Transparent Incremental State Saving in Time Warp Parallel Discrete Event Simulation

Robert Rönngren, Michael Liljenstam, Rassul Ayani, Johan Montagnat

▶ **To cite this version:**

Robert Rönngren, Michael Liljenstam, Rassul Ayani, Johan Montagnat. Transparent Incremental State Saving in Time Warp Parallel Discrete Event Simulation. Parallel And Distributed Simulation (PADS96), May 1996, Philadelphia, United States. pp.70-77. hal-00691810

HAL Id: hal-00691810
https://hal.science/hal-00691810v1

Submitted on 27 Apr 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Transparent Incremental State Saving in Time Warp Parallel Discrete Event Simulation

Robert Rönngren, Michael Liljenstam
and Rassul Ayani
Email: parsim@it.kth.se
SimLab, Dept. of Teleinformatics
Royal Institute of Technology
SWEDEN

Johan Montagnat
Ecole Normale Superieure de Cachan
Cachan (Paris)
FRANCE

Abstract

Many systems rely on the ability to rollback (or restore) parts of the system state to undo or recover from undesired or erroneous computations. Examples of such systems include fault tolerant systems with checkpointing, editors with undo capabilities, transaction and data base systems and optimistically synchronized parallel and distributed simulations. An essential part of such systems is the state saving mechanism. It should not only allow efficient state saving, but also support efficient state restoration in case of roll back. Furthermore, it is often a requirement that this mechanism is transparent to the user. In this paper we present a method to implement a transparent incremental state saving mechanism in an optimistically synchronized parallel discrete event simulation system based on the Time Warp mechanism. The usefulness of this approach is demonstrated by simulations of large, detailed, realistic FCA and a DCA-like cellular phone systems.

1. Introduction

Many systems rely on the ability to rollback (or restore) parts of the system state to undo or recover from undesired or erroneous computations. Examples of such systems include fault tolerant systems with checkpointing, editors with undo capabilities, transaction and data base systems and optimistically synchronized parallel and distributed simulations. An essential part of rollback based systems is the underlying state saving mechanism. This mechanism should not only allow efficient saving of states, but also support efficient state restoration in case of rollback. The implementation of the state saving and restoration mechanism is in many systems transparent to the user. The reason for this is obvious: the user should not have to bother with the intricate details of this mechanism such as whether complete states are saved or only a list of changes to the state. In this paper we examine some issues regarding state saving mechanisms in an optimistically synchronized parallel discrete event simulation (PDES) system [9] based on the Time Warp synchronization mechanism [13].

The motivation for PDES is twofold: (i) to increase the execution speed; and/or (ii) to enable execution of larger simulation models compared to sequential DES. During the last decade, researchers have proved the efficiency of PDES methods in a number of application areas [6, 9, 11]. Today, very challenging simulation problems are common in the industry and the necessary hardware for PDES is widely available, such as multi-processor workstations or reasonably efficient networks of workstations. Taking this into consideration one could expect PDES methods to be commonly used outside the PDES research community. This is however not yet the case [11]. One of the motivating factors for this is that very few, if any, PDES systems are sufficiently transparent. In most systems the user has to understand the underlying mechanisms and modify his (sequential) simulation code accordingly. This situation is by large a consequence of the quest for best possible performance. However, the execution of a simulation is only part of the simulation lifecycle [23]. Thus the performance gain from using a non-transparent PDES system is often outweighed by the additional time and effort that has to be spent in using the system.

In this paper we examine the possibilities to transparently implement incremental state saving in a PDES kernel based on Time Warp synchronization. The kernel is implemented in the C++ programming language. The rest of the paper is organized as follows. Section 2 describes several methods that have been proposed to reduce the state saving overhead in Time Warp. In Section 3 we describe the simulation kernel used in the experiments and in Section 4 a novel method to implement incremental state saving (ISS) in simulation kernels based on C++. This method has been implemented and tested in a cellular phone system simulator which is described in Section 5. Section 6 presents experimental results which show the impact on performance of the ISS method. Section 7 summarizes the contributions of the work presented in this paper.

2. State saving Issues in Time Warp

A PDES system can be expected to require significantly more memory than the corresponding sequential DES system to execute efficiently [20]. In the case of Time Warp based PDES a naive implementation can use an arbitrarily large amount of memory compared to the corresponding sequential system [18, 17, 19]. Since the motivation for PDES is to speed up the execution and/or enable the execution of larger simulation models it is essential to find mechanisms by which the state saving overhead can be reduced. We can distinguish between methods that: (i) reduce the amount of state information that is saved, thus reducing both execution time and memory consumption; and (ii) methods that can reclaim memory on demand, including memory of future objects (i.e. associated with events with timestamps greater than GVT [13]), limiting the maximum memory needed [14, 17]. In the following we concentrate on the former category.

2.1. State Saving Techniques in Time Warp

The simplest method for state saving in Time Warp is to copy the entire state of a logical process (LP) each time it executes an event message. This is often referred to as *copy state saving* (CSS). However, we can expect rollbacks to be relatively infrequent compared to ordinary event executions in most Time Warp based simulations [8]. Furthermore, a state of an LP can be regenerated from an earlier state by re-execution of intermediate events. Accordingly, an LP does not need to save (or checkpoint) its state at each event execution but can choose to checkpoint only every χ^{th} state [16]. This is referred to as *infrequent* (or sparse or selective) *checkpointing*. Several methods have been proposed by which LPs adaptively can select their checkpoint intervals χ [21, 12]. These methods can easily be made transparent to the user.

Many challenging simulations, such as battle field simulations or simulations of large communication systems, are characterized by LPs with very large states where only a fraction of the state is updated in each event execution. In such applications it may be inefficient or even infeasible to save copies of the complete state which can be in the order of hundreds of kilobytes [6]. In such applications it is often appropriate to use *incremental state saving* (ISS) [2, 4, 24, 25, 26], in which only the updated parts of the state are saved. Thus the state saving mechanism builds a chain of state changes. In case of rollback the state is restored by undoing these changes.

Few systems, if any, implement transparent ISS primarily due to problems associated with identifying which parts of the state that are updated and when. This could be accomplished by use of special purpose hardware [10] or by special purpose simulation languages with compiler support for ISS. However, due to cost issues, a majority of PDES systems are implemented on top of some general purpose programming language such as C or C++. Several of these systems implement ISS. The SPEEDES environment features several interesting and efficient techniques to implement ISS [24, 25]. Good performance results with ISS in the context of VLSI simulations [2] and simulations of large telecom networks [6] have been reported. ISS has also been implemented using persistent objects [4] in an interesting effort to achieve a transparent implementation of ISS. However, these implementations of ISS have the common denominator that they put the responsibility on the user of writing either special purpose code for the ISS or calling special functions when updating state variables or to explicitly save state variables that have been updated. If the user fails to supply the necessary code for ISS in these systems, the state restoration in case of rollback may be corrupted. This is likely to generate non-deterministic, erroneous, simulation results. Finding this type of programming errors is often hard, even for a user with a thorough understanding of the Time Warp mechanism. Hence, we conclude that it is essential that ISS can be implemented as transparently as possible to the user. The question is to which extent this is possible.

3. Parallel Simulation Kernel

The Parallel Simulation Kernel (PSK) used in this study is based on Time Warp synchronization and runs on shared memory multiprocessor workstations. It is written in C++ and uses static assignment of the LPs to the processors, aggressive cancellation of events (i.e. events are cancelled as soon as an antimessage is received), and the direct cancellation optimization for shared memory machines described in [8]. The basic synchronization primitives (such as locks, and barriers) are supplied by the p4 macro library [3] making the PSK portable to a variety of multiprocessors. One modification has been made to this library, however, the queuing locks supplied in the package have been replaced by spin-locks.

3.1. PSK Structure

The PSK provides an application independent basis on top of which discrete event simulations can be built. The system is object oriented. When creating application specific logical processes the user inherits from a virtual LP class, see Figure 1. Associated with an LP is a *StateHandler* object which implements the state saving and restoration method. Through the StateHandler the LP object is able to save its state and to rollback to a previously saved state. These mechanisms are transparent to the user.

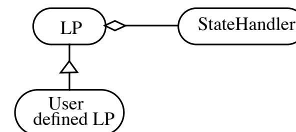

Figure 1. The LP inside the PSK

Each LP is also associated to an *Event Set* containing both executed and un-executed events and a *State Set* containing copies of old states to be used for state restoration purposes in case of rollback, Figure 2. When an event is executed, it usually modifies the LP state. Thus before an event execution the StateHandler constructs a data structure containing the information necessary to restore the old state and links it to the event.

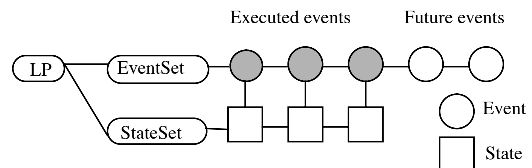

Figure 2. The events and the state set

In the case of incremental state saving, the data structure storing the state restoration information is a list of back ups of old values of state variables which have been modified in the execution of the event. In case of rollback, the StateHandler parses all executed events backward up to the rollback point. For each event in the rollback the StateHandler traverses the list of old state variable values in reverse order to restore the state.

4. Transparent Incremental State Saving

ISS can be implemented by identifying all updates to state variables at runtime and backing up the old values of the state variables before the state variable in itself is modified. In particular this could be implemented by changing the semantics of all side effect operators on state variables. In C++ it is possible to define data types and change the semantics of the operators on

these data types by what is referred to as *operator overloading*. Thus, ISS can be implemented by creating special data types to encapsulate all state variables with overloaded operators that automatically perform the necessary ISS back up. Introducing special data types for state variables is not a severe limitation as many DES systems and most PDES systems require that the user declares his state variables so that they can be easily identified. Though this method may appear as straight-forward to implement there are several aspects of this approach which merit further attention.

4.1. Data Encapsulation

Only data which is part of the state of an LP should be backed up by the ISS mechanism. Hence, it is necessary to be able to distinguish between state variables and other variables. To solve this problem, a template class has been created to encapsulate any data type used to declare state variables, Figure 3. It is referred to as the State<> class.

```
template<class T> class State {
   public:
      ....       // overloaded operators
   private:
      T m_data;
};
```

Figure 3. The State class declaration

The user is required to declare his state variables as State<> encapsulated data. The declaration of a state variable of type integer:

```
   int m_int;
```

is thus written as:

```
   State<int> m_int;
```

This will create `m_int` as an object for which the operators are overloaded to backup the data by transparently calling a backup method of the StateHandler object. The state variable data is stored in the type int `m_data` data member of the State<> object.

4.2. Overloading Operators

The backup method must be called each time a state variable is modified. Thus any side effect operator must be overloaded. In C++ this includes the =, ++, --, +=, -=, *=, /=, %=, ^=, &=, |=, >>= and <<= operators. But to achieve complete transparency, all other operators on the State <> objects must be overloaded. This is to allow the user to write expressions such as: `m_int + 2` or `2 + m_int` or `m_int + m_int` which perform calls to:

```
   operator+<int>(State<int>&, int);
   operator+<int>(int, State<int>&);
   operator+<int>(State<int>&, State<int>&);
```

The proposed State<> template, its associated operators and copy constructor provide the user with a transparent ISS mechanism for all simple data types, such as integers, floats etc. However, arrays and other compound data objects deserve special attention.

Some operators, such as the subscript([]) operator, may be overloaded using different prototypes. That is, the user can define compound classes for which he can overload the subscript operator to return an object of any type of data member. Hence, one cannot make generic assumptions on the resulting type when applying subscript. Consequently, the State<> class cannot provide a generic subscript operator to ensure transparency. A simple solution is not to declare indexed state variables as State objects. Instead, each data member of indexed objects or classes should be State<> objects. This ensures that the state of the LP will be correctly backed up. An example illustrating this is found in Figure 4.

4.3. Initialization of Incremental State Saving

A problem related to automatic incremental state saving is how to initialize the state saving. When the LPs are created, their data is initialized, i.e. modified. These initializations will cause calls to the backup method of the StateHandler. This is not desirable since the simulation has not yet started and the StateHandler objects may not yet exist. Consequently, the proposed mechanism must be able to distinguish an initialization from an assignment, though both may use the same methods in C++.

In our PSK this has been solved by performing a call to the backup method through a pointer. When a State<> object is created, the pointer is set to a dummy backup method which does nothing. Consequently, backup calls performed during the initialization are harmless. When required, each LP enables the backup ability of all its internal State<> objects by changing the backup method pointer of these objects. To enable this, the State<> objects link themselves into a list accessible to the LP when they are created. Currently, the backup capability of the State<> objects of an LP is only enabled on the completion of the initialization of the LP. Hence, the state of an LP cannot be augmented after the initialization unless special provisions are made to enable their backup capabilities as these parts otherwise would not be backed up.

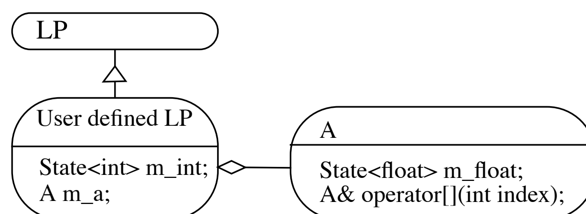

Figure 4. An example of correct data encapsulation in the proposed incremental state saving scheme

4.4. Temporary Objects

C++ compilers can sometimes create temporary objects automatically. Consider the example in Figure 4. If the user writes a statement like:

```
   m_a = A();
```

a temporary object of type `A` is created on the stack, this object is copied to `m_a` and immediately deallocated from the stack. This could interfere with the incremental state saving mechanism if the temporary object contains State<> objects, in which case the temporary object would backup itself. This should not occur since the temporary object is not part of the LP state. Furthermore, it could cause the StateHandler to try to restore the state of non-existing objects in the case of rollback.

Two cases has to be taken into consideration. The temporary object could be created: (i) during the initialization phase; or (ii) after the initialization phase. In the first case the backup mechanism is not yet enabled but the temporary object will link itself to the LP list as described in Section 4.3. Since the temporary object will be deleted at the end of the LP creation, it should be unlinked from the linked list when its destructor is called.

If the temporary object is created during the execution it will not have its backup ability enabled. However, the State<> objects of the temporary object links and un-links to/from the list of State<> objects of the LP as described in Section 4.3. This could potentially be costly if the temporary object contains a large number of State<> objects. To prevent this a global variable is used as a flag which is set upon completion of the initialization. When created, a State<> object checks this flag to see if it should link to the list or not.

4.5. Pointers and Dynamic Memory Allocation

In a general case the state of an LP may contain pointers to data objects. These data objects could be dynamically allocated and deallocated. Our current implementation of ISS does not yet support these data types. In fact many PDES systems put the restriction that the state size of an LP has to be staticly defined. However, we will outline a possible solution to this problem which we intend to implement in a near future.

Integer arithmetic on pointers has to be supported. To guarantee transparency, operators acting both on State<> encapsulated pointers and integers should be overloaded in addition to the operators overloaded for non-pointer types. Since these operators do not exist for other objects than pointers, a template subclass of State<> will be declared, referred to as RefState<>. This will guarantee that pointers are correctly handled.

Transparent backup of dynamically allocated objects can be performed with the restriction that such objects only contain data members declared as State<> objects. To ensure that the backup capability of these objects are enabled, Section 4.3, the dynamically allocated object has to inherit from a base class where the constructor enables the backup capability. Furthermore, the method has to guarantee that a dynamically allocated object is not deallocated before it can be guaranteed that it will never be used in a rollback. For this purpose, the new and delete operators of the base class has to be overloaded. The base class contains a flag indicating if the user tried to delete the object or not. On a delete call, the memory block is not really deallocated, only the flag is set. The block can only be deleted in fossil collection and if it is tagged as deleted.

4.6. Memory Management Overhead

The proposed method for incremental state saving will cause the creation of backup objects each time a State<> variable is modified. Thus it is essential to have an efficient memory handler. We can expect that a majority of the data objects which will be backed up are small (such as 4-byte integers). Dynamically allocating and manipulating a large number of such objects in a list is not efficient. To alleviate this problem an improved memory manager for ISS has been introduced which treats backup objects containing less than 8-bytes of data (the size of a double i.e. the biggest C++ built-in type) in a special way. The memory manager implements linked lists of arrays of update structures. Each update structure can hold backup data of up to 8-bytes length. When an event is executed the StateHandler will request arrays of update structures to store backups of state variables less than 8 bytes in from the memory handler on demand. Thus the linked list operations for small backup objects are replaced by incrementing an index in an array. This scheme also alleviates the problem of deallocating the backup structures at fossil collection. With this method all small backup objects can be deallocated efficiently by returning the update structure arrays to the memory handler. Only backup objects larger than 8 bytes are dynamically allocated and stored in the linked list.

4.7. Level of Transparency Achieved

The proposed mechanism is not completely transparent. In particular, the user has to explicitly declare state variables as State<> or RefState<> objects. In addition, some minor problems remain. The compiler is sometimes able to automatically cast from a user defined type T to a given type X, but not from State<T> to X, though a cast operator from State<T> to T is provided in the State<> class. Thus, the user might have to explicitly make some casts which were not previously needed.

The user should also be cautious when redefining the copy constructor or the assignment operator of compound classes used for state variables. If the user defines his/hers own copy constructor and/or assignment operator for such classes, he/she should call the copy constructor or assignment operator of all contained State<> declared objects.

The proposed method differs from what can be achieved by compiler based methods in that there is no way of preventing the user from, intentionally or not, bypass the backup mechanism by modifying State<> data through pointers.

5. The Cellular Phone Simulator

In this study we have tested our ISS implementation on a simulator of a cellular communication system. Good performance for parallel simulation of similar systems has been reported in e.g. [5]. Our simulator has previously been described in [15] and we will limit ourselves to describing some of its general properties.

5.1. The Cellular Phone Model

A cellular communication system is divided into a number of *cells*. To each cell a certain number of radio channels are allocated. Mobile Stations (MSs) residing in a cell can allocate any of the free radio channels allocated to the cell to send and receive phone calls. The same channel can be used in several cells that are sufficiently spaced apart since interference will be negligible.

When a mobile terminal moves from one cell to another it may become necessary to connect the MS to another base station, this is called a *handover*. If an attempt is made to make a call to or from a MS in a cell where there are no available channels, the call is *blocked*. If an ongoing call can not be handed over to a cell because there are no available channels it may be forced to terminate in which case the call is said to be *dropped*.

Radio channels can be allocated to cells staticly using Fixed Channel Assignment (FCA) or dynamically using some Dynamic Channel Assignment (DCA) scheme.

The simulation model consists of three submodels in a similar manner to the MaDRAS simulator [1]: the tele-traffic model, the mobility model, and the propagation model.

The teletraffic model describes the arrival process of new calls and their duration. The call arrivals form a Poisson process with a mean arrival rate λ, and exponential call duration with mean $1/\mu$. The mobility model describes the movements of the mobile stations (MSs). The propagation model describes distance dependent propagation loss using a statistical model for correlated log-normal shadow fading. The path losses at a specific position are regarded as constant over time, so that at a position p the gain from a transmitting station s is $G_{s,p}$ forming a two-dimensional matrix for each base station that describes the geography.

Each MS performs a “resource reallocation” procedure every 500 ms of simulated time. This procedure involves calculating the uplink and downlink Signal-to-Interference Ratio (SIR) and comparing it to a “least acceptable”-threshold to determine if the call should be dropped. Other BSs are also compared against the current connection and if another BS is found to be significantly better a handover is attempted. If a handover can not be completed due to lack of free channels the MS remains with the current connection.

Channel selection is done stochastically with equal probability among the available free channels on the BS. The uplink and downlink channels are changed at the same time and hence treated as one unit. Path losses are regarded as equal in both directions due to reciprocity. The radio channels are assumed to be orthogonal, so adjacent channel interference has been neglected.

The positions of the mobiles are updated every time there is new data to be read from the gain matrix, i.e. each mobile schedules a position update event for itself to occur when it has travelled the distance between two samples in the gain matrix. This event will cause the mobile to read the new radio gain from the matrix.

Each channel is a Logical Process (LP) in our model. There is also one generator LP that creates a mobile station when a new call arrives and sends it to the first channel assigned to it. Figure 5 shows an example of the communication patterns in the model. When a new call arrives to the system, a corresponding MS is created by the generator. After a BS has been selected, the new MS then proceeds to make its initial channel selection. A request is sent to all channels that are available on the selected BS to find out which channels are free at this time. One of the free channels is selected (if there is one, otherwise the call is blocked) and the MS object is sent to that channel. Similarly, when an MS decides to perform a handover to another BS a new set of requests are sent out to all channels available on the new BS and the selection is made. If no free channels are found and link quality is insufficient the call is eventually dropped. Requests made for the same channel from different MSs at the same simulation time is handled through an event priority scheme ordering the LPs so that one channel selection interaction is always completed before the next request message is processed. A channel holds information about all entities (base stations and mobiles) communicating on that channel.

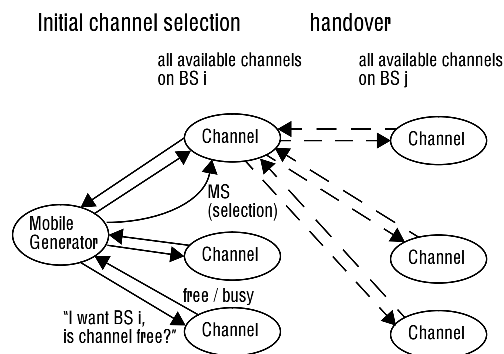

Figure 5. Basic communication pattern in the simulation model.

5.2. Characteristics of the models relevant to state saving issues

With each model we have simulated two scenarios: (i) a system of 7 cells; and (ii) a system of 67 cells. These scenarios are referred to as the small and large area respectively, where the larger area is closer to a realistic system. Radio signal propagation is described by a read only matrix which is 0.3 Mb for the small area and 27Mb for the large area.

Each event will only modify one mobile station out of all the connections contained in the Channel object. The size of the Channel state for the FCA case is 1720 bytes and one BS-MS connection is only 156 bytes. Furthermore, most events are simple position updates in which case only part of the BS-MS connection is modified. Hence, on average only about 2% of the state is updated in an event which makes incremental state saving seem like a good candidate as suggested in [6]. However, the state saving overhead is only a relatively small fraction of the total time to process an event due to the large event granularity. The mean event execution time is about 260 microseconds for the small area and 750 microseconds for the large area. The higher event execution time for the large area is primarily due to an increased cost for interference computations. The state size remains the same for both areas and the average time to save a state using copy state saving is about 140 microseconds. This means that for any improved state saving scheme (compared to pure copy state saving) we can at most cut the execution time by at about 45% for the small area and 15% for the large area by reducing the state saving overhead.

When using states sizes adjusted for a DCA model the situation becomes quite different. The mean event execution time increases slightly to about 430 microseconds for the small area and 830 microseconds for the large area, but the average state saving time increases to about 2200 microseconds for both models. This is due to a dramatic increase in state size for the Channel LP to 42776 bytes. The size of the BS-MS connection increases to 636 bytes. The most frequent events are still position updates. Consequently, the average fraction of the state updated in an event execution is even less for the DCA model than for the FCA model. In an average event execution less than 0.2% of the state

is updated. The state saving overhead is quite severe when using copy state saving and in this case accounts for about 75 - 85 % of the total time to process an event. Thus, we hypothesize that the DCA models will benefit even more from ISS than the FCA model.

6. Experimental Results

In this study we have compared the impact of four different state saving mechanisms on the cellular phone simulator for the FCA and DCA models:

- Copy State Saving (CSS). The state of the LP is saved at each event execution.
- Sparse State Saving (SSS) with fixed state saving interval. The best state saving intervals were experimentally determined to be 5 for the FCA model and 10 for DCA model respectively.
- Transparent Incremental State Saving (TISS) which is the method described in this report.
- User dependant Incremental State Saving (UISS). This is a more conventional ISS method in which the user is required to explicitly call a backup function. The cellular phone models are such that only a well defined part of the state is updated on each event execution. Consequently, only a few calls to the backup function are necessary on each event execution.

An important aspect of the transparency achieved by the TISS method is that the *only* changes that had to be made to the user code of the simulation models was to declare the state variables as State<> objects. By defining a macro for the State<> declarations exactly the same code could be used for CSS, SSS and TISS. This has several important implications: i) it is of great value when selecting the appropriate state saving method for a particular application also making it possible to support a wider variety of applications with a single user interface to the simulation system; (ii) it forms a basis for building a system which automatically selects the best state saving method for individual LPs.

The experiments were performed on a SUN Sparcstation 10 with 4 processors, 128 Mb primary memory and 1 Mb cache per processor. The average rollback length, in these experiments, varies between 3 and 4.

6.1. Speed-up

The performance results in Figures 6 through 9 are shown as the relative speed of the proposed state saving algorithms compared to copy state saving. Figures 6 and 6 depicts the performance for the FCA model for small and large areas respectively. As predicted in Section 5.2 the improvement from reducing the state saving for this model would be limited to 15% for the large area and 45% for the small area. The performance results indicate that such improvements are possible with sparse state saving. To some extent the reduced memory consumption also results in improved performance of the memory system. Furthermore, the ISS methods are outperformed by the SSS method in these experiments. This is primarily due to the lower overhead in execution time for this method for the relatively small state sizes of this model.

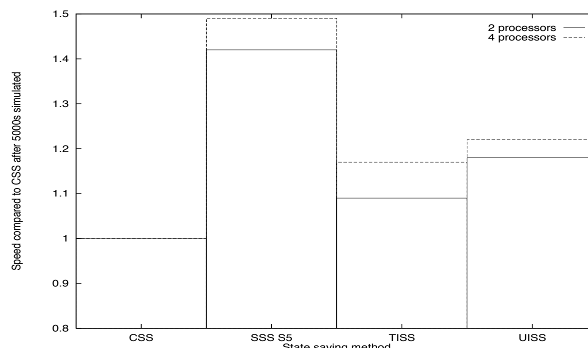

Figure 6. FCA model, small area simulated on 2 and 4 processors.

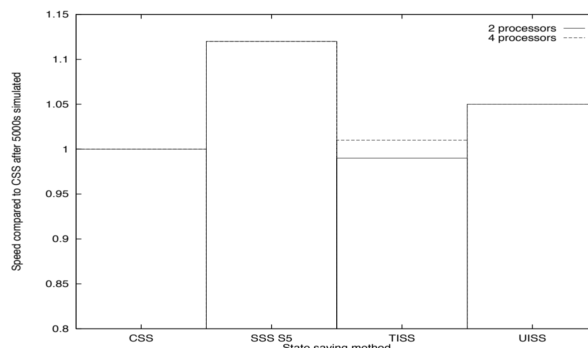

Figure 7. FCA model, big area simulated on 2 and 4 processors.

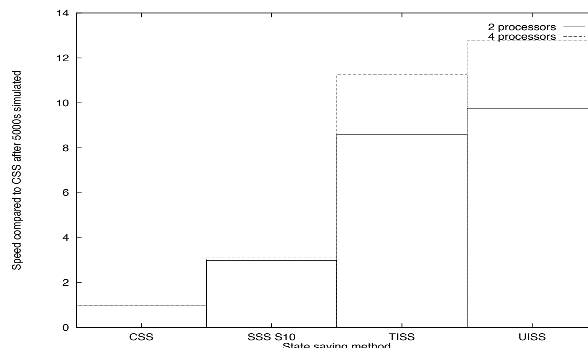

Figure 8. DCA model, small area simulated on 2 and 4 processors.

Figures 8 and 9 show the performance of the state saving algorithms for the DCA model. The DCA model differs from the FCA model in that the states of the LPs are significantly larger in the DCA model. Consequently, these figures reveal that the ISS methods perform significantly better than CSS and SSS. Furthermore, we see that there is a significant cost for the transparency of the TISS method compared to the UISS. We do, however believe that this cost is justified in most cases as it alleviates the user from the burden of having to explicitly deal with the underlying state saving mechanism. The speed-up achieved by the ISS methods are higher than what could be attributed to the reduction of the state saving overhead alone. Examining Figure 13 we see that this phenomenon is not due to an increased efficiency. We

believe that the additional performance improvement mainly is caused by an improved locality which improves cache performance. This hypothesis is supported by the fact that the smaller model, for which the relative memory consumption reduction is larger, exhibits a larger improvement.

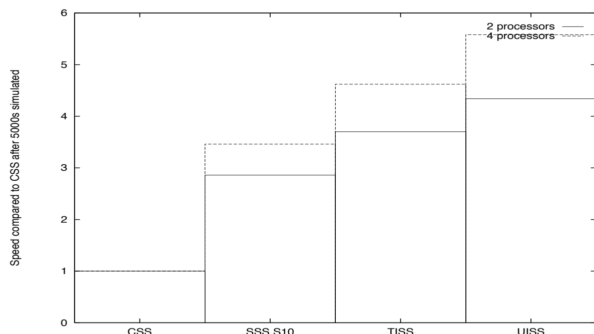

Figure 9. DCA model, big area simulated on 2 and 4 processors.

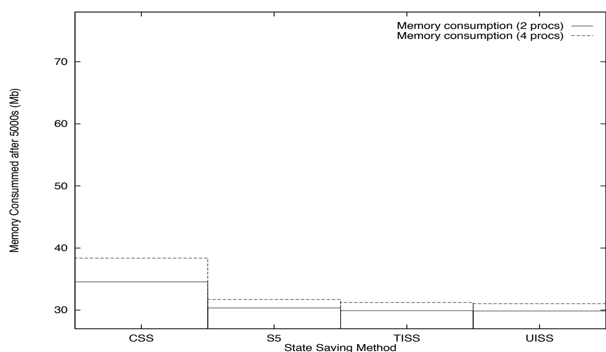

Figure 10. Memory consumption for the FCA model

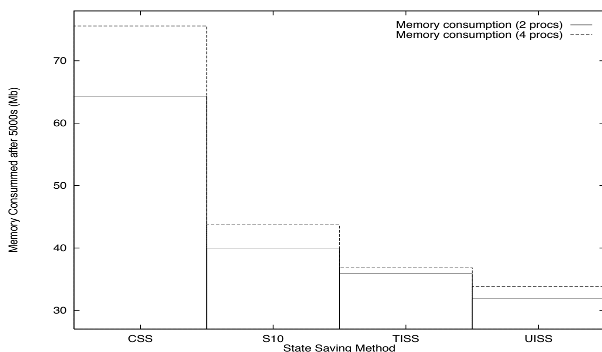

Figure 11. Memory consumption for the DCA model

6.2. Memory Consumption

Figure 10 and 11 show the memory consumption for the FCA and DCA models respectively for the large area. The memory consumption was recorded during long runs (5000 seconds of simulated time) for which the memory used by the simulator stabilized. In both models, the simulator first allocates a 27 Mb data structure to store the area simulated. Hence, only memory above 27 Mb is significant as far as state saving is concerned. These figures clearly show that both sparse and incremental state saving can reduce the memory consumption significantly. UISS performs slightly better than TISS which is explained by the fact that while TISS saves several small pieces of the state UISS can save the same information as a single or a few larger pieces reducing the overhead.

6.3. Efficiency

Changes to one of the mechanisms in a Time Warp system can sometimes affect the optimistic synchronization substantially. This effect has been demonstrated for event list management [22] and other types of memory management policies [7]. Figures 12 and 13 show the impact of the state saving mechanism on the efficiency, defined as the ratio of committed events to executed events (committed events and events that are rolled back). These figures show that the state saving methods had little impact on the efficiency for these benchmarks.

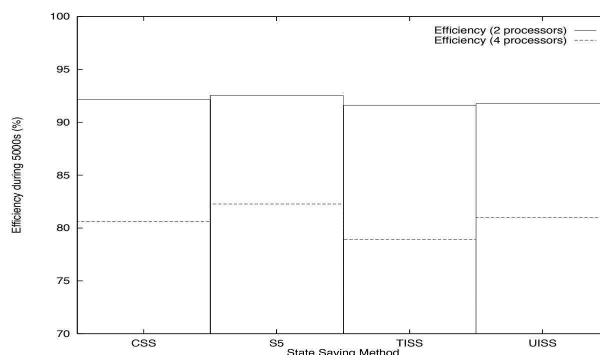

Figure 12. Efficiency of the state saving algorithms in the FCA model

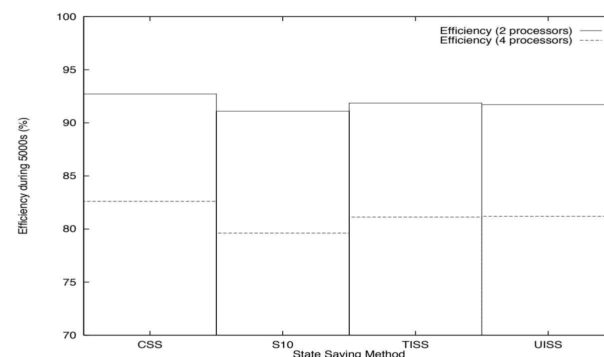

Figure 13. Efficiency of the state saving algorithms in the DCA model

7. Conclusions

Transparency of the state saving mechanism is essential for the acceptance of PDES methods by a wider audience as it relieves the user from the burden of having to understand and interact with an often intricate mechanism. In this paper we have discussed a method to implement incremental state saving in a Time Warp simulation system built on top of the C++ programming language. The method is based on the ability to overload operators in C++. The most prominent characteristics of this approach is that it achieves a high degree of transparency with

acceptable overhead compared to a non-transparent implementation of ISS.

In the proposed method the *only* changes required to the users application code is to use special type declarations of the state variables. In particular, this allows the same user application code to be used regardless of whether the underlying state saving mechanism used is copy state saving, sparse state saving or incremental state saving.

The usefulness of the proposed method has been demonstrated by simulations of large realistic simulation models of cellular phone systems. The experimental results show that incremental state saving is important to achieve good performance in cases where the state vectors are large and only a small fraction of the state is updated on average. In some cases the execution time was reduced by more than an order of magnitude compared to conventional copy state saving.

In future systems we believe that the system should be able to select the best state saving method for individual LPs at run time. This will be facilitated if the state saving mechanism can be made transparent.

8. References

1. Andersin, M., Frodigh, M., Sunell K-E, “Distributed Radio Resource Allocation in Highway Microcellular Systems”, Fifth WINLAB Workshop on Third Generation Wireless Information Networks, Rutgers University, New Jersey, -95
2. H. Bauer et al., “Reducing Rollback Overhead in Time-Warp Based Distributed Simulation with Optimized Incremental State Saving”, Proceedings of the 26th Annual Simulation Symposium, pages 12-20, March 1993.
3. R. Butler, E. Lusk, “Monitors, messages, and clusters: the p4 parallel programming system”, Parallel Computing, 20, April 1994
4. D. Bruce, “The Treatment of State in Optimistic Systems”, Proceedings of the 9th Workshop on Parallel and Distributed Simulation (PADS95), pages 40-49, June 1995.
5. C. Carothers, R. Fujimoto and Y.-B. Lin, “A Case Study in Simulating PCS Networks Using Time Warp”, 9th Workshop on Parallel and Distributed Simulation, Lake Placid, NY, 1995.
6. J. Cleary, F. Gomes, B. Unger, X. Zhonge and R. Thudt, “Cost of State Saving & Rollback”, Proceedings of the 8th Workshop on Parallel and Distributed Simulation, Vol. 24, No. 1, pages 94-101, July 1994.
7. S.R. Das and R. F. Fujimoto. “A Performance Study of the Cancelback Protocol for Time Warp Parallel Simulation”, Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems. pages 201-210, May 1994.
8. R. Fujimoto, “Time Warp on a Shared Memory Multiprocessor”, Transactions of the Society for Computer Simulation, Vol. 6, No. 3, pages 211-239, July 1989.
9. R. Fujimoto, “Parallel Discrete Event Simulation”, Communications of the ACM, Vol. 33, No. 10, pages 30-53, October 1990.
10. R. Fujimoto. et al., “Design and Evaluation of the Rollback Chip: Special Purpose Hardware for Time Warp”, IEEE Transactions on Computer, Vol. 41, No. 1, pages 53-64, January 1992.
11. R. Fujimoto, “Parallel Discrete Event Simulation: Will the Field Survive?”, ORSA Journal on Computing, Vol. 5, No. 3, pages 213-230, Summer 1993.
12. J. Fleischmann and P. Wilsey, “Comparative Analysis of Periodic State Saving Techniques in Time Warp Simulators”, Proceedings of the 9th Workshop on Parallel and Distributed Simulation (PADS95), pages 50-59, June 1995.
13. D. Jefferson, “Virtual Time”, ACM Transactions on Programming Languages and Systems, Vol. 7, No. 3, pages 404-425, July 1985.
14. D. Jefferson, “Virtual Time II: Storage Management in Distributed Simulation”, Proceedings of the 9th Annual ACM symposium on Principles of Distributed Computing, pages 75-90, August 1990.
15. M. Liljenstam, R. Ayani, “A Model for Parallel Simulation of Mobile Telecommunication Systems”, To appear in Proceedings of the International Workshop on Modeling Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), San Jose, CA, February, 1996
16. Y.-B. Lin, B. R. Preiss, W. M. Loucks and E. D. Lasowska, “Selecting the Checkpoint Interval in Time Warp Simulation”, Proceedings of the 7th Workshop om Parallel and Distributed Simulation (PADS93), pages 3-10, May 1993.
17. Y-B. Lin and B. Preiss, “Optimal Memory Management for Time Warp Parallel Simulation”, ACM Transactions on Modeling and Computer Simulation, Vol. 1, No. 4, pages 283-307, October 1991.
18. B. D. Lubachevsky, A. Schwartz and A. Weiss, “Rollback sometimes works… if filtered”, 1989 Winter Simulation Conference Proceedings, pages 630-639, December 1989.
19. B. D. Lubachevsky and A. Weiss, “An Analysis of Rollback-Based Simulation”, ACM Transactions on Modeling and Computer Simulation, Vol. 1, No. 2, April 1991.
20. B. Preiss and W. M. Loucks, “Memory Management Techniques for Time Warp on a Distributed Memory Machine”, Proceedings of the 9th Workshop on Parallel and Distributed Simulation (PADS95), pages 30-39, June 1995.
21. R. Rönngren and R. Ayani, “Adaptive Checkpointing in Time Warp”, Proceedings of the 8th Workshop on Parallel and Distributed Simulation, pages 110-117, July 1994
22. R. Rönngren, R. Ayani, S. Das and R. Fujimoto, “Efficient Implementation of Event Sets in Time Warp”, Proceedings of the 7th Workshop on Parallel and Distributed Simulation (PADS93), pages 101-108, May 1993.
23. R. G. Sargent, “Verification and Validation of Simulation Models”, 1994 Winter Simulation Conference Proceedings, pages 77-87, December 1994.
24. J. Steinman, “SPEEDES: A Multiple-Synchronization Environment for Parallel Discrete-Event Simulation”, International Journal in Computer Simulation, Vol. 2, No. 3, Pages 251-286, 1992.
25. J. Steinman, “Incremental State Saving in SPEEDES Using C++”, Proceedings of the 1993 Winter Simulation Conference, pages 687-696, December 1993.
26. B. W. Unger, J. G. Cleary, A. Convington and D. West, “An external state management system for optimistic parallel simulation”, Proceedings of the 1993 Winter Simulation Conference, pages 750-755, December 1993.