Updateable Simulation of Communication Networks

Steve L. Ferenci, Richard M. Fujimoto, Ph.D., Mostafa H. Ammar, Ph.D.,Kalyan Perumalla, Ph.D.
College Of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280
{ferenci,fujimoto,ammar,kalyan}@cc.gatech.edu

George F. Riley, Ph.D.
Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30032-0280
riley@ece.gatech.edu

Keywords: shared computation, event reuse, event composition, incremental simulation

ABSTRACT

A technique called updateable simulations is proposed to reduce the time to complete multiple executions of a discrete event simulation program. This technique updates the results of a prior simulation run rather than re-execute the entire simulation to take into account variations in the underlying simulation model. A framework for creating updateable simulations is presented. This framework is applied to the problem of simulating a set of cascaded ATM multiplexers and a network of ATM switches. Performance measurements of sequential and parallel implementations of these simulations on a shared memory multiprocessor are presented, demonstrating that updateable simulations can yield substantial reductions in the time required to complete multiple simulation runs if there is much similarity among the runs.

1. Introduction

It is almost always the case that multiple executions of a simulation program are required to develop conclusions from a study. For example, sensitivity and perturbation analysis involve incrementally tweaking a parameter of the model in order to determine what affects these changes have on the simulation results. This usually requires many executions of the simulation in order to thoroughly explore the problem space. It becomes prohibitively time consuming to perform such analyses if each simulation run requires many hours or days to complete.

In other situations, previously completed simulation analyses may have to be updated in order to incorporate new information. For example, *on-line simulations* can be used to predict the future behavior of an operational system such as a communication network in order to guide management decisions. Events such as new, unexpected traffic loads call for simulation analyses to be repeated. A means to quickly update the prior results, rather than completely re-execute the simulations would be very beneficial.

The problem of completing multiple independent simulation runs for perturbation and sensitivity analysis lends itself to trivial parallelization by simply performing each run on a different machine. Here, we focus on more sophisticated techniques to gain additional performance improvement, and to attack the problem of quickly redoing previously completed studies in on-line simulation applications.

We observe that when multiple runs are required, the runs are often similar. It is reasonable to believe that there may be many computations that are similar among the different runs. For example, consider a packet-level simulation of an ATM network where different runs vary the buffer size of certain switches in order to examine the effect of this parameter on packet loss rates. Traffic generation computations may be identical across the different runs. If buffer overflows seldom occur (the usual case for most networks), buffer size will have little impact on much of the simulation computation of each switch. Traditional parallel replication techniques do not exploit this fact.

In this paper a technique for improving the performance of discrete event simulations by reusing event computations is presented. This approach focuses on reusing previously completed computations. The premise for this work is that consecutive executions of models will have portions of the computation that are similar or identical.

Section 2 summarizes related work in this area. The updateable simulation technique is described in section 3. Section 4 describes an implementation of this technique to realize an *updateable* ATM multiplexer simulation and Section 5 describes an implementation to realize an updateable simulation of a network of ATM switches. Measurements of a sequential and parallel implementation of the algorithm are presented. Section 6 outlines future work, and is followed by concluding remarks.

2. Related Work

Techniques for reusing or sharing computation have been proposed before. The standard clock technique in [1] is an implementation of a Single Clock Multiple Systems simulation. Certain computations among executions of the simulations are shared under the assumption that the time stamp assigned to events remains the same across the different

runs. A sample path of the simulation is pre-generated and stored. Changing a parameter of the simulation and then using a state update mechanism to construct the new simulation execution from the stored data can generate new executions of the simulation. Similar techniques are explored in [2] and [3]. However, these techniques rely heavily on the use of Poisson input processes, and are only applicable to limited classes of models. They cannot be generally applied to arbitrary discrete event simulations.

Other techniques improve performance by sharing computation that is common to multiple simulation runs. Splitting [4], a technique used in rare-event simulation, splits the simulation at a point just before the rare-event occurs and creates copies of the simulation to increase the number of "hits" on the rare event. In cloning [5] decision points are placed in the simulation where the simulation can take one of many branches of interest. At the decision point the simulation is cloned once for each branch thus sharing all the computation before the cloning point.

Incremental simulation techniques described in [6] and [7] have similarities to the approach described here. Incremental simulation has been proposed to aid in the design of VLSI circuits where new simulations are needed because minor changes are made to an existing circuit. When changes are made, incremental simulation only *simulates* the portion of the circuit that has been modified, reusing the rest of the previously completed simulation.

The approach described here differs from the previous approaches in a couple of key aspects. The approach, though initially targeted at communication networks, is broad enough to be applied to other types of simulations and simulation domains. There is no assumption made of the simulation domain. The technique also lends itself to parallelization. As described in later sections a sequential and parallel implementation of this technique is presented.

3. Updateable Simulation Framework

In a conventional replicated experiment, each run is completed, independent of the other runs. In particular, each run does not attempt to reuse intermediate results computed during the other runs. The basic idea in an *updateable simulation* is to update a previously completed simulation run to take into account model variations rather than re-execute the simulation "from scratch." Our objective is to create algorithms whereby the updated execution produces exactly the same results as a complete re-execution. We target discrete event simulations in this work.

A naive approach to realizing an updateable simulation is to simply apply Time Warp [8]. The original execution can be logged, and new messages introduced to effect changes to the model, e.g., changing certain model parameters. These messages trigger Time Warp's rollback mechanism, which can then update the computation to take into account the introduced model variations. Optimizations such as lazy cancellation [9] and lazy re-evaluation [10] can be applied.

In principle, such an approach will correctly update the simulation, however there are many instances where this approach will result in an excessive amount of re-computation. For example, consider a communication network simulation where the model changes involve modifying the size of message buffers in each switch. Simply applying Time Warp will cause the entire simulation to be rolled back to the beginning. Specifically, all state vectors in the original execution corresponding to logical processes modeling the switches will be incorrect in the modified run because they contain incorrect message buffer size information. This suggests Time Warp will have to completely re-execute the simulation of the switches to create the correct state vectors.

The above discussion suggests that Time Warp's "black box" view of event computations will not be sufficient for many applications. Rather, some knowledge of the event computation itself will have to be utilized. Automated analysis of event computations is an open question that will not be addressed here. Rather, we define a framework in order to illustrate some of principles that come into play in creating updateable simulations.

An updateable simulation relies on an initial *primary execution* of the simulation to create a base line from which results from other runs will be derived. The conjecture is that two simulations with only a small change in their initial state or input parameters will have similar computation histories. By utilizing the record of the previous execution a computational savings can be gained in three areas. The first is the cost of maintaining a time-stamp ordered list of pending events. The events from the primary execution are already in timestamp order and do not need to be reordered. Second, since events are being reused the cost of creating and scheduling new events can be saved. Third, if one can determine the next k events that will be executed before processing those events, one can instead process a *composed event* that produces the same results as the k events, but much more quickly. For example, if one could determine the next k events simply increment a variable, those events could be replaced by one event that increments the variable by k. Realizing this capability is not trivial in general, as will be discussed below.

Here, a packet level ATM multiplexer simulation shall be used as a concrete example to illustrate key concepts. The ATM multiplexer has two inputs, each connected to bursty ON/OFF sources and a buffer of finite capacity. The multiplexer model includes arrival and departure events and keeps statistics on the number of arrivals, departures, losses, and average delayed encountered by each packet.

The following notation will be used to describe the updateable simulation framework. The execution of any discrete event simulation can be described as a set of events and a set of states where the events define transitions from one state to another. If the simulation is at state S_i then event e_{i+1} defines a transition from state S_i to S_{i+1}. This will be

denoted by $S_i \xrightarrow{e_{i+1}} S_{i+1}$. The execution of event e_{i+1} may also schedule other events that are not explicitly shown in the notation. The execution of the simulation can be described as a series of state transitions:

$$S_0 \xrightarrow{e_1} S_1 \xrightarrow{e_2} S_2 \xrightarrow{e_3} \Lambda \xrightarrow{e_n} S_n$$

Where S_0 is the initial state and S_n is the final state of the simulation.

The actions of the execution of an event can be placed into two categories, the modification of state and the scheduling of new events. Define $\oplus$ as the execution of an event that modifies state and schedules new events, $S_i \oplus e_{i+1} = (S_{i+1}, E_{new})$ where E_{new} is the set of events scheduled by e_{i+1}. Define $\otimes$ as the state modification portion of the event computation, i.e., $S_i \otimes e_{i+1} = S_{i+1}$. The composition of j events is defined as $S_i \otimes e_{i+1} \otimes e_{i+2} \otimes ... \otimes e_{i+j-1} \otimes e_{i+j} = C^{\otimes}(S_i, e_{i+1}, ... e_{i+j})$. Similarly a composition can be defined using $\oplus$, $S_i \oplus e_{i+1} \oplus e_{i+2} \oplus ... \oplus e_{i+j-1} \oplus e_{i+j} = C^{\oplus}(S_i, e_{i+1}, ... e_{i+j})$.

An *Updateable Simulation* consists of two phases. The first or primary phase is an execution of some base-line simulation. This produces a set of states and a set of events that will be used to derive new simulation executions. The second or *update phase* starts with applying an *Update* transformation to S_0 to produce S'_0, the initial state of the new simulation. For the remainder of this paper, events and state associated with the update phase will have an apostrophe.

$$S_0 \xrightarrow{e_1} S_1 \xrightarrow{e_2} S_2 \xrightarrow{e_3} \Lambda \xrightarrow{e_n} S_n$$

$$\Downarrow U(S_0)$$

$$S'_0 \xrightarrow{e'_1} S'_1 \xrightarrow{e'_2} S'_2 \xrightarrow{e'_3} \Lambda \xrightarrow{e'_m} S'_m$$

If an event occurs in both the primary and update phase, and that event schedules the same new event(s) in the update phase that were scheduled in the primary phase, the event is said to be *re-useable*. New events need not be re-created and re-scheduled for re-useable events. Certain conditions based on the current state and possibly *other* information will be discussed that can be used to identify re-useable events.

Updateable Simulation Algorithm

Figure 1 shows the updateable simulation algorithm. This algorithm assumes the primary phase has been completed, and E, the ordered set of events processed in the primary phase, has been preserved. The initial state for the Update phase S'_0 is first created. In the multiplexer example, only the `BufferCapacity` variable differs from the initial state used in the primary phase. The initial state for the multiplexer consists of all zeros except for the buffer capacity.

The set E_w is a time stamp ordered working event list and is initialized with the set of events from the primary phase. As the execution progresses E_w diverges from the primary phase as new events not used in the primary phase are created and events in the primary phase not re-used in the update phase are canceled. The function *Reuse* returns the number of events r at the head of E_w that can be reused. This means these r events will schedule the same events in the update phase that they scheduled in the primary phase. A more detailed description of *Reuse* shall be given shortly. These r events can be reused in one of two ways. The events could be individually applied using the $\otimes$ operator to perform the necessary state transition. Alternatively, a composite of the r events could be defined and applied to the state. The composite of the r events, under the $\otimes$ operator, performs the same state transitions as applying the individual events, but does so more efficiently in one event computation. We defer discussion of the composite event computation until later.

If e cannot be reused (as determined by the *Reuse* procedure), it must be re-executed. If e was an event processed in the primary phase, events scheduled by e during the primary phase are canceled, and e is re-executed. This approach is similar to aggressive cancellation in Time Warp. It is easy to see a lazy cancellation approach could easily be utilized. This algorithm continues through E_w until all of the events have been processed.

```
Given:
  E = {e1, e2,...,en} (in time stamp order)
  S0 = initial state vector in primary phase
Set:
  Ew = E, /* Ew in time-stamp order. */
  i = 0
  S'0 = U(S0)
While (Ew not empty)
  (r,C) = Reuse(S'i,Ew)
  If (r > 0) then
    /* C = composition of next r events */
    S'i+r = S'i ⊗ C
  else
    e = Next event in Ew
    If (e not canceled) then
      Mark events scheduled by e canceled
      S'i = S'i-1 ⊕ e
      execute e, place all Enew in Ew
      (Enew are new events created by e)
    else
      Mark events scheduled by e canceled
    endif
  endif
  remove events from Ew that were processed
endwhile
```

Figure 1 Updateable simulation algorithm using aggressive cancellation.

The algorithm bears some similarity to Time Warp. The key innovations are the *Reuse* function and event composition. This *Reuse* function identifies when an event from the primary phase can be reused. This function is based on a predicate that is tested against the events in E_w, the current simulation state of the update phase, and information stored during the primary phase. The predicate $R^j(S'_i, E_w)$ evaluates to true if the following three conditions hold:

1. The first j events on E_w are in E
2. $E'_{new} = E_{new}$
3. None of the first j events has been canceled

The first condition requires the events must have originated from the primary execution, rather than be new events generated during the update phase. Obviously, in order to reuse an event's computation (specifically its event

scheduling computations), it must have been previously executed. The second condition requires that if the events were re-executed during the update phase, they will create and schedule the same events (same timestamp, event type, etc.) that they did in the primary phase. This is not guaranteed because the state of the simulation prior to processing the events in the update phase may be different from what it was in the primary phase. Determination of this condition requires analysis of the event computation and the current state of the simulation during the update phase, as will be discussed momentarily. Finally, the j events must not have been canceled. Optimizations to accommodate cancelled events are possible, but beyond the scope of the current discussion. With this predicate we can define the *Reuse* function as "return the maximum j such that R^j is true".

This algorithm does not attempt to update previously scheduled events, but rather, cancels and re-creates them. One optimization to our algorithm would relax the second condition above, and provide a mechanism to update events scheduled in the primary phase for use in the update phase. This is beyond the scope of this paper, however.

ATM Multiplexer R^j Predicates

Two R^j predicates for the ATM multiplexer example are presented next to illustrate the *Reuse* function. The first shown in Figure 2 is defined for j equal to one so it is only able to tell if the very next event on E_w can be reused. This predicate determines if the current state of the simulation in the update phase is sufficiently similar to the corresponding state during the primary phase to allow the event to be reused. For example, an arrival must have the same buffer occupancy as it did in the primary phase otherwise the departure timestamp will be different (violating condition 2 above). If the buffer capacity has decreased then arrival events that scheduled departure events may now find a full buffer and must now be dropped. In this case the event must be re-executed. Similarly if the buffer capacity has increased then an arrival event that was lost in the primary phase may now be able to schedule a departure event, so must be executed. On the other hand, departure events can always be reused unless they have been canceled.

The most notable drawback of R^1 is that it can only determine if one event can be reused. The next R^j predicate operates on a set of j events, where j is specified before executing the primary phase. To support more powerful predicates some additional processing must be done during the primary phase. This is required to derive some information concerning the future of the execution for each event in the primary phase. For instance, the number of packets lost over the next j events in the ATM multiplexer example. If this information is available during the update phase then more than one event can be tested using the R^j predicate efficiently. In general, it is important that the additional processing that is performed during the primary phase be significantly smaller than the corresponding efficiency gain in the update phase.

During the primary phase we compute for each event the number of packets lost over the next k events. Using this information a simple R^j predicate can be defined for j equal to k (see figure 3). Let e_j be the event at the head of E_w and S_{j-1} be the state in the primary phase before e_j is executed. The next k events can be reused if:

1. none of the next k events has been canceled,
2. none of the next k events is new,
3. the buffer occupancy in S'_i is the same as that in S_{j-1},
4. the new buffer capacity is greater than or equal to the buffer capacity in the primary phase,
5. there are no losses over the next k events.

Conditions 1 and 2 are easy to verify by examining the next k events in E_w. Comparing buffer occupancy and buffer capacity in state S'_i and S_{j-1} verifies conditions 3 and 4. Finally, condition 5 is tested using the information stored during the primary phase. As long as there are no losses over the next k events and the buffer occupancy has not changed then the next k events have not been affected by the change in initial state.

```
R1(S'i,Ew)
  BC' = Buffer Capacity of S'i
  BO' = Buffer Occupancy of S'i
  e = top of Ew, Sj is the state in the primary phase before e
      is executed (if e∈E)
  /* state logged from primary phase */
  BC = Buffer Capacity of Sj
  BO = Buffer Occupancy of Sj
  if (e ∉ E or e is canceled) then
    return(0)
  else if (e is an arrival) then
    if (BO'==BC' and BC'<BC) then
      /*arrival now must be dropped */
      return(0)
    else if (BO'==BC' and BC'>BC) then
      /* arrival must now be queued */
      return(0)
    else
      return(1)
    endif
  else if (e is a departure) then
    return(1)
  endif
```

Figure 2 R^1 function for ATM multiplexer. Only tests one event at a time.

```
Rk(S'i, Ew)
  None of the first k events of Ew is new
  None of the first k events has been canceled
  L = number of losses over next k events
  If (BC' >= BC and BO' == BO and L == 0) then
     reuse
  else
     do not reuse
```

Figure 3 R^j function for ATM multiplexer uses number of lost packets per k events to test k events for reuse.

Composing Event Computations

The discussion thus far has focused on the problem of determining when a set of events could be guaranteed to schedule the same events during the update phase as they did

in the primary phase. Once this has been determined, the state of the simulation in the update phase must be transformed to a new state to reflect processing the events that could be reused. Event composition is used to allow this state transformation to be efficiently performed.

Obviously, one could simply execute each of the *k* events with event creation and scheduling turned off. Event composition improves upon this by applying the state transition of *k* consecutive events to the current state as one new event computation. In the case of the ATM multiplexer one could derive the state transition for groups of *k* events. Figure 4 shows the state transitions for four consecutive events, and four combinations of event types. For instance, to apply two arrivals then two departures apply the changes specified in the AADD row to the current state. The buffer occupancy will increase by *min(2,BR)*, where *BR* is the buffer capacity remaining (buffer capacity – buffer occupancy). If the remaining buffer capacity is greater than two, then the two arrival events will queue packets on the buffer. If the remaining buffer capacity is one then only one packet will be queued, and if the remaining buffer capacity is zero then both packets are lost. The number of arrivals and departures is incremented by two. The number of lost packets is determined similar to determining the buffer occupancy. Finally, the time in buffer can be calculated based on the number of packets that are lost. This can be done for each valid combination of four events. Note any combination where there are more than two arrivals back-to-back is invalid. This is a two-input multiplexer and during each time unit can accept only two arrivals.

	BO	A	D	Losses	TIB
AAAA	Invalid				
AADD	min (2,BR)	+2	+2	max (0,2-BR)	L=0: 2BO+1 L=1: BO L=2: 0
DDDA	-2	+1	+3	0	BO-2
DDDD	0	0	+4	0	0

Figure 4: Change in state caused by composite of four events. BR = Buffer Capacity Remaining, BC = Buffer Capacity, BO = Buffer Occupancy before the event s are executed, A = Arrival Event, D = Departure Event

The information in Figure 4 can be created in at least two ways. For a given *k* the table can be created *a priori* and then used during the update phase as needed. If a group of *k* events can be used, then obtain the state transition for that sequence of events from the table. General methods for performing the composition are an interesting area of future research.

4. Case study: ATM Multiplexer

The ATM multiplexer simulation described in section 3 was implemented to demonstrate the techniques outlined above. The implementation uses the R^1 predicate defined in Figure 2 to determine event reuse. Event composition is not being used in these experiments. The updateable simulation technique was applied to a sequential and a parallel version of the ATM multiplexer. The parallel version is implemented as a time warp optimistic simulator [8] on a shared memory multi-processor. The application of the updateable simulation technique is identical in both implementations.

4.1. Performance

A 31 multiplexer model is used to evaluate the performance and effectiveness of the updateable simulation technique. The model has 32 input sources that feed into a bank of 16 two-input multiplexers (upstream multiplexers). These 16 multiplexers feed into a second bank of 8 multiplexers and so on until all traffic is fed into a single multiplexer (downstream multiplexer). The buffer sizes of the individual multiplexers can be modified, and the input sources can be set to produce varying traffic loads. For simplicity all traffic sources are set to the same link utilization. For example, a source with link utilization of 10% will generate about 10 packets per 100-time units.

Two experiment types are used to gather performance data. The first sets of experiments vary the capacity of the input buffer; the second adds additional sources.

Execution time speedup is used to evaluate the performance of the updateable ATM simulation relative to a non-updateable simulation. To calculate this metric the time taken to run all the update phases of the updateable simulation is summed. Then the time taken to run the corresponding non-updateable simulations is summed. Then speedup is calculated by dividing the non-updateable execution time by the updateable execution time. This particular metric does not include the overheads incurred during the primary phase. In all of the ATM multiplexer experiments below the primary phase ran about 17% slower than the corresponding non-updateable simulation. The overhead is due to saving the state and events. *Event Reuse* measures the percentage of events that are reused from the primary phase.

The goal of these experiments is to obtain the same results from the updateable simulation as would have been obtained by running the non-updateable simulation. For each update phase the statistical output produced was verified against the output from a non-updateable simulation run.

Buffers Experiment

In the Buffers experiment the capacity of the input buffers is decreased before each update phase. There are four variations of this experiment 1) all of the multiplexer's buffer capacities are changed, 2) half changed, 3) one upstream multiplexer is changed, and 4) the downstream multiplexer is changed. Buffers capacities are decreased between update sub-phases. For each experiment the background traffic intensity is varied.

Figures 5 and 6 show the speedup and reuse ratio for the parallel simulations. Note the speedup of the updateable parallel simulation is relative to a non-updateable parallel simulation. The background traffic level is 4the existing traffic (x-axis of Figures 5 and 6) from the primary phase. For example, a background traffic level of 15% means that all the sources are set to a link utilization of 15%. Speedup varies considerably depending on which multiplexer is updated and the level of traffic. If the downstream multiplexer is updated

over six-fold speedup is obtained for all traffic levels. Since only the "last" multiplexer is updated most of the events in the simulation are unaffected resulting in a nearly 100% reuse of events. However, if an upstream multiplexer is updated then speedup ranges from eleven for light traffic levels to one (no speedup) for heavy traffic levels. An update at an upstream multiplexer will affect more events resulting in a lower reuse rate particularly at the heavier traffic levels.

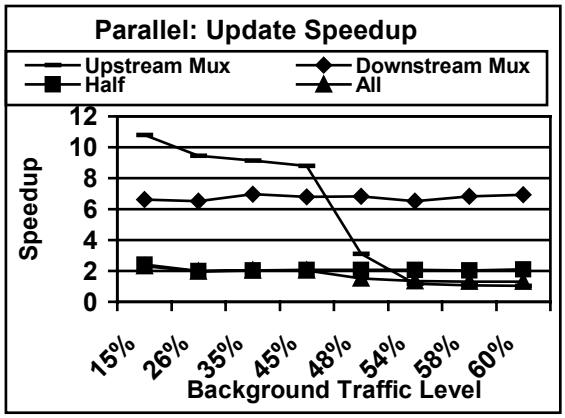

Figure 5: Speedup of update phase versus non-updateable ATM simulation.

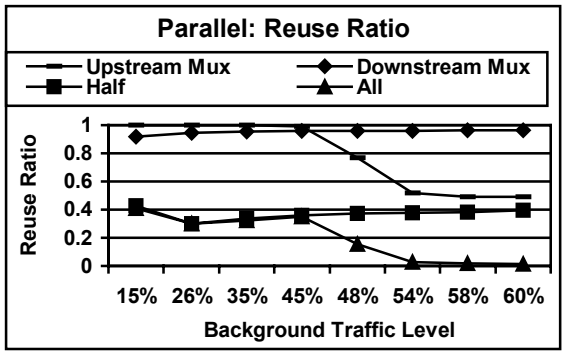

Figure 6: Percentage of events reused during the update phase.

When the capacities of all multiplexer buffers are changed the speed up is slightly less than two for the lightest traffic level. As the traffic level is increased there is little to no speedup, and for the highest three traffic levels there is a slight slow down. As more and more packets arrive the input buffers become saturated causing packets to be dropped. This effect spreads to multiplexers down stream and causes most events not to be reused, so the reuse rate declines to almost zero. The slow down for the three highest source levels are not unexpected. Nearly all of the events must be executed and there is an added cost of applying the reuse predicate. One possible solution to avoid this performance degradation is to detect when the state of the simulation as a whole has diverged sufficiently from the primary run then switch off all updateable simulation support.

Despite the fact that the majority of events were not reused when *half* or *all* of the multiplexer's buffers were changed, further optimization may still be possible. During the update phase it is common for event sequences to remain the same as in the primary phase, except the events are shifted in time. For instance, adding an extra arrival may cause a sequence of departure events to be delayed one time unit. This optimization is explored briefly in the section 6.

New Sources Experiment

The New Sources experiment uses the same model as the Buffers experiment but now the buffer sizes are held constant between runs and new sources are turned on in each update phase. One, two, three, or four new sources are activated at upstream multiplexers for each of the background traffic levels.

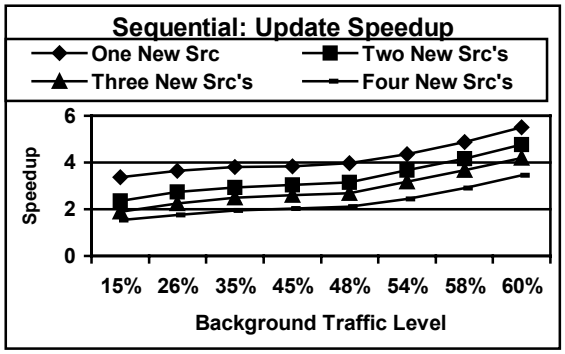

Figure 7: Speedup of update phase versus non-updateable ATM simulation.

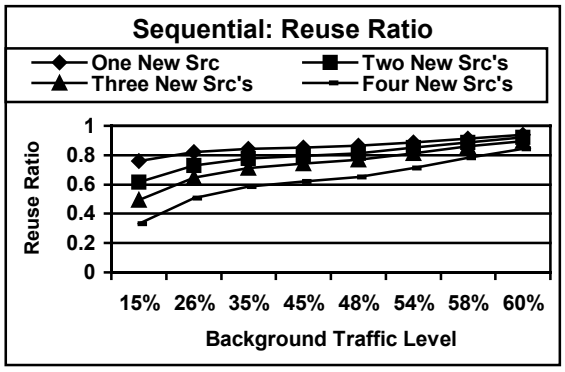

Figure 8: Percentage of events reused during the update phase.

Figures 7 and 8 show the speedup and reuse ratio for the sequential implementation of the simulation. The x-axis is the same as in Figures 5 and 6. As the background traffic level increases the reuse rate and speedup increase. This seems counter-intuitive but at higher background traffic levels there is a higher probability of packets being lost. So in fact the packets from the new sources are being dropped earlier limiting the changes caused by the new sources.

5. Case Study: ATM Switch

The ATM switch simulation uses the same event types as the ATM multiplexer but with additional code to forward packets along the designated circuit. The update algorithm used in the ATM multiplexer does not reuse events where the only difference is the time stamp. The update algorithm used in the ATM switch simulation will detect when an event's time stamp must be changed and make the necessary change to allow the event to be re-used. Care is taken to ensure that all events are processed in time stamp order. The algorithm used here is based on the algorithm in Figure 2 except now the Reuse function now returns a set of events with altered time stamps.

For the event reuse analysis events are placed into five categories. *Identical* events are events that have the same time stamp and perform the same state transition. *Delta TS* events are events where the time stamp is changed but the event itself does not change. *Skipped* events are *identical* events but application of the reuse function to these events can be avoided. If an update does not affect an object then all of the events will be the same. The object can be "fast-forwarded" to its final state. If the update does not affect the object until simulation time *t* then all of the events with time stamp *t* will be unaffected. By saving an objects state before each event is executed it is possible implement a fast forward mechanism that eliminates having to apply the Reuse function to these events. The final two categories are *new* events and *canceled* events.

5.1. Performance

A network of fifty-one switches is used to evaluate performance. The switches are divided into ten subnetworks containing five switches each and one central switch connecting the ten subnetworks together. Switches within each subnetwork are fully connected.

An experiment similar to the new sources experiment for the ATM multiplexer is used to evaluate performance. The experiment has five local circuits in each subnetwork each with a link utilization of 20%. In each update phase a new circuit is added with a link utilization of 20% that connects two subnetworks. Essentially global flows are added to the simulation in each update phase. In update phase 1 one new circuit is added, in update phase 2 two new circuits are added, and so on.

With only one new circuit the speedup is nearly 7 times, see Figure 9. A single new circuit does not perturb the simulation greatly allowing many events to be skipped and the majority of the rest of the events can be reused, see Figure 10. As the number of new circuits is increased there is a greater perturbation to the simulation causing fewer events to be *skipped* but more events are *identical*. The events created by the new circuits increase the probability that a simulation object will be affected by the update earlier in simulation time. The earlier in simulation time an object is affected by an update the fewer events that are skipped. However, in these experiments a large portion of the events is still *identical.* This explains why the percentage of *skipped* events drops and *identical* events rises between one and five new circuits. After that point the new circuits cause more packets to be delayed resulting in fewer *identical* events and more *delta TS* events.

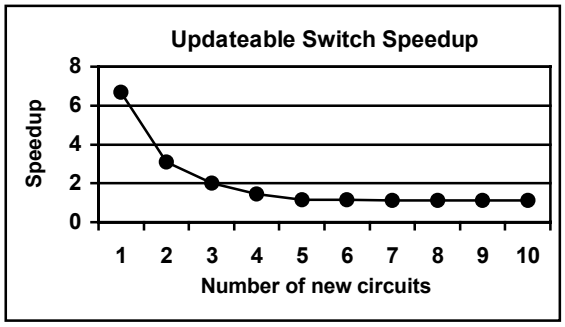

Figure 9: Speedup of update phase versus non-updateable ATM switch simulation

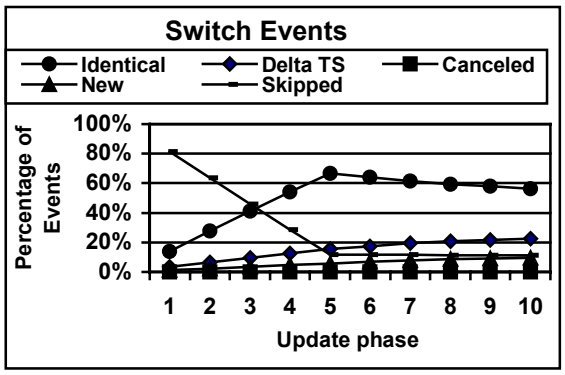

Figure 10: Categorization of events in switch simulation.

6. Future Work

The simulations described in this paper are used to illustrate techniques to realize updateable simulation. There are many areas requiring further research. First, managing the information that is saved during the primary phase must be done efficiently to realize good performance. For long running simulations the amount of information that will need to be stored can be vast. Efficient techniques to compress, store (to secondary storage), and load (from secondary storage) this data are needed.

Another enhancement to the technique includes developing R^j for j larger than one. Unfortunately time did not permit implementation of the reuse-predicates described in section 3. We believe a more powerful reuse predicate and use of event composition will improve performance.

The correctness of the simulation results will also have to be addressed. For now the results of the updateable simulation are compared against the results of a traditional simulation to verify correct execution. The correctness of the updateable

simulation will rest heavily on the correctness of the reuse predicate. In addition to automatically generating reuse predicates for a given simulation there must be a methodology to verify correctness. However, relaxing correctness requirements may lead to a class of reuse predicates the will produce approximate results bounded by some error. It may be reasonable to sacrifice some accuracy for a significant gain in performance.

To provide more robust, meaningful results we are examining implementation of this technique in a large existing simulation package such *ns2* [11]. An analysis of a simple *ns* simulation showed that this technique shows promise even with more complex TCP/IP simulations. A base line simulation with four TCP flows was run logging all of the events. Then 10 additional runs were made adding a new TCP flow for each run. The analysis showed that a significant number of events were delayed (time stamp is different) but not altered, see Figure 11. This suggests that an updateable simulation capable of dealing with time stamp changes events could perform well.

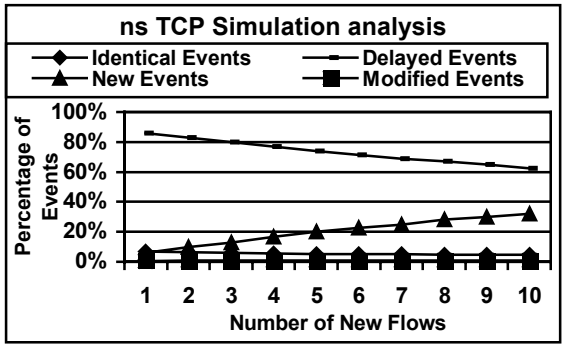

Figure 11: Categorization of events in an *ns* simulation

7. Conclusions

A general framework for realizing updateable simulation was presented. The premise of the framework is that multiple simulation runs may share a considerable amount of computation and reusing this computation will provide a speedup. An important task is to identify when an event or events can be reused. This task is performed by a *Reuse* procedure that uses information stored during the primary phase to quickly determine when an event or events and be reused. Efficiency of the update then becomes tightly bound to how efficiently the *Reuse* procedure can be implemented. Unfortunately due to space constraints an analysis of the reuse procedure used in this paper could not be presented here.

The updateable simulation techniques were applied to a sequential and parallel packet level ATM multiplexer simulation. In both implementations execution time speedup was achieved despite using a *Reuse* procedure that only evaluated one event at a time. An ATM switch was also implemented using an update algorithm that could accommodate changing the time stamp of an event.

Key areas of further research were identified ranging from event composition, state compression, and correctness of updateable simulations. Finally, additional work is required to consider implementation of an updateable simulation using existing simulators such as *ns2*.

8. Acknowledgements

This research was funded by Defense Advanced Research Projects Agency, contract N66001-00-1-8934, and National Science Foundation Grant ANI-9977544.

9. References

1. Vakili, P., *Massively Parallel and Distributed Simulation of a Class of Discrete Event Systems: A Different Perspective.* ACM Transactions on Modeling and Computer Simulation, 1992. **2**(3): p. 214-238.
2. Cassandras, C.G., J.-I. Lee, and Y.-C. Ho, *Efficient Parametric Analysis of Performance Measures for Communication Networks.* IEEE Journal on Selected Areas in Communication, 1990. **8**(9): p. 1709-1722.
3. Glynn, P.W. and P. Heidelberger, *Analysis of Parallel Replicated Simulations Under a Completion Time Constraint.* ACM Transactions on Modeling and Computer Simulation, 1991. **1**(1): p. 3-23.
4. Glasserman, P., P. Heidelberger, and P. Shahabuddin. *Splitting for Rare Event Simulation: Analysis of Simple Cases*. in *Winter Simulation Conference*. 1996. Coronado, California, USA.
5. Hybinette, M. and R.M. Fujimoto. *Cloning, a Novel Method for Interactive Parallel Simulation*. in *Winter Simulation Conference*. December 1997.
6. Hwang, S.Y., T. Blank, and K. Choi, *Fast Functional Simulation: An Incremental Approach.* IEEE Transactions on Computer-Aided Design of Integrated Circuit Systems, 1988. **7**(7): p. 765-774.
7. Choi, K., S.Y. Hwang, and T. Blank. *Incremental-in-Time Algorithm for Digital Simulation*. in *25th ACM/IEEE Design Automation Conference, 1988 Proceedings.* 1988.
8. Jefferson, D.R., et al., *The Time Warp Operating Systems*, in *11th Symposium on Operating Systems Principles*. 1987. p. 77-93.
9. Gafni, A., *Rollback Mechanisms for Optimistic Distributed Simulation Systems*, in *Proceedings of the SCS Multiconference on Distributed Simulation*. 1988. p. 61-67.
10. West, D., *Optimizing Time Warp: Lazy Rollback and Lazy Re-evaluation*, in *Computer Science Department*. 1988, University of Calgary: Calgary, Alberta Canada.
11. Team, T.V.P., *ns Notes and Documentation*. 1998.