Parallel Simulation of Statistical Multiplexers*

Richard M. Fujimoto, C. Anthony Cooper, and Ioanis Nikolaidis

Revision of September 30, 1994

Abstract

The simulation of high–speed telecommunication systems such as ATM (Asynchronous Transfer Mode) networks has generally required excessively long run times. This paper reviews alternative approaches using parallelism to speed up simulations of discrete event systems, and telecommunication networks in particular. Subsequently, a new simulation method is introduced for the fast parallel simulation of a common network element, namely, a work–conserving finite capacity statistical multiplexer of bursty ON/OFF sources arriving on input links of equal peak rate. The primary performance measure of interest is the cell loss ratio, due to buffer overflows. The proposed method is based on two principal techniques: (1) the derivation of low–level (*cell level*) statistics from a higher level (*burst level*) simulation and (2) parallel execution of the burst level simulation program. For the latter, a *time–division* parallel simulation method is used where simulations operating at different intervals of simulated time are executed concurrently on different processors. Both techniques contribute to the overall speedup. Furthermore, these techniques support simulations that are driven by traces of actual network traffic (trace–driven simulation), in addition to standard models for source traffic. An analysis of this technique is described, indicating that it offers excellent potential for delivering good performance. Measurements of an implementation running on a 32 processor KSR–2 multiprocessor demonstrate that, for certain model parameter settings, the simulator is able to simulate up to 10 billion cell arrivals per second of wallclock time.

*Richard Fujimoto and Ioanis Nikolaidis are with the College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280. C. Anthony Cooper is with Bellcore, NVC–1H225, 331 Newman Springs Road, Red Bank, NJ 07701. This work was supported by a grant from Bellcore. Portions of this manuscript also appeared in the *11th International Conference on Analysis and Optimization of Systems*, June 1994 and the *32nd IEEE Conference on Decision and Control*, December 1993.

Parallel Simulation of Statistical Multiplexers

Richard M. Fujimoto, C. Anthony Cooper, and Ioanis Nikolaidis

1 Introduction

In recent years, an effort for standardization of high–speed telecommunication networks has been underway. The need for new standards and technologies is especially pronounced in light of a wide variety of applications ranging from low speed data transfers to high quality high–definition television (HDTV) distribution. The diversity of traffic sources has a direct impact on the design of high–speed networks as well as on their expected performance. Traditional traffic assumptions, e.g., Poisson arrival streams, are not applicable at the ATM cell level. Thus, the analytical approach to performance modeling is generally numerically intensive and often approximate. Further, the approximate analytical models require validation that is performed traditionally by simulations. In any case, simulation is an indispensable tool when it comes to testing the performance of a system over a wide variety of traffic loads and scenarios.

However, simulations of high–speed networks require relatively long execution times. This is especially true if we are interested in the collection of statistics over several minutes of real time operation. Collecting statistics over such long intervals is necessary in asynchronous transfer mode (ATM) networks because the dynamics of the system are not captured in short intervals. Connections may last several minutes, or perhaps hours. Hence, if we wish to collect meaningful statistics for the time scale of typical connections, time intervals of similar time scale (minutes to hours) must be simulated. Moreover, the conveyed data unit (the *cell*[1]) is very small, so several thousand such cells can be conveyed on a single high–speed link in one second. For example, in one second of real time operation of a 155 Mbps ATM link, nearly three hundred thousand cells may arrive, depending on how heavily the link is loaded. Since each arrival corresponds to an event in a traditional cell level simulation, at least as many events will have to be processed.

The stringent Quality of Service (QoS) objectives associated with ATM networks also result in longer simulation runs. QoS objectives for cell loss on an ATM connection are frequently described in terms of a cell loss ratio or cell loss probability. The cell loss probability for high-speed Broadband–ISDN can be of the order of 10^{-9}. In other words, on the average, one in every 10^9 cells is lost, mainly due to buffer overflows. In order to calculate the cell losses with a sufficiently good confidence interval, we need to simulate the system for a large number of events (cell arrivals), many times 10^9. The cell losses are so–called *rare events*, and the interval between cell losses is very long. Suppose we consider the same 155 Mbps ATM link we mentioned before under heavy load. Then, one loss in 10^9 arrivals is actually an event every 45 minutes of real time. Actually, as many authors have observed, losses are clustered in bursts, but even under this assumption we must be able to simulate more than 45 minutes of system operation in order to encounter several occurrences of interesting events, i.e., cell losses. Such simulations may take days to complete.

The general network of interest consists of two types of elements (see Figure 1), *multiplexers* and *switches/cross–connects*. A multiplexer collects traffic from a number of incom-

[1]A cell is a 53 byte fixed size packet.

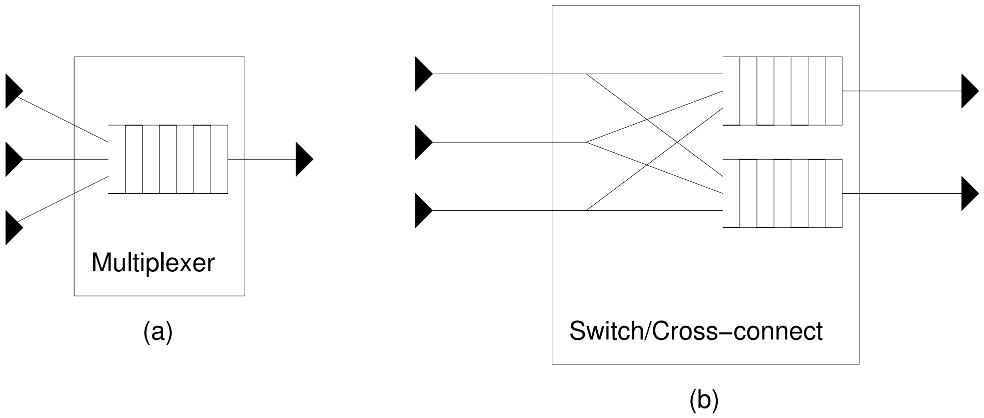

Figure 1: Typical network elements, (a) multiplexer and (b) switch/cross–connect.

ing (low speed) links and transmits the aggregate traffic on an outgoing (high speed) link. Switches/cross-connects are used primarily for routing the traffic by diverting traffic that enters from the input links to one (or more) of the output links. Within a switch/cross–connect, it is also possible to have a form of (de–)multiplexing (e.g., an incoming link's traffic is split into two separate outgoing links, or the traffic from two incoming links merged into one outgoing link). Hence, the simulation of multiplexers is a vital and central issue in high–speed network design. Both multiplexers and cross–connects have finite buffers, so we expect cell losses to occur when these network elements are overloaded.

Here, we will focus attention on the simulation of multiplexers under bursty arrivals from several input streams. The most common and widely accepted model for a bursty traffic source is the so–called *ON/OFF model*. This model originates from the early days of analysis related to the silent and active intervals for voice activity [Gru82]. The source can be in one of two states, "ON" or "OFF." When in the ON state, it generates cell arrivals. We will assume that in the "ON" state, one cell is generated per time slot, as shown in Figure 2(a). Figure 2(b) and (c) illustrate two other possible ON/OFF sources, differing only on the arrival structure during the "ON" state. In all cases, cells arrive in *time slots*.

The ATM cells are the product of the segmentation of larger packets. In its simplest form, the segmentation of a packet generates a sequence of cells. This sequence of cells can be represented by an ON period. Since currently there exists no information as to how the segmentation might affect the shape of the generated cell burst, we assume back–to–back arrivals as depicted in Figure 2(a). This selection also represents the worst case traffic for the multiplexer with respect to congestion and cell loss behavior compared to the other alternatives where arrivals do not occur back–to–back.

In particular, for a given ON period of a source, the maximum traffic load that can arrive at the multiplexer during this period is when arrivals occur back–to–back. The cell loss ratio is dependent on the utilization of the input links, but this utilization is dependent on the load during the ON periods (for the same OFF periods) of the sources. Hence, the worst cell loss ratio is anticipated for the maximum utilization corresponding to the maximum arrivals during the ON period. Therefore the worst case corresponds to back–to–back arrivals during the ON period.

We formulate the simulation problem as follows: consider the simulation of a finite buffer statistical multiplexer receiving constant size packet (cell) arrivals from a number of bursty

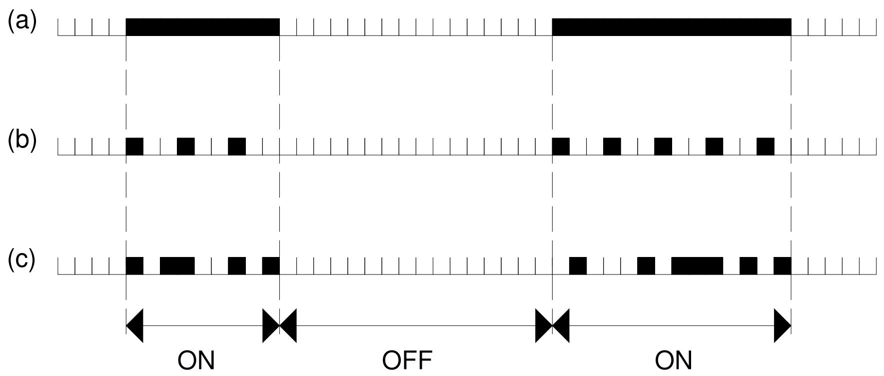

Figure 2: ON/OFF source, (a) with one arrival per slot in the ON state, (b) one arrival every 3 slots in the ON state and (c) random (geometric cell interarrival time) arrivals in the ON state.

sources. A cell arriving at a full queue is *lost* (i.e., dropped). The bursty sources are described as ON/OFF sources. The service time received at the multiplexer is deterministic and the service discipline is first–come–first–serve (FCFS). The measures of interest are the average utilization and the cell loss probability. Our objective is to dramatically reduce the simulation time for such multiplexers. Ultimately, we will consider the simulation of feed–forward networks of multiplexers and switches, however, this paper will only be concerned with the simulation of a single multiplexer.

In the next section we review techniques for using parallel computers to speed up the execution of discrete event simulations. Our approach is based on the exploitation of the specific arrival processes, as presented in section 3, where we also present results concerning the speedup that may be expected using this technique. In section 4 we develop the parallel simulation algorithm and present its performance. Finally, we conclude (section 5) by speculating on the possible use and extensions of the new method.

2 Review of Parallel Discrete Event Simulation Techniques

One may view a simulation as a computation that must compute the values of certain *state variables* across simulated time. The state variables capture the state of the system, e.g., the number of cells waiting for service in a simulation of a multiplexer. Changes in the state of the system occur at discrete points in simulated time, e.g., when a new cell is generated by a source. This so-called space-time view of the simulation [BLC91] is depicted in Figure 3 using a two-dimensional graph, where the vertical axis represents the state variables, and the horizontal axis represents simulated time. The goal of the simulation program is to "fill in" the graph by computing the values of each of the state variables across simulated time. A parallel simulator attempts to use multiple processors that simultaneously "fill in" different portions of the space-time graph.

In Figure 3(a) the graph is divided into horizontal "strips," with a *logical process* (or *LP*) responsible for the computation within each strip. This *space-parallel* decomposition of the simulation can be viewed as partitioning the system being modeled into a collection

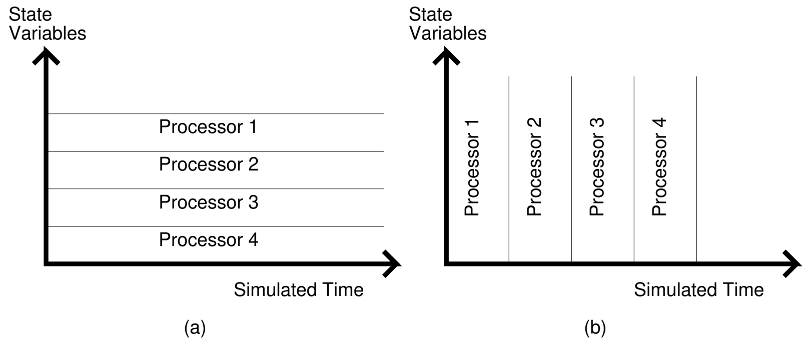

Figure 3: Space-time diagram depicting simulated time on the horizontal axis, and state variables on the vertical axis. (a) approach exploiting space-parallelism. (b) approach exploiting time-parallelism.

of subsystems, and assigning a logical process to simulate each one. For instance, in a simulation of a network of multiplexers, each LP might model a single multiplexer.

An alternative approach called the *time-parallel* method partitions the space-time graph into vertical strips as shown in Figure 3(b) and assigns a separate processor to each strip. The simulated time axis is divided into intervals $[T_1, T_2]$, $[T_2, T_3]$, $\ldots$, $[T_i, T_{i+1}]$, $\ldots$ with processor i assigned the task of computing the portion of the space-time graph within the interval $[T_i, T_{i+1}]$.

In the following we provide a brief introduction to space- and time- parallel simulation techniques. This is followed by a more detailed examination of new techniques using time-parallel simulation to simulate statistical multiplexers in asynchronous transfer mode (ATM) telecommunication networks. This paper is only concerned with the use of parallelism to speed up the execution of a *single* execution of the simulation program. Another useful form of parallel execution is to perform multiple, independent executions of the simulator on different processors, e.g., to reduce variance of output statistics or to examine different parameter settings. Such approaches offer clear benefits when they can be applied, but are beyond the scope of the present discussion.

2.1 Space-Parallel Simulation

Space–parallel simulation has been widely studied in the context of general purpose parallel simulation software. A central issue of space–parallel simulation is the so–called synchronization problem. Each logical process must process incoming events in timestamp order, or events in the simulated future might affect those in the past. The constraint that each LP must process events in timestamp order is referred to as the *local causality constraint.* The synchronization algorithm must ensure adherence to this constraint. Two classes of synchronization algorithms have been proposed to accomplish this. *Conservative* algorithms guarantee that events within an LP are never processed out of timestamp order, i.e., synchronization errors never occur. *Optimistic* algorithms allow synchronization errors to occur, but provide a mechanism to recover.

The principal task of any conservative simulation algorithm is to determine when it is "safe" to process an event, i.e., when an LP has received and processed all events containing a smaller timestamp. A variety of techniques have been proposed. Early algorithms focused on attacking deadlock problems that arise when LPs block because they are unable to identify safe events. Algorithms based on deadlock avoidance [CM79] and deadlock detection and recovery [CM81] were developed. A variety of techniques have since been proposed for identifying safe events [Fuj90, NF94].

A number of optimistic methods have also been proposed. Jefferson's Time Warp mechanism [Jef85] is the most widely known. It operates by detecting out–of–order execution of events and recovering using rollback. Other methods using techniques such as time windows [SBW88], memory management [ACDFS92] or preventing transmission of messages that may later be rolled back [DR90], among others [Fuj90], have been proposed. In addition to general purpose simulation algorithms, space–parallelism may be applied to problem specific algorithms. Applications include switching networks [GGN93], and queueing networks [HN93].

2.2 Time-Parallel Simulation

Recently, time-parallel simulation methods have received considerable attention for attacking specific simulation problems with well-defined objectives, e.g., measuring the loss rate of a finite capacity queue. Recall that time-parallel algorithms divide the simulated time axis into intervals, and assign each interval to a different processor (see Figure 3(b)). This allows for massively parallel execution because simulations often span long periods of simulated time.

A central question that must be addressed by time-parallel simulators is ensuring the states computed at the "boundaries" of the time intervals "match.". Specifically, it is clear that the state computed at the end of the interval $[T_{i-1}, T_i]$ must match the state at the beginning of interval $[T_i, T_{i+1}]$. Thus, this approach relies on being able to perform the simulation corresponding to the ith interval without first completing the simulations of the preceding $(i-1, i-2, ...1)$ intervals.

Because of the "state-matching" problem, time-parallel simulation is really more of a methodology for developing massively parallel algorithms for specific simulation problems than a general approach for executing arbitrary discrete-event simulation models on parallel computers. Time-parallel algorithms are currently not as robust as space-parallel approaches because they rely on specific properties of the system being modeled, e.g., specification of the system's behavior as recurrence equations and/or a relatively simple state descriptor. This approach is currently limited to a handful of important applications, e.g., queueing networks, Petri nets, and cache memories. General purpose parallel methods using Space-parallel simulations offer greater flexibility and wider applicability, but concurrency is limited to the number of logical processes. In some cases, both time and space–parallelism can be used [GGN93].

One approach to solving the state matching problem is to have each processor "guess" the initial state of its simulation, and then simulate the system based on this guessed initial state [LL91]. In general, the initial state will not match the final state of the previous interval. After the interval simulators have completed, a "fix-up" computation is performed to account for the fact that the wrong initial state was used. This might be performed, for instance, by simply repeating the simulation, using the final state computed in the previous interval as the new initial state. This "fix-up" process is repeated until the initial state of each interval matches the final state of the previous interval. In the worst case,

N such iterations are required when there are N simulators. However, if the final state of each interval simulator is seldom dependent on the initial state, far fewer iterations will be needed. In [HS90], the above approach is proposed to simulate cache memories using a least-recently-used replacement policy. This approach is effective for this application because the final state of the cache is not heavily dependent on the cache's initial state. The time-parallel approach has also been used to simulate G/D/1/k queues in [Lin93, WA92]. In both cases, fix–up phases are required even though the state–matching is not necessarily exact (e.g., the residual service time may be ignored or assumed to be the maximum possible value).

Here, we use a variation on this technique where we use a precomputation phase to identify points in simulated time where the state of the system is known. Specifically, we identify time periods that are guaranteed to produce an overflow (full queue) or underflow (empty queue). Because the state of the system, namely, the number of occupied buffers in the queue, is known at these points in simulated time, independent simulations can be begun starting at these points in simulated time. This eliminates the need for a fix-up computation.

Another approach to time-parallel simulation is described in [GLM91]. Here, a queueing network simulation is expressed as a set of recurrence equations that are then solved using well-known parallel prefix algorithms. The parallel prefix computation enables the state of the system at various points in simulated time to be computed concurrently. Another approach also based on recurrence equations is described in [BC93] for simulating timed Petri nets. New massively parallel algorithms for less tractable recurrence equations were developed for trace-driven cache simulations [NGLR92], and for circuit-switched communication networks [EGLW93, GGN93]. Ammar and Deng also use a related approach for simulating Petri nets [AD92].

2.3 Other Techniques

Rare event simulations and in particular buffer overflows in high–speed networks, require particularly lengthy simulations. Simulations of this type can be accelerated using a number of techniques based on *importance sampling* [CFM83, Fra92, PW89]. Typically, in importance sampling, the distributions describing in analytical terms the original system are manipulated in order to derive a new set of distributions. By replacing the original distributions by the new in the simulated system, the events of interest become more frequent. Thus, the simulation time requirements are decreased. The probability of the rare events in the original system is derived from the estimated probability of the modified system multiplied by the *likelihood ratio* which relates the probability of the sample paths of the two system settings. Thus, results of the simulation can be readily converted into performance results of the original system. Recent extensions [CHJS92] use the concept of *effective bandwidths* [GAN91] in the context of accelerations, and these extensions have been applied to accelerate the simulation of networks with tree–based topologies.

Importance sampling includes an overhead in selecting the distribution transformation that leads to the largest possible speedup and in establishing the likelihood ratio. At the same time, it necessitates the analytical description of the source traffic which is not always possible. In particular, recent findings in the area of source characterization [LMSTW93] suggest a fractal nature for the traffic process and reject commonly held beliefs on the analytical nature of traffic. Consequently, one important goal is to develop efficient parallel simulation techniques for trace–driven simulations, i.e., simulations driven by traces of actual network traffic, so no particular assumptions on the nature of source traffic are needed.

Finally, the parallel simulations of continuous time Markov chains using uniformization, as described [HN93], is not applicable to the systems we consider because it is not possible to define a Markov chain that exactly describes the system under investigation. At the same time, the construction of a Markov chain description for the system implies that numerical methods can also be applied for the solution of the stationary distribution of the system. Hence, the task of simulating the Markov chain may not be more efficient than solving it numerically in certain instances.

3 Exploiting the Structure of the Arrival Process

We now return to the problem of simulating statistical multiplexers and present a new approach to accelerating and parallelizing such simulations with particular emphasis on the fast generation of estimates for the cell loss ratio (CLR). The key assumptions that allow the development of the scheme are the following:

- The multiplexer is fed by a number of links with the same peak rate.
- The output link rate is an integer multiple of the input link rate.
- The service discipline is work–conserving (typically FCFS).
- The source arrivals occur back–to–back during the ON period.

All of the above assumptions are reasonable for an actual ATM multiplexer environment.

If we focus on the structure of the arrival process, we can identify one potential source of speedup. Namely, for the sake of generating cell loss statistics, it is not necessary to generate the individual cell arrival instances but rather use the concept of a "burst" and generate "burst" arrivals instead. We will call this simulation a *burst–level* simulation as opposed to the (traditional) *cell–level* simulation. Since bursts (i.e., aggregations of cells) are simulated instead of individual cells, we can expect significant simulation speedups. The more cells in a simulated burst, the better the overall efficiency compared to a traditional cell–level simulation.

To illustrate this point, consider the case of a single bursty source. Suppose also that arrivals of cells during the ON period occur at the rate of one per cell slot. One way of describing the arrivals that occurred during an ON interval is to generate all individual cell arrivals during the interval. Another way, is to generate a *burst descriptor* of the form $\langle s_i, d_i \rangle_{\mathcal{S}}$ where s_i describes the state as either ON or OFF and d_i is the actual length of the interval during which the source is in state s_i. The intervals of one source alternate between ON and OFF. Consequently, each $\langle \mathrm{ON}, d_i \rangle_{\mathcal{S}}$ corresponds to d_i consecutive cell arrivals while $\langle \mathrm{OFF}, d_i \rangle_{\mathcal{S}}$ corresponds to d_i consecutive empty (no cell arrival) slots.

Consider now the case of a multiplexer fed by a number of such sources. To simplify our analysis we will use the following notation:

- $Q(t_i)$: Queue length at the multiplexer at time t_i.
- $S(t_i)$: Number of cells serviced up to time t_i.

- $L(t_i)$: Number of cells lost up to time t_i.
- t_i: Time point at which a change in the number of active (ON) sources occurs.
- δ_i: Time interval between t_{i-1} and t_i, during which the number of active sources is constant.
- a_i: The number of sources active during time interval δ_i.
- K: The multiplexer buffer size.
- C: The output link capacity.

All the above quantities are integers. Because the size of cells is fixed, time can be expressed as integer multiple of the cell transmission time. Hence, time is discretized. The integer capacity C of the outgoing link expresses the number of cells that can be transmitted on the output link in time equal to a single cell transmission time on an input link. The queue length is observed at input link cell transmission epochs. Since an input link cell transmission epoch coincides with the C–th output link cell transmission epoch, i.e., with the termination of every C–th service epoch, there is no residual service to be accounted when the queue length observation is made. Hence, the state of the system is totally captured by the queue length.

Note that similarly to the individual source burst descriptor $\langle s_i, d_i \rangle_{\mathcal{S}}$, the burst descriptor of the aggregate arrivals can be represented as a sequence of $\langle a_j, \delta_j \rangle_{\mathcal{MS}}$ tuples, where a_i and δ_i are according to the definitions we presented. During an interval where the number of active sources is less than C, no increase of the queue size can occur and hence there will be no cell loss. We will call such an interval an *underload* interval. The definition of the *overload* interval is similar. That is, an underload interval is when $a_i \leq C$ and an overload when $a_i > C$. One may notice that $Q(t_i), S(t_i)$ and $L(t_i)$ are cell level statistics yet, as stated earlier, the simulation is performed at the burst level. Therefore, a key question is determining the cell level statistics from the burst level simulation. We address this question next.

Queue Length: During an overload, a queue fills at a rate $a_i - C$ until it exceeds K. Conversely, during an underload, the queue may decrease or even empty with a rate $C - a_i$. Thus:

$$Q(t_i) = \begin{cases} \min\{Q(t_{i-1}) + (a_i - C)\delta_i, K\} & a_i > C \\ \max\{Q(t_{i-1}) - (C - a_i)\delta_i, 0\} & a_i \leq C \end{cases} \tag{1}$$

Serviced Cells: While in overload, service is provided at a rate of C. However, during underload two things happen. First, none of the incoming cells is lost. Second, some of the capacity can be used to clear the queue that had build up previously. It may even be the case that the entire backlog in the queue is cleared and the queue becomes, eventually, empty. Hence:

$$S(t_i) = \begin{cases} S(t_{i-1}) + C\delta_i & a_i > C \\ S(t_{i-1}) + a_i\delta_i + Q(t_{i-1}) - Q(t_i) & a_i \leq C \end{cases} \tag{2}$$

Lost Cells: The number of lost cells does not increase so long as the system is in underload. When in overload, the number of lost cells *may* increase. The excess cells are received with a rate $a_i - C$. Of them $K - Q(t_{i-1})$ can be accommodated in the space remaining in the queue. The rest are lost. Therefore:

$$L(t_i) = \begin{cases} L(t_{i-1}) + \max\{(a_i - C)\delta_i - (K - Q(t_{i-1})), 0\} & a_i > C \\ L(t_{i-1}) & a_i \leq C \end{cases} \tag{3}$$

It is straightforward to extract the utilization and the cell loss ratio from the above quantities. The utilization is the fraction of serviced cells over the available cells on the output link (the latter is easily calculated as C times the sum of the cell slot times on the input links). The cell ratio is the fraction of lost cells over the total number of cells serviced or lost. Figure 4 demonstrates use of defined notation over a sample path of the system evolution. As an example, consider the first tuple $\langle 2, 17\rangle_{\mathcal{MS}}$ of the trace in Figure 4, that is, $\delta_i = 17$ and $a_i = 2$. Thus, the descriptor $\langle a_i, \delta_i\rangle_{\mathcal{MS}}$ represents $2 \times 17 = 34$ individual cell arrivals. We only need one update of equations (1–3) for simulating time which in a traditional simulation would require generation of 34 individual arrivals.

Three observations can be made at this point:

- The recurrences are not restricted to FCFS but to any work–conserving scheduling discipline where the queue size is enough to describe the state of the system. In this sense, e.g., a non–preemptive LCFS is also represented by the same recurrences, although its use in communication networks is limited.

- Despite their similarity to fluid approximation, the recurrences are exact because the arrivals during each period represented by an $\langle a_j, \delta_j\rangle_{\mathcal{MS}}$ descriptor are deterministic and with the given deterministic rate of a_i per input link slot time. Hence, the per–$\langle a_j, \delta_j\rangle_{\mathcal{MS}}$ description of arrivals is not a fluid approximation.

- Besides cell loss ratio, other statistics can also be derived including the delay and queue length distributions as seen by the aggregate traffic and the number of cells lost back–to–back. The first two can be calculated because the state of the queue is known while the last (cells lost back–to–back) can be calculated based on the observation that losses are always clustered together for certain overflow burst descriptors, as the definition of $L(t_i)$ indicates. For the sake of brevity we limit the discussion to the estimation of the cell loss ratio which is our primary interest. Note that it is not necessary that all performance measures of interest are calculated using recurrences. The presented recurrences simply facilitate the fast generation of cell loss ratio estimates.

Using equations (1–3), we can collect the values necessary for the estimation of the cell loss ratio without actually having to simulate each individual cell arrival. However, there is still need to generate the description of traffic arrivals in a certain format. That is, we need to generate tuples of the form $\langle a_j, \delta_j\rangle_{\mathcal{MS}}$. The simulation algorithm (still sequential) can then be split into two logical parts:

1. Generation of the $\langle a_j, \delta_j\rangle_{\mathcal{MS}}$ tuples from the $\langle s_i, d_i\rangle_{\mathcal{S}}$ tuples.

2. Updates of the queue length, serviced cells and lost cells using equations (1–3).

Note that in the proposed method, simulated time "jumps" forward from t_i to t_{i+1}. In this sense, the algorithm is not much different from any other simulation where a global clock

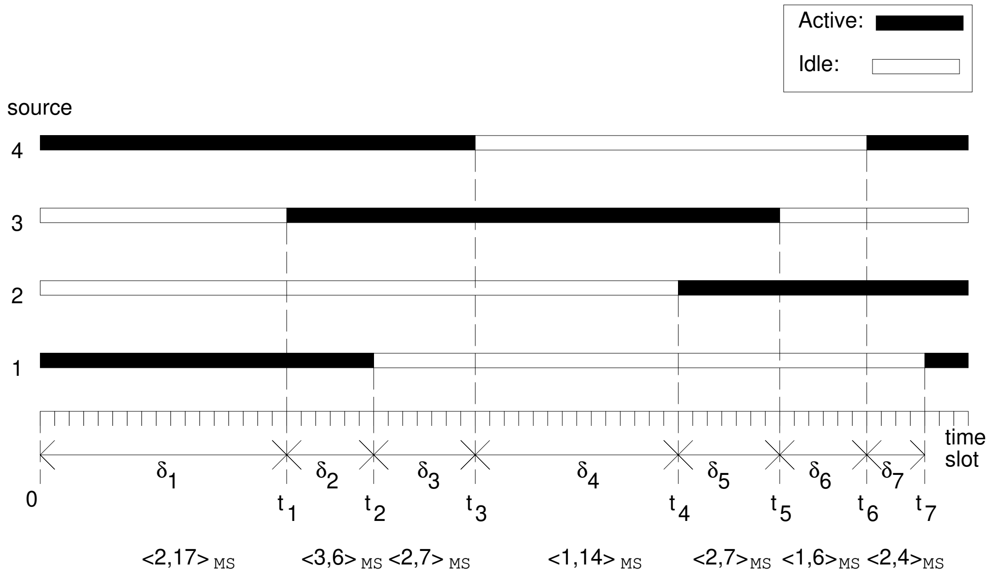

Figure 4: Example evolution of the multiplexer system demonstrating use of the defined notation.

is used. However, instead of scheduling the cell arrival events we schedule the events that describe changes of the number of active sources. The processing for every such event is the calculation of equations (1–3) using a single $\langle a_j, \delta_j \rangle_{\mathcal{MS}}$ that corresponds to the event. Also note that the generation of the $\langle a_j, \delta_j \rangle_{\mathcal{MS}}$ tuples from the $\langle s_i, d_i \rangle_{\mathcal{S}}$ tuples is straightforward since δ_i represents the minimum residual time to the next state change of the N sources. Therefore, it involves finding the minimum of N values[2]. Figure 4 indicates the fact that all t_i's are actually the time points where the intervals defined by the $\langle s_i, d_i \rangle_{\mathcal{S}}$ tuples terminate.

3.1 Expected Performance of the Technique

Before getting into the specifics of the approach and in particular the way to generate $\langle a_j, \delta_j \rangle_{\mathcal{MS}}$, we will comment on its expected performance. An indication of the performance for the proposed method is the average δ_i. The larger the average δ_i, the longer the average time interval represented by a single $\langle a_j, \delta_j \rangle_{\mathcal{MS}}$ tuple. That is, the larger the average δ_i, the more time we "compress" by using the burst–level simulation approach. We need to execute equations (1–3) only once for each tuple.

Note however, that δ_i is *not* the speedup over a cell–level simulation. Moreover, the actual speedup is also related to the value of a_i. In fact, an $\langle a_j, \delta_j \rangle_{\mathcal{MS}}$ tuple is equivalent to $a_i \times \delta_i$ individual cell arrivals. Hence, a more comprehensive performance measure is the average value of the product $a_i \times \delta_i$, representing the average number of cell arrivals "compressed" in one tuple. The resulting performance index (which we will call the *compression*) is more closely related to the speedup over a traditional event–list based simulation. Indeed, in traditional simulations the execution time is proportional to the number of processed events. The *compression* we introduce represents the average number of events in the traditional simulation that correspond to the processing of only one event in the proposed method. An

[2]Later we will indicate how the problem can be restated such that no calculation of the minimum residual time is needed.

exact analytical prediction of the actual speedup over a traditional simulation is difficult to derive since the simulation methods involve different costs for processing the events (event list management and updates of the event counts in the cell–level simulation, state transition event management and updates of the related equations in the burst–level simulation) hence a break–even point exists that is dependent on the different constant costs.

We base both the construction of the aggregate traffic description, $\langle a_j, \delta_j \rangle_{\mathcal{MS}}$, and the derivation of the potential performance of the scheme, on the construction of an underlying DTMC. First, we define the stochastic process that represents an individual source process. Suppose that we are given N identical ON/OFF sources where a source remains in the ON state with probability $1-p$ or changes to the OFF state with probability p. Similarly, a source remains in the OFF state with probability $1-q$ or changes to the ON state with probability q. Let the average sojourn time (in slots) in the ON state and the OFF state be respectively E[ON] and E[OFF]. Then, obviously E[ON]$=(1/p)$ and E[OFF]$=(1/q)$.

In the second step, we define the Markov chain which describes the aggregate activity of all N sources. Fortunately, the state space of the Markov chain is not of size 2^N as one would expect because the ON and OFF states of all the sources are identical. It is sufficient to define an $N+1$ state Markov chain where the meaning of state i is that exactly i sources are in the ON state (the rest $N-i$ being in the OFF state). Assume now that we are given a DTMC, as the one defined above, with non–zero transition probabilities from a state to itself. We can construct a new DTMC without transition probabilities from a state to itself by setting for each i the probability $P_{i,i}$ to zero (i.e., not allowing transitions from a state to itself) and normalizing for each state i all the probabilities $P_{i,j} (i \neq j)$ such that their sum over all j's is one. We now define a new experiment whereby we follow a sample path on the new DTMC and where in every state i we produce a sojourn time from a geometric distribution with parameter $P_{i,i}$. What we can accomplish with this transformation is instead of generating a sample path on the original DTMC of the form, say "..., 1, 1, 2, 2, 2, 2, 3, 1, 2, 2, 5, ...", we can use the new DTMC and the independent geometric distributions to produce equivalently the same trace in the form "..., 1(2), 2(4), 3(1), 1(1), 2(2), 5(1), ...". The number in parentheses is the number of subsequent time units that we stay in the same state. The two experiments are equivalent, in the sense that they can be used interchangeably to denote the same process. However, the latter DTMC reflects directly the sequence of $\langle a_j, \delta_j \rangle_{\mathcal{MS}}$ tuples. Indeed, subsequent values of the a_i's are the states visited by subsequent transitions of the new DTMC, while the δ_i's are the sojourn times sampled from the geometric distribution corresponding to the current state[3].

First, the calculation of the transition probabilities of the underlying DTMC is necessary. Let the state space of the DTMC be indexed from 0 to N where the index of the state stands for the number of active sources. We can identify three cases:

- transitions from a state k to itself $(0 \leq k \leq N)$:

$$P^*_{k,k} = \sum_{x=0}^{\min(k,N-k)} \binom{k}{x} \binom{N-k}{x} (1-p)^{k-x} (1-q)^{N-k-x} p^x q^x \tag{4}$$

[3]Observe, that to simplify our analysis, the δ_i is redefined to account only for changes in the cumulative number of active sources (without regard for *which* sources are active). This redefinition simplifies considerably our analysis since we are interested only in the cumulative number of active sources in order to generate the $\langle a_j, \delta_j \rangle_{\mathcal{MS}}$ tuple sequence.

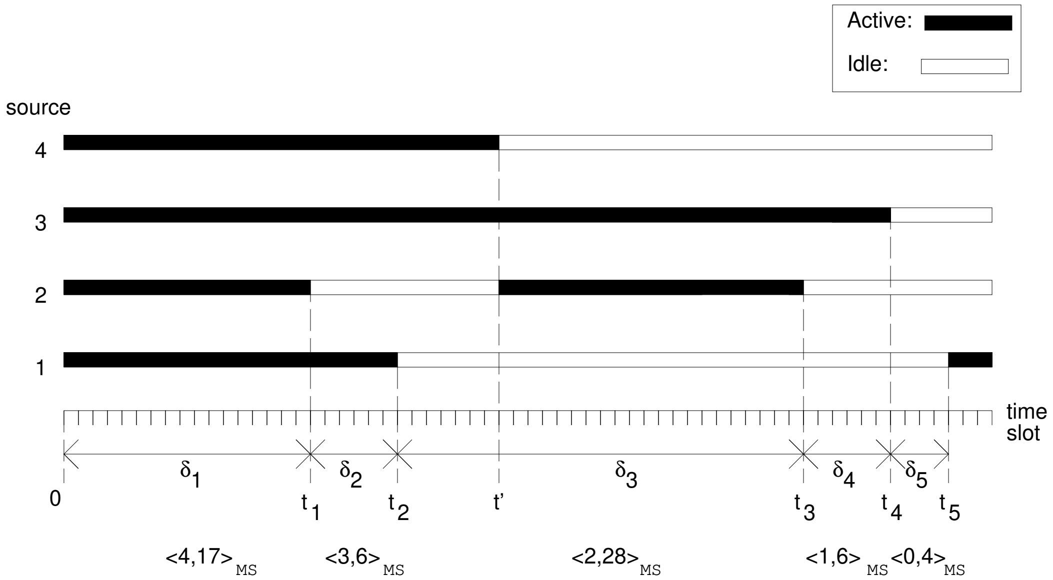

Figure 5: System evolution where at time t' the total number of active sources does not change although the sources that are active change. That is, at the same time that source 4 becomes idle, source 2 becomes active.

- transitions from a state i $(0 \leq i < N)$ to a state k $(0 < k \leq N)$ where $i < k$ (i.e., increase of active sources):

$$P^*_{i,k} = \sum_{x=0}^{\min(i,N-k)} \binom{i}{x} \binom{N-i}{k-i+x} (1-p)^{i-x} (1-q)^{N-k-x} p^x q^{k-i+x} \tag{5}$$

- transitions from a state i $(0 < i \leq N)$ to a state k $(0 \leq k < N)$ where $i > k$ (i.e., decrease of active sources):

$$P^*_{i,k} = \sum_{x=0}^{\min(k,N-i)} \binom{i}{i-k+x} \binom{N-i}{x} (1-p)^{k-x} (1-q)^{N-i-x} p^{i-k+x} q^x \tag{6}$$

Notice that the sum in equation (4) (similarly for equations 5 and 6) is required to account for all possible combinations of sources that "flip" from the ON state to the OFF state and vice versa *without* changing the total number of ON sources (cf. Figure 5). Moreover:

$$Pr\{k \text{ active sources (cumulatively) for } l \text{ cell times}\} = (P^*_{k,k})^l (1 - P^*_{k,k}) = p^*_k(l) \tag{7}$$

Thus, the average sojourn time in state i will be:

$$\mathrm{E}[\text{in state } i] = \frac{1}{1 - P^*_{i,i}} \tag{8}$$

At this point we can use the method we described earlier to construct a new DTMC from the DTMC that we have just defined by separating the transitions from a state to itself and normalizing the remaining probabilities for each state. That is, the new DTMC is defined by the transition probabilities:

$$P_{i,k} = \frac{P^*_{i,k}}{\sum_{l=0 \quad l \neq i}^{N} P^*_{i,l}}, \qquad k \neq i \tag{9}$$

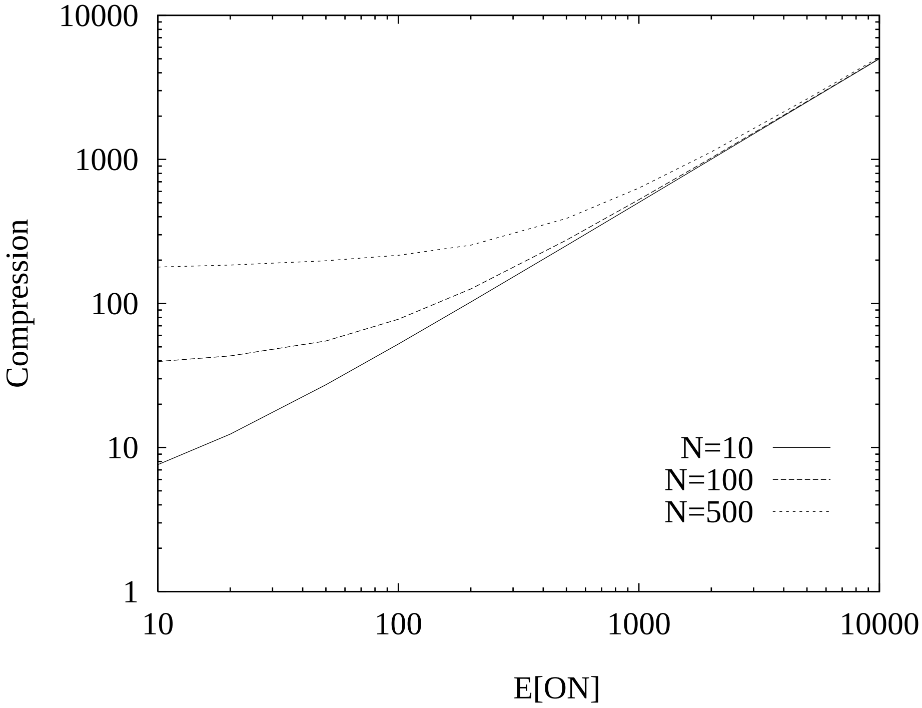

Figure 6: Influence of the average active period, E[ON], in cell times on the compression.

Evidently, $P_{i,i} = 0 \ \ \forall i$.

This DTMC will drive the generation of the a_i's based on a trajectory using the $P_{i,k}$'s while the δ_i's will be sampled from the distribution of equation (7). Moreover, the new DTMC will have a steady state distribution, say π. Thus the steady state probability of being in state i will be π_i. Consequently, it follows that the average value of the product $a_i \times \delta_i$ is:

$$\text{compression} = \mathrm{E}[a_i \times \delta_i] = \sum_{x=0}^{N} \frac{x}{1 - P^*_{x,x}} \pi_x \tag{10}$$

The two parameters that control the compression (expressed as cells per tuple) are the number of sources N and the average sojourn in the active state E[ON].[4] Figures 6 and 7 demonstrate the influence of the average sojourn E[ON] and of the number of sources N respectively on the compression. Figure 8 presents a three–dimensional view of the analytical results concerning the compression. As expected, the compression is decreasing for an increasing number of sources or for a decreasing average ON sojourn time.

4 Parallelization of the Method

One of the problems with parallelizing the presented method is the need to calculate equations (1–3). These equations contain a strong dependence on the queue length calculated in the previous iteration. However, an approach similar to [GLM91] cannot be used because

[4] For the sake of simplicity, the mean ON interval is set to half the mean OFF interval – this is approximately consistent with certain voice traffic models. Hence, only one of the two intervals (namely, the E[ON]) need to be changed, since E[OFF]=2×E[ON].

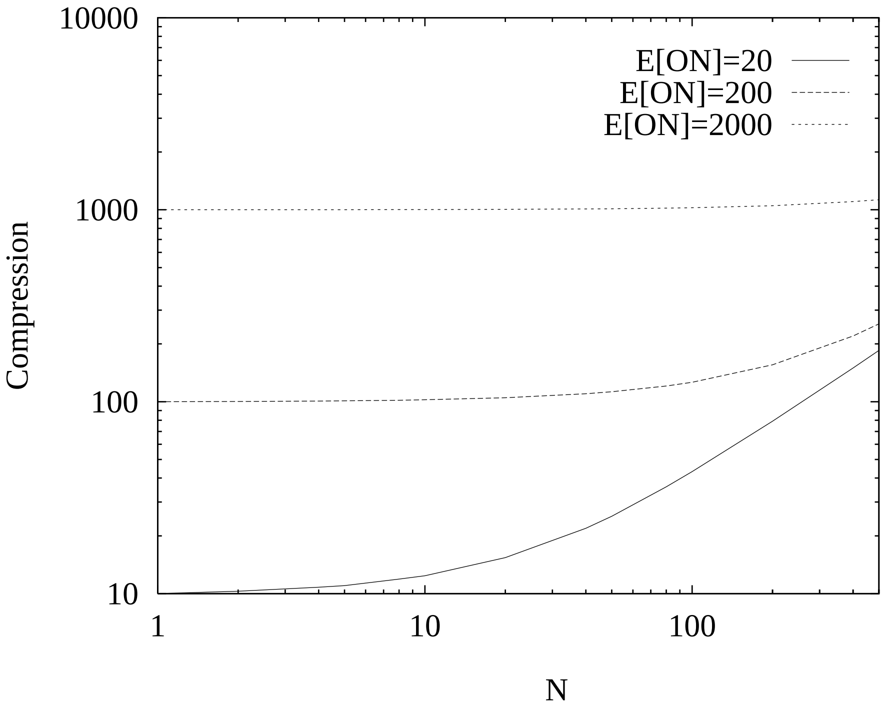

Figure 7: Influence of the number of sources, N, on the compression.

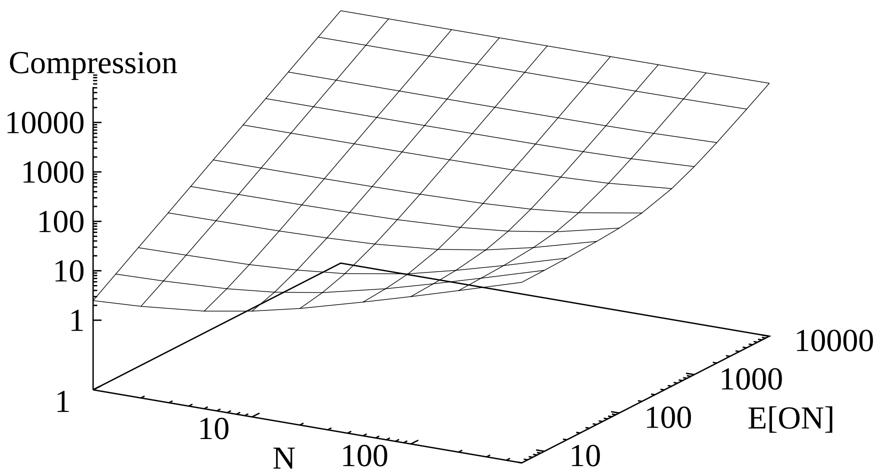

Figure 8: A three-dimensional view of the expected average compression under different E[ON] sojourn times and number of sources.

we would have to discriminate[5] between overload and underload periods when calculating the recurrences. Thus, two operators (defining a semi–ring as [GLM91] assumes) are not sufficient to describe the finite buffer queue. Therefore, we can consider that equations (1–3) are calculated sequentially.

In order to include parallelism, we can turn to a form of time–division (time–parallel) simulation based on the following observations: Assume a multiplexer is fed with an arrival stream represented as a sequence of $\langle a_j, \delta_j \rangle_{\mathcal{MS}}$ tuples, then there exist two well–defined types of events for a buffer size K multiplexer:

- **Guaranteed Overflow.** This is a point in time (for convenience identified at a state transition epoch, using the definitions we introduced) where we can guarantee that the queue length is K at end of the tuple's period. The way to identify this point is to look for an $\langle a_j, \delta_j \rangle_{\mathcal{MS}}$ tuple where $a_i > C$ and $\delta_i > K/(a_i - C)$. That is, the system is in overload for sufficiently long to guarantee that even an empty buffer will become full before the next state transition (i.e., the next tuple). Consequently, an *independent* simulation can begin immediately after the tuple that guarantees buffer overflow because the system (the number of active sources and the length of the queue) are known. The simulation starting at this point can use the accurate assumption that the initial buffer occupancy is K. The probability to find an $\langle a_j, \delta_j \rangle_{\mathcal{MS}}$ tuple that causes a guaranteed overflow is:

$$p_{over} = \sum_{j=C+1}^{N} \pi_j (1 - \sum_{k=0}^{K/(j-C)} p_j^*(k)) \tag{11}$$

- **Guaranteed Underflow.** This is a point in time where we can guarantee that the queue length is zero. The way to identify this point is to search for an $\langle a_j, \delta_j \rangle_{\mathcal{MS}}$ tuple where $a_i < C$ and $\delta_i > K/(C - a_i)$. That is, the system is in underload long enough to guarantee that even a full buffer will be empty by the end of the interval (in time for the next tuple). Hence, an independent simulation can begin immediately after the tuple that guarantees underflow of the buffer. This simulation can assume accurately that the initial buffer occupancy is zero. The probability to find a tuple that causes a guaranteed underflow is:

$$p_{under} = \sum_{j=1}^{C-1} \pi_j (1 - \sum_{k=0}^{K/(C-j)} p_j^*(k)) \tag{12}$$

As a consequence, the guaranteed overflow and underflow $\langle a_j, \delta_j \rangle_{\mathcal{MS}}$ tuples can be used as the points defining the intervals of the time division simulation. Their identification (from the cumulative descriptors) is trivial to parallelize; we simply split the sequence of cumulative descriptors to all available processors and each processor tries to locate the tuples conforming to the rules we presented. The method seems promising but an extensive validation is required to support its effectiveness. First, we must ensure that there are a sufficient number of time division points, as this indicates the amount of parallelism that can be exploited. To illustrate the case of guaranteed overflow/underflow we provide a table of the p_{over} and p_{under}

[5] Or, alternatively, introduce a new operator to take care of the related implicit "if" statement in the separate of overload and underload $\langle a_j, \delta_j \rangle_{\mathcal{MS}}$ tuples.

values for a simulation of a multiplexer with 10 bursty sources with mean active period of 100 cells and mean inactive period of 200 cells for different link capacities and buffer sizes:

C	K	p_{over}	p_{under}	p_{TD}
6	5	0.012426	0.768843	0.781269
6	10	0.008455	0.664632	0.673087
6	20	0.004088	0.518076	0.522164
3	5	0.324936	0.225882	0.843012
3	10	0.250769	0.178151	0.428920
3	20	0.158550	0.115765	0.274315

The resulting probability to generate a time division point is the sum of the two independent events to generate a guaranteed overflow or a guaranteed underflow tuple. That is:

$$p_{TD} = p_{over} + p_{under} \tag{13}$$

In practice, there will usually be many guaranteed underflows because multiplexers are designed to have few overflows under anticipated traffic patterns. In the following we will describe p_{TD} as a percentage and will call it the *time division density*. It will denote the portion of the generated tuples that can be used as time division points because they represent either guaranteed overflow or guaranteed underflow points. It is also important to note that a very small time division density (e.g., as low as .01 %) is sufficient for effective use of our method. Under most configurations, this density is exceeded by several orders of magnitude. To illustrate the point, consider an LP with a tuple trace of approximately 500000 tuples (such a trace is not unreasonable since it typically requires only two to three Megabytes of memory). Then with a time division density of .01 %, we can expect on the average 50 such time division points in the tuple trace. We can thus achieve up to 50–fold parallelism.

The interval between two consecutive time division points can be very short when the time division density is high. In this case, it may not be efficient to allocate a processor for the execution of each interval. That is, the cost of starting a time division simulation between two time division points that are very close together may exceed the benefit of parallel execution. Consequently, it is preferable to run time division simulations that span over a number of successive time division points. In general, if the simulation is rich in time division points, one would only utilize enough to execute a separate simulation on each available processor. In this case, the selection of time division points would be made to balance the workload among the available processors. In the case of the implementation we present, we do not use a sophisticated load–balancing scheme. Each LP generates a string of $\langle a_j, \delta_j \rangle_{\mathcal{MS}}$ tuples and starts simulating from the first $\langle a_j, \delta_j \rangle_{\mathcal{MS}}$ that is detected as a guaranteed overflow or underflow. Hence, the reader should take into account that better performance may be possible if elaborate, albeit complex, load balancing is used.

The time division density is not only influenced by the active sojourn times and the number of sources, but also the link capacity and the multiplexer buffer as well. In Figure 9, the dependence of the time division density on the buffer size is indicated for different buffer sizes. Note that the ratio of buffer size to average burst length plays a vital role in the cell loss probability which is the central objective of our simulations.

Figure 10 describes the dependency of the time division density on the fraction of sources that the link capacity represents. That is, to achieve multiplexing, the capacity of the output

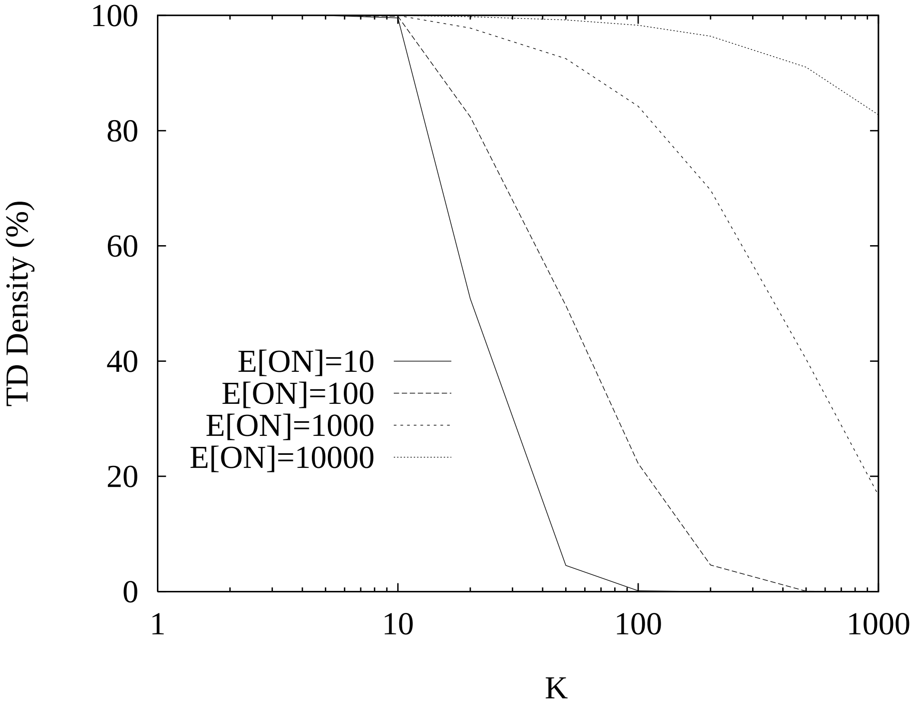

Figure 9: Dependence of time division density on the buffer size, K. The number of traffic sources, N, is 50 and the link capacity, C, is 35.

link is a fraction of the aggregate capacity of the incoming links. This fraction plays a role on how fast/slow the buffer can be emptied/filled during underload/overload.

A three-dimensional plot of the TD density relative to the buffer size and the ratio of capacity over the number of sources is given in Figure 11. Note that the dip present in Figure 11 and, more pronounced, in Figure 10 near a C/N ratio of .33 is due to the fact that the sources remain active for approximately one third of the time. This is an artifact of the assumption that the average idle time is twice the active time[6]. At this point, the average rate of traffic entering the multiplexer exactly matches the output link capacity, so the number of guaranteed overflow and underflow points is diminished. As noted earlier, in practice, one would expect most multiplexers to operate far to the right of this point so that overflows seldom occur.

An obvious benefit of the parallelization technique we have described is that the time–division simulation can be performed without any need for a second “fix-up” stage. It is enough to accumulate separately the statistics of each time–division simulation partition and then calculate the overall statistics at the end of the simulations. Summarizing, the steps needed to perform a parallel simulation of a bursty source multiplexer are:

1. Generation of the $\langle a_j, \delta_j \rangle_{\mathcal{MS}}$ tuples.

2. Identification of the guaranteed overflow/underflow points.

3. Time–division simulation between the guaranteed overflow/underflow points.

[6]Since E[OFF] = 2 × E[ON], an input link is utilized one third of the time. Consequently, in order to have a stable system (under the queuing theoretic notion of stability) we need $C \geq (1/3)N$. The range to the left of 0.3 represents unstable systems for which it is unlikely that simulations will be run.

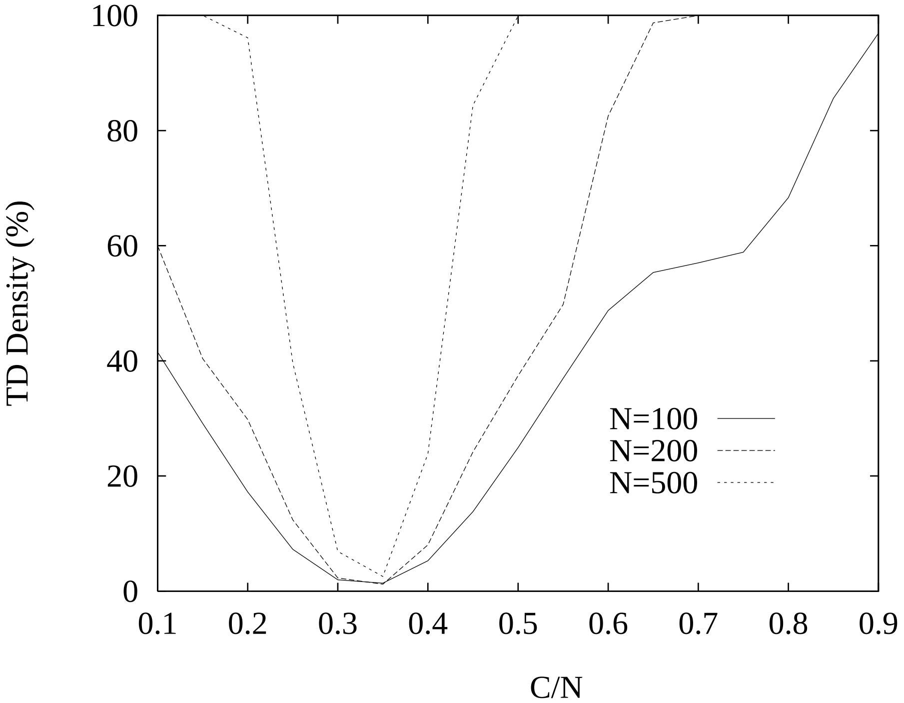

Figure 10: Dependence of time division density on the ratio of link capacity over number of multiplexed sources, C/N. The buffer size, K, is 50 cells and the average burst length, E[ON], is 100 cell times.

4. Calculation of overall statistics from the independent time–division runs.

The second and third steps are parallelizable. The last step may be inherently sequential but fortunately it is not the dominant part of the algorithm's execution time. Next, we discuss the parallelization of the first step for the case of source models specified as discrete-time Markov chains.

4.1 Parallel Generation of $\langle a_j, \delta_j \rangle_{\mathcal{MS}}$ Tuples

The generation of a sequence of $\langle a_j, \delta_j \rangle_{\mathcal{MS}}$ tuples may, in general, be accomplished by simulating the behavior of each source on a separate processor (we assume the sources are independent) and using a parallel merge operation to combine the separate streams. This approach is trivially extended to perform simulations based on traces of source behavior. Here, we describe an alternative approach for the case where each source's behavior is specified by a two state Markov chain. This approach involves the generation of subsequent state transitions in the Discrete Time Markov Chain discussed earlier. Consequently, the problem becomes the parallel generation of state transitions of the DTMC. Although methods such as the PUCS [HN93] simulation have been developed to deal with the parallel simulation of Markov chains, they are more likely to be used in the simulation of large queuing systems where the decomposition of the state space can aid the parallelism. However, PUCS implies communication overheads. We would like to (ideally) avoid these communication/synchronization costs. In order to avoid communication/synchronization costs, the following property of the Markov Chain can be exploited: For the specific DTMC in our problem, all states are recurrent. That is, any state will be visited an infinite number of

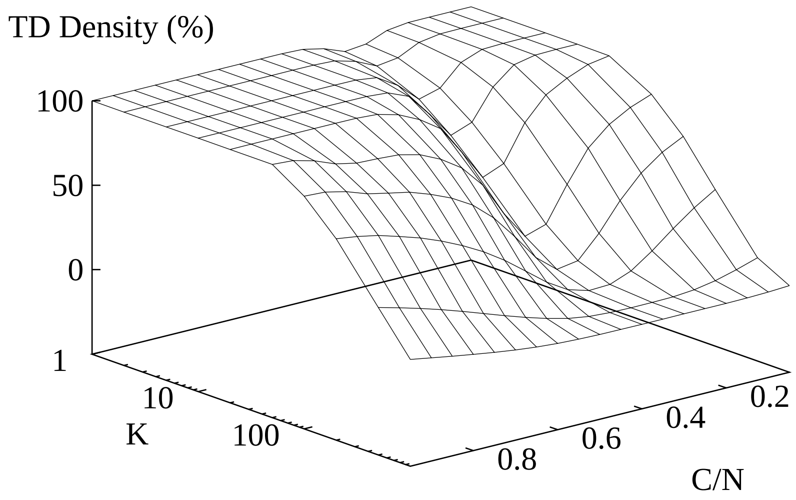

Figure 11: Dependence of time division density on the ratio of link capacity over number of multiplexed sources, C/N, and the buffer size, K. The number of sources, N, is 20 and the average burst length, E[ON], is 500 cell times.

times. Therefore, if we start simulating state transitions from a state, we can be sure that sooner or later we will end up in the same state again.

The above observation offers an interesting opportunity for a regeneration–based parallelization. Suppose we wish to simulate the state transitions of the DTMC for a sufficiently long time. In our case, we need to generate a string of tuples of the form $\langle a_j, \delta_j \rangle_{\mathcal{MS}}$. Generating the δ_i part is not an issue since it can be generated by sampling a geometric random variable as equation (7) indicates, provided we know the number of active sources a_i. Generating a_i is the actual issue. However, successive a_i's are the trace of the state transitions occurring in the DTMC.

Assume we have k logical processes, then we can assign a specific state of the DTMC (say a) as the state from which, and back to which, each LP produces a sample path of the DTMC. That is, each LP starts simulation of the DTMC from state a and it stops the generation when it returns back to a. Since all LPs use the same state as beginning and end of the traces they generate, there is no problem of "matching" the states when the traces produced by different LPs are put together, i.e., assumed to be concatenated in time. Figure 12 illustrates the operation of the different LPs.

The selection of the a with the highest steady state probability is beneficial if for each sample path we allow many recurrences back to a. That is, we can fix a maximum number of tuples L that we wish to produce for each logical process and then we use as a the state with highest π_a. We then generate this maximum number of tuples beginning from one with state a. It is possible that this generated trace will not end at a. In that case, we discard the part of the trace from the last recurrence to a to the end of the trace. Since the average time between recurrences of a is $1/\pi_a$, the average length (in tuples) of the trace we discard is less than $1/\pi_a$. The remaining trace begins at a and ends with a state from which we will

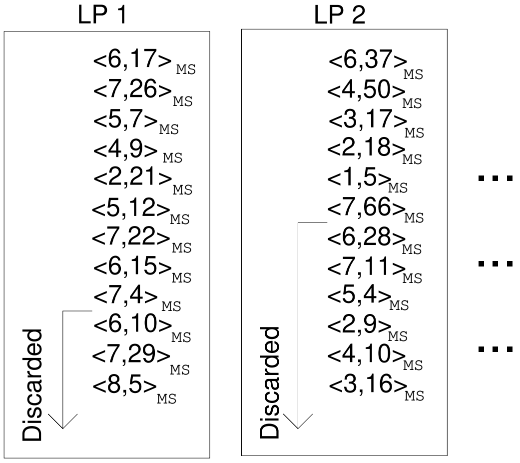

Figure 12: Example of sample paths produced from the LPs with $N = 10$, $a = 6$ and $L = 12$ indicating the truncated (discarded sequences) after the last recurrence to a (including the last visit to a).

transit back to a and can thus be used to achieve parallelism. This approach is similar to the one used in [WA92] for the temporal decomposition using recurrent states.

We note that the restriction, L, on the number of tuples generated for the trajectory of the DTMC, may result to serious problems to the accuracy of the DTMC simulation because there exists a correlation between the L and the maximum possible length of sample trajectory between the selected recurrent state. By setting a "short" L, we may in fact severely misrepresent the dynamics of the DTMC. For this reason, we take an approach suggested in [Heid88], and we enforce on each LP to complete *at least one* complete regeneration (from a back to a) even if this entails exceeding the limit on the recorded trajectory, L. In practice, a large value of L, implies that this exception is rarely (if ever) used. In our experience, with runs ranging from a few minutes to several days, a call to this exception, i.e., extending the generated trace beyond L tuples in order to generate at least one complete recurrence path from a back to a, has not been observed yet.

A block diagram of the successive parallel stages for the simulation of a statistical multiplexer, including the parallel generation of the tuples, is shown in Figure 13.

4.2 Parallel Implementation

An implementation of the proposed algorithm on a 32 processor KSR 2 multiprocessor has been completed. Each processor generates its own trace of tuples independently of each other. All traces start from the same recurrence point of the underlying DTMC, the one with the highest steady state probability. While the generation of the tuple trace is performed, the first time division point is located. Upon termination of the tuple generation (back to a tuple with active sources corresponding to the recurrence point), the simulation using the tuples and the recurrence equations can begin. The simulation can cover the trace from the first time division point until the end of the tuple trace. However, the part of the trace before the first time division point (called the *preamble*) cannot be simulated before the simulation of the previous segment (with the exception of its preamble) is completed and an initial state regarding the queue length can be provided to the current segment. After receiving

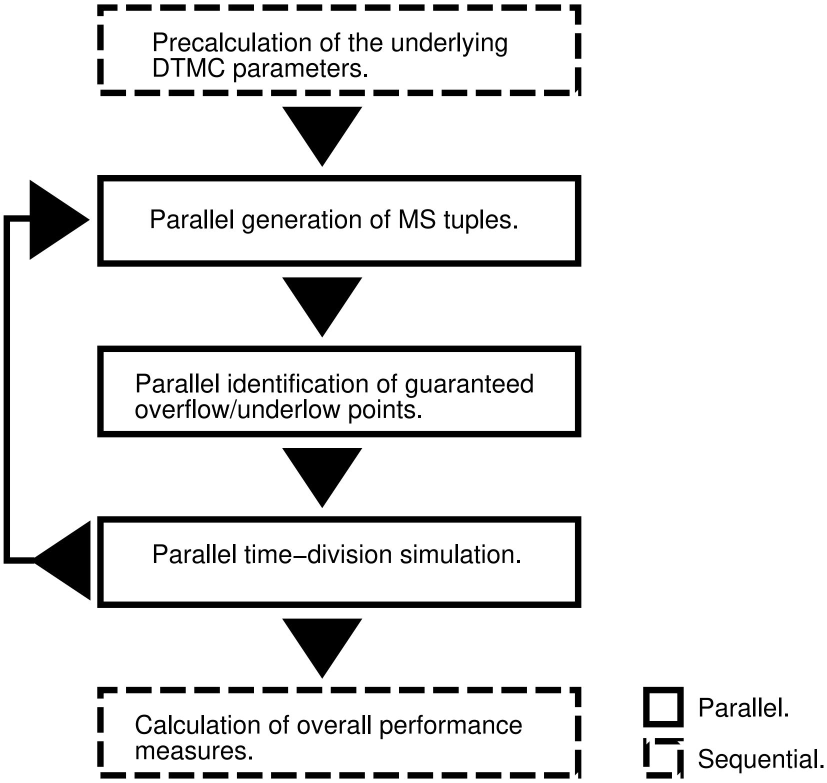

Figure 13: Successive parallel stages for the simulation of statistical multiplexers.

this initial state, the preamble simulation is run and the statistics updated. Finally, all processors synchronize together at a barrier after their individual simulations have ended in order to collect their individual results and report the statistics. A new iteration of the entire algorithm can then begin. A number of optimizations are possible, including, but not limited to, keeping processors busy using a suitable load–balancing scheme and not synchronizing them at a barrier. However, the objective of this implementation was to indicate the achievable speedup without much sophistication in the actual coding of the algorithm. Thus, load balancing and low–level synchronization techniques, other than barrier synchronization, were not considered.

The difficult part of a comparison with a sequential event–list algorithm is to set a common ground that does not give unfair advantage to the parallel algorithm. Hence, several optimizations were performed in the sequential code, including total disregard for departure events (since they are not essential for the statistics we gather, they were omitted with suitable coding) and minimization of the required random number generator invocations. The measure of comparison is the number of cell arrivals processed per unit of time (second). Since the parallel simulation has no cell arrivals, we use the information in the tuples to find the *equivalent* number of cell arrivals represented by a tuple.

Figure 14 presents four different experiments that were used to test the simulator efficiency. The last two columns ("Speed") is the simulation speed in terms of thousands of simulated cell arrivals per second of real time on a 1 and a 32 processor configuration. The single processor performance represents essentially the benefit from using the compression, i.e., burst–level simulation alone, without the introduction of parallelism. Note that the time–parallel simulation we use achieves linear speedup with respect to available processors

	N	E[ON]	C	K	Speed (1)	Speed (32)
A	5	10	3	10	402	12528
B	10	100	7	100	3415	106330
C	50	1000	35	1000	29611	859935
D	10	10000	6	3000	323344	10077840

Figure 14: Four experiments (A, B, C, and D) and the simulation speed (in *thousands* of cell arrivals processed per second) on a 1 and a 32 processor configuration. N is the number of ON/OFF sources, E[ON] is the average ON (burst) length, C is the relative output link speed of the multiplexer and K is the buffer size of the multiplexer.

when there exist no fix–up phases, as is the case in our technique. Hence, e.g., for experiment D, a speed of about ten billion cell arrivals per second was observed on a 32 processor configuration. Similarly a speed of about twelve million cell arrivals per second was observed for a 32 processor configuration of experiment A. Compared to these results, the highly optimized sequential simulation, in the best case, could process a maximum of 140 thousand cell arrivals per second on a SUN Sparc 10 utilizing a splay–tree event list algorithm without other interfering user load.

To illustrate the benefit of estimating cell loss ratio with good confidence intervals very fast, consider the example configuration B. The cell loss ratio was found to be 3.16×10^{-5} within a range of plus or minus 3.84 % for a confidence interval of 95 % in just 72.4 wallclock seconds of 32 processor execution time.

5 Conclusions

In this paper, we describe some of the inherent challenges and opportunities of simulating cell multiplexers for ATM telecommunication systems under bursty traffic. The limited applicability of existing parallel simulation techniques lead us to the development of a new technique. The technique presented herein combines burst–level simulation with parallel simulation to achieve significant overall reductions in the run times needed to study some practical applications in ATM telecommunication networks. The results indicate that indeed a better understanding of the burst–level characteristics of the sources can lead to substantial savings in simulation times. Note that the approach of separating the burst–level behavior has also been introduced in the framework of analytical work in [Hui89] and is known as the *hierarchical approach*.

It is interesting to further exploit the possibilities of such an approach. Notably, the inclusion of Constant Bit Rate (CBR) sources does not complicate the simulation approach since a CBR connection can be considered as reserving a fixed portion of the link bandwidth away from other connections. Currently, the mechanisms to support different arrival patterns in the ON state as well as the generalization to arbitrary multiplex hierarchies are under investigation, focusing on the extensions to topologies that are of practical interest. Some preliminary results to this extend have also been generated and they involve a tedious construction process for characterizing the departures from multiplexers while maintaining the concept of guaranteed overflows and underflows for the sake of parallelism.

Furthermore, the identification of time division points is based on the observation of the impact that an $\langle a_j, \delta_j \rangle_{\mathcal{MS}}$ tuple has on the queue state. It is possible to find more time division points if the cumulative effect of a sequence of tuples on the queue state is considered. This is especially beneficial in configurations where our analytical results have indicated that the expected time division density is very close to zero.

References

[ACDFS92] I. F. Akyildiz, L. Chen, S. R. Das, R. M. Fujimoto and R. Serfozo. Performance analysis of Time Warp with limited memory. In *Proceedings of the 1992 ACM SIGMETRICS Conference on Measuring and Modeling Computer Systems*, pages 213–224, May 1992.

[AD92] H. Ammar and S. Deng. Time warp simulation using time scale decomposition. *ACM Transactions on Modeling and Computer Simulation*, 2(2):158–177, April 1992.

[BC93] F. Baccelli and M. Canales. Parallel simulation of stochastic petri nets using receurrence equations. *ACM Transactions on Modeling and Computer Simulation*, 3(1):20–41, January 1993.

[BLC91] R. Bagrodia, W.-T. Liao, and K.M. Chandy. A unifying framework for distributed simulation. *ACM Transactions on Modeling and Computer Simulation*, 1(4):348–385, October 1991.

[CFM83] M. Cottrell, J.-C. Fort, and G. Malgouyres. Large deviations and rare events in the study of stochastic algorithms. *IEEE Transactions on Automatic Control*, AC-28(9):907–920, September 1983.

[CHJS92] C.-S. Chang, P. Heidelberger, S. Juneja, and P. Shahabuddin. Effective Bandwidth and Fast Simulation of ATM Intree Networks. Technical Report RC 18586, IBM T. J. Watson Research Center, November 1992.

[CM79] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–452, September 1979.

[CM81] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(4):198–205, April 1981.

[DR90] P. Dickens and P. Reynolds, Jr.. SRADS with local rollback. *Distributed Simulation*, 22(2):161–164, January 1990.

[EGLW93] S. G. Eick, A. G. Greenberg, B. D. Lubachevsky, and A. Weiss. Synchronous relaxation for parallel simulations with applications to circuit switched networks. *ACM Transactions on Modeling and Computer Simulation*, 3(4):287–314, October 1993.

[Fra92] M. R. Frater. Application of fast simulation techniques to systems with correlated noise. In *Proc. 1992 Winter Simulation Conference*, pages 448–452, 1992.

[Fuj90] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[GGN93] B. Gaujal, A. G. Greenberg, and D. M. Nicol. A sweep algorithm for massively parallel simulation of circuit-switched networks. *Journal of Parallel and Distributed Computing*, 18(4):484–500, August 1993.

[GLM91] A. G. Greenberg, B.D. Lubachevsky, and I. Mitrani. Algorithms for unboundedly parallel simulations. *ACM Transactions on Computer Systems*, 9(3):201–221, 1991.

[Gru82] J. G. Gruber. A comparison of measured and calculated speech temporal parameters relevant to speech activity detection. *IEEE Transactions on Communications*, COM-30(4):728–738, April 1982.

[GAN91] R. Guérin, H. Ahmadi, and M. Naghshineh. Equivalent capacity and its application to bandwidth allocation in high–speed networks. *IEEE Journal on Selected Areas in Communications*, 9(7):968–981, September 1991.

[Heid88] P. Heidelberger. Discrete event simulations and parallel processing: statistical properties. *SIAM Journal on Scientific and Statistical Computing*, 9(6):1114–1132, November 1988.

[HN93] P. Heidelberger and D. M. Nicol. Conservative parallel simulation of continuous time Markov chains using uniformization. *IEEE Transactions on Parallel and Distributed Systems*, 4(8):906–921, August 1993.

[HS90] P. Heidelberger and H. Stone. Parallel trace-driven cache simulation by time partitioning. In *Proceedings of the 1990 Winter Simulation Conference*, pages 734–737, New Orleans, December 1990.

[Hui89] J. Y. Hui. Resource allocation for broadband networks. *IEEE Journal on Selected Areas in Communications*, 6(9):1598–1608, December 1989.

[Jef85] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[Lin93] Y-B. Lin. Parallel trace-driven simulation for packet loss in finite-buffered voice multiplexers. *Parallel Computing*, 19(2):219–228, February 1993.

[LL91] Y.-B. Lin and E.D. Lazowska. A time-division algorithm for parallel simulation. *ACM Transactions on Modeling and Computer Simulation*, 1(1):73–83, January 1991.

[LMSTW93] W. E. Leland, W. Willinger M. S. Taqqu, and D. V. Wilson. On the self-similar nature of ethernet traffic. In *Proceedings of the 1993 ACM SIGCOMM Conference*, pages 183–193, 1993.

[NF94] D. M. Nicol and R. M. Fujimoto. Parallel Simulation Today. *Annals of Operations Research*, to appear.

[NGLR92] D. Nicol, A. Greenberg, B. Lubachevsky, and S. Roy. Massively parallel algorithms for trace-driven cache simulation. In 6^{th} *Workshop on Parallel and Distributed Simulation*, volume 24, pages 3–11. SCS Simulation Series, Jan. 1992.

[PW89] S. Parekh and J. Walrand. A quick simulation method for excessive backlogs in networks of queues. *IEEE Transactions on Automatic Control*, 34(1):54–66, January 1989.

[SBW88] L. M. Sokol, D. P. Briscoe, and A. P. Wieland. MTW: a strategy for scheduling discrete simulation events for concurrent execution. In *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 19, pages 34–42. SCS Simulation Series, July 1988.

[WA92] J. J. Wang and M. Abrams. Approximate time–parallel simulation of queueing systems with losses. In *Proceedings of the 1992 Winter Simulation Conference*, pages 700–708, 1992.