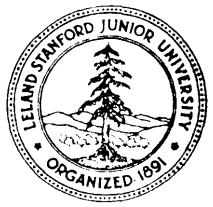
COMPUTER SYSTEMS LABORATORY

STANFORD UNIVERSITY STANFORD, CA 943054055



THE ACCURACY OF TRACE-DRIVEN SIMULATIONS OF MULTIPROCESSORS

Stephen R. Goldschmidt
John L. Hennessy

Technical Report No. CSL-TR-92-546

September 1992

This research has been supported by DARPA contract N00039-91-C-0138. Authors also acknowledge support from a Fannie and John Hertz Foundation Fellowship for Stephen Goldschmidt.

THE ACCURACY OF TRACE-DRIVEN SIMULATIONS OF MULTIPROCESSORS

Stephen R. Goldschmidt and John L. Hennessy

Technical Report: CSL-TR-92-546

September 1992

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

Abstract

In trace-driven simulation, traces generated for one set of machine characteristics are used to simulate a machine with different characteristics. However, the execution path of a multiprocessor workload may depend on the ordering of events on different processors, which in turn depends on machine characteristics such as memory system timings. Trace-driven simulations of multiprocessor workloads are inaccurate unless the timing-dependencies are eliminated from the traces.

We measured such inaccuracies by comparing trace-driven simulations to direct simulations of the same workloads. The results were identical only for workloads whose timing dependencies were eliminated from the traces. The remaining workloads used either first-come first-served scheduling or non-deterministic algorithms; these characteristics resulted in timing-dependencies that could not be eliminated from the traces. Workloads which used task-queue scheduling had particularly large discrepancies because task-queue operations, unlike other synchronization operations, were not abstracted.

Two types of simulation results had especially large discrepancies: those related to synchronization latency and those derived from relatively small numbers of events. Studies that rely on such results should use timing-independent traces or direct simulation.

Key Words and Phrases: multiprocessors, performance evaluation, software instrumentation, trace-driven simulation

Contents

Chapter 1

Background

Modem multiprocessor memory systems make extensive use of caches, buffers, and directories, any of which may be crucial to system performance. Analytic models cannot deal with the full complexity of such systems, so designers rely on simulation for insight into design trade-offs. Memory system and network designs are often evaluated using trace-driven simulation.

1.1 Principles of trace-driven simulation

We define ***an environment*** to be a deterministic model of the characteristics of a machine. Simulation seeks to emulate the execution of a particular workload, w, in a particular environment, 7, the ***target environment.***

Trace-driven simulation is based on the divide-and-conquer principle; the problem of simulating the execution of w in $\mathcal{T}$ is divided into two subproblems: trace generation and trace simulation. The events that ***occur*** when w executes ***in*** a ***trace generation environment*** $\mathcal{G}$ are captured and recorded in a ***trace.*** The simulator reads the trace, and the results are taken to describe the behavior that would be observed if w were run in $\mathcal{T}$. (See Figure 1.1.)

In this paper, we will consider memory system simulation only. In such simulations, the trace is composed primarily of memory references.

1.2 Multiprocessor trace-driven simulation

Multiprocessors complicate trace-driven simulation. When tracing a multiprocessor workload, each processing element (PE) generates a logically-distinct trace. A multiprocessor memory/network simulator reads a ***trace-set*** consisting of one logical trace for each PE.

How does the simulator interleave memory references from different PEs? Following Bitar [2], we classify multiprocessor trace-driven simulation techniques into two types, ***synchronous*** and ***asynchronous.***

In synchronous trace-driven simulation, the simulator preserves the reference sequence that occurred during trace generation. In practice, this technique is implemented in the trace generator, either by putting time-stamps in the traces or by merging references from different PEs into a single trace stream.

In asynchronous trace-driven simulation, the simulator adjusts the reference sequence to reflect differences between the reference latencies of $\mathcal{G}$ and those of $\mathcal{T}$. Asynchronous TDS has been used, for example, by Eggers and Katz [5] to study the relative performance of two cache coherency protocols.

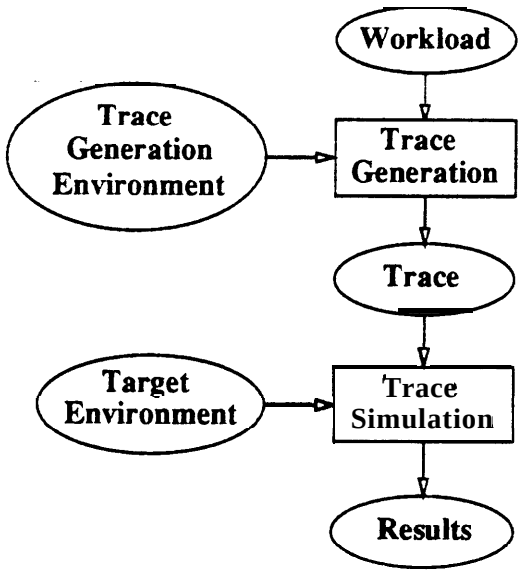


Figure 1.1: The trace-driven simulation process.

Both data and instruction references were recorded in their trace-sets. They assumed a nominal CPI of one, to which memory reference latencies were added by their simulator.

1.3 Accuracy issues

The trace generation environment and the target environment should ideally be identical, but the costs and limitations of existing trace generation techniques usually make this impractical. Realistic traces can be generated by executing workloads with instrumented hardware (as in ATUM [1]) or instrumented software (as in TRAPEDS [12]). Hardware instrumentation produces traces that reflect the characteristics of the hardware used. These characteristics are unlikely to match those of 7. Software instrumentation offers greater flexibility, but is comparatively slow. Regardless of which technique is used, trace storage costs limit the number traces that can be stored. When evaluating a large number of target environments, a single trace generation environment is usually chosen, and the same traces are used to simulate each target environment. For these reasons, the trace-generation environment seldom matches the target environment This mismatch is a potential source of inaccuracies.

This study is concerned only with the inaccuracies that are caused by $\mathcal{G} \neq 7$. It is not concerned with inaccuracies can that arise from approximations made during trace generation and simulation. We therefore assume that the trace generator and simulator are both accurate, so that trace-driven simulation with $\mathcal{G} = 7$ is accurate.

What conditions will ensure accurate trace-driven simulation if $\mathcal{G} \neq \mathcal{T}$?

$\mathcal{G}$ can influence the simulator only through the trace. If the workload were known to generate the same trace in both environments, accurate simulation would be assured This condition is satisfied in many uniprocessor simulations that do not time-stamp trace-events. For workloads in which memory latencies do not affect the sequence of memory references, environments that differ only in their memory latencies will generate identical traces.

Accurate trace-driven simulation can also be achieved by translating a trace generated in $\mathcal{G}$ into the

trace that would have been generated in 7. For this reason, accurate simulation of a time-stamped trace is possible if the simulator compensates for the differences between the memory reference latencies of $\mathcal{G}$ and 7 as it reads the trace. For example, suppose memory references in 7 always take one cycle longer than the corresponding references in $\mathcal{G}$. To translate a trace from $\mathcal{G}$ into one for 7, one could simply increase each timestamp by the number of preceding memory references.

For many environments, the cost of translating the trace into one that is valid for 7 may be comparable to the cost of generating a new trace. For example, it would be costly to adjust time-stamps if the memory reference latencies of $\mathcal{G}$ were so complex that they had to be simulated. Also, if the processor were to issue memory references out of order, the sequence of memory references generated by a basic block could depend on the environment. Asynchronous processes (such as interrupts) can introduce dependencies in the execution path itself. Such dependencies complicate accurate trace translation.

Certain workload features introduce timing-dependencies in the traces they produce. Consider a workload which (among other things) reads the elapsed time and formats it for output. The execution path through the formatting routine would then depend on the value of the elapsed time. The trace of the workload is therefore timing-dependent.

Multiprocessor workloads are particularly likely to have timing-dependencies, due to the presence of asynchronous interactions between processes. Such interactions include dynamic allocation, barrier synchronization, and unsynchronized read-write access to shared data.

Any workload which allocates shared resources on a first-come, first-served (FCFS) basis is likely to produce timing-dependent trace-sets. FCFS policies are commonly used to allocate tasks, loop iterations, memory, and access to critical-sections. The trace-set of such a workload is likely to depend on which resources are assigned to which PEs. Resource allocation depends on order in which PEs make their requests, which depends on the latencies of the preceding memory references made by each PE.

Barrier synchronizations also produce timing-dependent trace-sets if PEs spin while they wait at the barrier. Spins generate memory references, and the number of references generated depends on the relative completion times of the various PEs entering the barrier. Workloads that use FCFS allocation and barriers may produce timing-dependent trace-sets even if the results of the workload itself are not timing-dependent.

Some workloads allow unsynchronized access to shared data by different PEs. The trace-sets produced by such workloads will be timing-dependent if the order of the accesses affects the trace. For example, in simulated annealing algorithms, the final solution is an approximation. To permit greater parallelism, implementations of such algorithms often permit simultaneous accesses to the current optimum. Such workloads are likely to produce timingdependent traces.

When timing-dependencies are present, a small change to the environment can induce numerous changes to the execution path of a PE. These changes induce further changes, and so on. As a result, environments that differ in minor ways may generate radically different trace-sets. For instance, changing cache sixes can affect the number of misses occurring in each task. In workloads where tasks are assigned to PEs on a FCFS basis, this may impact the assignment of tasks to PEs, altering the large-scale communication patterns of the workload.

In Chapter 2, we will address the problem of accurate TDS in the presence of timing dependencies.

1.4 Direct simulation

Direct simulation provides an alternative to trace-driven simulation. In direct simulation, memory references are simulated as they occur. No traces are required, and only one environment is involved.

The accuracy of direct simulation is limited only by approximations in the workload execution software (which corresponds to the trace-generation software in TDS) and in the simulator itself.

1.5 Related work

In his critique [2], Bitar argued that trace-driven simulation is not generally valid for multiprocessor studies. After considering both asynchronous and synchronous trace-driven simulation, he concluded that multiprocessor trace-driven simulation techniques must be validated by analysis or low-level simulation. He showed that the inter-process interactions found in access-control algorithms are inaccurately modeled by trace-driven simulation. However, he did not connect the inaccuracies with timing dependencies, nor did he prove that the inaccuracies were ever great enough to be of practical concern.

In [3], we pointed out that the nondeterminis m in many parallel workloads makes accurate tracing difficult We emphasized that a trace-set obtained for one environment could represent an execution order that was impossible for the target environment. To address this problem, we presented the Tango simulation system, a flexible direct-simulation tool for studying shared-memory applications and environments.

In [7], Holliday and Ellis studied the amount of re-simulation required to create an accurate trace-set for one environment from a trace-set of another. Finding that traditional trace-sets were inadequate for accurate trace-driven simulation, they based their approach on intrinsic traces. ***(An intrinsic trace*** consists of the control-flow graph of the workload plus address and timing data for each basic block.) Their findings are difficult to apply to traditional trace-sets, which are composed of isolated events. They noted that traditional trace-sets can provide accurate simulation for some workloads (by abstracting the locks and barriers) but, beyond this, their work did not address the use of traditional trace-sets.,

In [8], Koldinger, Eggers, and Levy studied the effects of time dilation on trace generation. They introduced various degrees of time dilation into a software-based trace generation system. They found that time dilation had a slight impact on the accuracy of two simulation results: miss ratio and bus utilization. Errors occured only because their instrumentation system was unable to preserve the global order of memory references in the presence of time dilation. Modem instrumentation systems (such as Tango) have the ability to maintain this order. With such systems, time dilation need not be a source of error. Thus, the errors Koldinger, Eggers, and Levy observed were not intrinsic to trace-driven simulation.

1.6 Goals of this work

We have seen that a connection exists between timing dependencies and the accuracy of trace-driven simulation.

In the next chapter, we will present a technique that permits accurate trace-driven simulation of multiprocessors in the presence of timing dependencies caused by locks and barriers. Our approach is based on the methods used by Eggers and Katz in [5]. We will discuss difficulties in dealing with timing-dependencies which affect large portions of the trace-set.

In the chapters that follow, we quantify the inaccuracies in actual trace-driven simulations and explain the observed behavior. We find that the inaccuracies are very small in most cases. Results related to synchronization latency have large inaccuracies, as do results based on small numbers of events. The workloads which use task-queue scheduling have particularly poor accuracy because the timing-dependencies caused by task-queue operations are difficult to eliminate.

Chapter 2

Techniques for dealing with timing-dependent trace-sets

In a multiprocessor environment, events occurring on different PEs are largely asynchronous; the ordering of memory references from different PEs is generally unconstrained The interleaving of memory references from different PEs will depend on the environment. Synchronous TDS generally cannot accurately simulate environments other than $\mathcal{G}$, because it must simulate memory references in the order they are generated.

On the other hand, for some workloads, asynchronous TDS can accurately simulate multiple environments with a single trace-set. In asynchronous TDS, the interleaving of memory references from different PEs is determined during simulation, based on the latencies associated with the target environment Without re-interpreting the workload, a simulator cannot make extensive changes to the execution paths of individual PEs, but many timing-dependencies have only limited impact on the execution path. The ability to eliminate these timing-dependencies from *the* trace-set by means of ***abstraction*** is the key to accurate trace-driven simulation. We now show how this abstraction is accomplished for locks and barrier synchronizations.

2.1 Abstracting locks and barrier synchronizations

Suppose the only timing-dependency in w were caused by two PEs trying to acquire the same spin-lock: in 7, p_1 gets the lock after spinning three times, but in $\mathcal{G}$, p_2 gets it after spinning once. The execution paths would be different, so the raw trace-sets must also differ.

However, if we could ignore the number of times each PE spins during lock acquisition, the trace-set would look the same in both environments. We can make the trace-set timing-independent by translating each lock-acquisition loop into an abstract "spin-loop" event. The environment-dependent features (the number of iterations) are thus eliminated from the trace-set, which is now valid for both environments.

Naturally, for this technique to work, the simulator must process the new "spin-loop" event properly; when it reads a "spin-loop" event from a trace, it must determine the correct number of iterations for the target environment. The simulator can do this by simulating spins until another process releases the lock.

The approach we have described is essentially the same as that used in [5]. It relies on two seldom-violated properties of the spin-loops used to implement locks:

- Spin-loops have few parameters. To simulate the loop, the simulator need only know the address of

the lock and the address of the start of the loop. This information can be embedded in the trace-set. (We assume that all spin-loops in the workload are implemented in exactly the same way.)

- Spin-loops do not return a value, and their side-effects (changes to the state of the lock) *usually* do not affect any address calculations in the workload outside of related spin-loops. The timing-dependencies are thus limited to small portions the trace-set.

An application could violate the latter property if state of the lock were used for some purpose other than spinning. Such behavior might occur, for instance, in a workload that monitors its own locking patterns. Abstraction alone cannot handle such workloads correctly, because large portions of the trace-set could be affected.

Barrier synchronizations possess much the same properties as spin-loops. The effect of a barrier is to delay each PE by a different amount. If barrier entry is treated as an abstract trace event, these delays can be easily recomputed for each environment. (The simulator does this by simulating spins until all processes have entered the barrier.) Thus, the technique of abstracting timing-dependencies from the trace-set works for barriers as well as locks.

2.2 Abstracting FCFS scheduling primitives

-Abstracting locks and barriers eliminates most of the timing-dependencies found in ***statically-scheduled workloads***, that is, workloads in which, for a given input, the assignment of work to PEs is fixed. In ***dynamically-scheduled workloads,*** work is assigned to idle PEs on a first-come, first-served (FCFS) basis.

FCFS scheduling introduces timing dependencies. These dependencies can influence branch conditions and address computations over large portions of the workload. The fetch-and-add operation illustrates this problem. Fetch-and-add is commonly used to distribute iterations among PEs in a distributed loop; when a PE completes an iteration, it performs a fetch-and-add on a shared counter to determine the next iteration for that PE. While it is possible to abstract the fetch-and-add itself from the trace-set, the returned index is timing-dependent and affects the remainder of the trace in a complex fashion. It is typically used as an array index, and value read from the array can affect other data and/or branch addresses.

Such timing dependencies could be eliminated, at the cost of additional simulator complexity, if each iteration had a deterministic trace. The trace-generator could record the trace of each iteration in a separate file. The simulator could then assign iterations to PEs dynamically. This approach would allow accurate simulation of fetch-and-add in simple cases, however, we do not know of any simulators that have implemented it.

Shared queues and heaps pose a somewhat more complex problem than fetch-and-add operations. Like fetch-and-add operations, the values returned by queue operations may affect large portions of the trace-set. In addition, queues permit many possible operations, some of them quite complex. For example, the situation in which a PE tries to take work from an empty queue may be handled in many different ways, depending on the scheduling policies of the workload.

To summarize: the operations used in dynamic scheduling introduce trace-set timing-dependencies that are not localized. It is difficult to accurately simulate such workloads using conventional traces.

Chapter 3

Methodology

A major goal of this study is to evaluate the accuracy of trace-driven simulation. To this end, we simulated a simple shared-memory multiprocessor environment running various workloads. We obtained simulation results for this environment using two different trace-generation environments, one idealized, one similar to the target environment. We compared the results of these simulations to nominally "accurate" results obtained by direct simulation.

To eliminate errors due to approximations introduced by the instrumentation and simulator, the same instrumentation and memory simulator were used for both the trace-driven and direct simulations. The measured discrepancies therefore reflect approximations intrinsic to the trace-driven simulation techniques used.

The remainder of this chapter presents details of the instrumentation system, memory simulator, and workloads used.

3.1 The Tango Lite Instrumentation System

The workloads were compiled into executable simulations using **Tango Lite**, an assembly-language instrumentation system for the MIPS processor architecture. Tango Lite allows a uniprocessor to perform direct simulation of a multiprocessor workload.

Tango Lite is a successor to the Tango simulation system described in [4, 6]. Unlike **Tango, Tango Lite** represents application processes as light-weight threads executing in a common virtual address space. This change eliminates the need to call to operating system's scheduler, allowing much faster simulation.

Tango Lite instruments application code with calls to the simulator at strategic points during execution, such as loads, stores, and synchronization events. It also supports a distributed notion of time, in which each thread has an independent simulation clock. **Tango Lite** supports the scheduling of application threads in such a way that events from different threads are simulated in the correct temporal order.

3.1.1 Limitations of Tango Lite

Tango Lite has several limitations which affect the accuracy its results:

- **Tango Lite** does not instrument the operating system kernel. (The applications we study are computate-intensive; the kernel should have relatively little impact on their performance.)

- Tango Lite can only instrument those portions of the run-time libraries that are not used by its own run-time system. Uninstrumented library routines are treated as atomic operations.

  Most of the library code used by the applications in this study was instrumentable. The math library (libm) was instrumented, as was much of the C run-time library (libc). The estimated fraction of cycles spent in uninstrumented functions ranged from 0.16% of total cycles (in Barnes) to 14% (in LocusRoute). The uninstrumented portions of the run-time library included functions that provide output formatting (such as printf), memory allocation (malloc), and file I/O (fopen).

- Because the application and the simulator share a common address space, the addresses of application data are displaced by simulator data and by expansion of the application code by the instrumentation.

  Techniques to correct for such displacements are known (see [12]), but we have not implemented than for Tango Lite, feeling that the additional cost of these features was not justified. Since application data tend to be fairly contiguous in memory, we expect the effects of such displacement to be minor.

- Because Tango Lite instruments at the assembly-language level, it perturbs the reordering phase of the assembler. As a result, fewer delay slots in the application code can be filled, inflating the instruction counts somewhat. The order of independent data references within a basic block may also be perturbed.

Because these limitations affect both the trace-driven and direct simulations, they do not contribute to the discrepancies reported in Chapter 4.

3.1.2 Overhead in Tango Lite

The execution overhead of Tango Lite can be substantial. Fortunately, time dilation does not affect results obtained from Tango Lite-based simulations because (unlike most prior trace generation systems for multiprocessors [13]) event order in Tango Lite is determined by event-driven simulation. Thus, while simulation overhead limits the problem sizes that can be reasonably simulated, it has no effect on the accuracy of trace generation and simulation.

Instrumentation of the applications at data references and basic blocks typically increased application static size by a factor of four and uniprocessor execution time by a factor of 45. However, these figures do not include the memory system simulator. This instrumentation overhead was insignificant compared to the cost of simulating the memory system. In this study, memory simulation typically increased execution times by another factor of 17 over the execution time of the instrumented application with no simulator. Thus, the net slowdown factor was roughly 750. (45 x 17 = 765.)

3.2 Memory Simulator

We used a single memory system simulator for both trace generation and simulation. The simulator computes reference latencies for a hypothetical shared-memory multiprocessor with snoopy caches. Each PE has a fully-associative TLB and a direct-mapped cache. Writes pass through the cache and are broadcast on the bus, where they invalidate any corresponding copies in other caches. (For the target environment, $\mathcal{T}$, we used a TLBs with 28 entries and caches with 1024 4-word blocks.) The simulator also has an option allowing it to simulate an ideal memory system which satisfies all references in a single cycle.

In this study, all synchronizations except task-queues were treated as abstract operations. Task-queues are implemented slightly differently by each application, so they are much more difficult to abstract.

3.2.1 Limitations of the simulator

The simulator makes a number of simplifying assumptions about the environment:

- Pipelining is not simulated, and data accesses precede the corresponding instruction fetches, rather than following them, as they would in a real machine.
- Memory references associated with synchronization are filtered out, to avoid distorting the cache miss rates. Similarly, TLB faults are treated as idle time. As a result, synchronizations and TLB faults do not cause any memory references.
- We use an idealized synchronization model that assumes negligible communication delay between PEs. Thus, for instance, there is is no delay between the release of a lock and acquisition by a waiting PE.
- A trivial physical-to-virtual page mapping is implemented in the TLB. Contiguous virtual addresses always have contiguous physical addresses.
- Interference from DMA and interrupts are not simulated.

Because these limitations affect both trace-driven and direct simulations, they do not contribute to the discrepancies reported in Chapter 4.

3.2.2 Tracing

We optimized the tracing system to reduce the time- and space-requirements for trace generation:

- To eliminate the need for an explicit process ID in each trace record, separate trace files were generated for each PE.
- Explicit time-stamps were eliminated by recording instruction references in the traces. Bach instruction ideally takes one cycle, so the simulator can determine the number of cycles between two data references events by counting the number of intervening instructions and adding latencies due to TLB faults and cache misses. (This technique was inspired by the methodology of [5].)
- To further reduce trace size, consecutive instruction references were encoded into a single trace record and combined with any following data reference. (This technique is particularly effective with Tango Lite because it turns out that each instrumentation point generates only a single trace record.)
- Writing a trace-set to disk and then reading it back introduces substantial overhead, so we developed a buffering system to pass trace data directly from the trace generator to the simulator using fixed-size shared memory buffers, one for each simulated PE. Semaphores were used to synchronize the reading and writing of these buffers. Such a scheme may encounter deadlock because different PEs generate trace data at different rates. The simulator automatically detected deadlock conditions, and simulations that aborted due to deadlock were re-run using disk-based traces.

3.3 Applications

The application codes that make up the workloads in this study were derived primarily from the SPLASH suite described in [11].

The SPLASH suite consists of six applications: one **in FORTRAN (Ocean)** and five **in C (Cholesky, LocusRoute, MP3D, PThor,** and **Water).** Each application is a program of significant size and complexity. All except **Cholesky** solve complete problems in scientific simulation or computer-aided design. The applications were explicitly **parallelized** using **m4** macros based on the **monmacs**[9] macro library. [1]

Since many timing-dependencies are associated with dynamic scheduling, the scheduling algorithms used by the applications were particularly important to this study. Three of the original SPLASH applications **(MP3D, Ocean,** and **Water)** use static task scheduling; the remainder use taskqueues. We also introduced an alternate version of **MP3D,** called **Dynamic MP3D**. The **Dynamic MP3D** application is identical to **MP3D** except that it schedules loops dynamically, rather than statically.

We further supplemented SPLASH with a new application, called **Barnes,** which uses the **Barnes-Hut** algorithm to solve n**-body** problems similar to those arising in astrophysics. **(Barnes** has since been added to the SPLASH suite.) The **Barnes code is** interesting because of its unusual scheduling technique [10]. On the first iteration, it distributes tasks dynamically, using a distributed loop. On successive iterations, it partitions work among **PEs** using deterministic heuristics that tend to equalize the amount of work assigned to each PE. For comparison, we introduced a second version of this application, named **Dynamic Barnes,** which uses the distributed loop on every iteration.

In the end, a total of nine applications were used: three with static scheduling and six with dynamic scheduling.

33.1 Changes made to the applications

The SPLASH applications were written for a vitual-memory model which gives each process a private copy of all variables not declared "shared." (This follows **the semantics** of **the fork** system call.) However, the **Tango Lite** instrumentation system is threads-based. Threads provide a shared virtual address space, which is inconsistent with the model assumed in SPLASH. We therefore had to port the applications to the shared virtual-memory model. This was done by manually editing each application to replicate statically-allocated data that would be modified during parallel computation.

Most of the synchronizations in the applications were coded using **monmacs,** allowing **Tango Lite** to identify them and treat them as abstract operations. A few synchronizations were written as spin-loops in the original codes. Wherever possible, these synchronizations were converted to standard macros. The only remaining synchronizations were task-queue operations, found **in Cholesky, LocusRoute**, and **PThor**. Special instrumentation was added by hand to these applications to delimit the spins, allowing **Tango Lite** to recognize them as synchronization.

Throughout this study, the applications were simulated to completion; both parallel and serial **computatations** were simulated. To keep simulation costs reasonable, the problem sizes we used were (except in the case of **MP3D**) somewhat smaller than those used in [11].

Official SPLASH bug **fixes** through August 28, 1991 were incorporated into the codes used in this study.

[1] **Monmacs** was developed by the Advanced Computing Research Facility at Argonne National Laboratories.

3.3.2 Application characteristics

Table 3.1: Application characteristics with 10 PEs.

Application	Problem size	Synch. cycles millions	I-fetches millions	D-reads millions	% sh	writes millions	% sh
ocean	62 x 62 grid	0.9	34.6	9.4	95.7	3.2	89.2
Water	2 steps/4 x 4 x 4 lattice	25.8	51.5	8.5	37.4	4.3	4.9
MP3D	30 steps/5K particles	2 5	47.5	10.1	71.4	5.3	52.2
Dynamic MP3D	30 steps/5K particles	2.0	48.2	10.4	69.7	5.1	53.4
Barnes	10 steps/l28 bodies	18.8	61.5	12.0	63.5	7.5	35.4
Dynamic Barnes	10 steps/l28 bodies	8.8	61.0	12.0	63.4	7.5	35.5
Cholesky	**lshp.O**	88.8	44.5	11.5	90.6	4.0	78.8
LocusRoute	**2 steps/bnrE.grin**	1.5	25.3	5.0	69.9	2.2	43.5
PThor	500 steps/risc processor	53.9	74.3	17.2	75.2	5.8	50.3

Table 3.1 summarizes the synchronization and reference behavior of the applications. The measurements in this table are from simulations of the target environment (described in Section 3.2) with 10 PEs.

The "% sh" figures indicate the fraction of references of each type which were to shared addresses. The sharing fractions in the table are higher than those reported in [1 1]. This is due to data that were originally private to each PE, but were replicated in shared memory when the applications were ported to the shared virtual-memory model.

Chapter 4

Results

This chapter describes the experiments we performed and the behavior we observed

We evaluated the accuracy of asynchronous trace-driven simulation using two different trace-generation environments (G's) to simulate the target environment, $\mathcal{T}$. We chose the bus-based memory system described in Section 3.2 for our target environment. In each trace-generation environment, we simulated the nine parallel applications described in Section 3.3, varying the number of simulated PEs, p, from 1 to 10. Of the 90 possible workloads for each trace-generation environment, only 85 were actually simulated. (The Ocean application could not be simulated with p = 3, 5, 7, 8, or 9 PEs. Ocean constrains p to be either 1 or an even number that divides $g - 2$, where g is the grid size. Our choice of g = 62 limited us to p = 1, 2, 4, 6, or 10 PEs.)

The trace-generation environments we chose typify two approaches to trace generation. For our first set of experiments, we generated traces with an idealized environment, hereafter denoted by $\mathcal{G}_{\text{ideal}}$. $\mathcal{G}_{\text{ideal}}$ models an idealized memory system capable of satisfying any number of simultaneous references in a single cycle. It exemplifies the trace-generation environments used in studies in which traces are generated by software instrumentation. The target environment, T, was fairly complex, so the differences between $\mathcal{G}$ and $\mathcal{T}$ were substantial.

For the second set of experiments, we chose a more complex trace-generation environment, hereafter denoted by $\mathcal{G}_{1/4}$. This environment was identical to $\mathcal{T}$ except that its caches were only 1/4 as large (256 blocks). Such minor differences between the trace-generation and target environments might be found in studies that use hardware instrumentation to generate traces.

Among the 85 workloads we simulated, there were 22 that gave completely accurate TDS results in both trace-generation environments. These workloads are described in Section 4.1. The results for the remaining workloads are organized by trace-generation environment. Section 4.2 defines metrics for assessing simulation accuracy. The experimental data for $\mathcal{G}_{\text{ideal}}$ are presented in Section 4.3, and those for $\mathcal{G}_{1/4}$ may be found in Section 4.4. The patterns that emerge will be discussed and explained in Chapter 5.

4.1 Cases with accurate trace-driven simulation

The trace-driven results and the nominal results agreed perfectly for 22 workloads. In the remaining 63 workloads, agreement between TDS and nominal results was the exception rather than the rule.

The workloads with accurate TDS were identical in both trace-generation environments, namely:

- workloads with a single PE (p = 1) and

- workloads where the application was **Ocean** or **Water.**

These workloads illustrate -important properties of TDS which will be discussed in Section 5.1. However, we exclude them from the data presented in the following sections.

4.2 Error metrics

Our evaluation of TDS accuracy is based on discrepancies ("errors") between the trace-driven simulation results and nominal results. (Because the simulations are deterministic, we present data from a single simulation of each workload in each environment.)

In individual workloads, we quantify the discrepancies as a percentage of the nominal value. More precisely, given a nominal result, x_{nom}, and the corresponding TDS result, x_{tds}, we define the error:

$$err(x) = \begin{cases} (x_{tds} - x_{nom})/x_{nom} \times 100\%. & \text{if } x_{nom} > 0; \\ 0\%. & \text{if } x_{tds} = x_{nom} = 0. \end{cases}$$

$err(x)$ exaggerates positive errors, resulting in skewed error distributions. When making statistical tests, we compensate for this skew by applying a logarithmic transformation ***to*** x, obtaining ***the normalized error,*** $nerr(x) = \ln(x_{tds}/x_{nom})$.

For many results, $|err(x)|$ was not significantly correlated with the number of PEs. We therefore tabulate ***the mean error magnitude*** for each application:

$$\text{mean } |err(x)| = \frac{1}{9}\sum_{p=2}^{10} |err(x)|$$

Since the mean error magnitude obscures the signs of the errors, we append a "+" or a "−" to the mean magnitude (to indicate the sign of the mean) in cases where the sign is significant. Significance is determined by means of a one-sided f-test with $\alpha = 0.05$. (The t-test assumes normally-distributed data, so we apply the test to $nerr(x)$ rather than $err(x)$.)

When comparing the contributions of different components of a result to its error, we use the magnitude of the difference between the TDS and nominal values of the component, $|x_{i,tds} - x_{i,nom}|$, denoted $|\Delta(x_i)|$.

4.3 Errors with $\mathcal{G}_{ideal}$

We now present data from simulations of 63 traces generated in the $\mathcal{G}_{ideal}$ environment.

This section is divided into subsections dealing with different simulation results. In Section 4.3.1, we present error data for elapsed tune. In Section 4.3.2, we decompose the elapsed time to distinguish the parallel and serial stages. In Section 4.3.3, we compare the errors in the elapsed time to errors in the cycle count. In Section 4.3.4, we decompose the cycle count into instructions and idle cycles. We find that timing errors (including elapsed time, parallel time, and cycle counts) were primarily due to errors in the simulation of idle time. We then analyze the three sources of idle time: Section 4.3.5 presents data on synchronization latency errors, Section 4.3.6 deals with errors in memory reference latencies, and Section 4.3.7 deals (briefly) with address-translation latency.

4.3.1 Elapsed time

We start by considering the elapsed time, T_e, measured by trace-driven simulation $Err(T_e)$ is particularly important because it is tied directly to $err(S)$, the error in the overall **speedup.**[1]

The majority of the T_e errors were small or insignificant; $|err(T_e)|$ was < 0.2% for **52.%** of the workloads. The largest T_e error was **+2.03 %** (for **Dynamic Barnes,** with eight **PEs**) and the overall mean $|err(T_e)|$ was only 0.41% (N = 63).

It is **difficult** to account for the T_e errors in individual workloads because T_e is influenced by numerous minor perturbations of the simulated execution path. To make sense of the data, some organization is needed. An analysis of variance **(ANOVA)** test on $nerr(T_e)$ showed that $err(T_e)$ did not depend on p any more than could be accounted for by chance. We therefore organized the data on an **application-by-**application basis.

Table 4.1 reports the mean and extreme $|err(T_e)|$ for each application.

Table 4.1: Elapsed Time Errors with $\mathcal{G}_{\text{ideal}}$ for 2 to 10 **PEs.**

Application	mean $\|err(T_e)\|$	max $\|err(T_e)\|$
MP3D	**0.03 %**	**-0.09%**
Dynamic MP3D	0.08%+	+0.13%
Barnes	**0.03%-**	**-0.07%**
Dynamic Barnes	1.33 %+	-12.03%
Choleskp	0.63%	-1.72%
LocusRoute	0.45%	i- 1.40%
PThor	0.34%+	+0.54%

We partition the applications into two groups based on the size of $|err(T_e)|$. The ***large-error*** **group,** containing **Choleskp, Dynamic Barnes, LocusRoute**, and **PThor**, consists of applications with mean $|err(T_e)| > 0.2\%$. The ***small-error*** group, containing **MP3D, Barnes,** and **Dynamic MP3D**, consists of applications with mean $|err(T_e)| < 0.1\%$. The ratio between the variances of $nerr(T_e)$ for the two groups was 190, indicating that the distinction between the two groups was fairly great.

The signs of the T_e errors were biased toward the positive side. Whereas $err(T_e)$ was positive for 45 workloads, it was negative for only 18. (In other words, trace-driven simulation exaggerated the elapsed time five times for every two times that it understated it.) Four of the applications showed a significant bias in $nerr(T_e)$: three were biased toward positive errors and one **(Barnes) was** biased toward negative errors.

4.3.2 Parallel time vs. serial time

To determine the source of the errors in T_e, we decompose T_e into three components:

- the duration of the (serial) initialization stage, T_i,
- the duration of the parallel stage, T_{par}, and

[1] **Because** $S = (T_{e,p=1})/T_e$, and $T_{e,tds,p=1} = T_{eeds,p=1}$, it follows from the **definition** of $err(x)$ that:

$$err(S) = \left(\frac{100\%}{err(T_e) + 100\%} - 1\right) \times 100\%.$$

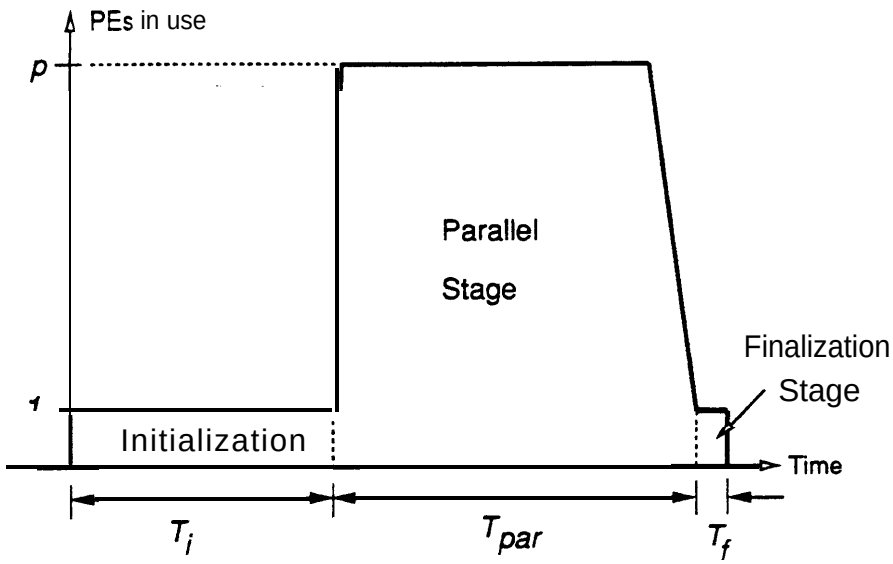


Figure 4.1: Idealized workload execution profile.

- the duration of the (serial) finalization stage, T_f.

Figure 4.1 illustrates this decomposition for an an idealized workload.

The proportions of these components depended strongly on both the application and the number of PEs simulated. In particular, the parallel fraction tended to decrease as the number of PEs grew. Figure 4.2 (on the next page) shows the decomposition of T_ϵ for each workload.

We are primarily interested in the parallel stage, since that is the stage that would dominate in simulations of full-scale workloads. However, for the problem sixes we used, a significant portion of each T_ϵ was due to the serial stages. The serial fraction, $(T + T_f)/T_\epsilon$, ranged from 1.6% to 85.7%, and $T_i + T_f$ represented more than half of T_ϵ for 56. % of the workloads. Despite their importance to the elapsed time, the serial stages do not seriously distort our error measurements; they simply dilute errors in the parallel stage. To support this claim, we present Table 4.2, which summarizes the effect of trace-driven simulation on the duration of each stage.

Table 4.2: Stage Duration Errors with $\mathcal{G}_{\text{ideal}}$ for 2 to 10 PEs.

Application	mean $\lvert err(T_i)\rvert$	mean $\lvert err(T_{par})\rvert$	mean $\lvert err(T_f)\rvert$
MP3D	0%	0.05%	10.1%
Dynamic MP3D	0 %	0.16%+	8.83%
Barnes	0%	0.03%	8.12%
Dynamic Barnes	0%	1.39%+	6.01%−
Cholesky	0%	3.20%	3.81%
LocusRoute	0%	1.34%	0.45%
PThor	0%	1.93%+	549.%

The initialization stage, which made up the bulk of the serial time, was accurately simulated for all workloads. The finalization stage, meanwhile, was greatly distorted. T_f exceeded 0.1% of T_ϵ in only two applications: **Cholesky** and **LocusRoute**. The finalization stage was too short to noticeably affect $err(T_\epsilon)$. In all 63 workloads, the parallel stage contributed more to $err(T_\epsilon)$ than the finalization stage.

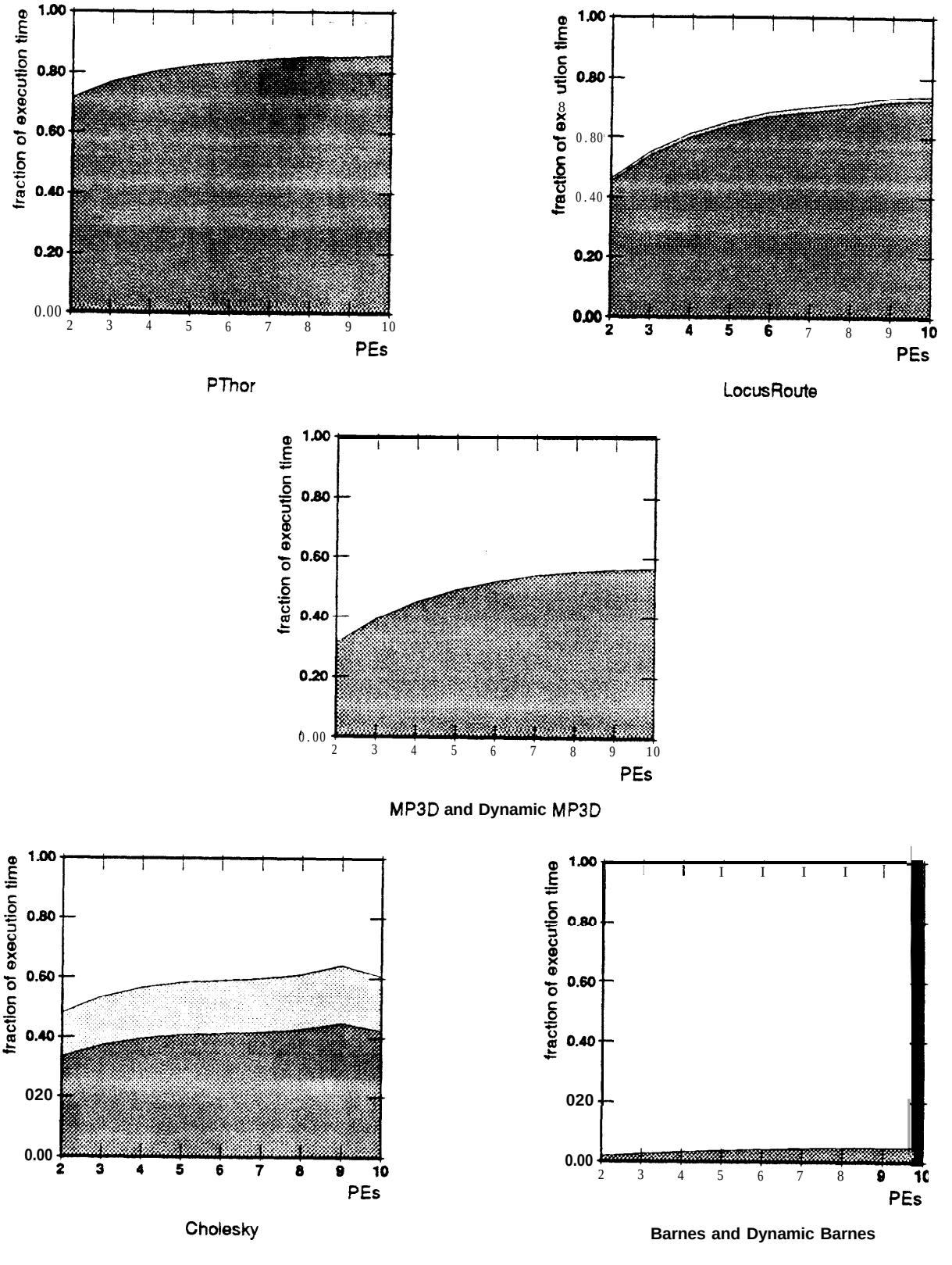


Figure 4.2: White areas = parallel. Dark gray = initialization. Light gray = finalization.

We should therefore expect the T_{par} errors to be slightly inflated versions of $err(T_\epsilon)$. This is exactly what we observe in our data. The signs of the $err(T_\epsilon)$ and $err(T_{par})$ agreed in every workload, and $|err(T_{par})|$ always exceeded $|err(T_\epsilon)|$. The largest T_{par} error was − 18.9% (for Cholesky with six PEs) and the overall mean $|err(T_{par})|$ was 1.16% (N = 63).

The distinction between the two groups of applications is even clearer from the T_{par} errors than it was from $err(T_\epsilon)$. The ratio of the group variances of $nerr(T_{par})$ was 1600, as opposed to 190 for $nerr(T_\epsilon)$. Each of the four large-error applications had one or more workloads with $|err(T_{par})| > 2\%$, but among the small-error applications $|err(T_{par})|$ never exceeded 0.3%.

The T_f errors were larger than the T_{par} errors in 41 out of 63 workloads. $|Err(T_f)|$ exceeded 10% in 16 out of 63 workloads. The T_f errors in PThor were especially remarkable. The largest T_f error was 4850.%, for PThor with two processors. This one workload was responsible for most of the $err(T_f)$ in PThor. All but one of the remaining PThor workloads had $err(T_f) = 0$.

Negative T_f errors outnumbered positive ones by ***30 : 26,*** and **Dynamic Barnes** was the only application which showed significant sign-bias in $err(T_f)$.

For the remainder of this report, it is important to remember that errors are reported for complete workloads, including the initialization stage. For larger workloads, the initialization stage would be insignificant, so many of the errors would be larger by a factor of two or so.

4.3.3 Cycle counts

The cycle count, C, is another important simulation result, closely related to ***T,.*** We define C to be the sum (over all PEs) of cycles consumed by the workload. C includes idle cycles (latencies due to synchronization, memory references, and address translation) but not the cycles consumed by inactive PEs during serial stages of the computation. In to Figure 4.1, T_ϵ is the length of the curve along the z-axis, while C corresponds to the area under the curve. We can approximate the area by $T_\epsilon + (p - 1)T_{par}$, so C is closely related to both T_ϵ and T_{par}.

Table 4.3 summarizes the effect of trace-driven simulation on C'.

Table 4.3: Cycle Count Errors with $\mathcal{G}_{ideal}$ for 2 to 10 PEs.

Application	mean $\|err(C)\|$	max $\|err(C)\|$
MP3D	**0.04%**	-0.12%
Dynamic MP3D	0.13%+	***10.25%***
Barnes	***0.03%***	***-0.07%***
Dynamic Barnes	1.38%+	+2.11%
Cholesky	***10.2%-***	- 18.2%
LocusRoute	1.19%	-3.23%
PThor	1.08%+	+2.39%

Except for the **Cholesky** workloads, $err(C)$ resembled $err(T_{par})$ and, to a lesser extent, $err(T_\epsilon)$. There was a mild (r = 0.422) correlation between $nerr(C)$ and $nerr(T_\epsilon)$ and a slightly stronger one [r = 0.4971 between $nerr(C)$ and $nerr(T_{par})$. In 49 out of 63 workloads, $err(C)$ fell somewhere between $err(T_\epsilon)$ and $err(T_{par})$. ***The*** largest C error was -18.2% (for **Cholesky** with ten PEs) and the overall mean $|err(C)|$ was 2.00% (N = 63).

The applications fell into the same groups as before, and the ratio of the normalized error variances was 5200.

The sign of $err(C)$ was biased toward the positive side by a 40 : 23 margin.

4.3.4 Instructions vs. idle cycles

We now decompose C as the sum of the dynamic instruction count, I, and the number of idle cycles, $C - I$. This is possible because all instructions execute in a single cycle, and we do not count instructions that perform synchronization.

Figure 4.3 illustrates this decomposition of C for each workload. Idle time represented 42.1% to 75.8% of C in our workloads, and the idle fraction, $(C - I)/C$, grew at a fairly uniformly rate with respect to the number of PEs. The plot of idle fraction versus p was similar for all seven applications.

Table 4.4 presents our measurements of the errors in I and $C - I$.

Table 4.4: Cycle Count Errors with $\mathcal{G}_{ideal}$ for 2 to 10 PEs.

Application	mean $\|err(I)\|$	mean $\|err(C - I)\|$
MP3D	0.03%	0.10%
Dynamic MP3D	**0.04%**	**0.24%+**
Barnes	**0.00%**	0.05 %
Dynamic Barnes	**0.00%**	2.57 %+
Cholesky	**0.00%**	**15.1%-**
LocusRoute	**0.04%**	2.33%
PThor	0.20%	**1.74%+**

I was barely affected by TDS. The largest I error was only +0.37% (for Cholesky with ten PEs) and the overall mean $|err(I)|$ was 0.04% (N = 63). Negative I errors outnumbered positive ones by a 32 : 26 margin. Five workloads experienced no distortion of I at all.

$C - I$ usually suffered much greater distortion than I. In all but two workloads, $|\Delta(C - I)|$ was $> |\Delta(I)|$. The largest $C - I$ error was -24.3% (for Choieskp with six PEs) and the overall mean $|err(C - I)|$ was 3.16% (N = 63). Positive $C - I$ errors outnumbered negative ones by a 41 : 22 margin

The large-error applications can be clearly distinguished from the small-error ones by $|err(C - I)|$ but not by $|err(I)|$.

We conclude that the discrepancies in C were primarily due to errors in the simulation of idle cycles. Furthermore, noting the interrelationships between T_ϵ, T_{par}, and C', as well as the correlations between their errors, we attribute the majority of $err(T_\epsilon)$ and $err(T_{par})$ to errors in $C - I$.

Next we decompose the idle time. In our simulations, idle time is the sum of the latencies from three types of events: synchronizations, memory references, and address-translation faults.

We define *synchronization latency*, L_σ, to be the number of idle PE-cycles spent waiting on synchronization primitives. The applications use a variety of synchronization primitives, including locks, barriers, and waits. Time spent spinning on shared variables (such as task queues) is also considered to be synchronization, as are the latencies due to mutual exclusion during fetch-and-increment operations.

Memory reference latency, L_{ref}, is defined to be the number of PE cycles spent waiting for responses from their caches. *Address-translation latency,* L_{TLB}, is the total latency due to TLB faults.

Figure 4.4 shows the decomposition of the idle time into L_σ, L_{ref}, and L_{TLB}. From the figure, it appears that memory reference latencies were the most important of the three. In 58 out of 63 workloads, L_{ref} dominated the idle time, exceeding both L_σ and L_{TLB}. Synchronization latency generally *increased* with the number of PEs, but it accounted for a substantial portion of the idle time only in PThor and Cholesky. (Translation latencies were relatively small in most workloads; this may reflect the small

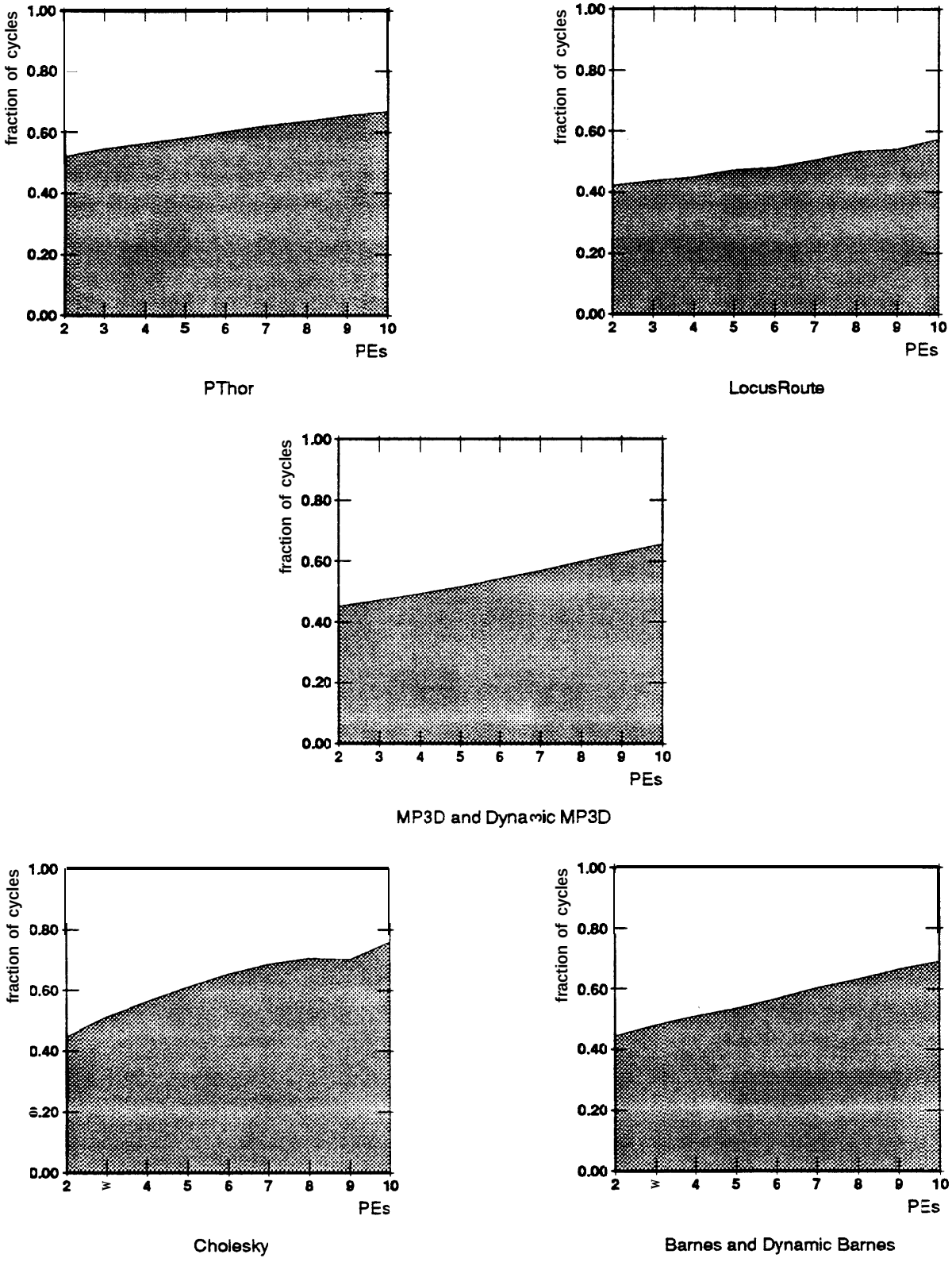


Figure 4.3: White areas = instructions. Dark areas = idle cycles.

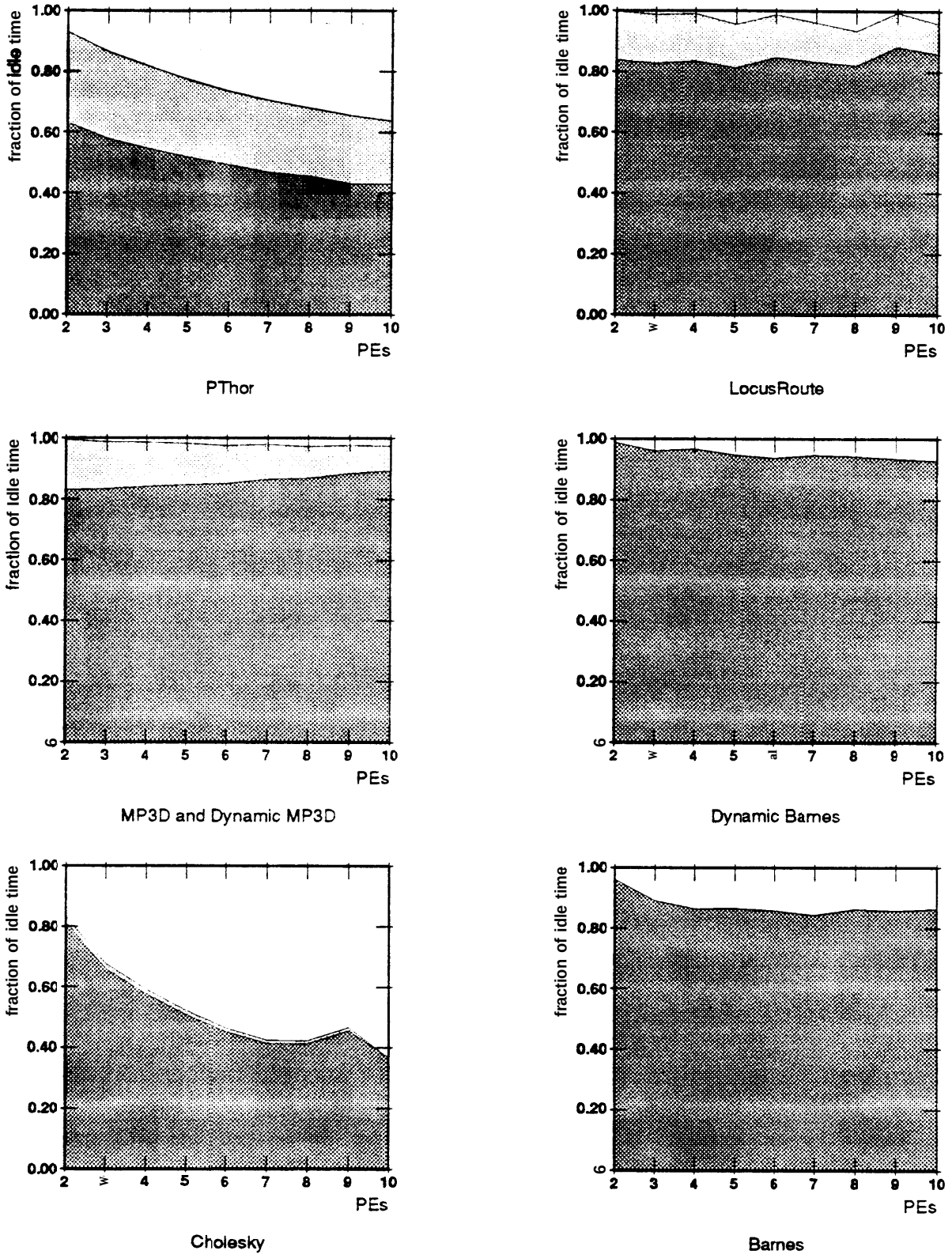


Figure 4.4: White areas = synchronization. Dark gray = reference. Light gray = translation.

problem sizes used in this study.)

The errors in $C - \mathrm{I}$ were **not uniformly** distributed among its three components. $\boldsymbol{L}_\sigma$ made the largest contribution to $err(C - \boldsymbol{I})$ despite its small contribution to $C - \mathrm{I}$. Indeed, $\boldsymbol{L}_\sigma$ errors dominated in 53 out of 63 workloads. (That is, $|\Delta(L_\sigma)|$ exceeded both $|\Delta(L_{\mathrm{ref}})|$ and $|\Delta(\boldsymbol{L}_{TLB})|$ in those workloads.) Errors in L_{ref} dominated in eight of the remaining 10 workloads, and errors in $\boldsymbol{L}_{TLB}$ dominated in only two workloads.

4.3.5 Synchronization latency

Because synchronization **latency errors** account for so much of the idle time error, **we** shall treat **them** first. Table 4.5 summarizes the errors in total synchronization latency, L_σ, and the number of synchronization operations, Σ.

Table 4.5: **Synchronization** Latency Errors with $\mathcal{G}_{\mathbf{ideal}}$ for **2** to 10 **PEs.**

Application	mean $\lvert err(L_\sigma)\rvert$	mean $\lvert err(\Sigma)\rvert$
MP3D	**5.71%**	**0.04%**
Dynamic MP3D	18.1%+	**0.05%**
Barnes	**0.29%-**	0.16%
Dynamic Barnes	66.5%+	0.18%+
Cholesky	28.6%-	0%
LocusRoute	106.%	0.09%
PThor	7.85%+	0.58%

The L_σ errors were often substantial, exceeding 5% in 45 out of 63 workloads. The mean error magnitude was 33.3% (N = 63), and **the** largest error was +231% (for **LocusRoute, with six PEs).**

The signs of the L_σ errors were predominantly positive, by a 42 : 21 margin.

We see from the table that the number of synchronization operations was hardly affected by TDS. We conclude that $err(L_\sigma)$ arises from inaccurate simulation of the latencies of synchronization operations.

There are substantial differences between the $err(L_\sigma)$ distributions of the different applications. To understand the differences, we decompose L_σ into three components. We distinguish two major components of $\boldsymbol{L}_\sigma$: mutex latency and delay latency. ***Mutex latency,*** $\boldsymbol{L}_\mu$, results from PEs trying to execute a common critical section; locks, for instance, produce mutex latency. ***Delay latency*** is due to completion waits in the workload; barriers and task-queue spins produce this type of latency. We subdivide delay latency into two s&components: that due to task queues, $\boldsymbol{L}_q$, and the remaining delay latencies, L_d. Figure 4.5 illustrates the breakdown of the synchronization latency in each workload.

The proportions of the synchronization components varied, but delay latencies always dominated. Task-queue delays were absent from **MP3D, Dynamic MP3D, Barnes,** and **Dynamic Barnes. Only in Cholesky** and **LocusRoute** did they form a substantial fraction of the synchronization latency.

Table 4.6 presents error data for each component of the synchronization latency.

The observed L_q errors were negative in 26 of the 27 workloads that had task-queue latency. (The sole exception **was LocusRoute** with six **PEs.)** The mean $|err(L_q)|$ for those 26 workloads was **54.7%,** and the largest L_q error was **-97.3%** (for **LocusRoute** with eight **PEs).** $|\Delta(L_q)|$ dominated the other components of $err(\boldsymbol{L}_\sigma)$ in 13 workloads.

L_d occurs at the end of computational phases, when **PEs** that exhaust their supply of tasks must wait for other **PEs** to finish their outstanding tasks. The smaller $\boldsymbol{L}_d/C$ **is,** the better the load balance of the

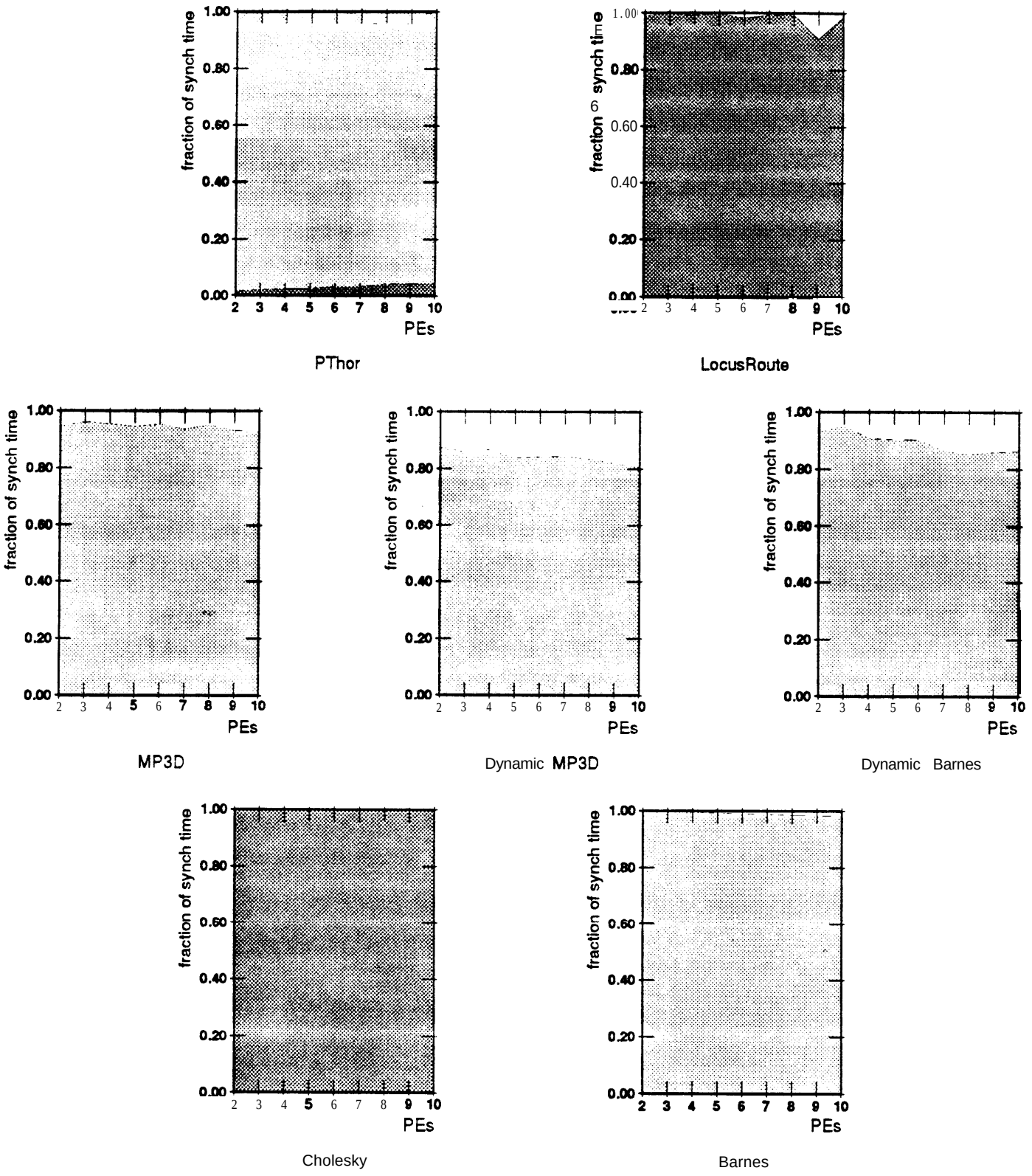


Figure 4.5: White areas = mutex. Dark gray = task-queue delay. Light gray = non-queue delay.

Table 4.6: Synchronization Latency Errors with $\mathcal{G}_{\text{ideal}}$ for 2 to 10 PEs.

Application	mean $\lvert err(L_q)\rvert$	mean $\lvert err(L_d)\rvert$	mean $\lvert err(L_\mu)\rvert$
MP3D	0%	5.99%	9.12%
Dynamic MP3D	0%	21.1%+	3.35%
Barnes	0%	0.23%+	26.8%−
Dynamic Barnes	0%	75.6%+	11.5%−
Cholesky	40.6% −	1560000.%+	21.4%
LocusRoute	64.9%-	128000.%+	16.1%
PThor	59.3%−	9.81%+	9.67%

workload. The largest L_d error was +2.470.000%, for **Cholesky** with three PEs. There were a total of 17 workloads with $\lvert err(L_d)\rvert > 200\%$, all involving either **Cholesky** or **LocusRoute.**

L_d dominated the other components of $err(L_\sigma)$ in 60 out of 63 workloads.

The observed $err(L_d)$ values were positive in 57 out of 63 workloads, indicating that load balance in the trace-driven simulation was poorer than it was in the direct simulation. From Table 4.6 we see that this tendency was most pronounced in applications that used dynamic scheduling extensively, especially **Cholesky** and **LocusRoute**. Of the two workloads which had negative $err(L_d)$, one involved **MP3D** and the other involved **Barnes: MP3D** is statically scheduled, and **Barnes** was dynamically-scheduled only during the first of its ten steps. The dynamic (FCFS) versions of both applications had larger L_d errors that were consistently positive.

There was one workload with $err(L_d) = 0$, namely **Cholesky** with six PEs.

The largest L_μ error was +52.6%, and the mean $\lvert err(L_\mu)\rvert$ was 14.0% ($N = 63$). Negative L_μ errors outnumbered positive ones, by a 41 : 22 margin, indicating that trace-driven simulation tends to underestimate mutex wait times.

The overall $err(L_\sigma)$ was typically determined by the balance between negative $err(L_q)$ and positive $err(L_d)$. L_q errors dominated $err(L_\sigma)$ in only eight of the 63 workloads. Those eight workloads all involved either **Barnes** or **MP3D**.

Next we examine the errors in memory reference latency and other memory statistics.

4.3.6 Memory reference latency

We have already seen that the memory reference latency, L_{ref}, dominated $C - I$, yet the errors in L_{ref} did not dominate $err(C - I)$. This occurred because $err(L_{\text{ref}})$ was relatively small in most workloads. The largest L_{ref} error was only -1.99% (for **PThor with** 10 PEs) and the mean $\lvert err(L_{\text{ref}})\rvert$ was 0.27% ($N = 63$).

The large-error applications had L_{ref} errors that were distinctly larger than those of the other applications; the ratio between the variances of $nerr(L_{\text{ref}})$ for the two groups was 130. Table 4.7 tabulates mean error magnitudes for L_{ref} and the count of total memory references, Ref.

Trace-driven simulation gave a highly-accurate measurements of Ref in all workloads; the largest error was only +0.38% (for **PThor** with seven PEs). The seven largest Ref errors were all due to **PThor,** making this application distinctive.

By comparing $err(Ref)$ to $err(L_{\text{ref}})$, we can determine to what extent $err(L_{\text{ref}})$ was due to errors in the average reference latency. Among large-error applications, $err(L_{\text{ref}})$ clearly exceeded $err(Ref)$, indicating that $err(L_{\text{ref}})$ was due primarily to errors in the average reference latency. For the remaining

Table 4.7: Memory Reference Errors with $\mathcal{G}_{ideal}$ for 2 to 10 PEs.

Application	mean $\|err(L_{ref})\|$	mean $\|err(Ref)\|$
MP3D	0.05% ---	0.03%
Dynamic MP3D	0.05%-	0.04%
Barnes	0.04%	0.00%
Dynamic Barnes	0.24%-	0.00%
Choleskp	0.47%	0.00%
LocusRoute	0.73%-	0.03%
PThor	*0.32%*	*0.20%*

applications, $err(L_{ref})$ was smaller; for these applications the errors in Ref were comparable in size to the errors in the average reference latency.

We decompose Ref into instruction fetches, ***I,*** data reads, ***R,*** and writes, W. The relative proportions of these operations were fairly uniform across all 63 workloads, though each application had a slightly different mix. Instructions fetches made up 74.2% to 77.9% of the references, data reads 14.8% to 19.2%, and writes 6.0% to 9.3%. We also distinguish the shared writes, W_{sh}, as a portion of W. Shared writes represented 35.3% to 78.8% of the total number of writes.

Table 4.8: Memory Reference Count Errors with $\mathcal{G}_{ideal}$ for 2 to 10 PEs.

Application	mean $\|err(I)\|$	mean $\|err(R)\|$	mean $\|err(W)\|$	mean $\|err(W_{sh})\|$
MP3D	*0.03%*	*0.03 %*	*0.03 %*	0.03%–
Dynamic MP3D	0.04%	0.04%	*0.03 %*	*0.03 %*
Barnes	0.00%	0.00%	0.00%	0.00%
Dynamic Barnes	0.00%	0.00%	0.00%	0.00%
Choleskp	0.00%	0.00%	0%	0%
LocusRoute	0.04%	0.04%	*0.03%*	*0.06%*
PThor	0.20%	*0.23%*	*0.14%*	0.19%

Table 4.8 shows that the errors in the components of Ref were small. However, Ref is not very useful for analyzing $err(L_{ref})$. The majority of the reads and instruction fetches are satisfied by the caches, so they do not contribute any latency to L_{ref}. Reference latency is predominantly due primarily to ***non-local*** references, that is, references not satisfied by the local cache. (Local references ***can stall the*** PE, but this is a very minor effect.)

The non-local references, ***F,*** may be decomposed in various ways. One decomposition partitions them into instruction misses, I_m, data read misses, R_m, and writes, W; L_{ref} may then be similarly partitioned into L_{Im}, L_{Rm}, and L_W. An alternative decomposition divides L_{ref} into cache latency, L_c, and bus latency, L_b. Table 4.9 presents both decompositions. Columns ***2 through 4*** tabulate I_m, R_m, and W as fractions of the total number of non-local references, ***F.*** (For the simulated memory system used in this study, each non-local reference occupies the bus for exactly two cycles. The distribution of the latencies are similar for both types of non-local references, so I_m/F is roughly equivalent to L_{Im}/L_{ref}.) Columns 5 and 6 show the decomposition of L_{ref} into L_c and L_b.

Most instruction fetches and reads were satisfied by the caches, so the majority nonlocal accesses in each workload were writes. Memory latency was divided fairly evenly between L_c and L_b, with I_m dominating L_{ref} ***in 36*** out of 63 workloads.

Table 4.10 summarizes the observed TDS errors in ***F*** and its components.

Table 4.9: Memory Reference Latency Decompositions for 2 to 10 PEs.

Application	mean I_m/F	mean R_m/F	mean W/F	mean L_c/L_{ref}	mean L_b/L_{ref}
MP3D	5.0%	14.7%	80.3 %	46.9%	53.1%
Dynamic MP3D	4.6%	14.7%	80.7%	48.6%	51.4%
Barnes	10.3%	10.0%	79.7%	42.0%	58.0%
Dynamic Barnes	7.3%	7.5%	85.2%	44.0%	56.0%
Cholesky	5.4%	17.3%	77.3%	60.1%	39.9%
LocusRoute	17.7%	13.1%	69.3%	49.9%	50.1%
PThor	20.7%	23.8%	55.5%	54.0%	46.0%

Table 4.10: Non-Local Reference Errors with $\mathcal{G}_{ideal}$ for 2 to 10 PEs.

Application	mean $\|err(F)\|$	mean $\|err(I_m)\|$	mean $\|err(R_m)\|$	mean $\|err(W)\|$
MP3D	0.03 %	0.43 %+	0.14%	0.03 %
Dynamic MP3D	**0.04%**	0.40%	0.04%	0.03%
Barnes	0.03%	0.10%	0.21%	0.00%
Dynamic Barnes	0.14%+	1.34%+	0.72%+	0.00%
Cholesky	0.08%	0.84%+	0.43%	0%
LocusRoute	**0.09%**	**0.29%**	**0.99%**	0.03%
PThor	0.24%	0.45%	0.31%+	0.14%

$\boldsymbol{E}rr(F)$ was small, closely resembling $err(Ref)$. The component errors were larger, but the largest I, and R_m errors were only +2.79% and +1.61%, respectively.

The errors in $\boldsymbol{F}$ were concentrated in I,,, and R_m. Though W dominated over I,, and ***R,, in*** all workloads, $\Delta(W)$ dominated over $\Delta(\boldsymbol{I,,})$ and $\Delta(R_m)$ in only 11 workloads. The fact that I,, and R_m generally had larger errors than $\boldsymbol{I}$ and $\boldsymbol{R}$ suggests that much of the error in $\boldsymbol{I,,}$ and R_m was due to errors in the miss rates.

Table 4.11 summarizes the observed errors in L_c and L_b.

Table 4.11: Memory Reference Latency Errors with $\mathcal{G}_{ideal}$ for 2 to 10 PEs.

Application	mean $\|err(L_c)\|$	mean $\|err(L_b)\|$
MP3D	0.03 %	0.07%
Dynamic MP3D	0.04%	0.06%
Barnes	0.01%	0.06%
Dynamic Barnes	0.12%-	0.31%-
Cholesky	0.49%	0.45%
LocusRoute	0.18%-	1.19%-
PThor	0.27%	0.41 %

The errors in L_b tended to be larger than those in $\boldsymbol{L,;}$ $\Delta(L_b)$ dominated over $\Delta(L_c)$ ***in 44*** out of 63 workloads. The largest L_c error was -0.91% (for **Cholesky** with eight **PEs)** and the largest L_b error was **-3.06%** (for **LocusRoute** with 10 **PEs).** The four largest L_b **errors were all in LocusRoute**; bus latency errors seem **to** be **the** primary cause of the **large** L_{ref} errors **in LocusRoute**.

We now examine some statistics that are commonly used to parameterize memory system behavior. Cache effectiveness is measured in terms of per-reference miss rates. **We** tabulate this statistic for both for instruction fetches, $m_I = I_m/I$, and data reads, $m_R = R_m/\boldsymbol{R}$. The average number of cache

invalidations per shared write, in r, reflects the degree of data sharing, and the fractional bus occupancy, f_{busy}, reflects the average level of bus traffic. Table 4.12 summarizes the observed TDS errors in these results.

Table 4.12: Memory Statistic Errors with $\mathcal{G}_{ideal}$ for 2 to 10 PEs.

Application	mean $\|err(m_I)\|$	mean $\|err(m_R)\|$	mean $\|err(inv)\|$	mean $\|err(f_{busy})\|$
MP3D	0.44%+	0.13%	0.82%	0.05%
Dynamic MP3D	0.41%	0.06%	0.48%	0.08%–
Barnes	0.10%	0.21%	0.34%	0.02%+
Dynamic Barnes	1.34%+	0.71%+	1.41%+	1.18%–
Cholesky	0.84%+	0.43 %	4.91%	0.70%
LocusRoute	0.30%	1.01%	8.22%-	0.47%
PThor	0.26%	0.25 %+	3.25%	0.24%–

Our study of I_m and R_m suggested that errors were occurring in the miss rates. The table shows that this was indeed the case, though the errors were generally too small to be of practical concern. The largest m_I error was only $\pm 2.78\%$, and the largest m_R error was only $+1.67\%$. Positive errors outnumbered negative ones for both miss rates.

The invalidation ratio exhibited large errors. inv errors as large as -20.9% were measured for LocusRoute. We known that TDS did not significantly distort W_{sh}, so the inv errors must be due to errors in the number of invalidations. Invalidations are caused by communication between PEs and false sharing between caches. By simulating LocusRoute with one-word cache lines, we were able to eliminate false-sharing invalidations. The large inv errors persisted, indicating that they are due to errors in communication misses, not false sharing misses.

4.3.7 Address translation (TLB) latency

The third and final source of latency to consider is address translation. We denote the total latency due to TLB faults by L_{TLB}. The latency of a TLB fault was fixed at at 50 cycles in our simulator, so L_{TLB} is simply 50 times the number of T'LB faults. Errors in L_{TLB} were due entirely to errors in the number of TLB faults, not the average fault latencies.

Table 4.13 summarizes the TLB behavior of the applications and the errors due to trace-driven simulation.

Table 4.13: TLB Errors with $\mathcal{G}_{ideal}$ for 2 to 10 PEs.

Application	mean L_{TLB}/C	mean $\|err(L_{TLB})\|$	mean $\|err(L_{TLB,i})\|$	mean $\|err(L_{TLB,d})\|$
MP3D	6.56%	***0.40%***	***0.77%***	***0.40%***
Dynamic MP3D	6.61%	0.41%	0.96%	0.37%
Barnes	***0.07%***	***2.87%***	***4.85%***	***3.46%***
Dynamic Barnes	***0.05 %***	***4.45%***	***7.27%***	***4.39%***
Cholesky	0.78%	0.64%	2.45%	0.55%
LocusRoute	6.50%	0.58%+	0.81%+	0.55%+
PThor	14.8%	0.80%	0.90%	0.81%

The large TLB penalty for **PThor** reflects the large working-set of that application.

The mean $|err(L_{TLB})|$ was 1.48%, and the largest L_{TLB} error was -9.57%. In the table, we

distinguish TLB faults on instructions ($L_{TLB.i}$) from TLB faults on data ($L_{TLB.d}$). The instruction TLB faults found to show somewhat larger errors.

Only Barnes and **Dynamic Barnes** had any workloads with $|err(L_{TLB})| > 5\%$. For these applications, the impact of TLB latency errors on err (C − I) was mitigated by the low fault rates.

4.4 Errors with $\mathcal{G}_{1/4}$

We have observed sizable TDS errors for a number of important simulation results. However, the trace-generation environment used, $\mathcal{G}_{\text{ideal}}$, was quite different from the target environment. To determine whether the magnitude of the errors reflects the vast differences between $\mathcal{G}_{\text{ideal}}$ and $\mathcal{T}$, we ran a second set of experiments with $\mathcal{G}_{1/4}$, which was almost identical to the target environment. A total of 83 workloads were simulated using $\mathcal{G}_{1/4}$. (Due to limited simulation resources, we were unable to simulate **Cholesky** with more than eight PEs.)

As noted in Section 4.1, all uniprocessor workloads were accurately simulated, as were workloads that used the applications **Ocean** or **Water.** The remaining 61 workloads behaved generally like their $\mathcal{G}_{\text{ideal}}$ counterparts except that the errors were slightly ***larger.***

The following sections present error statistics for the simulations of the $\mathcal{G}_{1/4}$ traces and compare them with the corresponding statistics for the $\mathcal{G}_{\text{ideal}}$ traces.

4.4.1 Elapsed time

Table 4.14 presents error data for T_ϵ (and its components) for simulations of $\mathcal{G}_{1/4}$ traces. (Compare with Tables 4.1 and 4.2.)

Table 4.14: Elapsed Time Errors with $\mathcal{G}_{1/4}$ for *2* to 10 PEs.

Application	mean $\lvert err(T_\epsilon)\rvert$	max $\lvert err(T_\epsilon)\rvert$	mean $\lvert err(T_{par})\rvert$	mean $\lvert err(T_f)\rvert$
MP3D	***0.05 %***	+o. 14%	0.10%	9.89%
Dynamic MP3D	0.12%+	+0.14%	0.23 %+	5.19%+
Barnes	0.03%	-0.09%	0.03%	8.12%
Dynamic Barnes	0.69%+	+1.07%	0.72%+	4.37%−
Cholesky	***2.74%***	+5.63%	6.79%	0.83%
LocusRoute	**0.44%-**	-1.58%	1.22%−	30.9%+
PThor	0.40%+	+0.59%	2.22%+	21.9%

As with the $\mathcal{G}_{\text{ideal}}$ traces, the majority of the timing errors were small or insignificant. T_ϵ errors increased slightly on average; for 35 out of 61 workloads, $|err(T_\epsilon)|$ was larger in $\mathcal{G}_{1/4}$ than it was in $\mathcal{G}_{\text{ideal}}$. The overall mean error increased from 0.40% ($N = 61$) to **0.57%,** the largest T_ϵ error increased from +2.03% to +5.63%, and the fraction of workloads with $|err(T_\epsilon)| < 0.2\%$ increased from 52% to 54%.

The T_ϵ errors continued to be biased toward the positive side in $\mathcal{G}_{1/4}$: 45 out of 61 workloads had positive $err(T_\epsilon)$**.**

As before, we decomposed the elapsed time into parallel and serial components. Once again, errors in T_{par} dominated $err(T_\epsilon)$ for most workloads. T_i continued to be undistorted, and errors in T_f dominated $err(T_\epsilon)$ **in only six** workloads, all of which involved **LocusRoute. The** T_{par} ***errors grew in 34*** out of 61

workloads, and the mean $|err(T_{par})|$ grew from 1.13% (N = 61) to 1.45%. The largest $err(T_{par})$ fell, however, from −18.9% to +13.9%.

The number of workloads with $|err(T_f)| > 10\%$ grew from 16 to 22. The overall mean $|err(T_f)|$ fell, however, from 86.5% (N = 61) to 12.0%. The decline in the mean was due largely to the two **PThor** workloads that had huge errors when $\mathcal{G}_{ideal}$ **was used. As** before, the majority of the **PThor** workloads had $err(T_f) = 0$.

The ranking of the applications by mean $|err(T_{par})|$ was nearly the same with the $\mathcal{G}_{1/4}$ traces as it was with the $\mathcal{G}_{ideal}$ ones, suggesting that application characteristics outweigh trace-generation differences in determining the size of TDS timing errors. The large-error applications continued to be distinguished by higher $err(T_\epsilon)$ and $err(T_{par})$ than the small-error applications. The ratio between the variances of $n\,err(T_\epsilon)$ for the two groups of applications rose from 190 to 330, while the ratio of the variances of $n\,err(T_{par})$ fell, from 1600 to 690.

4.4.2 Cycle counts

Table 4.15 presents error data for C and its components from simulations using the $\mathcal{G}_{1/4}$ traces. (Compare with Tables 4.3 and 4.4.)

Table 4.15: Cycle Count Errors with $\mathcal{G}_{1/4}$ for 2 to 10 PEs.

Application	mean $\|err(C)\|$	max $\|err(C)\|$	mean $\|err(I)\|$	mean $\|err(C-I)\|$
MP3D	0.08%	+0.26%	0.03%-	0.13%
Dynamic MP3D	0.20%+	+0.27%	0.03%	0.39%+
Barnes	0.03 %	-0.09%	0.00%	0.06%
Dynamic Barnes	0.72%+	+1.11%	0.00%	1.34%+
Choleskg	4.53%	+ 10.0%	0%	7.10%
LocusRoute	0.84%-	-2.97%	0.02%	1.64%−
PThor	1.24%+	+1.78%	0.20%+	2.17%+

$Err(C)$ continued to resemble $err(T_\epsilon)$ and $err(T_{par})$. Its magnitude fell between theirs in 54 out of 61 workloads. The mean $|err(C)|$ decreased overall, from from 1.55% (N = 61) to 0.98%. This decline was largely due to a decrease **in Cholesky's** $err(C)$**,** yet **Cholesky** continued to exhibit the largest cycle-count errors of any application.

The applications fell into the same groups as before, but the ratio of the variances of $n\,err(C)$ fell from 5200 to only 460. $Err(C)$ was biased to the positive side even more strongly than before, this time by a 43 : 18 margin.

We again decomposed of C into I and $C - I$. *As* before, I was hardly distorted by TDS. The errors in the idle cycles, C − I, accounted for most of $err(C)$. The largest $|err(I)|$ climbed from +0.37% to −0.44%, while the largest $|err(C-I)|$ fell from -24.3% to +14.4%. The overall mean of $|err(I)|$ remained at 0.04% (-I- = 61), whereas that of $|err(C-I)|$ fell from 2.56% (N = 61) to 1.66%. Positive C' − I errors outnumbered negative ones by a 43 : 18 margin.

As before, we attribute much of the error in T_ϵ and T_{par} to error in the **idle** time, C − I. Decomposing $C - I$ as before, we find a familiar pattern: most of the idle time comes from memory reference latency, but most of the error in the idle time comes from synchronization latency. In $\mathcal{G}_{1/4}$, synchronization latency errors dominated $err(C - I)$ in 55 out of 61 workloads, reference latency errors dominated in five workloads, and translation latency errors dominated in only in one workload.

4.4.3 Synchronization latency

Table 4.16 summarizes the synchronization latency errors for the $\mathcal{G}_{1/4}$ traces. (Compare with Tables 4.5 and 4.6.)

Table 4.16: Synchronization Latency Errors with $\mathcal{G}_{1/4}$ for 2 to 10 PEs.

Application	mean $\|err(L_\sigma)\|$	mean $\|err(L_q)\|$	mean $\|err(L_d)\|$	mean $\|err(L_\mu)\|$
MP3D	5.95%+	0%	6.14%+	12.2%
Dynamic MP3D	39.1%+	0%	45.9%+	3.87%
Barnes	0.35%	0%	0.25%	22.2%−
Dynamic Barnes	36.9%+	0%	42.8%+	10.9%−
Cholesky	15.1%	15.6%	98900.%+	30.6%−
LocusRoute	57.3%	59.6%	1890.%	22.9%
PThor	10.9%+	12.8%+	10.9%+	9.22%+

$Err(\boldsymbol{L},)$ remained substantial in $\mathcal{G}_{1/4}$, exceeding 5% in 45 out of 61 workloads. The overall mean $err(L_\sigma)$ fell from 33.2% (N = 61) to 24.0%. The Largest L_σ error fell from +231% to +110.%.

The L_σ error signs went from 67% positive to 77% positive. The sign changes in **Cholesky** were particularly notable; **Cholesky** went from 100% negative $\epsilon rr(L_\sigma)$ ***to*** 71% positive $\epsilon rr(\boldsymbol{L}_\sigma)$.

Decomposing $\epsilon rr(L_\sigma)$ as before, $\Delta(\boldsymbol{L}_q)$ dominated in 16 workloads, $\Delta(\boldsymbol{L}_d)$ dominated in 39 workloads, and $\Delta(L_\mu)$ in the remaining six workloads. This decomposition is similar to the decomposition of $\epsilon rr(\boldsymbol{L}_\sigma)$ for $\mathcal{G}_{\text{ideal}}$, except that the impact of L_q errors increased at the expense of errors in L_a and L_μ.

$Err(\Sigma)$ for $\mathcal{G}_{1/4}$ (not shown in the table) was similar in size to the Σ errors for $\mathcal{G}_{\text{ideal}}$. The largest Σ error fell slightly, from − 1.64% to only − 1.17%. As with $\mathcal{G}_{\text{ideal}}$, the L_σ errors were clearly due to errors in the average synchronization latencies rather than errors in the number of synchronizations.

The L_q errors changed dramatically in the new trace-generation environment. $\mathcal{G}_{1/4}$ measured L_q more accurately than $\mathcal{G}_{\text{ideal}}$ did, and the signs of $\epsilon rr(\boldsymbol{L}_q)$ ***shifted,*** from being mostly negative in $\mathcal{G}_{\text{ideal}}$ to being mostly positive in $\mathcal{G}_{1/4}$. The number of positive $\boldsymbol{L}_q$ errors jumped from one to 18, while negative errors fell from 24 to seven.

The $\boldsymbol{L}_d$ errors in $\mathcal{G}_{1/4}$ were similar to those seen in $\mathcal{G}_{\text{ideal}}$. The largest L_d error was 306.000%. The most important change from $\mathcal{G}_{\text{ideal}}$ was that more workloads had $\epsilon rr(\boldsymbol{L}_d) = 0$. Where $\mathcal{G}_{\text{ideal}}$ had $err(L_d) = 0$ for only one workload, $\mathcal{G}_{1/4}$ had $err(L_d) = 0$ for nine workloads: seven workloads of **LocusRoute** and two of **Cholesky.** As a result, only six workloads (all involving **Cholesky** or **LocusRoute**) had $|\epsilon rr(L_d)| > 200\%$, and $|\epsilon rr(L_d)|$ decreased in 35 out of 61 workloads.

As before, the L_d errors were mostly positive, but positive errors outnumbered negative ones by a 46 : 6 margin, down from the 60 : 2 margin observed for the $\mathcal{G}_{\text{ideal}}$ traces.

The L_μ errors for $\mathcal{G}_{1/4}$ were similar to those for $\mathcal{G}_{\text{ideal}}$, but were slightly more evenly balanced between positive and negative errors. The largest L_μ error was +83.8% (as opposed to +52.6% in $\mathcal{G}_{\text{ideal}}$) and the mean $|err(L_\mu)|$ rose from 13.7% (N = 61) to 15.5%. Negative errors continued to outnumber positive ones, but by a narrower margin, 35 : 26 instead of 41 : 22.

Next we examine the errors in memory reference latency and other memory statistics.

4.4.4 Memory reference latency

Table 4.17 summarizes the errors in reference latency and reference counts for $\mathcal{G}_{1/4}$. (Compare with Tables 4.7 and 4.8.)

Table 4.17: Memory Reference Errors with $\mathcal{G}_{1/4}$ for 2 to 10 PEs.

Application	mean $\lvert err(L_{ref})\rvert$	mean $\lvert err(Ref)\rvert$	mean $\lvert err(R)\rvert$	mean $\lvert err(W_{sh})\rvert$
MP3D	0.05 %	0.02%-	0.02%-	0.02%-
Dynamic MP3D	0.09%-	0.03%	0.03%	0.04%
Barnes	0.02%	0.00%	0.00%	0.00%
Dynamic Barnes	0.36%-	0.00%	0.00%	0.00%
Cholesky	0.40%	0%	0%	0%
LocusRoute	0.22%	0.02%	0.03%	0.05%
PThor	0.46%-	0.20%-	0.24%-	0.19%-

The L_{ref} errors were similar to those 0f $\mathcal{G}_{ideal}$. For one thing, the errors were small; the largest L_{ref} error fell from – 1.99% to +1.35%, and the mean $\lvert err(L_{ref})\rvert$ fell from 0.25% (N = 61) to 0.22%. As with the $\mathcal{G}_{ideal}$ traces, the large-error applications suffered the largest L_{ref} errors, but the ratio between the $nerr(L_{ref})$ variances of the two groups of applications fell from 130 to 54.

As with the $\mathcal{G}_{ideal}$ traces, $err(Ref)$ was quite small; the largest Ref error rose slightly, from +0.38% to – 0.44%. The components of Ref had similarly small errors.

As before (for the large-error applications at least) $err(L_{ref})$ was due mostly to errors in the average reference latency rather than errors in the number of references. **PThor** stood out once again for its large Ref errors.

Table 4.18 summarizes the observed TDS errors in F and its components. (Compare with Table 4.10.)

Table 4.18: Non-Local Reference Errors with $\mathcal{G}_{1/4}$ for 2 to 10 PEs.

Application	mean $\lvert err(F)\rvert$	mean $\lvert err(I_m)\rvert$	mean $\lvert err(R_m)\rvert$	mean $\lvert err(W)\rvert$
MP3D	0.03%	0.38%	0.09%-	0.02%-
Dynamic MP3D	0.04%	0.43%	0.08%	0.04%
Barnes	0.02%	0.09%	0.16%	0.00%
Dynamic Barnes	0.08 %	0.64%-	0.45 %	0.00%
Cholesky	0.06%	0.39%	0.28%	0%
LocusRoute	0.11%	0.13%	0.76%	0.02%
PThor	0.19%-	0.49% –	0.24%-	0.14%-

Once again, $err(F)$ was similar in magnitude to $err(Ref)$, while $err(I_m)$ and $err(R_m)$ were larger, and $err(W)$ was smaller. The largest in I, and R_m errors were -1.37% and +1.61%, respectively.

Table 4.19 summarizes the observed errors in L_c and L_b. (Compare with Table 4.11.)

L_b errors dominated $err(L_{ref})$ more strongly than they did in $\mathcal{G}_{ideal}$: L_b dominated over L_c in 47 out of 61 workloads. The largest L_c error rose from -0.91% to + 1.62% and the largest L_b error fell from -3.06% to – 1.26%. The mean $err(L_b)$ for LocusRoute declined in $\mathcal{G}_{1/4}$, and LocusRoute no longer exhibited the largest L_b errors.

Table 4.20 summarizes the observed errors in the miss rates, invalidation rates, and bus occupancy. (Compare with Table 4.12.)

Table 4.19: Memory Reference Latency Errors with $\mathcal{G}_{1/4}$ for 2 to 10 PEs.

Application	mean $\|err(L)\|$	mean $\|err(L_I)\|$
MP3D	**0.04%-**	**0.07%**
Dynamic MP3D	0.06%–	0.11%–
Barnes	**0.01%**	**0.04%**
Dynamic Barnes	**0.15%-**	**0.49%-**
Cholesky	**0.43%**	**0.34%**
LocusRoute	0.12%	0.41%
PThor	0.34%-	0.59%-

Table 4.20: Memory Statistic Errors with $\mathcal{G}_{1/4}$ for 2 to 10 PEs.

Application	mean $\epsilon rr\|(m_I)\|$	mean $\|err(m_R)\|$	**mean** $\|err(inv)\|$	**mean** $\|err(f_{busy})\|$
MP3D	0.37%+	0.08%+	0.27%	0.04%
Dynamic MP3D	**0.42%**	0.07%	0.36%	0.13%–
Barnes	**0.09%**	**0.16%**	0.20%	0.02%
Dynamic Barnes	0.64%+	0.45%	1.15%+	0.75%–
Choleskp	0.39%+	0.28%+	5.73%	2.68%–
LocusRoute	0.26%	0.74%	6.24%	0.45%
PThor	0.31%	0.18%+	2.03%	0.56%–

The miss rate errors were once again too small to be of concern to most studies. The largest m_I error declined from -2.78% to – 1.37%, and the largest m_R error declined from + 1.67% to + 1.61%.

Negative m_I errors outnumbered positive ones by a 39 : 22 margin, and negative m_R errors outnumbered positive ones by 35 : 26. (Before, positive errors dominated in both miss rates.)

Once again, invalidation rates exhibited larger errors. $Err(inv$ I magnitudes above 14% were observed **in both Choleskp** and **LocusRoute**. As with the $\mathcal{G}_{ideal}$ traces, the inv errors reflect errors in the total number of invalidations.

4.4.5 Address translation (TLB) latency

Table 4.21 summarizes the TLB behavior of the applications and the errors caused by trace-driven simulation. (Compare with Table 4.13.)

Table 4.21: TLB Latency Errors with $\mathcal{G}_{1/4}$ for 2 to 10 PEs.

Application	mean $\|err(L_{TLB})\|$	**mean** $\|err(L_{TLB,i})\|$	**mean** $\|err(L_{TLB,d})\|$
MP3D	0.40%	1.28%	0.34%
Dynamic MP3D	**0.28%**	0.83%	***0.29%***
Barnes	4.61%	5.83%	4.24%
Dynamic Barnes	**3.78%**	9.34%	3.90%
Cholesky	1.59%	***2.83%***	1.53%
LocusRoute	0.83%	0.78%	0.89%
PThor	0.48%-	0.55%-	0.47%–

As before, **only Barnes** and **Dynamic Barnes** had workloads with $|err(L_{TLB})| > 5\%$. The impact of TLB latency on $\epsilon rr(C$ – I) was again mitigated by the low fault rates in these applications. Once

again, the TLB fault rates for instruction references had a larger errors than those for data references.

Chapter 5

Discussion

This chapter discusses the patterns in the experimental results and, where possible, provides explanations for them.

5.1 Cases with accurate trace-driven simulation

We have argued in Section 1.3 that accurate trace-driven simulation across multiple environments is possible only with timing-independent trace-sets. The experimental data support this claim: all cases with timing-independent trace-sets were simulated accurately, and all cases with inaccurate simulation had timing-dependent trace-sets.

As noted in Section 4.1, all uniprocessor cases were accurately simulated The uniprocessor trace-sets were timing-independent because none of the workloads generated addresses that were dependent on a clock reading. (In uniprocessor simulations there is only a single thread of execution, so the clock is the only source of timing dependencies.)

Additionally, the **Ocean** and **Water** workloads were accurately simulated in all cases. The trace-sets from these workloads were timing-independent because not only did these workloads lack clock-dependencies, but their inter-process timing dependencies were confined to simple synchronization constructs such as locks and barriers. The synchronization constructs were abstracted, leaving no timing dependencies in the trace-sets.

Aside from the two groups described above, every case was, to some degree, inaccurately simulated. Each inaccurately-simulated case had timing-dependencies in its trace-set caused by inter-process timing dependencies. Dynamic scheduling was a major source of timing dependencies. Except for **MP3D**, all the workloads that were inaccurately simulated used FCFS scheduling to some extent. In FCFS-scheduled workloads, task assignments depend on memory system timings. Such dependencies are inevitably reflected in the trace-set.

Both of the accurately-simulated workloads **(Ocean** and **Water) used static** scheduling. The only statically-scheduled workload that was inaccurately simulated was **MP3D**.

5.2 The MP3D workload

The timing-dependencies in **MP3D** were not due to FCFS scheduling; rather, they were due to races in its collision-pairing algorithm. (See [1 1] for an overview of the **MP3D** algorithm.) In **MP3D**, collision

pairings for each space cell are determined by the order in which the particles occupying the cell are updated. Different particles occupying the same cell may be updated by different PEs, and memory reference delays can alter this order. Thus, the pairings depend on the environment. [1]

The indeterminacy of the MP3D collision pairings does not substantially affect the accuracy of MP3D itself, since its algorithm seeks to approximate random collision pairing. While pairing changes alter the trajectories of individual molecules, they have no statistically-significant effects on the macro-scale results of MP3D. However, changes in the collision pairings perturb the execution path, making the trace-set timing-dependent. The trace-sets are therefore invalid for simulations of environments with different memory system timings.

Other workloads, notably **Dynamic MP3D** and **LocusRoute, are known to have** similar races. Without rewriting the workloads, however, the effects of those races on simulation accuracy are impossible to isolate from the effects of dynamic scheduling.

5.3 Results that were accurately simulated

T_i, the initialization time, was the only result that was accurately simulated for all workloads. During initialization, each workload has only a single thread of execution, so, as in a uniprocessor workload, there are no timing-dependencies.

5.4 Results with large errors

The observed errors in the small-error workloads (MP3D, **Dynamic MP3D**, and **Barnes)** were on the order of 0.1% for most of the results studied. Such errors are negligible for most purposes.

The errors were small because the the perturbations caused by individual timing-dependencies in a workload are very slight. Such perturbations affect the results substantially only when their influence is biased (toward positive or negative errors) or when the number of events reflected in the result is very small. Over large numbers of events, the total effect of these perturbations grows slowly unless the individual effects are biased. Execution path perturbations, such as those due to collision-pairing races and FCFS-scheduhng , have no systematic effect on most simulation results.

A few results (such as synchronization latencies, TLB latencies, and T_f) were either systematically influenced by execution path perturbations or else were derived from relatively small numbers of events. For these results, large errors were observed. even for small-error workloads.

We will now analyze the errors exhibited by these results. First we will analyze the errors in the components of L_σ, namely: L_d, L_q, and L_μ. Then we will discuss the effects these errors had on L_σ itself. Then we will discuss the errors in TLB latency and T_f.

5.4.1 Errors in L_d

The largest errors observed for any result were for the non-spin delay latency, L_d. *Errors* of over 2,000,000% were observed. The L_d ***errors*** were overwhelmingly positive.

L_d measures the time spent at the barriers and completion waits that mark the end of computational steps. L_d is thus a good indicator of the load balance in a workload. The large errors in L_d indicate

[1] The version of MP3D used in [11] had additional races, caused by simultaneous updates of the space array by different PEs. For this study, the update races were eliminated by locking space array cells during updates.

that trace-driven simulation distorts the load balance of the workload. Specifically, positive L_d errors indicate that the load balance, as measured by trace-driven simulation, was poorer than it was in the direct simulation.

Large positive errors were observed in FCFS-scheduled workloads because perturbations of the execution path tend to disrupt the load balance. The goal of FCFS scheduling is to optimize load balance, so that trace-sets of workloads with FCFS scheduling tend to reflect schedules that have good load-balance for the trace-generation environment. When the simulator applies the memory latencies of the target environment to a trace-set, the execution times of individual tasks change. However, the scheduling decisions made during trace-generation are hard-coded into the trace-set. (In effect, trace-driven simulation forces the workload to use a static schedule.) Unless there is a high correlation between task execution times in the trace-generation environment and the target environment, the target environment will appear to have poor load balance. In direct simulation, tasks are dynamically load balanced for the target environment, so less time is spent at barriers and L_d is smaller.

The largest L_d errors occurred in workloads where FCFS scheduling gave good load balance, that is, where L_d/C was small. The two applications with the best load balance, **Cholesky** and **LocusRoute** accounted for all 21 cases with $|err(T_e)| > 160\%$.

Ten simulations had $err(L_d) = 0$, even though there were errors in other latencies. These ten simulations involved either **Cholesky** or **LocusRoute**, which used barriers only to synchronize the start of the parallel phase. The barrier waits in these applications were very short, so there was a reasonable chance of having no execution path differences before the barrier.

5.4.2 Errors in L_q

Three of the workloads used task-queues. Errors of up to 117% were observed in queueing latencies of these workloads. The trace-sets generated in the $\mathcal{G}_{\text{ideal}}$ environment produced L_q ***errors*** that were predominantly negative, while the $\mathcal{G}_{1/4}$ traces produced L_q errors that were mostly positive.

The large errors in L_q were due to the fact that shared-memory spins, used to implement queueing operations, were not resimulated. The trace-driven simulator set the duration of each spin in the target environment to be the number of cycles the spin lasted in the trace-generation environment. This technique provides a reasonable approximation for spin latencies only when the two environments execute non-spin instructions at the same average rate. If the trace-generation environment executes instructions more slowly than the target environment, the simulated spins tend to complete too late (relative to events occurring on other PEs in the target environment) and positive L_q errors result. Similarly, if the trace-generation environment executes more quickly than the target environment, negative L_q errors will result.

In the target environment, the average CPI (exclusive of synchronization) ranged from 1.72 to 3.38, depending on the workload. The CPIs in $\mathcal{G}_{\text{ideal}}$ were exactly 1, due to the idealized memory system, so large negative errors were observed in L_q with the $\mathcal{G}_{\text{ideal}}$ traces. In $\mathcal{G}_{1/4}$ the average CPIs were higher than the target environment, due to the smaller caches. Positive L_q errors were observed with most of the $\mathcal{G}_{1/4}$ traces, except for **LocusRoute**. The negative errors for these traces may be due to a drop in the CPIs of active PEs when many PEs are spinning.

Is there an approach that would simulate task-queue latencies more accurately?

The best solution would be to abstract the task-queue operations out of the trace-sets. This is difficult to do because task-queue operations are complex. In **PThor** for instance, every task has a preferred PE, but idle PEs may "steal" tasks from the queues of busy PEs. The code to de-queue a task is spread over 180 lines of C code. Even if task-queue operations were abstracted from the trace-sets, their side-effects would introduce complex timing-dependencies, so that accurate simulation would not be achieved.

Another approach would be to multiply the length of each spin by the average CPI ratio, $CPI_{\mathcal{T}}/CPI_{\mathcal{G}}$. However, the CPI of a workload is not a constant; it varies substantially during execution. It is also not clear that the CPI ratio can be accurately estimated without simulating the workload in both evironments.

A third approach would be to treat the spins as normal memory references, rather than as synchronization. This has accuracy problems similar to those of the approach taken in this study, since the CPI of a spinning PE is not the same as that of one that is doing useful work. (The working set of a spinning PE is only a few addresses, so for long spins it is likely to have a miss rate very close to zero.)

5.4.3 Errors in L_μ

L_μ basically measures lock contention. Lock contention was rare in the workloads we studied; L_μ never ***exceeded 0.67%*** of C. Errors as large as 83.8% were observed in L_μ, and the errors were more often negative than positive, especially in **Barnes** and **Dynamic Barnes**.

The large L_μ errors are undoubtedly due to the fact that most of the workloads had few lock operations: one complete run of **LocusRoute** had only 5570 such operations. Most of the individual lock latencies are zero, so an error in the latency of an individual lock operation could noticeably affect L_μ. However, this does not explain the sign-bias in $err(L_\mu)$. There are two reasons why L_μ might tend to be smaller in the trace-driven simulations: either the locks were held for a smaller fraction of the total execution time or lock accesses by different PEs were less correlated in time. We do not know the relative importance of these two sources of error.

5.4.4 Errors in L_σ

The errors in L_σ were a composite of the errors in L_d, L_q, and L_μ. In **Cholesky and LocusRoute**, the majority of L_σ was due to queueing. L_q errors dominated in **Cholesky and in the $\mathcal{G}_{1/4}$ simulations** of **LocusRoute**, so the sign-biases of $err(L_\sigma)$ and $err(L_q)$ were the same for these simulations. In all applications except **Choieskp** and **LocusRoute**, the majority of L_σ was due to non-queue delays. L_d errors dominated in simulations of all these applications except **Barnes,** producing positive L_σ errors. The L_i errors dominated in **Barnes,** due to high lock contention in the application; as a result, **Barnes** had small, negative L_σ errors.

5.4.5 Errors in TLB latencies

Since TLB faults are very infrequent, their errors were computed based on small numbers of events — less than 10^6 faults per application. The only simulations with significant sign-bias were **LocusRoute (in $\mathcal{G}_{ideal}$)** and **PThor** (in $\mathcal{G}_{1/4}$), and the errors in these simulations averaged less than 1%. Because instructions have relatively low TLB fault rates compared to data references, the $L_{TLB,i}$ ***errors*** were based on an even smaller number of events, and had somewhat larger errors. We expect that the L_{TLB} errors in the remaining simulations would decline substantially if longer workloads were simulated.

5.4.6 Errors in T_f

While the finalization stage, like the initialization stage, has only a single thread of execution, it was not accurately simulated for most workloads. The reason for this is that the simulator is not stateless. The state of the simulator at the start of the initialization stage is the same for all simulations, but at the start

of the finalization state, the trace-driven simulator is in a state the reflects any errors that occurred during the parallel stage.

The relative errors in T_f were largest in the applications where T_f itself was small, so that the errors were derived from small samples of events. Thus, despite the large relative errors, the impact on T_ϵ was seldom **significant.** There were only 11 simulations (out of 124) in which $|\Delta(T_f)|$ exceeded 10.000 cycles.

5.4.7 Errors in $C - I$ and related results

Large errors in the simulated idle time, $C - I$, are the distinguishing feature of the large-error workloads. The sign-biases in T_ϵ, T_{par}, and C' all reflect the sign-biases of the $C - I$ results, and the relative magnitudes of these results usually reflect dilution of $|err(C - I)|$ by serial time and execution latencies. Large $C - I$ errors caused by T_ϵ, T_{par}, and C errors in large-error workloads to be distinctly larger than the corresponding errors in the small-error workloads.

The T_{par} errors in the $\mathcal{G}_{ideal}$ simulations of **Cholesky** were exceptional because they did not reflect the much larger errors in C' and $C' - I$ for the application. This occurred because the child processes **in Cholesky** finish over a relatively long period of time. In the $\mathcal{G}_{ideal}$ simulations, the parallel times of individual processes were consistently too short, producing large negative errors in C'. T_{par} was less affected by these errors because it reflects the error in the parallel time of the process which ran longest in the trace-driven simulation. This effect was much less prominent **in** $\mathcal{G}_{1/4}$ simulations of **Cholesky** because the parallel times of individual processes were likely to sometimes be too short and sometimes be too long.

The errors in $C - I$ were dominated by synchronization latency errors in 108 out of 124 simulations, including all but two of the 70 simulations of large-error applications. The errors in L_{ref} and L_{TLB} were small or were based on small numbers of events, so they did not greatly impact $C - 1$.

$Err(C - I)$ was smaller in the small-error applications, either because the L_σ errors were dominated by L_{μ} errors **(Barnes)** or because the application did not have appreciable amounts of synchronization latency (**MP3D** and **Dynamic MP3D**).

The large-error applications were the ones that either had large amounts of queueing latency **(Choleskp** and **LocusRoute**) or else used dynamic scheduling yet had relatively poor load-balance **(Dynamic Banes** and **PThor**). These features produced large $\Delta(L_\sigma)$ errors, which in turn caused large errors in $C - I$ and related results.

5.5 Differences between trace-generation environments

Comparison of the results obtained from the two trace-generation environments shows that a more detailed trace-generation environment will not necessarily produce **more accurate results. In** most cases, the $\mathcal{G}_{ideal}$ trace-sets gave results that were roughly as accurate as those from the $\mathcal{G}_{1/4}$ traces-sets. For most of the results we studied, trace-driven simulation accuracy was determined more by the workload than by the trace generation environment.

The most noticeable differences between the $\mathcal{G}_{ideal}$ data and the $\mathcal{G}_{1/4}$ data were in the measurements of queueing delays, where accuracy was tied to the CPI of the trace-generation environment.

Chapter 6

Conclusions

Trace-driven simulation is often performed using trace-sets from environments other than the one being simulated. To the extent that the simulator cannot compensate for the timing-dependencies in the trace-sets, errors are introduced in the simulation results.

For studies of uniprocessor memory systems, timing-dependencies are not a problem. However, the execution path of a multiprocessor workload is likely to depend on memory latencies. Accurate simulation of multiprocessor workloads is possible in asynchronous simulation of statically-scheduled, deterministic applications. The simple synchronization primitives used by such workloads can be abstracted from the trace-sets to eliminate timing dependencies. Memory system differences are not reflected in the resulting trace-sets.

For the workloads we studied, only a few results were greatly distorted by trace-driven simulation. The most important distortions occurred in synchronization latencies. Large positive errors were observed in the barrier latencies of dynamically-scheduled applications because the simulated task assignments were set by the trace. Large errors of both signs were observed in task-queue latencies because the trace-generation environments had different CPIs than the target environment. The resulting synchronization latency errors affected the accuracy of the cycle count and execution time, especially in workloads that had poor load-balance or large task-queue latencies.

For results not tied to simulation latency, the errors introduced by changes to the execution path had little or no sign-bias. For such results, as the number of events sampled grew, the relative error became very small. Large errors were observed in such results only when the result was based on a relatively small number of events.

Application characteristics seemed to play a greater role in detemining the accuracy of the simulation results than the trace-generation environment did The $\mathcal{G}_{1/4}$ traces did not produce results that were any more accurate than the results produced from the $\mathcal{G}_{\mathbf{ideal}}$ traces, even though the memory system of $\mathcal{G}_{1/4}$ was more similar to $\mathcal{T}$.

While it is difficult to generalize from our limited data, researchers and memory system designers should be aware of the pitfalls in multiprocessor trace-driven simulation. Where possible, direct simulation should be used, or else the trace generation environment should exactly match the environment being simulated. If neither approach is possible, asynchronous trace-driven simulation should be used, and statically-scheduled, deterministic workloads should be preferred over dynamically-scheduled or nondeterministic workloads, which produce timingdependent trace-sets.

If timing-dependent trace-sets are unavoidable, suitable accuracy may still be achieved for speedups and other global measurements. On the other hand, results that are either strongly influenced by syn-

chronization latency must be treated with suspicion. Because results based on small numbers of events generally have poor accuracy, trace-driven simulation is poorly suited to studies of infrequent events and minor architectural features.

Chapter 7

Acknowledgments

The authors wish to thank those who helped make this work possible. Margaret Martonosi helped port PThor to run under Tango Lite, and Josep Torrellas and Jaswinder P. Singh made useful suggestions on early drafts of this paper. Members of the SPLASH group at Stanford laid the groundwork for this study, by collecting and documenting the SPLASH applications.

This work was supported in part by a Hertz Foundation graduate fellowship and in part by DARPA contract N00039-91-C-0138. This support is gratefully acknowledged.

Bibliography

[1] Anant Agarwal, R. L. Sites, and Mark Horowitz. ATUM: A New Technique for Capturing Address Traces Using Microcode. In *Proceedings of the 13th international Symposium on Computer Architecture,* June 1986.

[2] Philip Bitar. A critique of trace-driven simulation for shared-memory multiprocessors. ***Cache and Interconnect Architectures,*** pages 27-52, 1990.

[3] Helen Davis, Stephen R. Goldschmidt, and John Her-messy. Tango: A multiprocessor simulation and tracing system. Technical Report CSL-TR-90439, Stanford University Computer Systems Laboratory, July 1990.

[4] Helen Davis, Stephen R. Goldschmidt, and John L. Hennessy. Multiprocessor simulation and tracing *using* Tango. In ***Proceedings of the 1991 International Conference on Parallel Processing,*** August 1991.

[5] Susan J. Eggers and Randy H. Katz. The characterization of sharing in parallel programs and its application to coherency protocol evaluation. In ***Proceedings of the 15th Annual International Symposium on Computer Architecture,*** pages 373-382, June 1988.

[6] Stephen R. Goldschmidt and Helen Davis. Tango introduction and tutorial. Technical Report CSL-TR-90-410, Stanford University, January 1990.

[7] Mark A. Holliday and Carla S. Ellis. Accuracy of memory reference traces of parallel computations in trace-driven simulation. Technical Report CS-1990-8, Duke University CS Department, July 1990.

[8] Eric J. Koldinger, Susan J. Eggers, and Henry M. Levy. On the validity of trace-driven simulation for multiprocessors. In ***Proceedings of the 18th Annual International Symposium on Computer Architecture,*** pages 244-253, May 199 1.

[9] E. L. Lusk and R. A. Overbeek et ***al. Portable Programs for Parallel Processors.*** Holt, Rinehart and Winston, Inc., 1987.

[10] Jaswinder Pal Singh, Chris Holt, Takashi Totsuka, Anoop Gupta, and John L. Hermessy. Load balancing and data locality in hierarchical N-body methods. Technical Report CSL-TR-92-505, Stanford University, 1992.

[11] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford parallel applications for shared-memory. Technical Report CSL-TR-91469, Stanford University Computer Systems Laboratory, April 199 1.

[12] Craig B. Stunkel and W. Kent Fuchs. TRAPEDS: Producing traces for multicomputers via execution driven simulation. *Performance Evaluation Review,* 17(1):70–78, May 1989.

[13] *Craig* B. Stunkel, Bob Janssens, and W. Kent Fuchs. Address tracing for parallel machines. *IEEE Computer,* 24(1):31–38, January 1991.