

# Configuration Independent Analysis for Characterizing Shared-Memory Applications

Gheith A. Abandah

Edward S. Davidson

Advanced Computer Architecture Laboratory, University of Michigan, Ann Arbor  
{gabandah,davidson}@eecs.umich.edu

## Abstract

*This paper demonstrates that configuration independent analysis of shared-memory applications is useful tool to characterize inherent application characteristics that do not change from one machine configuration to another. Although traditional configuration dependent analysis, or simulation, may directly provide more information about performance on specific configurations, it requires developing a machine model and repeating the analysis for each target configuration. A judicious combination of the two constitutes a comprehensive and efficient methodology. In this paper, we use configuration independent analysis to characterize seven aspects of application behavior: general characteristics; working sets; concurrency; communication patterns, variation over time, and locality; and sharing behavior. Case-studies of eight scientific and commercial benchmarks are used to illustrate the advantages and limitations of this approach.*

## 1. Introduction

Computer architects increasingly rely on application characteristics for insight in designing cost-effective systems. Programmers can use them to identify performance bottlenecks and improve the performance of their applications. 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 configuration dependent and configuration independent analysis tools that enable collecting and analyzing data, instruction, and I/O stream traces of shared memory applications.

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. In contrast, configuration independent analysis extracts the inherent application characteristics directly from a parallel execution trace.

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

This paper demonstrates that configuration independent analysis is useful for characterizing several important aspects of shared-memory applications, and is more efficient than configuration dependent analysis in doing so. It can also capture some application properties that are easily missed by configuration dependent analysis on a fixed configuration. Judicious use of both configuration dependent and independent analysis is an efficient and comprehensive means of characterization.

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. 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 seven subsections, where each subsection describes how we used configuration independent analysis to characterize a particular aspect of the benchmarks, states the advantages and disadvantages of this approach, and interprets the results of the eight characterizations. Section 6 presents conclusions.

## 2. Shared-Memory Application Characteristics

This paper addresses seven 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 size is a measure of the application's temporal locality, which affects its cache performance. Working set characterization is important for cache size selection for a new system and, during application performance tuning, for adjusting the working set size to better fit the existing cache.
- The amount of *concurrency* available in an application influences how well application performance scales as more processors are used. High concurrency implies the potential to utilize a large number of processors efficiently. The amount of available concurrency in an application provides insight in selecting machine size. Characterizing the factors that adversely affect an application's concurrency helps guide efforts to improve its 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. Since coherence misses and traffic are a function of the communication patterns and the system configuration, characterizing the volume of various patterns is important. A successful system design efficiently supports the common communication patterns of the target applications. Conversely, pattern characterization enables application tuning to avoid expensive patterns.

- *Communication variation over time* is as important as characterizing overall communication volume, as it enables identifying communication-intensive program segments. Knowledge of the average and peak communication rates helps specify appropriate bandwidths in the system interconnect.
- *Communication locality* is a measure of the distance between the communicating processors. Some applications have local communication where a processor tends to communicate with its near neighbors, and others have uniform communication where a processor communicates evenly with all other processors. Knowledge of communication locality is useful in selecting the system organization and its interconnection topology, and helps assign threads to physical processors.
- The *sharing behavior* of an application refers to which memory locations are shared and how. A *real shared* memory location is accessed by multiple processors during the program execution; only one processor accesses each *private* location. Characterizing the behavior of accesses to shared data helps in mapping application data into the shared and private spaces of memory so as to reduce access time.

### 3. Characterization Approach

This section describes our methodology for characterizing shared-memory applications and evaluating scalable shared-memory system design alternatives. Four flexible tools enable collecting and analyzing detailed traces of shared memory applications as shown in Figure 1. The *Shared-Memory Application Instrumentation Tool* (SMAIT) instruments a code for trace collection, the *Configuration Dependent Analysis Tool* (CDAT) and the *Configuration Independent Analysis Tool* (CIAT) perform trace analysis, and the *Time Distribution Analysis Tool* (TDAT) produces event time distributions.

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-driven analysis* by producing code that (i) generates a trace file or pipes trace information directly to either CDAT or CIAT and (ii) accepts feedback to control the execution timing on the host multiprocessor according to CDAT's simulated configuration or CIAT's analysis model. Execution-driven analysis enables analyzing longer traces by using piping to eliminate storage for huge trace files and feedback to control 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. This paper concentrates on CIAT, more detail on other tools is in [2].

CIAT analyzes application properties that do not change from one configuration to another, thus relieving CDAT from repeating

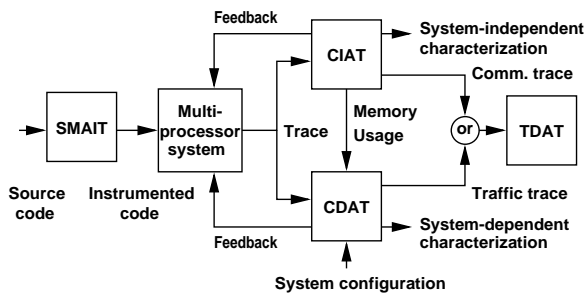


Figure 1. A comprehensive shared-memory application analysis methodology.

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 [3] assembly language files (based on a tool called RYO [4]), and a run-time library that is linked with the instrumented program. The perl script program replaces some PA-RISC instructions with calls to 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 this 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 can also analyze CDAT's traffic trace. The characterizations reported by these tools are used to support application tuning, early-stage design of scalable shared-memory systems, parameterizing synthetic work-load generators, comparing alternative design options, and investigating new design approaches.

CIAT, as in the PRAM model [5], assumes that  $p$  processors can execute  $p$  instructions concurrently and each instruction takes a fixed time. CIAT thus 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.

CIAT assumes that the application has one or more *execution phases*, each with its own properties. CIAT identifies serial and parallel phases automatically and identifies user-defined phases delimited by special markers. CIAT reports phase characteristics at the end of each phase, and also reports aggregate characteristics over all phases.

### 4. Applications

We have analyzed Radix, FFT, LU, and Cholesky from SPLASH-2 [6], CG and SP from NPB [7], and TPC benchmarks C and D [8, 9]. The SPLASH-2 benchmarks are drawn from scientific, engineering, and graphics computing. NPB mimic the computation and data movement characteristics of large-scale computational fluid dynamic applications. TPC-C is an on-line transaction processing benchmark.

TPC-D is a decision support application benchmark that performs complex and long-running queries against large databases. 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 high disk I/O and communication rates [10]. A comprehensive characterization of TPC-D's queries is beyond the scope of this paper.

Table 1 shows the set of problem sizes analyzed in this study. For conducting comparisons, smaller problem sizes were also analyzed, as reported in [11]. The scientific benchmarks were analyzed on 1 to 32 processors using the default options.

We analyzed the Convex Exemplar [12] implementation of NPB. 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 running SPP-UX 4.2.

The six scientific benchmarks are multi-threaded, each starts with a serial *initialization phase*, then  $p$  threads are spawned to run on the  $p$  available processors in the main *parallel phase*. Then these threads join and only Thread 0 remains active in the *wrap-up phase* to do validation and reporting. Reported scientific application characteristics, unless otherwise stated, are for the parallel phase using 32 processors.

**Table 1. Sizes of the problems analyzed.**

Benchmark	Problem Size	Total Instructions
Radix	2 M integers	0.63 G
FFT	1 M points	0.51 G
LU	512 × 512	0.57 G
Cholesky	tk29.O file	2.13 G
CG	14, 000	2.43 G
SP	64 <sup>3</sup>	189 G
TPC-C	16 users	1.0 G
TPC-D	1 GB data base	2.8 G

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. The TPC-C and TPC-D traces are composed of trace files for 45 and 23 processes, respectively. The OS serves active processes by performing context switching on the limited number of server CPUs. To capture the characteristics of each process, CIAT runs each process trace on a dedicated processor.

These TPC traces include a record for each user-space memory instruction, taken branch, system call, and synchronization instruction (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; although this correctly captures inter-process communication, its conservative ordering of the slices lengthens execution time, and consequently does not accurately characterize the concurrency and communication variation over time of the TPC benchmarks.

## 5. Characterization Results

The following seven subsections discuss the configuration independent analysis used, advantages and disadvantages of this approach, and the results of characterizing the eight benchmarks for one of the targeted characteristics of shared-memory applications.

### 5.1. General Characteristics

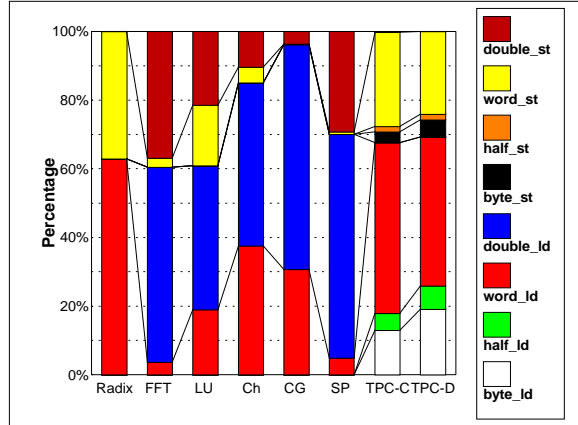
Each memory instruction accesses one or more consecutive locations, where each "location" in CIAT is one byte, e.g. a load-word instruction accesses four locations. CIAT maintains a hash table with an entry (status bits and access information) for each accessed location. One status bit, the code bit, is set when the location is accessed by an instruction fetch. CIAT reports the # of set code bits as "touched code size" and # of clear code bits as "touched data size," as shown in Table 2. While Cholesky and SP have about 85 KB of touched code, the other four scientific kernels have less. The TPC benchmarks touch hundreds of code kilobytes in the traced period.

LU has the smallest data size; CG, the largest. TPC-C's data size is larger than TPC-D's, mainly due to differences in their disk access patterns. TPC-C processes independent transactions that generate short random-access disk reads and writes, and uses a large disk cache in memory to improve disk access time. TPC-D queries generate long sequential disk reads with little data reuse; consequently it uses few small memory buffers to temporarily hold and process disk reads in chunks.

Table 2 also shows five aggregate characteristics of the parallel phase: # of executed instructions, % of memory instructions, avg. instructions executed per taken branch, and # of barriers and locks. CG, with the highest percentage of memory instructions and the largest data size due to its simple reduction operations on long vectors, can potentially stress the processor cache. The six scientific benchmarks have more instructions between taken branches than the TPC benchmarks. Infrequent branching is typical of scientific applications which spend most of their time in large-body

**Table 3. 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, or double).**

loops with one backward branch. Table 2 also 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, trace collection perturbs execution and the traces have synchronization rates higher than unperturbed execution. In order to minimize the effect of this perturbation, we ignore all accesses to synchronization variables when characterizing communication and sharing.

Table 3 shows some disk I/O statistics of the TPC traces. TPC-C accesses relatively little data per disk 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 % of load and store accesses to byte, half-word (2 bytes), word (4 bytes), and double (double-precision floating-point) data. While the byte and half-word percentage 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 remaining benchmarks average about 2 loads per store, i.e. typically two operands per 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 many load-word instructions to find non-zero element indices of its sparse matrix.

### 5.2. Working Sets

An application's working set size is often characterized by simulations to obtain the miss ratios of fully-associative LRU caches of various sizes [13, 6]. The knees of a graph of cache miss ratio vs. cache size then determine the working set sizes; a knee at size  $C$  indicates a working set of size  $\leq C$ . This is a time consuming procedure. Furthermore, it does not differentiate between coherence and capacity misses, and may over-estimate the working set size when cache lines are larger than the sizes of individually ac-

Table 2. General characteristics using 32 processors.

	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
No. of instructions in (M)	110	480	540	2,000	2,000	190,000	1,000	2,900
Memory Instructions	29%	29%	40%	26%	51%	35%	36%	48%
Instructions/taken branches	33	16	25	24	21	68	10	10
No. of barriers	11	7	67	4	1185	1600	0	0
No. of locks	442	32	32	72,026	0	0	$7.9 \times 10^5$	$5.3 \times 10^5$

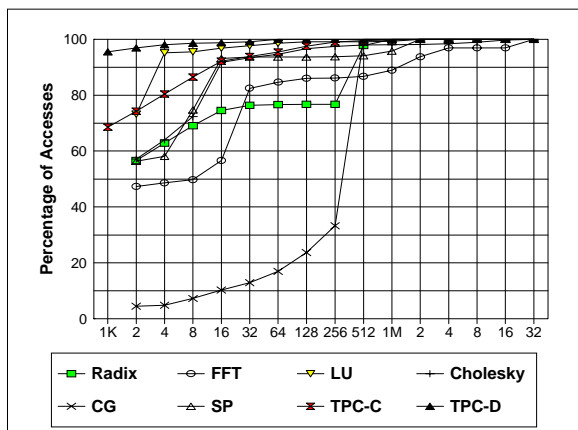


Figure 3. The cumulative distribution function of the access age. A point  $(x, y)$  indicates that  $y\%$  of the accesses have access age of  $x$  bytes or less.

cessed data elements.

CIAT characterizes the inherent working sets of an application in one experiment using the *access age* of each load and store access. The access age of an instruction accessing location  $x$  is the total number of distinct (1-byte) locations accessed between this access and the previous access of  $x$ , inclusively. Access age is set to  $\infty$  for the first access of  $x$ . For example, the word access sequence (A, B, C, A, A, B, B) has access ages  $(\infty, \infty, \infty, 12, 4, 12, 4)$ .

The access ages yield the miss ratio of a fully-associative LRU cache with a line size equal to 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. CIAT uses an efficient algorithm to find the access age, as described in [11].

Figure 3 shows the cumulative access age distribution using 32 processors, ignoring infinite ages. A point  $(x, y)$  indicates that  $y\%$  of the accesses have access age  $\leq x$  bytes. A distinguishable rise to a plateau beginning at  $x$  bytes indicates that the respective benchmark has an important working set of size  $\leq x$ .

CG has one important working set of size  $\leq 512$  KB. CG may incur frequent capacity misses when the cache size is smaller than this working set size. FFT reaches important plateaus at 32 KB and 4 MB. TPC-D has better temporal locality than TPC-C; TPC-C performance could be improved by increasing the cache size beyond 64 KB.

### 5.3. Concurrency

Concurrency is often characterized by measuring the execution time for a number of machine sizes; a good speedup indicates high concurrency. CIAT characterizes concurrency by measuring the time (in instructions) that processors spend executing instructions vs. waiting at synchronization points. Figure 4 shows a 2-processor execution profile of an application running on a perfect system (with fixed memory access time and zero synchronization

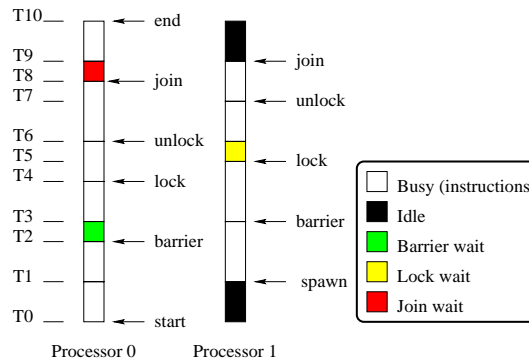


Figure 4. Execution profile of a parallel application running on a perfect system using 2 processors.

overhead). The concurrency is reflected in the busy time relative to the total time of both processors (P1 and P2).

Figure 4 demonstrates three factors that adversely affect concurrency: *serial fraction*, *load imbalance*, and *resource contention*. At the application start (T0), P0 is busy in a serial phase and P1 is idle. At T1, a parallel phase begins by spawning an execution thread on P1. P1 joins at the end of the parallel phase (T9) where P0 starts a final serial phase. Some parallel phase load imbalance is visible as P0's synchronization barrier wait at T2 (due to P1 having more busy cycles (work) than P0) and its join wait at T8 (due to P0 reaching the join point earlier than P1). P1 fails to acquire a lock that protects a shared resource at T5, and waits until P0 releases this lock at T6 before it enters the critical region and accesses the shared resource between T6 and T7.

Based on this model, the application speedup is calculated as

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

where  $p = \#$  of processors,  $\text{Busy}(1)$  = busy time for the basic work when using one processor,  $\text{Busy}(p)$  = total busy time summed over the  $p$  processors ( $\text{Busy}(p) - \text{Busy}(1)$  = parallel overhead busy work including redundant and added computations),  $\text{Idle}(p)$  = total idle time during serial phases,  $\text{Imbalance}(p)$  = total wait time on barriers and joins, and  $\text{Contention}(p)$  = total wait time on locks. Perfect speedup is only possible when the serial fraction, parallel overhead busy work, imbalance, and contention are zero.

For the scientific benchmarks, Figure 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  $p$  increases. LU and Cholesky may thus have worse speedup than the other four benchmarks; LU's load imbalance and Cholesky's busy time increase as  $p$  increases.

Cholesky has the most lock attempts, but has negligible contention time due to its relatively small critical regions. In

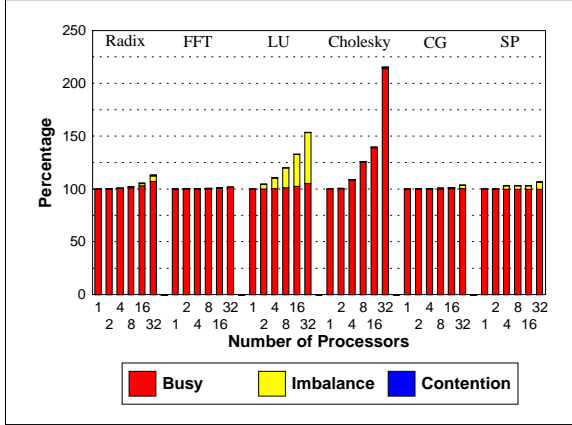


Figure 5. Concurrency and load balance in the parallel phase.

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 may be more significant than reported by CIAT.

#### 5.4. Communication Patterns

Total (inherent plus artifactual) communication is the traffic generated by processors when accessing data that is not allocated in its local memory, including traffic due to inherent coherence communication, cold-start misses, finite cache capacity, limited cache associativity, and false sharing [14]. Inherent communication is that which must occur in order to access a shared location, assuming that unlimited replication is allowed and that a memory location’s status is not affected by accesses to other locations (i.e. no false sharing). CDAT reports total communication; CIAT, inherent communication.

CIAT characterizes inherent communication by tracking, for each memory location, the type and the processor ID of the last access. For consecutive load accesses by multiple readers, their IDs are stored in a sharing vector. CIAT captures the inherent communication for a shared location whenever the accessing processor’s ID changes. CIAT classifies communication into four main patterns, and reports the volume of each pattern, 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 some processor, and at least one reader is not the writer. Moreover, when a processor performs multiple loads from the same memory location, only its first load is counted a RAW access. The second part of the common producer-consumer pattern is RAW (the first is WAR).
2. *Sharing degree* for RAW is a vector  $\mathbf{S}$ , 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 other processors have loaded. A WAR access generates a miss when the accessed location is not cached. Additionally, with an invalidate-based cache coherence protocol, it generates invalidation traffic.
4. *Invalidation degree* for WAR is a vector  $\mathbf{I}$ , 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

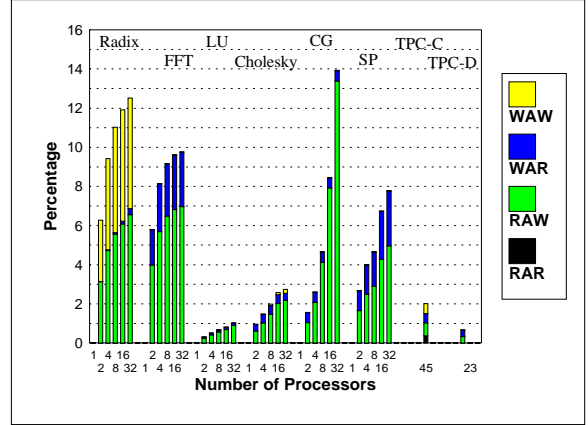


Figure 6. Percentage of the four classes of communication accesses among all data accesses, as a function of the number of processors for the two problem sizes.

turns accessing a memory location where in each turn a processor stores and loads, and its first access is 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.

For a particular cache coherence protocol, not all the above access patterns generate coherence traffic. For example, for iterative WAR and RAW with an update-based protocol, RAW accesses are 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 coalesces some CIAT-reported communication events since each miss operates on a cache line that often contains multiple shared elements.

Configuration dependent analysis does account for these effects in the given system configuration. However, it can miss some inherent communication events. For example, with finite write-back cache, a RAW access may generate coherence traffic with large caches in which the written line still resides, but it will not with a smaller cache that has replaced that line.

Figure 6 shows the distribution among the four communication patterns. Most communication in these benchmarks is RAW or WAR. Only Radix has a significant number of WAW accesses due to permuting sorted keys between two arrays. TPC-C shows some RAR and WAW accesses which may actually become RAW and WAR accesses in a complete execution trace that also includes operating system activity. Generally the TPC benchmarks have less communication and should have a lower coherence miss ratio, especially when multiple processes are run on each processor.

The communication percentage generally increases with  $p$  due to (i) the increase in RAW accesses of widely shared data, and (ii) the increase in boundary elements when the data is partitioned among more processors. Often, most communication occurs when accessing boundary elements.

Figure 7 shows the distribution of the 32 possible sharing degrees ( $(\mathbf{S}[k] \times 100 / \sum_{i=1}^{32} \mathbf{S}[i]); k = 1, \dots, 32$ ) for RAW accesses when using 32 processors. TPC-C has 45 possible sharing degrees and TPC-D has 23. Although only the first 32 are shown, 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  $p$  increases.

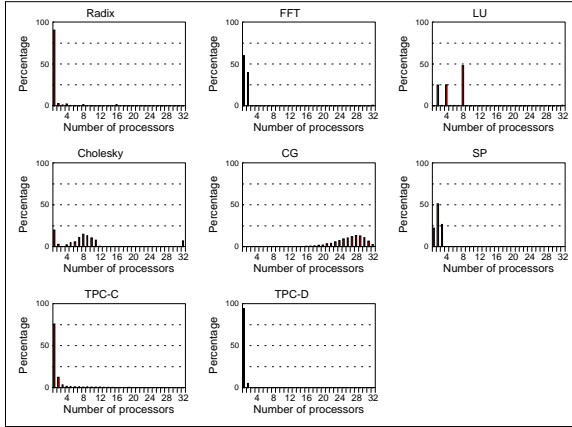


Figure 7. RAW sharing degree for 32 processors.

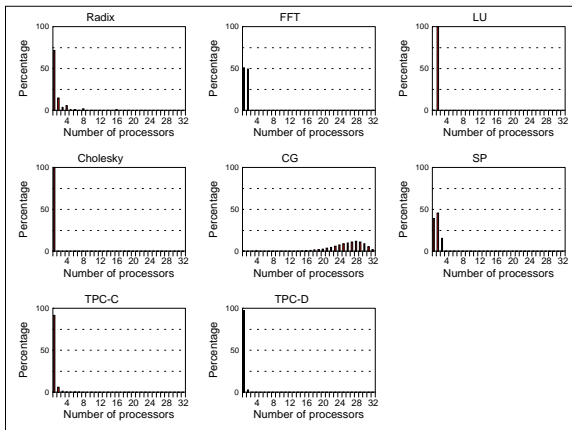


Figure 8. WAR invalidation degree for 32 processors.

Likewise, Figure 8 shows the distribution of invalidation degrees for WAR accesses when using 32 processors ( $(I[k] \times 100 / \sum_{i=1}^{32} I[i]); k = 1, \dots, 32$ ). TPC-C has 45 possible invalidation degrees and TPC-D has 23. All but LU and Cholesky have invalidation degree similar to their sharing degree; LU and Cholesky always have a WAR invalidation degree of 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 indicate that most of their 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 in which each communication event has a record that specifies the event type and time (in instructions). The time distribution analysis is summarized as follows:

1. The execution period is divided into 1000-instruction intervals, and the number of communication events in each interval is counted.
2. 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.
3. The communication rate density function is calculated (not including rate=0 which is excluded to minimize the effect of the serial initialization phase which has no communication).

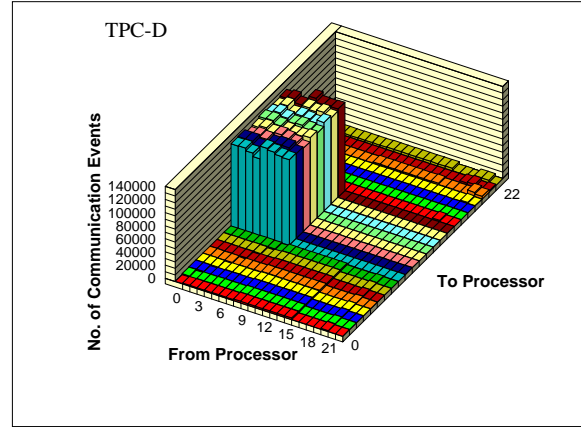


Figure 9. Number of communication events per processor pair.

4. The density function is integrated to find the distribution function.

LU, Cholesky, and CG each have a high burst of communication when the parallel phase starts (P1 through P31 begin to access the shared data initialized by P0). 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 is not uniform, CG and SP have periodic communication with fixed cycle (CG has a simple periodic behavior, SP has more complex periodic behavior with a longer cycle). For more detail, see [11].

### 5.6. Communication Locality

CIAT characterizes communication locality by reporting (in the matrix COMM\_MAT) the number of communication events for each processor pair.  $COMM\_MAT[i, j]$  is the number of communication events from  $P_i$  to  $P_j$  which is incremented by one for each  $P_j$  RAW access to a location after a  $P_i$  store, each  $P_j$  WAW access to a location after a  $P_i$  store, and each  $P_i$  invalidation by a WAR access after a  $P_j$  load.

The eight benchmarks differ significantly in their communication localities. As an example, Figure 9 shows COMM\_MAT for TPC-D. Most of TPC-D's 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 between two 8-process groups and spatially by partitioning the data into 8 parts. The communication matrices of the other benchmarks are available in [11].

### 5.7. 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 10 shows the number of referenced memory locations in three classes: code locations, private data locations, and shared data locations. Code locations are so much fewer than data locations that the code size is not even visible in the chart. Generally, more locations become shared as  $p$  increases; all the scientific benchmarks show this trend. Using 32 processors, more than 93% of Radix, SP, FFT, and LU data locations are shared, but only 15% in TPC-C and 12% in TPC-D. Furthermore, each additional thread may require some new private memory locations, causing an increase in the size of private memory, and hence total data memory. This trend is particularly visible in Cholesky, and somewhat in Radix.

Figure 11 shows the number of private and shared data accesses relative to the number of data accesses using one processor. In CG,

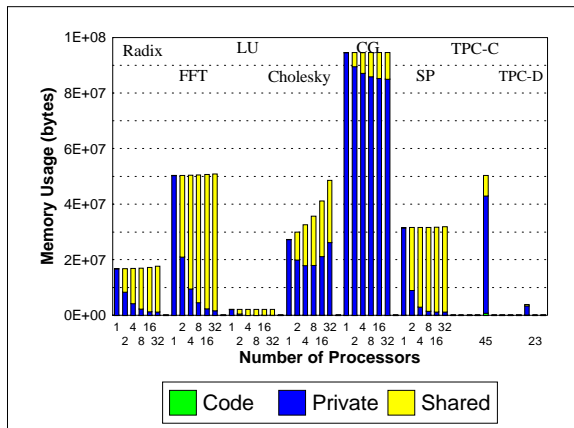


Figure 10. The size of code, private data, and shared data locations.

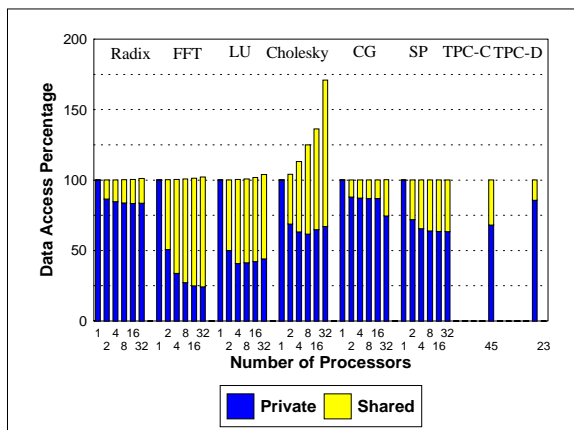


Figure 11. Number of private and shared data accesses normalized to the number of data accesses using one processor.

Cholesky, TPC-C, and TPC-D, shared locations are more intensely accessed than private locations. For example, using 32 processors, 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 Radix has only 17% of all its accesses to shared locations. The scientific benchmarks show some increase in the total number of accesses due to the increase in private accesses as  $p$  increases. However, the large increase in total accesses in Cholesky is due to increases in both private and shared accesses.

## 6. 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.

We have demonstrated that configuration independent analysis is a viable approach to characterizing shared-memory applications. CIAT efficiently and mechanically characterizes several important aspects of 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 character-

izes the working sets of an application by doing only one experiment and is not confused by coherence misses. CIAT also characterizes 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 characterizes communication variation over time.

We have demonstrated our analysis approach by analyzing eight benchmarks using a varying number of processors. This study shows the power of this approach and the insights that can be gained from configuration independent analysis of targeted benchmarks. The results are reported 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 are grateful to Rajiv Gupta, Josep Ferrandiz, Tom Rokicki, Lucy Cherkasova, and Milon Mackey of HP Labs for their guidance and assistance, and Isom Crawford and Herb Rothmund of the HP Convex Technology Center for providing access to the Exemplar NPB implementation. This characterization was carried out on the Convex SPP1600 of the University of Michigan Center for Parallel Computing.

## References

- [1] P. J. Denning, “Working Set Model for Program Behavior,” *Commun. ACM*, vol. 11, no. 6, pp. 323–333, 1968.
- [2] G. Abandah, “Tools for Characterizing Distributed Shared Memory Applications,” Tech. Rep. HPL–96–157, HP Labs, Dec. 1996.
- [3] Hewlett-Packard, *PA-RISC 1.1 Architecture and Instruction Set*, third ed., Feb. 1994.
- [4] D. F. Zucker and A. H. Karp, “RYO: A Versatile Instruction Instrumentation Tool for PA-RISC,” Tech. Rep. CSL–TR–95–658, Stanford University, Jan. 1995.
- [5] S. Fortune and J. Wyllie, “Parallelism in Random Access Machines,” in *Proc. 10th ACM Symp. on Theory of Computing*, pp. 114–118, 1978.
- [6] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, “The SPLASH-2 Programs: Characterization and Methodology Considerations,” in *Proc. 22nd ISCA*, pp. 24–36, 1995.
- [7] D. Bailey *et al.*, “The NAS Parallel Benchmarks,” Tech. Rep. RNR-94-07, NASA Ames Research Center, Mar. 1994.
- [8] Transaction Processing Performance Council, *TPC Benchmark C, Standard Specification*, Aug. 1992.
- [9] Transaction Processing Performance Council, *TPC Benchmark D, Decision Support, Standard Specification*, May 1995.
- [10] “Transaction Processing Performance Council Home Page.” <http://www.tpc.org/>.
- [11] G. Abandah and E. Davidson, “Configuration Independent Analysis for Characterizing Shared-Memory Applications,” Tech. Rep. CSE-TR-357-98, University of Michigan, Jan. 1998. <http://www.eecs.umich.edu/home/techreports/compsci.html>.
- [12] T. Brewer, “A Highly Scalable System Utilizing up to 128 PA-RISC Processors,” in *Digest of Papers, COMPCON’95*, pp. 133–140, Mar. 1995.
- [13] E. Rothberg, J. Singh, and A. Gupta, “Working Sets, Cache Sizes, and Node Granularity Issues for Large-Scale Multiprocessors,” in *Proc. 20th ISCA*, pp. 14–25, 1993.
- [14] 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*, pp. 189–199, 1994.