High Level Tools for the Debugging of Real-Time Multiprocessor Systems.

M. Timmerman, F. Gielen, P. Lambrix.
Dept. of Computing Science
Royal Miltary Academy
Renaissance Av. 30
B-1040 Brussels
Belgium
E-mail: Frank.Gielen@rma-brussels.rtt.be

Abstract

Todays' real-time systems become ever more complex and multiprocessor hardware starts pervading many types of equipment. This complexity indicates that real-time programming has matured into a true software engineering discipline which demands the appropriate set of tools for the support of the entire system lifecycle. Since studies show that debugging takes about half of the development time it is obvious that powerful debuggers are needed. The characteristics of real-time systems impose specific requirements on the debugger. The system must be capable of detecting logical as well as timing errors and the invasive nature of the debugging system must not alter the temporal logic of the application. We describe the design and the implementation of a debugger for a tightly coupled multiprocessor real-time system which complies with those specific requirements This paper emphasizes on the high level tools which are necessary for the analysis and the interpretation of the trace data which has been collected at runtime. Those tools are necessary due to the high amounts of trace data being collected.

1 Introduction

A debugging session on a real-time multiprocessor system is characterized by large volumes of data which have to be analyzed. Classical debuggers mainly display the data and leave the job of data analysis to the software engineer who has to build a mental model of the system behavior. The job of converting the raw textual data into information upon which debugging hypotheses can be made, is performed by the programmer
This observation has resulted in an effort to create a distributed debugging system for Unix based workstations which captures and analyses the behavior and the errors on a real-time shared memory multiprocessor target system. The host system has two modules for the interpretation and analysis of the data: a graphical visualizer for the representation of dynamic information about tasks and processors and a rule based query system which draws inferences about the real-time behavior of the target system.
The real-time debugging process is a two phase process. During the first phase the target system is monitored and the execution history of the system is recorded. In this article we will emphasize the second phase which covers the analysis and the interpretation of the data.
Our approach is to consider debugging as a database application. The debugger uses a relational database management system with a query language that is extended with temporal relations.

2 The DARTS project.

2.1 Application Domain.

Our system, "DARTS", which is an acronym for *D*ebug *A*ssistant for *R*eal-*T*ime *S*ystems, is designed for real-time systems which are developped in a host-target environment. The host system is a Unix Workstation and the targets are single board computers on a VMEbus. The workstation and the targets are connected via ethernet.

2.2 Data Collection.

The debug data on the multiprocessor target is collected as event traces. Therefore we had to provide hooks in the runtime system for software probe procedures that log system level events into a trace buffer. At present four classes of events can be selected for logging: scheduler resynchronization, system calls, interrupt activity and watchpoints on shared resources. Once a buffer is filled it is transmitted to the host system while the target system continues with another buffer. The design and the implementation of the event trace system were the subject of previous publications [1,2].

2.3 The Visualizer.

The Visualizer is the front end module on the host system which interacts with the user and provides the following functions.
Session Control enables the user to establish a connection with the target system and to control the operation of the debugger server on the target system.
The *Monitor* gives a graphical representation of the activities on the target system. This includes scheduler resynchronization and multiprocessor load balancing operations per processor, processor selection and activation per task and the occurrence of system events. It offers the possibility to visualize the system activities in a suspicious region which has been identified with the query system.
The *Filter* options allow the user to refine his hypothesis set about the real-time bug by incrementing the level of detail of the collected data and by increasing the selectivity of the trace options.
The *Data Recorder* controls the input stream to the host system. The visualizer can acquire data in real time from the target system or it can replay a previously recorded session.

3 The Rule Based Query System.

3.1 Temporal Logic.

As a temporal framework for the reasoning about the dynamic behavior of real-time systems, we use a temporal logic. Temporal logics allow us to reason about the development of a world in time, and enable us to discuss how situations change due to the passage of time. The runtime behavior of a real-time system can be seen as a sequence of states that undergo transformations determined by the program's instructions and by external stimuli. We have selected a temporal framework based on predicate calculus. Properties of the world are extended with a supplementary parameter representing time.

3.2 Implementation.

We have chosen Prolog as the implementation language Prolog is a programming language based on predicate calculus. It provides the adequate mechanisms to represent the time-extended predicates as well as to declare the temporal relations between time intervals (such as before, after, during ...) [3]. Using Prolog we are also able to model knowledge from different sources in a uniform way. For instance, data flow diagrams from the design phase as well as the event stream captured at runtime can be described with Prolog programs. Moreover, Prolog can be used as a query language and as a verification tool by the user. Our system parses the raw event stream from the target into a set of Prolog facts.

3.3 Source Code Tracking.

During a debugging session the programmer uses hypotheses about the source code and the system design and receives event traces from the target system. The event traces do not contain any symbolic information. Since the programmer uses this information for the modification of his set of hypotheses, symbolic analysis must be an integral part of the debugger. In non real-time systems we can direct the compiler to produce additional symbolic information for the purpose of source level debugging. This solution is not acceptable for real-time applications since in that case we affect the code generation of the compiler. Most available compilers can only produce symbolic information if they do not optimize the code. It is clear that this can have significant influence on the temporal logic of the real-time system. We have tried to build a system that allows source level debugging without altering the executable code.
The system is based on control flow analysis where we try to match the sequences of system events which occur at runtime with sequences of system events inferred from the source code. The main idea is to extract, on the basis of the source code and the semantics of the runtime system, all the information concerning the sequences of system events that may be generated.
In a first step the real-time program is analyzed in two stages: a language dependent front-end and a back-end generator of Prolog clauses. The front-end can be considered as part of the compilation system. It parses the source code and gathers all the information which is useful for the source code control flow analysis. This includes information about system events and information about control statements. This information is then transformed into a list of Prolog terms of the form :

$$sequence(SC, SL).$$

The SC is a source context, while SL is a source list.
The basic unit for control flow analysis is a task. The analyzer first isolates the tasks within the source code module and starts the analysis at the task level.
The source context SC is a Prolog list denoting a particular sequential piece of code. For instance, [if1.2,loop2,task1] is the source context of the second branch of the first if-statement within the second while-statement of task 'task1'. The control flow statements are labeled by the code analyzer in order to obtain an explicit context name. The if-statement receives the selection context label 'if1' and the while loop is labeled as iteration 'loop2'.
The source context of a task changes upon entering the body of a subprogram, a branch of a selection and the body of an iteration.
The source list SL is a Prolog list which represents for a particular source context the sequential list of system events and control flow statements in that source context. The elements in a source list may point to sub-contexts, who in their turn have source lists. These elements are predicates of the following types.

- *service calls (svc).* These denote the parts in the source code which explicitly perform straight-line system calls.
- *selections (select).* Selects denote the composite source contexts where one out of several possibilities can be chosen (if, case). They

point to sub-contexts containing the branches of the selection

- *iterations (loop).* Parts of the source which can be executed a number of times, depending on a boolean condition are denoted by iterations As with the selects, the loops point to sub-contexts, which are the bodies of the iterations

- *subprogram calls (user-call).* A user subprogram corresponds to a function or procedure call at the application level. The call itself does not invoke the runtime environment The user-call points to a context which is the body of the called subprogram

- *runtime (library-call).* We also have calls to functions or procedures which are language libraries or syntactic language constructs that unwind in a series of system calls for their implementation. Library-calls point to tables containing explicit knowledge about the runtime executive.

Using pointers to sub-contexts we obtain a hierarchy of source contexts Unfolding the source list of a particular source context in the proper way gives us all the possibilities of sequences of service calls generated by the code of that source context. The code for an individual task *task*1 is represented by SL which satisfies *sequence*(*task*1, SL) and the possible expansions of the elements in SL.

For each service call we store the symbolic parameter information, together with the return value parameter and the line number information For each other element in a source list we keep also the line number information.

Having gathered this information, we match it with an event trace. During the second phase we process the sequence of event traces received from the target system We classify the service calls of the event traces per task and find the corresponding source code. Per task the source code and the event traces can be matched at the service call level. We use the following rules·

- A service call in a source list matches a service call in the trace, if they have the same name and if the values of the (command dependent) relevant parameters are the same
However we allow for the possibility that not all parameter values are available from the trace

- A selection in a source list matches a part of a trace if one of the branches matches that part of the trace

- An iteration in a source list matches a part of a trace if the body of the iteration matches 0 or more times that part of the trace
(In our implementation we try to find the maximum number of iterations first.)

- A user-call in a source list matches a part of a trace if the subprogram body matches that part of the trace.
(In our implementation we do not yet allow recursive functions which contain service calls.)

- For runtime contexts we use a table-driven approach For each library function we have listed the different possibilities of sequences of service calls in a table such that we can prefer the occurrence of specific sequences:
a library-call in a source list matches a part of a trace if one of the preprocessed table sequences for that call matches that part of the trace.

As a result of the matching we have per task a trace/source sequence of service calls with for each service call the place of occurrence in the source code, the parameter-value pairs (with 'not available' for the values for which the trace does not provide any information), the time stamp and the processor on which the service call was performed

In most cases a unique relation between the source and the event stream is found. In some cases several possibilities remain. The user must add this to his set of hypotheses and use other means to narrow the suspicious region.

In our implementation we included an extension to cope with the problem of incomplete information at the boundaries of a trace. We allow to start the matching at a specific line in the source code (for incomplete information at the beginning of a task) and to stop the matching at a specific place in the trace (for never-ending tasks, such as cyclic tasks).

In the last step the original Prolog fact base is replaced with a new fact base which contains symbolic information and line numbers The user can now query this fact base at a high level Possible queries are for instance, to find all or the nth system event in a particular task, or to find all instantiations of the system event on a specific line in the source, or to find the corresponding create-start pairs for a particular task.

3.4 Temporal Distorsion.

The debugger is only useful when the temporal distorsion introduced by the event trace functions does not alter the schedulability of the set of tasks of the applications A set of tasks which is schedulable before the insertion of trace functions must remain schedulable with event trace functions. Although only a few real-time languages make sufficient provisions for schedulability analysis, the analyzer offers the possiblility to determine the worst case overhead per task which is introduced by every trace function As the event trace functions are written in assembler, the instruction execution time can be computed in a table-driver fashion as a function of the opcode and the addressing mode and the processor rate of the application. This method gives us the worst case time distortion per task In order to have an estimation about the total temporal distortion per task the individual distorsions must be added to the distortion of other kernel level trace functions: scheduler trace functions, interrupt trace functions and variable trace functions. With this information we can evaluate the inference of the trace functions with the real-time requirements of the system.

4 Feasibility Demonstration.

4.1 Real-Time remote procedure calls.

The application which we will use for the feasibility demonstration is a multiprocessor system with hard real-time as well as soft real-time deadlines which uses remote procedure calls between processors for the scheduling of work requests. We emphasize on the development of an interprocessor communication protocol in support of those real-time remote procedure calls. The case study covers the implementation of the transmitter. The system is temporal fault tolerant, which implies that it has to be able to recover from a missed deadline

One of the processors is set up as a master front-end processor for handling all interactions with the external world The management of all peripheral hardware is the responsibility of the front-end processor and all possible interrupts have to be redirected to it. The other processors in the system are reserved resources for the execution of application code.

Upon the arrival of an external event the global scheduler, executing on the front-end processors, sends a work request to any of the available application processors where some work has to be performed The work request is characterized by a hard-real time deadline. If the deadline cannot be met the system exhibits a mode change and continue to operate in a degraded soft real-time mode until it regains the ability to guarantee hard real-time deadlines. The remote procedure calls are to be categorized as real-time remote procedure calls (RT-RPC) because the timeliness of the client-server action has to be guaranteed.

Using the RT-RCP the front-end processor, which is the RPC client, defines the deadline. This specifies how long the front-end processor is willing to wait for the result.

In the hard real-time mode the RT-RPC fails if the application processor cannot provide a reply before the deadline or, in case of the soft real-time mode, before a certain time after the deadline

In the RT-RPC protocol which we use for this study, the communicating entities exchange a request, an acknowledgement and a result message Figure 1 shows an example of the message flow in a two processor system The front-end processor sends out a request message with two deadlines, D_a and D_f. One of the servers tries to send out an acknowledgement to the front-end before the acknowledgement deadline, D_a.

If the server can handle the request it tries to deliver the request before the result deadline D_f.

The front-end only continues in hard real-time mode if the acknowledgement from the server is received before the first deadline. The server in his turn only sends out the acknowledgement if both deadlines can be met. In all other cases the hard real-time operation mode fails and the front-end processor shifts the entire system to the soft real-time fall back mode

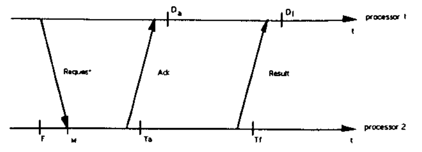

Figure 1· RT-RPC messages

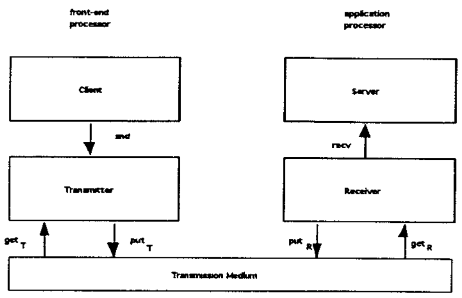

Figure 2: The layered model for RT-RPC.

In the soft real-time mode the front-end processor waits for an acknowledgement even if it arrives after the first deadline and it waits for the result if the acknowledgement is received before the second deadline. Only in the case where no acknowledgement is received before the second deadline, the front-end processor reports failure We concentrate on the communication protocol layers that allow a smooth shift between the hard and the soft real-time operation modes. The Stenning protocol which is the subject of the next section is very well suited for this type of problems

4.2 The Stenning Protocol.

Figure 2 depicts a typical situation for protocol specification: two protocol entities communicate via a medium in order to provide some service. The service provided by the protocol entities will be referred to as the upper layer service, the service of the underlying medium is the lower layer service The Stenning protocol works as follows (figure 2)

At the transmitter each data unit from the upper layer, Upper Data Unit or UDU, is associated with a unique sequence number The UDU is sent repeatedly to the medium until an acknowledgement with

the corresponding sequence number arrives. At the reciever end the UDUs are delivered to the upper layer service requester in the order of the sequence number.
For each data unit received from the medium, Low Data Unit or LDU, the receiver sends back an acknowledgement with the corresponding sequence number.

The Stenning protocol can be used to support the operation mode shift of the RT-RPC service in the following way. The retransmission period for the UDU over the medium equals the first deadline of the system. If an acknowledgement arrives before the expiration of this period, the system continues to operate in hard real-time mode and no retransmission of the UDU will occur. If on the other hand the acknowledgement does not arrive before the acknowledgement deadline then the transmitter continuously outputs the request again until the second deadline expires.
The use of the formal design method FOCUS is applied to the Stenning communication protocol [4]. In the FOCUS trace formalism actions are used to model the service primitives. The specifications are predicates on the actions and yield a set of possible action trace sequences.
After the stepwise refinement the specification of the transmitter contains the following real-time predicate: p an q denote sequences of actions of the system and k is a sequence number.

$$\begin{aligned}&\forall p \in Act^* : p \circ put_T(k,d) \Rightarrow\\ &\exists q \in Act^* : p \circ put_T(k,d) \circ q \circ get_T(k,d)\\ &\qquad \text{and } \#\surd\text{©}q \leq N\end{aligned}$$

The hard real-time predicate specifies that if the transmitter puts an LDU to the medium (put_T (k,d)) then at the most N units of time may have elapsed before the acknowledgement has arrived (get_T (k,d)). This is the formalization of the hard real-time requirement of the real-time remote procedure calls.
Predicate calculus propositions can be translated into a clausal form which closely resembles Prolog clauses. We use this property and present the predicate in an appropriate format for our purpose.

$$\begin{aligned}&getT(Data, K, T1) :-\\ &\qquad putT(Data, K, T2), T1 > T2.\\ \\ &max\_duration(K, N) :-\\ &\qquad putT(\_, K, T1), getT(\_\, K, T2), T2 < T1 + N.\end{aligned}$$

The first clause indicates that the occurrence of an acknowledgement is implied by the fact that a packet has been sent out to the medium.
The second clause states that the acknowledgement occurs within a predefined timeframe.

4.3 Implementation.

The Stenning protocol has been implemented on a shared memory system with two processors: the front-end processor and a second processor that handles the work request. The program is written in C and uses the services of MTOS-UX for the communication between the processors.
During the execution of the program, the software probes for the scheduler monitor and the system call monitor are active. The debugging system verifies the system's execution history against the real-time constraints. In order to do this, a number of translations are necessary.
The first translation converts the raw event stream into a number of Prolog fact bases. Figure 3 shows part of the fact base for the system calls after the translation. Every entry of the fact base has the following relation scheme:

svc(SystemCall,CallingTask,
ArgList,TimeStamp,ProcessorId).

The major inconvenience with this representation is that the arguments are still in machine format and that the events are not related to the source code. Therefore we compare the event stream with the output of the source code parser and add line numbers, context information and symbolic information. This produces a new fact base with cm facts (cm stands for context match). The Prolog relation scheme is given by:

cm(TaskContext,(LineNumber,SymbolicTrace)).

The *SymbolicTrace* is an updated version of the svc facts with line numbers and symbolic information. *CallingTask* has been replaced by a more complete *CallingContext* . This is shown in figure 4 which contains some of the corresponding cm-facts for the svc-facts of figure 3.
The interpretation of this information is as follows. Event 1 matches the MTOS-UX sndmbx system call in task *clnt* at line 126. It can be found inside a loop which has been assigned the label *loop4* by the source code analyzer. The list of arguments is presented as a set of couples: the first element is the symbol name in the source code, the second element holds the runtime value. The litteral *na* in the second field means that the runtime value is not available. Certain events have no line number information. This means that the event trace has not been generated directly by a statement in the source code but that it was called indirectly from a library module. In this case the context information points to the part of the code which called the library function. For the second event this was a *malloc* call inside the subprogram *add_to_queue*.

For the next translation step we analyze the source code and write the rules that transform a sequence of source code events into a sequence of higher level actions. The rules are implemented in Prolog and we apply them to the cm-fact base. We now obtain a set of action lists, which match the abstraction

level of the specification phase.
If the action alphabet is the same as the action alphabet of the trace specification formalism, we are able to verify the specification predicates on those action lists. Figure 5 shows how it was possible to detect the packets which needed retransmission during a certain interval, using this method According to the specifications this implies that the system has missed a hard real-time deadline at that moment
As a matter of fact we observe that for all elements of the list the acknowledgement time is less than 200 milliseconds (Da), while request 4 only receives an acknowledgement after 379 milliseconds.

4.4 Latency Error.

The first example shows how the combination of the graphical system and the Prolog fact bases enables the software engineer to detect a latency error in the communication system. The error condition can be summarized as follows.
The transmitter task sends a data packet to processor two. Although processor two is in a wait state it takes 60 milliseconds before the receiver task starts processing the packet (figure 6). Moreover, if we examine the same situation at other moments we see that the latency delay is arbitrary but never higher than 100 milliseconds. In some cases this phenomenon even causes the transmitter to miss the acknowledgement deadline and trigger a mode change, even though the second processor was theoretically capable of doing the work request.

The origin of this problem is the implementation of the wait state in the scheduler of MTOS-UX. If a processor has no work to do, it halts itself and waits for the next clock tick.
In order to localize this bug both the graphical system and the query system were necessary. The query system was asked to give the instants of all putT actions. With this timing information we can scroll and zoom into the appropriate time interval on the graphical system. At that moment, the visualization of the tasking activity on the processors shows us that there exists an abnormal delay between the end of the transmitter (xmtr) and the start of the receiver (rcvr).

4.5 Deadline Miss.

The second example demonstrates how the Prolog system can be used to detect the messages which have not been acknowledged within a predefined time frame. It is direct application of the verification of the hard real-time predicate of section 5.2.
From the moment that we obtain the action list with the appropriate sequence numbers, the verification of the hard real-time predicate is straightforward. We use the following Prolog procedure:

max_duration(Number, Da):-putT(Number,T1),
getT(Number,T2), T2 < T1 + Da.

This Prolog procedure is almost identical to the clausal representation of the real-time predicate in section 5.2 The Prolog procedure can be used to generate all the sequence numbers of actions that satisfy the predicate or it can verify an action with a particular sequence number

5 Conclusion.

We have presented a debugging system that captures the dynamic behavior of a system as an event stream.
The uniform representation of the different sources of knowledge about the real-time system (event stream, design, source code and human expertise, ...) allows us to reason about the faulty behavior of the system with a hypothesis set which integrates elements from the different phases of the life cycle.
This work is an improvement over existing debugging systems because it carries an explicit notion of time and because the data collection in the targets has a limited and deterministic influence on the real-time behavior of the system.

References

[1] TIMMERMAN M., GIELEN F., *The Design of DARTS: A Dynamic Debugger for Multiprocessor Real Time Applications.*, Proceedings of the 3th EuroMicro Workshop on Real-Time Systems, IEEE CS Press, 1991.

[2] GIELEN F., *A debugger for multiprocessor real-time applications*, Ph. D. thesis, Free University of Brussels, forthcoming, 1993.

[3] LE DOUX C.H., PARKER D.S. Jr., *Saving Traces for Ada Debugging.*, Ada in Use, Proceedings of the Ada International Conference, Cambridge University press, 1985.

[4] DENDAFER C., WEBER R., *Development and Implementation of a Communication Protocol - An Exercice in Focus*, Sonderforschungsbereich 342, Technische Universität München, 1992.

```
1: svc(send_to_mailbox,'clnt',[24656820,16'd6e0,0],18280.39, 1).
2: svc(allocate_memory,'xmtr',[-1,24,16'd494],18280.77, 1).
3: svc(pause,'clnt',[16'402],18281.33, 1).
4: svc(send_to_mailbox,'xmtr',[24656728,16'17a5908,0],18282.64, 1).
5: svc(receive_from_mailbox,'xmtr',[24656820,16'17a5808,16'd4dc],18282.97, 1).
6: svc(pause,'rcvr',[16'101],18320.31, 2).
7: svc(send_to_mailbox,'rcvr',[24656820,16'17a5a08,0],18420.31, 2).
8: svc(receive_from_mailbox,'rcvr',[24656728,16'17a5a08,16'd2e4],18420.67, 2).
9: svc(free_memory,'xmtr',[-1,24,16'17a5b00],18440.43, 1).
```

Figure 3 System call Prolog fact base.

```
1: cm(clnt,(126,svc(sndmbx,[loop4,client],
      [(output_gate,24656820),('&test_packet',55008),('0L',0),('&status',na),
       (32768,na)],18280.38,1,[return_status]))).
2: cm(xmtr,(x,svc(alloc,['malloc_if1.1',malloc,add_to_queue],
      [('-1L',-1),(alosiz,24),('&aloadr',54420),(0,na)],18280.77,1,[]))).
3: cm(clnt,(124,svc(pause,[loop4,client], [(1026,1026)],18281.33,1,[]))).
4: cm(xmtr,(74,svc(sndmbx,['if10.1','if4.2',loop1,transmitter],
      [(pipe_out,24656728),(pipe_output,24795400),('0L',0),('&status',na),
       (32768,na)],18282.64,1,[return_status]))).
```

Figure 4: Traces with line numbers and symbolic information.

```
actions([ snd(20300),  snd(22320),  snd(24340),  snd(26360),  snd(28381)]).
actions([putT(20301), putT(22321), putT(24342), putT(26361), putT(28392)]).
actions([getT(20441), getT(22440), getT(24540), getT(26740), getT(28540)]).
actions([retransmit(26560)]).
```

Figure 5 Action lists derived from event traces

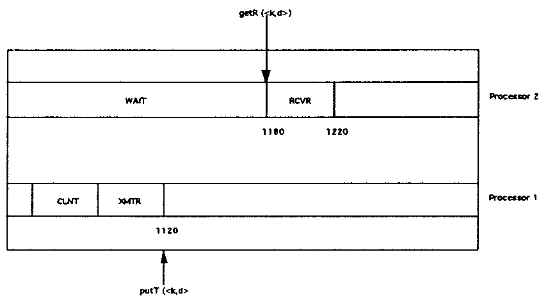

Figure 6: Graphical representation of the scheduler activity

Developing Parallel Applications Using High-Performance Simulation

Eric A. Brewer
William E. Weihl
MIT Laboratory for Computer Science*

Abstract

Researchers already use high-performance simulators for algorithm development and architectural studies. Advances in simulation technology and workstation performance have made program development on top of simulators—debugging, testing, and some tuning—fast enough for real applications. Simulators provide many advantages over running directly on a multiprocessor, including versatility, trivial repeatability, and detailed nonintrusive data collection and debugging. We make the case for application development via simulation and address simulation's traditional disadvantages. We also propose a development methodology that integrates simulation into the multiprocessor development environment. Finally, we examine the software engineering issues required by this integration and the use of parallel simulators for development.

We believe that most multiprocessor application development should be done on top of a high-performance simulator (running on a workstation) rather than directly on the multiprocessor. Researchers already use these high-performance simulators for algorithm development [CBDW91], architectural studies [HM92], and language and compiler design [WBC+91, HWW93]. The primary advantages of development via simulation are the cost effectiveness of workstations, the ability to exploit powerful sequential debuggers, the support for nonintrusive data collection and invariant checking, and the versatility of simulation.

Although traditional multiprocessor simulators are too slow to run real applications, high-performance simulators, including TangoLite [DGH91], RPPT [CDJ+91], and our own system PROTEUS [BDCW91], achieve orders of magnitude speedup over traditional simulators; typical development simulations take only a few minutes. The rapid increases in workstation performance will continue to diminish the turn-around time.

Although we designed PROTEUS to explore algorithms and language issues, our experience and that of other users indicates that development is easier and faster with PROTEUS than with real multiprocessors. For example, Colbrook found that the development of novel search-tree algorithms was about six times faster using PROTEUS compared to the previous development of search-tree algorithms on a transputer-based multiprocessor [Col, CBDW91], even though the algorithms developed on PROTEUS were far more complex. Furthermore, a typical simulation took only about a minute and produced higher quality measurements. Although our data comes from PROTEUS, the methodology applies to high-performance simulation in general.

After discussing high-performance simulators in Section 1, we make the case for application development via simulation We discuss the advantages and disadvantages of development on simulators in Sections 2 and 3, and then propose a methodology in Section 4. We change gears starting with Section 5, where we examine the issues that arise given this approach to development. These issues include simulating from a log, extending sequential debuggers, and ensuring source-code compatibility between the simulator and the target multiprocessor. Finally, in Sections 6 and 7, we address development on a concurrent simulator and summarize our conclusions.

1 Background

Traditional multiprocessor simulators interpret one instruction at a time. This allows arbitrary accuracy at tremendous cost. Such simulators typically require around 200 instructions per simulated instruction [CLN90], although the fastest versions (with less accuracy) can achieve 20–40 instructions per simulated instruction for uniprocessor architectures [Bed90].

High-performance simulators like PROTEUS and TangoLite use *direct execution* to improve performance Most simulated instructions are mapped to host instructions and thus approach zero overhead. Typically, assembly language basic blocks must be augmented with code that tracks the passage of simulated time, so these simulators have a lower bound of about two instructions per simulated instruction in practice. Early uses of direct execution for simulation include work by Fujimoto [FC88],

*E-mail brewer@lcs.mit.edu This work is supported by the National Science Foundation, grant CCR-8716884, by DARPA, Contract N00014-91-J-1698, by an equipment grant from Digital Equipment Corporation, and by grants from AT&T and IBM Eric Brewer is supported by an Office of Naval Research Fellowship The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U S government