This paper appeared in the Proceedings of the 9th Workshop on Parallel and Distributed Simulation, PADS-1995.
© 1995, IEEE. Personal use of this material is permitted. However, permission to reprint or republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

Comparative Analysis of Periodic State Saving Techniques in Time Warp Simulators*

Josef Fleischmann
Institute of Electronic Design Automation
Technical University of Munich
D-80290 Munich, Germany
jsf@regent.e-technik.tu-muenchen.de

Philip A. Wilsey
Center for Digital Systems Engineering
Dept of ECECS, PO Box 210030
Cincinnati, Ohio 45221-0030
phil.wilsey@uc.edu

Abstract

Checkpointing in a time warp synchronized parallel simulator is a necessary and potentially expensive operation. In the simple case, a time warp simulator checkpoints every χ events, for some fixed value χ. For larger values of χ, the simulator requires less overhead for saving the state, but incurs an increased latency during rollback. Thus, the problem is to balance the time to save states against the time to coast forward upon rollback. Unfortunately, a static determination of a optimal value for χ is very difficult and can vary widely, even between closely related instances of a time warp simulator. Furthermore, the optimal checkpoint interval may actually vary over the lifetime of the simulation.

To address these problems, several investigators have proposed dynamically adjusting the checkpoint interval χ as the simulation progresses. This paper analyzes three previous techniques for dynamically sizing checkpoint intervals and presents a new, heuristic algorithm for this purpose. All four techniques are implemented in a common application domain (digital system simulation from VHDL descriptions) and a direct comparison between the algorithms is performed. The results show a significant difference in the performance of the implemented algorithms. However, in virtually all cases, the dynamic algorithms performed near or better than the best static value. Furthermore, the best algorithms performed as much as 12% better than the best static value.

1 Introduction

The time warp mechanism, because of its relaxed synchronization criteria, is an appealing technique for controlling parallel discrete event driven simulators. Time warp parallel simulators trade the overhead of enforcing strict time-ordered event processing for the overhead of recovery from out-of-order event processing [8, 11]. Thus, instead of forcing time-ordered event processing, time warp relaxes the synchronization scheme so that out-of-order event processing is possible (although in the desirable case, infrequent). The chief overheads in time warp needed to allow recovery from out-of-order event processing are the (i) processing time for saving state and event information, (ii) memory space for the saved event and state information, (iii) processing time for computing a global minimum time of the simulation (the so-called *Global Virtual Time, GVT*), and (iv) the cost of reclaiming old memory consumed by saving state and event information (*i.e., fossil collection*).

The benefits of the time warp approach is that it can potentially uncover higher degrees of parallelism in the system being simulated. This may lead to a speedup even if a certain amount of the "lookahead" computation is wasted due to rollbacks (recovering from out-of-order event processing). On the other hand, one obvious drawback of time warp lies in the extra overhead. One of the most significant costs in supporting rollback is the cost of state saving and state restoration. Several suggestions have been made to reduce this overhead. In particular, the solutions can be broadly classified into the following three categories: hardware accelerators [9], saving only incremental changes in the state (called *incremental state savings*) [3, 2], and reducing the frequency of state saving *called periodic checkpointing* [4, 13]. While the hardware solution can produce dramatic performance improvements, the requirement of extra hardware for simulation is not always acceptable. The two software solutions can be more broadly applied. In applications where a large fraction of the process state is modified with every simulation step, incremental state saving is less promising than periodic checkpointing [6, 16]. However, the latter solution requires the establishment of a value for the number of events to be processed between state saves (this value is called the *checkpoint interval*).

In general, establishing a static value for the checkpoint interval that produces optimal performance is difficult [15]. Furthermore, for many applications, the optimal value for the checkpoint interval is likely to vary over the lifetime of the simulation [7]. In fact, several researchers have proposed distinct solutions for dynamically (during the simulation execution) determining an operating checkpoint interval [14, 16, 18]. This paper presents a comparative analysis of four approaches to dynamically

*This work was partially supported by the Advanced Research Projects Agency and monitored by the Department of Justice under contract number J-FBI-93-116.

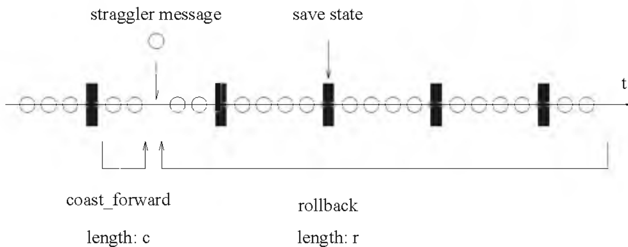

Figure 1: Rollback and Coasting Forward ($\chi = 4$)

adjusting the checkpoint interval. In particular, three previously developed approaches that derive their checkpoint interval adjustment strategy from analytical models are compared with a heuristic method developed in this paper. The results indicate that in one instance of a time warp simulator (simulations formed from descriptions written in the hardware description language VHDL [17]), significant speedup can be obtained using dynamically adjusted periodic checkpointing. However, significant differences between the effectiveness of the algorithms compared were observed, with the heuristic method performing best.

The remainder of this paper is organized as follows. Section 2 briefly reviews the basic tradeoffs in determining the optimal checkpoint interval. Section 3 reviews previous work to dynamically establish an operating checkpoint interval. A new heuristic algorithm for dynamic checkpointing is presented in Section 4. Section 5 introduces the method used to analyze the performance results. Empirical results comparing the relative performance of each implemented algorithm are shown in Section 6. Finally, Section 7 contains some concluding remarks.

2 Optimizing the Checkpoint Interval

In general, distinct LPs within a specific application to be simulated show different characteristics regarding rollback frequency, state size, and granularity of events to be executed. Furthermore this behavior may change over the lifetime of the simulation. Because of this disparate and dynamic nature of LPs, it is difficult to establish a fixed value for the optimal checkpoint interval that produces optimal performance. A reasonable alternative to statically defining the checkpoint interval is to dynamically (during simulation) adjust the period for each individual LP. Of course, the dynamic frequency for which this calculation need occur may vary significantly based on the time-varying behavior of the application. Likewise, the overhead of the method through which the checkpoint interval is dynamically established may influence the utility of recalculation.

The cost of checkpointing must be balanced against the frequency of rollback and the cost of reprocessing events without intervening checkpoints. For example, Figure 1 illustrates a process rolling back upon receipt of a straggler message. The circles denote events, the filled boxes denote the action of saving the process state. Since checkpointing does not occur after every processed event, a rollback may have to reexecute several intermediate events before actually processing the straggler event. This reexecution is called the *coast forward* phase. Thus, the value of the time-optimal checkpoint interval reflects a tradeoff between two costs which adversely affect the performance of an LP. In particular:

- Increasing the checkpointing interval reduces the overall time consumed by the state saving routine and the memory space used by the state copies.
- Increasing the checkpoint interval results in an increase in the average number of events reexecuted during the coast forward phase. This stems from the fact that the average length of coasting forward c is proportional to the checkpoint interval.

In our application domain, time is a more valuable resource than space. Hence, we concentrate on minimizing the execution time. However, there is a similar tradeoff for a space-optimal checkpoint interval. And a complete disregard for space costs is also not feasible. More precisely, while sparse checkpointing reduces the amount of memory allocated for state copies, a side effect of large checkpoint intervals is a corresponding increase in the size of the input event queue. The larger the checkpoint interval, the more input events have to be maintained in order to reconstruct intermediate states. This means that, even if a process shows no rollbacks, the checkpoint interval cannot be set to infinity because of space consumption by the input queue.

3 Adaptive Checkpointing Algorithms

There have been a variety of approaches to dynamically adjusting the checkpoint interval. As with any control system, dynamically adjusting the simulator's control parameters requires that several output values be monitored [1]. While the particular output values monitored by each method vary, several of them are shared, such as the average time to save the process state (δ_s), the average time to execute one event (δ_e) and the number of rollbacks. All of the adaptive approaches measure characteristic data for each individual process during a certain period of time, and then recalculate the optimal checkpoint interval. This period between recalculations of the checkpoint interval is referred to as the *observation cycle*.

In the next few subsections, three particular techniques for dynamically adjusting the checkpoint interval are examined [14, 16, 18]. Each of these approaches derives an analytical model of the time warp simulator and uses the model to derive a formula for dynamically establishing values for the checkpoint interval. Unfortunately each approach results in a fairly complex formula that may require significant processing time for evaluation and data capture. Consequently, in addition to the previously devel-

oped methods, a heuristic algorithm for dynamically calculating checkpoint intervals is presented in Section 4.

3.1 Lin's Model

Lin *et al* [14] derive a sophisticated analytical model for establishing upper and lower bounds on the optimal checkpoint interval. This work extends an earlier model that was developed by Lin and Lazowska in 1989 [13]. Essentially, the derivation establishes an accurate upper and lower bound for the execution time overhead that checkpointing imposes on an LP. The optimal checkpointing period is thus found according to these two bounds. Formally, the lower (χ^-) and upper (χ^+) bounds are:

$$\chi^- = \left\lceil \sqrt{\frac{\delta_s}{\delta_e}(\bar{\alpha}-1)} \right\rceil \quad \leq \chi \leq \quad \left\lceil \sqrt{\frac{\delta_s}{\delta_e}(2\bar{\alpha}+1)} \right\rceil = \chi^+.$$

In this formula, $\bar{\alpha}$ denotes the average number of process executions between two subsequent rollbacks, excluding events being coasted forward.

Based on empirical studies the authors suggest choosing the upper bound χ^+ as the checkpoint interval. Furthermore, in order to avoid too many potentially expensive recalculations, the optimal interval size is adjusted only until the simulation reaches a "steady state." In several experiments, the authors find that the regulating algorithm terminates after only a few recalculations.

3.2 Palaniswamy's Derivation

Palaniswamy [15, 16] begins with a model similar to Lin's [13]. Palaniswamy estimates the cost of a rollback due to periodic state saving. That is, assuming that a large number of events are committed, he models the average overhead involved during a given time interval. Minimizing this overhead function leads to the following expression for calculating the optimal value χ_{opt}:

$$\chi_{opt} = \left\lceil \sqrt{2\frac{\delta_s}{\delta_e}\left(\frac{N}{k_r}+\gamma-1\right)} \right\rceil. \qquad (1)$$

where the variables N, k_r and γ are the following output values that must be metered:

- N: number of committed events,
- k_r: numger of rollbacks, and
- γ: mean value of the rollback length.

This approach relies on the observation of committed events and thus it is dependent on updates to GVT calculations (because events are committed only when GVT advances). Thus, the frequency of updating the checkpoint interval in this algorithm cannot be higher than the frequency of GVT computations.

3.3 Rönngren's Approach

Rönngren [18] attempts to minimize the overhead involved in processing R_{obs} events, where R_{obs} is the total number of events executed minus the events executed during the coasting forward phase. The algorithm is computed as follows. If k_r is the number of rollbacks that occur during an observation period, then the optimal checkpoint interval is computed as:

$$\chi_{opt} = \left\lceil \sqrt{2\frac{\delta_s}{\delta_c}\frac{R_{obs}}{k_r}} \right\rceil. \qquad (2)$$

If the event execution time during the coasting forward phase, δ_c, is approximately equal to the normal event execution time, δ_e, then we can rewrite Equation 2. More precisely, consider the fact that the number of process executions R_{obs} equals the number of committed events N plus the number of events undone due to rollbacks:

$$R_{obs} = N + k_r\gamma. \qquad (3)$$

Then Equation 2 transforms into

$$\chi_{opt} = \left\lceil \sqrt{2\frac{\delta_s}{\delta_e}\left(\frac{N}{k_r}+\gamma\right)} \right\rceil. \qquad (4)$$

This result is only marginally different from Palaniswamy's solution, Equation 1, and leads to a slightly larger checkpoint interval.

4 Deriving a Simple Heuristic

As previously discussed, the analytically based algorithms for sizing checkpoint intervals have resulted in formula that are costly to evaluate and monitor. In this section we present a simple and easy to implement method for dynamically sizing checkpoint intervals. This is a heuristic and it has been developed after extensive profiling and analysis of the operation of a locally developed digital system parallel simulator. Some of this analysis is shown below. Unfortunately space limitations prevent a more detailed discussion; interested readers should see [7] for additional details. The heuristic algorithm pursues a regulating mechanism similar to the method of *supervisory control* in control theory [5].

4.1 The Overhead of State Saving

The time warp optimization of Periodic State Saving (PSS) attempts to balance two types of overhead. That is, to be effective, PSS must balance the time spent saving state against time spent coasting forward. More precisely:

Saving States: Periodic State Saving significantly reduces the amount of time lost on this operation. The observed costs for saving state decrease monotonically with the checkpoint interval (Figure 2).

Coasting Forward: Reexecuting intermediate events between the last saved state before the rollback point and the straggler message causing the rollback consumes processor time. This computational costs increase monotonically with the period of checkpointing (Figure 3).

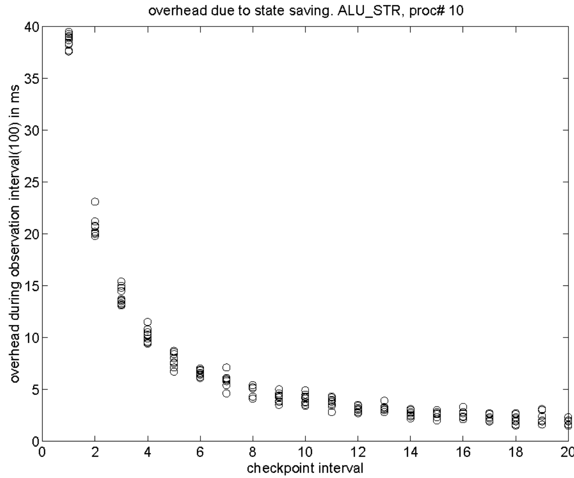

Figure 2: State saving costs vs. period

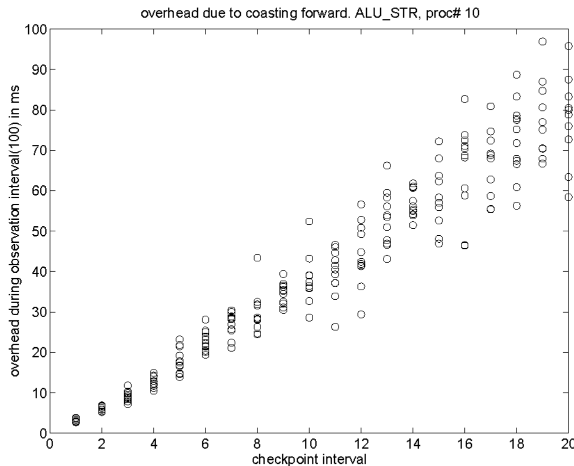

Figure 3: Coasting forward costs vs. period

Figures 2 and 3 show the costs for state saving C_{SS} and coasting forward C_{CF} in terms of cpu time. The sum of these costs gives the value E_c of our *cost function* for periodic state saving:

$$E_c = C_{SS} + C_{CF}. \tag{5}$$

In Figure 4 a typical behavior of the filtered[1] cost function with respect to the checkpoint interval is shown. The checkpoint interval is to be chosen in a way that the overhead is minimized. In our example the minimum of the cost function clearly indicates the optimal period $\chi = 4$.

[1] Where necessary, measured data is filtered using low-pass filtering techniques in order to reduce statistical noise. In particular, throughout the experiments described in the remainder of this paper, first order IIR filters with coefficients $\alpha_0 = .4$ and $\alpha_1 = .6$ have been used.

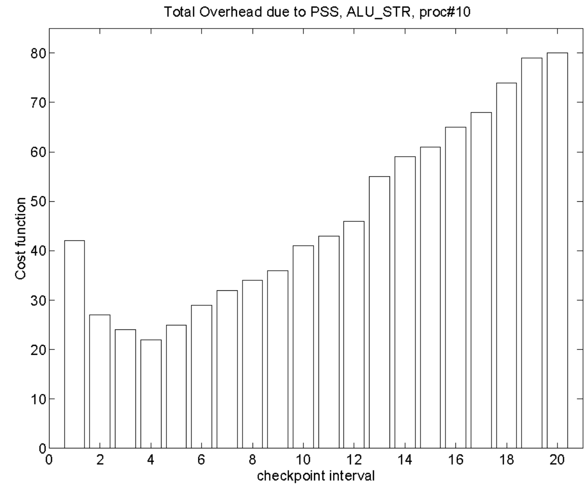

Figure 4: Evaluating the cost function

4.2 Regulating Algorithm

The goal is to find the location of the minimum and to adjust the checkpoint interval correspondingly. According to the empirical results obtained from a variety of benchmarks we based the adjusting algorithm on the following assumptions:

1. The cost function is monotonously increasing for checkpoint intervals larger than the optimal period, except for a process without any rollbacks.
2. The cost function shows a minimum. The location of this minimum indicates the optimum checkpoint interval. The optimal checkpoint interval may be equal to or greater than one ($\chi_{opt} \geq 1$).

We recalculate the period after every N events executed. In [18] guidelines are given for choosing N. (In our experiments we used $N = 100$ events.) Initially a checkpoint interval of one is used ($\chi_{initial} = 1$). In the successive observation intervals (intervals over which the cost function E_c is reevaluated) the checkpoint interval is incremented by one if E_c (as measured during the last observation interval) did not significantly increase. If the costs in the current observation cycle become greater than in previous cycles, we change our "adaptation direction" and decrement (by one) the checkpoint interval.

To prevent our resulting period from oscillating we use an appropriate threshold before adjusting the checkpoint interval. Furthermore, if the process shows no rollbacks (over its complete history), the checkpoint interval is set to χ_{max}. (In accordance with empirical observations we use $\chi_{max} = 30$.) The ability to adapt is maintained, as an old value of the cost function is stored and periodically compared to the current value. If the system behavior changes over time, the checkpoint interval changes as well. Despite its simplicity, the performance results in Section

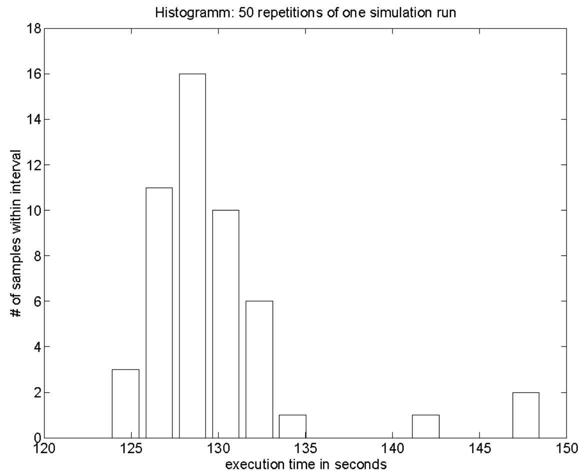

Figure 5: Distribution of execution times

6 show that this is an efficient technique for dynamically sizing checkpoint intervals.

5 Analyzing Performance Data

Before reporting the results of our comparative analysis, we first address the problem of analyzing simulation performance figures to obtain verifiable results. Performance analysis is a nontrivial task, and it is difficult to characterize the performance of a simulation in any specific configuration as a single number. In general, repeating a simulation several times gives different results. We often see varying results in terms of execution time, number of rollbacks, and number of simulation cycles. In the following subsections, we detail the analysis process followed in this investigation. This analysis draws heavily from the discussions found in [10, 12].

5.1 Distribution of Performance Data

Figure 5 depicts the distribution of the execution times of 50 simulation runs for an identical configuration and environment. We would assume the data to be normally distributed. While the histogram undoubtedly bears some resemblance to a normal distribution, this observation can only be derived by discarding some of the outlying samples at the upper end of the range. Thus, in our analysis these samples are treated as *outliers*, since they are far away from the majority of the other observation points. The reason for this abnormally "slow" simulation runs may either be found in the non-deterministic nature of Time Warp or in the nature of our implementation, where every LP is implemented as a *lightweight process* (LWP) on an SMP (symmetric multiprocessor) SUN SparcCenter 1000 workstation.

Because of resource constraints, it is not possible to perform a sufficiently high number of simulation repetitions to prevent these outliers from disturbing the mean value and variance of the performance data. Therefore, such obvious outliers will be discarded in the analysis of Section 6.

5.2 Confidence Intervals

Comparing two sets of identically configured simulation runs, where each set consists of N_r repetitions the following observation is made: These two sets differ both in their mean value and in their variance. Thus, the fundamental concept of *confidence intervals* has to be taken into account. From a limited number of samples we can estimate the actual mean value μ only at a certain *confidence level.* Formally we write:

$$Prob(c_1 \leq \mu \leq c_2) = 1 - \alpha. \qquad (6)$$

where the interval $[c_1, c_2]$ is called the *confidence interval* with respect to a confidence level of $100(1-\alpha)\%$. Thus, α is called the level of significance. The confidence intervals depicted in Section 6 are obtained using a normal probability distribution.

5.3 One Factor Analysis

One Factor Analysis is a method suitable for comparing "several alternatives of a single categorical variable" [10]. In our case the categorical variable is the type of the algorithm used for computing and eventually adjusting the checkpoint interval. The following section gives a short description of this statistical method. More detailed information may be found in [7, 10].

Regression Modeling

In a single factor design we model the actual value of the obtained data point y_{ij} as follows:

$$y_{ij} = \mu + \alpha_j + e_{ij} \qquad (7)$$

where y_{ij} is the ith measured data point using algorithm j. It can be split into a sum of the mean response μ, the effect of the specific algorithm α_j, and the statistical error e_{ij}.

Performance data is measured by running r repeated simulations for all a different algorithms. Applying our model equation (7) gives a set of ar equations and leads to:

$$\mu = \frac{1}{ar} \sum_{i=1}^{r} \sum_{j=1}^{a} y_{ij}. \qquad (8)$$

Thus, μ is the *grand mean* of all samples. The difference between the mean value $\bar{y}_j$ of a specific alternative j and the grand mean are described by the *effect* α_j :

$$\frac{1}{r} \sum_{i=1}^{r} y_{ij} = \bar{y}_j = \mu + \alpha_j. \qquad (9)$$

Analysis of Variance

The total variation we can observe in the measured sample can be attributed to both the *effects* α_j of the different algorithms and the statistical errors. In a One-Factor-

Analysis this variation is calculated as a squared sum of errors (SSE) and of the effects (SSA), which sums to the total variation observed (SST).

An F-test can be computed on the ratio of the mean square of errors and the effects. The resulting F-value (F-computed) is to be compared to the $(1-\alpha)$-quantile of the F-variate (F-table). This helps decide whether one algorithm performs significantly better or worse than another. If F-computed is greater than F-table, then a decision is possible at a given confidence level $(1-\alpha)$.

6 Experiments and Results

The simulation experiments described in this section were conducted on a 4-processor SMP SparcCenter 1000. The LPs execute as lightweight threads using the SOLARIS 2.3 threads library and the processes communicate by event messages in shared memory. An aggressive cancelation strategy was used and GVT was calculated using Samadi's Algorithm [19].

6.1 Implemented Algorithms

Following implementations for dynamically adjusting the checkpoint interval were studied:

Method 1: Lin's model [14].

Method 2: Palaniswamy's derivation, recalculation every GVT-cycle, times for saving state and executing events measured across the complete simulation history [16].

Method 3: Palaniswamy's model with less frequent recalculations; recalculation is performed in a new GVT cycle only if a minimum of N new events have been processed since the last update.

Method 4: Rönngren's model [18].

Method 5: The heuristic cost function model described in Section 4 of this paper.

For all analytical methods (except Method 2) the averages δ_s and δ_e were measured only at the very beginning of the simulation and assumed as constants. The simulation results are obtained from following VHDL descriptions:

`ALU_BEH`: arithmetic logic unit, 6 processes.

`PARMULT16`: 16 bit parallel multiplier, 21 processes.

`ARRAYMULT`: 4 bit array multiplier, 53 processes.

`ADDER16`: 16 bit adder, 85 processes.

6.2 Overhead

As previously mentioned, all of the above mentioned algorithms monitor output values of the *Logical Processes* at runtime. Furthermore, the frequency at which dynamic adjustments are made to the checkpoint interval directly impacts processor execution time overhead. But how significant are these computational costs? There are basically two ways to determine the influence of the additional computation required. First, the additional calculations for dynamic checkpointing can be isolated and their execution time measured. Second, the dynamic computations can be turned on or off with a fixed checkpoint interval as the simulation executes and the differences compared. In these experiments, the second alternative was used because it more accurately reflects the execution time costs of the dynamic adjustment.[2] Thus, we ran simulations using the (previously determined) ideal static checkpoint interval. Running the simulation without any monitoring (*plain method*) gave a base time. Then, we ran the same simulations applying the various methods, but ignoring the computed checkpoint intervals and still using the ideal static checkpoint interval. The additional time taken by the adaptive methods in these runs is an estimate for the overhead of the individual algorithms. The results of our experiments with the example `ADDER16` are summarized in Table 1. The conclusion we draw from the resulting confidence intervals is that the overhead for all implementations besides Method 2 is comparably low ($< 1\%$) and almost disappears within the statistical fluctuations of the execution time observed.

Method	execution time (s)	confidence interval		estimated overhead
plain	131.799	130.789	132.816	0.000
Method 1	132.312	131.294	133.329	0.513
Method 2	134.375	133.357	135.393	2.576
Method 3	132.785	131.767	133.802	0.986
Method 4	132.300	131.283	133.318	0.502
Method 5	132.808	131.791	133.826	1.010

Table 1: Overhead

6.3 Performance

The Analysis of Variance (ANOVA) [10] helps to determine the significance of the differences in performance for each algorithm studied. The ANOVA results for the `PARMULT16` experiments (Figure 6) are given in Table 2. These results indicates that F-Computed is significantly greater than F-table. Thus, we can make a decision at a 95%-confidence level. The *effects* (the difference in performance between the individual algorithms and the overall mean) of Method 2 (Palaniswamy's) and Method 5 (Cost function) differ by almost five percent. The confidence intervals for the effects are given in the last two columns of Table 2. The remaining three Methods (1, 3, and 4) show similar confidence intervals, *i.e.,* indicating only a small performance difference for this benchmark.

A visual depiction of the mean execution times and confidence intervals for our benchmarks is given in Figures 6, 8, 9 and 10. The x-axis indicates the Method number assigned in subsection 6.1. For `PARMULT16`, Method 2 (Palaniswamy) performs significantly worse than the other models. This seems to be due to its amount of overhead, as Method 3, a less expensive implementation of the same

[2] That is, it more accurately captures the real cost of the overhead including, for example, cache misses and page faults.

	Sum Of Squares	Percentage of Var	Mean Square	F-Computed	F-Table
SSY	1931525.8				
SS0	1930336.4				
SST	1189.4	100.0			
SSA	418.6	35.2	104.6	14.3	2.6
SSE	770.8	64.8	7.3		

Parameter	Mean Effect	Standard Deviation	Confidence Interval	
μ	132.4708	0.2583	131.9575	132.9841
01	-0.4191	0.5167	-1.4457	0.6076
02	3.7201	0.5167	2.6935	4.7468
03	-0.3732	0.5167	-1.3999	0.6534
04	-0.9223	0.5167	-1.9489	0.1044
05	-2.0056	0.5167	-3.0322	-0.9789

Table 2: ANOVA-Table, PARMULT16

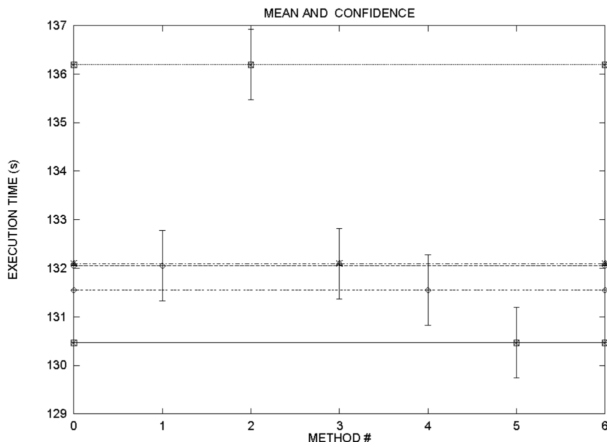

Figure 6: Comparison of the Methods: PARMULT16

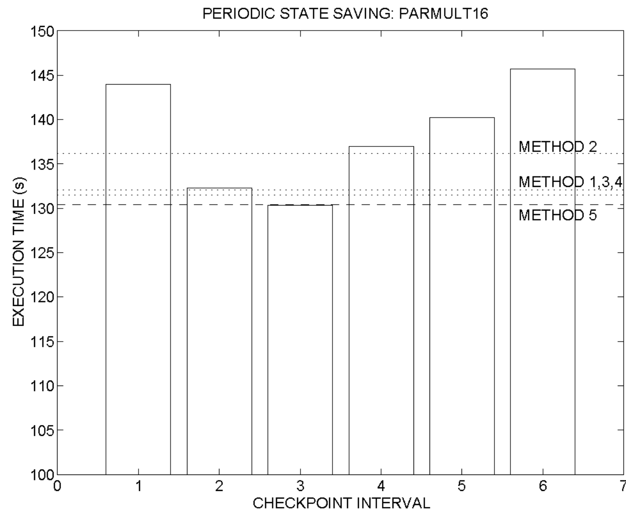

Figure 7: Performance of adaptive and static PSS

algorithm, gives significantly better speedup. The new heuristic algorithm developed in Section 4 (Method 5) performs best in this experiment.

A comparison of the performance data in Figure 6 to the execution times obtained with different static checkpointing intervals (Figure 7) shows that near optimal speedup can be obtained. The speedup compared to the static checkpointing increases significantly if the optimal period changes over simulation time and differs for the individual LPs [7].

Figure 8 shows results for the `ARRAYMULT` example. As with `PARMULT16`, Methods 4 and 5 result in the best speedup with Method 2 performing the worst.

A quick examination of the remaining experiments (Figures 9 and 10) also report good performance for Methods 4 and 5 (with 5 performing much better with the smallest example). In general, the performance of Lin's Method (Method 1) depends on the simulated application. In contrast, the other methods show fairly consistent behavior for the simulated examples. Thus, supporting our belief that a continuously adjusting algorithm is needed; the continual adjustment is useful because the optimal checkpoint interval changes over the lifetime of the simulation (LP behavior is dynamic over the lifetime of the simulation). This belief may not hold for different application domains.

6.4 Memory Usage

As already mentioned, this analysis is focussed on the performance in terms of execution time. Nevertheless, we have also observed that the distinct algorithms also differ with regard to memory consumption. Thus, we measured the maximum length of the state queue for all LPs in each of the simulations. The maximum state queue length is rather a vague indicator of the memory consumption than

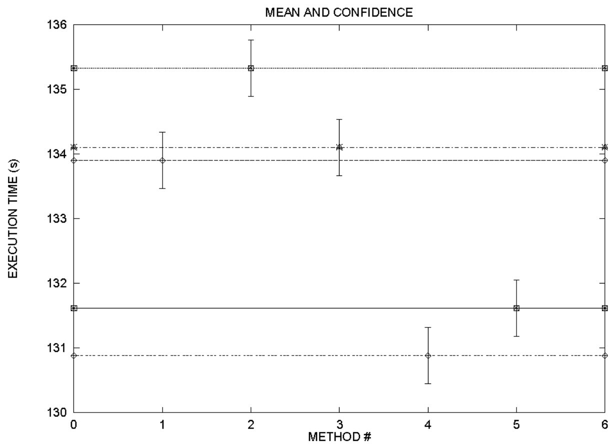

Figure 8: Comparison of the Methods: ARRAYMULT

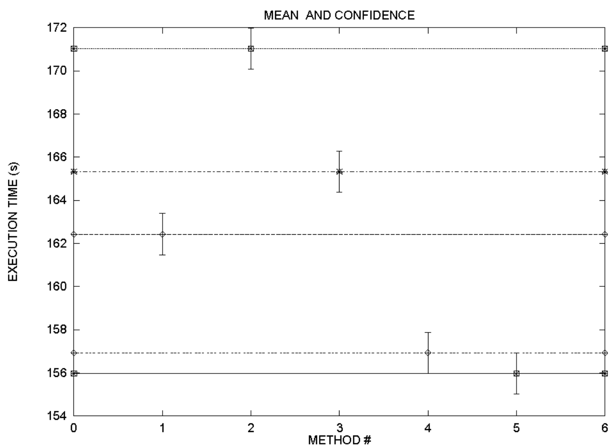

Figure 9: Comparison of the Methods: ADDER16

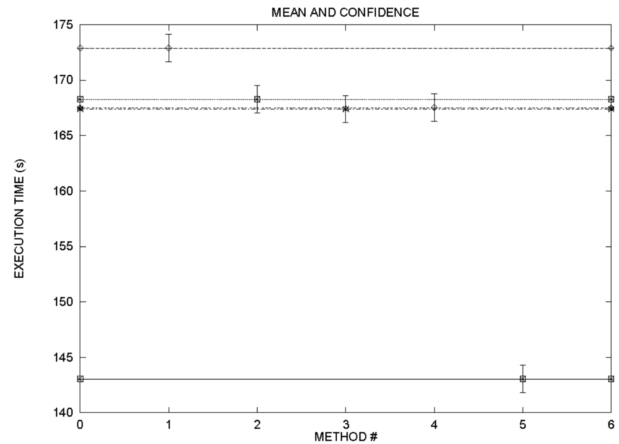

Figure 10: Comparison of the Methods: ALU_BEH

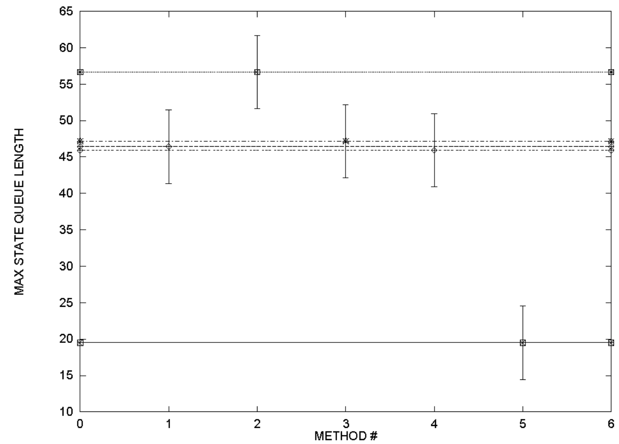

Figure 11: Maximum State Queue Length: ADDER16

a precise measure, but it is the only (somewhat meaningful) measure that we can report.

Figure 11 represents the typical behavior of the different algorithms with respect to memory usage. The values shown were obtained for the ADDER16 benchmark, they are qualitatively similar for the other benchmarks. The heuristic algorithm from this paper, Method 5, performs significantly better than any of the analytical methods (Method 1 – 4).

7 Conclusions

In a simulation where all the different LPs show similar behavior with little fluctuation over simulation time, an optimal execution time can be reached by applying periodic checkpointing with a fixed checkpoint interval. However, in most cases, the optimal static checkpoint interval cannot easily be determined at compile time. Furthermore, in the common case many LPs show different and even time-varying behavior. Thus, an adaptive checkpointing technique is essential. In this paper we compared several possible derivations for optimal checkpointing. We found that adaptive techniques are effective for speeding parallel simulations with VHDL description as the application domain. Furthermore we emphasized the need for statistical analysis in order to compare the different checkpointing algorithms. Our results indicate, that the computational overhead for the suggested methods can be reduced to a comparably low level. But nevertheless, some of the benchmarks reveal significant differences in their resulting performance.

For the new heuristic algorithm presented in this paper, we estimate that the additional overhead is less than one percent. In the worst case, the new heuristic adaptive method performs as well as the optimal static checkpointing method. In the best case (ALU_BEH), we outper-

form optimal static checkpointing by more than 12% which means a speedup of almost 20% over the simulation without sparse checkpointing. The analysis and the performance results presented help decide which algorithm to favor in an actual implementation for the sake of optimizing the performance of the time warp simulator in the domain of digital system simulation. Future work is to evaluate the different checkpointing methods on larger platforms and to empirically compare them to incremental state saving in our application domain. Also a more detailed investigation with regard to memory consumption of the different algorithms is necessary.

References

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

[2] Bauer, H., and Sporrer, C. Reducing rollback overhead in time-warp based distributed simulation with optimized incremental state saving. In *Proc. of the 26th Annual Simulation Symposium* (April 1993), Society for Computer Simulation, pp. 12–20.

[3] Bauer, H., Sporrer, C., and Krodel, T. H. On distributed logic simulation using time warp. In *VLSI 91* (Edinburgh, Scotland, August 1991), A. Halaas and P. B. Denyer, Eds., IFIP TC 10/WG 10.5, pp. 127–136.

[4] Bellenot, S. State skipping performance with the time warp operating system. In *6th Workshop on Parallel and Distributed Simulation* (January 1992), Society for Computer Simulation, pp. 53–61.

[5] Caianiello, E. R. *Functional Analysis and Optimization.* Academic Press, New York, 1966.

[6] Cleary, J., Gomes, F., Unger, B., Zhonge, X., and Thudt, R. Cost of state saving & rollback. In *Proc. of the 8th Workshop on Parallel and Distributed Simulation (PADS)* (July 1994), Society for Computer Simulation, pp. 94–101.

[7] Fleischmann, J. Parameter regulation in optimistic parallel simulation. Diplomarbeit, Technische Universität München, December 1994.

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

[9] Fujimoto, R. M., Tsai, J., and Gopalakrishnan, G. C. Design and evaluation of the rollback chip: Special purpose hardware for time warp. *IEEE Transactions on Computers 41*, 1 (January 1992), 68–82.

[10] Jain, R. *The Art of Computer Systems Performance Analysis.* John Wiley & Sons, Inc., New York, 1991.

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

[12] Kant, K. *Introduction to Computer System Performance Evaluation.* McGraw-Hill, Inc., 1992.

[13] Lin, Y.-B., and Lazowska, E. D. The optimal checkpoint interval in time warp parallel simulation. Tech. Rep. 89–09–04, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, September 1989.

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

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

[16] Palaniswamy, A., and Wilsey, P. A. An analytical comparison of periodic checkpointing and incremental state saving. In *Proc. of the 7th Workshop on Parallel and Distributed Simulation (PADS)* (July 1993), Society for Computer Simulation, pp. 127–134.

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

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

[19] Samadi, B. *Distributed Simulation, Algorithms and Performance Analysis.* PhD thesis, Computer Science Department, University of California, Los Angeles, CA, 1985.