Proceedings of the 1997 Winter Simulation Conference
ed. S. Andradóttir, K. J. Healy, D. H. Withers, and B. L. Nelson

CHECKPOINT AND RECOVERY METHODS IN THE PARASOL SIMULATION SYSTEM

Edward Mascarenhas

Silicon Graphics Computer Systems
2011 N. Shoreline Blvd. MS 510
Mountain View, CA 94043
U.S.A.

Felipe Knop

IBM Corporation
522 South Road, MS P963
Poughkeepsie, NY 12601
U.S.A.

Reuben Pasquini
Vernon Rego

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907, U.S.A.

ABSTRACT

State-saving operations are a major source of overheads in optimistic and adaptive parallel discrete-event simulations. We present some techniques for saving state in the the context of the PARASOL multithreaded parallel simulation system. In this system, threads are used to implement both logical processes and active transactions which access passive simulation objects. Hence, system state is a combination of thread-state and object-state. We introduce a new save-if-modified method for incremental checkpointing of threads and objects. Because of the PARASOL system's domain-oriented support, checkpointing is transparent to the user. Application-level objects that are foreign to a domain may be saved via invocations to primitives in the system's ParaState module.

1 INTRODUCTION

PARASOL is a parallel discrete-event simulation system based on the process-oriented and active-transaction view. In the queueing domain, for example, servers are implemented as passive objects and transactions are active (threads). As a customer (thread) moves through the system, its execution may require suspension. When a customer acquires a server (object), execution of the thread representing the customer is suspended until the customer's (simulated) service-time has elapsed. When service is complete, the thread resumes execution from its point of suspension. At this instant, both object-state and thread-state may undergo change. The thread continues to execute until it either suspends itself again, or leaves the system. Thus, as control is passed between transactions which suspend themselves for some simulation time, the simulator's clock advances.

PARASOL uses well-known optimistic synchronization (Fujimoto 1990) and novel adaptive synchronization methods (Mascarenhas 1996). A key advantage of optimistic synchronization is that it enables reduced programming effort. That is, application level programs can be developed without explicit synchronization directives; this is a highly attractive feature, considering that a major source of distributed program complexity lies in efforts to synchronize processors. This advantage, however, comes at a potentially high price. Each processor is free to run along a simulation trajectory at its own pace, stopping to correct its trajectory only when a causality error is detected. When detected, a causality error forces PARASOL-kernel-supported rollback to a checkpoint, state-recovery, and coast-forward to be invoked.

A causality error is said to occur when a logical process (LP) with simulation time t_n receives a (late, or "straggler") transaction with timestamp $t_p < t_n$. The computation is rolled back and restarted from a previously checkpointed and error-free state, valid at some simulation time $t_s \leq t_p$. During rollback, anti-transaction messages cancel transactions that were already sent to other LPs, but invalidated by the straggler's arrival. To support rollback, optimistic simulations must checkpoint their state periodically. After an error-free state is restored, the LP coasts forward (i.e., retraces simulation steps) to simulation time t_p and then processes the straggler. During the coast-forward phase, user-code is re-executed but simulator-code is only selectively executed. The idea is is to rebuild application-state at time t_p as rapidly as possible, using state checkpointed at time t_s. Checkpoint and rollback overheads can have a serious impact on execution efficiency.

State-management is a major source of overheads in PARASOL. Checkpointing, rollback, and polling (i.e., the simulator must repeatedly poll the network for incoming transactions) can be expensive. An example of such costs is shown in Table 1, to make the relative impact of these operations clear. These costs pertain to a queueing simulation on a SPARC 5 cluster and on an Intel Paragon; checkpointing overheads can be seen to be as high as 18% and 29%

Table 1: An Example of PARASOL Overheads (Percentages)

	Execution	St. Save	Rollback	Poll Net	Schedule	GVT	Other	Idle
Cluster	24.99	18.40	21.43	24.48	0.93	0.86	8.68	0.21
Paragon	19.26	29.55	21.60	17.49	0.83	0.22	10.68	0.34

of total cost, respectively. In this paper, we discuss present PARASOL's checkpoint and recover mechanisms, emphasizing that system state consists of both object-state and thread-state. We employ a save-if-modified method for incremental checkpointing of threads and objects. Through appropriate domain-layering, PARASOL provides the user with transparent checkpoint and recovery mechanisms. Application level objects that are foreign to a domain may also be checkpointed and recovered, but this requires user-level invocations to a special ParaState module.

The PARASOL system's state changes each time a transaction is processed. Any object modified by the executing transaction also changes state. Checkpointing can be done after each transaction runs to suspension, or after every χ transactions run to suspension; the quantity χ defines the interval between checkpoints, in units of transaction-executions or events. Increasing χ has the effect of reducing checkpoint frequency and thus state-saving costs, while simultaneously increasing the cost of the coast-forward phase. To enable efficient operation, we develop a strategy that minimizes the sum of checkpoint and coast-forward costs. The proposed model is implemented in PARASOL and marginally improves upon a method proposed earlier (Rongren and Ayani 1994), with similar motivation. More details of this method can be found in Mascarenhas (1996).

2 RELATED WORK

Optimistic synchronization methods operate in conjunction with mechanisms for state-saving and restoration. Various methods for efficient checkpointing of objects have been proposed. These include copy state-saving – complete state is saved, incremental state-saving – only modified state is saved (Steinman 1993), hardware-based state-saving (Fujimoto, Tsai, and Gop-alakrishnan 1988), and language extensions for state-saving (Gomes, Unger, and Cleary 1996). Efficient schemes for incremental state-saving, such as those employed by Steinmain (1993), require user-intervention for checkpointing and state-saving. Users need to get involved because the state to be checkpointed depends on the type of event executed.

In PARASOL, the idea is to achieve both user-level transparency as well as efficiency in checkpoint and recovery. To checkpoint object-state at any given time, we perform copy-state saving of the read-write state of all objects that were modified after the last checkpoint. In PARASOL, this requires checkpointing both object-state (passive components) as well as thread-state (active components). Thus, both computation state as well as object-state are saved in a way that is user-transparent. Checkpointing operations are managed by the runtime system, and enabled at a frequency that minimizes overall cost. The *save-if-modified* method that we propose is efficient because it transparently saves only that state which is modified between consecutive checkpoints. We achieve this by associating a timestamp with each checkpoint, and use this timestamp to decide what must be saved or restored at any given point in time.

3 OVERVIEW OF THE PARASOL SYSTEM

PARASOL is a process-oriented parallel simulation system based on the active-transactions paradigm. In contrast to existing event-based systems in which LPs communicate and synchronize with timestamped messages, PARASOL enables communication and synchronization between LPs via transparent thread migration. Indeed, this transparency leads to greatly simplified model development at the application level and was an important design consideration (Mascarenhas, Knop, and Rego 1995).

PARASOL presents the user with an environment that offers transactions (i.e., dynamic, computational units with some private data) and a set of domain-dependent global objects. All transactions and objects are distributed among the physical processors hosting a simulation. In execution, transactions either perform local computations or access objects. When a transaction accesses an object at a remote host, PARASOL transparently enables the transaction to *migrate* to the remote host, thus enhancing locality.

The basic system architecture is shown in Figure 1. The kernel provides basic support services, and the domain libraries support application-level functionality. The kernel insulates the upper layers from all parallel simulation details, including transaction management, migration, communication, rollback, etc. The kernel is supported from below by

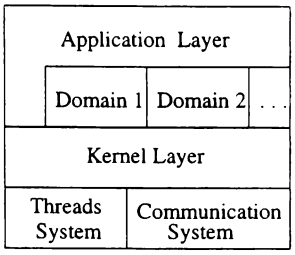


Figure 1: PARASOL's Layered Architecture

the *Ariadne* threads system (Mascarenhas and Rego 1996) and a suitable communications substrate, e.g., PVM (Sunderam 1990). Thread migration is supported by the Ariadne threads system. The kernel programming interface is represented by the public methods of class `PSol`, the main simulation class in PARASOL's C++ interface.

Because the PARASOL project was motivated by the need for parallel simulation in different domains, its kernel layer is designed to support distinct domain libraries. For example, the queuing domain provides functionality (e.g., operations on servers and queues) that is different from the functionality provided by a particle-physics domain (e.g., grid definition, cluster generation, particle movement). Thus, distinct domain-libraries provide interfaces to objects and resources that are domain-specific. A domain that provides high functionality can potentially relieve the user of much programming detail. A user may select domain-specific functions that suit an application, eliminating nontrivial code redesign. Typical domain libraries may include switching systems, particle physics, manufacturing systems, digital logic circuits, and combat simulations. User-level code is developed with the help of domain-layer and kernel-layer functions.

4 STATE-SAVING AND RESTORATION

The State Module in the PARASOL kernel is responsible for saving LP-state at a frequency which depends on run-time data. System state is given by the state of all threads (LP and active-transactions) and objects (associated with LPs or transactions). Thus, in checkpointing system-state, it is necessary to save all data (object) and computation (thread) state. The problems inherent to checkpointing thread state differentiate PARASOL's checkpointing mechanisms from that of other existing parallel simulation systems where only data is saved.

An advantage of our approach is that any data locally declared and modified by transactions is transparently saved and restored as thread state. Objects that are local to threads (stack-allocated local variables and data structures) are automatically saved – as local thread state – whenever a thread is saved. But global objects require explicit save actions. For domain-supported objects, PARASOL's checkpointing mechanism is transparent to the user. Global objects that are foreign to the domain layer must first be registered for state-saving through an invocation to the kernel's `objRegister()` primitive. These user-defined objects must also contain application-defined and kernel-invocable `save()` and `restore()` class methods. Because threads are automatically saved by the kernel, the saving of computation state is handled by PARASOL without user intervention. Domain-level global objects come equipped with with state-saving/restore functions. Thus, users developing code at the application level are spared the expense of state management.

State management overheads are known to have a serious negative impact on the performance of optimistic protocols (Fujimoto, Tsai, and Gopalakrishnan 1988). We describe a novel *save-if-modified* method for state-saving in PARASOL. The pseudo-code for this procedure is shown in Figure 2. In PARASOL, state-saving is performed incrementally and infrequently; incrementally, because instead of saving all threads and objects within an LP at each checkpoint, only those threads that ran, and those objects that were modified during the last interval, are saved. Infrequent state-saving is a result of intermittent checkpointing, where frequency is fixed at a user-specified event count, or varies in an adaptive manner, depending on run-time costs.

The State saving module in PARASOL (i.e., the SSM class) provides a `snapshot()` class function which saves the state of an LP. For each thread and object, the system tracks any state that changes; this is done with the help of a "dirty" flag which identifies all threads and objects that are modified after the last checkpoint (lines 6 and 24 of Figure 2). Dirty threads trigger invocation of the `saveThread()` primitive, enabling the saving of thread context in a buffer. Clean threads or objects force a "reference count", contained in a previously saved state for the same thread or object, to be incremented. This eliminates repeated saving of state (lines 12 and 30). During memory reclamation (i.e., fossil collection), state reference counts are decremented by one, and states with counts of zero are reclaimed.

The system also examines all objects registered with a dirty thread for potential state-saving. The flag of each registered object is examined; if found dirty, the read-write state of the object is saved in a buffer. Otherwise the object's reference count is incremented by one.

```
/* save-if-modified method for saving state in   1
ParaSol SSM is the state saving module */
void SSM::snapshot() {
    if (snapshot has not been taken at this time) {
        for (each active thread in this LP) {
            if (thread is dirty) {                  6
            savedThread = new SavedThread;
            savedThread->saveThread(context);
            hashQSavedThreads->
                insert(savedThread);
            } else {
            savedThread =
              hashQSavedThreads->lookup(id,time);
            savedThread->incrementRefCount();   12
            }
            listOfSavedThreads->
                insert(savedThread);
        }
    }
}
void SavedThread::saveThread(context) {
    imagePtr = context_save(); /* threads system
      support */
    threadContextPtr = context;
    referenceCount = 1;
    listOfSavedObjects = saveObjects(context);  20
}
SavedObjectsList*
  SavedThread::saveObjects(context) {
    for (each object registered with this context) {
        if (object is dirty) {                      24
        savedObject = new SavedObject;
        savedObject->saveObject(objectPtr);
        hashQSavedObjects->insert(savedObject); }
          else {
        savedObject =
          hashQSavedThreads->lookup(id, time);
        savedThread->incrementRefCount();           30
        }
        listOfSavedObjects->insert(savedThread);
    }
}
```

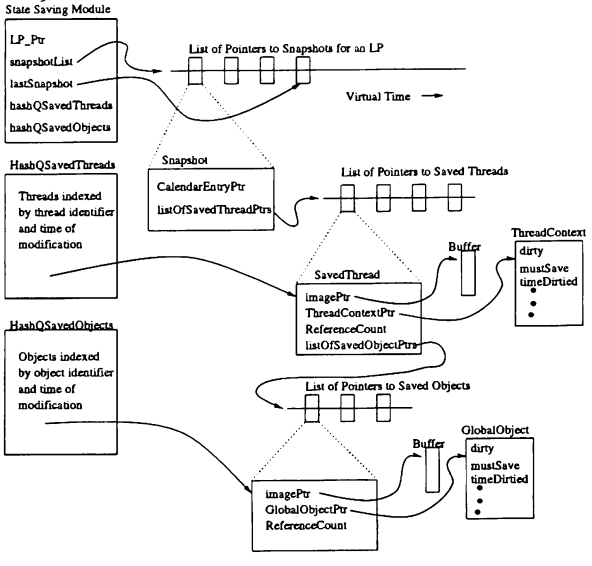


Figure 2: Algorithm and Data Structures Used for Save-if-modified State Saving

In Figure 2 is also shown a summary of the state management data structures and their relationships. The data structures used for storing checkpointed states include a linked list of "snapshots" (each pointing to a checkpointed state) and hash queues of "saved-threads" and "saved-objects", for quick access. Snapshots on the linked list are stored in order of increasing time-stamp. When a causality error occurs, the linked list is traversed to locate the most recent state saved prior to the arrival of the straggler. Since LP states saved after a straggler's timestamp are invalid, these states are discarded during the rollback phase.

Each hash queue is indexed by a thread/object identifier and the last time the thread/object was modified. The hash queue organizations allow for quick lookups. This is useful when a thread/object state reference count is to be incremented or decremented. The hash queues depict how thread and object states evolve over time as the simulation progresses, capturing these states at the snapshot instants.

A snapshot consists of a list of pointers to saved-threads, each of which may contain a list of pointers to the saved state of objects that the thread has registered for state-saving. In the recovery phase, the list is traversed and the saved state of a thread/object is written over the existing state of the thread/object. Saving a thread's state amounts to saving its "image" (context), done with the help of the Thread Interface Module. The Thread Interface Module provides a programming interface to the PARASOL kernel and separates the threads system from the PARASOL simulator. Saving an object's state amounts to linearizing its data; this is done by following its pointers and copying data to a buffer with the help of the ParaState module.

PARASOL's state restoration algorithm is further optimized to restore only necessary threads and objects from a snapshot. The algorithm first checks to determine if the current modification time on the thread or object is different from that specified in the snapshot. If these are different, state restoration occurs; otherwise, the thread or object is *not* restored because its current state is the same as that contained in the snapshot.

4.1 The ParaState Module for State Management

The *ParaState* module was engineered to make the state saving and restoration of complex objects a simple task for domain-level or application-level code developers. The method is also amenable to the automatic generation of routines for object state-saving

and restoration. The module simplifies the task of writing state management routines by unifying procedures for state saving and restoration. That is, the same procedure that saves the state of an object also restores the object's state. A control object, passed as a parameter, specifies whether state is to be saved or restored.

The **PARASTATE** control object, shown in Figure 3, provides a set of basic functions (actually a single C++ template function `paraState()` suffices) that can save and restore simple data objects like `int`, `float`, `char`, etc. Based on these, an additional set of functions operates on arrays, strings, pointers to objects, etc. Using the complete set of routines, objects that contain pointers to other objects, such as singly linked lists, can also be saved and restored. Objects that have circular references, however, cannot be saved without additional effort. An example of saving and restoring an object X which contains a pointer to object Y is shown in Figure 3. The functions `paraStateX()` and `paraStateY()` save and restore objects `X` and `Y` respectively, when invoked by the `save` and `restore()` methods of class `X`. Simple objects `x` and `y` are handled by the `paraState()` template function. The `parasPointer()` call saves and restores the object `Y` pointed to by object `X`.

5 PERFORMANCE

Performance measurements were conducted on two different execution environments: an Intel Paragon and a workstation cluster. The former consists of the XP/S model 10 distributed memory environment, with 142 nodes connected by a two-dimensional mesh backplane topology. Each node has a 50MHz i860XP compute processor with 32MB of memory, and an i860XP message processor. The second execution environment consists of a SPARCstation 5 cluster, connected by an Ethernet. These are 70MHZ SPARC processors with 32 MB memory. For convenience, we label these as the PGON and CLUS environments, respectively.

To test the new methods, we chose two different examples of closed queuing systems. Applications consist of Facilities (where a Facility contains a server and a queue, from the queuing domain) that are initialized with a given number of jobs. Upon completing service at a Facility, a job moves on to another Facility in the network, depending on routing probabilities and paths. In both examples, we assume FIFO queueing at each Facility. Upon leaving a Facility, a job selects a destination Facility uniformly randomly. The size of the configuration – the number of Facilities in the model read from an input file – is a control parameter which is fixed over a run. The

```
enum paraStateOp {
    NOOP = 0, SAVE = 1, RESTORE = 2
};
struct PARASTATE {
    paraStateOp eOp; // operation
    char* pPublic; // users buffer ptr
    char* pBase; // next position
    u_int uiSize; // total user Buffere size
    PARASTATE(const u_int bufSize, const
      paraStateOp op=SAVE);
    PARASTATE(char *buf, const paraStateOp
      op=RESTORE);
};
template <class TYPE> Boolean
  paraState(PARASTATE* pP, TYPE* pI);
extern Boolean parasPointer(PARASTATE*, char**
  ppObj, u_int objSize, const parasProcT
  parasObjProc); /* save object pointed */
/* other primitives are parasArray(),
  parasBytes(), parasString(), etc.  */
/* Example of save()/restore() */
class Y {
    private:  int y;
    public:  Boolean paraStateY(PARASTATE*);
};
class X : public GlobalObject {
private:  int x;
    Y* pY;
public:  void *save(void);
    void restore(void*);
    Boolean paraStateX(PARASTATE*);
};
void * X::save(void)
{
    PARASTATE* pS;
    char *pSaveBuf;
    pS = new PARASTATE(Y_SAVESIZE, SAVE); /*
      controller */
    if (paraStateX(pS) == FALSE) { delete pS;
      return (void*)0; }
    pS->paraSaveDone(&pSaveBuf); // saved object
      is in saveBuf
    return pSaveBuf;
}
void X::restore(void* pBuf)
{
    PARASTATE* pS;
    pS = new PARASTATE((char*)pBuf, RESTORE); /*
      controller */
    if (paraStateX(pS) == FALSE) /* report error
      */;
}
Boolean Y::paraStateY(PARASTATE* pS)
{
    if (paraState(pS, &y) == FALSE) return FALSE;
    return TRUE;
}
Boolean X::paraStateX(PARASTATE* pS)
{
    if (paraState(pS, &x) == FALSE) return FALSE;
    if (parasPointer(pS, (Y**)&y, sizeof(Y),
      paraStateY) == FALSE) return FALSE;
    return TRUE;
}
```

Figure 3: The ParaState Object and Example of Saving and Restoring a Complex Object's State

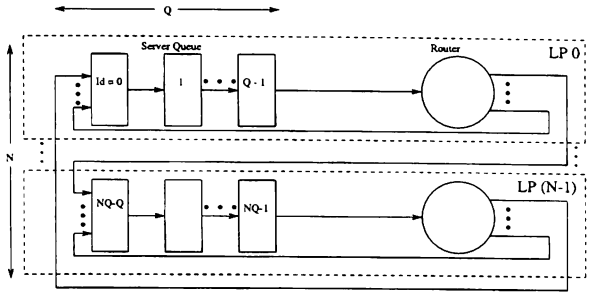


Figure 4: Closed Queuing Network with N Switches and Q Servers per Switch. The N Switches are Mapped onto N LP's, Numbered 0 through $N-1$. Each LP i Contains Q Server Objects, Labeled as iQ through $iQ+Q-1$.

example configurations include:

CQN. This is a closed queuing network configuration, as shown in Figure 4.

TORUS. A torus consists of Facilities arranged in a two-dimensional mesh. Each Facility has four outgoing, and four incoming channels. The probability of a job leaving a Facility on a given outgoing channel can be defined via an input parameter file. Thus, to reduce the number of channels some outgoing probabilities may be set to zero. Unless mentioned otherwise, we use a branching probability of 0.25 on each Facility's four outgoing channels.

Besides application type and network size, other parameters that can be varied include transaction density, average service time, service distribution, and run-length. The transaction density (denoted by TD in the figures) is the ratio of the total number of jobs to the total number of Facilities in the system. Service-time distributions are changeable. In our examples, we use service times that are biased exponentials, i.e., $service\ time = rT + exp((1-r)(T + factor * lpid))$, where $T = 10$ is the average service time, and $r = 0.01$. These parameters yield strictly positive service times. The granularity *factor* in the service time expression allows us to vary the mean service time across LPs. Facilities hosted by an LP with a larger *lpid* will have a larger average service time if *factor* is non-zero.

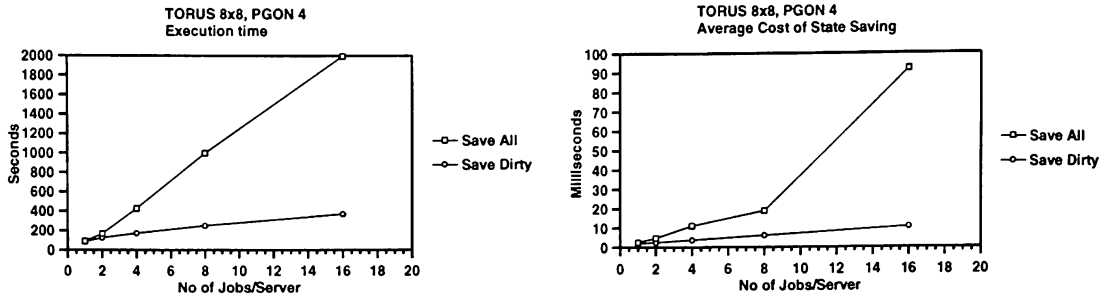
We chose simulation execution time (EXEC), measured by the statistics module in the ParaSol kernel, as our performance metric. Each simulation run is terminated when the global virtual time (GVT) exceeds a specified value. Results for the CLUS environments are averages over ten independent runs. Because of very low variability, results on the PGON environment are from single runs. Experiments were run at times when interference from other users was minimal. Standard errors were found to be low. For example, for the experiments reported next, the standard errors in execution time were less than 1% for the CLUS environment.

Models were partitioned equally across processors. Objects in the model were assigned to processes in a round robin fashion, moving from left to right across the queuing network. The number of LPs per process was set to one, except in the case of CQN, where no more than one complete row of Facilities in a switch was assigned to one LP.

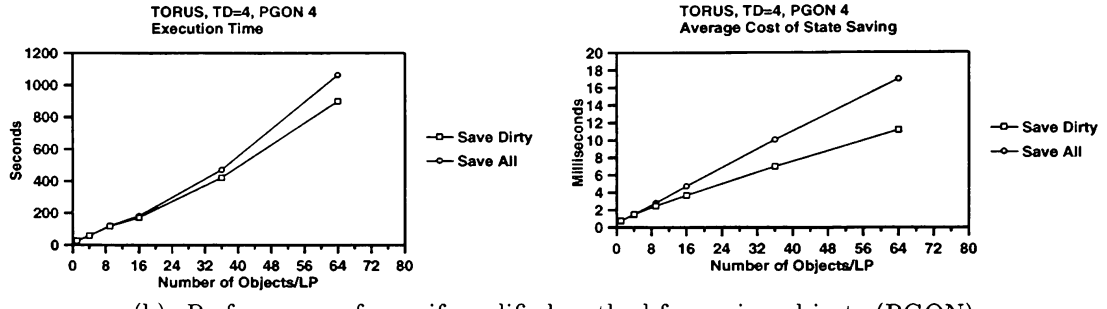
5.1 Performance of the Save-if-modified Algorithm

Figures 5(a) and (c) show reductions achieved in state-saving overhead when only dirty threads are saved (instead of all threads being saved). Figures 5(b) and (d) show state-saving overhead reduction when only dirty objects are saved instead of all objects being saved. Experiments for graphs (a) and (b) were conducted on a PGON environment using a torus, with $factor = 5.0$. When measurements were made for threads, dirty object saving was turned on. Similarly, when measurements were made for objects, dirty thread saving was turned on. When the number of objects in graph (b) is increased, transaction density is kept fixed. The effect is to increase the total number of threads in the system as the number of objects increases. The figure shows that as the number of threads (jobs) and/or objects in the simulation increases, the amount of overhead reduction using the save-if-modified method also increases. The graphs clearly illustrate the efficacy of the save-if-modify algorithm. In this example, the reduction in average cost was as high as 34% with 64 objects per LP, and as high as 89% with 16 transactions/Facility.

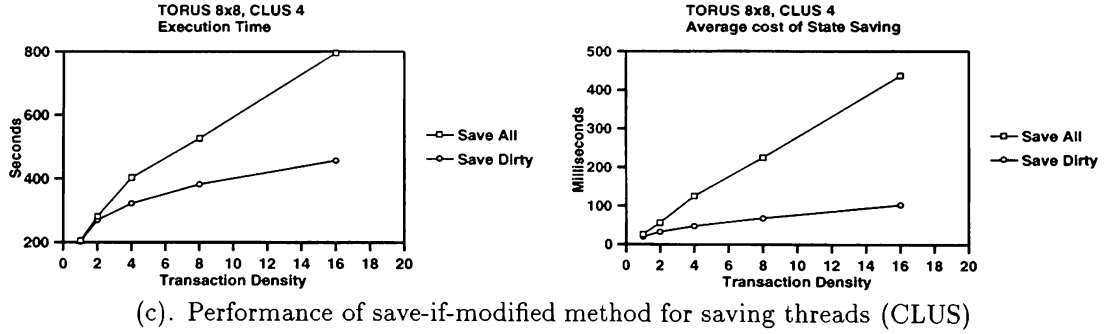
Similar results are observed in the CLUS environment, as shown in Figure 5(c) and (d). The results shown in Figure 5(c) correspond to a TORUS with $factor = 0$. A 79% reduction in average cost is observed with a transaction density of 16. The results shown in Figure 5(d) correspond to a CQN with 8 switches and a fixed total number of threads in the system. As the number of objects is increased, the effect on the average state-saving cost is the same as seen earlier in the PGON environment. The reduction in average state-saving cost is 55% with 64 objects per switch. The execution time curve in Figure 5(d) for CQN is concave upwards, instead of being concave downwards. For this case, as the number of objects is increased, the computation granularity increases and rollback overheads decrease. The result is a reduction in total execution time. With the save-



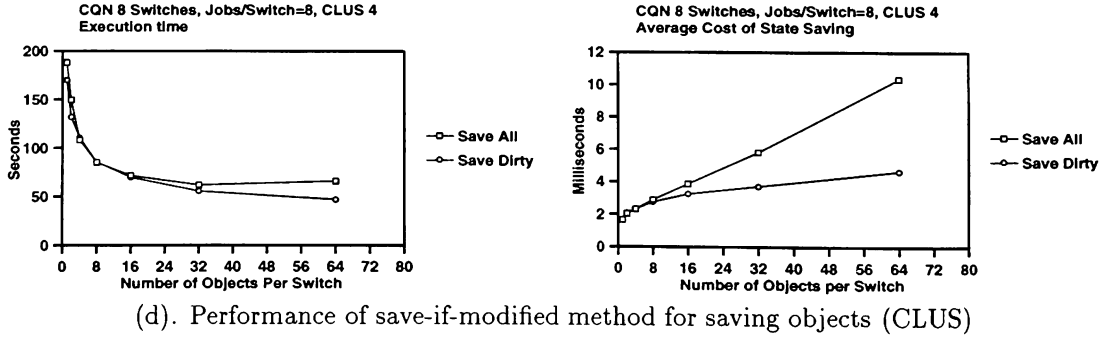
(a). Performance of save-if-modified method for saving threads (PGON)



(b). Performance of save-if-modified method for saving objects (PGON)



(c). Performance of save-if-modified method for saving threads (CLUS)



(d). Performance of save-if-modified method for saving objects (CLUS)

Figure 5: Performance of the Save-if-modified Method

if-modified method, execution time is equal to or less than the execution time with all objects saved.

6 CONCLUSION

Based on experimentation with a queueing domain, we feel that a threads-supported process-based parallel simulator offers serious advantages to the user. Because of our use of threads and optimistic/adaptive synchronization, new methods are required for state-saving and recovery in the PARASOL system. However, unlike existing parallel simulation systems, the threads-support and domain-layering enables PARASOL to provide transparent checkpoint and recovery mechanisms for domain-level objects. User-defined objects that are foreign to a domain may be saved and recovered through support from the system's ParaState module. Our experiments appear to indicate that our state-managment methods reduce overheads, and thus reduce simulation time.

ACKNOWLEDGMENTS

This research was supported in part by ONR-9310233, BMDO-34798-MA, and NSF-CCR 9311862. The second author was supported in part by CNPq-Brazil, grant 260059/91.9.

REFERENCES

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

Fujimoto, R., J. Tsai, and G. Gopalakrishnan. 1988. The roll back chip: Hardware support for distributed simulation using Time Warp. In *Proceedings of the SCS Distributed Simulation Conference*, 81–86.

Gomes, F., B. Unger, and J. Cleary. 1996. Language based state saving extensions for optimistic parallel simulation. In *Proceedings of the Winter Simulation Conference*, 794–800.

Mascarenhas, E., F. Knop, and V. Rego. 1995. ParaSol: A multithreaded system for parallel simulation based on mobile threads. In *Proceedings of the Winter Simulation Conference*, 690–697.

Mascarenhas, E., and V. Rego. 1996. Ariadne: Architecture of a portable threads system supporting thread migration. *Software-Practice and Experience*, 26(3):327–356.

Mascarenhas, E. 1996. A System for Multithreaded Parallel Simulation and Computation with Migrant Threads and Objects. PhD thesis, Department of Computer Sciences, Purdue University.

Ronngren R., and R. Ayani. 1994. Adaptive checkpointing in time warp. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, 94–100.

Steinman, J. 1993. Incremental state saving in SPEEDES using C++. In *Proceedings of the Winter Simulation Conference*, 687–696.

Sunderam, V. S. 1990. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339.

AUTHOR BIOGRAPHIES

EDWARD MASCARENHAS, is a Member of Technical Staff at Silicon Graphics Computer Systems. He received a Masters degree in Industrial Engineering from NITIE (Bombay, India), an M.S. in Computer Sciences from Purdue University in 1993, and a Ph.D. degree in Computer Sciences from Purdue University in 1996. His research interests include parallel computation, distributed simulation, and multithreaded programming environments.

FELIPE KNOP, Ph.D., Computer Sciences Department, Purdue University, (August 1996), received a Masters degree in Computer Sciences from Purdue University in 1993 and a Masters degree in Electrical Engineering from University of São Paulo, Brazil, in 1990. He joined IBM, RS/6000 Scalable POWERparallel division, in August 1996. His current research interests include parallel and distributed simulation, and multiprocessor operating systems.

REUBEN PASQUINI is a Ph.D. student in Computer Sciences at Purdue University. He received his B.S. in computer engineering from the University of Illinois at Urbana Champaign in May, 1994 and his Masters in Computer Science from Purdue University in December, 1996. Reuben was awarded an Intel Fellowship in June, 1997. His research interests include parallel discrete event simulation.

VERNON REGO is a Professor of Computer Sciences at Purdue University. He received his M.Sc.(Hons) in Mathematics from B.I.T.S (Pilani, India), and an M.S. and Ph.D. in Computer Science from Michigan State University (East Lansing) in 1985. He was awarded the 1992 IEEE/Gordon Bell Prize in parallel processing research, and is an Editor of *IEEE Transactions on Computers*. His research interests include parallel simulation, parallel processing, modeling and software engineering.