Scalability of Parallel Simulation Cloning

Maria Hybinette

Computer Science Department
University of Georgia
Athens, GA 30602-7404, USA

maria@cs.uga.edu

Richard M. Fujimoto

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280, USA

fujimoto@cc.gatech.edu

Abstract

In previous work we presented an algorithm for cloning parallel simulations that enables multiple simulated execution paths to be explored simultaneously. The method is targeted for parallel discrete event simulators that provide the simulation application developer a logical process (LP) execution model. The cloning algorithm gains efficiency by cloning logical processes only as necessary. In this work we examine the scalability of cloning in detail. Specifically, we examine how the number of clones impacts the performance of cloning as we vary the "size" of the simulation problem.

1 Introduction

Parallel processing has traditionally been applied to executing discrete event simulation programs in at least two different ways. The first, called parallel discrete event simulation, involves distributing the execution of the simulation program across multiple processors in order to reduce execution time. A major challenge is ensuring the simulation is properly synchronized, and a variety of techniques have been developed to address this problem (Fujimoto 2000). The second approach, referred to as replicated trials, involves executing independent simulation runs on different machines. This approach yields trivial parallelization because there is no need to address synchronization issues during the execution of the simulation programs. Both techniques are useful in different situations. Parallel discrete event simulation is applicable if simulation results are required quickly (e.g., for designers trying to understand the behavior of a system) or the simulation requires more memory than is available on a single machine. Replicated trials is the preferred approach if throughput is the primary goal, e.g., completing many runs to evaluate different parameter settings where users are willing to wait possibly long periods of time for the different runs to complete.

Here, we are concerned with the simultaneous application of *both* of these techniques. We envision parallel discrete event simulation techniques to be used to reduce the execution time of each run to an acceptable level, and exploitation of replicated trials to complete as many simulation runs as possible given the available hardware resources and time to yield results.

Further, we seek to achieve additional speedup beyond that which each of these techniques can achieve. Both parallel discrete event simulation and replicated trials can achieve at most N-fold speedup using N processors, except in rare situations where superlinear speedup can be obtained due to (for instance) memory caching effects. Obtaining greater speedup requires the use of more sophisticated techniques. We focus on one such technique called *simulation cloning*. As discussed momentarily, the basic idea behind cloning is to share simulation computations that are common among multiple runs. These computations are performed only once, and their results are shared across the multiple runs. Simulation cloning was proposed in prior work (Hybinette and Fujimoto). Here, we extend this work to examine scalability issues concerned with the cloning technique.

The paper is organized as follows: The cloning mechanism is briefly described next. We then describe its implementation in Section 3. Next, we discuss the scalability of cloning. We conclude with a summary and a discussion of future directions.

2 Simulation Cloning

Simulation cloning is a technique that replicates a running sequential or parallel simulation program during its execution in order to explore different possible futures of the simulation. It can be used, for instance, to concurrently explore alternate courses of action to deal with emerging events. For example, in an air traffic control setting, one might wish to use simulation to compare alternate approaches to managing the flow of aircraft when inclement weather is forecast for one portion of the air space. This may require controllers to restrict the number of aircraft entering this portion of the airspace. Operationally, this is handled by increasing the spacing between aircraft (called the *miles in trail* or *MIT*) restriction. One might wish to evaluate the overall impact of such restrictions on the flow of traffic throughout the entire traffic network, since delaying certain flights will have a "ripple" effect that propagates throughout the system.

In this example, the simulation can be initialized with the current state of the traffic space, and executed forward until reaching the point where new restrictions are to be imposed. The simulation can then be cloned (replicated), with each clone simulating the traffic space with a different MIT restriction. The point where the simulation is cloned is referred to as a *decision point*. The cloned simulations can execute concurrently, and will produce identical results as a traditional replicated simulation experiment where the entire simulation is executed, from the beginning, using different MIT restrictions that are imposed at the time of the decision point.

Simulation cloning can improve the performance of the simulation in two different ways. First, it is clear that the computation prior to the decisions point is performed only once, and its results are shared among the different clones. This is in contrast to a replication experiment where this computation will be repeated for each replication.

Second, it is often the case that there is much computation that is common among the clones, even *after* the decision point has been reached. For example, traffic congestion in the eastern part of the U.S. will not affect traffic on the west coast for some time. Therefore, the simulation of air traffic in the west coast will be identical immediately after the clones are created. One would like to also perform these computations only once, and share their results, rather than repeat them within the different clones.

A technique called *incremental cloning* has been developed to allow computations after the decision point to be shared among the different clones (Hybinette and Fujimoto). The basic idea in incremental cloning, elaborated upon below, is to provide mechanisms to detect when portions of the cloned simulations diverge from each other, and replicate portions of the simulation only as needed.

The incremental cloning algorithm assumes a message-based computation paradigm like that commonly used in parallel discrete event simulations. Specifically, the simulation is composed of a collection of *logical processes* (LPs) that communicate exclusively by exchanging time stamped events or messages. A synchronization algorithm is used to ensure that each LP processes its events in time stamp order, or in the case of *optimistic* simulation protocols, the net effect of each LP's computation is as if its events were processed in time stamp order. The incremental cloning algorithm described here can be used with either conservative or optimistic synchronization techniques.

The cloning mechanism is implemented by defining an abstraction called a *virtual simulation*. A virtual simulation is defined for each clone. Virtual simulations are composed of *virtual LPs* that communicate by exchanging *virtual messages*. As its name implies, virtual simulations are not "real" in the sense that they do not include memory to hold state, nor simulation code. Rather, virtual LPs and virtual messages map to *physical LPs* and *physical messages* that perform the actual simulation computations.

Virtual and physical simulations are analogous to virtual and physical memory. Like virtual memory, a virtual simulation appears to the user as an actual running simulation. However, in actuality, the virtual simulation only consists of data structures that map the virtual simulation to physical LPs and messages. Each virtual simulation is referred to as a *version* of the original simulation. Replicating a virtual simulation requires very little computation since one need not replicate the state of the simulation; rather, only a few data structures need to be updated to create a new version (clone). Each virtual LP and each virtual message maps to exactly one physical LP and one physical message, respectively.

When a new version (clone) of the simulation is created, all of the virtual LPs making up the version are replicated. However, as mentioned earlier, replicating virtual LPs requires very little computation because one need only update a few internal data structures. Only those *physical* LPs that are different in the two clones need be replicated when the clone is created. Replicating a physical LP is a more expensive operation, requiring replication of the LP's state variables. Virtual LPs that are identical in the two clones map to the same physical LP, while those that are different in the clones map to different physical LPs.

Computation sharing is accomplished by mapping virtual LPs corresponding to different virtual simulations (clones) to the same physical LP. For example, in the air traffic example, suppose an LP is defined to model each airport. Suppose two clones are created to evaluate alternate policies for managing inclement weather on the east coast. In this case, two virtual simulations are created. The virtual LPs corresponding to airports on the west coast that are not immediately impacted by the inclement weather are mapped to the same physical LPs. Computations by this physical LP are automatically shared between the two clones. The LPs modeling east coast airports that are immediately impacted by the inclement weather are realized by distinct LPs, since their computations in the two clones will differ.

As time goes on, the simulations in the western part of the U.S. for the different clones will typically diverge. The incremental cloning algorithm replicates LPs as their behavior diverges. The incremental cloning software can detect when the computations diverge by monitoring the messages sent among the LPs. For example, suppose there are two (virtual and physical) LPs modeling the Atlanta airport in the two clones (See Figure 1). Suppose the LP modeling Orlando is realized by two virtual LPs that map to *the same* physical LP because Orlando has not yet been affected by the inclement weather. Suppose the Atlanta LP in the first

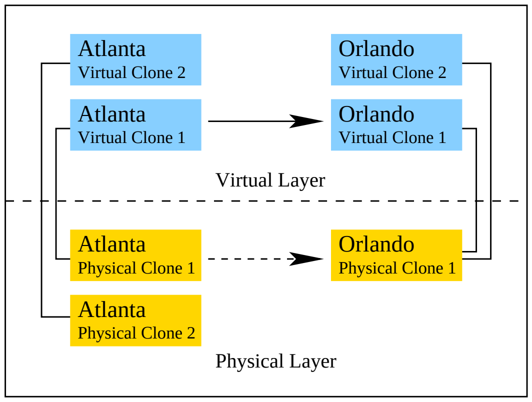

Figure 1: An instance in an air traffic control simulation: Atlanta LP in the first clone (virtual simulation) sends a message to Orlando indicating a new flight arrives at Orlando at 5 PM.

clone (virtual simulation) sends a message to Orlando indicating a new flight arrives at Orlando at 5 PM, but no such message is generated in the second clone because that flight has been delayed in that execution. The cloned simulations for Orlando will thus diverge at simulated time 5 PM. The cloning mechanism, which can be implemented as a layer of software that resides between the simulation application and the underlying simulation executive, can detect this, and replicate the Orlando LP at 5 PM. In this way the cloned simulation is incrementally replicated as portions diverge from each other.

The cloning mechanism intercepts the sending and delivery of messages from and to the application. It must determine which physical LPs must receive messages that are sent. It also determines whether physical processes and/or messages must be cloned, and invokes primitives to clone these portions of the simulation as needed. This requires knowledge of the versions of the simulation that are sending/receiving the message. This information is provided to the cloning software by appending the *virtual send* and *virtual receive* set of processes to each message.

The virtual send and receive set information attached to each message is used to determine when processes must be cloned. A physical logical process is cloned if (1) there is a virtual logical process in the receive set that should not be influenced by the incoming message or if (2) the destination address of the message is a physical logical process that has not yet been created.

Details of the cloning mechanism and its implementation are beyond the scope of this paper, but are described elsewhere (Hybinette and Fujimoto). Here, we are primarily concerned with the scalability of these cloning mechanisms. Before proceeding to the discussion on scalability we discuss how cloning is implemented.

3 Cloning Implementation

The goals for implementing cloning are:

- efficiency,
- transparency and
- simulator independence

Efficiency is in terms of the number of alternatives evaluated in a time-constrained period and memory resource usage. Efficiency is achieved by enabling multiple scenario analysis and allowing different versions of the simulation to share computations between each other. Transparency is with respect to the simulation application and is accomplished by monitoring preexisting primitives (*send* and *receive*). Simulator independence refers to the choice of optimistic or conservative synchronization. Clone-Sim provides simulator independence with respect to this framework.

A simulation consists of a simulation application (provided by the user) and a simulation executive that implements the synchronization protocol. The simulation executive provides primitives that allow simulation programmers to define their own applications. This is a layered system, with the operating system at the bottom, the simulator executive in the middle and the simulation application at the top (See Figure 2).

The cloning mechanism is implemented in a package called Clone-Sim. Clone-Sim enables on-demand "cloning" of parallel and distributed discrete event sim-

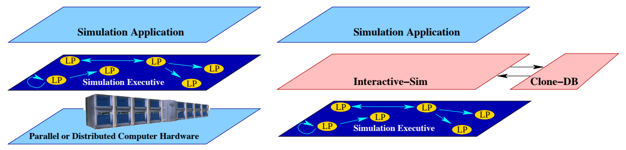

Figure 2: Views of simulations: Traditional parallel discrete event simulation is shown on the left; the monitoring layer called interactive-sim in relation to the simulation executive and the simulation application is shown on the right.

ulations. Clone-Sim achieves efficient cloning by intercepting the communication primitives between the user application and the simulator executive. The package can be used in interactive as well as non-interactive environments. Both optimistic and conservative simulators can be supported. Currently, Clone-Sim has been implemented with Georgia Tech's Time Warp simulation executive (Das et al. 1994) called GTW, an optimistic simulator.

Clone-Sim consists of two modules: the **Interactive-Sim** module which is layered between the simulation executive and the application simulation program and the **Clone-DB** database that is independent of the synchronization primitives of the simulation executive. The key function of Interactive-Sim is (1) to intercept message sends and (2) to process events. For example, Interactive-Sim needs to know the message send and message receive primitives in order to intercept the invocation to process events or to forward copies of a message to cloned LPs. After interception, Interactive-Sim queries Clone-DB to determine message or process cloning (via respective inquire functions). The architecture of Clone-Sim is shown on the right of Figure 2. From the point of view of the simulation executive, Interactive-Sim is a simulator application. In the context of the layered system Interactive-Sim is between the simulation executive and the user's application simulation program. Interactive-Sim itself is decomposed into sub-modules, where each is implemented for a particular simulation executive. The sub-modules are "pluggable" in that the appropriate submodule is plugged in for a specified simulation executive. New sub-modules can be implemented using a specified application interface. The general idea behind Interactive-Sim is that it is transparent to the simulation program, and also to the programmer utilizing the cloning primitives.

4 Assumptions

We assume the simulation executive provides `ScheduleEvent` (send and schedule) and `ProcessEvent` (receive) primitives. The relationship between the user application and the simulation executive is illustrated on the left of Figure 2. Here, the user application defines the events and the simulation executive manages synchronization. If the application uses GTW as a simulation executive, it must define `ProcessEvent`, and tell GTW when to schedule the event. GTW then schedules a call to `ProcessEvent` at the appropriate time.

Clone-Sim also assumes that the simulation executive supports dynamic LP creation and allocation, initialization and copying of LPs. It is assumed that one can schedule events conservatively, i.e. the event can be scheduled with the assumption that it will never roll back, (this is trivial if the simulation executive is conservative).

Another assumption is that simultaneous events are addressed by the underlying simulation engine (e.g. by prioritization). Clone-Sim ensures, that simultaneous events do not interfere with cloning. This is done by prioritizing cloning events (such as instantiations of new simulations and physical processes) above all other events. To summarize, Clone-Sim assumes that the simulation executive supports:

- a *send and schedule* primitive (ScheduleEvent)
    - including the capability to schedule an event conservatively,
- a *process event* primitive (ProcessEvent), and
- a capability to clone individual logical processes and
- simultaneous events

The next section will describe the simulation application programmer interface to the Cloning functions. These are the primitives that enables a programmer to clone simulations.

5 Simulation Application

Clone-Sim accounts for the mapping of virtual LPs to physical LPs by assigning identifiers to each physical LP. There are three types of identifiers:

- unique
- global and
- simulation.

The unique identifier (`UID_LP`) distinguishes physical logical processes. In addition to a unique identifier a physical logical process is assigned to a global identifier (`GID_LP`), the same global identifier may be shared by multiple physical logical processes. The `GID_LP` corresponds to the original physical ID (before cloning) of the LP at time 0 (if this physical logical process is cloned the GID_LP may corresponds to multiple physical processes). Finally each simulation or cloned simulation is associated to a unique simulation identifier (Clone_ID). Each version of a simulation consists consists of a set of virtual LPs. These identifiers can be accessed by specified functions that are described below.

There are six cloning functions available to the simulation application. These are primitive functions; complex cloning scenarios are developed by composing the primitives appropriately. We list the function names below, then describe them in detail later. The API functions are

`void CloneSim_InitAppl`(int argc, char ** argv)
`void CloneSim_CloseAppl(` void)
`int CloneSim_Create(` int UID_LP, double current_sim_time)
`int CloneSim_Delete(` int clone, double start, double end, CSFunc_p trig)
`int CloneSim_GetCloneID(` int LP)
`int CloneSim_GID(` void)
`int CloneSim_UID(` void)

The utility of each function depends on the state of the simulation.

We view the state of the simulation as moving through three phases: initialization, execution, and wrap-up. The initial number of logical processes, the mapping between processes and processors, and event handler specification is determined at initialization. Events are scheduled and event handlers are called at appropriate times during the execution phase. In the wrap-up phase the simulation is complete and cleanup functions are called before the application terminates. These phases are described in more detail below.

5.1 Initialization Phase

Clone-Sim assumes that the simulation executive allows the user application to set up the assignment of LPs to processors. The assignment is thus exposed and Clone-Sim can manipulate it. This is important, because Clone-Sim exploits this ability to maintain LP assignments transparently from the simulation application's point of view.

To initialize Clone-Sim,
`void CloneSim_InitAppl()` is called at *the end* of the initialization phase. The function has two arguments: `argc` and `argv` that specify command line parameters for Clone-Sim. Initializing Clone-Sim sets up data structures that: (1) control the mapping between logical processes and processors, (2) provide buffer space for cloned physical logical processes and (3) determine message or process cloning or determine child, sibling, and parent relationships between cloned simulations.

The assignment of logical processes to processors is assumed to be static after the initialization phase, however the mapping can be easily made to be dynamic if these systems include load balancing functions. An example initialization is shown below:

```
void InitializationPhase( int argc, char **argv )
  {
  /* code initialization is defined here */

  /* call cloning initialization procedure */
  CloneSim_InitAppl( argc, argv );
  }
```

5.2 Execution Phase

During the execution phase, cloning allows for the insertion and deletion of decision points via the cloning primitives `CloneSim_Create()` and `CloneSim_Delete()`. This can be implemented interactively or non-interactively by the simulation programmer. An event that clones a simulation must be **conservative** (guaranteed to never rollback). If the decision point occurs on a set of LPs then a conservative event must be scheduled at the same simulation time by each of the LPs in the set. Successive LP clones within a new version may occur optimistically and may be rolled back.

A call to `CloneSim_Create` returns the identification number of the newly cloned simulation, so that one can refer to the clone when deleting or pruning it. A negative number is returned upon error.

This (decision) point represents the location in the execution path where the state of the newly created version starts to diverge from the version that called it. When the function is called, the physical logical process calling the primitive is replicated. Any assignment to variables or calls to functions within this conservative event *after* the call *only affect the original clone.* Assignment or calls to functions within the conservative event before `CloneSim_Create()` affect both versions of the simulation: the newly created clone and the original clone.

During the execution phase, Clone-Sim provides access to three LP identifiers that are helpful to the user:

1. **unique:** the unique identifier `UID_LP` can be used to distinguish between all physical logical processes.
2. **global:** each logical process is assigned a global identifier, `GID_LP` that corresponds to the ID of the original LP (before cloning) to which the current LP corresponds. The same global identifier may be shared by multiple physical logical processes.
3. **simulation:** the `Clone_ID` identifies the version of the simulation in which the LP is running.

These identifiers can be accessed using the corresponding functions described above.

An example use of the function that creates a simulation is included below:

```
void A_Conservative_Event( arguments )
  {
  int   unique_LP_identifier;
  int   clone_identifier;

  /* code that effects both original LP and
     instantiated LP below */

  /* access the unique logical process identifier
     of callee */
  unique_LP_identifier = CloneSim_UID();

  /* instantiates a new clone, a new logical process
     is created */
  clone_identifier
      = CloneSim_Create( unique_LP_identifier,
                         current_sim_time );

  /* code here and below only effects caller LP
     of original simulation.  The new LP created via
     the Clone_SimCreate is un-effected */
  }
```

In addition to creating clones, Clone-Sim provides a mechanism to eliminate simulations that are not needed. This is done by installing a "trigger". To install a trigger, the application should call `CloneSim_Delete()`. The function can be called interactively or non-interactively. The prototype to delete a simulation is:

> `int CloneSim_Delete(` int clone double start double end CSFunc_p trigger `)`

The trigger is a condition defined by the argument **trigger** that is sampled within the simulation period specified by the arguments: `start`, and `end`. The installation of the trigger only effects the logical process that installs it and only the simulation whose version is given by the first argument: `clone`. So if all versions in the simulation need to be monitored the trigger needs to be installed for each version. If trigger is `NULL` the version that calls `CloneSim_Delete()` is pruned un-conditionally (at the appropriate time given by the argument specifying the time period). Currently, the pruning function only provides un-conditional pruning, conditional pruning is only available in an un-released version of Clone-Sim.

An example use of the function that deletes a simulation is included below:

```
void Some_Event( arguments )
  {
  /* simulator dependent code here */

  if( some condition )
    {
    /* prune if the simulation time of the callee
       is within the simulation period:
          [0.00, END_TIME] */
    CloneSim_Delete( cloneID, 0.00, END_TIME, NULL );
    }

  /* simulator dependent code here */
  }
```

5.3 Wrap-up Phase

When the simulation completes `CloneSim_Close(` void `)` is called to clean up data structures and compute statistics. It is called after the simulation code has completed and before terminating the program. The prototype of this function is defined below:

> `int CloneSim_Close(` void `)`

6 Scalability

Here, we investigate scalability in terms of problem size. We evaluate the performance of our system as the size of the problem increases. For a replicated simulation application with N execution paths, each LP is duplicated N times. In many cases, messages are duplicated N times as well. On average, we expect a replicated simulation to consume N times the resources (in terms of space and time) as a simulation with a single execution path.

When evaluating the performance and scalability of a cloned simulation it is important to keep in mind that there are three key phases of an application's "life cycle" as follows:

- **Before decision point:** in this phase the simulation has not been cloned, and there is a single execution path. It is during this phase that cloning offers a tremendous advantage over replicated simulations. All events processed in this phase run once, whereas in the replicated case the events must be processed redundantly in each copy of the simulation.
- **Spreading:** our approach uses an incremental method that only duplicates (clones) LPs as necessary. At the decision point only a small number of LPs must be replicated. In this phase a large proportion of computations can be shared, and the approach retains a significant advantage over replicated simulations. However, as the replicated LPs interact with other LPs it becomes necessary to clone the affected LPs as well.
- **Spreading complete:** eventually all LPs will have been cloned. From this point on, the fully cloned simulation must process the same number of events as a replicated simulation. The additional overhead of cloning in this case is a single comparison per event. Cloned simulations in this

phase are only slightly more expensive than replicated simulations.

The performance improvement of a cloned simulation in comparison to replicated simulations depends on the relative length of each of these phases. For instance, if the decision point occurs late in the simulation, the cloned simulation would have a tremendous performance advantage over replicated simulations. On the other hand, the worst case for cloned simulations is an early decision point combined with rapid spreading. In this case the cloned simulation's performance will be slightly worse than a replicated simulation (an extra comparison for every event).

In this research we consider how the "size" of the simulation effects scalability. The size of a simulation problem, or application, is reflected in the number of LPs used and the number of messages generated. Cloning does not introduce additional dependencies between LPs and can only reduce the number of messages in the system relative to a replicated approach. Further it does not impact how the messages are received and processed. For these reasons, the cloning algorithm will scale to the extent that the underlying simulation engine scales as the size of the simulation is increased.

7 Performance

To evaluate how the number of clones impacts performance, a suite of experiments were run on an SGI Origin 2000 with sixteen 195 MHz MIPS R10,000 processors. The first level instruction and data caches are each 32 KB. The unified secondary cache is 4 MB. The main memory size is 4 GB.

The experiments use 4 processors unless otherwise indicated. The experiments use a benchmark called P-Hold. P-Hold provides synthetic workloads using a fixed message population. Each LP is instantiated by an event. Upon instantiation, the LP schedules a new event with a specified time-stamp increment and destination LP (for more details on P-Hold see (Fujimoto 1990)). Each data point in these results represents at least 15 runs.

In previous work we have evaluated scalability with respect to the number of clones and compared its performance to replicated simulation (Hybinette and Fujimoto). These results showed that as the number of clones increases, run time increases linearly. It also showed that in all cases, cloning is more efficient than replication. The advantages of cloning over replication scale with respect to the number of clones. Cloning also performs similarly using a commercial air-traffic control simulation called the Detailed Policy Assessment Tool, or DPAT. For more detail on DPAT please see (Wieland 1998). In the previous work we also showed that the algorithm did not have any sequential bottlenecks (i.e. speedup is linear with regard to the number of PEs). In this paper we extend the investigation of scalability further by varying additional parameters with the number of clones.

7.1 Number of LPs

To examine how the performance of cloning scales with respect to the number of LPs we vary the number of LPs and the number of clones and evaluate overall run time. The number of LPs was varied from 4 to 512, while the number of clones was varied from 1 to 32.

The performance of cloned P-Hold simulations are shown in Figure 3. These simulations were run using 4 processors on a SGI Origin. They use a fixed message population of 128 and run for 300 simulated seconds. Performance is measured as the running time of the simulation excluding initialization and wrap-up time. Clones are generated at 10 seconds of simulated time. Other parameters are set so that the spreading is rapid — in order to emphasize any overhead introduced by the cloning mechanism. Each point in the plot represents the average of 5 runs. The plot shows that run time increases linearly with the number of clones.

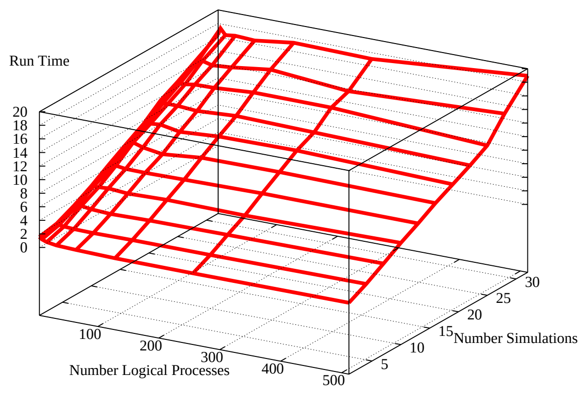

Figure 3: Cloning scales with the number of LPs. Performance is measured as the run time of the simulation, excluding initialization and wrap-up. Smaller numbers indicate better performance.

This suggests that the cloning scales with the number of clones and does not incur any significant overhead as the number of LPs in the simulation increases. Note that the simulation parameters ensure the *worst case* for cloned simulations; we would expect even better performance if cloning occurs later in simulated time or spreading is slowed.

7.2 Message Population

The plot in Figure 4 shows how message population impacts performance as the number of clones increase (increasing the message population increases the work required of each LP in the simulation). In this experiment, the message population varies from 64 to 2048, while the number of clones varies from 1 to 32. These simulations were run using 12 processors on an SGI Origin and 128 logical processes.

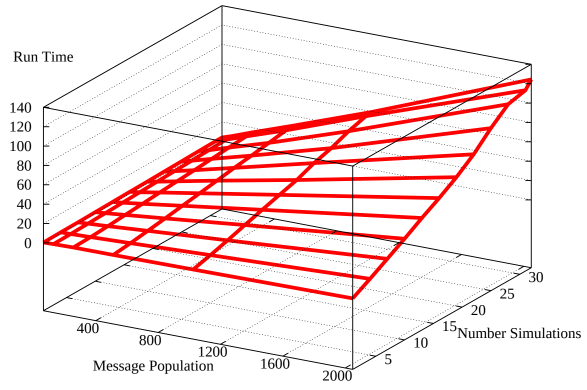

Figure 4: Cloning scales with the message population. Performance is measured as the run time of the simulation, excluding initialization and wrap-up. Smaller numbers indicate better performance.

The plot shows that run time increases linearly with the number of clones, and for each cloned simulation the running time doubles as the workload is doubled (the slope is linear). Also note that the slope is steeper as the message population increases (i.e. there is a larger difference between the run time of 1 clone and 32 clones with a message population of 2048 than the difference between the run time of 1 clone and 32 clones at a message population of 64). As in the previous experiment, these plots show a linear relationship between performance and the number of clones.

8 Related Work

Cloning was suggested by von Neumann (von Neumann 1956) more than 40 years ago to provide fault-tolerance. Cloning is also suggested as a solution for concurrency control in real-time databases (Bestavros 1994) and to improve accuracy of simulation results (Glasserman et al. 1996; Vakili 1992). In the latter, the approach is to run multiple independent replications then average their results at the end of the runs.

Related work in interactive parallel discrete event simulation is described in (Steinman 1991) and in (Franks et al. 1997). In (Steinman 1991) a message type called an *external* event enables interaction with the simulator. These events can steer, sample and query the simulation in progress. The approach of (Franks et al. 1997) allows for the testing of what-if scenarios provided they are interjected before a deadline. Alternatives are examined one after the other and the simulation must undo the effect of the previous alternative before considering another. In contrast to these methods, our algorithm dynamically creates and evaluates multiple alternatives concurrently using a cloning mechanism. New versions of a simulation in progress are cloned to evaluate alternatives. The alternative simulation proceeds in parallel with the original. Thus an increasing number of alternatives can be evaluated before resolution.

The incremental update schemes of process migration algorithms such as (Zayas 1987) are similar in philosophy to our virtual logical process scheme (covered in a later section). The common goal is to reduce the cost of copying the virtual address space between clones. The process migration algorithms differ in that only one active clone is supported while we allow for multiple clones. Our main motivation is to develop a parallel model that supports an efficient, simple, and effective way to explore and compare alternate scenarios.

9 Summary

Simulation cloning enables the exploration of multiple possible futures in interactive parallel simulation environments. The mechanism may be applied to simulations using either conservative or optimistic synchronization protocols.

In this paper we discuss the programmer application interface of cloning parallel simulations. We also demonstrate that cloning scales with the "size" of the simulation. The size of a simulation problem, or application, is reflected in the number of LPs used and the number of messages generated.

Acknowledgments

This work was supported by U.S. Army Contract DASG60-95-C-0103 funded by the Ballistic Missile Defense Organization and the MITRE Corporation.

References

Bestavros, A. 1994. Multi-version speculative concurrency control with delayed commit. In *Proceedings of the 1994 International Conference on Computers and their Applications* (1994).

Das, S., Fujimoto, R., Panesar, K., Allison, D., and Hybinette, M. 1994. GTW: A Time Warp system for shared memory multiprocessors. In *Proceedings of the 1994 Winter Simu-*

lation Conference Proceedings (December 1994), 1332–1339.

Franks, S., Gomes, F., Unger, B., and Cleary, J. 1997. State saving for interactive optimistic simulation. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation (PADS-97)* (1997), 72–79.

Fujimoto, R. M. 1990. Performance of Time Warp under synthetic workloads. In *Proceedings of the SCS Multiconference on Distributed Simulation*, Volume 22 (January 1990), 23–28. SCS Simulation Series.

Fujimoto, R. M. 2000. *Parallel and Distributed Simulation Systems*. John Wiley & Sons, New York, NY.

Glasserman, P., Heidelberger, P., and Shahabuddin, P. 1996. Splitting for rare event simulation: Analysis of simple cases (December 1996). 302–308.

Hybinette, M. and Fujimoto, R. M. Cloning parallel simulations. submitted.

Steinman, J. 1991. SPEEDES: Synchronous parallel environment for emulation and discrete event simulation. In *Advances in Parallel and Distributed Simulation*, Volume 23 (January 1991), 95–103. SCS Simulation Series.

Vakili, P. 1992. Massively parallel and distributed simulation of a class of discrete event systems: A different perspective. *ACM Transactions on Modeling and Computer Simulation 2*, 3, 214–238.

von Neumann, J. 1956. *Probabilistic logics and the synthesis of reliable organism from unreliable components*. Princeton University Press.

Wieland, F. 1998. Parallel simulation for aviation applications. In *Proceedings of the IEEE Winter Simulation Conference* (December 1998), 1191–1198.

Zayas, E. 1987. Attacking the process migration bottleneck. In *Proceedings of the 11th ACM Symposium on Operating System Principles* (1987), 13–24.