Tango Lite:

A Multiprocessor Simulation Environment

Introduction and User's Guide

Stephen Alan Herrod
Computer Systems Laboratory
Stanford University
Stanford, CA 94305

herrod@cs.stanford.edu

This software is available by anonymous ftp from meadow.Stanford.EDU.

Abstract

Tango Lite is a software-based multiprocessor simulation environment useful in the study and design of both parallel architectures and the software for these machines. Tango Lite runs on MIPS-based uniprocessors and provides a multiprocessor environment through the multiplexing of lightweight threads onto this single context machine. These threads differentiate Tango Lite from Tango, its predecessor, and have led to greatly improved performance. Tango Lite also achieves efficiency by allowing the user to specify the level of detail appropriate for the simulation requirements.

1 Introduction

With the proliferation of multiprocessors , the need for accurate simulations has become a necessity for programmers and architects alike. New multiprocessor machines are significantly more complex than uniprocessors, with a multitude of possible configurations. An ideal development environment would allow real-world applications to run on a number of differently configured machines to determine the strengths and weaknesses of various design decisions.

While hardware prototyping is one possible means of evaluating new architectures, these systems can be slow, expensive, and difficult to reconfigure. Such factors limit the scope of studies available once the prototype is built. Analytic models are often ineffective in the evaluation of multiprocessor designs. Due to the complexity of the designs, these methods are often too restrictive and require simplifying assumptions that hinder the accuracy of the results. When such methods are not ideal for the knowledge desired, simulations are used.

A simulator emulates the behavior of various workloads in a specified environment. This environment encompasses all desired features and determines how this system would behave on a given workload. Attributes such as interconnects, processor architecture, and even operating system intervention may be included in this virtual machine. The architecture under study is referred to as the *target system* to distinguish it from the *host system* upon which it is run.

As with the architectures, parallel software can be quite complex, with few effective performance tools available. The pronounced effect of memory systems on parallel application performance requires the developer to carefully identify program bottlenecks in order to achieve the speedups attainable through multiprocessors. While there are many profiling tools available for software developers, very few can give accurate information as to the cause of performance bottlenecks. The ideal software development environment would have knowledge of the hardware and present statistics describing the interactions between application and system. The ability to obtain and present such information clearly is a requirement for effective simulations.

Another important issue in simulations is speed. Good simulators encourage successive refinement of design decisions. It can be extremely informative to alter a single system attribute and measure the effect it has on overall performance. Slow simulations restrict the ability to effectively use this feedback in the study of large systems with many variables. The fact that most interesting applications tend to be large and long-running only strengthens this performance requirement.

Tango Lite has been developed with these issues in mind. It provides a clean interface for the development of memory systems, along with an easily understandable programming model. The memory system can be as simple or as complex as needed, expediting data collection without unneeded overhead. On the application side, Tango Lite achieves much of its speed by running compiled user code to simulate the functional execution of the application on the target machine. Time is simulated with simple constant delays for CPU operations and by calculating the more costly memory operations only when necessary. Tango Lite also makes it easy to filter out events that are not needed in a simulation. For instance, local memory references, such as stack references, are often uninteresting in the study of shared-memory machines and can be eliminated to save time. Additional facilities are available to limit the scope of simulation to any desired subsections of the application.

When using Tango Lite, applications can be written in either C or fortran and must use the m4 macros developed at Argonne National Laboratory[4]. This macro package provides machine-independent abstractions for shared memory allocation, process creation and control, communication, and synchronization. Tango Lite is designed to run on MIPS R3000-based uniprocessors that run under an operating system providing support for shared memory and semaphores.

This paper is structured as follows. Section 2 discusses the general simulation approach employed by Tango Lite. Section 3 discusses the installation and general framework of the simulation environment. Section 4 and Section 5 discuss the options available in creating a complete simulation. The process for creating new memory systems for simulation is covered in Section 6. Section 7 gives a brief look at the performance of Tango Lite. Finally, the paper ends with a discussion of recent optimizations and some useful appendices.

2 Simulation Approach

Tango Lite simulates both the *functional execution* and also the *progression of time* of an application on the specified target system. During the compilation process, the user application routines are augmented with code to simulate time, manage the simulation threads, and perform desired synchronization operations. A simulation thread is created for each application process, which Tango Lite assumes is associated with a unique processor on the target machine. A Tango Lite scheduler multiplexes the threads during the simulation in a way that preserves the order of the events of interest while still allowing efficient execution. This method of instrumentation and execution allows the preservation of both functionality and timing behavior.

2.1 Functional Simulation

We simulate the functional behavior of a process by compiling the original application code and executing it on available hardware. To emulate synchronization primitives that are not available on the host machine, the software makes calls to special-purpose macros and libraries. Running compiled code directly on the uniprocessor is much more efficient than using a software interpreter, where each target instruction would be emulated through the execution of many instructions on the host machine. Because we are not interested in the details of local computations, this approach adequately mimics the functional behavior of the target system.

During simulation, the simulated processors are represented as lightweight threads. Compiled application threads are multiplexed to maintain the correct ordering for the operations being studied. A distributed scheduler is used: application threads are augmented to perform rescheduling before each of the operations under study (i.e., either synchronization, global, or all data operations). This ensures that only the thread furthest behind in simulation time performs such an operation. Because a "processor" performing local CPU operations is assumed not to affect any of the others, rescheduling threads at every individual instruction can be avoided without compromising the correct ordering of global operations.

2.2 Timing Simulation

Tango Lite divorces the progression of time on the target machine from the passage of real time; the actual time needed to perform the simulation does not affect the times recorded. There is a software clock associated with each thread that maintains the simulated time for that "processor". A threads clock is updated in each program basic block, at synchronization points, and at various data references. Whenever the delay for an operation can be determined at compile time, instructions are simply inserted into the code to increment the clock by the appropriate amount. Sequences of local CPU computation, for example, have such predictable delays. When the delay for an operation is not known at compile time, the clock update is handled through user-defined routines. Shared memory latencies, for example, may be known at compile time when a simple memory model is used, but in some studies they will have to be determined through simulation at runtime.

The individual thread clocks are not precisely synchronized, but are loosely regulated so that the timing of the operations of interest is preserved. There is a direct correlation between the degree of clock synchronization and overall simulation performance. A number of options for maintaining synchronization among clocks have been explored. The strictest and therefore most costly option is to synchronize at all local data references and all global operations. This is necessary for detailed cache simulations in which a cache miss on a local reference can cause network traffic that affects other processors in the system. Under a second option, the clocks are kept within one global operation of each other, differing only by the duration of sequences of local computation. This option is useful for studies that require precisely-ordered shared data references. For higher simulation performance, a third option ensures only that clocks be within one synchronization event of each other. This option is useful when studying synchronization patterns or shared memory reference patterns under certain programming models. A final alternative allows the user to specify an approximate number of clicks by which clocks may differ. This cannot be used to ensure that clocks are synchronized any closer than a single basic block and is intended to be used in conjunction with one of the first three options. By permitting the user to select an appropriate degree of clock synchronization, Tango Lite provides the accuracy required for a variety of studies while avoiding the expensive overhead of extraneous context switching.

3 Typical Tango Lite Installation

When you extract Tango Lite from the *Tango-Lite.tar* file, you will find that the directory "Tango-Lite" has been created. For the automatic makefiles to work, you should set the environment variable **LITEDIR** to be the path of this Tango-Lite directory. Within this **LITEDIR** are the following subdirectories:

- **applications**: As discussed in Section 4.1, Tango Lite provides a shared virtual-memory programming environment. Several applications adhering to such a programming model are included in this subdirectory. These SPLASH applications[5] employ different memory simulators to demonstrate how they may be used. Additionally, the makefiles for these applications will be helpful in understanding the process of simulation creation.
- **aug**: Discussed in Section 5.2, ***aug*** is the tool used to instrument events of interest in the simulated application. This sub-directory contains this executable in addition to its source code, documentation, and makefile.
- **docs**: Documents, such as this one, that deal with Tango-Lite.
- **include**: Several files used by Tango Lite are located in this subdirectory.
- **lib**: With the exception of the target system-specific files, all Tango Lite procedures will be accessed through library routines. This method allows inclusion of Tango Lite routines as needed without adding extra complexity to the compilation process.
- **libc**: Because Tango Lite instruments assembly code, special concern must be taken to handle calls to the standard C library. We have obtained the source code for these commonly used routines, and they can be included in the simulation makefile to be instrumented with the application. This method, which is demonstrated in the sample applications, leads to coverage rates of up to 99%. Section 5.3 discusses the use of these libraries in more detail.
- **libm**: Like **libc**, this sub-directory contains the assembly coded routines from the math library. By including them in the application's makefile, ***aug*** can instrument them as well.
- **macros**: This sub-directory holds the various ANL macro files that are used in the initial compilation phase of a simulation application. The different macro files available are discussed in Section 5.4.
- **make-irix**: This subdirectory holds the various core makefiles that will be included within your application's makefile. Files from this directory are to be used when running the simulation on a machine running the IRIX™ operating system. The different makefiles available are discussed in Section 5.5.

- **make-ultrix**: As above, this subdirectory holds the various core makefiles for inclusion in an application's makefile. This directory is to be used on machines running the ULTRIX™ operating system.
- **sims**: This sub-directory holds a number of sample machine simulations. They range greatly in complexity and provide a good example of what is involved in the creation of new simulation modules.

4 Creating a Parallel Application

This section discusses the steps involved in creating and compiling an application into a Tango Lite simulation. It would be helpful to look at the sample simulations as you read through this section. All of the examples assume that you have the environment variable LITEDIR set to the directory where Tango Lite has been installed.

4.1 Programming Environment

Many applications written for a shared-memory environment assume multiple address spaces where shared data must be explicitly allocated. In this model, each process gets a private copy of all variables except those that are explicitly declared to be shared. In the code fragment of Figure 1, the only shared data is `*globalMem`, an integer explicitly allocated using `GMALLOC()`. Writes to `temp1` and `local1` are never seen by the other processes and can be used to store data private to each process such as the process id. This model reflects the semantics of processes created by the UNIX `fork` system call.

The simulated processors are represented by threads in Tango Lite and, as a result, there is a single address space per program. In such a *shared virtual-memory* environment all variables are shared unless allocated from a thread's stack. In the code fragment of Figure 1, the only non-shared data are `arg1` and `local1`. Writes to `temp1` and `globalMem` will be seen by all other processes. Applications assuming that each process gets a private copy of all variables must be adapted to run under this model. In order to port such applications, all statically-allocated data modified during the parallel stage must be replicated.

```
int *globalMem;
char temp1;

slave(arg1)
   int arg1;
{
   int local1;

 globalMem = GMALLOC(sizeof(int));
 local1 = 42;
}
```

Figure 1:Code Fragment Illustrating Shared-Memory Semantics

4.2 Application Source Files

The application to be simulated on this virtual machine can be written in either C or Fortan and should utilize the m4 macros in $(LITEDIR)/macros for generic parallel constructs. This macro set originated at Argonne National Labs, and the original standard is described in [4]. These macros have been modified to support the tracing of synchronization operations in addition to allowing flexibility in the tracing method. The standard for naming application source files incorporating these macros is to append a ".U" suffix to the filename (e.g. SolveMatrix.U). Header files utilizing the macros are typically suffixed with ".H".

Following is a description of the macros that are part of the Tango Lite environment:

MAIN_ENV

This macro defines the global symbols, variables, and functions that make up the Tango Lite environment. This macro should appear at the beginning of the main source file for the application. (Every application should contain exactly one instance of this macro.)

EXTERN_ENV

This macro declares the symbols, variables, and functions that make up the Tango Lite environment. Unlike `MAIN_ENV`, however, this macro declares variables with the extern qualifier to indicate that the actual definition appears in another file. This macro should appear at the beginning of each source file except the main source file.

MAIN_INITENV(*share-mem*)

This macro initializes the Tango environment. It should be the first executable statement of the application (so that the environment gets initialized as soon as possible). The first argument is not used. The optional second argument *share-mem* may be used to alter the number of bytes of shared memory that are available to Tango Lite and the application.

CLOCK(*variable*)

This macro sets the specified *variable* to the current simulation time.

MAIN_END

This macro terminates the Tango Lite environment. It should be the last executable statement of the procedure main.

The following macros implement shared memory and process-creation primitives:

CREATE(*function-name*)

This macro creates and initializes a new Tango process and causes it to call the function function-name. The called function should have no arguments. The child process gets a copy of all of the parent process's code and data. It is a good idea to flush all output file buffers before invoking this macro. When the called function returns, the child process will terminate.

G_MALLOC(*amount*) and **G_MALLOC_F**(*amount*)

These macros allocate shared memory. Each returns a pointer to amount bytes of unused shared memory on the heap. The `G_MALLOC` macro is terminated by a semicolon, so it cannot be used in a C expression; otherwise, the two macros are identical.

G_FREE(*pointer*)

This macro releases a block of shared memory that was acquired with `G_MALLOC`.

The following macros implement synchronization primitives:

LOCKDEC(*name*)

This macro declares a lock variable with the specified *name*. Locks should only be declared in shared memory.

LOCKINIT(*name*)

This macro initializes the lock variable with the specified *name*. Locks are initially unlocked.

LOCK(*name*)

This macro acquires a lock variable with the specified name. Locks can only be acquired by one process at a time. If the desired lock is busy, the process waits until it is released.

UNLOCK(*name*)

This macro is used to release a lock variable with the specified *name*.

ALOCKDEC(*name*,*number*)

This macro declares an array of *number* lock variables with the specified *name*. Lock arrays should only be declared in shared memory.

ALOCKINIT(*name,number*)

This macro initializes an array of *number* lock variables with the specified *name*. All locks in the array are initially unlocked.

ALOCK(*name*,*i*)

This macro acquires the *ith* lock in a lock array with the specified *name*. Locks can only be acquired by one process at a time. If the desired lock is unavailable, the process waits until it is released.

AULOCK(*name,i*)

This macro releases the *ith* lock in a lock array with the specified *name*.

BARDEC(*name*)

This macro declares a barrier with the specified *name*. Barriers should only be declared in shared memory.

BARINIT(*name*)

This macro initializes a barrier with the given *name*.

BARRIER(*name*,*number*)

This macro causes the process to enter the barrier specified by *name*. Each process entering a barrier must wait until the specified *number* of processes have entered, at which time all the processes that have reached the barrier are released.

GSDEC(*name*)

This macro declares a self-scheduled loop subscript with the specified *name*. The subscript should be declared in shared memory.

GSINIT(*name*)

This macro initializes a self-scheduled loop subscript with the specified *name*.

GETSUB(*name,variable,subscripts,processes*)

This macro causes the process to schedule itself for the next available *subscript* of the loop specified by *name*. The total numbers of *subscripts* and *processes* are specified in the invocation of the macro. If no more subscripts are available, *processes* wait for the completion of all outstanding subscripts before proceeding, and the loop subscript is reinitialized.

WAIT_FOR_END(*num*)

This macro causes the process to wait for the completion of *num* other processes before proceeding.

PAUSEDEC(*name*, *number*)

This macro declares an array of *number* event flags with the specified *name*. If the second argument is omitted, a single flag is declared. Event flags should be declared in shared memory.

PAUSEINIT(*name*, *number*)

This macro initializes the array of *number* event flags with the specified *name*. If the second argument is omitted, a single flag is initialized.

SETPAUSE(*name*, *i*)

This macro sets the *ith* flag in the named array of event flags. All processes waiting for that flag to be set are released. If the second argument is omitted, the first flag is set.

CLEARPAUSE(*name*, *i*)

This macro resets the *ith* flag in the named array of event flags. If the second argument is omitted, the first flag is reset.

WAITPAUSE(*name*, *i*)

This macro causes the process to check the *ith* flag in the named array of event flags. If the flag is not set, the process waits on the flag until it is set. If the second argument is omitted, the first flag is used.

EVENT(*name*, *i*)

This macro causes the process to check the *ith* flag in the named array of event flags. If the flag is set, the process waits on the flag until it is reset. Once the flag is reset, the process sets the flag. If the second argument is omitted, the first flag is used.

PAUSE(*name*, *i*)

This macro causes the process to check the *ith* flag in the named array of event flags. If the flag is reset, the process waits on the flag until it is set. Once the flag is set, the process resets the flag. If the second argument is omitted, the first flag is used.

Here are the Tango extensions for simulation control:

AUG_DELAY(*n*)

This macro causes the current process to delay for *n* clicks. This can be used to stall a processor, to simulate some global operation, or hand-code desired contention.

AUG_ON and **AUG_OFF**

This macro is used to statically mark sections of user code that are not to be augmented in any way.

Appendix A provides a simple program that exemplifies the usage of these macros.

5 Building an Application into a Complete Simulation

While the makefile tries to shield the user from Tango Lite implementation, some information on the compilation process may be both interesting and helpful in tracking down problems. The overall process is illustrated below.

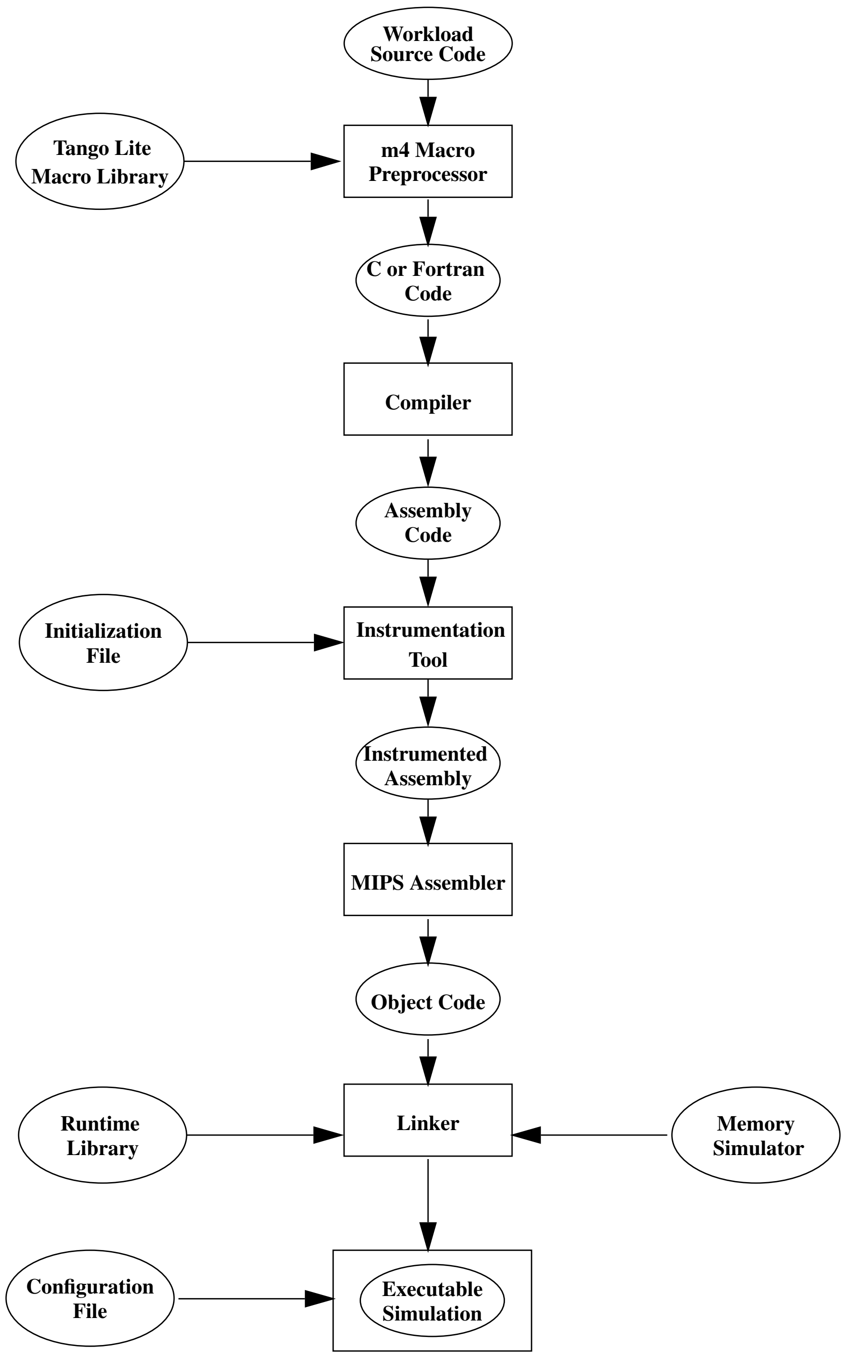

Figure 2:Compilation of a Tango Lite Simulation

Tango Lite instrumentation starts with the source code for the application. The first step in the compilation process involves the expansion of all ANL macros in the code. These synchronization calls may be broken down to the level of atomic reads and writes or kept at a higher monitor abstraction. Next, Tango Lite compiles this C or Fortran source into MIPS assembly code. This assembly code is the level at which ***aug*** will instrument the events to be simulated. After code has been inserted to preserve the register state, update the clock, and call the appropriate event handlers, this assembly language module can be compiled into an object file using a normal MIPS assembler.

This process is repeated for each module in the workload. The final step is to link the instrumented modules together along with the user's memory system simulator and Tango Lite's runtime library. This runtime library manages the processor clocks, handles thread scheduling, collects statistics, and performs other actions needed in the simulation execution. The linking step results in an executable program. When executed, this program will behave as if it were run on the simulated machine. Additionally, the application may utilize various run-time parameters and report various program performance statistics.

5.1 Application Makefiles

While the macro-filled source files could be compiled into a simulation by hand, the use of a makefile is strongly encouraged. This makefile will inform Tango Lite of source file locations, the run-time library to use, compilation options, and other important specifications. A commented makefile showing how Barnes might be built can be found in Appendix B.

The makefile is structured such that a number of user definitions are placed at the top followed by the inclusion of a Tango Lite makefile. In this fashion, the user can simply alter certain features of the simulation and **make** will do the rest. Following is a description of the flags used in a makefile along with a description of their purpose:

- `ASFLAGS` An optional flag used to pass arguments to the assembler. Typically "-O2" is used.
- `ASMS` All assembly files to be used should be listed here
- `AUGFLAGS` This flag tells ***aug*** what events are of interest and is used for various optimizations. These and other options for ***aug*** are discussed in Section 5.2.
- `CFLAGS` Any flags needed in the compilation stage of the application such as defines and optimizations. Using "-O2" is suggested.
- `CS` A listing of all ".c" files used in the application. If needed, they will be macro-expanded from a ".U" file.
- `HS` A listing of all headers used in the application. If needed, they will be macro-expanded from a ".H" file.
- `LDFLAGS` Flags for the loading stage. Typical uses include -O2 or -p for profiling.
- `LIB` The run-time library used to access Tango Lite functions. "liteu" is the optimized uniprocessor version.
- `LOADLIBES` Runtime libraries needed by the application such as -lmalloc.
- `MACS` There are a number of files available for macro expansion. The different options are discussed in Section 5.4.
- `OBJS` A listing of all object files used by the application
- `SIMOBJS` This is where all memory simulation object files are listed. After setting the SIMDIR variable to the location of your memory system simulators, you might use: $(SIMDIR)/basic_simm.o $(SIMDIR)/memsys.o and $(SIMDIR)/simple_uma.o

- `SS` Instrumented assembly files of your application. This is where libc and libm library files should be included.
- `TARGET` The name for the final Tango Lite application.

After all of the needed flags have been set, you should include one of the makefiles from the appropriate directory (make-irix or make-ultrix). These included makefiles provide the "glue" for building an entire simulation and can be written to provide additional functionality. For instance, one included makefile provides the framework for passing instruction addresses to the simulator. Section 5.5 discusses the various makefile's available for inclusion.

5.2 Aug

Aug is the instrumentation tool that takes one or more assembler files and augments them with simulation code. This code includes clock management, implementation of context switching, and handling of events of interest. The user specifies which events are to be simulated, optional names for the event handlers, the information that is to be passed to the simulator, and an initialization file. There are a number of options to ***aug***, some of which are automatically called as determined by the included makefile. The three most commonly used ***aug*** options are to simulate global memory references (-r flag), simulate frame references (-f flag), and enter the memory system at the beginning of basic blocks (-b flag). There are actually quite a few options that can be specified on the AUGFLAGS line of the makefile. Many of the currently implemented augmentation flags are used for flexibility in the build sequence and in various register optimizations. The options that the majority of users will want to include are:

- -P instrument procedure calls and returns. This may be useful in measuring the memory performance of particular procedures.
- -a don't pass addresses to event handlers. This can save time in the cases where more general memory behavior is of interest.
- -b instrument all basic blocks.
- -c combine reference type and event handler. This flag is discussed later and is used in conduction with "comb_sim.c". This is the most effective optimization discussed in this report.
- -f instrument stack-frame references. These can often be filtered out without losing too much memory system accuracy.
- -h specify that the simulator will handle data on loads and stores. This option requires quite a bit of effort by the memory system writer to ensure that the correct data is being stored in caches and is communicated over any simulated networks.
- -n pass source-code line numbers to event handlers. This may be useful for profiling the memory behavior of the code.
- -r instrument all global (non-stack) references. This is the most commonly used ***aug*** option.

5.2.1 aug_init

In order to instrument an application's assembly language files, ***aug*** looks at an initialization file. This file is named "aug_init" and can be found in the $LITEDIR/aug directory. This file describes each assembler mnemonic, giving the type of operation (load, add, branch, etc.), the default number of cycles, and which machine registers are used and modified. Through the information in this file, ***aug*** can statically determine how long certain non-memory operations will execute. While the included file should be adequate for most users, it can easily be altered to evaluate instruction implementation issues.

5.2.2 Instruction References

In-line instrumentation of a workload will substantially distort the applications instruction addresses. If these addresses are desired, special methods must be employed to minimize this distortion. In Tango Lite, this is accomplished through a two-pass compilation. The first compilation labels the workload and compiles it into an uninstrumented executable. During the second compilation, the same assembly code is instrumented normally. The original instruction address is extracted from the labeled executable and is encoded in the instrumentation to be passed to the memory system simulator at runtime. This labelling is enabled through the inclusion of special makefiles as discussed in Section 5.5.

5.2.3 Interfacing aug with the Memory System

To simulate these events, ***aug*** adds assembly code that sets up the current thread pointer with the needed information. The event type, reference address, and other information is recorded before jumping into the memory simulator. Entrance to the memory simulator starts with a call to *sys_call_routine*. *sys_call_routine* saves all of the registers to be used by the simulator and calls the appropriate event handler. The next layer in the memory simulation hierarchy is either *basic_sim* or *comb_sim*.

comb_sim is used in conjunction with the ***aug*** “-c” option and is an optimization combining the reference type and the event handler into one file. It immediately calls routines to load or store the appropriate memory. I will be concentrating on *basic_sim* from now on.

basic_sim has a procedure for each of the different event types that may have occurred. On a memory reference (*sim_ref_event*), a reschedule occurs first. This is to ensure that the thread furthest behind in execution time is simulated next. It then calls *sim_memsys*() and adds the instruction execution time to the current thread’s clock. *sim_memsys* makes calls to *sim_block_read*, *sim_block_write*, and other procedures based on the thread’s event. This is the level where people should add their own memory systems. Figure 3 is a pictorial representation of this type of entry into the memory system.

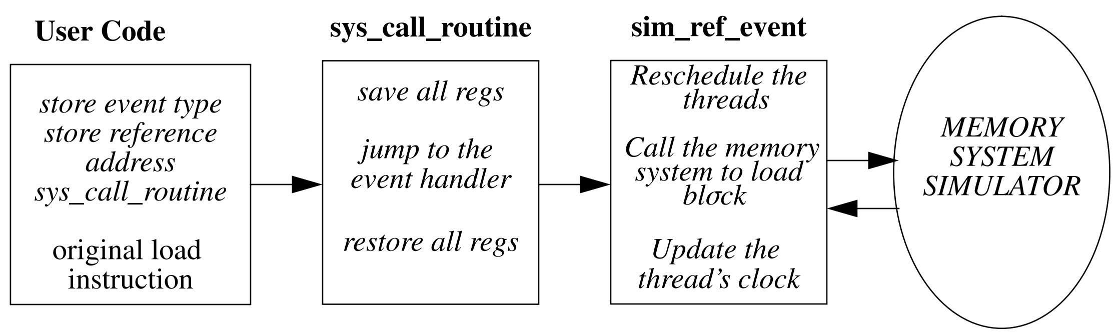

Figure 3:Interfacing with the Memory Simulator

5.3 Library Files

One problem with instrumenting an application at the assembly code level is that library routines included at link time will not be measured. This is one of the better arguments for instrumentation at the object code level. To help alleviate this deficiency, we have obtained the assembly code for a number of the *libc* and *libm* procedure calls. Inclusion of these files on the `ASMS` and `SS` lines of the application makefile will link in the instrumented versions of the desired library procedures.

While the coverage achieved through this method is not as complete as object code instrumentation, it has proven to be quite effective. Table 1 reports the estimated instrumentation coverage for eight applications. Much of the uninstrumented code is in the standard I/O routines. Since most of the I/O occurs during initialization, the coverage of the parallel stage of computation must be much higher. Because most simulations are concerned with the parallel sections only, uninstrumented code has very little effect on reported results.

Table 1: Instrumentation Coverage

Application	Instrumentation Coverage
Barnes	99.84%
Water	99.66%
Ocean	99.51%
Dynamic MP3D	98.46%
MP3D	98.44%
Cholesky	95.01%
PThor	90.94%
LocusRoute	85.53%

5.4 Choice of Macros

The macros used to expand synchronization events can model a number of different implementations. These range from expansion into low-level reads and writes to higher level monitor implementations. The files are found in the *$(LITEDIR)/macros* directory and behave as follows:

- **c.m4.mon** Implements a monitor-level synchronization model for “C”.
- **c.m4.not** Null set of macros that allow applications to run on a uniprocessor without Tango Lite.
- **c.m4.sim** Implements a high-level synchronization model for “C”.
- **c.m4.spin** Implements a reference-level synchronization model for “C”.
- **c.m4.tds** Implements a high-level synchronization model for “C”. The only difference between this and **c.m4.sim** is the implementation of GETSUB.
- **f.m4.not** Null set of macros that allow applications to run on a uniprocessor without Tango Lite.
- **f.m4.sim** Implements a high-level synchronization model for Fortran.
- **f.m4.tds** Implements a high-level synchronization model for Fortran. Again, the only difference between this and **f.m4.sim** in the implementation of GETSUB.

The ability to abstract the synchronization operations to multiple levels can be quite useful. For example, some architectures may use a separate synchronization bus while others must use atomic reads and writes to perform locks and barriers. Additionally, users may wish to write the synchronization macros for their own designs to measure the effect of various design decisions.

5.5 Choice of Included Makefiles

The final information needed in the application's makefile is the Tango Lite makefile to include. This makefile provides information on handling dependencies, sets a number of global definitions, and provides the "glue" for linking the application and memory system simulator together. The desired makefile should be included from either the *make-irix* or *make-ultrix* directory depending on your machine type. The available makefiles are:

- **lite.make** This is the most commonly used makefile. The compilation process will follow all of the default rules to produce a complete simulation executable.
- **lite-bigClock.make** This makefile provides the facilities for extremely long-running applications by providing a 64 bit clock. The memory system being simulated will automatically use these long clocks if it follows the clock handling conventions found in *basic-sys.h*.
- **not.make** This version generates an uninstrumented application that can be run directly on top of the host system. This method is used for debugging and performance measurements.
- **lite-execInst.make** As discussed in Section 5.2.2, this makefile forces a two-pass compilation process to take place. It requires three additional application makefile flags to be set. These flags are **LLOADLIBES**, **LOS**, and **LSS**, and are described within this makefile. The memory system must also be written to abstract the instruction addresses and simulate the desired i-cache behavior. This version uses labeled executable files to generate the instruction addresses.
- **lite-objInst.make** This version is nearly identical to lite-execInst.make. The difference is that this version uses labelled object files to generate the instruction addresses and doesn't require the **LLOADLIBES** flag to be set.

5.6 Running the Simulation

The compilation process produces a self-contained simulation application which is invoked as it would be on the parallel machine. During execution, the application will read a simulation configuration file. This file, *config.sim*, must be in the same directory as the application. It can contain various system parameters, debugging flags, and other simulator information required at runtime.

Output from the application itself will be interleaved with memory system statistics. While these statistics are flexible and memory-system specific, information typically includes overall execution time, cache miss rate, and useful "per-processor" timing data. The included memory simulators collect statistics for both the serial and parallel sections of an application.

6 Building New Memory/Machine Simulators

Tango Lite was designed to give the user a clean interface with which to build new memory simulators. Section 5.2.3 described one interface between the event generator and the simulated memory system. ***Aug*** will not add instructions for jumping into the memory system until an event of interest occurs. More formally, ***aug*** instruments the assembly code with four types of events. Each type of event is handled by a separate simulator routine.

- SYNCHRONIZATION EVENTS: generated from the original application code through the ANL macros and handled by *sim_sync_event*.
- BASIC BLOCK EVENTS: caused by application basic blocks that don't require loads or stores and handled by *sim_bb_event*.
- FRAME REFERENCE EVENTS: caused by data references with the stack pointer and handled by *sim_frame_event*.
- NORMAL REFERENCE EVENTS: caused by loads and stores not involving the stack pointer and handled by *sim_ref_event*

To better understand the creation of a new simulator, examination of the included memory systems is suggested. While a number of different types of simulators have been written, two simple methods will be discussed here.

The first simulator, **memsim4**, is the easiest to understand and will make a good starting point. It includes the procedures referred to in *sim_memsys* (*sim_block_read* and *sim_block_write*). In this example, contention within the memory system is not simulated, i.e., this memory reference will complete before another "processor" can access the memory system. This greatly simplifies creation by allowing simple procedure calls between the simulated TLBs and caches. The obvious trade-off is one of simplicity and speed at the cost of result accuracy.

The second type of simulation requires elaborate communication between modules to achieve more accurate results. This more complex system consists of a number of modules communicating through "ports" to allow contention in the memory system. The modules are *tlb.c*, *cache.c*, and *bus.c,* and they communicate using Tango Lite's included messaging utility (*message.c*). Contention is added at each module through the rescheduling of threads.

This messaging model is just one of a number of different communication protocols that have been used with Tango Lite. Complexity of the simulators has ranged from that of *memsim4* all the way up to a complete simulation of the DASH multiprocessor[3].

7 Performance

Tango Lite instrumentation increases both the execution time and the static size of a workload. This section quantifies these overheads by comparing the sizes and execution times of instrumented and uninstrumented versions of the same workloads. The workloads used were the Cholesky and MP3D applications running on top of an optimized *memsim4*-like simulator. The applications were run with 24 simulated processors.

The static size overhead is dominated by the Tango Lite run-time library and the memory system simulator linked in to the application. For these applications, this static overhead was just under nine megabytes, most of which was used for simulated cache tags.There is an additional size overhead due to the in-line instrumentation added by ***aug***. The amount of code instrumented varies with the code and desired information, but most applications will see the code grow by a factor of about 4.

Like static size overhead, runtime overhead depends strongly on the desired information and on the memory system simulator used. Simulating 24 processors with this baseline simulator ran between 50 and 140 times slower than the uninstrumented applications did on the same uniprocessor. With no memory system simulator, but preserving the order of loads and stores, the executions were between 10 and 40 times slower.

8 Conclusions and Future Work

Much of the current work on Tango Lite deals with methods for increasing performance. In order to accurately model more complex memory systems, quicker simulation methods will be needed. Context switching is extremely expensive in Tango Lite. Every time contention must be introduced into the system, a reschedule is performed and the state of another "processor" may need to be restored. Several relaxations have been applied to this rescheduling requirement to quantitatively measure the trade-off between speed and accuracy.

Another approach towards faster simulation has been to run these simulations on an existing multiprocessor. By running such simulations in parallel, similarities between the host and target systems can be exploited to achieve increased simulation performance. Coupled with the extra processing power of parallel machines, these architectural similarities can greatly decrease the simulation time required by even the most complex of systems.

Appendices:

Appendix A: Sample Program Utilizing m4 Macros

```
/*
   Example: Calculate the square root of a set of random numbers
   Usage: SquareRoots numberOfProcessors
*/
#include <stdio.h>
#include <math.h>

#define MAX_PROCS 4

/* Always include this at the beginning of the main source file
   for the application */
MAIN_ENV

struct globalMemoryType {
   int Index;                 /* Used by processors to determine which
                                 input to take */
   double input[MAX_PROCS];   /* Vector of input numbers */
   double output[MAX_PROCS];  /* Vector of the input number’s square roots */
   BARDEC(slaveBarrier);      /* Barrier declaration */
   LOCKDEC(indexLock);        /* Lock declaration to protect access
                                 to the Index */
} *globalMemory;

int numProcs;

void main(argc, argv)
int argc;
char *argv[];
{
   int i;
   double number;

   if (argc<2) {
    printf(“Usage: SquareRoots numberOfProcessors\n”);
    exit(-1);
   }

   /* Get the number of processors from the command line */
   sscanf(argv[1], “%d”, &numProcs);
   if (numProcs < 1 || numProcs > MAX_PROCS) {
    printf(“Bad number of Processors\n”);
    exit(-1);
   }

   /* Include this macro before any others are executed */
   MAIN_INITENV
```

```
   /* Allocate global, shared memory */
   globalMemory=(structglobalMemoryType*)G_MALLOC(sizeof(struct
                  globalMemoryType));

   /* Initialize the Index and currentProcessNum */
   globalMemory->Index = 0;

   /* Initialize the barrier and the locks */
   BARINIT(globalMemory->slaveBarrier)
   LOCKINIT(globalMemory->indexLock)

   /* Set up the input assignments */
   number = 1.0;
   for (i=0; i<numProcs; i++) {
    number *= 12.3;
    globalMemory->input[i] = number;
   }

   /* Create the slave processes. There are numProcs -1 since the
      master will do work, too. */
   for (i = 1; i < numProcs; i++)
    CREATE(SlaveStart)

   /* Make the master do slave work so we don't waste a processor */
   SlaveStart();

   /* Output the result */
   for (i=0; i< numProcs; i++)
    printf("The square root of %f is %f.\n",
            globalMemory->input[i], globalMemory->output[i]);

   /* Include this as the last executable statement of main() */
   MAIN_END
}

/* This is the routine that each processor will be executing in parallel */
SlaveStart()
{
   int myIndex;

   /* Get an index to work on */
   LOCK(globalMemory->indexLock)
   myIndex = globalMemory->Index;
   globalMemory->Index++
   UNLOCK(globalMemory->indexLock)

   /* Do the work of square rooting */
   globalMemory->output[myIndex] = sqrt(globalMemory[myIndex]);

   /* Stop at the barrier to synchronize then exit */
   BARRIER(globalMemory->slaveBarrier, numProcs)
}
```

Appendix B: Sample Simulation Application Makefile

```
#!/bin/make -f

# Flags used in the assembly stage (optional)
ASFLAGS  = -O2

# All of the .s files used in the application
ASMS     = code.s code_io.s load.s grav.s getparam.s util.s

# Flags telling aug what sort of instructions to instrument. This
# particular example tells aug to instrument both global and stack references
AUGFLAGS = -fr

# Flags for the compilation stage (optional)
CFLAGS  = -DOWNEDCELLS -DTHREEDIM -O2

# All of the .c files used in the application
# If needed, these will be macro-expanded from a .U file
CS       = code.c code_io.c load.c grav.c getparam.c util.c

# All headers used in the application
# If needed, these will be macro-expanded from a .H file
HS       = $(SRC).hcode.h defs.h stdinc.h vectmath.h

# Flags for the loading stage (optional)
LDFLAGS  = -O2

# The run-time library to be used. "liteu" is the uniprocessor version.
LIB      = liteu

# This simulation should use a high-level synchronization model
MACS     = $(LITEDIR)/macros/c.m4.sim

# All of the object files used by the application
OBJS     = exp.o floor.o log.o matherr.o pow.o sqrt.o frexp.o ldexp.o modf.o
         strchr.o code.o code_io.o load.o grav.o getparam.o util.o

# This is where all memory simulation object files are listed.
SIMDIR   = $(LITEDIR)/sims/obj-u
SIMOBJS  = $(SIMDIR)/basic_simm.o $(SIMDIR)/memsys.o $(SIMDIR)/memsim4.o

# Instrumented assembly files of your application
SS       = exp.S floor.S log.S matherr.S pow.S sqrt.S frexp.S ldexp.S modf.S
         strchr.S code.S code_io.S load.S grav.S getparam.S util.S

# Final Tango Lite application file
TARGET   = BARNES

# Create the default simulation
include $(LITEDIR)/make/lite.make
```

Acknowledgments

This material is based upon work supported under a National Science Foundation Graduate Fellowship.

Much of this work comes from the original *Tango Introduction and Tutorial* written by Stephen Goldschmidt and Helen Davis. The performance numbers in Section 7 are taken from Stephen Goldschmidt's thesis. I am additionally grateful to Stephen Goldschmidt for his patience in helping me understand the intricacies of Tango and Tango Lite.

Bibliography

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

[2] Stephen R. Goldschmidt. *Simulation of Multiprocessors: Accuracy and Performance*. PhD thesis, Stanford University, 1993.

[3] Dan Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John L. Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, pages 63-79, March 1992.

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

[5] Pal Singh, J., et al. "SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*. vol. 20, no. 1, p. 5-44