WARPED: A Time Warp Simulation Kernel for Analysis and Application Development

Dale E. Martin, Timothy J. McBrayer, and Philip A. Wilsey
Dept. of ECECS, PO Box 210030 Cincinnati, OH 45221-0030

Abstract
WARPED is a publicly 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 and shared memory 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 Intel Paragon, and IBM compatible PCs running Linux. WARPed is distributed with several applications and includes a sequential kernel implementation for comparative analysis. The kernel supports LP clustering, various Time Warp algorithms, and several optimizations that dynamically adjust simulation parameters.

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, 4]. However, these investigations are generally conducted in distinct environments with each optimization reimplemented for comparative analysis. Besides the obvious waste of manpower to reimplement 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 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 a 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 modify.

This paper describes the general structure of the WARPed kernel and its integration with three distinct application domains. WARPed is implemented as a set of C++ libraries from which the user builds simulation objects. The WARPed kernel uses the MPI [2] portable message passing interface and has been ported to several architectures, including the IBM SP1/SP2, the Intel Paragon, a network of SUN workstations, an SMP SUN workstation, and a 486 PC running Linux. The ports have required nominal effort and the code has been extensively analyzed and improved using purify, a memory analysis tool, quantify, an execution analysis tool, and purecov an execution path analysis tool. The WARPed project is ongoing and the software will continue to be updated, revised, and distributed. This paper describes version 0.5 of WARPed software which 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 high level overview of the WARPed kernel and its interface to an application environment. Section 3 demonstrates, through three examples, the construction of simulation applications using the WARPed kernel. The testing and current status of the WARPed kernel is briefly described in Section 4. Finally, Section 5 contains some concluding remarks.

2 Overview of the WARPed System
The WARPed kernel provides the functionality to develop applications modeled as discrete event simulations. Considerable effort has been made to hide the details of Time Warp from the application interface. For example, sending events from one simulation object to another is done in the same way regardless of whether the objects are on a single processor or on different processors; all Time Warp specific activities such as state saving, rollback, and so on are performed automatically by the kernel without intervention from the application. A list of services that the kernel supplies to the application follows:

• Event delivery: communication between simulation objects in the WARPed universe is provided by the kernel.
• Optimistic synchronization between parallel processes: WARPed can be run on parallel processors ranging from clusters of workstations to large scale multiprocessors, and Time Warp activities will be performed transparently to the user process.

In order for these services to be provided by the kernel, the application must provide certain constructs to the kernel (Figure 1). In particular, the application specific definitions of events and state must be defined. If a non-integer
definition time of time is desired, then this definition must also be provided.

Simulation objects are grouped together into entities called logical processes, or LPs (Figure 2). Processor parallelism occurs at the LP level and each LP is responsible for GVT management, communication management, and scheduling for the simulation objects that it contains. In addition, communication between simulation objects within the same LP is performed by direct insertion into the input queue of the receiving object.

Since the parallelism occurs at the logical process level, simulation objects which execute relatively independently of each other can be placed on separate LPs to maximize parallelism. Conversely, simulation objects that frequently communicate with each other should be placed on the same LP to benefit from fast intra-LP communication.

Partitioning in WARPED occurs explicitly in the instantiation and registration of simulation objects with an LP. Processor allocation to LPs occurs at runtime using the MPICH mechanism of group files — each machine in the simulation is listed in the order that they will be assigned ids. Automatic partitioning of simulation objects to LPs is not provided in WARPED; furthermore, load balancing is not currently implemented.

The WARPED system is composed of a set of C++ libraries which the user accesses in several ways. Where the kernel needs information about data structures within the application, they are passed into kernel template classes. When kernel data or functions need to be made available to the user, they can be accessed by one of two mechanisms:

1. Through the C++ inheritance mechanism. That is, certain classes that the user defines (to be described later) must be derived from kernel defined classes.

2. Through “normal” function calls to methods defined by objects in the WARPED kernel.

To use the WARPED kernel, the application programmer must provide three class definitions corresponding to (i) the simulation object, (ii) the notion of state for that simulation object, and (iii) a definition (or definitions) for events. The simulation object classes must be derived from the class TimeWarp and the state class is derived from the class BasicState. The TimeWarp class is templated on state, so a TimeWarp object must be instantiated with each state class that is defined.

Events to be passed between simulation objects are defined by the application programmer as a set of class definitions. These must be derived from the BasicEvent class. In the current version of WARPED events cannot contain dynamically allocated data or pointers to data.

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 a simulation with the WARPED kernel may have different requirements for the concept of time. For example, simulators for the hardware description language VHDL require a more complex definition of time. Thus, WARPED includes a mechanism for defining a more complex structure for 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 define 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 of type VTime named ZERO, INFINITY, and INVALID_VTIME, which define, respectively, the smallest, largest, and invalid time values. INVALID_VTIME must be < ZERO.
- The insertion operator (<<) for class ostream, for type VTime.

A more detailed description of the internal structure and organization of the WARPED kernel is available on the www at http://www.ece.uc.edu/~paw/warped.

3 Applications for WARPED

Three applications have already been developed that use the WARPED kernel. These applications are (i) KUE, a simple queuing model simulation package, (ii) RAID, a level 5 RAID disk array model, and (iii) Tyvis, a simulation kernel for the VHDL hardware description language. KUE and RAID are simple packages developed for debugging, testing, and initial profiling of WARPED and any extensions thereof. Tyvis is a larger package designed to

---

Figure 1: The Relationship Between the Application and the WARPED Kernel
stress the simulation kernel with large examples of digital systems. It also demonstrates the extensibility of the \textsc{warped} kernel. The developers hope that other investigators will implement additional applications with \textsc{warped} that they can include as part of the distribution.

3.1 KUE: A Queuing Model Simulator

The KUE system is a library built on top of the \textsc{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 queuing to allow "point and click" creation of queuing models.

Two examples are distributed with the \textsc{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: (i) the number of processors/caches to simulate, (ii) the number of LPs to generate, (iii) the speed of cache, (iv) the speed of main memory, and (v) the cache hit ratio.

The second example models a simple traffic police telecommunications network. The model is divided into districts, counties, and stations. Each district contains many counties; each county has one or more police stations; and each police station has a fixed amount of resources such as traffic police cars, motorcycles, and so on. The queuing model attempts to represent a set of simple police stations coexisting within a given city. Each police station receives a random number of calls and each of these calls are routed by the telecommunication network to the nearest police car or motorcycle. Each car may take a random amount of time to process a call and sends back a message when the job is done.

3.2 RAID: A Level 5 RAID Disk Array

The RAID application is a simulation of a nine disk RAID level 5 array of IBM 6661 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.

3.3 TYSIS: A Parallel VHDL Simulator

The TYSIS VHDL simulation kernel was designed to take advantage of the object-oriented design of the \textsc{warped} kernel. It requires no modifications to the kernel, yet extends \textsc{warped} with full VHDL simulation capability (as described in [3]). Its implementation takes advantage of several design features of \textsc{warped}, and even reuses some
of WARPED's basic classes for TVVIS's internal data structures. The main class of TVVIS is VHDLKernel, which is derived from the TimeWarp 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 write an entirely new data structure, the WARPED SortedList class was reused, becoming a base class for the MarkedQueue class. The public interface to MarkedQueue is identical to that of SortedList; 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 SortedList, any changes in the implementation of SortedList will be transparent.

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 derived from WARPED's BasicState class. 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=.

Any Time Warp operations performed by the WARPED kernel are completely hidden from the TVVIS simulator, except for garbage collection. The original design goal was to have no time warp-specific code in TVVIS. However, the MarkedQueue, which contains a superset of the events in the OutputQueue must be garbage collected along with the OutputQueue. This is accomplished by overriding TimeWarp::garbage() with VHDLKernel::garbage() so that the MarkedQueue is garbage collected in addition to the work normally performed in TimeWarp::garbage().

A VHDL process is invoked from the TimeWarp kernel by calling VHDLKernel::executeProcess(), which overrides the standard method in the TimeWarp 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, higher up in the derivation tree. Complete replacement of the WARPED kernel with a conservative, synchronous simulation kernel would have no effect on TVVIS; it is completely isolated from whatever processing is performed by WARPED.

4 Testing and Status of WARPED

The WARPED simulation kernel has been developed and released for public use. Queueing simulations processing up to 10 million tokens have successfully run on a four processor Sparc Center 1000 workstation. Limited testing has occurred on an IBM SP1, the Intel Paragon, and a Linux-based 486 PC. The entire WARPED kernel consists of slightly less than 8000 lines of C++ source code and compiled with the g++ compiler (-O2) generates almost 650K bytes of object code. The WARPED kernel is freely available with no restrictions (not even the GPL requirement of source code release). Information about the system and access to the software can be found on the www at http://www.ece.uc.edu/~paw/warped/.

5 Conclusions

The WARPED project is an attempt to produce a widely available, highly portable, 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 or on a network of workstations. Three applications have been developed and are jointly released with the software (albeit the TVVIS VHDL simulation kernel does have some restrictions).

The intent of this effort is to make a testbed available for experimentation and analysis of Time Warp and its affiliated optimizations. 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, analyses 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.

Acknowledgments

The authors would gratefully like to acknowledge the suggestions and contributions of John Penix, Lantz Moore, Radharamanand Radharkrishna, Raghu Raveendran, C. Rajan, Balakrishnan Kannikeswaran, and Chris Young.

References