# TANGO: A MULTIPROCESSOR SIMULATION AND TRACING SYSTEM

Helen Davis
Stephen R. Goldschmidt
John Hennessy

Technical Report: CSL-TR-90-439

July 1990

This research has been supported by DARPA contract N00014-87-1-0828. Authors also acknowledge support in part by fellowships provided by the Hertz Foundation and Xerox.

# Tango: A Multiprocessor Simulation and Tracing System

Helen Davis, Stephen R. Goldschmidt and John Hennessy

**Technical Report: CSL-TR-90-439**

July 1990

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California **943054055**

## Abstract

Tango is a software simulation and tracing system used to obtain data for evaluating parallel programs and multiprocessor systems. The system provides a simulated multiprocessor environment by multiplexing application processes onto a single processor. Tango achieves high efficiency by running compiled user code, and by focusing on the information of greatest interest to multiprocessing studies. The system is being applied to a wide range of investigations, including algorithm studies and a variety of hardware evaluations.

**Kev Words and Phrases:** multiprocessors, simulation, trace generation, parallel programming, memory systems, performance evaluation

# Tango: A Multiprocessor Simulation and Tracing System

Helen Davis
Stephen R. Goldschmidt
John Hennessy
Computer Systems Laboratory
Stanford University, CA 94305

## Abstract

Tango is a software simulation and tracing system used to obtain data for evaluating parallel programs and multiprocessor systems. The system provides a simulated multiprocessor environment by multiplexing application processes onto a single processor. Tango achieves high efficiency by running compiled application code, and by focusing on the information of greatest interest to multiprocessing studies. The system is being applied to a wide range of investigations, from algorithm studies to detailed hardware evaluations.

## 1 Introduction

The potential advantages of multiprocessing have motivated studies in parallel algorithms and multiprocessor architectures. One area of particular interest to us is the availability of increased performance for individual applications. In our research, we need to do detailed evaluations of proposed hardware designs, and to understand the behavior of applications running on novel systems with more processors than are currently available to us. This paper describes Tango, a software simulation system developed to support these studies by providing useful data for evaluating parallel programs and multiprocessor systems.

Simulators, such as Tango, play an important role in studies of both hardware and software systems. Results from simulations can provide valuable information to guide the development and verification of analytical models. Systems too complex to model accurately in detail can often be simulated and resulting measurements used in accurate design evaluations. Simulation can also help identify system bottlenecks, determine resource requirements, or predict performance on hypothetical or unavailable hardware.

The understanding of parallel processing depends critically on two types of events which do not occur in serial programs: shared data communication and synchronization operations. These global operation are the primary concern in many multiprocessing studies. Many multiprocessing studies involve conventional processing elements (PE) and focus on network and memory system design, so computations local to a process are generally only relevant with respect to how they impact the timing of global operations. Allowing users to focus on global operations was key in making Tango simple and efficient.

In this paper, we discuss approaches to measuring multiprocessors and present the general methodology and design of the Tango system. First, we discuss critical issues in simulation and our goals in building Tango. Then, in Section 3, we consider basic techniques that have been used gather information about program execution and the approach taken in the design of Tango. After describing our computing environment, we present the implementation of Tango in Section 4. Tango has been used to support a variety of experiments and, in Section 5, we give a brief description of some of our experiences. Preliminary evaluations show Tango has a wide performance range, depending on the configuration and data gathered; we examine performance trade-offs in Section 6.

## 2 Issues and Goals

Simulation efficiency and accuracy are critical issues in simulator design and evaluation. Efficiency is important, since interesting applications tend to be long-running, and time and space limitations may force users to limit tracing to short sections of program execution. This is a problem for applications that consist of a number of distinct phases of computation, since limited tracing may only capture one phase of the computation and thus give a misleading impression of the overall program behavior. The problem of efficiency is even more acute in multiprocessor studies than in uniprocessor studies. This is because simulation costs are higher, since the interactions of processes must be modeled and serialization is required when simulating more processors than are available. Also, synchronization events are relatively infrequent but have a large impact on performance.

The accuracy of a simulator's output is also an important consideration to its users. Any simulation or tracing facility must assume a target machine model that defines the available operations and their timings. The accuracy of the information gathered depends on the accuracy and appropriateness of this machine model with respect to the systems and measurements of interest. Accurately tracing the execution of parallel programs is difficult because of the non-determinism of many parallel programs. The execution path through a parallel program, and sometimes results, may depend on the real time behavior of the hardware system. Small changes in the ordering and timing of events may also have an unpredictable impact on program execution time and process interactions. Tracing and simulation is therefore very system dependent, and data gathered for one multiprocessor may even represent an impossible execution ordering on another. Therefore, for parallel programs, accurate global ordering and timing of operations are required to ensure a correct simulation of a program's execution path, interactions between processes, and performance.

Simulators vary greatly in their generality and flexibility for supporting a variety of studies. This is because tradeoffs must be made between a system's efficiency, accuracy, and generality. If a system is too inefficient or too inaccurate, it will restrict the amount and the type of information that can be gathered. Simulating a multiprocessing system in more detail will generally increase accuracy, but will reduce efficiency and generality. Performance tradeoffs result in many systems being targeted for a specific set of studies. Some systems accurately support studies using a particular architecture, but are relatively inefficient, and cannot be used to accurately study program execution on alternative architectures. Other systems are more general and efficient, but have limited accuracy when studying the impact of implementation design decisions. Tango must efficiently support both software and hardware studies.

Our goal in Tango was to provide a general purpose simulation facility that yields required accuracy as efficiently as possible. We believe simulation costs should depend on the amount and accuracy of the information required, not the maximum accuracy and tracing available. As a result of this "pay only for what you need" philosophy, Tango provides an unusually flexible interface for program monitoring, tracing, and memory system simulation. The user may monitor (and optionally trace) shared data references, all data references, or just synchronization operations. The user can incorporate various memory models so that the most efficient memory simulation possible for a set of experiments may be used. The simulation costs for an application may vary by several orders of magnitude, depending on the information gathered and the complexity of the memory system simulation used.

## 3 Measurement Techniques

The measurements we desire relate specifically to parallel aspects of multiprocessing systems. From this perspective, we can view computation as consisting of global events separated by local computation. *Global* events are operations that can be observed by some other process, and are assumed to be shared memory and synchronization operations (or messages). *Local computations* are any operations that are private to a particular process, and are assumed to be deterministic 'and are generally unaffected by computation within other processes. We are concerned with the time spent doing local computation, but not in the details, such as the number of floating point operations or parallelism within processing elements. Characterizing execution in this manner will allow concise and efficient collection of the information pertinent to parallel processing.

In this Section, we consider basic techniques that have been used to gather information about program execution. The techniques presented are fairly general, but we will concentrate on their suitability to multiprocessing studies. We conclude the section by describing the basic approach and structure of Tango.

## 3.1 Monitoring

Monitoring is a technique in which observations of execution on an existing machine are made. It can be used to gather statistics summarizing execution behavior, or to produce a trace of events for later post-processing. Monitoring may be done using hardware or software techniques. Monitoring the machine with special hardware [6] is attractive because it can be done with little intrusion on program execution, and can therefore be very efficient and accurate. Simple counters, such as those provided by the CRAY-XMP, can be used to count events [5, 13, 31]. Alternatively, more complex hardware may provide special processing and storage of traces [22]. A related approach, which requires no special hardware, was taken by Agarwal et. al. [3]. He modified the microcode of the VAX to produce address traces. This technique is somewhat less efficient than attached hardware, slowing programs down by a factor of 10, but is simpler to implement and more flexible in the data that can be gathered. Hardware monitoring can provide data about the aggregate system behavior, and so is especially valuable in studies of multiprogramming, system software behavior, and operating system policies. Unfortunately, there are a limited set of signals that can be practically monitored, which makes hardware monitoring inflexible; it may be impossible to directly monitor the events of interest, or to relate low-level machine actions to the application events. Hardware monitoring requires detailed understanding of the hardware and access to the hardware, which makes the technique inappropriate for studying design proposals.

Software can also be used to monitor program execution on an actual machine. One option, previously used in uniprocessor studies, is to have a tracing facility single-step the program and monitor the execution of each assembly language instruction, much as debuggers do. Instructions can then be disassembled and interpereted to gather a wide variety of statistics, or a complete reference trace. Due to its relative implementation simplicity and flexibility, this idea has been extended to multiprocessing monitoring systems. The difficulty in extending this technique comes from the fact that there is no clear way to “single-step” a parallel program. Some systems execute one instruction per target PE, using a round robin scheme or some other simple algorithm [ 11, 17]. IBM PSIMUL creates a separate sequential tracing process for each target PE, which are then run using the normal system scheduler on a multiprocessor [28]. In general, multiplexing techniques such as these may distort the application behavior, since the sequencing of various global events will not necessarily match the ordering that would occur during an actual execution. Since parallel programs are often non-deterministic, it is difficult to estimate the error that results in these systems, or to know if the monitored execution path would ever occur on an actual machine. In addition, single-stepping a program is not a very efficient scheme and can slow programs down by factors of 100,000 or more, and all timing information is lost. Therefore, although this technique offers flexibility and a simple implementation, it is not efficient and assuring accuracy is difficult.

Another software technique, which is much more efficient than single stepping, is to modify software to be “self-instrumenting” and gather information in an execution-driven manner “on-the-fly” during its execution [ 15, 30]. These modifications may be explicitly made by the user (this is an option in most systems), inserted by a pre-processor or compiler system [4, 7, 12, 32], integrated into the language system [ 10, 15, 19, 27], included in library or operating system kernel code [9, 16, 23], or some combination [ 1, 22, 24, 26]. The information that can be gathered will depend on where instrumentation is added. The application can be instrumented in terms of the abstractions used by the programmer; this makes it easier to report results in terms the user will understand, and also easier to study the use of the abstractions provided. The instrumented software approach can lead to simple, flexible, and often efficient tracing system implementations. As will be described in Section 4.3, Tango uses this basic instrumentation method to monitor simulated program events.

The accuracy of monitoring systems depends on how they perturb system behavior. Several types of program distortions that result when using various monitoring techniques are discussed by Agarwal in [ 3]. If monitoring is very intrusive, then there will be large errors in any timing information and-the program's execution path may even be wrong. If the frequency and cost of instrumentation is low, the perturbation due to monitoring may be insignificant, and traces may include fairly accurate timing information for each event. However, while accurate monitoring is attractive for studying ex*isting* machines, extrapolating results to proposed dissimilar machines is problematic. In particular, in our multiprocessor studies, we would like to study systems with more processors than are currently available to us, and systems with novel network and memory systems. In these cases, it is particularly difficult to assure that results obtained from simple monitoring of an existing machine can accurately be used in studies of proposed architectures.

## 3.2 Simulation

High flexibility is provided by software systems that simulate the behavior of a target system. In this case, software running on an available machine is used to emulate the program execution on some target machine. This approach is very adaptable and information can be gathered on the behavior of abstract and unimplemented machines. It is especially useful when studying detailed aspects of proposed hardware, or behavior intrinsic to an application. Unfortunately, simulation is often complex and time consuming, requiring hundreds or thousands of times longer than than running an application directly [3, 11]. To make simulation more tractable, the entire target multiprocessing system is not simulated, and the components that are simulated are not always simulated in full detail. In particular, simulation of the execution of system code, multiprogramming, and I/O effects is complex and so is often omitted. When focusing on network and memory system design, it is also possible to avoid costly emulation of simple processor instructions (such as arithmetic instructions) by executing them directly on the available host machine. In the remainder of this section we discuss two techniques, *trace-driven* and *execution-driven* simulation, which both gain efficiency by directly executing application code.

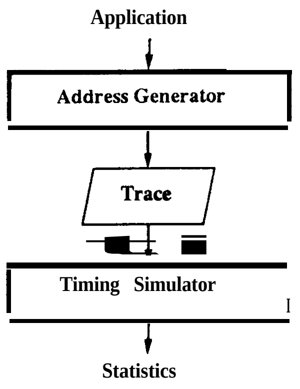

Figure 1: Basic structure of a trace-driven simulator.

### 3.2.1 Trace-Driven Simulation

The implementation of a conventional uniprocessor simulator is often divided into two components: an event generator and a post-processor. The event generator produces a stream of execution events that is input for a post-processor; the post-processor gathers behavioral statistics, and may also emulate the target system in enough detail to estimate the timing the program would have on the target machine. This partitioning allows the techniques used in event generation to be divorced from those used for post-processing and timing simulation; this can simplify the simulation system implementation, and allows the results of a single trace generation run to be re-used with different post-processors and simulation models. The trace generator may use a model to produce a synthetic trace, monitor a program's execution on an actual machine (as discussed in Section 3. 1), or interpret the program and emulate its execution on a target machine. The post-processor may gather simple event frequency counts, or include a timing simulator which may mimic the very detailed actions of the hardware system. When the post-processor is a simulator, this decomposition is often referred to as *trace-driven simulation.* For our studies, it is useful to think of the event generator as *an address trace generator,* which generates a trace of memory operations (including any synchronization operations), and the post-processor as a *timing simu-Zator* for the target network and memory system (see Figure 1).

Even when performance evaluation is the primary goal of simulation, the program execution path and the actions taken by a system must be known. Hence, sim-

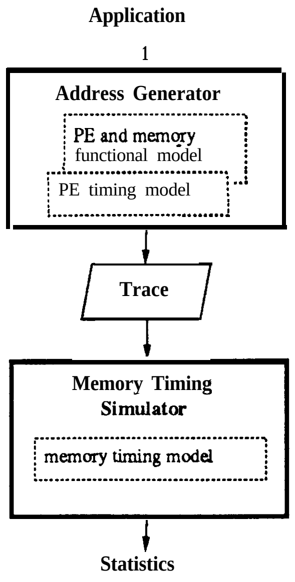

Figure 2: Simulation task decomposition common in (uniprocessor) trace-driven simulators.

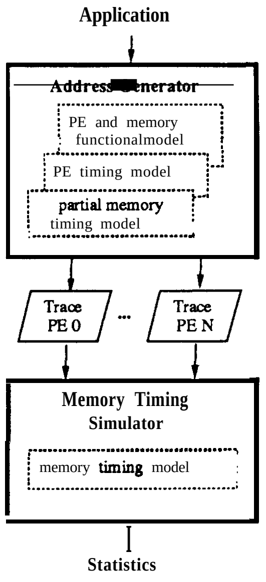

Figure 3: Common structure for a trace-driven multiprocessor simulator.

ulation requires simulating both *the functional execution* and *also the progression of time* of an application on some target system. Where these tasks are implemented varies in trace-driven simulators. When traces contain addresses, but no memory data or timing information, the functional execution is encapsulated in the address generator, and the timing simulation is implemented in the memory timing simulator. When the time between traced operations is difficult to reconstruct in the memory system timing simulator (for example, if a variable number of PE instructions is executed between traced events), issue times may be included in traces. *In this* case, *the PE timing model as well* as the *functional simulation* is incorporated into the address generator; the timing simulator contains a *memory timing model,* but no knowledge about the PE system (see Figure 2). Trace-driven simulation has worked well in uniprocessor studies because, in many cases, program execution is deterministic and so the functional execution can be correctly performed independent of memory system timing information, which can then be determined later.

Trace-driven simulators have also been used in multiprocessor studies [2, 11, 27]. In extending this approach to multiprocessing, several implementation issues arise. The address generator must generate event traces for multiple execution streams; this may be done by having each stream independently generate a trace [28], or by having some centralized manager control the progression of each stream [ 11, 17]. If the number of processors used by the simulator is less than the number of simulated processes, scheduling is required to multiplex execution streams. The event traces for each execution stream must be correlated with those from other streams; usually event time-stamps or a global trace file is used to guarantee that correlation is possible. The most difficult issue in trace-driven multiprocessor simulators is that of assuring accuracy. Many parallel programs are nondeterministic and so the order of traced events, and the execution path through the program, will depend on the latency of operations. For example, the process to next acquire a lock in a program depends on the relative execution rates of all competing processes. Therefore, some memory timing assumptions are embedded into the address generator. The result is a system in which detailed memory timing simulation is trace-driven, but the address generator does include a simple memory timing model. In general, after the functional simulation is done, the trace-driven simulator may adjust timing values, but cannot change the execution path through the program [1] (see Figure 3). Therefore, the accuracy of the later timing simulation will be limited by any inaccuracy in

[1]For some restricted programming models in which tasks do not interact, some task re-ordering may be possible [27].

the timing assumptions made by the address generator. So, for non-deterministic parallel programs, trace-driven simulators may yield incorrect execution traces and erroneous timing information. It seems likely that the overall behavioral characteristics of some programs will remain similar in spite of the errors, while in other programs the repercussions may be less predictable. More work needs to be done to determine how much error occurs, its impact on program behavior, and the predictability of the error.

### 3.2.2 Tango Execution-Driven Simulation

Tango may be used in a trace-driven simulator configuration, much as described in the previous section; however, Tango also allows the user to perform simulations in an *execution-driven* manner [8]. In execution-driven simulation, the execution of the application is interleaved with the simulation of the target system architecture. One possible Tango configuration partitions the simulation into two modules that share memory and interact during execution: an address generator that simulates PE operations, and a memory simulator that simulates memory operations. In the simplest case, the memory simulator is a subroutine that is evaluated on demand whenever memory operations are encountered in the execution stream by the address generator. Alternatively, the memory simulator may be a separate process with its own internal state. This minimizes the size of application processes, simplifies accessing shared data, and allows easier simulation of interactions between references. When a separate process is used, the address generator and the memory simulator interact in a co-routine fashion: when the address generator encounters a memory operation, it issues the operation to the memory simulator; the memory simulator simulates forward until some memory operation completes, or immediately before the next memory operation to be issued by the address generator. Notice that whenever Tango execution-driven simulation is used, the address generator makes *no* assumptions about the timing of memory operations – it always calls the memory simulator for accurate memory timing information.

In many studies, PEs are assumed to have a consistent view of memory, and so program variables have a single data value at any time. In studies such as these, the memory simulator can be simplified by storing and manipulating data values in the address generator. The

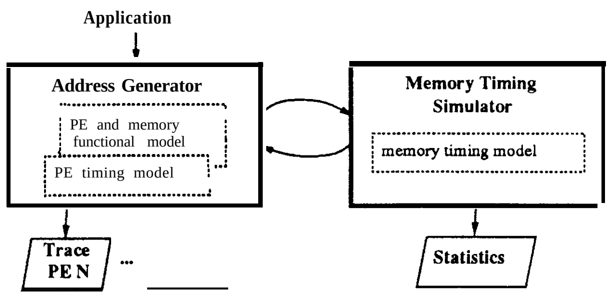

Figure 4: Basic structure of Tango, an execution-driven simulator.

address generator issues a request into the memory system *timing* simulator whenever a timing estimate is needed. The memory simulator responds with the operation's latency, or delays responding to that request until the appropriate simulated time has elapsed. This is a commonly used configuration in our studies, and is pictured in Figure 4.

A memory simulator used with Tango can be further simplified, and the system made more efficient, by reducing the number of operations issued to it. As discussed in Section 1, often only the global operations are of interest in multiprocessing studies. Some studies have shown that network traffic consists primarily of *shared* data accesses [11], so it may be that *private* data accesses do not significantly impact the network/memory system. It is therefore often adequate to simulate only the shared data accesses in the memory timing simulator, and to use a constant access delay for private data operations (which is easily incorporated into the address generator). Similarly, when studies are concerned primarily with synchronization, it may be possible to simulate those operations in detail and use simple compile-time estimates for all data references.

Unlike trace-driven simulation, execution-driven simulation permits accurate simulation and tracing of non-deterministic parallel programs. It also allows users the option of avoiding the time and space required to store large trace files. The added accuracy and flexibility of execution-driven simulation is necessary in systems, such as Tango, that must support a wide range of experiments. In the next section we discuss in more detail how this simulator structure was implemented in Tango.

## 4 Tango Implementation

This section describes our computing environment and the implementation of Tango., While the details of our environment are not important, we do assume that synchronization operations are explicit, and that shared data operations can be distinguished from private data operations at run-time. During the compilation process under Tango, the user application is automatically augmented with code to simulate time, manage the simulation, and optionally monitor events. Compile time options allow the user to specify a simulation system that meets the desired level of accuracy with the maximum efficiency.

### 4.1 Environment

Our applications are written in C or Fortran using macros developed at the Argonne National Laboratory (ANL) [20, 21] [2]. The macro package provides machine-independent abstractions for shared memory allocation, process creation and control, communication, and synchronization. The macro package is easily ported to new multiprocessors and runs on many systems, including the Alliant FX8, Encore Multimax, Sequent Balance, Intel iPSC, SGI 4D/240S, Denelcor HEP, Cray 2, and clusters of workstations on an Ethernet. In the UNIX shared-memory version of the ANL macros we use, each process may have access to globally shared memory as well as memory private to that process. The macros present a variety of abstractions that the programmer can use. For example, **Lock** acquires a binary lock, while **Unlock** releases it. **Barrier** holds processes at a particular synchronization point in the program until a specified number have reached that point, at which time they are all released. **Getsub** is used to implement dynamically distributed loops. It returns the next available loop index; when no loop indices remain, all processes wait until all loop iterations have been executed (i.e., there is an implied barrier at the end of a distributed loop). Currently our applications all use a shared memory programming model, although our system can be used to study applications written using a message passing model.

Our implementation is designed to run on a uniprocessor under an operating system that provides support for shared memory and semaphores. Currently Tango is running on the MIPS M/120 and the SGI 4D/240S under System V, and on the DECstation 3 100 under Ultrix. The number of application processes is limited by the number of processes and semaphores the system allows and by various table sizes. The limit is currently 256 in our systems.

[2]Other work using this set of macros includes a monitoring system developed by Lucier [19].

### 4.2 Implementation of Simulation

As discussed in Section 3.2.1, simulation requires both a functional simulation and also a timing simulation. In Tango, the functional simulation of a program is done by running an augmented version of the user's compiled application on an available machine. The simulation of time is done with software clocks, which are updated by code inserted into the user's application. This section discusses how these simulation tasks are realized in Tango.

#### 4.2.1 Functional Simulation

A process is created for each process in the application, and is assumed to be associated with a unique processor in the target multiprocessor system. [3] We simulate the functional behavior of a process by compiling the associated application code and executing it on an available machine. Novel target machine instructions, such as proposed synchronization primitives, that do not exist on the available machine are implemented in libraries and macro packages. Running compiled application code in this manner is much simplier and much more efficient than using a software interpreter which emulates every instruction. Since we are not interested in the very detailed behavior of the computation, our approach adequately mimics the behavior of the target system.

The compiled application processes are multiplexed to run on a uniprocessor in a way that preserves the ordering and timing of the events of interest with respect to the target system's simulated global time. The memory system simulator may be implemented as a subrou-

[3]Tango is being upgraded to allow simulations of multiprogrammed systems or applications running with more processes than processors, however, to simplify our discussion we omit these issues from this paper.

tine or as a separate process. Since a process executing between events of interest is assumed not to be affected by any other process, preserving the order of events can be done by rescheduling processes immediately before these events. At that time, the process farthest behind in simulation time is scheduled to execute next. Thus, no operation under study is issued until it is known to be the next event of interest. The multiplexing of processes is implemented by a distributed scheduler: each application process is augmented to reschedule itself before each event of interest. Using a distributed scheduler avoids extra context switches and overhead associated with repeatedly calling a centralized scheduler.

### 4.2.2 Timing Simulation

The progression of time is simulated as it would occur on the target multiprocessor, and the actual time to perform the simulation does not affect reported times. Simulating a global system clock would be complex and slow in Tango, so instead there is a software clock associated with each process that represents its simulated time.

During simulation, the individual process clocks do not run in lockstep, but instead are loosely synchronized so that the timing of the operations of interest can be preserved. Efficiency is therefore gained without compromising accuracy. There are three primary options for controlling the relative progression of time among application processes. The strictest, and most costly, option is to synchronize the process clocks at all data references and synchronization operations. This is only necessary for detailed simulations in which the complete data cache must be simulated in detail. Under a second option, the process clocks are kept within one global operation of each other, differing only by the duration of sequences of exclusively private computation. Since global operations include shared data and synchronization operations, this option is useful for studies that are not concerned with private data operations, but require precisely-ordered shared data references. For higher simulation efficiency, a third option assures only that clocks be within one synchronization event of each other. This option is most useful when studying synchronization. It can also be when using programming models in which processes only communicate through messages or explicit synchronization operations, or when trading-off some accuracy for increased efficiency.

A process's clock is updated in each program basic block, at simulated memory references, and at synchronization points. For the purpose of timing estimates, the target PE is assumed to have a basic instruction set architecture that can be approximated by the architecture of the machine used to run Tango simulations. Each assembly language instruction of the *available* machine is assigned a default delay corresponding to the latency of the target machine. During the compilation process, the application is augmented to update process's clock by these latency values. Since all of the instructions in a program basic block are executed whenever any one is, Tango incorporates all of the increments within a basic block into a single increment. Using this approach, whenever the delay for an operation can be determined at compile time, the progression of time is simulated very efficiently by simply incrementing a counter. Sequences of computation local to a PE, for example, have such predictable delays. When the delay is not known at compile time, the clock update is done by calling routines to perform the necessary calculation or simulation. In particular, shared memory latencies may be known at compile time when a simple memory model is used, but in other studies latencies will have to be calculated at run-time by a simulator. When doing algorithm studies, such as determining program resource requirements, a simple shared memory model is more architecturally independent and so may be the most appropriate model. In other studies, such as comparing distributed memory systems, it may be necessary to model the impact of network contention or non-uniform memory access times. In this case, Tango can call routines that interface with a memory system simulator to accurately emulate the target memory system. The latency of synchronization operations can be calculated similarly. Synchronization wait times, during which a process is delayed due to contention or a dependency on another process, are not known at compile time, and so are determined at run-time. When a synchronization operation results in unblocking other processes, the process doing the operation updates the simulation time of the released processes.

## 4.3 Monitoring Simulation Events

Tango can be used to gather a wide variety of useful information about the execution of an application. Integrated into Tango is a simple facility to log interesting events in trace files. A compile time option also allows the use of a monitoring interface, very similar to the memory system simulation interface, which allows users to incorporate various monitoring tools or to call their own custom monitor as the program executes.

Simple logging is performed by code added to the ANL macros and the application's assembly code. Synchronization events, such as barriers or lock operations, are implemented in the macros and so are easily logged by modifying the macros to call a trace routine. Data references cannot always be inferred statically from the source code, and so the assembly language of an application is augmented to optionally log them. Thus, the data reference stream produced is generated by the application running on *the available* system, and it is assumed that the target system has a similar instruction set architecture.

The user can statically or dynamically turn tracing on and off for particular sections of code by using provided macros in the source code. When tracing is turned off statically, the specified code will not be augmented with tracing code; no tracing overhead is incurred for these sections. In other sections, code is inserted to conditionally produce a trace, depending on the runtime tracing status.

When an application is run, the simulation will produce summary output files and, optionally, a trace file for each process. A *synchronization summary file* contains the name and type of each synchronization variable used by the application and is useful for post-processing routines. The simulation also writes a *process summary file* for each process, which contains the starting and ending time for the process and memory access counts. When tracing is enabled, the simulation produces an *event trace file* for each process. Each event in the trace is a 12 byte binary record and includes: the issuing process, the originating source code line, the a memory reference or synchronization event type, the address of the data accessed, and a time-stamp.

Figure 5 shows how general monitoring is incorporated into the structure of the Tango system. A monitor

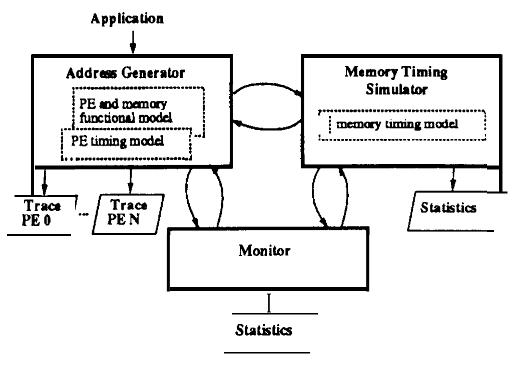

Figure 5: Structure of the Tango simulator with an integrated monitoring facility.

routine is called at interesting program events. As with tracing, user options specify interesting events to be all data references, shared data references, or synchronization events and user events. The monitor may interface to the memory system simulator to allow data specific to a particular memory hardware system to be gathered and displayed. Simple monitors may consist of functions which process trace information during simulation to avoid having to store the raw trace data. More complex monitors may run concurrently with the simulation system and provide information to user interface tools, such as run-time performance display tools. A monitoring system is currently being developed that will incorporate a general interface for incorporating architecture-specific information and also display data gathered from trace files or during simulation.

## 4.4 Application Process States

The behavior of a process during a Tango simulation depends on what simulation options the user has chosen. To illustrate how a Tango simulation proceeds, we will consider the actions taken by a process when Tango is configured to run with an incorporated monitor and a memory timing simulator (Figure 6). A process starts in the **Ready** state, waiting to be released by some process doing a reschedule operation. When the process is released, it is the process farthest back in simulated time. The process then executes (augmented) application code in **the Computation Simulation** state until it reaches either a monitoring event, an inserted reschedule operation, or a operation that needs to be simulated by the memory system simulator. At a monitoring event, the process enters **the Monitoring** state

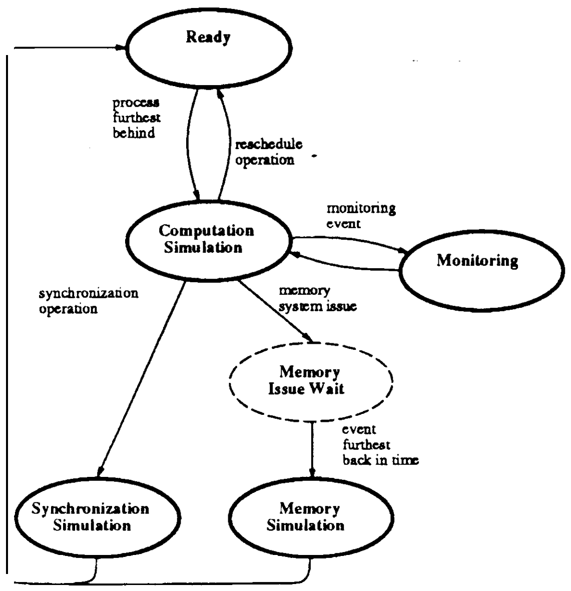

Figure 6: Tango application process state diagram.

by calling a monitoring routine; when the routine finishes, the process returns to the **Computation Simulation** state and continues executing the application. The reschedule operation is used to ensure correct ordering among processes. At a reschedule operation, a process transitions into the **Ready** state by placing itself on the ready queue and releasing the process farthest behind in simulated time. When a memory reference that needs to be simulated at run-time is reached, Tango will call the memory system simulator. If Tango is configured to incorporate a memory simulator implemented in a separate process, the process first enters the **Memory Issue Wait** state and remains blocked until its operation can be issued in order with any other concurrent operations. After memory simulation, the process again enters **the Ready** state. If instead the memory simulator is a simple subroutine, **the Memory Issue Wait** state is not required. In this case, when a process in the **Computation Simulation** state reaches a memory reference, it goes directly into the **Memory Simulation** state, which consists of evaluating the memory timing function.

High level synchronization operations are simulated differently in various architectural models. Libraries for synchronization abstractions can be written using target machine primitives. In this case, the libraries are executed as application code, and the synchronization primitives are issued into memory system simulator like memory references. Alternatively, synchronization abstractions can be simulated by a modified version of the ANL macros. If a memory system simulator is being used, the process issues a pseudo operation into the memory simulator whenever a process first reaches a synchronization abstraction. This ensures that the synchronization abstraction is executed in the correct order with respect to simulated memory operations. If a memory simulator is not used, reschedule operations inserted into the macros maintain correct ordering among synchronization operations. After a synchronization operation is completed, the process is put into **the Ready** state.

## 4.5 Compilation into a Tango Simulator

The Tango system is easy to use because it does not require the user to modify application code, and because default options exist for users not interested in customizing their simulation system to a particular hardware architecture. The application source code is only modified if the user wants to selectively disable tracing or simulation. During the compilation process, the application is integrated into a simulation system by automatically augmenting the application code and by using special libraries and a modified macro package. The compilation process consists of five steps: macro expansion, compilation into assembly language, augmentation, assembly, and linkage. A shell script, **tango,** is commonly called from a UNIX makefile to simplify the process of compiling a simulation. Given a list of file names, the script will perform all of the steps necessary to produce an executable simulation. Script parameters are used to select simulation and monitoring options, and also to specify the target machine model name (the script will infer which macro files and libraries to use from the model name).

Figure 7 shows the production of a simulator system using Tango. Steps required for normal compilation are drawn in solid lines, while steps and files added when using Tango are drawn with dashed lines. The first step is to expand the macros which provide machine and synchronization abstractions. An extended macro library is used; it implements synchronization primitives in terms of the target machine primitives, calls the memory simulator at synchronization events, provides user macros to turn on and off simulation and

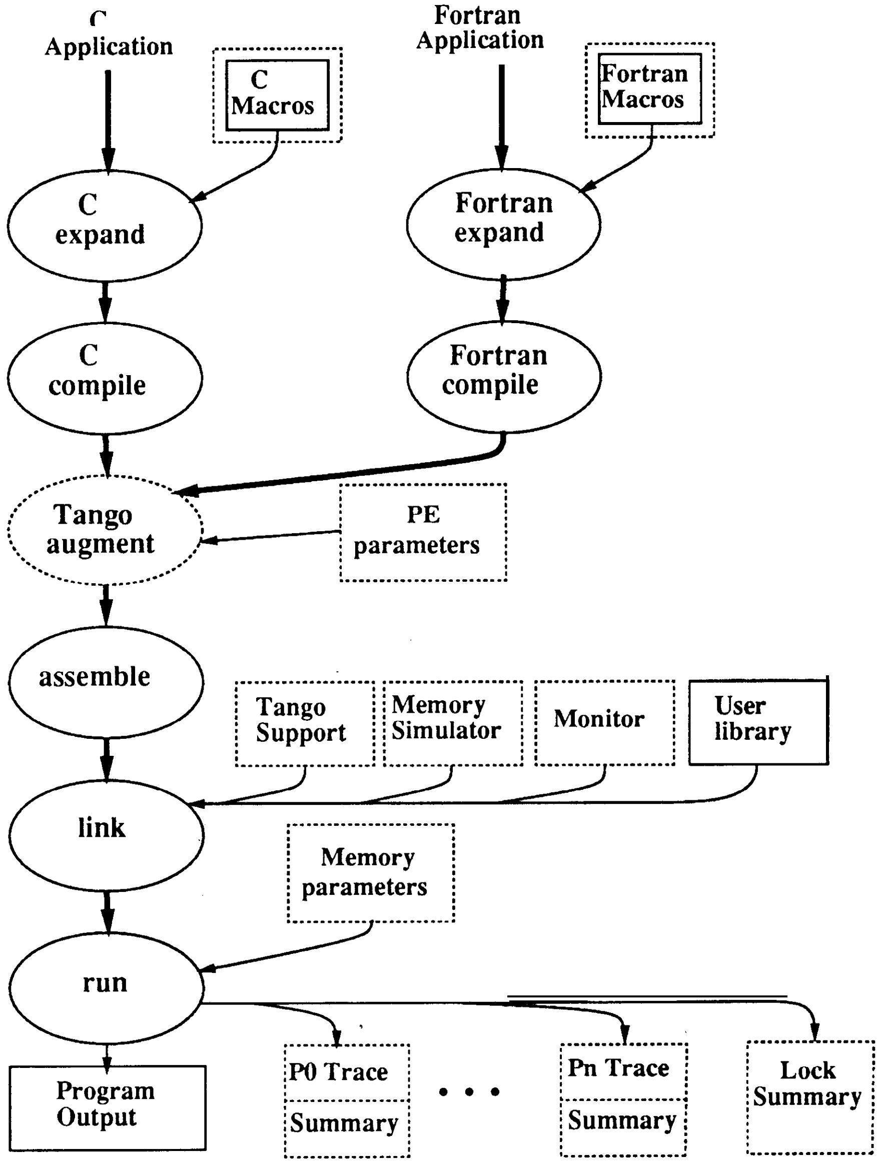

Figure 7: Production of a simulation system using Tango.

tracing, and declares routines and structures used in Tango. The output of the macro expansion is a standard language (C or Fortran) source file, and a standard compiler is used to produce an assembly language file from this source. An added augmentation step inserts code for incrementing the virtual time for each process, calling the memory simulator at data references, and writing trace records. A standard assembler is used to convert the augmented code into an object module. The final step is to link the object modules of the modified program with the memory simulator and Tango support routines.

In addition to the Tango macro library and run-time libraries, two parameter files are used as input to the simulator. The *CPU Parameters* file is read during the augmentation step, and contains timing values for each operation provided by the host machine used to run simulations. The *Memory Parameters* file is read during simulation, and contains synchronization delay and memory system parameters. One may use default files or substitute customized parameter tiles.

## 5 Using Tango

Tango will be used to support a variety of studies, including research in characterizing workloads, evaluating hardware, and program evaluation. To demonstrate the flexibility and use of Tango, this section briefly describes a few research efforts that are currently using Tango.

### Application Studies

Anticipating the performance of applications on unrealized machines is often too difficult to do analytically. Tango provides a simulated multiprocessor environment and gives the user much control and flexibility in specifying the simulated machine model used. Users can study the behavior of applications running on a variety of proposed machine models, or may select a very idealized machine model to identify performance bottlenecks inherent in an algorithm. Figure 8 shows the speedup curve for Pthor, a parallel logic simulator [29]. Prior to the availability of Tango, the user ran experiments on an available multiprocessor and was limited to measurements involving fewer than 16 processors.

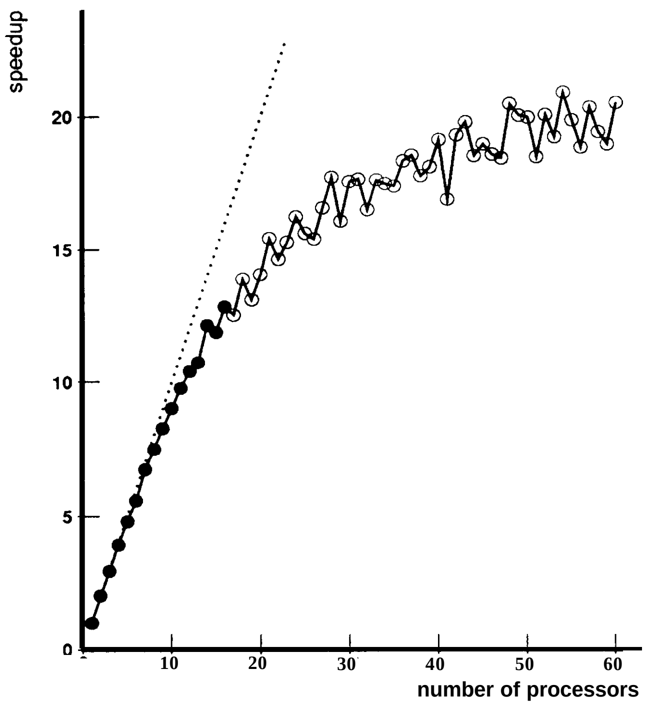

Figure 8: Observed speed-up for Pthor application.

With Tango, the user could see program scalability limitations that were not evident with lower numbers of processors.

### Synchronization and Concurrency

Delays associated with synchronization can substantially increase the running time of a parallel program. Tango has been used to identify program bottlenecks due to synchronization and to determine program synchronization resource requirements. Tango is also being used to study the use of synchronization and concurrency abstractions and to evaluate hardware support for them.

If a program includes frequent mutually exclusive accesses to shared data, its speedup can be limited. This was the case in a global wire router developed by Rose [25]. Figure 9 is a plot of the average time spent waiting to acquire locks as a function of the number of processors used to solve the problem. *The wire_lock* is used to distribute wires to route, and is an example of a well behaved lock. The time required to access the lock is small and relatively independent of the number of processors. The *routelock* is a shared data lock and an example of a performance critical lock: the wait time increases dramatically with increasing numbers of processors because of contention. Since the lock is accessed frequently, this limits the achievable speedup.

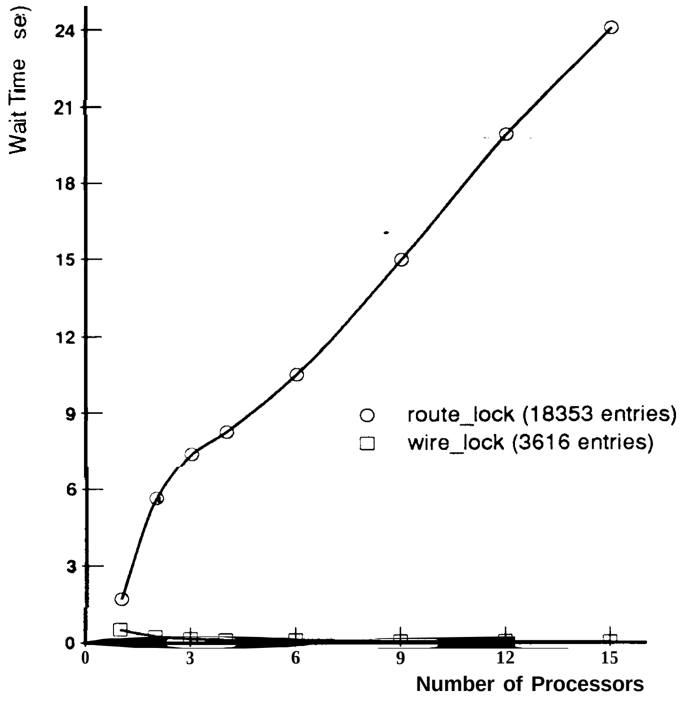

Figure 9: Wait time at locks in wire-router application.

In a second version of the program, the locking was eliminated, and the speed-up increased from less than 6 to more than 9 with 15 processors. As can be seen in this experiment, Tango trace data can be used to easily point out performance critical locks. Then the programmer can focus on program modifications that will increase speedup.

### Architectural Evaluation

In addition to evaluating hardware support for synchronization, Tango trace data is being be used in the evaluation of multiple CPU contexts, networks, and memory systems.

The time required for a process context switch can be greatly reduced by storing the state of a few different processes in the CPU. If process switches require only a few machine cycles, the impact of large memory latencies might be diminished. Preliminary work has been done to measure the impact on both processor utilization and application performance of context switching when the processor might otherwise be idle waiting for remote data.

A current focus of research is distributed shared-memory organizations and scalable cache consistency mechanisms. The DASH architecture [ 18], which is being developed at Stanford University, consists of small clusters of processors which are connected by a fast interconnection network. A cluster consists of a portion of the main memory and several processors-cache modules on a shared bus. Measurements of data traffic and delays resulting from sharing and nonlocal references are being used in evaluating alternative networks and cache consistency protocols [ 14].

### Shared Memory Behavior

The shared memory access behavior of a program can dramatically affect its performance. Low locality or high sharing can result in a long memory access latencies and high network contention. Research considers the impact of data locality, sharing, false sharing, placement, partitioning, and replication on performance.

One approach being studied to improve performance on non-uniform memory access machines involves scheduling each task to a processor which has efficient access to the task's data. New macros are being developed to dynamically assign tasks to processors and data to memories in order to reduce the frequency of accesses to non-local data. Processors give priority to tasks whose data is in nearby storage. In one application, changing the scheduling strategy has already resulted in a 20% improvement in execution time.

## 6 Tango Performance

This section investigates the performance of the Tango system and sources of simulation overhead. Although we have not yet tuned Tango to optimize its efficiency, the level of performance currently achieved by the system demonstrates the viability of the approach. As we shall see, simulation times depend primarily on the complexity of the memory system simulation, the type of events simulated at runtime, and the application's synchronization and data reference characteristics.

Measurements reported in this paper were taken on a DECstation 3100 workstation running Ultrix 2.0. The Ultrix `rus age ()` facility was used to measure execution times.

### 6.1 Applications

The complete execution of three programs were sim-

Table 1: Application characteristics.

| Application | Source lines | Uniprocessor run-time (ms) | Instructions executed (x $10^6$) | Operations as a % of instructions executed: Synch operations | Shared data references | Private data references | Total |
|---|---|---|---|---|---|---|---|
| Mincut | 462 | 640. | 47.8 | 1.28 | 9. | 25.5 | 35.7 |
| Mp3d | 1530 | 2410. | 10.1 | 0.00071 | 20.6 | 3.3 | 23.9 |
| Pthor | 8000 | 272. | 0.397 | 0.83 | 16.9 | 26.2 | 43.8 |

ulated to measure simulation performance of Tango. Table 1 shows the basic characteristics of the programs.

Pthor is a compiled logic-level circuit simulator [29]. It uses the Chandy-Misra simulation algorithm to avoid reliance on a single global time, and uses a separate task queue for each process. Pthor is a moderate-sized application; the source line count in the table does not include the circuit model. The test problem is a small circuit with 56 gates and latches, which explains the small number of instructions executed and the short uniprocessor run-time.

Mincut is a graph bisection algorithm using simulated annealing. The algorithm repeatedly selects a random node and decides whether to move it across the partition based on chance, the move's impact on the quality of the partition, and the simulated temperature. Processes perform moves asynchronously, locking only the relevant data structures, but perform barrier synchronization before and after each reduction in the temperature. The quality of the resulting partition depends somewhat on the number of processes, as does the number of accepted moves. The test problem is a 500-node graph with an average degree of eleven.

Mp3d is a three-dimensional rarified flow model using the Stanford particle method. The algorithm simulates individual particles moving and colliding in three-dimensional space. The majority of the parallel execution time is spent moving particles, which are statically assigned to processors. The only significant synchronization is due to the barriers between program phases. For these experiments, Mp3d is run with 10,000 particles in an empty 288-cell space array for five time steps.

The number of processes simulated is generally limited to thirty in the data presented here, although we present some data for 90 processes. All three applications can be simulated with more processes. Mp3d, for instance, has been simulated with up to 127 processors.

## 6.2 Memory Simulators

As described in section 3.2.2, the memory system simulator incorporated into Tango may be implemented as either a subroutine or as a separate process. In our measurements, we compare the minimum overhead associated with these two interfaces. To put the Tango overhead into perspective, we also evaluate the cost of simulating with a subroutine-style simulator that is being used to study program behavior on a proposed architecture.

The *Multiple-Issue* memory simulator uses the most general, and expensive, simulator interface. The simulation code runs as a separate UNIX process, and the application processes block each time the simulator is invoked. Simulated operations are issued in the correct global-order, and concurrent operations are issued to the memory simulator together. The interface used is intended to be used for detailed simulations that require accurate simulation of the interactions between multiple outstanding references. However, we use a very simple model of constant latencies for references to isolate the overhead associated with Tango from that of the memory simulator.

A second memory simulator, the *Single-hue* simulator, is used to determine the overhead associated with the subroutine-style memory simulator interface. This memory simulator is intended to demonstrate the Tango overhead that might occur in studies that do not consider the affects of contention very accurately. For this reason, the global order of synchronization operations is maintained, but the global order of memory data operations is not. This means that processes are rescheduled only for synchronization operations. Like the Multiple-Issue simulator, the Single-Issue simulator implements constant latencies.

The third memory simulator, called *Hyphen*, [4] uses the same simple Tango interface as the Single-Issue simulator, but has a more realistic memory timing model. The simulator models a grid of processing nodes in which each node is a bus-based multiprocessor. This model simulates two levels of caching at each processor and accounts for bus contention in an inexact fashion, requiring correct global ordering of all simulated operations. As a result, processes are rescheduled at both synchronization and data references. The Hyphen simulator is not intended for very accurate hardware design evaluations, but it is being used for detailed studies of program behavior. Results from this model should indicate the costs associated with a reasonably accurate, but low cost, memory system simulator.

In all three models, synchronization is simulated by a general default mechanism. The operations are simulated by ANL macros (see Section 4. 1), but the memory simulator is still called at significant events, so that the memory simulator may provide any desired additional latency. For locks, the request, acquire, and release phases are each issued to the memory simulator as separate events. When a process must block due to contention at a lock, additional block and release events are issued to the memory simulator. For barriers, the entry and exit of each process is treated as a separate event, so the number of events grows linearly with the number of processes. Because barriers are infrequent in the applications we are studying, the increase in simulated events due to barriers is small.

## 6.3 Sources of Overhead

To run under Tango, the applications have been modified to multiplex processes, simulate time, and optionally log events. Each of these activities adds to the cost of simulation.

Multiplexing application processes onto a uniprocessor requires scheduling at each event for which correct global ordering must be guaranteed. The cost of scheduling processes is dominated by the cost of any required context switches. The Tango scheduler uses system semaphores and full-weight UNIX processes. As a result, a context switch requires an estimated 180-250 microseconds. When large numbers of processes are being simulated, almost every call to the scheduler results in a context switch.

Each process updates its virtual clock at the beginning of each basic block to account for delays that are known at compile time. This typically requires five assembly language instructions, so if each basic block has five to ten instructions, these updates should produce a slowdown factor of 1.5 to 2. Depending on the simulation options and the particular application, 40% to 99% of the application code can be simulated in this way. Thus, most of the timing simulation is performed very efficiently.

The remaining application code consists of synchronization or memory operations that must be simulated at run-time. Invoking the simulator requires about 70 instructions when a subroutine-style simulator is used, but a context switch is required when the simulator runs on a separate process. If the simulator is only called at shared data references, there is an overhead of about a dozen instructions per memory reference to distinguish shared data references from private data references. In addition to this intrinsic Tango overhead, there is the cost of the memory timing simulation itself, which varies considerably, depending on its complexity.

Because Tango supports efficient execution-driven simulation, trace files are rarely needed. When they are, Tango produces separate trace logs for each process, so global event ordering is not required for tracing. The cost of writing events to disk dominates the cost of generating a trace log. Using buffered UNIX writes to a local disk, each 12-byte record requires about 31 microseconds to write.

Augmenting the program with assembly language instructions and linking in Tango libraries increases the size of programs, as shown in Table 2. The increase in size depends on the memory model used, the static frequency of instrumented operations, and the average basic block size. The increase is typically a factor of 2 to 3, and in no case did it exceed 3.51 for the applications studied. Program size does not affect simulation results, but it will degrade simulation performance by increasing the amount of memory swapping. In addition, the number of processes that can be simulated is

[4] Hyphen simulates a simplified model of the DASH architecture described in [29].

Table 2: Program size increase under Tango.

| Relative object code size for programs | | | |
|---|---|---|---|
| Event type simulated | Memory model | Mp3d (84232 bytes) | Mincut (64256 bytes) |
| None (Tango not used) | None | 1 | 1 |
| Synchronization | Single-issue | 1.90 | 1.87 |
| | Multiple-issue | 2.01 | 2.01 |
| | Hyphen | 2.10 | 2.14 |
| Global | Single-issue | 2.60 | 1.94 |
| | Multiple-issue | 2.71 | 2.08 |
| | Hyphen | 2.80 | 2.21 |
| Global and private refs | Single-issue | 3.25 | 2.14 |
| | Multiple-issue | 3.36 | 2.28 |
| | Hyphen | 3.51 | 2.41 |

sometimes limited by the available swap space.

## 6.4 Program Measurements

To evaluate Tango's performance, we compared the execution time of the three sample applications run directly on a uniprocessor (without Tango) to the time required to run the same applications for simulation using various Tango options. Each application was run from start to finish, including initialization code.

### 6.4.1 Simulation Overhead per Operation

Most of the overhead in Tango is associated with simulating events that must be simulated at run-time by invoking the memory simulator. To understand the nature of this overhead, we measured the amount of overhead per simulated event. Figures 10, 12, and 13 show the average overhead per event when simulating only global events, that is, shared data and synchronization operations. In the graphs, solid curves represent the observed Tango costs, computed by subtracting the uniprocessor runtimes from the Tango runtimes and dividing by the number of events simulated. Since much of the overhead is due to the high cost of process context switches, we made an attempt to compensate for this by subtracting 180 microseconds per Tango-induced context switch; these adjusted overheads are shown by the dashed curves.

The most important feature to note in these three graphs is that when more than a few processors are simulated, the simulation overhead does not increase significantly as more processors are simulated. Generally, the growth is cost is less than 10% as the number of processors is increased from 10 to 30. When simulating the Mincut program with the Single-Issue memory model, however, the increase is about 28% when the number of processors is increased from 10 to 30. As we describe later, the number of synchronization operations increases in this program, which increases the average overhead. The general leveling-off of simulation costs indicates that simulation costs become relatively independent of the number of processors simulated, and so we can use Tango to efficiently simulate many processors.

With the Single-Issue simulator, shown in Figure 10, the three applications exhibit similar overheads when simulating a single process. However, as the number of processes simulated increases, Mp3d shows no increase in cost, while the overheads for the other applications grow. We attribute these increases to two factors: first, increased likelihood that a process will block at each rescheduling point, and second, changes in the synchronization behavior of the applications which also increase the frequency of blocking.

When the Single-Issue simulator is used, memory operations have a small fixed cost, while synchronization operations require rescheduling and are more ex-

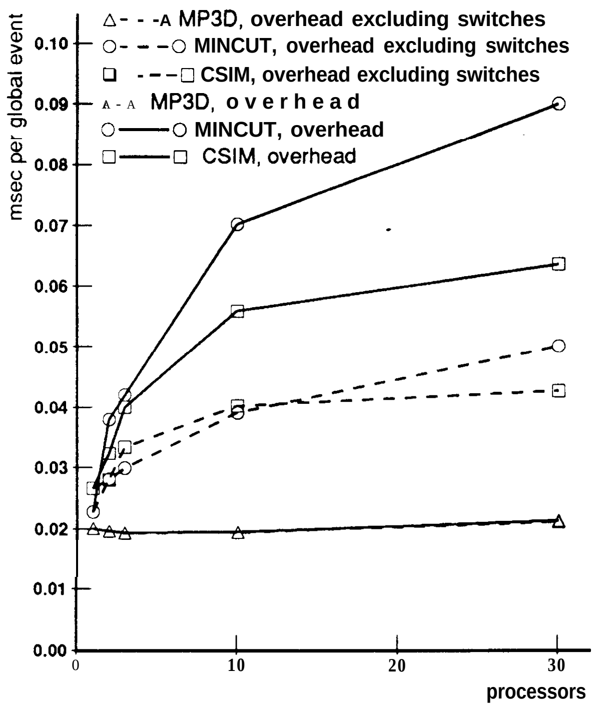

Figure 10: Simulation overhead per event for *Single*-Issue simulator at global events.

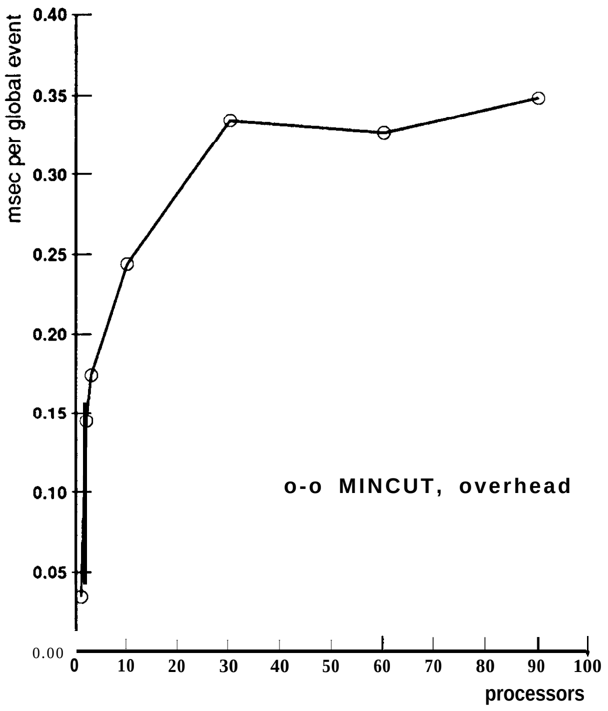

Figure 11: Simulation overhead per event for *Single-Issue* simulator at synchronization events.

pensive to simulate. It is instructive, therefore, to look at the cost of simulating synchronization events *only*. Figure 11 plots the cost per simulation event for Mincut when the simulator is called only at synchronization events. The simulation cost can be seen to initially rise very quickly and then level off. This is partly due to an increase in the cost of rescheduling to preserve the order of events, and partly because the cost of simulating locks increases when there is contention. Rescheduling at synchronization points requires a context switch whenever the current process is ahead of another. Thus, as the number of processors simulated increases, there is an increase in the likelihood that a context switch will be required. This results in an initial rise in the number of context switches. Eventually there is a context switch for each operation, and the overhead reaches a maximum.

More frequent contention for locks also contributes to the increasing average cost per synchronization event. In the system used for our measurements, synchronization primitives are implemented in macros rather than in the memory system. Using this approach, a context switch is required whenever a process blocks due to contention for a lock or synchronization variable. This means that if contention rises in the program as more processes are used, there will be additional Tango overhead due the increasing number of context switches. This accounts for much of the increase in overhead for Mincut, since it contains a lock that quickly becomes a point of high contention. Pthor also contains a lock which has some contention, however the lock has less contention and is accessed less frequently than the lock in Mincut.

In Mincut, the average cost of global events also increases because the number of synchronization events increases by a factor of two due to non-determinism in its algorithm (from 0.6 million operations with one processor to almost 1.2 million operations with 10 processors). The number of synchronization operations also increases in Mp3d, since the number of simulation events required for barriers increases linearly with the number of processes. However, the fraction of synchronization operations is still so low that simulation performance is not significantly affected (even with ninety processes, it is less than 0.3%).

In Mp3d, the memory events outnumber synchronization events by roughly three orders of magnitude, so the average cost per global event is not significantly affected by the cost required by synchronization events. In Mincut and Pthor, on the other hand, synchronization events are relatively frequent and so these programs are more sensitive to any rise in the frequency or cost of simulating synchronization operations.

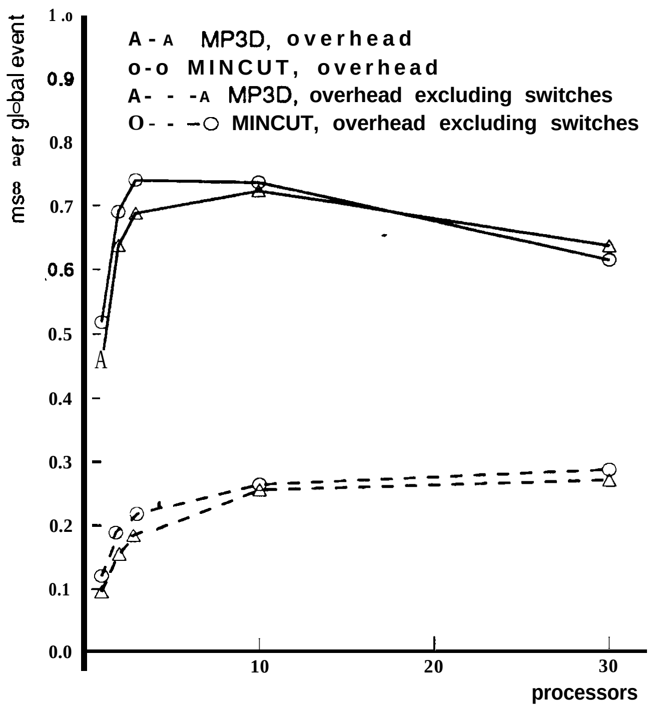

Figure 12: Simulation overhead per event for *Multiple-*Issue simulator at global events.

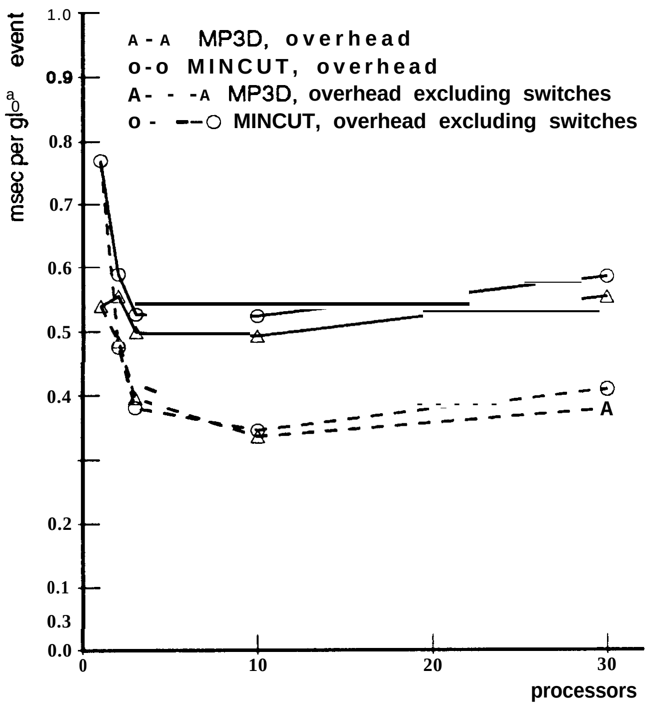

Figure 13: Simulation overhead per event for *Hyphen* simulator at global events.

When the simulator is implemented as a separate process, as with the Multiple-Issue model (Figure 12), two context switches are required each time the simulator is invoked, one to enter the simulator and one to exit. Rescheduling may also result in context switches, and occurs at both synchronization events and simulated data references. Rescheduling causes an initial rise in overhead that is even greater than that seen with the Single-Issue model, since rescheduling occurs more frequently. However the total observed overhead soon begins to *decrease* as more processors are simulated. This is because, with the Multiple-Issue model, concurrent memory operations are issued to the memory simulator in a single invocation; as the number of processors is increased, more operations are issued in parallel and so there are fewer memory system simulator invocations. If the cost of process context switches is omitted, the computational overhead is nearly constant. The total overhead is 7-30 times higher than seen with the Single-Issue model. In spite of this larger overhead, we found this interface useful for all three of our detailed hardware simulators, since it allows complex models to be more easily incorporated into Tango. For each of these hardware simulators, simulation costs are overwhelmingly dominated by the memory system simulator, so the overheads intrinsic to Tango are less important. The larger overhead for the Multiple-Issue model makes it evident that, although this interface is convenient for hardware design evaluations, it is worthwhile for Tango to provide the low-cost interface for other studies.

Figure 13 shows the overhead per event for the Hyphen model. The Hyphen model, unlike the single-event model, must reschedule on every shared data event, so it suffers significant overhead due to context switches with both Mincut and Mp3d. Even allowing for context switches, the Hyphen model runs about 10 times slower than the Single-Issue model, despite their using the same interface. This confirms that Tango overheads are small relative to the costs of even a simplified simulator for a complex memory system.

Note the differences in the shape of the cost curve for Hyphen and the other memory simulators. In Hyphen, simulation overhead first decreases as processors are added and then begins to increase. This is due to the cost of simulating bus contention in the Hyphen model. Contention simulation is done on a cycle-by-cycle basis, so the cost of simulating contention for an event is proportional to the number of cycles since the last event. For a single Mincut process, there are, on the an average, 10.7 simulated events per 1000 cycles. As the number of processes increases, the number of cycles between events initially decreases, since the programs speed up. This cost reduction bottoms

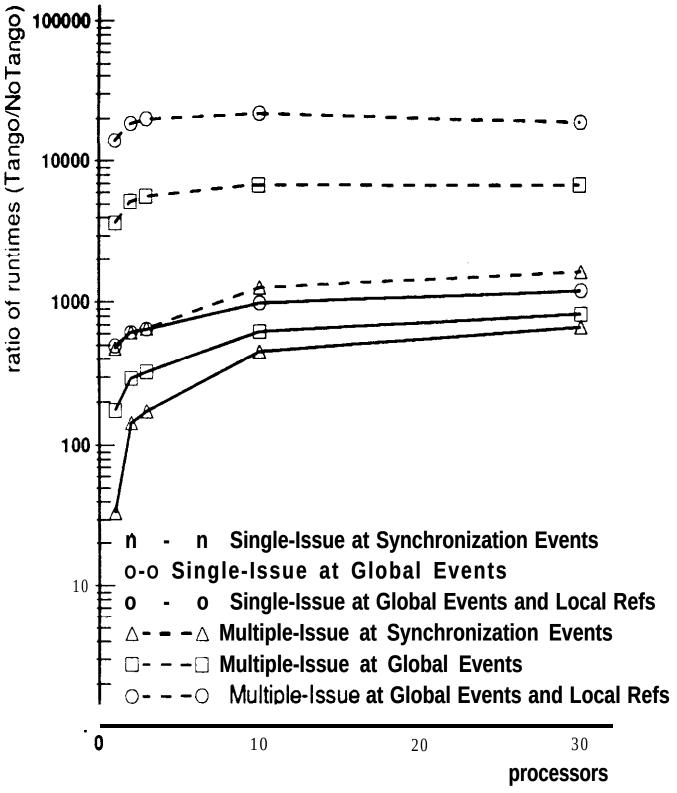

Figure 14: Relative simulation times for Mincut.

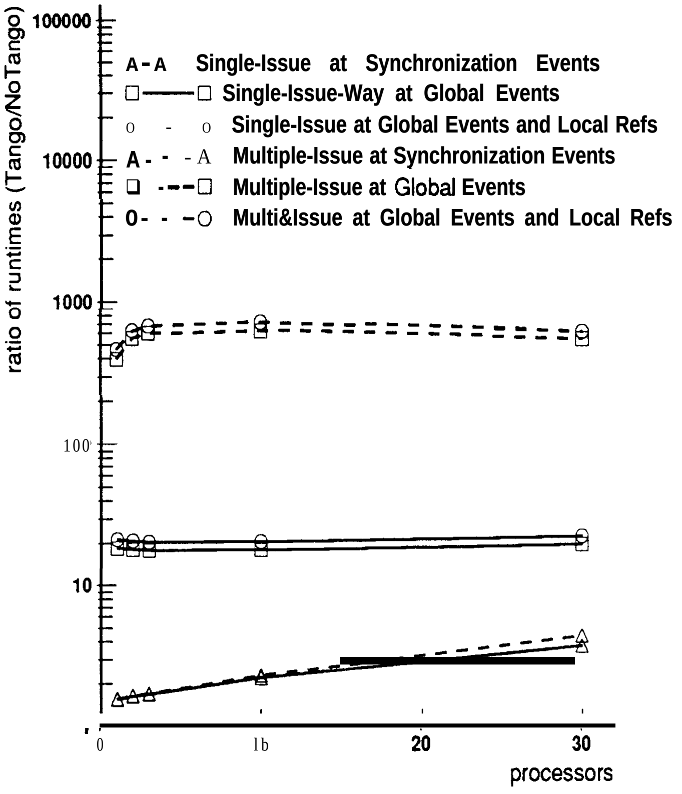

Figure 15: Relative simulation times for Mp3d.

out at about three processes, where serial initialization code and contention delays cause the average interval between events to level off. With larger numbers of processors, the extra work required for each event begins to dominate the savings from reduced cycles per call. The shape of the overhead curve is similar to that observed for our hardware simulators which simulate on a more detailed, cycle-by-cycle fashion.

### 6.4.2 Program Slow-down

In the previous section we considered the simulation overhead per event. The *frequency* of simulated events also strongly affects the elapsed time for a simulation. Figure 14 shows the observed slow-down as the number of simulated processors is varied for Mincut, and Figure 15 shows the slow-down for Mp3d. To make the graphs more comparable, the timing values have been normalized relative to the uniprocessor times for each application without Tango.

In looking at Figure 14, we see that the slow-down when simulating Mincut with 30 processors has a wide range, from less than 700 to over 18,000. As we would expect from the previous section, the simulation cost is always larger when using the Multiple-Issue memory model. For each memory model, the cost is shown for simulating (1) at only synchronization operations, (2) at global events (shared data and synchronization operations), and (3) at local references as well as global events. In many multiprocessor studies, local references are of little interest, since performance is determined primarily by the global operations. For Mincut, 74% of the data references are to private data, so omitting these events cuts costs by 3 1%-64%, depending on the memory model used. Similarly, if the user focuses on synchronization behavior, savings of 44%-91% are achieved.

Much lower simulation costs were observed when simulating Mp3d, as shown in Figure 15. When simulating 30 processors, even the most expensive Mp3d simulation has less slowdown than the fastest Mincut simulation. This is primarily because Mincut performs synchronization much more frequently, which is expensive to simulate in the default configuration. The maximum simulation cost for Mincut is also higher because it has more frequent data references (35% of instruction reference data in Mincut, while less than 24% do in Mp3d). The lower frequency of data references in Mp3d results in fewer calls to the memory simulator and more operations being simulated more efficiently by just executing application code. As with Mincut, there is a large range of slowdowns observed for Mp3d, depending on the memory model used and the types of events simulated. Since there are very few synchronization operations, both memory models perform very well simulating synchronization operations only. In Mp3d, there is little performance benefit in omitting the simulation of private data references, since fewer than 4% of the instructions reference private data.

## 6.5 Possible Performance Enhancements

Since one-third to two-thirds of Tango's simulation overhead is due to the cost of UNIX context switches, its performance could be enhanced by porting it to a light-weight threads package. This would provide less-costly context switching, but would require porting not only Tango, but also our applications and programming macros which have been written using UNIX processes.

The number of context switches, and so the cost of simulation, could also be reduced by allowing the simulation of application processes to be more loosely coupled. For some studies, accurate results do not require that every reference be simulated in the correct order. In particular, in some programs, maintaining just the order of synchronization operations will ensure that the correct execution path through the program is simulated. In other cases, the path through the program may not be exactly correct, but the overall program behavior may be similar. The impact of imprecisely ordered memory issues on memory system timing will depend on its simplicity and how it models contention and cache coherency. More research is needed to decide when the order of simulation events may be relaxed, and how to measure any impact on simulation results.

In hardware studies, or studies of detailed memory behavior, the overhead spent in Tango is small compared to the time spent in the memory system simulator. Thus, significant time-savings can be achieved by choosing the simpliest memory system simulator adequate for a set of experiments. This leads to a need to better understand the performance and accuracy implications of simulating memory systems at various levels of abstraction.

If simulation accuracy can be compromised somewhat, it might be possible to substantially increase simulation performance. Unfortunately, when program timing errors are introduced, it is generally not possible to determine a tight-bound on the potential error in simulation results. More work needs to be done to determine how to measure the resulting loss of accuracy when trading-off accuracy for increased efficiency. It would be very interesting to explore the possibility of using run-time monitoring of system parameters to provide better indications of confidence in simulation results.

If Tango itself were modified to be a parallel program, the overhead due to serializing the parallel programs might be reduced. Tango was implemented with multiple processes, and it would not be difficult to have these processes run in parallel. Unfortunately, the costliest Tango runs are those with complex memory system simulators that require preserving the order of memory operations. Thus, a simple parallel Tango implementation would not gain much performance due to frequent ordering dependencies and serialization in the memory system simulator. An effective parallel Tango implementation involve considering at what level of detail the system needs to be modeled for various studies, parallel memory system simulation, and trading-off accurate event ordering for additional speed.

# 7 Conclusions

As we consider building larger multiprocessors, we need to do detailed evaluations of proposed hardware designs and to understand the behavior of applications running on more processors than are currently available to us. We have found the Tango multiprocessor simulation facility to be extremely useful in supporting a wide range of research projects. Tango imposes few restrictions on the programming model or target machine, which makes it a general purpose tool applicable for use in many studies. Since Tango monitors synchronization abstractions, it can report results in terms of the abstractions being used by the programmer, and can be easily used to study those abstractions. When simulating non-deterministic programs, trace-driven simulation techniques which isolate timing simulations from address generation will yield incorrect results. Tango offers users the option of using execution-driven simulation, which ensures accurate ordering of program events even for non-deterministic programs. We have found compiled simulation to be very efficient and worthwhile. By allowing users to focus on the events of most interest and to incorporate various timing simulators, Tango does not sacrifice efficiency or accuracy for generality, but instead offers users a system built on a "pay only for what you need" principle.

Obvious performance improvements can be gained by porting Tango to a light-weight threads environment. It would be interesting to see how much performance could be gained by a parallel implementation of Tango. Current experiments using Tango have focused on the behavior of systems executing a single application implemented using a few standard synchronization primitives. In the future, Tango will be extended by developing libraries to provide additional programming abstractions and more complete multiprocessor environments. Possible directions for extensions include additional support for: additional synchronization abstractions, process scheduling, data allocation strategies, and visualization tools.

## 8 Acknowledgments

Helpful discussions throughout this work with Mark Horowitz, Greg Nelson, Michael Powell, Anoop Gupta, and Richard Simoni are appreciated. Applications were developed and generously made available to us by Jonathan Rose, Larry Soule, and Jeff McDonald. We also appreciate comments on drafts of this paper made by Josep Torrellas. This work was supported in part by fellowships provided by the Hertz Foundation and Xerox, and in part by DARPA contract N00014-87-K-0828; this support is gratefully acknowledged.

## References

[1] W. Abu-Sufah and A. Y. Kwok. Performance prediction tools for cedar: A multiprocessor supercomputer. In ***Proceedings of the 12th Annual Int'l Symposium on Computer Architecture,*** pages 406–413. ACM-IEEE, 1985.

[2] A. Agarwal, Richard Simoni, John Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. ***In Proceedings of the 15th Annual Int'l Symposium on Computer Architecture.*** ACM-IEEE, Jun 1988.

[3] A. Agarwal, R. L. Sites, and M. Horowitz. ATUM: A new technique for capturing address traces using microcode. ***In Proceedings of the 13th Annual Int'l Symposium on Computer Architecture.*** ACM-IEEE, Jun 1986. Published as Vol. 14, No. 2, of Computer Architecture News.

[4] Anita Borg, R. E. Kessler, Georgia Lazana, and David W. Wall. Long address traces from RISC machines: Generation and analysis. Technical Report 89/14, DEC Western Research Laboratory, Sept 1989.

[5] T.A. Cargill and B.N. Locanthi. Cheap hardware support for software debugging and profiling. In ***Proceedings of the Second Int'l Conf. on Architectural Support for Programming Languages and Operating Systems.*** ACM-IEEE, Oct 1987.

[6] Robert J. Carpenter. Performance measurement instrumentation for multiprocssor computers. Technical Report NBSIR 87-3627, Institute for Computer Sciences and Technology, National Bureau of Standards, Aug 1987.

[7] Ding-Kai Chen. MaxPar: An execution driven simulator for studying parallel sytems. Technical Report CSRD 917 and UILU-ENG-89-8013, University of Illinois, Oct 1989.

[8] R.C. Covington, S. Madala, V. Mehta, and J. B Sinclair. The Rice parallel processing testbed. ***In Proceedings of the 1988 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems.*** ACM, May 1988.

[9] Helen Davis and John Hennessy. Characterizing the synchronization behavior of parallel programs. In ***Proceedings of the ACM/SIGPLAN PPEALS Parallel Programming: Experience with Applications, Languages and Systems.*** ACM, July 1988. Published as Vol. 23, No. 9, of SIGPLAN Notices.

[10] Kenneth W. Dritz and James M. Boyle. Beyond "speedup": Performance analysis of parallel programs. Technical Report ANL-87-7, Argonne National Laboratory, Feb 1987.

[11] S. J. Eggers. Simulation analysis of data sharing in shared memory multiprocessors. Technical Report UCB/CSD 89/501, University of California, Berkeley, 1989.

[12] Susan J. Eggers, David R. Keppel, Eric J. Koldinger, and Henry M. Levy. Techniques for efficient inline tracing on a shared-memory multiprocessor. In ***Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems.*** ACM, May 1990.

[13] J.S. Emer and D. W. Clark. A characterization of processor performance in the VAX- 1 1/780. In ***Proceedings of the 11 th Annual Symposium on Computer Architecture.*** ACM-IEEE, June 1984.

[14] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In ***Proceedings of the***

*17th Annual Int'l Symposium on Computer Architecture.* ACM-IEEE, May 1990.

[15] Teemu Kerola and Herb Schwetman. Monit: A performance monitoring tool for parallel and pseudo-parallel programs. In ***Proceedings of the ACM SIGMETRICS Conf. on Measurements and Modeling of Computer Systems.*** ACM, May 1987.

[16] Kern Koh and Colin G. Harrison. GETTIMECARD - a basis for parallel program performance tools. Technical Report RC 14993, IBM T. J. Watson, Sept 1989.

[17] Frank Lacy. An address trace generator for trace-driven simulation of shared memory multiprocessors. Technical Report UCB/CSD 88/407, University of California, Berkeley, March 1988.

[18] D. Lenoski, K. Gharachorloo, J. Laudon, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. Design of scalable shared-memory multiprocessors: The DASH approach. ***COMPCON 90,*** 1990.

[19] Bradley J. Lucier. Performance evaluation for multiprocessors programmed using monitors. In ***Proceedings of the 1988 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems.*** ACM, May 1988.

[20] Lusk, Overbeek, and et al. ***Portable Programs for Parallel Processors.*** Holt, Rinehart and Winston, Inc., 1987.

[21] E. L. Lusk and R. A. Overbeek. Use of monitors in fortran: A tutorial on the barrier, self-scheduling do-loop, and askfor monitors. Technical Report ANL-84-51 Rev. 1, Argonne National Laboratory, Jun 1985.

[22] Allen D. Malony, Daniel A. Reed, Ruth A. Aydt James W. Arendt, Dominique Grabas, and Brian K. Totty. An integrated performance data collection, analysis, and visualization system. Technical Report TTR1 1, University of Illinois at Urbana-Champaign, March 1989.

[23] Barton P. Miller. DPM: A measurement system for distributed programs. Technical Report WISCCS 592, University of Wisconsin-Madison, 1985.

[24] Barton P. Miller, Morgan Clark, Steven Kierstead, and Sek-See Lim. IPS-2: The second generation of a parallel program measurement system. Technical Report WISCCS 783, University of Wisconsin-Madison, 1988.

[25] Jonathan Rose. LocusRoute: a parallel global router for standard cells. In ***2Sth ACM/IEEE Design Automation Conference.*** ACM-IEEE, 1988.

[26] Zary Segall and Larry Rudolph. PIE: A programming and instrumentation environment for parallel processing. ***IEEE Software,*** pages 22-37, Nov 1985.

[27] K. So, A. S. Bolmarcich, F. Darema-Rogers, and V. A. Norton. SPAN - a speedup analyzer for parallel programs. Technical Report RC12186, IBM, Sept 1986.

[28] K. So, F. Darema-Rogers, D. George, V. A. Norton, and G. F. Pfister. PSIMUL – a system for parallel simulations of the execution of parallel programs. Technical Report RC11674, IBM, Oct 1987.

[29] Larry Soule and Anoop Gupta. Parallel distributed-time logic simulation. ***IEEE Design & Test of Computers, .*** pages 32-48, Dec 1989.

[30] C. Stunkel and W. Fuchs. TRAPEDS: Producing traces for multicomputers via execution driven simulation. In . ***International Conference on Measurement and Modeling of Computer Systems,*** 1989. to be obtained.

[31] Charles P. Thacker and Lawrence C. Stewart. Firefly: a multiprocessor workstation. In ***Proceedings of the Second Int'l Conf. on Architectural Support for Programming Languages and Operating Systems.*** ACM-IEEE, Oct 1987.

[32] Pen-Chung Yew, Alexander Veidenbaum, and Hoichi Cheong. Chief: a parallel simulation environment for parallel systems. Technical Report CSRD 915, University of Illinois at Urbana-Champaign, Aug 1989.