Techniques for Efficient Cell-Level ATM Simulations

Mohit Aron*
Department of Computer Science
Rice University

Lawrence Brakmo
Western Research Laboratory
Compaq Computer Corporation

Abstract

This paper investigates the efficiency of a network simulator in simulating ATM networks with UBR service and compares it to the the efficiency of simulating conventional packet networks. A network simulator can be highly inefficient for cell-level ATM simulations as compared to analogous packet network simulations. We motivate and describe three techniques for improving the efficiency of ATM simulations. These techniques afford an increase in efficiency of 29-34% for ATM simulations on the x-sim network simulator.

1 Introduction

Analyzing new Internet technologies through simulation is critical to their testing and evaluation. This is mainly due to the cost effectiveness and greater control afforded by a simulated framework. It is imperative, thus, to efficiently simulate these technologies in order to study them. We define *simulator efficiency* as the ratio between the simulated time and the real time taken by a simulator. This paper focuses on improving the efficiency of network simulators for cell-level simulations of ATM technology.

ATM (Asynchronous Transfer Mode) technology is expected to gain widespread use both in wide-area networks and in high-speed local area networks (LANs). Therefore, it is important to study the interactions of existing Internet protocols with ATM. Our main intent was to study the performance of the TCP/IP protocol suite [12] over ATM as compared to conventional link level technologies such as Ethernet and FDDI (such networks are referred to as packet networks as opposed to ATM networks that use fixed sized cells).

For comparing the performance of two different link level technologies, it is important to simulate both on a single network simulator capable of incorporating different protocols. Additionally, it is important to simulate ATM networks at the granularity of an ATM cell for studying the effect of congestion conditions on switch-level algorithms and the corresponding impact on the performance of transport protocols like TCP. However, cell-level simulations with a faithful implementation of the ATM protocol can be very inefficient as compared to corresponding simulations involving packet network technologies. Our experiences with the x-sim[2] network simulator indicate that ATM simulations can take up to 40 times longer than those involving packet networks for the same simulated time and analogous network topologies.

Our study identified three principal factors responsible for simulator slowdown with ATM simulations: (1) The overhead imposed by data structures representing ATM cells, (2) Computation performed for the purpose of detecting corrupt AAL5[1] (ATM Adaptation Layer 5) frames, and (3) Significant increase in the number of events handled by the simulator. We propose simulator enhancements that address each of the above mentioned problems. First, we describe a low overhead ATM cell representation. Second, we provide an efficient method for detecting corrupt AAL5 frames. Finally, we propose improvements to the event management algorithms; specifically, we propose a Calendar Queue[3] based priority queue implementation enhanced with a technique called *Lateral Insertion*. While the latter is independent of the simulator's implementation of ATM, the first two proposals affect the implementation in a way that captures all the essential features of a faithful implementation, yet impose lesser computational overhead. Altogether, these improvements yield a 29-34% improvement in the simulator efficiency. While our study is based on the framework of the x-sim network simulator, the results are applicable to any general network simulator.

The remainder of this paper is organized as follows. In Section 2 we provide background on ATM that is relevant to the rest of the paper. Section 3 describes our simulation environment and Section 4 analyzes the reasons for decrease in simulator efficiency and motivates and presents techniques for improvement. Section 5 presents performance results and we summarize

*This work was done in 1997 while the author was a summer intern at DEC-WRL.

[1] The AAL5 protocol performs segmentation/reassembly of IP packets to/from ATM cells.

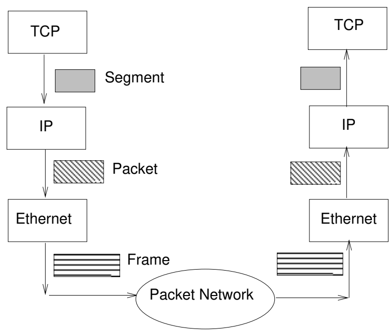

Figure 1: **Packet protocol stack**

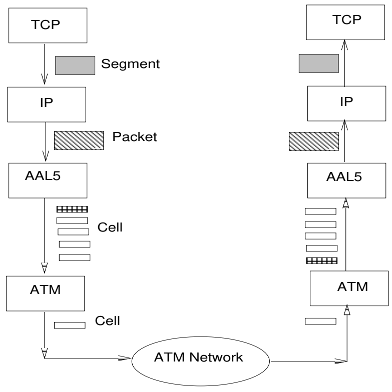

Figure 2: **ATM protocol stack**

our conclusions in Section 6.

2 Background

This section presents some background information on packet and ATM networks. We compare the protocol stacks used for the two network technologies. Specifically, we only consider the best-effort service provided by ATM[2] and packet networks. We also discuss the format of an ATM cell and the purpose of individual fields in the cell header.

2.1 Packet and ATM protocol stacks

Figures 1 and 2 show typical protocol stacks for two hosts using a TCP connection over packet and ATM[3] networks respectively. The sender's TCP collects the data handed down to it by the application in a TCP segment that is passed down to the IP layer. An IP header is prepended and the resulting IP packet is handed to the lower layer. In case of packet networks, the data link layer (e.g. Ethernet) attaches its own header to the IP packet and pushes the resulting frame onto the network. When this frame is received at the destination host, it undergoes the reverse process where each protocol strips off its header and passes it above until the packet is delivered to the application.

For an ATM protocol stack, the IP packet is handed to the AAL5 layer that attaches a header and a trailer and pads the packet so as to make its size a multiple of 48 bytes. The resulting AAL5 frame is broken up into 48 byte cells (this process is called *segmentation*) and passed down to the ATM layer below. The last cell from an AAL5 frame is specially marked (shown as the shaded cell in Figure 2) so as to enable the receiving end to *reassemble* the cells back into the AAL5 frame. These cells are transmitted on the network after a 5 byte header is prepended to them by the ATM layer. The reverse process is performed at the receiving end and cells are reassembled into AAL5 frames by the AAL5 layer. These frames are passed up as IP packets after a CRC check is performed over them.

2.2 ATM Cell Format

Figure 3 shows the format of a 53 byte ATM cell[4]; the size in bits of each field is shown above it. The first 5 bytes contain the ATM cell header and the last 48 bytes contain data (or payload). The VCI (Virtual Channel Identifier) and VPI (Virtual Path Identifier) identify a virtual connection at a network host/switch while the PTI (Payload Type Identifier) is used for marking the last cell of an AAL5 frame. The HEC (Header Error Check) field stores a CRC checksum that is computed over the ATM header. The GFC (Generic Flow Control) and CLP (Cell Loss Priority) are not used by the UBR service in ATM networks.

3 Simulation Environment

This section describes the simulation environment used in our study. We describe the simulator used, our implementation of ATM on this simulator and finally the hardware platform used for obtaining the simulation results.

The simulations were performed using the x-sim[2] network simulator, which is based on the x-kernel[6]. x-sim is an execution-driven network simulator where the actions of network protocols are simulated by executing its actual protocol implementation code rather than an abstract behavioral model of the protocol. x-sim supports multiple hosts, each running a full protocol stack, and several abstract link behaviors (point-to-point and ethernet links). The IP routers are modeled as network nodes that support a particular queuing discipline (e.g. FCFS). Host and router computa-

[2] Best-effort service in ATM is called UBR[8] (unspecified bit rate).

[3] On a real host, the AAL5 and ATM layers would be implemented in hardware.

[4] Our simulator only supports the UNI (user-network interface) format for the ATM cell; the NNI (network-network interface) format can be incorporated by a simple extension.

4	8	16	3	1	8	384 (48 bytes)
GFC	VPI	VCI	PTI	CLP	HEC (CRC–8)	Payload

Figure 3: **ATM Cell format**

tion is assumed to have zero overhead. The simulator clock has a granularity of $1\mu s$.

We added support in the simulator for ATM. Specifically, this involved adding the AAL5 and ATM protocols that could be used in the protocol stacks at hosts. Additionally, point-to-point links and switches supporting ATM were also added. The ATM switches are similar to IP routers, except they perform switching based on the VPI and VCI fields of the ATM cell header. For simplicity, our switch implementations do not modify the VCI and VPI fields of an ATM cell header; the tuple *(VPI, VCI)* uniquely identifies a virtual circuit in the simulator. This can, however, be easily changed by extending the simulator.

All simulation results reported in this paper were performed on Digital Unix (version 4.0) running on a 333MHz Digital's AlphaStation 600 processor equipped with 640MB of physical memory. The size of both the on-chip data and instruction caches was 8KB, while the cache block size was 32 bytes. The on-chip second level cache was 96KB and had a cache block size of 64 bytes.

4 Techniques for Improving Efficiency

In this section, we identify three of the main factors responsible for decrease in simulator efficiency while simulating ATM networks and propose techniques for improvement. In the next section, we'll show that as a result of these techniques, the efficiency of the simulator increases by 29-34%.

4.1 ATM Cell representation

Efficient network simulators use a generic data type to represent *protocol data units* (PDU) at each level in the host protocol stacks. Such data structures support buffer aggregation operations to enable efficient removal/addition of protocol headers without touching all the data as a packet moves up/down a protocol stack. *x*-sim uses the *x*-kernel messages[5][6] as the abstract data type to represent the PDU at each level in the protocol stack. In particular, IP packets, AAL5 frames and ATM cells are represented by *x*-kernel messages. While using a generic data type at all levels is quite efficient in a packet network, our results show that the process of segmentation and reassembly in ATM imposes high overhead on the *x*-kernel message operations. This overhead is a direct result of the following operations performed for each cell when maintained in the format shown in Figure 3:

1. Allocating and initializing the data structure for representing a cell.
2. Logically breaking off 48 bytes from the AAL5 frame and adding them to the data structure representing a cell. Although this is accomplished without any data copying, there is overhead associated with manipulation of the data structures.
3. Recombining the 48 bytes of data contained in the data structure with a partially formed AAL5 frame.
4. Deallocating the memory associated with the data structure.

Based on the observation that only the ATM cell header is needed for switching operations in the simulator, we modified the implementation to represent an ATM cell with the 8-byte entity shown in Figure 4. This implementation does not carry the 48 byte payload; the latter is carried implicitly, i.e., the new representation *behaves* as if it has a size of 53 bytes. We now describe this format in detail and show that it is capable of dramatically reducing computational and memory overheads.

While the VPI, VCI, GFC, CLP and PTI fields have the same functions and sizes as in the actual 5-byte ATM header, the remaining fields in Figure 4 are unique to our implementation and are not part of the ATM header found in real networks. The *Node Handle* field is used for accessing state information of the next-hop node required for executing events in the simulator. In the original information, this information was stored as an *attribute* with the *x*-kernel message representing a cell. The *Frame ID* field is used to associate ATM cells with AAL5 frames in the simulator.

Our new implementation no longer breaks an AAL5 frame into cells. Rather, the 8-byte entity shown in Figure 4 is allocated for each ATM cell that *would have* been produced from an AAL5 frame. Every AAL5 frame is allotted a new *Frame ID* that is stored in the corresponding field of each cell associated with it. As in the original implementation, the last cell of a frame is marked by setting its PTI field to 1.

[5] The *x*-kernel messages are tree-based data structures that implement buffer aggregates and provide efficient operations for logically joining, clipping and splitting aggregates.

8	16	16	16	4	1	3
VPI	VCI	Frame ID	Node Handle	GFC	CLP	PTI

Figure 4: **New Representation of ATM Cell**

As mentioned in Section 3, the VPI and VCI fields uniquely identify the source and destination of an ATM cell in our implementation. All cells associated with an AAL5 frame, therefore, have the same VPI and VCI fields. Thus, the tuple *(VPI, VCI, Frame ID)* uniquely identifies an AAL5 frame in the simulator and can be accessed by means of a hash lookup. AAL5 frames are stored by the simulator until accessed when the last cell associated with the frame reaches the destination host. The new cell representation affords the following advantages:

1. Overhead associated with segmentation and reassembly of AAL5 frames is avoided.
2. It reduces memory requirements – 8 bytes are required as opposed to 37 bytes required earlier with x-kernel messages (32 bytes for an empty x-kernel message, 5 bytes for ATM header; the payload was shared with the AAL5 frame).
3. It can fit in the 32-byte cache block on the alpha architecture. The earlier representation required at least two cache blocks with a higher probability of spanning more blocks owing to the fragmented nature of the tree-based x-kernel message data structure.
4. The individual fields in the 8 byte representation can be efficiently accessed on the 64-bit alpha architecture.

The new cell representation *behaves* as a 53-byte ATM cell in that transmission delays at network links are computed accordingly. Moreover, it was possible to transparently incorporate this implementation in x-sim without rewriting all code that worked with the original x-kernel messages. This was achieved by defining a dummy 8th byte in the data type definition for the x-kernel message structure that was always set to the value 255. As the 3-bit PTI field in the new representation only takes the values 0 and 1, the 8th byte can never be 255. Therefore, the simulator code can distinguish x-kernel messages from the 8-byte cell representation permitting the latter to be type casted into C pointers representing the former.

We mentioned in Section 3 that the tuple *(VPI, VCI)* uniquely identifies a virtual circuit in our simulator as the switches do not modify these fields in cell headers. If the implementation were to be extended to permit switches to modify these fields, a virtual circuit would be uniquely identified by the tuple *(VPI, VCI, switch)*. As the switches are identified by the *Node Handle* field in our new cell representation, AAL5 frames shall be uniquely identified by the tuple *(VPI, VCI, Frame ID, Node Handle)* in the extended simulator.

4.2 Detecting Corrupt Cells and Frames

Another source of computational overhead are the CRC computations performed in the simulator both for individual ATM cells as well as for AAL5 frames. An 8-bit CRC checksum is computed over the ATM cell header as it traverses through each switch in the network and serves to detect corrupt cell headers. In addition, a 32-bit CRC is computed for each AAL5 frame at both the source and the destination and serves to discard these frames at the latter if any associated cells are dropped/corrupted.

Our new implementation eliminates the overhead associated with CRC computations. For this purpose, three attributes *numCells, numInactive* and *dropFlag* are associated with each AAL5 frame. *numCells* contains the total number of cells associated with the frame, and *numInactive* reflects the total number of cells that are inactive, i.e., have either reached the destination or have been dropped. The attribute *dropFlag* is set as soon as any cell associated with an AAL5 frame is dropped, in which case the AAL5 frame is considered corrupted. If all cells associated with the frame are delivered to the destination, the AAL5 frame is passed as an IP packet to the IP layer. Otherwise, as soon as *numInactive* equals *numCells*, the frame is discarded and the memory associated with it is freed.

Our simulator does not model corruption of individual cells, but it can be easily extended to do so. This can be achieved by setting the *dropFlag* attribute on the associated AAL5 frame, ensuring thereby that the frame shall be ultimately discarded. Discarding the corrupt cell in a switch depends upon whether its header is corrupted. If it is, *numInactive* will be incremented and the cell discarded. Otherwise, the cell is delivered to the next-hop switch just as for an uncorrupted cell.

While our proposed techniques in Section 4.1 and Section 4.2 deviate from a faithful implementation of AAL5 and ATM in the simulator, they afford the advantage of significantly lower computational overhead.

4.3 Event Handling

Our study identified the increase in simulator events with ATM to be another cause for decrease in its efficiency. As x-sim is an event driven simulator, the transmission and propagation delays over network links are modeled by scheduling events. If each IP packet generates N ATM cells, then the number of events in ATM simulations is N times larger than those in packet network simulations. Therefore, it is imperative to have efficient event management in the simulator.

The following basic operations are typically provided for event management in a simulator:

- *schedule_event(ev, tm)*: schedule an event *ev* to fire at time *tm*.
- *get_next_event()*: provides the earliest event that has been scheduled and increments the simulated time to the time when the event needs to be fired.

The above two operations can be modeled by a priority queue algorithm [4]. The priority queue implementation used in the x-sim simulator was based on *Timing Wheels*[13]. Section 4.3.1 describes this implementation and the problems associated with it. In Section 4.3.2, we discuss a *Calendar Queue*[3] based priority queue and in Section 4.3.3, we present our improvements to the Calendar Queue algorithm.

4.3.1 Timing Wheels

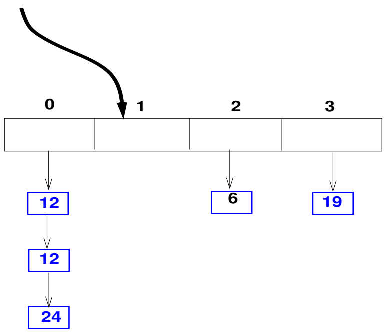

Figure 5: **Timing Wheels**

The timing wheels implementation in x-sim consists of an array of slots (technically referred to as buckets), each containing a list of events ordered by the time at which they are to fire. Figure 5 shows a snapshot of timing wheels in x-sim. The pointer shown indicates the value *(curr_time mod num_buckets)*, where *curr_time* is the current value of simulated time and *num_buckets* is the number of buckets in the array. The operations *schedule_event* and *get_next_event* are implemented as follows:

- *schedule_event(ev, tm)*: insert *ev* in sorted order in the list at the bucket given by the hash function *(tm mod num_buckets)*.
- *get_next_event*: keep advancing *curr_time* by 1 unit until an event that needs to be fired at *curr_time* is found in bucket *(curr_time mod num_buckets)*[6].

In order to prevent the lengths of the event lists in buckets from becoming large, the number of buckets is dynamically varied depending on the number of events scheduled in the simulator. Our study shows that this implementation is inefficient in locating events when the average time separation between consecutive events is large. Consider the scenario in Figure 5 when the value of *curr_time* is 1 unit. Even though nearly all the buckets contain events, *get_next_event* will scan all the array slots before picking the next event to fire (at time 6). The magnitude of the problem increases with the number of events as the number of buckets is varied proportionally. We quantify the performance impact of using timing wheels with ATM simulations in Section 5.

4.3.2 Calendar Queues

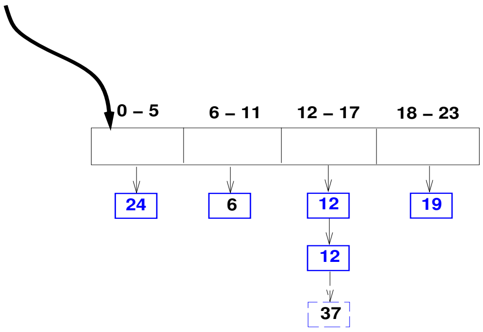

Figure 6: **Calendar Queue**

Calendar queues[3] aim towards providing O(1) *schedule_event* and *get_next_event* operations by associating a width with each bucket[7]. The *bucket_width* is computed as the average separation between the times at which consecutive events fire. The *schedule_event* and *get_next_event* operations are implemented as follows:

- *schedule_event(ev, tm)*: insert *ev* in sorted order in the list at the bucket given by the hash function *((tm div bucket_width) mod num_buckets)*.
- *get_next_event*: keep advancing *curr_time* by *bucket_width* units until an event is found in bucket *((curr_time div bucket_width) mod num_buckets)* that is to be fired at a time earlier than $curr\_time + bucket\_width$.

[6] Scanning the array multiple times is avoided by firing the event with the earliest time recorded during the first scan.

[7] The timing wheels are a special case of calendar queues where the bucket_width is 1.

Figure 6 shows the calendar queue snapshot (*bucket_width* is 6) that corresponds to the timing wheel shown in Figure 5. As for the timing wheels, the number of buckets is maintained to be proportional to the number of events, making the expected time for execution of *schedule_event* to be O(1). Similarly, as the *bucket_width* reflects the average time separation between events, the expected time for execution of *get_next_event* is also O(1). Simulation results indicate that less than 0.001% of the calendar queue operations result in a recomputation of the *bucket_width* and the number of buckets.

4.3.3 Calendar Queues with Lateral Insertion

We now propose a modification to the Calendar Queues aimed at increasing the efficiency of handling events scheduled to fire at the same simulation time instants. Such events are widespread in network simulations owing to the simultaneous packet receptions or transmissions at the different network nodes. We discuss the effect of these events on calendar queues and describe *Lateral Insertion*, a technique that efficiently handles such events.

All events scheduled to fire at the same time are inserted in the same bucket in a calendar queue. Additionally, for FIFO execution, each such event is inserted at the end of the chain of such previously scheduled events. These chains increase with the size of the simulated network topology and affect the speed of the *schedule_event* operation. This is illustrated in Figure 6 where two events are scheduled to occur at time 12. Even though such events remain attached to the same bucket, they affect the number of buckets, which is proportional to the number of events. This in turn affects the speed of the *get_next_event* operation when it unnecessarily scans empty buckets.

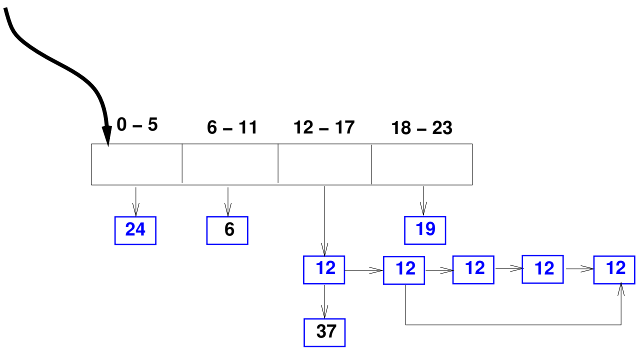

Figure 7: **Calendar Queue with Lateral Insertion**

With *Lateral Insertion*, an extra pointer is maintained with each event and is used for inserting simultaneously occurring events *laterally* in a chain. This is depicted in Figure 7 where multiple events scheduled to occur at time 12 are attached in a *lateral* chain. The free pointer (used originally to maintain the list) in the second event in the chain points to the tail, thus enabling efficient insertion for FIFO execution. The number of buckets is maintained to be proportional to the number of *distinct* event execution times, rather than the number of scheduled events. Thus, even though the number of events in Figure 7 is more than double the number of buckets, the latter is not increased.

5 Performance Evaluation

In this section we quantify the effects of the techniques proposed in Section 4. We first present a brief description of the profiling tools used to measure the performance of the simulator. Then, we describe the network configuration used for performing our simulations. Next, we present results showing the impact of our improvements on the simulator. This is followed by a discussion of the results.

5.1 Profiling Tools

We used the pixie[7] and prof[10] tools available in Digital Unix to profile the simulator. The pixie utility instruments an executable with additional code that performs basic-block counting when the program is executed. The basic-block counts are written to a file that is used by the prof utility for the purpose of producing a listing of the profile that includes information about the CPU cycles spent by the program in each subroutine.

5.2 Network Configuration

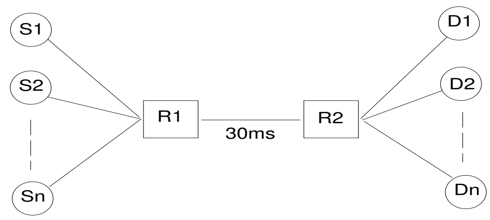

Figure 8: **Network configuration for simulations**

The basic network configuration that we used for our simulations is shown in Figure 8. This configuration was used both for ATM as well as packet network simulations. In case of the packet network, the interior nodes R1 and R2 are IP routers, while for ATM they are ATM switches. n senders (S1, S2, ... Sn) use the 4.4BSD-Lite version of TCP to send data to n destinations (D1, D2, ... Dn) across a bottleneck link that connects R1 and R2. The round trip propagation delay is 60ms and all links have the same bandwidth. The TCP segment size was chosen based on the default MTU of IP over ATM, which is 9180 bytes[1].

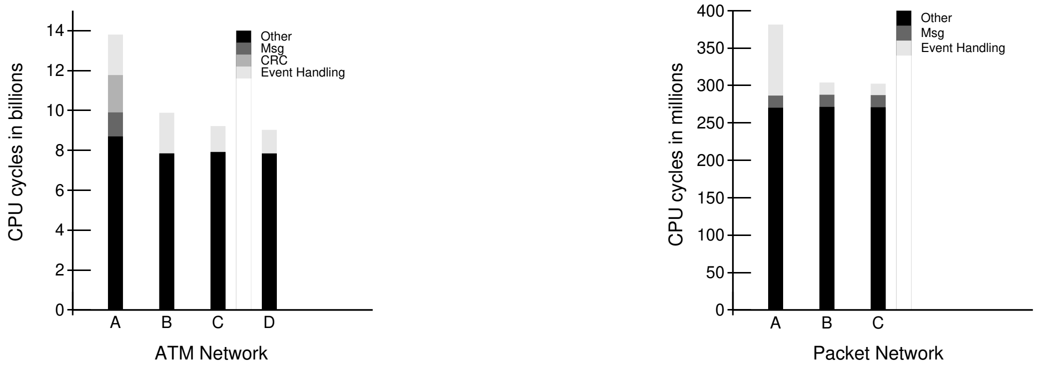

Figure 9: **Bulk Transfer Experiment with 10.09Mb/s links**

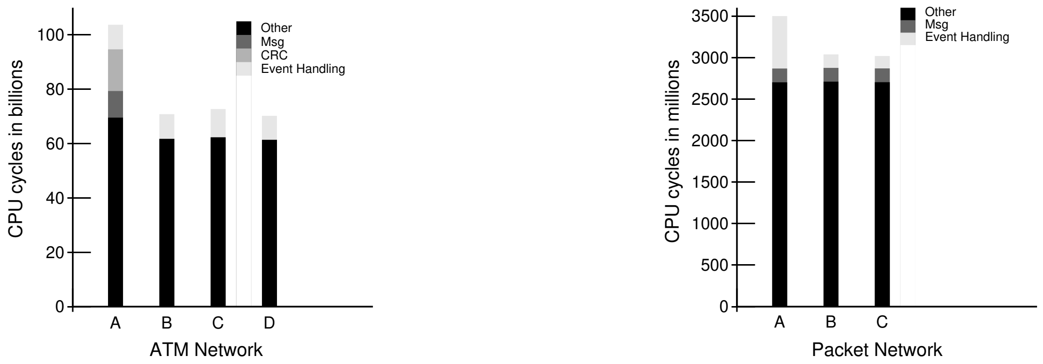

Figure 10: **Bulk Transfer Experiment with 141.33Mb/s links**

The parameter n, the link bandwidth and the applications running on individual hosts were varied with different experiments and shall be described in Section 5.3.

Our network configuration resembles those used in other studies on TCP dynamics[11, 9] but incorporates a high-delay link typical of wide-area networks. The switches used in the simulations are FCFS switches with output buffering and the drop-tail discipline for dropping packets/cells. Commercial ATM switches now support enhancements like Early Packet Discard [11] and per-VC queuing to enhance TCP performance. These enhancements, though supported, are outside the scope of this paper and are not used in the simulation results presented.

5.3 Results

In this subsection, we present results showing the impact of our improvements on the efficiency of the simulator. Both ATM and packet simulations are considered.

5.3.1 Bulk Transfer Experiments

We first study the simulator efficiency with bulk TCP transfers. The network configuration in Figure 8 was configured with 8 TCP senders, each sending 60 MB of data to 8 TCP receivers (i.e., the parameter n is 8). Two different values of link bandwidths, 10.09 Mb/s and 143.33 Mb/s, were chosen[8] and are considered to be representative of typical low and high bandwidth links found today. For simulations involving the higher link bandwidth, the TCP advertised window size was set to 512 KB to accommodate the larger bandwidth-delay product path. The simulation time was restricted to 15 secs.

Figure 9 and Figure 10 show the distribution of CPU cycles for link bandwidths of 10.09 Mb/s and 141.33 Mb/s, respectively. Each bar shows distribution of CPU cycles spent in Event Handling, CRC computations, managing *x*-kernel messages (shown as *Msg*) and for the rest of the program (shown as *Other*).

The graph for the ATM network shows four distributions, each quantifying the effect of successive improvements: (A) distribution for the original faithful ATM implementation with Timing Wheels, (B) the modified implementation with new representation of ATM cell and that avoids the CRC computational overhead, (C) with calendar queues for event management, and (D) with *Lateral Insertion* applied to Calendar Queues.

The corresponding graph for packet network shows

[8] The values were chosen so as to make the transmission time for an ATM cell to be an integral number of microseconds.

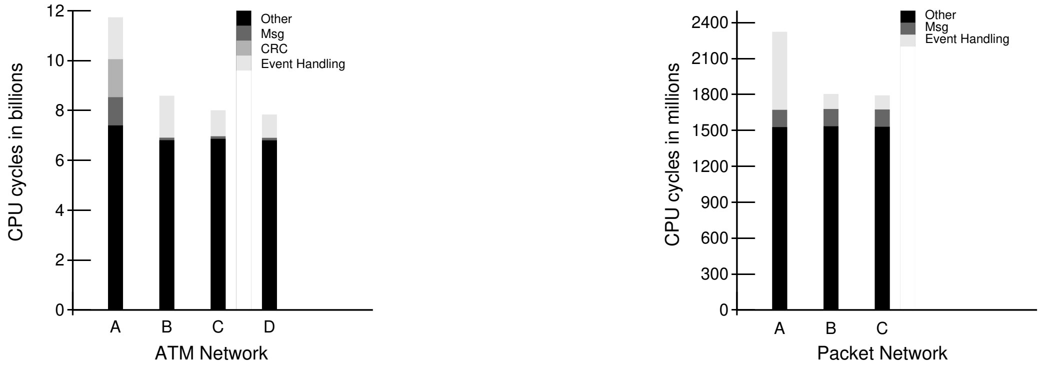

Figure 11: **Background Traffic Experiment with 10.09Mb/s links**

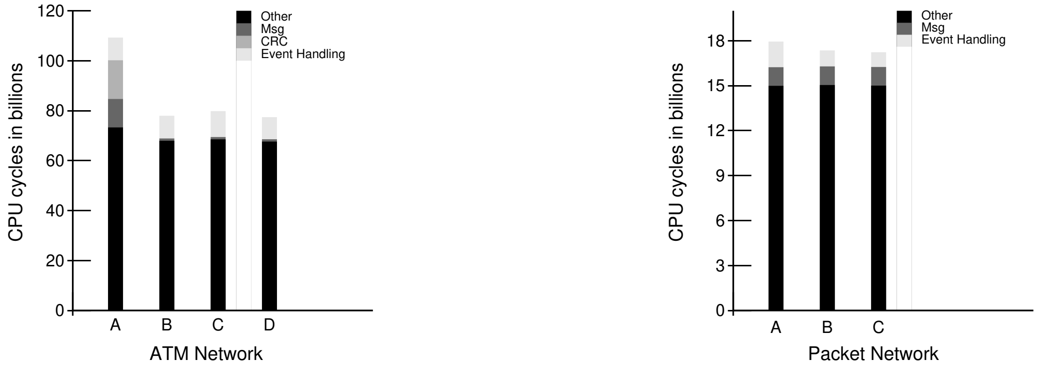

Figure 12: **Background Traffic Experiment with 141.33Mb/s links**

three distributions: (A) distribution for the original faithful implementation with Timing Wheels, (B) with calendar queues for event management, and (C) with *Lateral Insertion* applied to Calendar Queues.

Figure 9 and Figure 10 show a significant increase in the efficiency of the simulator both for ATM as well as for packet networks. For the configuration with 10.09Mb/s links, the efficiency increases by 34% for ATM and by 20% for the packet simulation. The corresponding numbers for the 141.33Mb/s link configuration are 32% and 13% respectively.

The *x*-kernel message overhead decreases by 99% with our improved ATM implementation. Upon replacing Timing Wheels with Calendar Queues, the event management overhead decreases by 36% and 82% respectively for ATM and packet simulations with 10.09 Mb/s links. With 141.33 Mb/s links, this overhead *increases* by 14% for ATM, while it decreases by 74% for packet network simulation. The increase with ATM is attributed to the higher overhead of computing *bucket_width* in Calendar Queues, while the small separation (3 μs transmission time for 53 byte ATM cells) between events with 141.33 Mb/s bandwidth results in efficient performance with Timing Wheels. The *Lateral Insertion* technique affords a 9% and 7% improvement, respectively, with ATM and packet simulations with 10.09 Mb/s links. The corresponding numbers for the 141.33Mb/s links are 15% and 8%.

The results for ATM also indicate a reduction in time spent in the *other* parts of the program by 10% with the improved implementation. This reflects the better cache locality afforded by our techniques.

5.3.2 Background Traffic Experiments

Our next set of experiments studies simulator efficiency with a single bulk TCP transfer performed in the presence of background traffic generated using *tcplib*[5]. The network topology in Figure 8 was configured with one TCP sender and receiver. The parameter n was chosen so as to have the remaining hosts generate sufficient background traffic that occupies 80% of the bottleneck link. We again used link bandwidths 10.09 Mb/s and 141.33 Mb/s; for the former the parameter n was set to 10 while for the latter it was set to 80. As for the bulk transfer experiments, the simulation time was restricted to 15 secs.

The results are presented in Figure 11 and Figure 12 where the distributions have the same meaning as defined in Section 5.3.1. For the configuration with 10.09Mb/s links, the simulator efficiency increased by 33% for ATM network and 23% for packet network. For the configuration with 141.33Mb/s, the respective improvements were 29% and 4%.

As a result of the improved ATM implementation, the x-kernel message overhead decreases by 91%. For the 10.09Mb/s configuration, replacing Timing Wheels by Calendar Queues reduces the event handling overhead by 44% for ATM and by 80% for the packet simulation. For the 141.33Mb/s simulations, the event handling overhead increases by 14% for ATM while it decreases by 37% for packet simulation. In the 10.09Mb/s simulations, The *Lateral Insertion* technique reduces the event handling overhead by 10% for ATM and by 6% for packet simulation. The corresponding numbers for the simulations with 141.33Mb/s links are 14% and 8%.

5.4 Discussion

The results from Section 5.3 show that our techniques increase the efficiency of the ATM simulations by 29-34% and the packet simulations by 4-23%. For ATM simulations, the computational overhead associated with CRC is eliminated and that with x-kernel messages is reduced by more than 90%. Calendar queues are effective over timing wheels both for packet simulations as well as for low bandwidth ATM simulations. The *Lateral Insertion* affords a consistent 15% and 8% improvement in performance for ATM and packet networks respectively.

Despite our enhancements, the ATM simulations remain significantly slower than the corresponding packet network simulations. This is attributed to the significantly larger number of events in ATM networks. While the processing involved with each event is comparable to that of a packet network (e.g., switching an ATM cell compared to switching an IP packet), the larger number of events inherently pose a limit to improvement in simulator efficiency.

6 Conclusions

This paper introduces several techniques to improve the efficiency of a network simulator. First, we have proposed a low overhead ATM cell representation in the simulator designed for enhancing cache locality. Second, we demonstrated techniques for efficiently detecting corrupt AAL5 frames and avoiding CRC computational overheads. Finally, we presented better event management algorithms for efficiently handling the large number of events in ATM simulations. Our techniques improve simulator efficiency by 29-34% in ATM and by 4-23% in packet network simulations. While we showed the impact of our techniques on the x-sim simulator, they can be deployed in other network simulators as well.

References

[1] R. Atkinson. Default IP MTU for use over ATM AAL5. *IETF Internet Draft*, Feb. 1994. Available by anonymous ftp from ds.internic.net:internet-drafts/internet-drafts/draft-ietf-atm-mtu-07.txt.

[2] L. S. Brakmo and L. L. Peterson. Experiences with Network Simulation. In *Proceedings of the ACM SIGMETRICS '96 Conference*, 1996.

[3] R. Brown. Calendar queues: A fast O(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, Oct. 1988.

[4] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[5] P. Danzig and S. Jamin. tcplib: A Library of TCP Internetwork Characteristics. Technical Report CS-SYS-91-495, Computer Science Department, USC, 1991.

[6] N. C. Hutchinson and L. L. Peterson. The x-kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[7] MIPS Computer Systems, Sunnyvale, CA. *UMIPS-V Reference Manual (pixie and pixstats)*, 1990.

[8] T. J. Ott and N. Aggarwal. TCP over ATM: ABR or UBR? In *Proceedings of the ACM SIGMETRICS '97 Conference*, Seattle, WA, June 1997.

[9] M. Perloff and K. Reiss. Improvements to TCP Performance in High-Speed ATM Networks. *Communications of the ACM*, 38(2):90–100, Feb. 1995.

[10] *prof.* Digital Unix man page.

[11] A. Romanow and S. Floyd. The Dynamics of TCP Traffic over ATM Networks. *IEEE Journal on Selected Areas in Communication*, 13(4), May 1995.

[12] W. Stevens. *TCP/IP Illustrated Volume 1 : The Protocols.* Addison-Wesley, Reading, MA, 1994.

[13] G. Varghese and A. Lauck. Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 171–180, Nov. 1987.