

Updateable Simulations

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. The algorithm used in the framework lends itself to straightforward realization on parallel computers. This framework is applied to the problem of simulating a set of cascaded ATM multiplexers. Performance measurements of a parallel implementation of this simulation 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 common 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. Similarly, when simulation analyses must be repeated to include new information, much of the computation in the new runs may be similar or identical to that of earlier runs. When this is the case, it may be possible to “reuse” computations from prior runs rather than re-compute them. It is not unreasonable to expect that reusing computations can greatly reduce the time required to complete multiple runs, or to redo previously completed runs to incorporate new information.

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. Measurements of a parallel implementation of the algorithm are presented. Section 5 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.

A technique used in rare-event simulation is to share computations by splitting and making copies of the simulation when the state of the simulation reaches some threshold [4]. The threshold usually indicates that the likelihood of the rare-event is great and by making multiple copies of the simulation, a larger number of “hits” on the rare event can be made. Since the events of interest occur rarely, the amount of time before the first rare event can be considerable. A significant timesaving can be realized by sharing the computation prior to the split.

Another technique described in [5] introduces the idea of cloning a simulation at certain points in the execution called decision points. At a particular point in simulation time there may be several branches the simulation can take. A replicated execution would repeat the computation prior to the decision point. With cloning, this computation need only be performed once, and clones of the simulation are created at the decision point to complete the different runs. Techniques are proposed to further share computations after the decision point is reached.

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. All computations in a discrete event simulation are performed as the result of processing events. Each event contains a time stamp. The result of the computation is the same as if all events were processed sequentially, in time stamp order.

A naive approach to realizing an updateable simulation is to simply apply Time Warp [6]. 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 [7] and lazy re-evaluation [8] 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 for other runs will be derived. The conjecture is that two simulations with only a small change in their initial state 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 time spent in buffer by packets. Figure 1 shows the multiplexer’s state, arrival event, and departure event.

Multiplexer State	BufferCapacity BufferOccupancy Arrivals Departures Losses TimeInBuffer
Arrival Event() { Arrivals++; if (BufferOccupancy < BufferCapacity) { TimeInBuffer += BufferOccupancy; BufferOccupancy++; Schedule Departure Event for time (now + BufferOccupancy); } else { Losses++ } } }	
Departure Event() { Departures++; BufferOccupancy-; Schedule Arrival at next Multiplexer at time (now + 1); } }	
Figure 1 Multiplexer state, Arrival event, and Departure event.	

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:

$$S_0 \xrightarrow{e^1} S_1 \xrightarrow{e^2} S_2 \xrightarrow{e^3} \dots \xrightarrow{e^n} S_n$$

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, $S_i \overset{\oplus}{\rightarrow} e_{i+1} = (S_{i+1}, E_{new})$ 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., $S_i \overset{\otimes}{\rightarrow} e_{i+1} = S_{i+1}$. The composition of j events is defined as $S_i \overset{\otimes}{\rightarrow} e_{i+1} \overset{\oplus}{\rightarrow} e_{i+2} \overset{\otimes}{\rightarrow} \dots \overset{\oplus}{\rightarrow} e_{i+j} = C^{\otimes}(S_i, e_{i+1}, \dots, e_{i+j})$. Similarly a composition can be defined using $\overset{\oplus}{\rightarrow}$, $S_i \overset{\oplus}{\rightarrow} e_{i+1} \overset{\otimes}{\rightarrow} e_{i+2} \overset{\oplus}{\rightarrow} \dots \overset{\otimes}{\rightarrow} e_{i+j} = C^{\oplus}(S_i, e_{i+1}, \dots, e_{i+j})$.

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.

$$S_0 \xrightarrow{e^1} S_1 \xrightarrow{e^2} S_2 \xrightarrow{e^3} \dots \xrightarrow{e^n} S_n$$

$$\Downarrow U(S_0)$$

$$S'_0 \xrightarrow{e'^1} S'_1 \xrightarrow{e'^2} S'_2 \xrightarrow{e'^3} \dots \xrightarrow{e'^m} S'_m$$

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 later can be used to identify re-useable events.

Using the ATM multiplexer as an example, suppose it is desired to study the affects of buffer size on the number of packets lost due to buffer overflow. The buffer capacities under consideration are 80, 90, 100, 110, and 120 packets. An initial run can be made with a buffer capacity of 100 packets. This simulation will be used to create simulations for the four other buffer capacities. There is no restriction on which buffer capacity is chosen for the primary phase, but some choices are better than others.

Updateable Simulation Algorithm

Figure 2 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.

```

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 2 Updateable simulation algorithm using aggressive cancellation.

If r is equal to zero then the very next event, e , on E_w either must be executed or it has been canceled. If e has been canceled then any events scheduled by e must also be canceled. This is similar to sending anti-events in a Time Warp simulation to annihilate previously scheduled events. In the sequential version of the update algorithm, roll back is not needed because the events being canceled are always in the future.

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.

The algorithm bares 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 top j events on E_w are in E
- 2) $E_{new} = E_{new}$
- 3) None of the top j events have 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

Three R^j predicates for the ATM multiplexer example are presented next to illustrate the *Reuse* function. The first shown in Figure 3 is defined for j equal one so it is only able to tell if the very next event on E_w can be reused. A reuse function based on this predicate will return either zero or one. 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.

```

R'(Si;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 3 R' function for ATM Multiplexer. Only tests one event at a time.

The most notable drawback of R' is that it can only determine if one event can be reused. The next two R^j predicates operate on a set of j events, where j is specified before executing the primary phase. To support these 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 4). 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 have been canceled,
- 2) none of the next k events are 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.

```

Rk(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)
  BC = Buffer Capacity of Sj
  BO = Buffer Occupancy of Sj
  None of the top k events of Ew are new
  None of the top k events have
    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 4 R^j function for ATM Multiplexer uses number of lost packets per k events to test k events for reuse.

The above allows groups of events to be tested but only applies if the buffer capacity is increased. An improvement, shown in Figure 5, tracks the maximum buffer occupancy during the k events as well as the number of lost packets. This will allow a greater number of events to be reused.

In this version an additional test is performed when the buffer capacity is decreased. If the difference between the current buffer capacity and the maximum buffer occupancy over the next k events is greater than or equal to zero then the next k events are not affected by the smaller buffer size. If in the primary phase there are no losses and as long as the buffer capacity is not reduced to the point where packets are lost then the next k events will not cause departures to

be canceled because the packet was dropped. A Reuse function can be created using either version of R^k .

```

R'(Si, Ew)
{
  BC' = Buffer Capacity of Si
  BO' = Buffer Occupancy of Si
  e = top of Ew, Sj is the state in the
      primary phase before e is executed
      (if e ∈ E)
  BC = Buffer Capacity of Sj
  BO = Buffer Occupancy of Sj
  None of the top k events of Ew are new
  None of the top k events have
      been canceled
  L = number of losses over next k events
  MBO = maximum buffer occupancy over next
      k events

  If (BC' >= BC and
      BO' == BO and
      L == 0) then
      reuse next k events
  else if (BC' < BC and
          BO' == BO and
          BC' - MBO >= 0)
      reuse next k events
  else
      do not reuse
}

```

Figure 5 R' function for ATM Multiplexer uses number of lost packets and maximum buffer occupancy for 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 6 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. As soon as an arrival occurs then it is guaranteed that a departure will be scheduled for the next time unit thus breaking up groups of arrivals and reducing the number of valid combinations.

	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 6: 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 6 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 is an interesting area of future research.

4. A 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' predicate defined in Figure 3 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 [6] on a shared memory multi-processor. The application of the updateable simulation technique is identical in both implementations.

4.1. Performance

Experimental Setup

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 multiplexers. These 16 multiplexers feed into a second bank of 8 multiplexers and so on until all traffic is fed into a single multiplexer (see Figure 7). The buffer sizes of the individual multiplexers can be modified, and the input sources can be set to produce light or heavy traffic loads. For simplicity all traffic sources are set to the same rate and the rate can take on one of 8 different settings. Figure 8 shows approximately how much traffic is generated by a source at each of the settings. The first row shows the traffic setting, and the second row shows on average how many packets will be generated during 1000 time units.

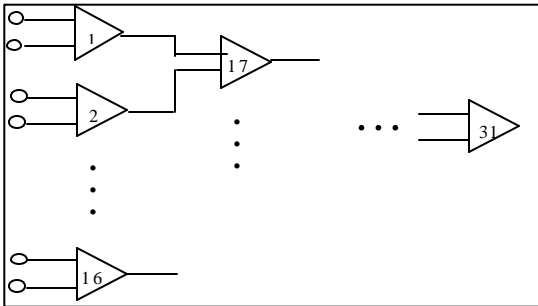


Figure 7 ATM multiplexer model used to gather performance and reuse statistics.

1	2	3	4	5	6	7	8
153	262	354	448	478	541	578	600

Figure 8: Number of packets generated by the ON/OFF sources for the 8 traffic levels.

Two different 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 with different traffic rates.

Execution time speedup is used to evaluate the performance of the updateable ATM simulation relative to a non-updateable simulation. To calculate the 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 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 (multiplexer 17-31), 3) multiplexer 1 is changed, and 4) multiplexer 31 is changed. Buffers capacities are decreased between update sub-phases. For each experiment the background traffic intensity is varied between the lowest level and the highest level. The performance graphs shows speedup and reuse rate for each background traffic level.

Figures 9 and 10 show the speedup, and reuse for the sequential simulations, and Figures 11 and 12 show the speedup and reuse ratio for the parallel simulations. Both sets of results are similar and the following discussion applies to both sets of data. If the buffer capacity of one multiplexer is changed between runs a speedup of between 6 and 9 is obtained. The amount of speedup is dependant on which multiplexer is changed. If multiplexer 31 is changed a speedup of about 6 is achieved for all background traffic levels. Here only events at the last multiplexer will be affected so most of the events from the primary phase can be reused. Since only events for the last multiplexer are affected by the change in buffer capacity the speedup and reuse rate is stable across the background traffic levels. When multiplexer 1 is changed the speedup for the lowest four background traffic intensities is approximately 9, but then drops off at the fifth background traffic level. The reuse rate follows the same trend. At the fifth traffic level multiplexer 1 has enough packets arriving to saturate the input buffer causing packets to be lost. Since the primary phase executes with the largest buffer size (remember buffer sizes are decreased between update sub-phases) the update phases become more sensitive to the background traffic level. The more packets arriving and the smaller capacity buffers cause more events to be executed than reused. This is the reason for the drop in speedup and reuse ratio.

If *half* or *all* of the multiplexer buffers are changed

then a significant drop off in speedup and reuse ratio is seen. When the capacity of *all* multiplexer buffers are changed the speed up is slightly less than two for the lightest traffic load. As the traffic rate is increased there is little to no speedup, and for the highest 3 traffic rates 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 rates are not unexpected. Nearly all of the events must be executed but 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. We believe optimizations to exploit this fact are possible, and significantly better event reuse and speedup can be obtained. This is an avenue of future research

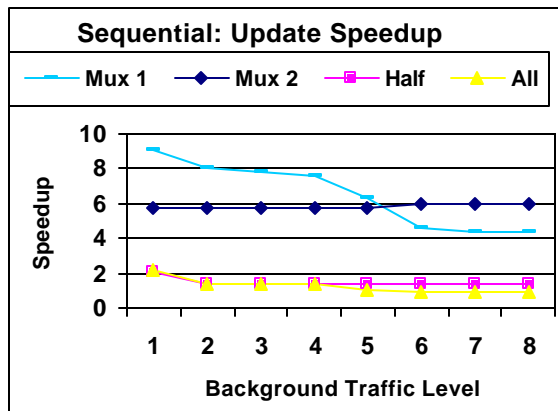


Figure 9: Speedup of update phase versus non-updateable ATM simulation.

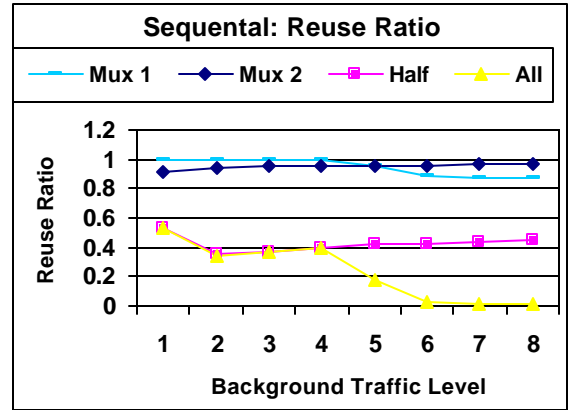


Figure 10: Percentage of events reused during the update phase.

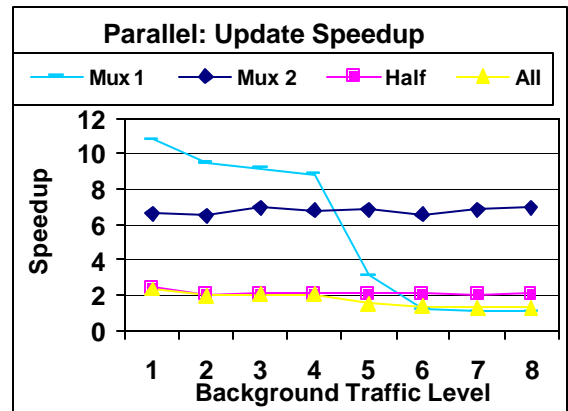


Figure 11: Speedup of update phase versus non-updateable ATM simulation.

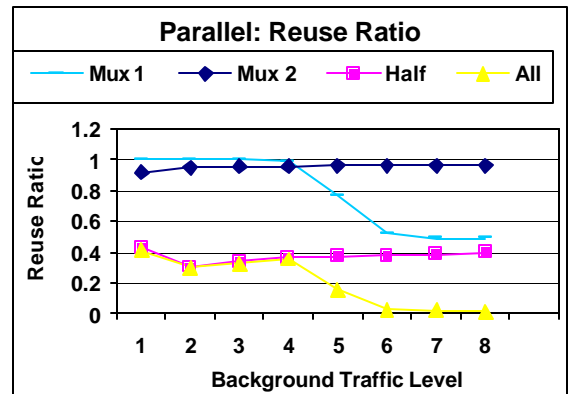


Figure 12: Percentage of events reused during the update phase.

New Sources Experiment

The New Sources experiment uses the same model as the Buffers experiment except now the buffer sizes are held constant between runs and new sources are added. One, two, three, or four new sources are

added for each of the background traffic level. The new sources are connected to multiplexer 1, multiplexer 16, multiplexer 4, and multiplexer 5 respectively for each new source. For a given background traffic level the primary phase runs without the new source. Then for each update phase the traffic level of the new sources are increased.

Figures 13 and 14 show the speedup and reuse ratio for the sequential implementation of the simulation. The x-axis in these figures represents the background traffic level not the traffic level of the new sources. 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.

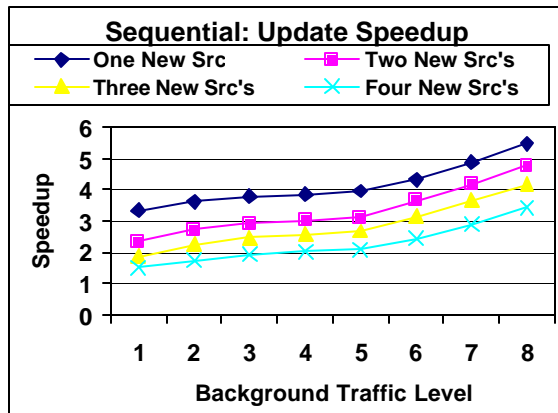


Figure 13: Speedup of update phase versus non-updateable ATM simulation.

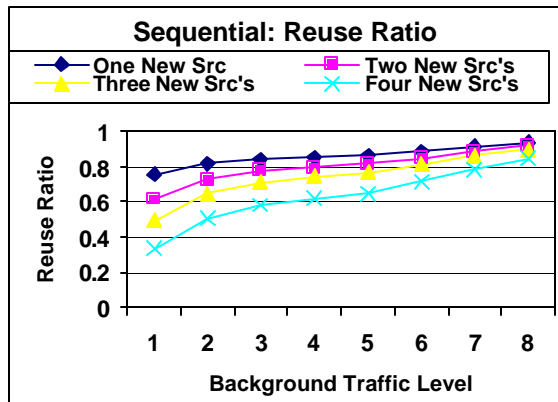


Figure 14: Percentage of events reused during the update phase.

5. Future Work

The simple 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, possibly running into the gigabytes. Efficient techniques to compress, store (to secondary storage), and load (from secondary storage) this vast amount of data need to be used.

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

As mentioned earlier, techniques are being explored to reuse events that currently are not reused because the time stamp is not correct. Especially in network simulators the introduction of new packets may simply cause the time stamp of other events to change but not change the computation of the events.

To provide more robust, meaningful results we plan to implement this in a larger existing simulation package such as *ns* [9]. 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 new flow shared links with all of the original flows, providing a challenging test case for updateable simulations. The analysis showed that a significant number of events were delayed (time stamp is different) but not altered, (see Delayed Events line in Figure 15). This suggests that an updateable simulation capable of dealing with time stamp changes events could perform well.

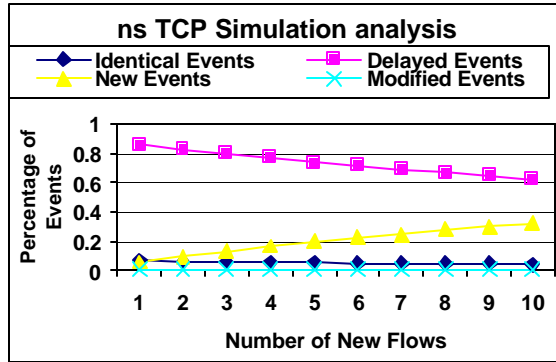


Figure 15: Percentage of events in each new run that are identical, delayed, new, or modified compared with events in the original run.

6. 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.

The update 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.

Key areas of further research were identified. Managing information stored during the primary phase will make a big impact on the performance of an updateable simulation. It is expected that long running simulation will generate gigabytes of data that will need to be efficiently saved and loaded from secondary storage. Techniques to reduce the amount of data stored must also be explored. Finally, there are plans to implement an updateable simulation using the *ns* simulator to demonstrate performance for more complex simulations, such as TCP networks.

7. 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. Jefferson, D.R., et al., *The Time Warp Operating Systems*, in *11th Symposium on Operating Systems Principles*. 1987. p. 77-93.
7. Gafni, A., *Rollback Mechanisms for Optimistic Distributed Simulation Systems*, in *Proceedings of the SCS Multiconference on Distributed Simulation*. 1988. p. 61-67.
8. West, D., *Optimizing Time Warp: Lazy Rollback and Lazy Re-evaluation*, in *Computer Science Department*. 1988, University of Calgary: Calgary, Alberta Canada.
9. Team, T.V.P., *ns Notes and Documentation*. 1998.