1994.

[CMM+95] M. Calzarossa, L. Massari, A. Merlo, M. Pantano, and D. Tessera. Medea: A Tool for Workload Characterization of Parallel Systems. *IEEE Parallel and Distributed Technology*, 3(4):72–80, Winter 1995.

[CSG98] D. Culler, J. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1998.

[CSV+97] S. Chodnekar, V. Srinivasan, A. Vaidya, A. Sivasubramaniam, and C. Das. Towards a Communication Characterization Methodology for Parallel Applications. In *Proc. HPCA-3*, pages 310–319, 1997.

[CXp93] CONVEX Computer Corp. *CXpa Reference Manual*, second edition, March 1993. Order No. DSW–253.

[Den68] P. Denning. Working Set Model for Program Behavior. *Commun. ACM*, 11(6):323–333, 1968.

[EC84] J. Emer and D. Clark. A Characterization of Processor Performance in the VAX-11/780. In *Proc. 11th ISCA*, pages 301–309, June 1984.

[EKKL90] S. Eggers, D. Keppel, E. Koldinger, and H. Levy. Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor. *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 37–47, May 1990.

[FBR93] S. Frank, H. Burkhardt, and J. Rothnie. KSR1: Bridging the Gap Between Shared Memory and MPPs. In *Digest of papers, COMPCON'93*, pages 285–294, February 1993.

[Fis95] P. Fishwick. *Simulation Model Design and Execution: Building Digital Worlds*. Prentice Hall, 1995.

[For94] MPI Forum. MPI: A Message-Passing Interface. Technical Report CS/E 94-013, Department of Computer Science, Oregon Graduate Institute, March 94.

[FW78] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Proc. 10th ACM Symp. on Theory of Computing*, pages 114–118, 1978.

[FW97] B. Falsafi and D. Wood. Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA. In *Proc. 24th ISCA*, pages 229–240, June 1997.

[FX] Digital FX!32 Home Page. http://www.service.digital.com/fx32/.

[Gal96] M. Galles. Scalable Pipelined Interconnect for Distributed Endpoint Routing: The SGI SPIDER Chip. In *HOT Interconnects IV*, pages 141–146, August 1996.

[GGH91] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors. In *ASPLOS-IV*, pages 245–257, 1991.

[GGJ+90] 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.

[GH92] S. Goldschmidt and J. Hennessy. The Accuracy of Trace-Driven Simulations of Multiprocessors. Technical Report CSL-TR-92-546, Stanford University, September 1992.

[Hew94] Hewlett-Packard. *PA-RISC 1.1 Architecture and Instruction Set*, third edition, February 1994.

[HLK97] C. Hristea, D. Lenoski, and J. Keen. Measuring Memory Hierarchy Performance of Cache-Coherent Multiprocessors Using Micro Benchmarks. In *Supercomputing*, November 1997.

[HNO97] L. Hammond, B. Nayfeh, and K. Olukotun. A Single-Chip Multiprocessor. *Computer*, 30(9):79–85, September 1997.

[Hun95] D. Hunt. Advanced Performance Features of the 64-bit PA-8000. In *Digest of papers, COMPCON'95*, pages 123–128, March 1995.

[Hwa93] K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, 1993.

[KCZ+94] G. Kurpanek, K. Chan, J. Zheng, E. DeLano, and W. Bryg. PA7200: A PA-RISC Processor with Integrated High Performance MP Bus Interface. In *Digest of papers, COMPCON'94*, pages 375–382, February 1994.

[KG96] S. Kaxiras and J. Goodman. The GLOW Cache Coherence Protocol Extension for Widely Shared Data. In *Proc. Int'l Conf. on Supercomputing*, pages 35–43, May 1996.

[KHW91] Y. Kim, M. Hill, and D. Wood. Implementing Stack Simulation for Highly-Associative Memories. In *Proc. ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, pages 212–213, 1991.

[KR92] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice-Hall, 1992.

[KS93] R. Kessler and J. Schwarzmeier. Cray T3D: A New Dimension for Cray Research. In *Digest of papers, COMPCON'93*, pages 176–182, February 1993.

[Lam79] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Trans. on Computers*, C-29(9):241–248, September 1979.

[LC96] T. Lovett and R. Clapp. STiNG: A CC-NUMA Computer for the Commercial Marketplace. In *Proc. 23rd ISCA*, pages 308–317, 1996.

[LD93] S. Leutenegger and D. Dias. A Modeling Study of the TPC-C Benchmark. In *Proc. of ACM SIGMOD*, pages 22–31, 1993.

[LL97] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proc. 24th ISCA*, pages 241–251, 1997.

[LLG+92] D. Lenoski, J. Laudon, K. Gharachorloo, W-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH Multiprocessor. *Computer*, 25:63–79, March 1992.

[LW95] D. Lenoski and W-D. Weber. *Scalable Shared-Memory Multiprocessing*. Morgan Kaufmann, 1995.

[McC95a] J. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Technical Committee on Computer Architecture Newsletter*, December 1995.

[MCC+95b] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tool. *Computer*, 28(11):37–46, November 1995.

[MD98] A. Moga and M. Dubois. The Effectiveness of SRAM Network Caches in Clustered DSMs. In *Proc. HPCA-4*, February 1998.

[MS96] L. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *Proc. USENIX'96 Ann. Technical Conf.*, pages 279–294, January 1996.

[MSSAD93] W. Mangione-Smith, T-P. Shih, S. Abraham, and E. Davidson. Approaching a Machine-Application Bound in Delivered Performance on Scientific Code. *IEEE Proc.*, 81(8):1166–1178, August 1993.

[NAB+95] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, B. Radke, and S. Vishin. The S3.mp Scalable Shared Memory Multiprocessor. In *Proc. ICPP*, pages I.1–I.10, August 1995.

[O2K96] SGI. *Performance Tuning for the Origin2000 and Onyx2*, 1996. http://techpubs.sgi.com/library/.

[PP84] M. Papamarcos and J. Patel. A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proc. 11th ISCA*, pages 348–354, May 1984.

[PS96] S. Perl and R. Sites. Studies of Windows NT Performance Using Dynamic Execution Traces. In *Proc. of the USENIX 2nd Symp. on Operating Systems Design and Implementation*, October 1996.

[PTM96] J. Protic, M. Tomasevic, and V. Milutinovic. Distributed Shared Memory: Concepts and Systems. *IEEE Parallel and Distributed Technology*, pages 63–79, Summer 1996.

[R1097] MIPS Technologies Inc. *MIPS R10000 Microprocessor User's Manual*, January 1997. Version 2.0.

[RAN+93] D. Reed, R. Aydt, R. Noe, P. Roth, K. Shields, B. Schwartz, and L. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proc. Scalable Parallel Libraries Conf.*, pages 104–113. IEEE Computer Society, 1993.

[RHWG95] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete Computer Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology*, Fall 1995.

[RSG93] E. Rothberg, J. Singh, and A. Gupta. Working Sets, Cache Sizes, and Node Granularity Issues for Large-Scale Multiprocessors. In *Proc. 20th ISCA*, pages 14–25, 1993.

[SA95] K. Shaw and G. Astfalk. Four-State Cache-Coherence in the Convex Exemplar System. Internal memo, Convex Computer Corp., October 1995. http://www.hp.com/wsg/tech/technical.html.

[SB95] S. Saini and D. Bailey. NAS Parallel Benchmark Results 12–95. Technical Report NAS–95–021, NASA Ames Research Center, December 1995.

[SCI93] IEEE Computer Society. *IEEE Standard for Scalable Coherent Interface (SCI)*, August 1993. IEEE Std 1596-1992.

[SE94] A. Srivastava and A. Eustace. ATOM: A System for Building Customized Program Analysis Tools. Technical Report 94/2, DEC WRL, March 1994.

[SGC93] R. Saavedra, R. Gaines, and M. Carlton. Micro Benchmark Analysis of the KSR1. In *Supercomputing*, pages 202–213, November 1993.

[Smi91] M. Smith. Tracing with Pixie. ftp document, Center for Integrated Systems, Stanford University, April 1991.

[SPE] SPEC CPU95 Benchmarks Results. See the Standard Performance Evaluation Corp., web page http://www.spec.org/.

[SRG94] J. Singh, E. Rothberg, and A. Gupta. Modeling Communication in Parallel Algorithms: A Fruitful Interaction between Theory and Systems? In *Proc. Symp. Parallel Algorithms and Architectures*, pages 189–199, 1994.

[SS86] P. Sweazy and A. Smith. A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus. In *Proc. 13th ISCA*, pages 414–423, June 1986.

[SS95] R. Saavedra and A. Smith. Measuring Cache and TLB Performance and Their Effect on Benchmark Runtimes. *IEEE Trans. on Computers*, 44(10):1223–1235, October 1995.

[SWG92] J. Singh, W-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[Tom95] K. Tomko. *Domain Decomposition, Irregular Application, and Parallel Computers*. PhD thesis, University of Michigan, 1995.

[TPC] Transaction Processing Performance Council Home Page. http://www.tpc.org/.

[TPC92] Transaction Processing Performance Council. *TPC Benchmark C, Standard Specification*, August 1992.

[TPC95] Transaction Processing Performance Council. *TPC Benchmark D, Decision Support, Standard Specification*, May 1995.

[WCNSH94] E. Welbon, C. Cha-Nui, D. Shippy, and D. Hicks. The POWER2 Performance Monitor. *IBM Journal of Research and Development*, 38(5):545–554, September 1994.

[Wol96] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[WOT+95] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodology Considerations. In *Proc. 22nd ISCA*, pages 24–36, 1995.

[ZK95] D. Zucker and A. Karp. RYO a Versatile Instruction Instrumentation Tool for PA–RISC. Technical Report CSL–TR–95–658, Stanford University, January 1995.

[ZT97] Z. Zhang and J. Torrellas. Reducing Remote Conflict Misses: NUMA with Remote Cache versus COMA. In *Proc. HPCA-3*, pages 272–281, 1997.