This paper appeared in the Proceedings of the Annual Simulation Symposium, ASS-1996.

A Comparative Analysis of Various Time Warp Algorithms Implemented in the WARPED Simulation Kernel*

Radharamanan Radhakrishnan, Timothy J. McBrayer, Krishnan Subramani, Malolan Chetlur, Vijay Balakrishnan, and *Philip A. Wilsey*
Computer Architecture Design Laboratory
Dept. of ECECS, PO Box 210030, Cincinnati, OH 45221–0030
(513) 556-4779, phil.wilsey@uc.edu

Abstract

The Time Warp mechanism conceptually has the potential to speedup discrete event simulations on parallel platforms. However, practical implementations of the optimistic mechanism have been hindered by several drawbacks such as large memory usage, excessive rollbacks (instability), and wasted lookahead computation. Several optimizations and variations to the original Time Warp algorithm have been presented in the literature to optimistically synchronize Parallel Discrete Event Simulation. This paper uses a common simulation environment to present comparative performance results of several Time Warp optimizations in two different application domains, namely: queuing model simulation and digital system simulation. The particular optimizations considered are: Lowest Timestamp First (LTSF) Scheduling, Periodic (fixed period) Checkpointing, Dynamic Checkpointing, Lazy Cancellation, and Dynamic Cancellation.

1 Introduction

The Time Warp Mechanism implements the concept of Virtual Time to optimistically synchronize parallel simulation. The Time Warp parallel synchronization protocol has been the topic of research for a number of years. However, the successful utilization of the Time Warp mechanism has been plagued by the time and space overheads of rollback, namely: state saving, state restoration and event reprocessing. Many modifications/optimizations to Time Warp have been proposed and analyzed [7, 10]. 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.

In this paper, we present empirical data on the performance of several Time Warp optimizations using the problem domains of queuing model simulation and digital system simulation. In particular, performance results for the optimizations lazy cancellation [8, 24], dynamic cancellation [20], periodic checkpointing, [10] and dynamic checkpointing [6, 12, 17, 22] are given. Performance data is collected using version 0.5 of the WARPED software [13, 14] running on a 4 processor SUN SparcCenter 1000 and for comparative purposes we also present data collected on the Intel Paragon.

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 discusses the various optimizations that have been developed for the WARPED kernel. Performance results with two distinct applications (queuing model simulation and digital system simulation) are shown 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 [13, 14].

*Support for this work was provided in part by the Advanced Research Projects Agency under contract J–FBI–93–116, monitored by the Department of Justice.

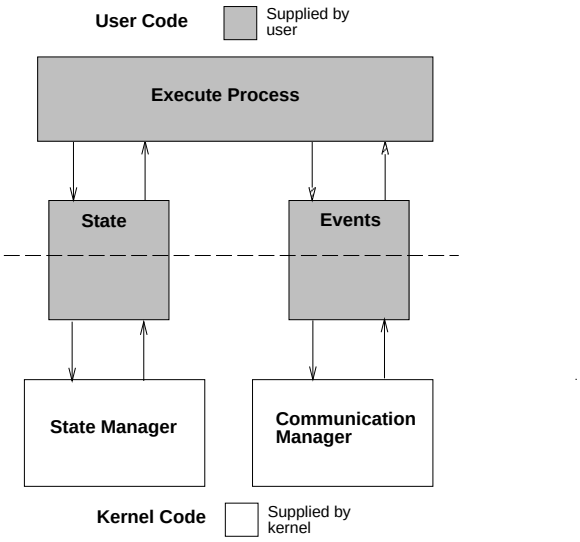

Figure 1. The Relationship Between the Application and the WARPED Kernel

Considerable effort has been made to define a standard programming interface 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. Consequently, an implementation of the WARPED interface can be constructed using either conservative [15] or optimistic [5, 10] parallel synchronization techniques; furthermore, the simulation kernel can also exist as a sequential kernel. In fact, the current software distribution of WARPED includes both sequential and parallel (Time Warp) implementations. As previously indicated, this independence is achieved through a standard interface [13]. This interface is briefly described below.

The WARPED interface requires that the simulation kernel provide the following services to the application:

- *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 also 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 LP 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 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 templatized 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.

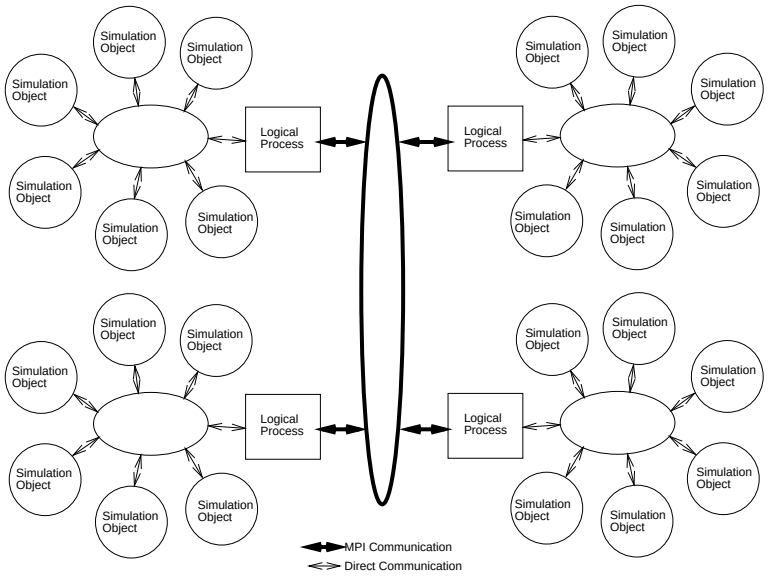

Figure 2. Structure of LPs and Simulation Objects in the WARPED System

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 [16, 18] 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`, `PINFINITY`, 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 Time Warp Optimizations Implemented in WARPED

Parallel discrete event simulation (PDES), using the Time Warp mechanism for synchronization, has the potential to produce large speedup, but has never been fully utilized due to implementation and memory overheads. Several optimizations have been proposed in the literature to reduce overheads, and thereby produce a larger speedup. These overheads can largely be classified along the following dimensions:

- Wasted lookahead computation, and unnecessary recomputation of states and event execution. Straggler event processing can result in the propagation of unnecessary rollback eventually creating a thrashing effect in optimistically synchronized simulators [7].
- Time and space overhead due to excessive state saving. Because most parallel systems have a somewhat limited memory space, excessive state savings can cause severe simulation slow-down (due to swapping of memory), and in some cases, failure [3].

Correspondingly, Time Warp optimizations can be divided into two types, namely: (i) optimizations to decrease the wasted lookahead computation, and (ii) optimizations to reduce the memory overhead. Lazy Cancellation and Dynamic Cancellation have been proposed to avoid unnecessary lookahead computation. Periodic Checkpointing and Dynamic Checkpointing have been proposed to alleviate the memory overhead problem.

In this paper, empirical data relating the effectiveness of these Time Warp optimizations are presented. In particular, several applications from the domains of queuing model simulation and digital system simulation are used to exercise the WARPED kernel with various configurations of optimizations enabled/disabled. The remainder of this section describes these optimizations in more detail.

3.1 Lazy Cancellation

The performance of a Time Warp simulator depends on the efficiency of the cancellation strategy employed to undo the effects of the erroneous computation. Two known cancellation strategies exist, namely aggressive cancellation [10], and lazy cancellation [8, 24]. Under aggressive cancellation the arrival of a straggler message and the subsequent rollback forces the immediate generation of antimessages for all output messages that were processed prematurely. In contrast, using lazy cancellation, the sending of antimessages is delayed until forward processing demonstrates (by comparison of old and new output) that the originally sent output messages were incorrect. Thus, there is a potential reduction in communication as well as a decrease in wasted lookahead computation. As a consequence of lazy cancellation the total number of rollbacks is frequently reduced. Lazy cancellation, however, relies heavily on the regeneration of the same output message for its performance. The performance under lazy cancellation deteriorates if the probability of the regenerated output messages being different from the originally sent messages is high. In contrast, aggressive cancellation performs poorly if the same messages are generated before and after a rollback most of the time. Several independent studies have shown that lazy cancellation frequently performs better than aggressive cancellation, but that, even within the same application domain, some executions perform better under aggressive cancellation [2, 21]. Unfortunately, practical techniques to statically analyze an application for selecting cancellation strategies have yet to be developed [11]. Consequently, many investigators simply use lazy cancellation. Recently, however, a technique for dynamically selecting the cancellation strategy has been proposed [20]

3.2 Dynamic Cancellation

Dynamic cancellation is a technique to have each Time Warp object decide for itself which cancellation strategy to employ [20, 19]. That is, each Time Warp object analyzes its recent history for evidence of success in lazy cancellation. More precisely, the object maintains a lazy hit/miss ratio and uses this ratio to decide which cancellation strategy to employ. The decision to dynamically switch at the Time Warp object level was made by a detailed analysis of the lazy hit/miss ratio for all objects in various applications [19]. In this study, Rajan observes that the best cancellation strategy varies even across the individual Time Warp objects of a simulation.

The selection of which cancellation strategy to use is decided using non-linear control techniques. In particular, Rajan uses a thresholding function with a dead zone to select the cancellation strategy. More precisely, all Time Warp objects begin by using aggressive cancellation. Then, after a sufficiently long measurement cycle, the lazy hit/miss ratio is used to select cancellation strategies; if the value falls below a lower threshold, aggressive cancellation is used; changes from aggressive to lazy occur when the ratio crosses an upper boundary; and values in the dead zone remain unchanged. The setting of the upper and lower bounds requires some tuning based on the specific time warp simulator and its implementation costs [19].

Dynamic cancellation does, however, require an additional cost for implementation. In particular, even when using aggressive cancellation, a Time Warp object must continue to perform comparison of output events in order to revise the lazy hit/miss ratio. Alternative techniques such as permanently switching into aggressive cancellation when crossing the lower threshold and other approaches are alluded to by Rajan [19], but not fully investigated.

3.3 Periodic Checkpointing

Memory consumption is a potentially onerous problem for large Time Warp simulations. Time Warp objects with large states require considerable memory space as well as CPU cycles for state saving. In general, states are saved after every event execution in a Time Warp simulation. However, it is possible to save state periodically, after a certain number of fixed event executions. Thus the arrival of a straggler message with periodic state savings may require the system to rollback to an earlier state and coast forward, reconstructing the state required to correctly execute the straggler event in its proper order. While coasting forward, no messages are sent out to the other processes in the system. The difficulty of periodic state saving is determining an appropriate fixed frequency for checkpointing. Some applications operate best with a fairly small value; while others require much larger values. As with cancellation strategies, no practical techniques for statically analyzing generic applications to decide the checkpoint frequency are known.

3.4 Dynamic Checkpointing

There have been a variety of approaches to dynamically adjusting the checkpoint interval [6, 12, 17, 22]. These techniques all employ an adaptive control mechanism to dynamically establish values for the checkpoint interval. As

with any control system, dynamically adjusting the simulators control parameters requires that several output values be monitored [1]. While the particular output values monitored by each proposed method vary, several of them are shared [6]. The WARPED system has the ability to choose from three algorithms to perform dynamic checkpointing. In particular, it includes implementations of Lin's model [12], Palainswamy's model [17] and Fleischmann's model [6]. For purposes of this paper, we show only results using Lin's model. Lin's model was chosen because it tends to produce slightly better results across the applications that are considered herein.

4 Comparative Results of the WARPED Algorithms

This section details the experiments conducted to characterize the performance and effectiveness of the optimization techniques in Section 3. To see the effect of the various optimizations presented earlier, a series of tests were conduced on two application domains, namely the queuing model simulations and simulation of digital systems described in the hardware description language VHDL [16, 18]. Performance results from these studies are reported using three chief measures: (i) total execution time (in seconds), (ii) total events processed per second, and (iii) total events committed per second.

The results in this paper have been collected by running the simulations on a 4-processor Sun SparcCenter 1000. In the reported results, the applications are tested with various configurations of LPs, namely: 1 (where it makes sense), 2, and 4. Also to illustrate performance on a parallel machine(the Intel Paragon), simulation results for the queuing model applications have also been included. The application description and performance results of these two applications are given, respectively, in the following two subsections.

4.1 Application: Queuing Model Simulation on the Sun SparcCenter 1000

Included in the WARPED distribution is a simulation library for building queuing models. This library is called *KUE*. In addition to the library, the distribution includes three instances of use of KUE. The three example queuing models are:

- Police: a queuing model of a simple traffic police telecommunications network). Instantiations of various sized models can be automatically generated. For purposes of this paper, a simulation model containing 96 Time Warp objects was used.
- SMMP: modeling a shared memory multiprocessor. Each processor is assumed to have a local cache with access to a common global memory. This application is also scalable, with instances generated by a C program. For the experiments in this paper, a 16 processor machine with settings as follows: a 10ns cache, a 100ns main memory, and a cache hit ratio of 90%.
- RAID: model of a nine disk RAID level 5 dik array of IBM 0661 3.5" 320Mb SCSI disk drives with flat-Left symmetric parity placement policy. Sixty processes send request for stripes of random lengths and location to forks which split each the requests into individual requests for each disk according to the placement policy. Thirty server process the requests in a FCFS fashion and route the appropriate requests back to the source process.

Simulation results from these models are summarized in 3–8 and discussed below. For comparison, the applications were also executed on the sequential implementation of the WARPED kernel. The event processing rates for the sequential kernel (computed in events per second) are: Police – 17,982; SMMP – 17,836 and, RAID – 16,908. Thus, in this version of *warped*, the configurations of the Time Warp kernel operate at about 50% of the efficiency of the corresponding sequential kernel.
All the optimizations are shown in each graph for each application. Optimizations are numbered 1 through 7. They are as follows:
1 No Optimizations with Round Robin Scheduling
2 No Optimizations with LTSF Scheduling
3 Periodic Checkpointing with LTSF Scheduling
4 Dynamic Checkpointing with LTSF Scheduling
5 Lazy Cancellation with LTSF Scheduling
6 Dynamic Cancellation with LTSF Scheduling
7 Dynamic Cancellation and Dynamic Checkpointing with LTSF Scheduling

Effects of the Scheduler

Two scheduling algorithms are available for testing, namely Round Robin (RR), and Lowest Timestamp First (LTSF). Both schedulers are written to remove inactive objects (objects with no events to be processed) from the ready to execute queue. Configurations of the queuing models for each of these scheduling algorithms (with no other optimizations enabled) are given in 3–8. While the overhead of sorting the events for LTSF scheduling produces a lower overall event processing rate, a faster execution time is actually achieved because the slowest advancing objects are given higher priority for execution. Comparing the data collected

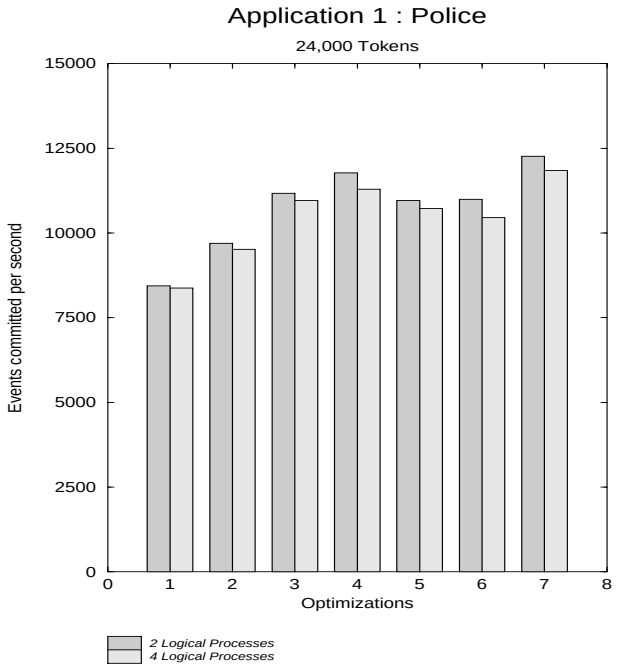

Figure 3. Sun SparcCenter 1000 results for Police

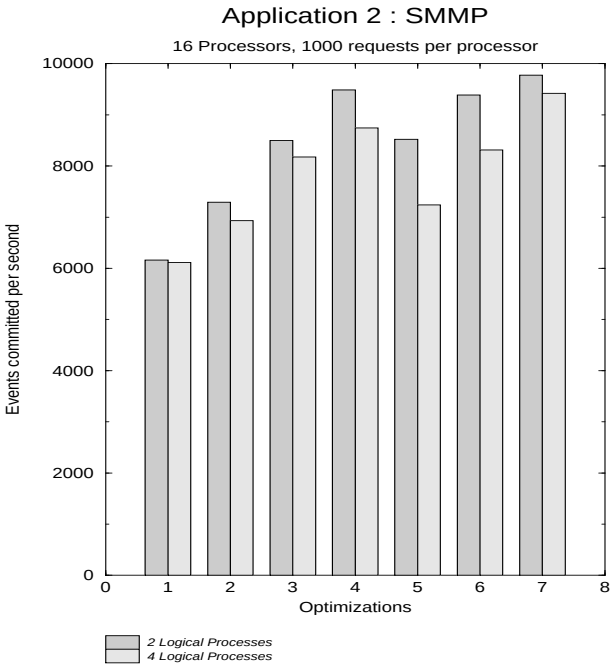

Figure 4. Sun SparcCenter 1000 results for SMMP

for the three applications, we can see that the LTSF scheduler gets a speedup of 10% over the Round Robin scheduler (with respect to committed events per second). Furthermore, preliminary studies (results not yet available) with data structures requiring less expensive sorting overheads (*e.g.*, lazy queue, calendar queue [4, 23]) have shown even greater performance improvements due to LTSF scheduling.

Effects of Periodic Checkpointing

For the (fixed) periodic checkpointing studies , a fixed period of 3 events between state saves was selected. This selection was chosen because it gives the best overall performance across all of the applications in the queuing model domain. As a result of the less frequent state saving, the execution time cost of state saving is reduced. The data collected for this optimization indicate that there is a 15% improvement in the number of committed events per second for at least two of the applications. A 15% improvement is achieved over the ordinary LTSF case. In some cases, a checkpoint interval of 2 or 4 produces even better results and in studies with the application of VHDL simulation, much larger checkpoint intervals (on the order of 10-20) give the best results. Consequently, the need for dynamic checkpointing.

Effects of Dynamic Checkpointing

Dynamic checkpointing periodically adjusts the checkpoint interval during the simulation in order to balance state savings costs against coast forward costs [6]. Two approaches

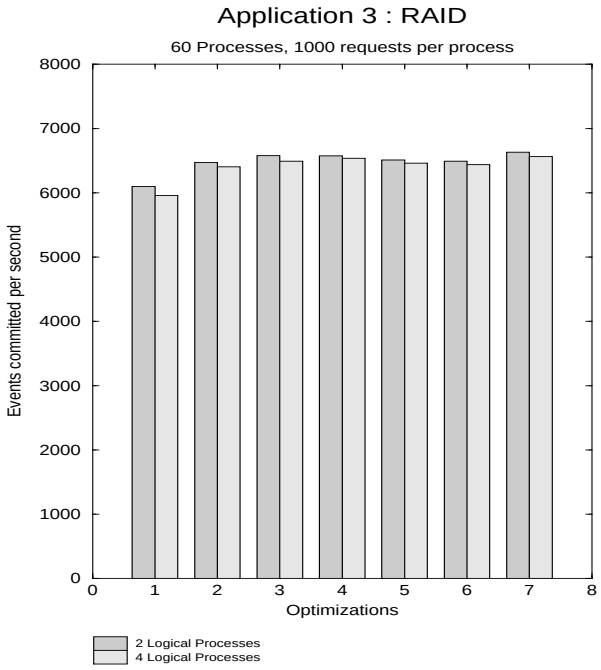

Figure 5. Sun SparcCenter 1000 results for RAID

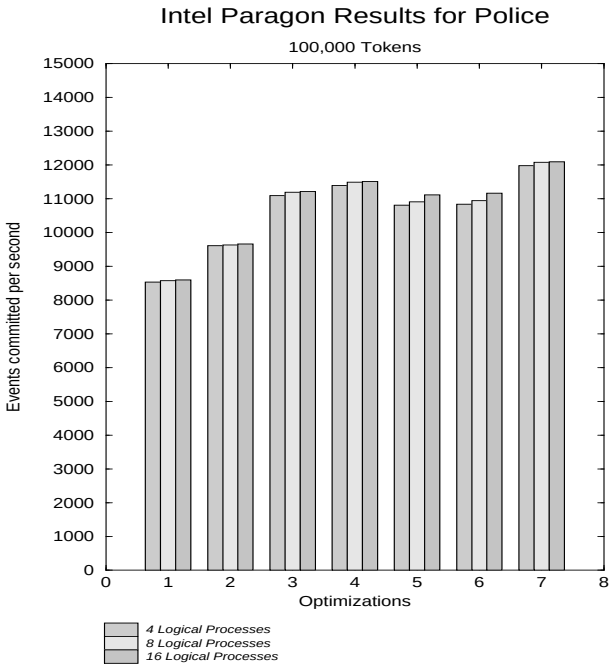

Figure 6. Intel Paragon results for Police

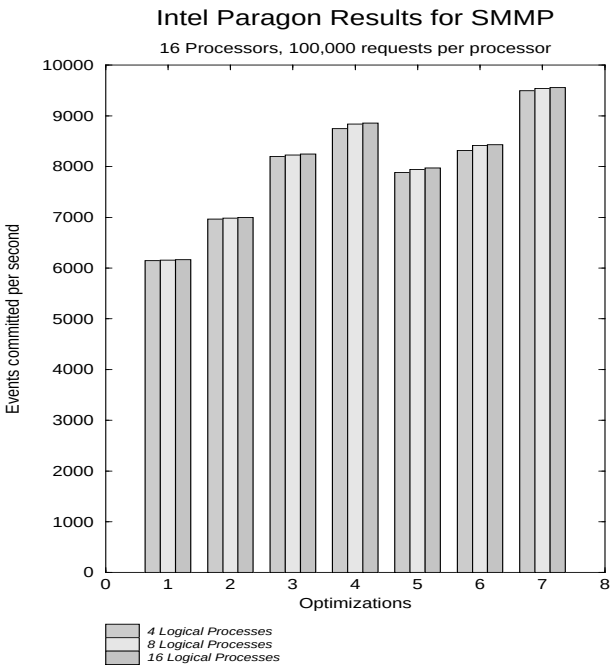

Figure 7. Intel Paragon results for SMMP

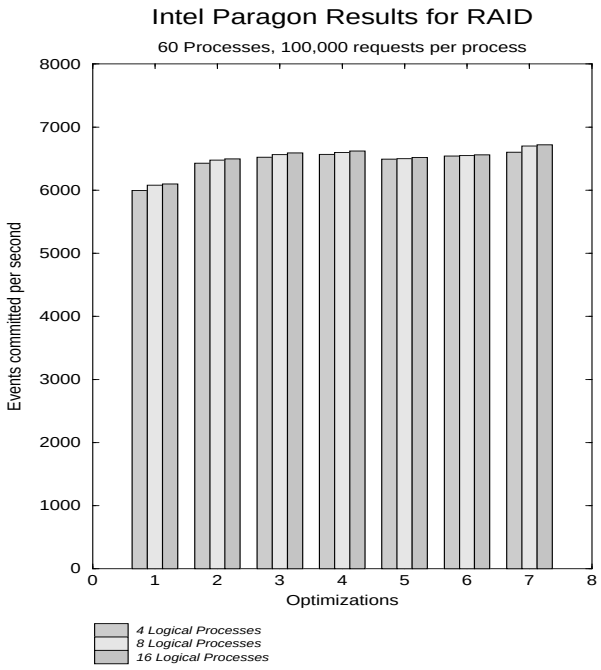

Figure 8. Intel Paragon results for RAID

have been considered in the literature. The first, explored by Lin et al [12], dynamically adjusts the checkpoint frequency at only the beginning of the simulation and leaves it fixed thereafter. The second approach is to continuously adjust the checkpoint period throughout the entire simulation [6, 17, 22]. The former has a lower total overhead and works well when the application reaches a steady state behavior. The latter works best when the application processing rates and rollback frequency vary over the lifetime of the simulation. For the queuing model studies, we have observed that Lin's model works best (3–8). From the graphs, we see that dynamic checkpointing gains a speedup of as much as 20% for two of the examples.

Effects of Lazy Cancellation

Lazy Cancellation is dependent on the number of rollbacks and how may of the rollbacks can be rolled forward. Given the queuing models of this study, lazy cancellation provides a 10% performance improvement over aggressive cancellation, in at least two of the applications(3 – 8).

Effects of Dynamic Cancellation

This optimization takes into consideration the fact that during the course of a simulation, it is necessary to apply aggressive cancellation at certain intervals of time and lazy cancellation during other intervals of time. Ideally, the dynamically selection should produce better execution times than either approach singlely applied. The data from the experiments support this claim. One of the applications gets a speedup of 25% whereas another gets a speedup of 13%.

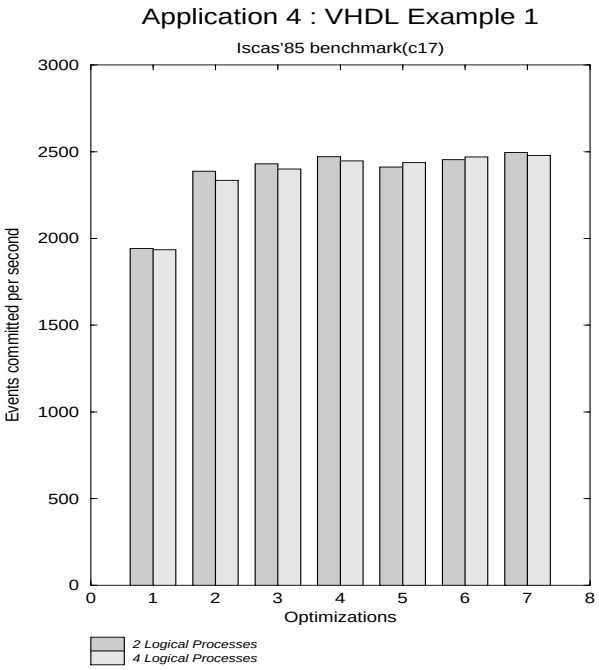

Figure 9. Sun SparcCenter 1000 results for c17

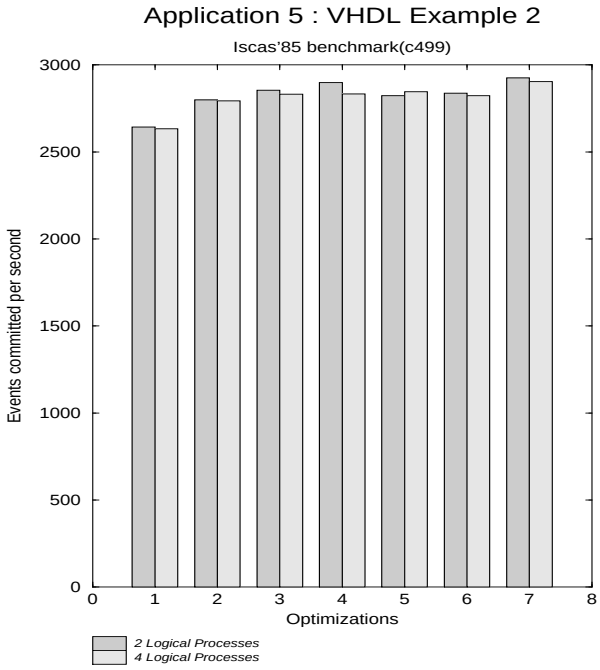

Figure 10. Sun SparcCenter 1000 results for c499

Effects of Dynamic Cancellation & Checkpointing

Performance results for these applications with LTSF scheduling, Dynamic Checkpointing, and Dynamic cancellation all enabled are summarized in 3 – 8. These optimizations produce an average speedup of 20% and clearly combine well to give the best overall performance for all of the parallel runs.

4.2 Application: Queuing Model Simulation on the Intel Paragon

The above 3 examples were simulated on the Intel Paragon, using a larger number of Logical Processes. It is clearly visible from the figures(6 – 8) that as we increase the number of processors, the overall committment rate improves as each processor has to do less. Consequently, the optimizations also show similar improvement when compared with the unoptimized case.

4.3 Application: VHDL Simulation

The second application studied is digital system simulation. In particular, we have developed a VHDL simulation kernel [9] that operates on top of the WARPED kernel. A VHDL code generator is currently under development. Consequently, we needed a simple technique for translating moderately sized VHDL examples into executable code to link with the VHDL simulation kernel. The ISCAS benchmarks provide a good example of moderately sized VHDL codes that are regular and easy to translate with a `perl` script.

Three examples from the ISCAS 85 (combinational circuits) and ISCAS 89 (sequential circuits) were selected for study. In particular, we chose the combinational circuits c17 and c499 and the sequential circuit s27. c17 contains 6 gates, c499 contains 202 gates, and s27 contains 11 gate elements. All of the tests were run with 10,000 input vectors. The results of these test cases are summarized in 9–11.

Effects of the optimizations on the VHDL Simulation Models

The results show similar performance gains as those discussed in Section 4.1. We see that LTSF scheduling gives a considerable performance improvement over RR; each input circuit show at least a 20% speedup. Periodic checkpointing does not produce as much performance improvement as it did in the queuing applications. However, dynamic checkpointing results in an additional 3% speedup over the improvement gained from LTSF scheduling. Not much speedup is gained from lazy cancellation or dynamic cancellation. Dynamic cancellation causes a performance improvement of 3 to 5%. But when dynamic cancellation and dynamic checkpointing are applied in conjunction to the VHDL simulation applications, a marked improvement in performance is obtained. In this case, all of the examples report at least a 5% improvement in performance.

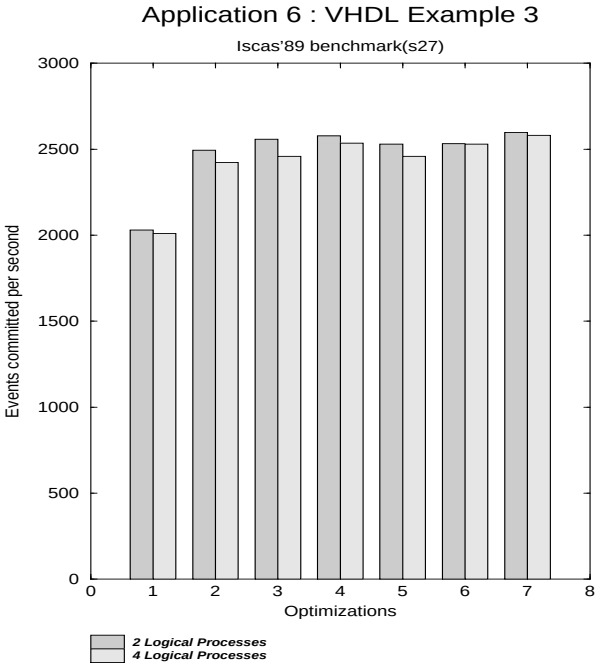

Figure 11. Sun SparcCenter 1000 results for s27

5 Conclusions

We discussed how the WARPED Time Warp kernel was extended to include dynamic selection of checkout intervals and event cancellation strategies. Empirical performance data was collected for two application domains, namely a queuing model application and a VHDL simulation application. Each application had three examples for simulation and the optimizations were performed on these examples. Lazy cancellation, dynamic cancellation, periodic checkpointing, and dynamic checkpointing were the prime focus of this paper. Performance results in terms of total execution time and event processing rates were tabulated and discussed for each Time Warp optimization discussed. In each case, the dynamic selection was shown to provide better performance and it was shown that the optimization with the best results was dynamic cancellation and dynamic checkpointing applied together.

Acknowledgments

The authors would gratefully like to acknowledge the suggestions and contributions of David Charley, John Penix, Dale E. Martin, Lantz Moore, Raghunandan Rajan, Balakrishnan Kannikeswaran, and Christopher H. Young.

References

[1] K. J. Astrom and B. Wittenmark. *Adaptive Control*. Addison Wesley, Reading, MA, 1989.

[2] D. Ball and S. Hoyt. The adaptive time-warp concurrency control algorithm. In *Distributed Simulation*, pages 174–177. Society for Computer Simulation, January 1990.

[3] J. V. Briner, Jr. *Parallel Mixed-Level Simulation of Digital Circuits using Virtual Time*. PhD thesis, Duke University, Durham, North Carolina, 1990.

[4] R. Brown. Calendar queues: A fast O(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, October 1988.

[5] K. M. Chandy and R. Sherman. Space-time and simulation. In *Distributed Simulation*, pages 53–57. Society for Computer Simulation, 1989.

[6] 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.

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

[8] A. Gafni. Rollback mechanisms for optimistic distributed simulation systems. In *Distributed Simulation*, pages 61–67. Society for Computer Simulation, January 1988.

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

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

[11] Y. Lin. Estimating the likelihood of success of lazy cancellation in time warp simulations. *International Journal in Computer Simulation*, 1995. (to appear).

[12] Y.-B. Lin, B. R. Preiss, W. M. Loucks, and E. D. Lazowska. Selecting the checkpoint interval in time warp simulation. In *Proc of the 7th Workshop on Parallel and Distributed Simulation (PADS)*, pages 3–10. Society for Computer Simulation, July 1993.

[13] D. E. Martin, T. McBrayer, and P. A. Wilsey. WARPED: A time warp simulation kernel for analysis and application development, 1995. (available on the www at `http://www.ece.uc.edu/~paw/warped/`).

[14] D. E. Martin, T. J. McBrayer, and P. A. Wilsey. WARPED: A time warp simulation kernel for analysis and application development. In *29th Hawaii International Conference on System Sciences (HICSS-29)*, January 1996. (forthcoming).

[15] J. Misra. Distributed discrete-event simulation. *Computing Surveys*, 18(1):39–65, March 1986.

[16] Z. Navabi. *VHDL: Analysis and Modeling of Digital Systems*. McGraw–Hill, New York, NY, 1993.

[17] A. Palaniswamy and P. A. Wilsey. Adaptive checkpoint intervals in an optimistically synchronized parallel digital system simulator. In *VLSI 93*, pages 353–362, September 1993.

[18] D. L. Perry. *VHDL*. McGraw–Hill, New York, NY, 2nd edition, 1994.

[19] R. Rajan. Cancellation strategies in time warp simulators. Master's thesis, Dept of ECECS, University of Cincinnati, Cincinnati, OH, December 1995. (expected).

[20] R. Rajan and P. A. Wilsey. Dynamically switching between lazy and aggressive cancellation in a time warp parallel simulator. In *Proc. of the 28th Annual Simulation Symposium*, pages 22–30. IEEE Computer Society Press, April 1995.

[21] P. L. Reiher, F. Wieland, and D. R. Jefferson. Limitation of

optimism in the time warp operating system. In *Winter Simulation Conference*, pages 765–770. Society for Computer Simulation, December 1989.

[22] R. Rönngren and R. Ayani. Adaptive checkpointing in time warp. In *Proc. of the 8th Workshop on Parallel and Distributed Simulation (PADS 94)*, pages 110–117. Society for Computer Simulation, July 1994.

[23] R. Rönngren, J. Riboe, and R. Ayani. Lazy queue: An efficient implementation of the pending-event set. In *Proc. of the 24th Annual Simulation Symposium*, pages 194–204, April 1991.

[24] D. West. Optimizing time warp: Lazy rollback and lazy reevaluation. Master's thesis, University of Calgary, Calgary, Alberta, 1988.