An Object-Oriented Time Warp Simulation Kernel*

Radharamanan Radhakrishnan, Dale E. Martin, Malolan Chetlur, Dhananjai Madhava Rao, and Philip A. Wilsey

University of Cincinnati, Cincinnati, OH, USA

Abstract. The design of a Time Warp simulation kernel is made difficult by the inherent complexity of the paradigm. Hence it becomes critical that the design of such complex simulation kernels follow established design principles such as object-oriented design so that the implementation is simple to modify and extend. In this paper, we present a compendium of our efforts in the design and development of an object-oriented Time Warp simulation kernel, called WARPED. WARPED is a publically available Time Warp simulation kernel for experimentation and application development. The kernel defines a standard interface to the application developer and is designed to provide a highly configurable environment for the integration of Time Warp optimizations. It is written in C++, uses the MPI message passing standard for communication, and executes on a variety of platforms including a network of SUN workstations, a SUN SMP workstation, the IBM SP1/SP2 multiprocessors, the Cray T3E, the Intel Paragon, and IBM-compatible PCs running Linux.

1 Introduction

The Time Warp parallel synchronization protocol has been the topic of research for a number of years, and many modifications/optimizations have been proposed and analyzed [1, 2]. However, these investigations are generally conducted in distinct environments with each optimization re-implemented for comparative analysis. Besides the obvious waste of manpower to re-implement Time Warp and its affiliated optimizations, the possibility for a varying quality of the implemented optimizations exists.

The WARPED project is an attempt to make a freely available object-oriented Time Warp simulation kernel that is easily ported, simple to modify and extend, and readily attached to new applications. The primary goal of this project is to release an object-oriented software system that is freely available to the research community for analysis of the Time Warp design space. In order to make WARPED useful, the system must be easy to obtain, available with running applications, operational on several processing platforms, and easy to install, port, and extend.

This paper describes the general structure of the WARPED kernel and presents a compendium of the object-oriented design issues and problems that were

* Support for this work was provided in part by the Advanced Research Projects Agency under contracts DABT63–96–C–0055 and J–FBI–93–116.

© Springer-Verlag Berlin Heidelberg 1998

required to be solved. In addition, a description of two distinct application domains for WARPED is presented. WARPED is implemented as a set of libraries from which the user builds simulation objects. The WARPED kernel uses the MPI [3] portable message passing interface and has been ported to several architectures, including: the IBM SP1/SP2, the Cray T3E, the Intel Paragon, a network of SUN workstations, an SMP SUN workstation, and a network of Pentium Pro PCs running Linux.

The WARPED system is implemented in C++ and utilizes the object-oriented capabilities of the language. Even if one is interested in WARPED only at the system interface level, they must understand concepts such as inheritance, virtual functions, and overloading. The benefit of this type of design is that the end user can redefine and reconfigure functions without directly changing kernel code. Any system function can be overloaded to fit the user's needs and any basic system structure can be redefined. This capability allows the user to easily modify the system queues, algorithms or any part of the simulation kernel. This flexibility makes the WARPED system a powerful tool for Time Warp experimentation.

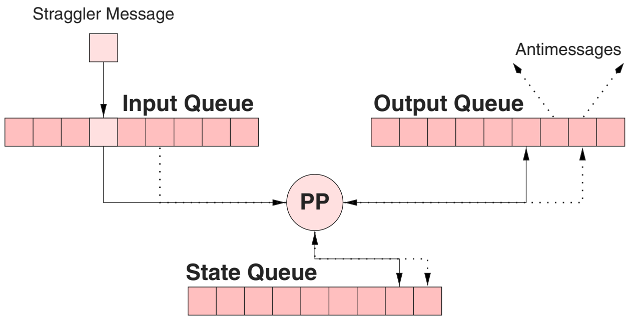

Fig. 1. A logical process in a Time Warp simulation

Another benefit of the object-oriented nature of the WARPED application interface is that by its very design it is simple to "plug in" a different kernel. A sequential simulation kernel is supplied in the WARPED distribution in addition to the Time Warp kernel. Version 0.9 of the WARPED is available via the www at `http://www.ece.uc.edu/~paw/warped/`. The remainder of this paper is organized as follows. Section 2 presents a description of the Time Warp paradigm. Section 3 details the WARPED kernel's application/kernel interface and presents a compendium of the design issues that were required to be solved for the development of the WARPED system. Section 4 demonstrates, through two examples, the construction of simulation applications using the WARPED kernel. Finally, Sect. 5 contains some concluding remarks.

2 Background

In a Time Warp synchronized discrete event simulation, *Virtual Time* [2] is used to model the passage of the time in the simulation. The virtual time defines a total order on the events of the system. The simulation state (and time) advances in discrete steps as each event is processed. The simulation is executed via several simulator processes, called simulation objects or logical processes (LP). Each LP is constructed from a physical process (PP) and three history queues. Figure 1 illustrates the structure of an LP. The input and the output queues store incoming and outgoing events respectively. The state queue stores the state history of the LP. Each LP maintains a clock that records its Local Virtual Time (LVT). LPs interact with each other by exchanging time-stamped event messages. Changes in the state of the simulation occur as events are processed at specific virtual times. In turn, events may schedule other events at future virtual times.

The LPs must be synchronized in order to maintain the causality of the simulation; although each LP processes local events in their (locally) correct time-stamp order, events are not globally ordered. Fortunately, each event need only be ordered with respect to events that affect it (and, conversely, events that it affects); hence, only a partial order of the events is necessary for correct execution [4]. Under optimistically synchronized protocols (*e.g.,* the Time Warp model [2]), LPs execute their local simulation autonomously, without explicit synchronization. A causality error arises if a LP receives a message with a time-stamp earlier than its LVT (a *straggler* message). In order to allow recovery, the state of the LP and the output events generated are saved in history queues as events are processed. When a straggler message is detected, the erroneous computation must be undone—a *rollback* occurs. The rollback process consists of the following steps: the state of the LP is restored to a state prior to the straggler message's time-stamp, and then erroneously sent output messages are canceled (by sending *anti-messages* to nullify the original messages). The global progress time of the simulation, called Global Virtual Time (GVT), is defined as the time of the earliest unprocessed message in the system [1, 5, 6]. Periodic GVT calculation is performed to reclaim memory space as history items with a time-stamp lower than GVT are no longer needed, and can be deleted to make room for new history items.

3 The WARPED Application and Kernel Interface

The WARPED kernel presents an interface to the application from building logical processes based on Jefferson's original definition [2] of Time Warp. Logical processes (LPs) are modeled as entities which send and receive events to and from each other, and act on these events by applying them to their internal state. This being the case, basic functions that the kernel provides to the application are methods for sending and receiving events between LPs, and the ability to specify different types of LPs with unique definitions of state. One departure

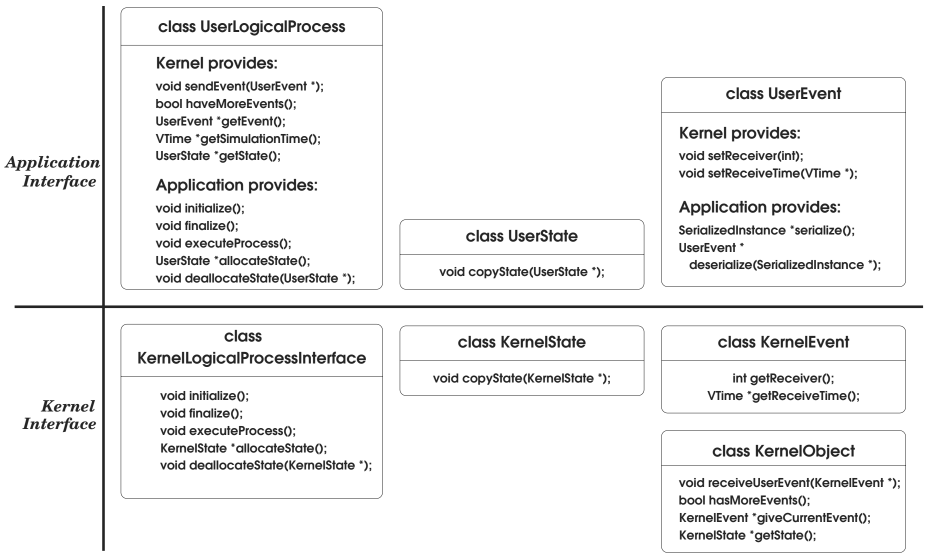

Fig. 2. Application and kernel interface

from Jefferson's presentation of Time Warp is that LPs are placed into groups called "clusters". LPs on the same cluster communicate with each other without the intervention of the message system, which is *much* faster than communication through the message system. Hence, LPs which communicate frequently should be placed on the same cluster. Another feature of the cluster is that it is responsible for scheduling the LPs. Note that the LPs within a cluster operate as Time Warp processes; even though they are grouped together, they aren't coerced into synchronizing with each other.

Control is passed between the application and the kernel through the cooperative use of function calls. This means that when a function is called in application code, the application is not allowed to block for any reason. Since the application has control of the single thread of control through its cluster, it could end up waiting forever. In order for the kernel to correctly interact with the application code, the user must provide several functions to the kernel. These functions define such things as how to initialize each LP, and what each LP does during a simulation cycle. In addition, if the user would like to use a non-standard definition of time, facilities are in place to provide a user-defined time class to the kernel. By default, WARPED has a simple notion of time. More precisely, time is defined in the class `VTime` as a signed integer. Obviously, particular instances of an application may have different requirements for the concept of time. For example, simulators for the hardware description language VHDL [7] require a more complex definition of time. If the simple, kernel-supplied version of time is not sufficient, the application programmer must define the class `VTime` with data members appropriate to the application's needs. In addition, the user must use the preprocessor macro `USE_USER_VTIME` during compilation. The WARPED kernel also has requirements about the defined methods of the type `VTime`. Specifically, the implementation of `VTime` must supply the following operators and data, either by default or through explicit instantiation:

– Assignment (=), Addition (+), and subtraction (–) operators.
– The relational operators: ==, !=, >=, <=, >, <.
– Constant objects `ZERO`, `PINFINITY`, and `INVALID_VTIME` of type `VTime`, which define, respectively, the smallest, largest, and invalid time values.
– `INVALID_VTIME` must be less than `ZERO`.
– The insertion operator (<<) for class ostream, for type `VTime`.

The application interface is implemented through the object-oriented features of the C++ language. The simulation kernel is built from several classes, allowing the user to define a system configuration by specifying the classes to use, without rewriting system code. Application specific code is derived from the WARPED kernel. This allows application code to transparently access kernel functions and is restrictive enough to hide communication and Time Warp details from the user. This section describes what is necessary for an application writer to provide the WARPED kernel, and what the simulation kernel provides to the application in return. To use the WARPED kernel, the application programmer must provide three class definitions corresponding to the logical process (LP), the notion of state for that LP, and a definition (or definitions) for events.

LPs form the core of the discrete event simulation. An LP represents an entity that can send/receive events to/from other LPs. As a result of these events, changes are made to the LP's internal state (and output may result). Figure 2 illustrates the application and the kernel interfaces presented by the WARPED system. The interface as seen by an user's LP is represented by the `UserLogicalProcess` class definition. The class definition is divided into two parts. The first part is the set of methods that the kernel provides to the LP. These methods are provided by the kernel to the LP for communication (`sendEvent`, `getEvent`), querying the kernel for information (`haveMoreEvents`, `getSimulationTime`) and for accessing its state (`getState`). In addition to these methods, there are some internal methods that the kernel calls periodically. These include message polling primitives to check for the arrival of messages from remote processors and garbage collection primitives. The second part consists of a set of methods that the application writer overrides. The kernel will call these methods at various times through out the simulation. Each method in this set has a specific function. The `initialize` method gets called on each LP before the simulation begins. This gives each LP a chance to perform any actions required for initialization. For example, initialization might include opening files, setting up the initial state of an LP or the transmission of initial setup events to the distributed processes in the simulation. Conversely, the method `finalize` is called after the simulation has ended. This allows the LPs to "clean up" after themselves, perform actions such as closing files, compute statistics, and produce output. The method `executeProcess` of an LP is called by the kernel whenever the LP has at least one event to process. The kernel calls `allocateState` in an LP when it needs the LP to allocate a state on its behalf. `deallocateState` is called by the kernel to hand back a state to the application when it is done with it. At this point, the application may deallocate it, or store it for later use.

Any LP will have some state that needs to be defined. The LP modifies its state in response to various events that it receives. This behavior is completely user application specific and the application must define certain methods related to state for the simulation kernel to call. These methods include the creation and the duplication of the state. Figure 2 illustrates the user application's interface to the state. The method `copyState` is called by the kernel to copy the data from the `UserState` into a newly created state which is then archived (for rollback recovery purposes). This method must be overridden by the user application. If the application's definition of state contains no pointers, then a bitwise copy is adequate for this method. If the application contains pointers in its state, or objects that contain pointers, then this method has to take appropriate actions to copy the pointers "correctly", as defined by the needs of the application. This is necessary because the kernel has no knowledge about the user application's state.

Events represent the communication between the LPs. Figure 2 illustrates the definition of the `UserEvent` class. Once again, the definition is a two part definition wherein one set of methods is provided by the kernel and the other set is overridden by the application writer. The method `setReceiver` allows the application to set the simulation id of the receiving LP [1]. `setReceiverTime` allows the application to set the simulation time that this event should be received at. The methods `serialize` and `deserialize` are provided so that the application may maintain architectural transparency and portability among events. It is also necessary for checkpointing in optimistic fossil collection [8] and failure recovery.

The design of the WARPED API was motivated by several design issues. These issues were central to the object-oriented design of the system and needed to be solved for constructing a simple and extensible programming interface. For example, when the kernel needs information about data structures within the application, they can be passed into the kernel in two ways : through template classes or through virtual interface methods. One example of this is the state class definition. The user state can be passed into the kernel through templates. All that is required is that the `UserLogicalProcess` class be templatized on `UserState`. However, to reduce overall compilation time, static executable size and facilitate the use of different types of states, the templatization approach was avoided. The convention currently followed is to have the LP and the kernel share the responsibility of allocating, maintaining and deallocating the state through the use of virtual methods. Although the common perception in scientific computing is that abstraction is the enemy of performance, we have found that the extensive use of virtual methods and other abstractions does not drastically affect performance. When kernel data or functions need to be made available to the user, they can be accessed by one of two mechanisms: through the C++

[1] As it is the user's responsibility to register an LP with a unique simulation id, the application can use the `setReceiver` method to connect LPs together.

inheritance mechanism (classes that the user defines must be derived from kernel defined classes), and through "normal" function calls to methods defined by objects in the WARPED kernel.

In addition, avoiding the use of templates facilitates the distribution of source code as stand alone libraries which do not require recompilation. This enables the development of "object factories" by independent vendors. With object factories, vendors can permit different users to use various components from their object factories without revealing the source code. To enable this type of "plug-and-play", C++ *composition* was carried out in preference to *inheritance* in the source code. Composition also helps in achieving dynamic algorithm/method reconfiguration (*i.e.*, reconfiguration "on-the-fly" without recompilation).

Also, avoiding the use of templates makes the WARPED system simpler to port to different compilers on different architectures. To achieve interoperability on heterogeneous platforms, the serialization and deserialization operations play a vital role. Currently serialization and deserialization of events as well as states is supported. These operations are invoked only when events or states cross architecture boundaries. Serialization and deserialization is also applied to checkpointing to facilitate failure recovery.

In the current version of WARPED, there are several Time Warp implementation optimizations that can be turned on/off. A configuration file is used to allow the user to change between the options of the simulation kernel at compile time. These options fall under several broad categories: Schedulers, Fossil Managers, State Managers, Memory Managers, and Time Warp optimizations (such as dynamic cancellation [9], dynamic checkpointing [10] and dynamic message aggregation [11]). The user specifies a selection from this set of options and compiles this selection. A better way to implement this is through dynamic configuration. Each optimization is implemented as a specific function and at run-time, a simulation object (or some central configuration object) can dynamically select and reconfigure (through function pointers) the optimization and switch between optimizations if the need arose [12].

4 Applications for WARPED

Several applications have already been developed that use the WARPED kernel. These applications primarily belong to two application domains: a queuing model simulation library called KUE, and TyVIS, a simulation kernel for the VHDL hardware description language [7]. KUE is a simple package developed for debugging, testing, and initial profiling of WARPED and any extensions thereof. TyVIS is a larger package designed to stress the simulation kernel with large examples of digital systems. It also demonstrates the extensibility of the WARPED kernel. The developers hope that other investigators will implement additional applications with WARPED which they can include as part of the distribution. As space constraints prevent us from presenting the performance of the WARPED system, the rest of this section is devoted to the description of WARPED applications.

4.1 KUE: A Queuing Model Library

The KUE system is a library of queuing models built on top of the WARPED kernel. KUE is a set of C++ classes that enable the creation of parallel queuing applications. XKUE is a TCL/TK front end for queue to allow "point and click" creation of queuing models. The KUE library contains class definitions of seven different queuing objects (source, fork, join, delay, queue, server and sink objects). Each object class definition encapsulates the functionality of the queuing object in accordance with the WARPED interface. Two examples are distributed with the WARPED kernel that make use of the KUE libraries.

The first, SMMP, is designed to simulate several processors, each with their own cache, and sharing a global memory. The model is generated by a program which lets the user adjust the following parameters: the number of processors/caches to simulate, the number of LPs to generate, the speed of cache, the speed of main memory, and the cache hit ratio. The second example, RAID, is a simulation of a nine disk RAID level 5 array of IBM 0661 3.5" 320MB SCSI drives with a flat-left symmetric parity placement policy. Sixteen processes generate requests for data stripes of random lengths and locations. These requests are sent to fork processes which split them into specific disk-level requests according to the RAID placement policy. The nine server processes, one per simulated disk, process the requests in a first-come first-served fashion. After processing each request, the disks route their responses back to the originating processes. Both these sample queuing applications posses class definitions that derive from the seven basic queuing model definitions in the KUE library. Further details regarding these applications are available in the literature [12].

4.2 TyVIS: A Parallel VHDL Simulation Kernel

The TyVIS VHDL simulation kernel was designed to take advantage of the object-oriented design of the WARPED kernel. It requires no modifications to the kernel, yet extends WARPED with full VHDL simulation capability (as described in [7]). Its implementation takes advantage of several design features of WARPED, and even reuses some of WARPED's basic classes for TyVIS's internal data structures. The main class of TyVIS is `VHDLKernel`, which is derived from the `UserLogicalProcess` class.

The semantics of VHDL require that certain events generated during a simulation cycle not be applied to a signal's value, based upon each event's timestamp. This process is called marking, and is best implemented with a time-ordered queue. Rather than writing an entirely new data structure, the `OutputQueue` class of the WARPED distribution was reused, becoming a base class for the `MarkedQueue` class. The public interface to `MarkedQueue` is identical to that of `OutputQueue`; all additional data members and methods are private. This reuse of the existing code allowed the `MarkedQueue` class to be written and debugged in a matter of a few hours. Also, since `MarkedQueue` only accesses the public interface of `OutputQueue`, any changes in the implementation of `OutputQueue` will be transparent.

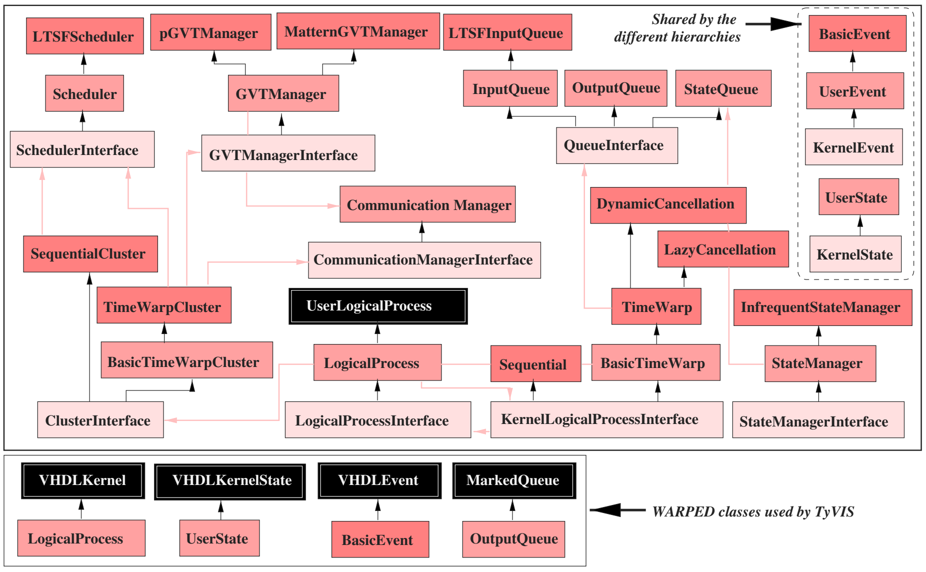

Fig. 3. A synopsis of WARPED's class derivation hierarchy

Each VHDL process has a unique state class which defines the VHDL signals and local variables that the process can access. This state class is built from WARPED's `UserState` class with the necessary user-defined methods. This allows the Time Warp functions of state queuing, rollbacks, and garbage collection to proceed normally. The only requirement to the state class for this is that the class define `operator=`.

Processes are invoked by calling `VHDLKernel::executeProcess()`, which overrides the similar method in the `UserLogicalProcess` class. This method updates LVT and applies all events in the input queue occurring on any signals contained in the process at the current time. The specific VHDL process code is then executed by calling the object's `executeVHDL` method, supplied by the user. When the process returns control to the VHDL kernel, the kernel then determines which newly generated events need to be transmitted to other processes, and transmits them, using the `sendEvent` call from the WARPED kernel. Eventually, control is returned to WARPED. If a process is rolled back, the VHDL kernel never knows about it, since all related processing is contained entirely in the WARPED code, lower down in the derivation hierarchy. Complete replacement of the WARPED kernel with a conservatively synchronized simulation kernel would have no effect on TyVIS; it is completely isolated from whatever processing is performed by WARPED. A synopsis of the WARPED class derivation tree is illustrated in Fig. 3. Base class definitions form the root of the derivation hierarchy. Figure 3 also depicts the WARPED classes reused by the TyVIS VHDL simulation kernel.

5 Conclusions

The WARPED Time Warp simulation project is an attempt to produce a widely available, highly portable, and an object-oriented Time Warp simulation kernel complete with operational applications for testing and analysis. The software is written in C++ and uses the MPI portable message passing interface. The system operates on a distributed or shared memory multiprocessor as well as on a network of workstations. Several applications have been developed and are jointly released with the software.

The intent of this effort is to make a testbed available for experimentation and analysis of Time Warp and all its affiliated optimizations. For this purpose, an object-oriented design approach has been followed with the aim of making the software easy to extend. Our experiences in the design and development of WARPED were also presented. In addition, a synopsis of the application programming interface of WARPED was also presented. We hope that as investigators use and extend the capabilities of the kernel that we will be allowed to integrate those extensions into the basic kernel release so that others can likewise benefit from, and independently confirm, the analysis of the extensions. Furthermore, we expect that additional test cases for the existing (and ideally, new) applications will be independently developed and submitted for inclusion into the kernel release (and thereby promoting reuse of source code).

References

[1] R. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[2] D. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):405–425, July 1985.

[3] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, Cambridge, MA, 1994.

[4] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of ACM*, pages 558–565, July 1978.

[5] Yi-Bing Lin. Memory management algorithms for optimistic parallel simulation. In *6th Workshop on Parallel and Distributed Simulation*, pages 43–52. Society for Computer Simulation, January 1992.

[6] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18(4):423–434, August 1993.

[7] *IEEE Standard VHDL Language Reference Manual.* New York, NY, 1993.

[8] C. H. Young and P. A. Wilsey. Optimistic fossil collection for Time Warp simulation. In H. El-Rewini and B. D. Shriver, editors, *29th Hawaii International Conference on System Sciences (HICSS-29)*, volume Volume I, pages 364–372, January 1996.

[9] R. Rajan, R. Radhakrishnan, and P. A. Wilsey. Dynamic cancellation: Selecting Time Warp cancellation strategies at runtime. *VLSI Design*, 1998. (forthcoming).

[10] J. Fleischmann and P. A. Wilsey. Comparative analysis of periodic state saving techniques in Time Warp simulators. In *Proc. of the 9th Workshop on Parallel and Distributed Simulation (PADS 95)*, pages 50–58, June 1995.

[11] M. Chetlur, N. Abu-Ghazaleh, R. Radhakrishnan, and P. A. Wilsey. Optimizing communication in Time-Warp simulators. In *12th Workshop on Parallel and Distributed Simulation*. Society for Computer Simulation, May 1998.

[12] R. Radhakrishnan, N. Abu-Ghazaleh, M. Chetlur, and P. A. Wils ey. On-line configuration of a Time Warp parallel discrete event si mulator. In *1998 International Conference on Parallel Processing, (ICPP'98)*. IEEE Computer Society Press, August 1998. (forthcoming).