# Configuration Independent Analysis for Characterizing Shared-Memory Applications

Gheith A. Abandah Edward S. Davidson

Advanced Computer Architecture Laboratory, EECS Department
University of Michigan
*gabandah,davidson@eecs.umich.edu*

September 9, 1997

**Abstract**

Characterizing shared-memory applications provides insight to design efficient systems, and provides awareness to identify and correct application performance bottlenecks. Configuration *dependent* analysis is often used to simulate detailed application traces on a particular hardware model. The communication traffic and computation workload generated by the application trace is used as a characterization of this application. This paper demonstrates that configuration *independent* analysis is a useful tool to characterize shared-memory applications. Configuration independent analysis characterizes inherent application characteristics that do not change from one configuration to another. While configuration dependent analysis is repeated for each target configuration, configuration independent analysis is only performed once. Moreover, configuration independent analysis does not require developing models for the target configurations and is faster than detailed simulation. However, configuration dependent analysis directly provides more information about specific configurations. A combination of the two analysis types constitutes a comprehensive and efficient methodology for characterizing shared-memory applications. In this paper, we 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 illustrate the advantages and limitations of this approach by analyzing eight case-study benchmarks from the scientific and commercial domains and interpreting the results.

# 1 Introduction

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. Moreover, architects of scalable shared-memory systems face more design dimensions including node and system organization, target communication latency and bandwidth, and memory caching, coherency, and consistency protocols. They need to select a design that best fits their objectives for the target applications.

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.

We have developed a methodology for characterizing shared-memory applications and evaluating scalable shared-memory system design alternatives. This methodology is based on a set of flexible tools that enable collecting and analyzing data, instruction, and I/O stream traces of shared memory applications. This tool set provides configuration dependent and configuration independent analysis.

Configuration dependent analysis uses a model of the target system configuration to simulate the application trace and predict its performance on the target system. The configuration of a multiprocessor specifies the way that processors are clustered in a hierarchy, the interconnection topology, coherence protocols, cache configuration, and other system properties that may change from one system to another. Configuration independent analysis, on the other hand, uses configuration independent analysis of a parallel execution trace to extract the inherent application characteristics.

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

This paper demonstrates that configuration independent analysis is a useful approach to characterize 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 both configuration dependent and independent analysis, it is possible to have an efficient and comprehensive methodology for characterizing shared-memory applications.

In this paper, we describe the tools and algorithms used in configuration independent analysis of shared-memory applications, and use them to characterize eight benchmarks from the Stanford SPLASH-2, NAS Parallel Benchmarks (NPB), and Transaction Processing Performance Council (TPC) application suites.

In the rest of this paper, Section 2 describes some important shared-memory application characteristics and why knowledge of them is useful to architects and software engineers. Section 3 describes our shared-memory application characterization approach. Section 4 describes the eight case-study benchmarks. Section 5 contains eight subsections, where each subsection describes how we used configuration independent analysis to characterize a particular aspect of the benchmarks, it states the advantages and disadvantages of this approach, and interprets the results of the eight benchmark characterizations. Finally, Section 6 describes some related work and Section 7 presents conclusions.

# 2 Shared-Memory Application Characteristics

This paper addresses eight characteristics of shared-memory applications:

- *General characteristics* of the application, including 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* [1] of an application in an execution interval is the number of distinct memory locations accessed in this interval. 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 the 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 the working set, e.g. by segmenting a matrix computation into blocks [2].

- The amount of *concurrency* available in an application influences how well 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 5.3 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 clustered in a 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 5.4 presents a classification of the communication patterns of shared-memory applications. 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 enables the programmer to avoid expensive patterns.

- *Communication variation over time* is as important as characterizing overall communication volume. Communication can be uniform, bursty, random, periodic, or exhibit 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.

- *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 are communication accesses. 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 Characterization Approach

This section describes our methodology for characterizing shared-memory applications and evaluating scalable shared-memory system design alternatives. This methodology is based on four flexible tools that enable collecting and analyzing detailed traces of shared memory applications as shown in Figure 1. There is one tool for trace collection called the *Shared-Memory Application Instrumentation Tool* (SMAIT), two tools for trace analysis: the *Configuration Dependent Analysis Tool* (CDAT) and the *Configuration Independent Analysis Tool* (CIAT), and one tool for characterizing event time distribution called *Time Distribution Analysis Tool* (TDAT).

In Figure 1, a shared-memory multiprocessor is used to execute and analyze instrumented application codes. However, the analysis tools can also accept trace files. SMAIT supports *execution-*

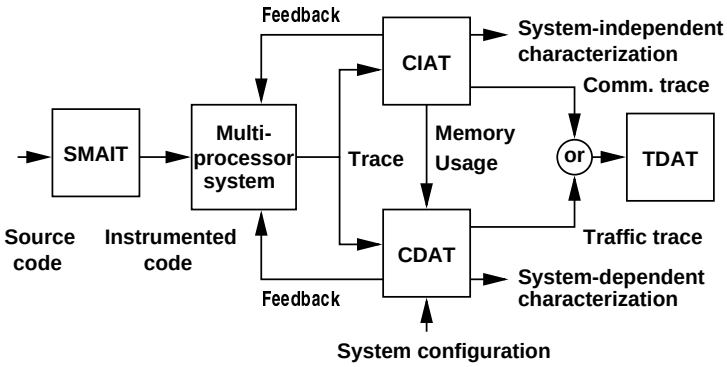


Figure 1: A comprehensive shared-memory application analysis methodology.

*driven analysis* by (i) piping the traces directly to either CDAT or CIAT, instead of generating trace files, and (ii) accepting feedback to control the execution timing on the traced multiprocessor according to CDAT's simulated configuration or CIAT's analysis model. Execution-driven analysis enables analyzing longer execution periods by piping to eliminate the need to store huge trace files and feedback to avoid the indeterministic behavior of some applications.

Usually, we first use CIAT to characterize the application characteristics outlined in Section 2. Then we use CDAT to characterize other aspects or to find the application performance and generated traffic on a particular system configuration. CDAT is used to characterize things like cache misses and false sharing that depend on configuration parameters, e.g. cache size and line width. Since this paper concentrates on CIAT, other tools are treated briefly. However, an interested reader can refer to [3].

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

SMAIT has two parts: a perl script program for instrumenting PA–RISC [4] assembly language files (based on a tool called RYO [5]), 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 trace records for the instrumented memory instructions, taken branches, and synchronization and I/O calls.

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 uses the memory usage file for some policies of 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 traffic trace. The characterization reported by these tools is used to support application tuning, early-design of scalable shared-memory systems, parameterizing synthetic work-load generators, comparing alternative design options, and investigating new design approaches.

CIAT uses a model similar to the PRAM model [6] 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 Section 5).

CIAT assumes that the application has one or more *execution phases* where each phase has its own properties. CIAT identifies serial and parallel phases automatically and identifies user-defined phases delimited by special marker records. CIAT performs analysis per phase, reports phase characteristics at the end of each phase, and reports the aggregate characteristics of all phases at the end.

# 4 Applications

We have analyzed Radix, FFT, LU, and Cholesky from SPLASH-2 [7], CG and SP from NPB [8], and TPC benchmarks C and D [9, 10]. SPLASH-2 consists of 8 applications and 4 computational kernels drawn from scientific, engineering, and graphics computing. NPB are 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 [11]). The TPC benchmarks are intended to compare commercial database platforms. The following is a short description of the eight benchmarks.

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

**Cholesky** is a kernel that factors a sparse matrix using blocked Cholesky factorization 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 1 shows the order of the matrix and the number of iterations of the two problem sizes.

**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 paper 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

Table 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.

| | *Problem Size I* | | *Problem Size II* | |
|---|---|---|---|---|
| Benchmark | 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 | $256 \times 256$ | 80 | $512 \times 512$ | 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 2: The SPP1600 host configuration.

| 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 |

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 paper is for a 2.8 Giga instruction trace representing the third phase of Query 3 where most of the query's 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 [12]. A comprehensive characterization of TPC-D's queries is beyond the scope of this paper.

Table 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 one order of magnitude more instructions than problem size I.

The SPLASH-2 benchmarks were developed in 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 [13] implementation of NPB which is written in Fortran. The performance of an earlier version of this implementation is reported in [14]. However, to get a general characterization of these benchmarks, we undid some of the Exemplar-specific optimizations. The six scientific benchmarks were instrumented, compiled, and analyzed on a 4-node Exemplar SPP1600 multiprocessor. Table 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 a conservative trace scheduling algorithm that does not violate process synchronization ordering. Namely, CIAT uses 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 observed when the trace was collected. CIAT sorts the slices into a list according to their start times and schedules 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 by CIAT until B completes.

Although this conservative scheduling correctly captures inter-process communication, its conservative ordering of the slices lengthens the execution time, and consequently we cannot accurately characterize the concurrency, communication variation over time, and communication slack of the TPC benchmarks.

# 5 Characterization Results

The following eight subsections are devoted to the eight targeted characteristics of shared-memory applications. 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.

## 5.1 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. 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, TPC-C 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 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 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 indicates that the scientific benchmarks use little synchronization; among them, CG has the fewest

Table 3: General characteristics of problem size II using 32 processors. Characteristics of problem size I is given in parenthesis if significantly different.

| | 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 instructions in (M) | 110 | 480 | 540 | 2,000 | 2,000 | 190,000 | 1,000 | 2,900 |
| | (22) | (27) | (69) | (770) | (130) | (600) | | |
| Memory Instructions | 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 | 1185 | 1600 | 0 | 0 |
| | (11) | (7) | (35) | (4) | (735) | (400) | | |
| No. of locks | 442 | 32 | 32 | 72,026 | 0 | 0 | $7.9 \times 10^5$ | $5.3 \times 10^5$ |
| | (442) | (32) | (32) | (54,419) | (0) | (0) | | |

Table 4: Disk I/O in TPC-C and TPC-D.

| | 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 |

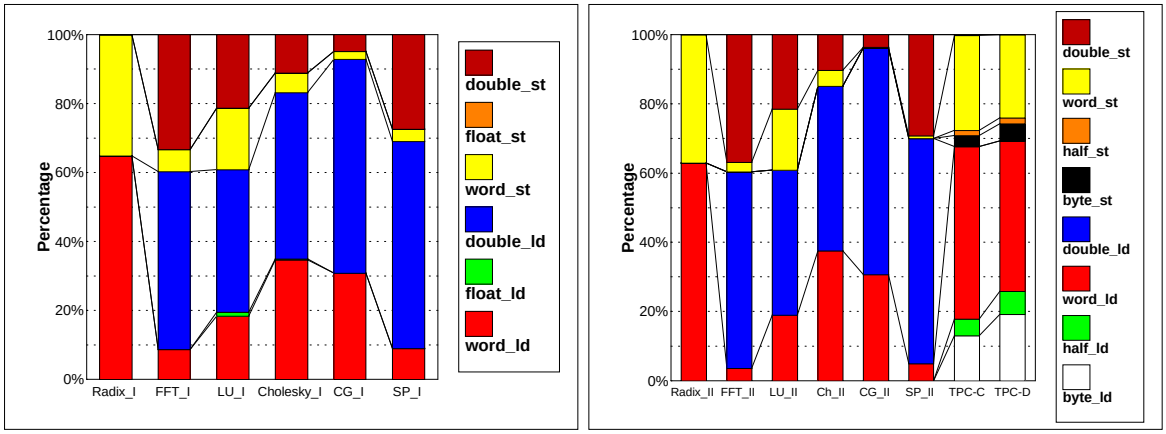


Figure 2: Percentage of the memory instructions according to the instruction type (load or store) and type of data accessed (byte, half-word, word, float, or double).

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 synchronization rates higher than unperturbed execution. In order to minimize the effect of this behavior on our analysis, we ignore all accesses to synchronization variables when characterizing communication and sharing.

Table 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 2 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. 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 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 times, 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 a procedure entry for a procedure that is called a fixed number of times.

## 5.2 Working Sets

The size of an application's working set is sometimes characterized by conducting multiple simulation experiments using a fully-associative cache with LRU replacement policy [15, 7]. 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 is expensive to simulate a fully-associative cache with LRU replacement. 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.

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 $\infty$ 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

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

**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 $s$.

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

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

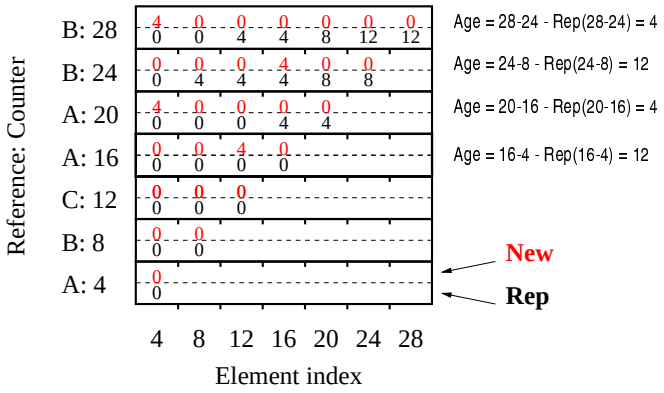


Figure 3: An example illustrating how the repeated bytes vectors **Rep** and **New** are updated to find the access ages of the sequence of word accesses (A, B, C, A, A, B, B). Each row shows the contents vector elements $4, 8, 12, \ldots$.

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 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. We use two optimizations to reduce this overhead: (i) 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 $\pm K$ confidence in the access age. (ii) 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 5% of the accumulations done with optimization (i) only. With these two optimizations, the overhead of characterizing the working sets is about 30% of CIAT's total analysis time.

Figure 4 shows the cumulative distribution function of the access age using 32 processors, ignoring infinite ages. The left graph is for problem size I and the right is for problem size II. 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 4 indicates that TPC-D has better

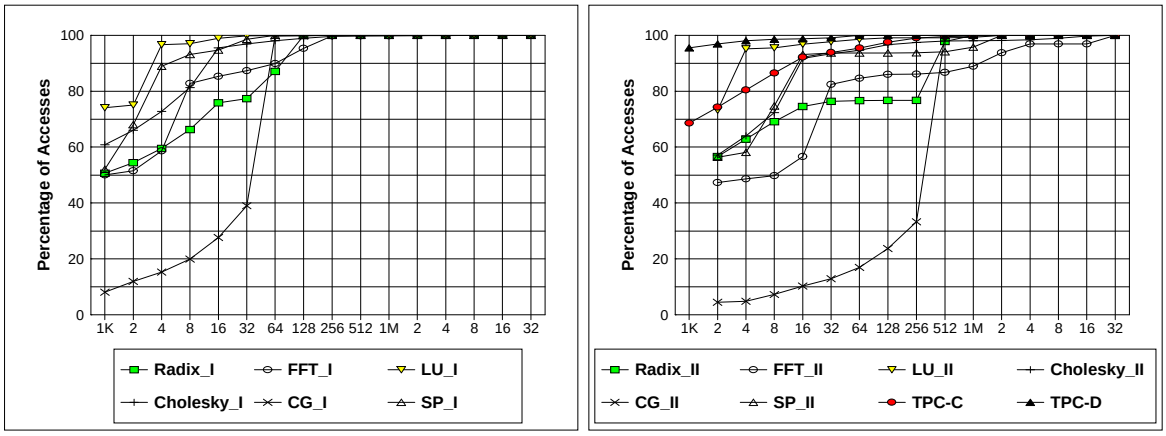


Figure 4: The cumulative distribution function of the access age. A point $(x, y)$ indicates that $y$% of the accesses have access age of $x$ bytes or less.

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-D's performance may also be improved by larger caches because the operating system interleaves multiple processes per processor.

## 5.3 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 5 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 5 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, $\text{Busy}(1)$ is the busy time for the basic work when using one processor, $\text{Busy}(p)$ is the total busy time summed over the $p$ processors ($\text{Busy}(p) - \text{Busy}(1)$

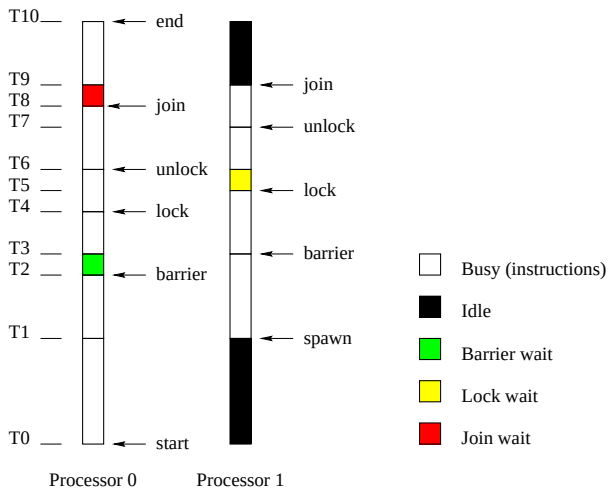


Figure 5: Execution profile of a parallel application running on a perfect system using 2 processors.

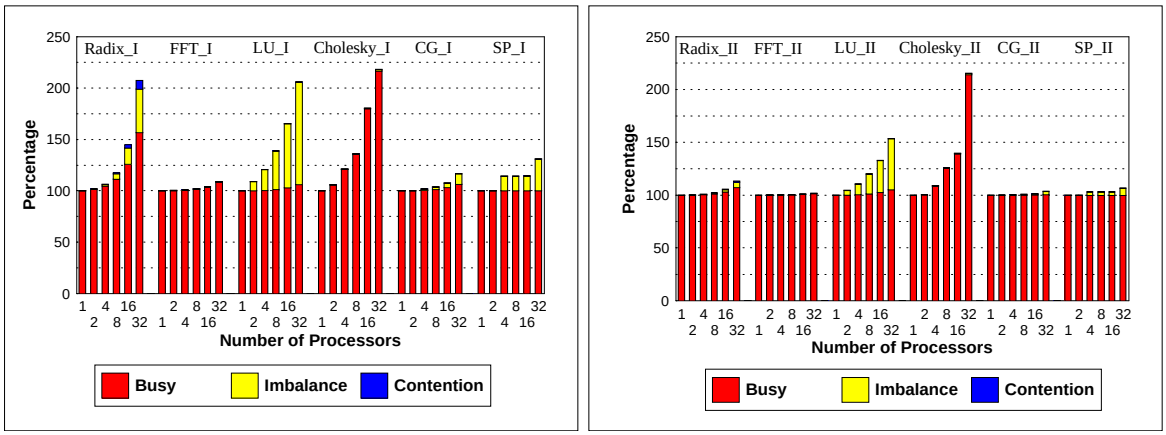


Figure 6: Concurrency and load balance in the parallel phase.

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 [16], 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 6 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, it has negligible contention time, unlike Radix, due to its relatively small critical regions. 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 reported by CIAT.

## 5.4 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 [17]. 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 into four main patterns and 8 variants to provide a thorough characterization. CIAT reports the volumes of the following communication 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 consumes 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 **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 previous writer that is not one of the readers, (iii) a new writer that is one of the readers, and (iv) same writer as previous writer that 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, it 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 **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, 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 is 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 7 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 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

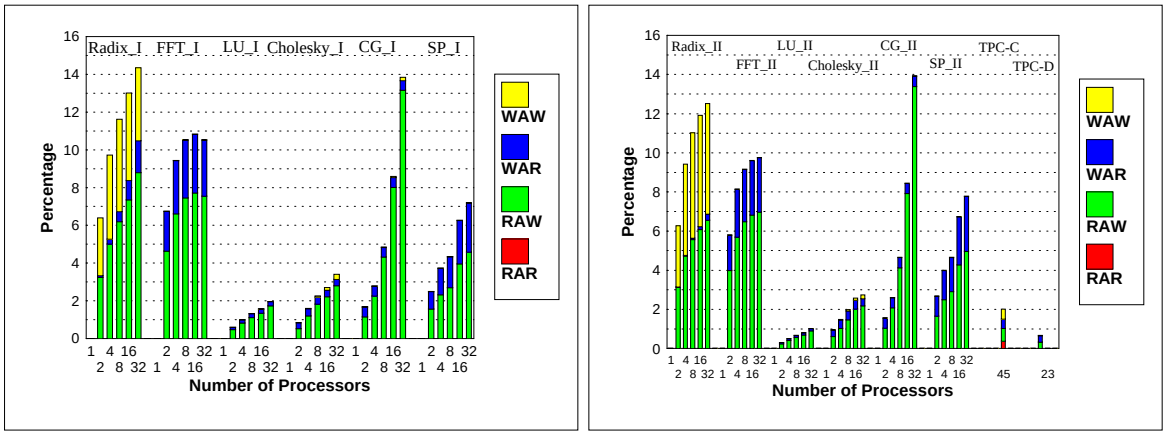


Figure 7: Percentage of the four classes of communication accesses among all data accesses, as a function of the number of processors for the two problem sizes.

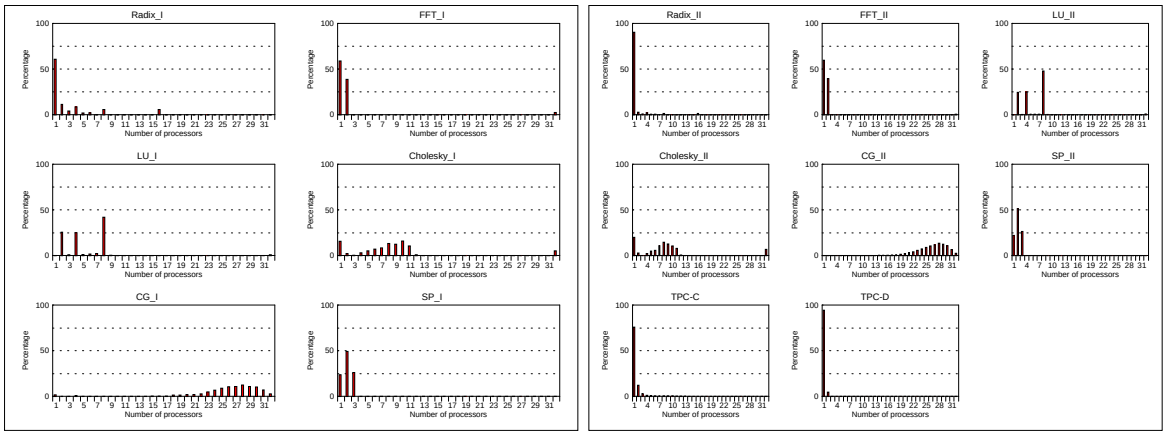


Figure 8: RAW sharing degree for 32 processors.

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 8 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 9 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 WAR 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

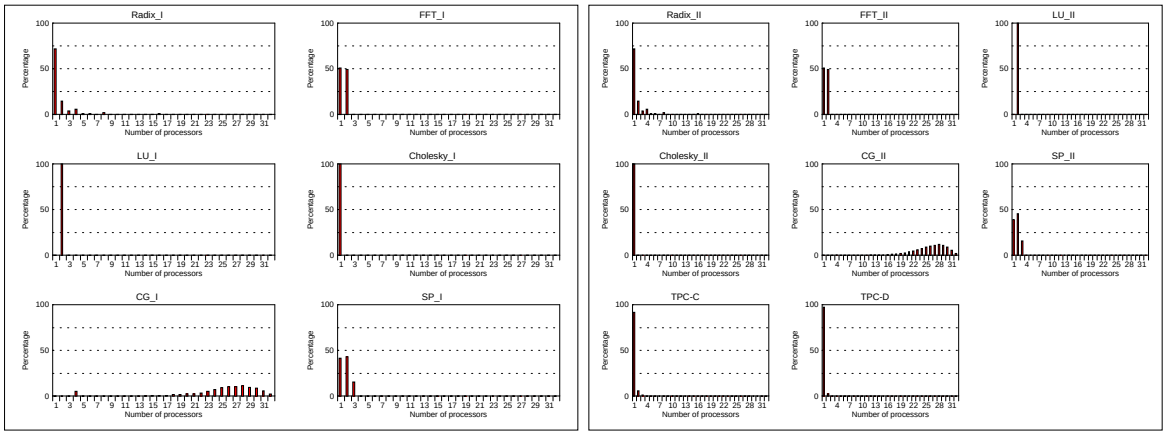


Figure 9: WAR invalidation degree for 32 processors.

and invalidation degrees indicates that most of the communication occurs in a producer-consumer pattern. However, in TPC-C the consumer generally updates the communicated value, while in TPC-D only one producer generally updates each location.

## 5.5 Communication Variation Over Time

TDAT is used to analyze CIAT's communication event trace which has one record per 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 benchmarks 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 10 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

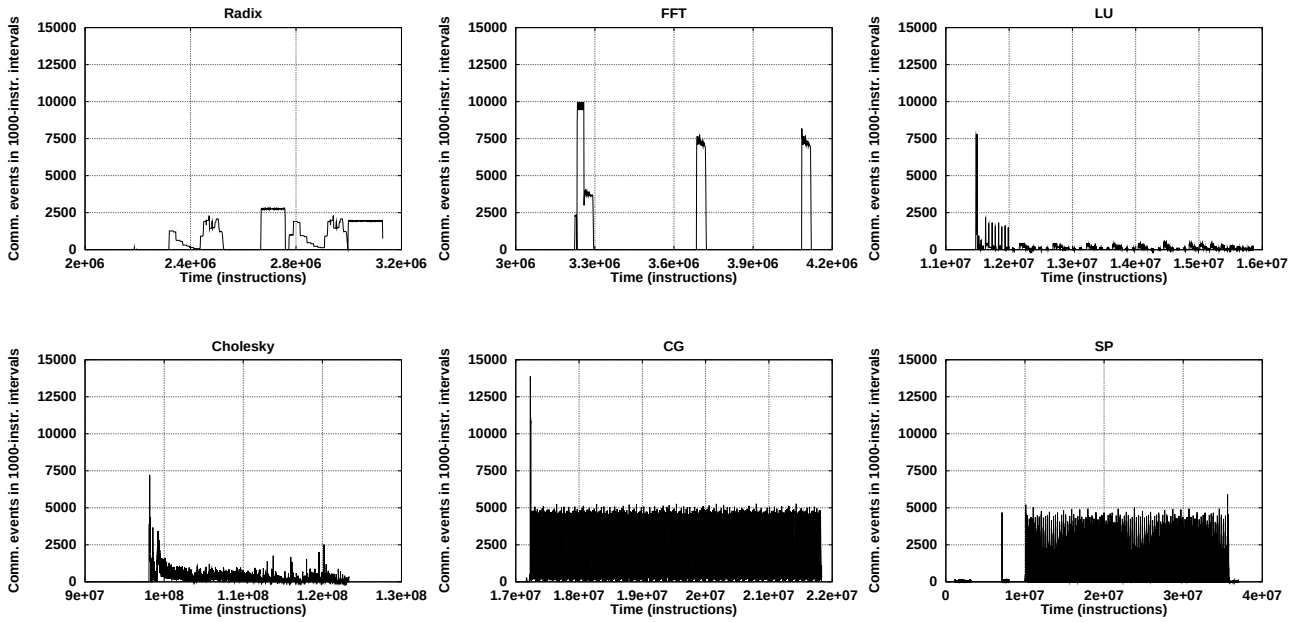


Figure 10: Number of communication events over time (32 processors, problem size I).

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 about 20,000 instructions; SP has a more complex behavior with a cycle of about 200,000.

Figure 11 shows the distribution function for 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 these two applications may suffer most from contention in systems with limited communication bandwidth.

## 5.6 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 12 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 90% 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. Nevertheless, the six benchmarks each have large enough slack to make prefetching rewarding.

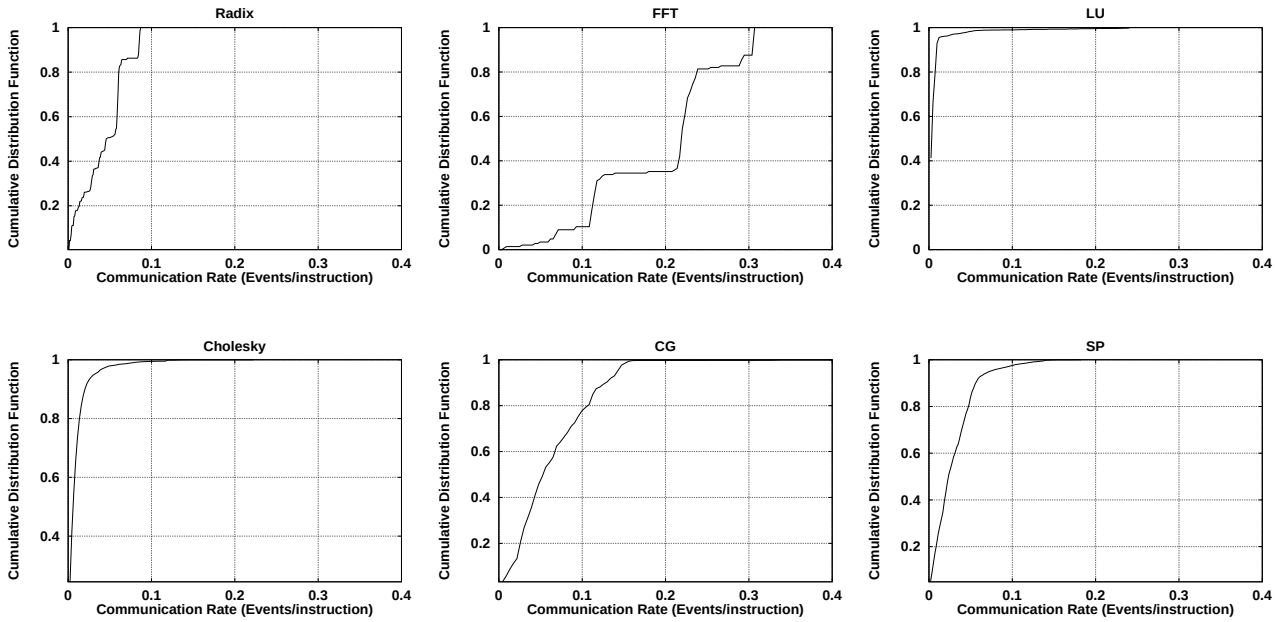


Figure 11: Communication rate (32 processors, problem size I).

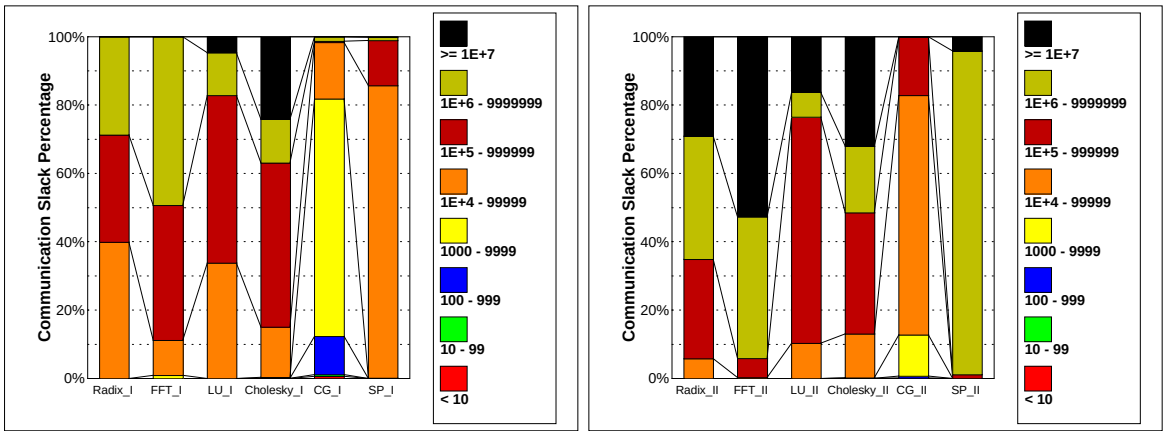


Figure 12: Communication slack distribution for 32 processors.

## 5.7 Communication Locality

CIAT characterizes the communication locality by reporting the number of communication events for processor pairs. 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. For each Processor $j$'s RAW access to a location stored into by Processor $i$.
2. For each Processor $j$'s WAW access of locations previously stored into by Processor $i$.
3. For each Processor $j$ invalidation by a WAR access of Processor $i$.

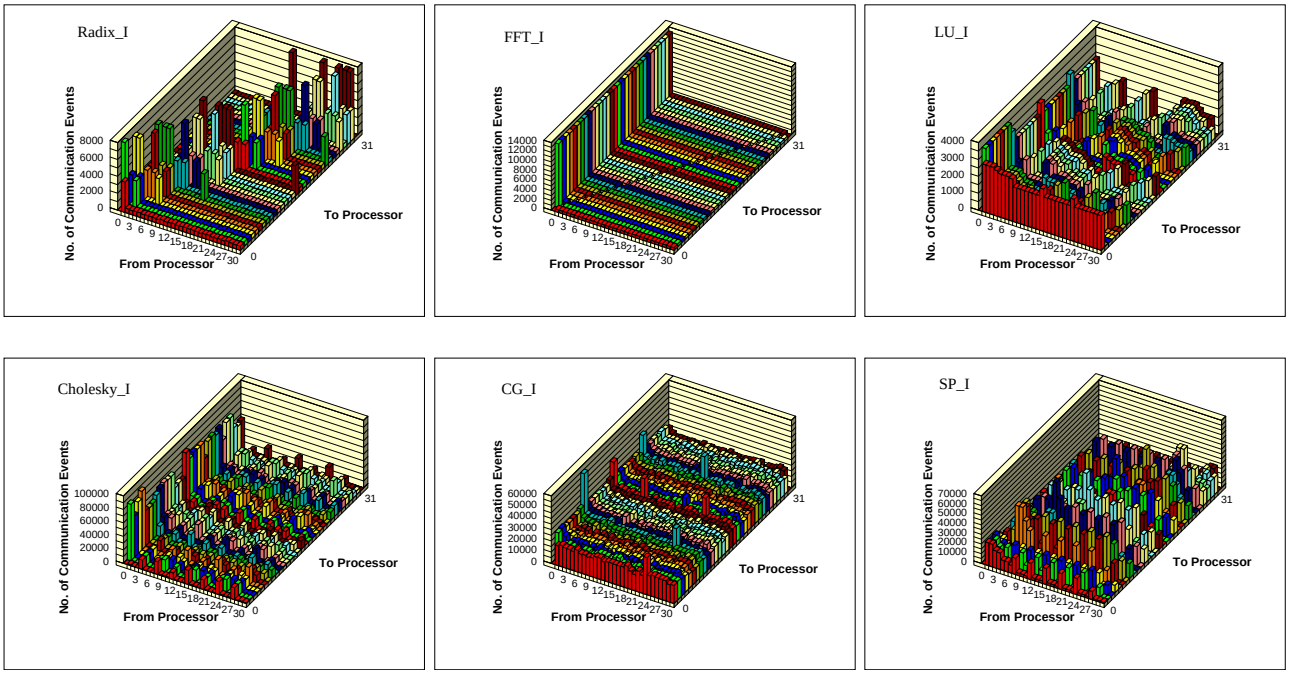


Figure 13: Number of communication events per processor pair (32 processors, problem size I).

Figures 13 and 14 show the reported communication matrix using 32 processors for problem size I and II, respectively. 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 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 $m$th group of 8 processors, where $m = i \pmod 4$.

In TPC-C, there is some communication among the first 16 processes (on processors 0 through 15). Apparantly, each is resposible 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 that 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.

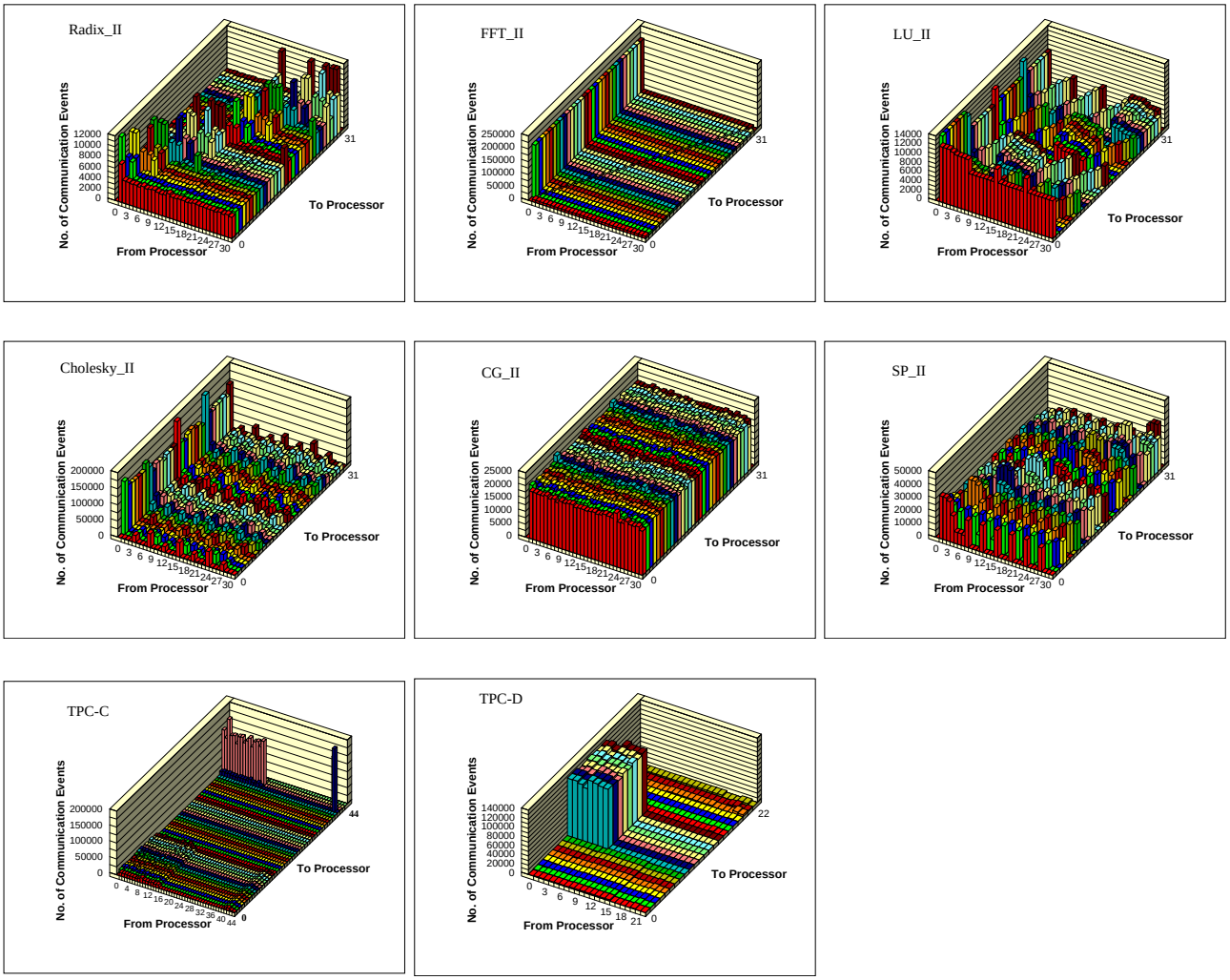


Figure 14: Number of communication events per processor pair (32 processors, problem size II).

## 5.8 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 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 Radix, SP, FFT, and LU data locations are shared. TPC-C's shared percentage is 15% and TPC-D's is 12%. 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 Cholesky, and somewhat in Radix.

Figure 16 shows the number of private and shared data accesses normalized to the number of data accesses using one processor. In CG, Cholesky, TPC-C, and TPC-D, shared locations are more intensely accessed than private locations. For example, using 32 processors to solve problem

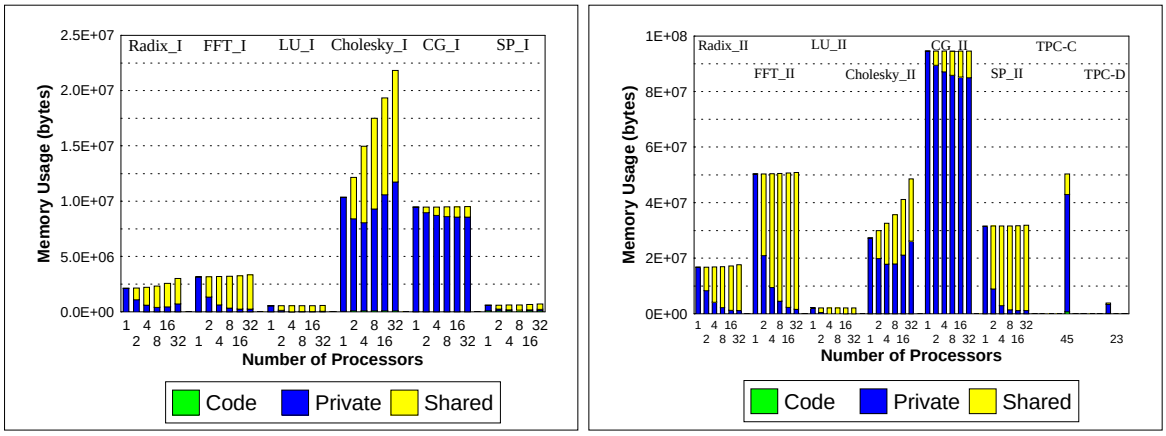


Figure 15: The size of code, private data, and shared data locations.

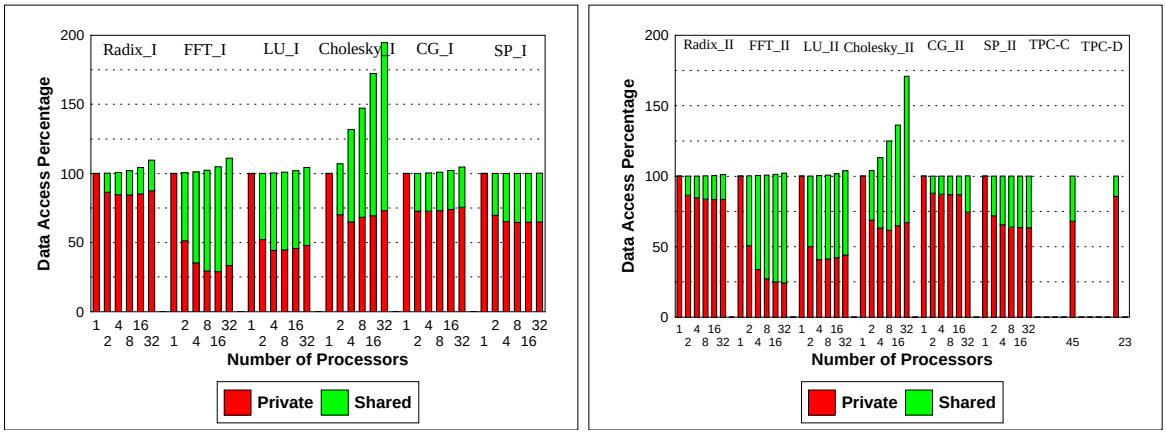


Figure 16: Number of private and shared data accesses normalized to the number of data accesses using one processor.

size II, 10% of CG's data locations are shared and these are referenced by 25% of all the data accesses. However, in Radix, SP, FFT, and LU, 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.

# 6 Related Work

Available parallel performance analysis tools have mainly been developed for analyzing message-passing applications, e.g. Pablo [18], Medea [19], and Paradyn [20]. Nevertheless, there is 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 [17]. They suggested developing general-purpose simulation tools to obtain empirical information for supporting

the design of parallel algorithms.

There are several studies that combine source-code analysis with configuration dependent analysis to characterize shared-memory applications [21, 15, 7]. Woo et al. have characterized several aspects of the SPLASH-2 suite of parallel applications [7]. Their characterization includes load balance, working sets, communication to computation ratio, system traffic, and sharing. They used execution-driven simulation with the Tango Lite [22] tracing tool. In order to capture some of the fundamental properties of SPLASH-2, they adjusted model parameters between low and high values.

Chandra et al. also used simulation to characterize the performance of a collection of applications [23]. Their main objective was to analyze where time is spent in message-passing versus shared-memory programs. Perl and Sites [24] 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 [25].

Leutenegger and Dias [26] 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.

# 7 Conclusions

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 a designer to evaluate the application performance on a particular system design.

In this paper, 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. Additionally, CIAT 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 in a form that can readily be exploited by application and system designers.

# Acknowledgments

This research was initiated in 1996 while Gheith Abandah was a research intern at HP Labs in Palo Alto, California. We would like to thank all the people Abandah 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 trace.

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

# References

[1] P. J. Denning, "Working set model for program behavior," *Commun. ACM*, vol. 11, no. 6, pp. 323–333, 1968.

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

[3] G. Abandah, "Tools for characterizing distributed shared memory applications," Tech. Rep. HPL–96–157, HP Laboratories, Dec. 1996.

[4] Hewlett-Packard, *PA-RISC 1.1 Architecture and Instruction Set*, third ed., Feb. 1994.

[5] D. F. Zucker and A. H. Karp, "RYO: a versatile instruction instrumentation tool for PA–RISC," Technical Report CSL–TR–95–658, Stanford University, Jan. 1995.

[6] S. Fortune and J. Wyllie, "Parallelism in random access machines," in *Proc. Tenth ACM Symposium on Theory of Computing*, pp. 114–118, 1978.

[7] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodology considerations," in *Proc. of the 22nd ISCA*, pp. 24–36, 1995.

[8] D. Bailey *et al.*, "The NAS parallel benchmarks," Technical Report RNR-94-07, NASA Ames Research Center, Mar. 1994.

[9] Transaction Processing Performance Council, *TPC Benchmark C, Standard Specification*, Aug. 1992.

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

[11] G. Abandah, "Characterizing shared-memory applications: A case study of the NAS parallel benchmarks," Tech. Rep. HPL–97–24, HP Laboratories, Jan. 1997.

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

[13] T. Brewer, "A highly scalable system utilizing up to 128 PA-RISC processors," in *Digest of papers, COMPCON'95*, pp. 133–140, Mar. 1995.

[14] S. Saini and D. H. Bailey, "NAS parallel benchmark results 12–95," Tech. Rep. NAS–95–021, NASA Ames Research Center, Dec. 1995.

[15] E. Rothberg, J. Singh, and A. Gupta, "Working sets, cache sizes, and node granularity issues for large-scale multiprocessors," in *Proc. of the 20th ISCA*, pp. 14–25, 1993.

[16] G. M. Amdahl, "Validity of single-processor approach to achieving large-scale computing capability," in *Proc. AFIPS Conf.*, (Reston, VA.), pp. 483–485, 1967.

[17] J. Singh, E. Rothberg, and A. Gupta, "Modeling communication in parallel algorithms: A fruitful interaction between theory and systems?," in *Proc. Symp. on Parallel Algorithms and Architectures*, pp. 189–199, 1994.

[18] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera, "Scalable Performance Analysis: The Pablo Performance Analysis Environment," in *Proc. Scalable Parallel Libraries Conf.*, pp. 104–113, 1993.

[19] M. Calzarossa, L. Massari, A. Merlo, M. Pantano, and T. Daniele, "Medea: A tool for workload characterization of parallel systems," *IEEE Parallel and Distributed Technology*, vol. 3, pp. 72–80, Winter 1995.

[20] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall, "The Paradyn parallel performance measurement tool," *Computer*, vol. 28, pp. 37–46, Nov. 1995.

[21] J. Singh, W.-D. Weber, and A. Gupta, "SPLASH: Stanford parallel applications for shared memory," *Computer Architecture News*, vol. 20, pp. 5–44, Mar. 1992.

[22] S. Goldschmidt and J. Hennessy, "The accuracy of trace-driven simulations of multiprocessors," Tech. Rep. CSL-TR-92-546, Stanford University, Sept. 1992.

[23] S. Chandra, J. R. Larus, and A. Rogers, "Where is time spent in message-passing and shared-memory programs," in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 61–73, 1994.

[24] S. E. Perl and R. L. Sites, "Studies of Windows NT performance using dynamic execution traces," in *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA* (USENIX, ed.), pp. 169–183, 1996.

[25] S. Chodnekar, V. Srinivasan, A. Vaidya, A. Sivasubramaniam, and C. Das, "Towards a communication characterization methodology for parallel applications," in *Proc. of HPCA-3*, pp. 310–319, 1997.

[26] S. Leutenegger and D. Dias, "A modeling study of the TPC-C benchmark," in *Proc. of ACM SIGMOD*, pp. 22–31, 1993.