Tango Introduction and Tutorial

Stephen R. Goldschmidt and Helen Davis

Technical Report No. CSL-TR-90-410

January 1990

This research has been supported by DARPA contract N00014-87-K-0828. Authors also acknowledge support by fellowships from the Fannie and John Hertz Foundation.

Tango Introduction and Tutorial

Stephen R. Goldschmidt and Helen Davis

Technical Report: CSL-TR-90-410

January 1990

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

Abstract

Tango is a software-based multiprocessor simulator that can generate traces of synchronization events and data references. The system runs on a uniprocessor and provides a simulated multiprocessor environment. The user code is augmented during compilation to produce a compiled simulation system with optional logging. Tango offers flexible and accurate tracing by allowing the user to incorporate various memory and synchronization models. Tango achieves high efficiency by running compiled user code, by focusing on information that is of specific interest to multiprocessing studies, and by allowing the user to select the most efficient memory simulation that is appropriate for a set of experiments.

Key Words and Phrases: multiprocessors, parallel processing, simulation, trace generation.

Contents

Chapter 1

Introduction

Simulation plays an important role in hardware and software studies by providing measurements needed to evaluate proposed architectures and algorithms. This paper discribes the methodology and use of Tango, a software-based simulation facility that supports a wide range of multiprocessing studies.

In multiprocessing studies, it is the parallel aspects of the computation that are of greatest interest. These can be studied by examinin***g the global operations,*** operations that ***are*** visible to more ***than*** one process. ***Private*** or ***local operations,*** which are visible to a single process, are generally interesting only with respect to how they affect the timing of global operations. In the shared-memory programming paradigm, global operations are shared data references and synchronization operations; in the message-passing paradigm, they are message sends and recieves. Because of their importance in many investigations, Tango allows the user to efficiently focus on these global operations. In very detailed studies, it may be necessary to consider the impact of local data operations on the performance of shared network and memory resources. For this reason, Tango can also simulate local data operations in detail.

The accuracy of results is an important issue in simulation. Accurately tracing parallel programs is difficult because the execution path through the program often depends on the real-time behavior of the hardware system (this is generally not the case for serial programs). Tango uses CPU timing parameters and a memory system simulator to model the real-time behavior of the target machine under study. The execution path taken during simulation is determined by this machine model. Simple abstract machine models permit us to study program requirements in a relatively machine-independent environment. More detailed and realistic machine models are necessary in some hardware studies. Tango also provides allows users to incorporate their own specialized simulator for network, memory, and sychronization systems. By allowing the user to vary CPU timing parameters and incorporate appropriate memory models, Tango accurately supports the needs of a wide range of studies.

Efficiency is also an important issue in simulation, since interesting applications tend to be large and long-running. The time and space required for event logging may be significant, so Tango allows the user to choose between: logging all data memory operations, logging global operations, logging synchronization operations, and simulation without logging. (Tango does not provide instruction traces, since this would very time consuming, and instruction references do not have a significant impact in many multiprocessor invesigations.) Tango gains much of its efficiency by running compilied user code to simulate the functional execution of the application on the target machine. Time is simulated by using simple constant delays for CPU operations, and using more costly simulations for memory operations only when necessary. Tango's simplest, and most efficient, memory simulation assumes uniform memory access times. Greater accuracy is provided by more detailed memory simulations that can simulate non-uniform memories and contention. Thus, Tango achieves high efficiency by running compiled user code, by focusing on information that is of specific interest to multiprocessing studies, and by allowing the user to select the most efficient memory simulation that is adequate for a set of experiments.

When using Tango, applications should be written in the C programming language, and use m4 macros developed at Argonne National Laboratory [3]. The macro package is a set of C macros that provide machine-independent abstractions for shared memory allocation, process creation and control, communication, and synchronization. Some initial work has been done toward providing a similar Fortran Tango environment.

Our implementation is designed to run on 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 3100 under Ultrix. The number of application processes is limited by the number of processes and semaphores the system allows and various table sizes, and is currently 256 in our systems.

Chapter 2 outlines the simulation methods used by Tango. For a more detailed discussion of Tango's implementation, see Reference [1]. Chapter 3 describes the application source code conventions and outlines the Tango macro set. Chapter 4 explains how to compile the application into an executable Tango simulation. The formats of parameter files are described in Sections 5.4 and 5.5, and a sample `makef ile` is presented in Chapter 7. Chapter 6 explains how to run a simulation and describes the output files generated. Chapter 8 gives advice on many pitfalls that are commonly encountered with Tango. Chapter 9 describes the Tango software and related utilties.

Chapter 2

Simulation Approach

Tango simulates both the ***functional execution*** and also ***the progression of time*** of an application on some target system. A simulation process is created for each application process, which Tango assumes is associated with a unique processor on the target machine. The user application routines and libraries are augmented during the compilation process with code to simulate time, manage the simulation, and log events. A Tango scheduler multiplexes processes during the simulation in a way that preserves the order of the events of interest.

2.1 Functional Simulation

We simulate the functional behavior of a process by compiling the associated application code and executing it on available hardware. To emulate synchronization primitives that are not available on the real machine, the software makes calls to macros or software libraries. Running compiled code in this manner is much more efficient than using a software interpreter, where each target instruction would be emulated by executing many instructions on the real machine. Beacuse we are not interested in the details of local compuatations, this approach adequately mimics the functional behavior of the target system.

During the simulation, compiled application processes are multiplexed to maintain the correct ordering for the operations being studied. A distributed scheduler is used: application processes 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 process furthest behind in simulation time performs such an operation. Since, a process performing local CPU operations is assumed not to affect any other process, this rescheduling maintains a correct ordering of the operations under study.

2.2 Timing Simulation

Tango 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 in traces. There is a software clock associated with each process that maintains the simulated time for that process. A process's clock is updated in each program basic block, at synchronization points, and optionally at shared data references or all 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 done by calling routines to perform the necessary calculation or simulation 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 by simulation at runtime.

The individual process clocks are not precisely synchronized, but are loosely synchronized so that the timing of the operations of interest is preserved. There are four user options for maintaining synchronization among clocks. The strictest, and most costly, option is to synchronize at local data references and 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 exclusively local computation. This option is useful for studies that require precisely-ordered shared data references. For higher simulation efficiency, a third option ensures only that clocks be within one synchronization event of each other. This option is useful when studying synchronization, and also can be used to study shared memory reference patterns in some programming models, or when trading off some accuracy for increased efficiency. 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 it is intended to be used in conjuction with one of the first three options.

By permitting the user to select an appropriate degree of clock synchronization, Tango provides the accuracy required for a variety of studies without doing more process switches than necessary.

2.3 Logging Events

When an application is run under Tango, an event log is optionally produced for each process. Each event in the log is a twelve-byte binary record, which includes the issuing process, the originating source code line, the operation (a memory or synchronization event type), the associated data address, and a time stamp. The format of these log files is described in Section 6.3.

To achieve this, Tango synchronization macros have been modified to optionally log synchronization events, such as when a lock is acquired or released. In addition, each application can be automatically instrumented at the assembly level to log data references. The user can disable logging for particular sections of code (either statically or dynamically) by means of special macros in the source code. When logging is disabled statically, the specified section will not be instrumented with logging code, and there will be no overhead associated with logging for these sections of code. Where logging has not been turned off statically, code is inserted to conditionally produce a trace log, depending on the run-tune logging status. In this case, since a check must be made at run time, there is a small overhead associated with logging even when it is dynamically disabled.

For applications which consist of multiple source files, Tango provides a back-referencing feature to allow log entries to refer back to a specific source file.

[1] a *click* is the smallest unit of time understood by 'Tango

Chapter 3

Application Source Files

The application program should be written in the C programming language and augmented with special m4 macros for parallel constructs. The macro set originated at Argonne National Labs, and the original standard is described in Reference [3]. Tango uses a modified macro set that has extensions to support synchronization tracing and target machine simulation.

C source code files (with macros embedded) should have the suffix ".C" on their names; C header files (with macros embedded) should have the suffix ".H".

The following macros implement the Tango environment:

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

`EXTERN_ENV` This macro declares the symbols, variables, and functions that make up the Tango 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 and the application.

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

`MAIN_END` This macro terminates the Tango 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 mcaro. When the called function returns, the child process will terminate. (See also Section 8.8.)

`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 relases 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 named barrier.

`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.

The following macros implement message-passing primitives:

SRDEC (*name* , *number*) This macro declares *an array* of *number* send-receive flags with the specified *name.* If the second argument is omitted, a single flag is declared. Event flags and associated message buffers should be declared in shared memory.

SRINIT (*name*, *number*) This macro initializes an array of *number* send-receive flags with the specified name.

SEND (*name*, *code*, *number*) *This* macro waits for the specified send-receive flag to be reset, executes **the** supplied user code to copy a message into a buffer, and sets the flag. with the specified name.

RECEIVE (*name*, *code*, *number*) This macro waits for the Specified send-receive flag to be set, executes the supplied user *code* to copy a message from a buffer, and resets the flag. with the specified *name*.

Here are the Tango extentions for simulation control:

ST_LOG (*type*, *address*) to write an event record with the *specified type* and *address* to the processor's log file (if logging is enabled) and pass the event to the timing simulator (if enabled).

AUGDELAY (*n*) to delay the current processor for *n* clicks.

AUG_ON and AUG,OFF to statically mark sections of code that are not to be augmented in any way.

TRACE-ON **and** TRACE-OFF to statically mark sections of code that are not to generate any log records.

REF_TRACE_ON **and** REF_TRACE_OFF to statically mark sections of code that are not to generate any data reference log records. (Synchronization event logging is unaffected.)

DYN_TRACE_ON **and** DYN-TRACE-OFF to dynamically start and stop generation of log records.

DYN_REF_TRACE_ON **and** DYN_REF_TRACE_OFF to dynamically start and stop generation of data reference log records. (Synchronization event logging is unaffected)

DYN_SIM_ON **and** DYN_SIM_OFF to dynamically start and stop calls to the memory model simulator.

Chapter 4

Compilation into a Tango Simulator

This chapter describes how to compile application code into a Tango simulation. There are five steps in compiling an application: macro expansion, compilation, augmentation, assembly, and linkage. This differs from the normal compilation process in several ways. Under Tango, the macro expansion uses a modified macro library, the augmentation step has been added, and the link includes a special Tango run-time library.

In addition to the Tango macro library and run-time library, two parameter files are used as input to the simulation process: `aug_init` (read during the augmentation step) and `tango_init` (read at run time). These files are used to specify timing and machine model parameters (see Sections 5.4 and 5.5). One may use the default files or substitute customized parameter files.

A csh script, tango, has been used to simplify the process of compiling a simulation. Given a list of file names, the script can perform the steps necessary to produce an executable simulation. When making changes to the application, the script can be invoked from the make utility to perform each compilation step only when it is required. Chapter 7 shows a sample `makefile` for this purpose.

In the following sections, the tango script options are defined, and then the compilation steps are described in detail.

4.1 Tango Script

The tango script takes a list of file names and options, which it reads from left to right. All options begin with a dash (-).

Here are the most commonly-used options:

`-s` *n* Specifies that *n* steps of the compilation process be performed on each file (see Section 4.2). The default is to perform a single compilation step.

`-o` filename Specifies that the name of the final executable file should filename. The default name is `a. out`.

-M *machineModel* Specifies *the* desired machine model, which determines the Tango object library and macro library to be used. Section 5.1 describes the currently defined machines models. One may specify a user-defined machine model using this option in conjunction with the `-tmac` and `-tlib` options, described in Section 5.2.

`-trace` *ops* Specifies which events should be logged (see Sections 2.3 and 6.3). *ops* should be one of the following:

A Log all data references and synchronization operations.

G Log all global operations, that is, shared data references and synchronization operations.

S Log synchronization operations only.

`-order` ***ops*** ***Specifies*** that the order of operations of type ***ops shall be*** maintained to agree with that of the simulated target machine. This results in rescheduling each process at every operation of type ***ops. The*** operations simulated by the machine simulator are always kept ordered by the simulation code.

The value of ***ops*** may be as follows:

A Preserve the order of all data references and synchronization operations.

D ***n Preserve the*** order of synchronization operations exactly, and also preserve the order of other operations to within approximately ***n clicks.*** (After each ***n clicks,*** rescheduling is done at the end of the current basic block).

G Preserve the order of all global operations, that is, shared data references and synchronization operations.

S Preserve the order of synchronization operations only. This is the default.

Here are some less-commonly used tango script options:

`-aug` ***"options" The string*** of options, ***options,*** is passed to any invocation to the **AUG** program during ***the*** augmentation phase of the compilation process.

`-augCalls` Indicates that subroutine calls are modified so that augmented versions of the system run-time routines will be called in place of the normal ones. If this option is omitted, each run-time routine will be heated as a single instruction and shared data references within runtime routines will not be instrumented,

`-cc` ***"options" The string*** of options, ***options,*** is passed to each invocation of the C compiler cc. The `-1` option (below) should be used to specify object libraries.

`-countRefs` Generate code to count data references *and* clicks. The totals for each process are writeen to the summary files (see Section 6.2). The default is to count clicks only.

`-1` ***"x"*** This option causes the loader **ld** to search for undefined symbols in the object library `libx.a`. Libraries specified in this fashion are searched before Tango run-time libraries.

`-v` Run tango and augmentation in verbose mode, informing the user of progress. The default is to run quietly.

Additional options, used to define new machine models, are described in Section 5.2.

Any word in the command line that is not recognized as an option is assumed to be a file name. When a file name is encountered, the script performs compilation steps on that file using the option settings read up to that point.

The suffix of the file name (the portion after the dot) is used to determine the type of file and thus the current step of the compilation process. Files with the extention "`.C`" are assumed to be *C* code with unexpanded macros. The suffix "`.c`" indicates expanded C code. "`.s`" indicates normal MIPS assembly code, "`.S`" indicates augmented assembly code, "`.o`" indicates object code, and "`.a`" indicates an object library.

Thus, for example, the command

```
tango main.C
```

performs macro expansion on ***the*** unexpanded ***source*** file `main.C` to produce ***an*** expanded ***source*** file `main.c` ***using all*** the default options.

```
tango -s 5 -0 mysim main.C
```

performs the complete compilation process on the same tile, producing an execuatble `mysim`.

```
tango -s 5 -o mysim -M hlsync -trace S main.C
```

performs a similar compilation process, but using the `hlsync` machine model and producing ***an*** executable that will generate a log of all synchronization activity.

4.2 Compilation Steps

As mentioned earlier, `Tango` compilation consists of five steps. The first step is to expand the **m4** macros which provide machine and synchronization abstractions. Normally, the user specifies the macro library indirectly, by specifying the target machine model with the tango script's `-M` option. In this case, the macro library name is simply `c . m4 .` ***machine-model.*** The standard macro libraries are maintained in the directory `/usr/local/lib/tango`. Alternately, the user may specify a different macro library directory to the tango script with the `-tmac` option (see Section 5.2).

If the script is used, the **awk** program is also invoked during the macro expansion phase to elimiate blank lines. This is done because the expansion tends to produce many blank lines. Since the back-referencing feature of `Tango` is **only** effective on the first 2047 lines of each expanded source tie, it is helpful to reduce the number of lines as much as possible. **On** some machines, the **indent** program may also be used to prettify the code.

The second compilation step is to compile the C code to produce a normal MIPS assembler file with the **suffix** "`.s`". In this step, the C pi-e-processor symbol `AUG` should be **defined** using the `-DAUG` option to the cc command. This symbol indicates that the code **will** be augmented (in the next step). The tango script defines this symbol automatically. In addition, a number of compilation options are **specified** in this step by **definining** other cpp symbols.

The third step in the `Tango` compilation process is to augment the assembler code with the **AUG** program (see Section 9.1) to produce a "`.S`" **file.** This step inserts assembly code to update the virtual time for each process, to call the memory system simulator, and to output data reference logging information. The `aug_init` (Section 5.4) **file** specifies to **AUG** the delay associated with each local operation. If augmented libraries are to be used, the augmented code may contain **calls** to unaugmented function, so the augmented code is merged with the plain assembler code to produce the "`.S`" **file**.

The fourth step is to assemble the "`.s`" **file** into an object module using **as,** the MIPS assembler.

When all the modules of the application have been compiled into "`.o`" **files,** the fifth and final step is to link them together with the required object libraries ("`.a`" files), to produce the executable for the simulation. The standard `Tango` libraries provide miscellaneous code that is part of the `Tango` environment, such as the routine that allocates shared memory.

When using the tango script, the correct library for the **specified** machine model is ordinarily included automatically on the basis of the machine model selected with the `-M` option. In this case, the library name is simply `lib`*machine-model* `. a`. The **standard** object libraries are maintained **in** the directory `/usr / local-/lib/tango`. Alternately, the **user may** specify a different object library **directory** to the script with the `-tlib` option (see Section 5.2).

If augmented libraries are to be used, then the special versions of the standard run-time libraries (such as `libc. a` and `libm. a`) must be included in the link The names of the augmented libraries are formed by inserting a string beginning with "`aug`" before the extention of the original library name. 'The characters in this string indicate the augmentation options that apply to the library. Thus, `libcaug. a` is a version of `libc. a` which has been augmented for timing only, `libcauglsf a. a` has been instrumented for timing and tracing, and so on.

Currently, augmented libraries are available **only** on the Mips **M/120,** and they are located in the directory `/usr/local/lib`. The tango script does not automatically add these to the **link,** so they must be specified with the `-1` option.

Chapter 5

Target Machine Models

This section describes standard Tango machine models. It also explains how to customize a Tango simulation by modifying the timing parameter files or by writing a new machine model simulation.

5.1 Standard Machine Models

Several standard machine models are currently defined. Some (such as `default`) are abstract models that may be useful in algorithm studies. Some (such as DASH) provide detailed simulation for specific architectures. Others are interfaces for those who would like to integrate their own memory system simulators into the Tango envrionment.

Once the machine model is chosen, parameter files allow the user to specify timing characteristics for machine models. There are two standard parameter files. The `aug_init` file, described in Section 5.4, is read at compile time and specifies CPU timing parameters; the `tango_init` file, described in Section 5.5, is read at run time and specifies timing characteristics of the memory and synchronization systems.

The tango script `-M` option is used to select one of the standard machine models. When a machine model provides an interface for a user memory system simulator, the user should also use the `-sim` option to specify the type of simulator provided (see Section 5.2).

`-M` *machineModel* This option selects the machine model for memory system timing simulations and synchronization abstractions. Current standard machine models include:

`default` This is the default model. Latencies for synchronization operations are calculated by the user-provided function, `Mem_Latency` (see Section 5.3). If no such function is defined, constant latencies for synchronization operations are read from the `tango_init` file, as described in Section 5.5.

If the `-sim TIME G` option is specified with this machine model, the extra latencies for a shared references (the numbers of clicks needed in excess of the times taken by the equivalent local references) are determined in a similar manner (by calling `Mem_Latency` or reading the `tango-init` file). Otherwise shared references do not take any time in excess of that required for local references.

`cachedLocks` Shared memory and synchronization operation latencies are determined as in `default`. Synchronization operations generate log records that correspond to the network traffic resulting in a system where locks are implemented as shared data and are cached with an invalidation protocol. (Blocked lock operations are re-issued after unlocks.)

`hlsync` The memory simulator, `Mem_Sim`, is called at each shared data reference (see Section 5.3). It is also called with "high-level" synchronization operations, which are those implemented by the synchronization macros. When data reference logging is active, each reference generates a two log records, one for the time of issue and one for the time when the request was satisfied.

`DASH` References and timing arc as in the proposed DASH architecture. [2].

5.2 User-Provided Machine Models

One can create a new machine model by substituting a custom macro file and creating a new Tango run-time library. The `tango_init` parameter file may also be extended to include the specification of new timing parameters for user-defined machine models (see Section 5.5).

The following **tango** options are used to customize a simulation:

`-sim` ***type ops*** Specifies the type of memory system simulator. If ***type*** is `TIME`, the simulator handles timing only and data values are managed by the application code; this is the default. If ***type*** is `DATA`, the simulator receives a copy of the data for each applicable write and is allowed to modify the data returned by each applicable read in addition to updating the clock.

The ***ops*** word specifies at the operations at which the simulator will be called:

`A` at all data references and synchronization operations.

`G` at all global operations, that is, shared data references and synchronization operations.

S at synchronization operations only.

`-M` *machineModel* Specifies the machine model name, which is used to construct library and macro file names. The Tango library file is named `lib`*machineModel*`.a`, and the the Tango macro file is named `c.m4.`*machineModel*. The directories for these files are determined by the `-tlib` and `-tmac` options.

`-tlib` *tangoLibrary* Search the directory *tangoLibrary* for the tango object library specified by the `-M` option. The default is `/usr/local/lib/tango`.

`-tmac` *macroDirectory* Search the directory *macroDirectory* for the tango macro library specified by the `-M` option. The default is `/usr/local/lib/tango`.

`-p` Turns on Tango profiling. Prints the (real) time spent executing memory simulations, context switches initiated by the tango scheduler, and the augmented application code.

Implementation code for the standard tango macros and libraries is maintained in the directory `/usr/local/src/tango/lib`. The common source files are in the subdirectory `/usr/local/src/tango/lib/src`. There is also a subdirectory `/usr/local/src/tango/lib/`*model* for each machine model containing the library routines and macro files unique to that model. It is suggested that new user machine models be constructed similarly.

Notice that, since user libraries are linked in before Tango libraries, a user may redefine subsets of the standard Tango library functions using the `-l` option.

5.3 User-Provided Functions

This section describes Tango functions that may be rewritten by the user to provide customized timing or machine models.

Most of the functions which provide customized timing and memory simulation are specific to a particular Tango machine model. For the `default` model, the interface consists of the functions `Mem_Latency_Init` and `Mem_Latency`. For the `hlsync` model, the interface consists of `Mem_Sim_Init`, `Mem_Sim`, and `Mem_Sim_End`.

In addition, Tango supplies hooks for passing input parameters into a simulation and writing results to the summary files. These include the functions `aug_read_other` and `aug_summary_other`.

The functions used to provide new machine models include `Mem_Issue`, `Mem_Init`, `Mem_Sync`, `Mem_Wait`, and `Mem_End`.

53.1 Simulator Interface for `default` Model (`MemLatency`)

`MemLatency` is used by the `default` machine model to allow the user to specify the latency of shared data references and synchronization operations as a simple function. It does not allow the simulator to return to Tango with any outstanding memory requests.

```
int Mem_Latency (pid, op, addr, data)
     int pid;            /* processor identification */
     int op;             /* operation */
     char *addr;         /* operation address */
     char *data;         /* operation data, used when appropriate */
```

If `MemLatency` is used, a function `Mem_Latency_Init` should be **defined** to initalize any data needed by `MemLatency`.

```
void Mem_Latency_Init ()
```

5.3.2 Simulator Interface for `hlsync` Model (`Mem_Sim`)

`Mem_Sim` is used by the `hlsync` machine model to allow the user to provide a detailed simulation of the memory and syncrhonization system. This machine model allows the simulator to have uncompleted memory requests in progress when it returns control to Tango.

```
  int Mem_Sim (issueQ, t_issue, t max, t_mem, doneQ)
        sc_blk **issuedQ; /* list-of memory ops to issue */
        int t_issue;      /* time of issue for memory ops */
        int t_max;        /* upper bound for simulation time */
        int *t_mem;       /* return time simulated up to */
        sc_blk *doneQ;    /* return list of ops exiting memory */
```

The simulator should simulate until simulation time `t_max` is reached or an operation exits the memory system, whichever comes first. The simulator sets `t` mem to the new simulation time, which indicates that the simulator is ready to accept operations that would occur on the target machine at that time. If the simulation time exceeds `ti s sue, the` list of operations, `is sueQ`, should be entered into the memory system at that time (and `issueQ` **set** to `NULL`). **If the** `return` **value of** `doneQ` **is non-**`NULL`, then `doneQ` **is** alistofoperationsthat have completed, and `tmem` is the time when they completed.

(See the file `/user/include/tango/tangodefs.h` **for a more complete description of the data structures** used by the `Mem_Sim` function.)

If the `Mem_Sim` interface **is used, the** functions `Mem_Sim_Init` and `Mem_Sim_End` should be definedto **initalize** `any data` needed by `MemLatency` and clean up the simulator state when the application is finished.

```
  void Mem_Sim_Init ()
  void Mem_Sim_End ()
```

5.3.3 Simulation Parameter Interface (`aug-read-other`)

`The` user `can write code` to handle otherwise **undefined** `tangoinit` tags (Section 5.5) by rewriting this fllnction.

```
void aug-read-other (tag, fp, fspec)
  char tag[], fspec[];
  FILE *fp;
```

This function is called during the initialization of the Tango run-time envrionment whenever an unrecognized tag is encountered in the `tango_init` file. It takes a tag word, a pointer to an open file, and the tile name. Based on the identity of the tag word, the user function should read the arguments from the file and initialize the simulator accordingly.

The userversionof `aug-read-other` may make use of the Tango functions `aug_read_word` and `aug_-read_num` to read the arguments of the user-defined tags:

```
int aug_read_word (buf, len, fp, fspec)
  char buf[], fspec[];
  int len;
  FILE *fp;

int aug_read_num (fp, fspec)
  FILE *fp;
  char fspec[];
```

`aug_read_word` takes a buffer with a specified length in bytes. It reads a word (delimited by whitespace) into the buffer and returns the length of the word. If the end of the line is read, it returns zero. If the end of the file is encountered, it returns -1.

`aug_read_num` reads a word and converts it to a number. The resulting integer is returned.

5.3.4 Simulation Results Interface (`aug-summary-other`)

This function is called at the completion of each Tango process. It may be used to print out simulation statistics. It is passed the name and file structure for the process's summary file (Section 6.2). User output to the summary file will appear after the normal Tango statistics.

```
void aug_summary_other (fname, sum fp)
  char fname[];
  FILE *sum_fp;
```

5.3.5 Machine Model Interface (`Mem_I s sue`)

This interface supplies hooks for creating new machine models. This interface is tied directly to Tango, unlike the interfaces provided by the standard machine models.

The functions that make up this interface include include `Mem_Issue`, `Mem_Init`, `Mem_Sync`, `Mem_Wait`, and `Mem_End`. Also, the `aug-summary-other` and `aug-read-other` functions are available at thislevel.

`Mem_I s sue` is called by Tango at every event where simulation is required. The `-s` im option in the `tango` script may be used to determine which events are simulated and whether the simulator handles data values. `Mem_Init` is called to initialize the memory system, and `Mem_End` is called when the application completes.

```
  Mem_Init()

  Mem Issue(vtime, type, address)
    unsigned int vtime;
    unsigned int type;
    unsigned int address;

  Mem_End()
```

The data value, if any, is passed between Tango and the memory system in an eight-byte buffer called `aug_data`.

`Mem_Wait` should cause the simulator to process the next outstanding memory request. `Mem_Sync` should cause the simulator to schedule other processes until the memory system and all other application processes have simulated beyond time on the current process clock.

```
Mem_Wait();
Mem_Sync();
```

5.4 CPU Timing Parameter File (`aug-init`)

The parameter file `aug-init` is a text file which defines the type and delay associated (for simulation purposes) with local operations. Fixed delays may be specified for each assembler mnemonic on the ***machine being used to run the simulation.*** This file is read during the augmentation phase of the compilation. For each basic block in the application, augmentation adds code to advance the local clock by the sum of the amounts associated the assembler operations in that block. The progression of time on the target machine is based on these delays, so they should approximate the timing of the target machine as closely as possible.

Thedefault `aug_init` file is located in the directory `/usr/local/lib/tango`. To specify a different file, the `AUGINIT` environment variable may be set to the full pathname of the desired parameter file. Alternately, the pathname may be specified with the `-I` option of the AUG program.

Each line describes a single mnemonic, using the following format:

```
<mnemonic> <type> <clicks>
```

For instance:

```
1.d DLOAD 1
```

defines the mnemonic `1. d` to be a double-word load instruction which takes one click.

The valid mnemonic types are:

- `data` reference `types` defined `in reftypes.h (BLOAD, HLOAD, LOAD, DLOAD, BSTORE, HSTORE, STORE,` and `DSTORE)`
- `JUMP` : for unconditional branches
- `CALL` : for branches that return
- `FORK`: for conditional branches
- `OP` : for instructions that do not branch or reference memory
- `DATA`: for data-generating mnemonics
- `LABEL` : for mnemonics that generate branch targets
- ☠⚐☠ : for other mnemonics that do not generate code or data
- `ERR`: for mnemonics that cannot be processed by AUG

5.5 Memory and Synchronization Timing Parameters

Machine models may accept `timing` parameters for the memory and synchronization system; the file `tango--init` is read at run time to determine such parameters.

If the environment variable `TRACEDIR` is defined, a `tangoinit` file in that directory will be used. Otherwise, the `tangoinit` file is assumed to be in the current directory. If the `tango_init` file is not found, a warning is printed, and the timing parameters are left at their default values.

The lines in this file have the following format:

```
<tag> <args>
```

Upper- and lower-case are considered to be distinct within this file.

The default memory model uses the scost tag to specify the (fixed) latency associated with synchronization events and the additional latency for shared references. For instance, the lines:

```
scost LOCK_ENTER 3
scost DLOAD 6
```

indicates that there is a three-click latency to enter a `LOCK` macro and six extra clicks are needed for shared double-word loads. Latencies that are not specified are zero by default.

The user can write a custom version of `aug_read_other` to handle other `tangoinit` tags. See Section 5.3.3.

Chapter 6

Output File Formats

When the augmented application is executed, the simulation will write output files to the directory specified by the environment variable `TRACEDIR`. If this environment variable is not defined, the current directory is used. Note that each user is allowed to run only one simulation at a time on a given machine.

The simulation will create a file named `syncVarTbl`. This file contains information about each synchronization variable used by the application. See Section 6.1 for a description of this tile.

The simulation also writes a summary file for each processor, with the name `summarynnn. dat` where *nnn* is the processor number (starting with 000). This file contains the virtual time when each process completed and a dump of the data reference counters. In addition, custom messages may be added to this file using the function `aug_summary_other` described in Section 5.3.4. See Section 6.2 for a description of this file.

In addition, if logging was enabled with the tango script `-trace` option, the simulation will write an event log file for each processor. These ties are named `t racennn . dat,` where *nnn* is the processor number. The format of the log file is described in Section 6.3.

6.1 Synchronization Varible Listing

The `syncVarTbl` file contains information about each synchronization variable in the application, including its name, address, and type. The lines in this file have the format:

```
<variable index> <hex_address> <group> <type> <measure> <plot> <name>
```

and typically look like this:

```
11  1021d75c   1000    1000       1        1      gm->idlock
```

In this example, the 1 lth synchronization variable in the program was at the hexadecimal address `1021d75c` and was named `gm->idlock`.

6.2 Simulation Summaries

Each summary file contains the time when the process completed and a dump of the data reference counters. If the `-countRef s` option was not specified, reference counters will all be zero.

The summary file has the format:

```
Summary (process <number>):

Final clock reading:   <number>
           byte      half      word   double
reads   <number>  <number> <number> <number>
writes  <number>  <number> <number> <number>
```

Here is a sample summary file:

```
Summary (process 0):

Final clock reading:   16673002
           byte      half      word   double
reads        106         0   7504595        6
writes         0         0   1088337        8
```

Users may add their own output to this file as described in Section 5.3.4.

6.3 Event Logs

When tracing is enabled, the simulation produces one event log file for each process. After the run is completed, **the tmerge** program (Section 9.4) can be used to merge the log files into a single file with the events in chronological order.

The tcat filter converts log files into a human-readable form, as described in Section 9.5. The **tfilt** filter can be used to find selected events in a log file, as described in Section 9.3.

63.1 Event Record Format

The event logs are made up of 12-byte binary records. Each record represents a synchronization event or data reference.

Here `is` a definition of a log `record as` a C `struct,` taken from the header file `trfile.h in /usr-/include/tango:`

```
typedef struct ref {
  int type;     /* including source location and processor number */
  int time;     /* simulation time of the event */
  int address; /* operation address, if applicable */
} ref;
```

Here is a byte-by-byte description (using big-endian conventions):

- Bytes O-l:
  If a data reference, this field is a code that indicates which line of the source code generated the reference. This field is not used for synchronization events.
- Byte 2:
  Processor number, an 8-bit binary integer. In the raw event log files, this is set to zero, but when the log is processed by **tmerge** this field is set to the processor number where the event happened.
- Byte3:
  The type of event, an 8-bit binary code. See the list in Section 6.3.2.

- Bytes 4-7:
  The click (simulated time) in which the event happened. It is a 32-bit integer.
- Bytes 8-11:
  The 32-bit virtual address of the shared data location or synchronization variable.

The filter **tcat** converts event log files into a human-readable form (see Section 9.5). The filter **tfilt** can be used to find selected events in a log file (see Section 9.3).

6.3.2 Event Type Codes

The type codes for synchronization events are defined in the header file `synctypes.h`. The type codes for data references are defined in the header file `reftypes.h`. Both of these header files are found in the directory `/usr/include/tango`. Event codes from 32 to 63 are reserved for user-defined events, which may be generated using the `ST_LOG` macro.

The type codes include:

- `BLOAD`**(0)** a byte load
- `HLOAD`**(1)** a 2-byte load
- `LOAD`**(2)** a 4-byte load
- `DLOAD`**(3)** an 8-byte load
- `BSTORE`**(8)** a byte store
- `HSTORE`**(9)** a 2-byte store
- `STORE`**(10)** a 4-byte store
- `DSTORE`**(11)** an 8-byte store
- `BLOAD+16` *thru* `DSTORE+16`**(16-27)** like **(0-11)** but used with the memory system simulator to indicate the completion of the memory reference, whereas 0-11 indicate the initiation of the reference.
- `ST_THREAD_DONE`**(70)** a process terminates
- `ST_THREAD_START`**(71)** a process starts execution
- `ST_LOCK_ENTER`**(72)** a process attempts to acquire a lock
- `ST_LOCK_EXIT`**(73)** a process acquires a lock
- `ST_UNLOCK`**(74)** a process releases a lock
- `ST_BAR_ENTER`**(75)** a process begins waiting at a barrier
- `ST_BAR_EXIT`**(77)** a process leaves a barrier
- `ST_BAR_EXIT_LAST`**(78)** a special case of (77) for the last process to leave the barrier
- `ST_GS_ENTER`**(79)** a process requests an index for a self-scheduled loop
- `ST_GS_EXIT`**(80)** a process acquires an index for a self-scheduled loop
- `ST_GS_EXIT_LAST`**(81)** a special case of (80) for the process that acquires the last available index
- `ST_EVENT_ENTER`**(94)** a process enters a `EVENT`, `SETPAUSE`, or `CLEARPAUSE` macro
- `ST-EVENT-EXIT(%)` a process leaves a `EVENT`, `SETPAUSE`, or `CLEARPAUSE` macro

- ST-PAUSE-ENTER(%) aprocessenters a PAUSE,WAITPAUSE macro
- ST_PAUSE_EXIT(97) a process leaves a PAUSE or WAITPAUSE macro
- ST_SEND_ENTER(98) a process enters a SEND macro
- ST_SEND_EXIT(99) a process leaves a SEND macro
- ST_RECEIVE_ENTER(100) a process enters a RECEIVE macro
- ST_RECEIVE_EXIT(101) a process leaves a RECEIVE macro
- ST_ALOCK_ENTER(104) a process attempts to acquire a lock from a lock array
- ST_ALOCK_EXIT(105) a process acquires a lock from a lock array
- ST_AULOCK(106) a process releases a lock from a lock array
- ST_WAIT_FOR(107) during a WAIT_FOR_END macro, another application process terminated
- ST_GMALLOC(108) a process requests a block of shared memory
- ST_GFREE(109) a process releases a block of shared memory

Other codes may appear if the application uses undocumented synchronization primitives or the ST_LOG macro (see Section 3). In addition, new codes may be added from time to time; all event log post-processors should degrade gracefully when undefined codes are encountered.

Chapter 7

Sample makefile

In this example, the source code is in two files: main. C and linear. C. Both of these depend on the header file lstruct.H, which contains **m4** macros. Various compilation options are shown as comments.

```
#!/bin/make -f
#   Sample makefile for Tango.
TARGET = ../linear

## Trace global operations; order synchronization only.
## machine model with cached locks and nonuniform memory accesses
## include object library libdir.a
#TFLAGS = -trace G -1 dir -M cachedLocks -sim TIME G

## custom user machine model and memory data simulator provided.
## calls full data memory simulator at every memory reference.
#TFLAGS = -M encore -sim DATA A -tlib ~davis/lib -tmac ~davis/macs

# order sync ops and shared refs; uniform memory machine
TFLAGS = -order G


.SUFFIXES:
.SUFFIXES: .o .S .s .c .C .h .H
.S.o:  ; $(TANGO) -s 1 $*.S
.s.S: ; $(TANGO) -s 1 $*.s
.s.o: ; $(TANGO) -s 2 $*.s
.c.s: ; $(TANGO) -s 1 $*.c
.c.S: ; $(TANGO) -s 2 $*.c
.c.o: ; $(TANGO) -s 3 $*.c
.C.c: ; $(TANGO) -s 1 $*.C
.C.S: ; $(TANGO) -s 2 $*.C
.C.s: ; $(TANGO) -s 3 $*.C
.C.o: ; $(TANGO) -s 4 $*.C
.H.h: ; $(TANGO) -s 1 $*.H

TANGO = tango $(TFLAGS)

# here begins the application-specific part of the makefile

OBJS = lu.o matio.o
```

```
install $(TARGET): $(OBJS)
$(TANGO) -o $(TARGET) $(OBJS)

lu.s matio.s: matrix.h
```

Chapter 8

Common Compilation and Execution Problems

This section covers problems you may encounter in compiling and running code in the Tango environment or in porting to System V from the Encore or other system.

8.1 Compiler Errors

The header file `strings.h` does not exist under System V. Many programs use this file to declare string functions such as `strcpy` and `strcat`. In this case, you will have to add the declarations for these functions to the code.

8.2 Compiler Warnings

The MIPS C preprocessor (**cpp**) will warn you when you re-`#define` a **cpp** macro. In particular, `NULL` is defined in the header file `stdio.h` and `M_PI` is defined in `math.h` under System V, but these are often redefined by applications written on other systems. If this bothers you, put `#ifdef's` around the extra definition.

Also, you may get a warning if the application uses the value returned by `sprintf`, since the return value is an `int` in System V and a `(char *)` pointer in other systems.

8.3 Syntax Errors

m4 macroes must not have any white space between the macro name and the arguments. In the following example, there is a blank between the `MAIN-INITENV` macro and its arguments. Therefore, the arguments are treated as normal C code, causing a syntax error during compilation.

```
main () {
  MAIN-INITENV (25,10000)
  printf ("Hello, world!\n");
  MAIN_END
```

Normally, Tango macros expand into C statements, that is, they do not expect the user to place a semicolon after the macro. In the following code, there is an extra semicolon after the `CLOCK` macro, so the compiler cannot associate the `else` clause with the `if`, causing a syntax error.

```
sub (i) {
  if (i == 0)
    CLOCK(i);
  else
    i = 0;
}
```

The `G_MALLOC` macro has a semicolon at the end This means that, like the other **m4** macros, its return value cannot be used in an expression. There is a special version of `G_MALLOC` called `G_MALLOC_F` which can be used as a function. (`G_MALLOC_F` is the only Tango macro that needs a semicolon; all the others are complete statements.)

Thus:

```
if ((glob = G_MALLOC(10)) == 0) {
```

should be rewritten as:

```
glob = G_MALLOC(10)
if (glob == 0) {
```

or:

```
if ((glob = G_MALLOC_F(10)) == 0) {
```

8.4 Undefined Symbols in Link

If there are undefined **m4** macros in your program, m4 and cc may not detect the error, treating the macro as a call to an (undefined) external function.

Some run-time library routines that are available on Ultrix systems do not exist under System V. Usually there is another routine (often with a different argument list or result type) that can be used to achieve the same effect. Compare the **man** pages for details For instance,

1. for `bcopy` use `memcpy`
2. for `bzero` use `memset`
3. for `srandom` use `srand`
4. for `random` use `rand` or `Drand48` and see also Section 8.10.
5. for `index` use `strchr`

Similarly, the `timer_init` function is Encore-specific; use the m4 `CLOCK` macro instead.

You might also get this error if you need an augmented object library in the link. For instance, if the symbol `printf_aug` is undefined, then the object library `libcaug.a` is needed in the link.

8.5 Semaphore Failures at Run-time

The message "Can't allocate *n* semaphores" indicates that the simulation is unable to allocate the semaphores it needs because of a system-imposed limit on the size of a semaphore request. It will reduce the size of its request until it either succeeds or reduces its request to a single semaphore.

In the context of a semaphore error, he message "No Space Left On Device" indicates that the simulation is unable to allocate the semaphores it needs because of a system-imposed limit on the number of semaphores. You may get this error if other simulations are running on your machine, or because of simulations which failed to deallocate their semaphores.[1]

Messages such as:

```
Identifier removed
36 Block failed.

Invalid argument
22 Block failed.

Identifier removed
36 Release failed.
```

and:

```
Invalid argument
22 Release failed.
```

indicate that semaphores used by the simulation were removed while they were still in use. One common cause for this error is failure to execute the MAIN_END macro at the end of an application.

8.6 Shared Memory Failures at Run-time

The message "Shared memory segments not contiguous" means that the simulation allocated two shared memory segments during the initialization of the Tango environment, but they were not contiguous. This will happen on systems where the maximum shared memory segment size is less than the segment alignment multiple. In this case, Tango's shared memory allocation will be reduced

The error message "Shared Memory Exhausted" means that the simulation needed more shared memory than was allocated during initialization. The error message will describe the details of the situation. The default amount is currently 2.5 megabytes. You can override the default by specifying the number of bytes as the second argument to the MAIN_INITENV macro. For instance:

```
MAIN_INITENV(,5000000)
```

will allocate 5 megabytes of shared memory.

8.7 "Too Many Locks to Trace"

This message indicates that Tango ran out of table space while creating the table of lock addresses for the **SyncVarTbl file.**

[1]The rmids program (Section 9.2) can be used to find stray semaphore and shared memory identifiers in the system. It notes all such identifiers and attempts to remove them.

The limit is set by the macro ST-MAX-LOCKS in the macro libraries. You can make your own version of the macro library with an increased limit or ask that the limit be raised in the public version. The limit is currently set to 20000 locks.

8.8 Output File Develops an Echo

When a process forks, each child gets a copy of its file buffers. Later, all the copies of the buffer are flushed, writing multiple copies of the buffer contents to the file. To avoid this problem, flush all output file buffers immediately before any CREATE macro.

8.9 Simulation Loops Indefinitely

Remember that the simulation will be significantly slower than the original application-it may just be taking a long time! When experimenting with Tango, you should run a small application at first to get a feel for the expected simulation time.

Some applications intentionally loop on shared memory locations for synchronization purposes. For example:

```
if (waiter)
  while (glob->Q == 0);
else
  glob->Q = 1;
```

Such code will loop indefinitely if the simulation reschedules processes only at sychronization events. You can cause the simulation to reschedule on shared data references by using the -order G option in the tango script.

Alternatively, you could modify the application. You can force Tango to reschedule at any point by adding a call tothefundion Sc_Reschedule:

```
if (waiter)
  while (glob->Q == 0) {
    AUG_OFF
    Sc_Reschedule();
    AUG ON

else {
  AUG_OFF
  Sc_Reschedule();
  AUG_ON
  glob->Q = 1;
```

Note that the correct way to use SC-Reschedule is to place calls *before* each read or write to shared memory that is not protected by a lock In particular, rescheduling *after* a write will *not* produce the desired effect.

8.10 Other Pitfalls

The rand function returns a value in the range 0 to 32767 under System V, while under Ultrix and on many other systems, it returns a value in the range 0 to 214748347.

If the application was written for a small multiprocessor, it may be a hard-coded limit (such as 4 or 16) on the number of processors. If the limit is in the main source file, you may want to replace it with `MAX-PROC`, which is an m4 macro set by `MAIN_ENV`. For example, if the application source contains:

```
#define maximum-number-of-processors 16
```

you could replace it with:

```
#define maximum-number-of-processors MAX-PROC
```

This will not work if the module does not contain `MAIN_ENV`.

Make sure that the first and last executable statements in the application source code are `MAIN_INITENV` and `MAIN-END`, respectively. This is more vital in Tango than it is in other environments, and many applications written for those environments omit `MAIN_END` entirely. If `MAIN-END` is omitted, the `syncVarTbl` file and the main summary file will not be written, and the semaphore and shared memory identifiers will not be recycled.

Finally, refer to Chapter 9 for information on many of the messages generated by **AUG, rmids**, and other Tango programs.

Chapter 9

Programs Associated with Tango

For each program, a synopsis of the arguments is given, followed by a description of the program and a list of important messages with explanations.

9.1 AUG Program

Synopsis:
`AUG [-a] [-am] [-ar] [-au] [-b] [-Bn] [-c] [-f] [-fm] [-fr] [-fu] [-i] [-I`*file*`] [-l] I-ml [-nm] [-nr] [-nu] [-r] [-s] [-Tn] [-u] [-U] [-v]`

This program reads assembler code from the standard input and writes augmented assembler code to the standard output. The executable for this program is maintained in the directory `/usr/local/bin` and the source is in `/usr/local/src/AUG`.

Switches in the **AUG** command line include:

-a Do not augment absolute data references. This is an optimization for simulations which check for shared data references, since shared data is always addressed indirectly, never as an absolute address. This option is equivalent to the combination of `-am`, `-ar`, and `-au`. Compare with the `-f` option.

-am This disables event logging augmentation of absolute data references.

`-ar` This disables reschedule augmentation of absolute data references.

`-au` This disables memory simulation augmentation on absolute data references.

`-b` Add the source file name to the back-reference file `back-ref s`, which lists the source files of the application. This file is useful for determining which source lines that generate particular data references. This option is recommended.

`-Bn` Don't instrument basic blocks containing less than ***n clicks*** of local operations. This might be used to speed up simulations (at the cost of some accuracy).

`-c` Generate code to count shared data references ***and*** simulation-time clicks. The totals for each process are written to the summary files (see Section 6.2). The default is to count clicks only.

`-f` Do not augment stack frame references. This is an optimization for simulations which check for shared data references, since the stack frame should never be shared This option is equivalent to the combination of the `-fm`, `-f r`, and `-f u` options. Compare with the `-a` option.

`-fm` This disables event logging augmentation of stack frame references.

`-fr` This disables reschedule augmentation of stack frame references.

`-fu` This disables memory simulation augmentation on stack frame references.

`-i` Augment subroutine calls so that the augmented versions of the run-time routines will be substituted for the normal ones. If this option is not used, each run-time routine will be treated as a single instruction and shared data references within run-time routines will not be detected.

`-I`*file* Read click counts for each assembler mnemonic from *file*. This option overrides the pathname implied by theenvironmentvariable `AUGINIT`. The default file is `aug_init` in the directory `/usr/local/lib-/tango`, which sets every executable mnemonic to one click. Using this option, one may customize a simulation to use different click counts for different instructions. A description of this file format is found in Section 5.4.

`-1` Augment data load operations. This option is modified by the `-f` (or `-fm`) and -a (or -am) options. See also the `-s` option.

`-m` Augment shared data references. This option is equivalent to the combination of the `-1` and `-s` options. This option is modified by the `-fm`, `-nm` and -am options (and also the `-f`, `-n`, and `-a` options).

`-nm` This causes event logging augmentation to bypass range checking on all references. Normally, the logging routine checks whether the reference address is in the range of shared addresses that was determined at initialization, and references outside that range ate ignored. This option is used to allow logging of *all* data references, shared and unshared.

`-nr` This causes rescheduling augmentation to bypass range checking on all references. Normally, the rescheduling routine checks whether the reference address is in the range of shared addresses that was determined at initialization, and references outside that range are ignored, This option is used to allow rescheduling on *all data* references, shared and unshared.

`-nu` This causes memory simulator augmentation to bypass range checking on all references. Normally, the memory simulation routine checks whether the reference address is in the range of shared addresses that was determined at initialization, and references outside that range are ignored. This option is used to *allow* memory simulation of *all* data references, shared and unshared.

`-r` Reschedule processes at loads and stores to preserve the order of data references. This option is modified by the `-fr`, `-nr` and `-ar` options (and also the `-f`, `-n`, and `-a` options).

`-s` Augment data store operations. This option is modified by the `-f` (or `-fm`) and `-a` (or -am) options. See also the `-1` option.

`-T`*n* Generate code to reschedule processes approximately every *n clicks.* If the shared data in the application is protected by locks or if the `-r` option is used, this should not be necessary.

`-u` Pass references to the memory simulator at shared data loads and stores. The simulator should update the simulation time. This option is modified by the `-fu`, `-nu` and `-au` options (and also the `-f`, `-n`, and `-a` options).

`-U` Pass references to the memory simulator at shared data loads and stores. Use timing results and data from the simulator. This option is modified by the `-fu`, `-nu` and `-au` options (and also the `-f`, `-n`, and `-a` options).

`-v` Run **AUG** in verbose mode, informing the user of its progress as it runs. The default is to run silently.

Error messages include:

- `error opening "`*file*`" for input`
  The named *file* does not exist or is not readable.

- `error reading "`*file*`"`
  The named *file* is incorrectly formatted.

- `invalid type "`*word*`" in initialization file`
  The initialization file specifies the type ***word*** which is not one of the valid types described in Section 5.4.

- `unknown mnemonic` ***"word"***
  The assembler mnemonic ***word*** appears in the input, but is not defined in the initialization file.

- `statement cannot be processed`
  The input file contained a statement that could not be augmented. Such statements should be rare in compiler-generated assembly files.

- `too many source files for backreferencing`
  The application uses more than 31 source files, so back-referencing is not possible.

Warning messages include:

- `unknown command-line argument ignored`
  An invalid option appeared in the AUG command line.

- `noreorder prevented block augmentation`
  The input file contained the `noreorder` directive, meaning that code-rearrangement is forbidden. Therefore, parts of the code were not augmented as requested.

9.2 rmids Program

Synopsis: `rmids [-q]`

rmids attempts to find all the shared-memory and semaphore identifiers in the system and delete them. If you are not super-user, this will remove your own identifiers but not ***those*** of other ***users. If you are super-user, this will*** remove ***all*** identifiers in the system, including those belonging to other users. The executable for this program is maintained in the directory `/usr/local/bin` and the source is in `/usr/local/src/rmids`.

If the `-q` option is specified, messages will only be printed when identifiers are actually deleted.

Messages include:

- `argument ignored`
  indicating that an argument other than `-q` appeared on the command line.

- `Searching for shared memory identifiers` and
  `Searching for semaphore identifiers`
  indicating normal progress of the program

- `shared memory identifier` ***#n*** `belongs to another user` and
  `semaphore identifier` ***#n*** `belongs to another user`
  indicating that the identifier could not be removed because you do not have permission to do so

- `error removing shared memory identifier` ***#n*** and
  `error removing semaphore identifier` ***#n***
  indicating that the identifier could not be removed for some reason other than lack of permission

- `Shared memory identifier` ***#n*** `was REMOVED` and
  `Semaphore identifier` ***#n*** `was REMOVED`
  indicating successful removal of identifiers

9.3 tfilt Program

Synopsis: `tfilt` [*file*] [*-a hexaddr hexaddr*] [`-c` ***code***] [`-p` ***processor***] [`-s` *filenum linenum*] [`-t` ***time1 time2***]

This program reads an event log from *file* and writes a filtered log to the standard output. The executable for `this` program is maintained in the directory `/usr/local/bin` and the source is in `/usr/local/src-/tfilt`.

If no *file* is specified, it reads the input log from the standard input. If the `-a` option is specified, it selects events with the address field in the range of hex addresses specified If the `-p` option is specified, it selects events on the ***specified processor*** (the first processor is numbered "0"). If the `-t` option is specified, it selects events between the two times indicated. If the `-s` option is specified, it selects events whose source file and line number match those specified. If the `-c` option is specified, it selects events whose type ***code*** matches that specified.

If multiple options are specified, **tfilt finds** the events which match ***all*** of the specified conditions.

9.4 tmerge Program

Synopsis: `tmerge` *file* [*file* `. ..`]

This program takes a list of event log files and merges them together into a single file, which is then written to the standard output. It uses the time-stamp on each event in the log files to put the events in chronological order. It also sets the processor number field in each event record to indicate from which log file the event came. This program assumes that the individual log files are already chronologically ordered.

The executable for `this` program is maintained `in the directory /usr/local/bin` and the source is in `/usr/local/src/tmerge`.

If you supply no arguments, **tmerge** prints a usage summary.

Error messages include:

`unable to open input file "`*file*`"` -The named *file* does not exist or is not readable.

9.5 tcat Program

Synopsis: `tcat` [*file*]

This program reads a event log file and writes a human-readable version to the standard output. If no *file* is specified, it reads the event log from the standard input. It `also uses the back_refs` file generated by `AUG`, if present. It writes one line of output for every log record

The executable for **this** program `is` maintained `in the` **directory** `/usr/local/bin` and the source is in `/usr/local/src/tcat`.

9.5.1 Sample tcat Output

A full **tcat** output line consists of:

```
<eventtype> at <time>: <hexaddress> on <proc#> at line <line#> in <file>
```

Here is an example:

```
STORE  at       2298: 1021d6fc on   0 at line  558 in "mainNew.c"
```

Synchronization events do not contain source references. Also, the line number may be unknown (if it was greater than 2047) or the **file** may be specified by a number (if no back,refs file was found by **tcat**).

Chapter 10

Acknowledgements

This work was supported in part by the Fannie and John Hertz Foundation by through graduate fellowships for Stephen R. Goldschmidt and Helen Davis. It was also supported in part by DARPA contract N00014-87-K-0828. This support is gratefully acknowledged.

Bibliography

[1] H. Davis and S. Goldschmidt. Tango: A multiprocessor simulation and tracing system. Technical Report (to appear), Stanford University, 1990.

[2] D. Lenowski, J. Laudon, K. Gharachorloo, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. Stanford dash multiprocessor. Technical Report CSL-m-89-403, ***Stanford*** University, 1989.

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