Techniques for Efficient Cell-Level ATM Simulations

Mohit Aron*
Department of Computer Science
Rice University

Lawrence Brakmo
Western Research Laboratory
Digital Equipment Corporation

Technical Report
Department of Computer Science
Rice University

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.

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 focusses 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 (in the rest of the paper, we'll refer to networks using these technologies as packet networks as opposed to ATM networks which 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. Thus, the simulator should be general enough to allow easy incorporation of different network protocols. Further, it is important to simulate ATM networks at the granularity of an ATM cell for studying

*Most of this work was done while the author was doing a summer internship at DEC–WRL 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 on a general network simulator can be very inefficient as compared to corresponding simulations involving packet network technologies. Our study is based on our experiences with ATM simulations on the *x*-sim[2] network simulator. We found that the simulations involving ATM networks can take upto 40 times longer than those involving packet networks for the same simulated time and analogous network topologies.

As a result of our study, we identified three of the main causes responsible for the simulator slowdown on ATM simulations. The first was the huge overhead imposed by the data structures for representing ATM cells in the simulator. The second was a lot of unnecessary computation being performed for detecting corrupt AAL5[1] frames in the simulator. The third was a steep increase in the number of events being handled by the simulator. Towards this end, we propose improvements which address each of the above mentioned causes for simulator slowdown. First, we propose a low overhead cell representation in the simulator. Second, we propose an efficient method for detecting corrupt AAL5 frames in the simulator. Finally, we propose improvements to the event management algorithms in the simulator. Specifically, we propose a Calendar Queue[3] based priority queue implementation enhanced with a technique called *Lateral Insertion*. Our first two proposals result in a slight variation to the implementation of the ATM protocol in the simulator, that captures all the essential features of a faithful implementation, yet imposes lesser computational overhead. The third improvement is independent of the implementation of ATM in the simulator. 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 give some background on ATM that is relevant to the rest fo the paper. Section 3 describes our simulation environment. Section 4 analyzes the reasons for decrease in simulator efficiency and motivates and presents the various improvements we propose to increase efficiency. Section 5 presents some performance results showing the impact of our proposed improvements. Finally, we summarize our conclusions in Section 6.

2 Background

This section presents a comparison of the protocol stacks on hosts when TCP is used over ATM and packet networks. Specifically, we only consider the best-effort service provided by ATM and packet networks. In ATM, this type of service is called UBR

[1] The AAL5 layer sits below IP in the protocol stack and is responsible for segmenting IP packets into ATM cells and reassembling them back into IP packets.

(unspecified bit rate) [8].

Figures 1 and 2 show typical protocol stacks for two hosts using a TCP connection over packet and ATM[2] networks respectively. TCP collects the data handed down to it by the application in a TCP segment (which also includes the protocol header). This segment is passed down to the IP layer which prepends its header and passes the resulting IP packet down 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 to the protocol above until the packet is delivered to the application.

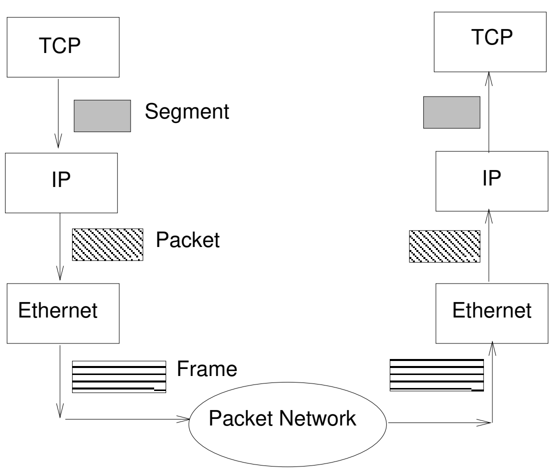

Figure 1: **Packet protocol stack**

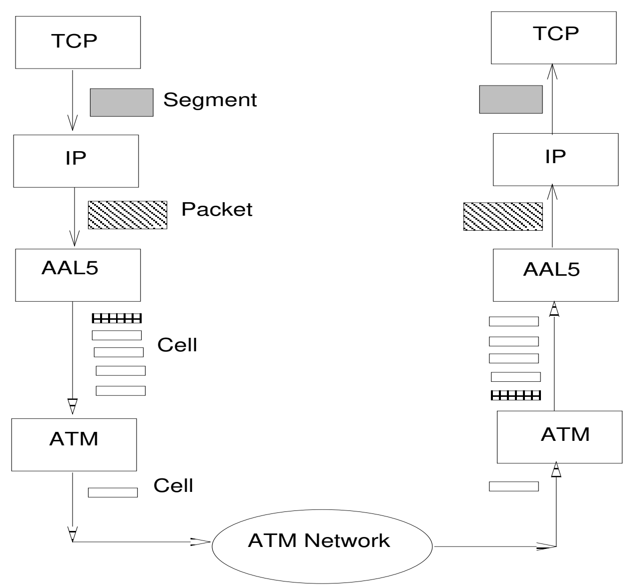

Figure 2: **ATM protocol stack**

In an ATM protocol stack, when IP hands down a packet to AAL5, the AAL5 layer 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*) which are 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 that the receiving end could tell when all the cells corresponding to an AAL5 frame have been received[3]. The ATM layer prepends a 5 byte header to each cell and pushes the resulting 53 byte ATM cells onto the network. On the receiving host, the ATM layer strips off the 5 byte header from each cell and passes it upto AAL5. On receiving the last cell (which was specially marked), AAL5 performs the process of *reassembly* wherein the cells are combined into the original AAL5 frame. Finally a CRC check is done on this

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

[3] The AAL5 layer actually tells the ATM layer to mark the last cell as the mark is stored in the ATM header.

frame and then it is passed upto IP as an IP packet.

3 Simulation Environment

This section describes the simulation environment. The simulations were done using the x-sim network simulator [2], 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 behaviours (point-to-point and ethernet links). The routers are modelled as a network node that supports a particular queuing discipline (e.g. FCFS). Host and router computation 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 (ATM Adaptation Layer 5) 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 to the simulator. The ATM switches are very similar to the routers mentioned earlier, except that they perform switching based on the VPI and VCI fields of the ATM cell header as opposed to the IP addresses in IP packets. Currently, the ATM switches do not modify the VPI and VCI fields of the cell header and the combination of VPI and VCI uniquely identify the virtual circuit in the network. However, there is nothing inherent in the simulator that prevents the switches from changing the VPI and the VCI fields.

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

4 Techniques for Improving Efficiency

In this section, we analyze three of the main reasons for decrease in simulator efficiency while simulating ATM networks and propose techniques to counter them. In the next section, we'll show that as a result of our improvements, the efficiency of the simulator increases by 29-34%.

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

Figure 3: **ATM Cell format**

4.1 ATM Cell representation

Figure 3 shows the format of a 53 byte ATM cell[4]. The size of each field in bits is shown above the field. 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. 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.

Network simulators come in two different flavors with respect to their implementation of *protocol data units* (PDU) at each level in the host protocol stack. The first kind of network simulators implement the addition/removal of packet headers by copying all the data into a larger/smaller buffer. Such simulators are extremely inefficient as all the data in a packet is touched at each transition between protocol layers. The second and more efficient kind of network simulators use some 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. Our results show that the process of segmentation and reassembly in ATM networks can also impose a high overhead on operations involved with maintaining such data structures.

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, the IP packet and the ATM cell are also represented by the *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:

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

[4] Our simulator currently uses only the UNI (user-network interface) format for the ATM cell. The NNI (network-network interface) format is not yet supported but the simulator can be easily extended to support it.

[5] The *x*-kernel messages are tree-based data structures that support buffer aggregation. Specifically, they provide efficient operations for logically joining one or more buffers into an aggregate, clipping data from one end of the aggregate, splitting an aggregate into separate buffers etc.

structure representing a cell. It is to be noted that the x-kernel message operations do not actually involve any data copying – rather a reference counting scheme is used to share the same data buffer between two x-kernel messages.

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.

In the wake of the knowledge that the only aggregation operation required on a cell is the prepending of a 5 byte ATM header, and that the ATM switches only read/modify this 5 byte header, the computational as well as memory overhead imposed by the above operations can be reduced dramatically. To achieve this, we chose to represent an ATM cell in the simulator by the 8 byte entity shown in Figure 4. The size in bits taken by each field is indicated above the corresponding field. While the new representation carries all the information necessary for switching in the original cell, it doesn't carry any data (or payload). Rather, the payload is carried *implicitly* – that is, the new representation *behaves* as if it has a size of 53 bytes. We'll explain this at length later in this Section.

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

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

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 are unique to our implementation and are not part of the ATM header found in real networks.

In the original implementation, the switches look up the VCI and VPI fields and compute a handle to the next network node to which the cell is to be delivered. This handle is stored as an attribute associated with the x-kernel message. An event is then scheduled in the simulator to deliver the cell to the next switch/host (nexthop node) on the path. When the event actually starts, this attribute is looked up to access the state information of the nexthop node. Due to the new cell representation, this is now stored as the *Node Handle* in the cell.

Instead of chopping up the data in an AAL5 frame into cells, AAL5 now allocates a new 8 byte entity to represent each cell thus keeping the original AAL5 frame intact. Every AAL5 frame is allotted a new *Frame ID* and is stored in each cell associated with the frame. Thus each cell associated with a particular AAL5 frame has the same value of Frame ID. To mark the last cell of an AAL5 frame, the PTI field is set to 1, otherwise it is cleared (i.e. its value is 0). The ATM layer fills the VCI and the VPI fields of the cell.

As the combination of VPI and VCI uniquely identify the source and destination of a cell in our implementation, thus all cells associated with a particular AAL5 frame also have the same VPI and VCI fields. The combination of VPI, VCI and Frame ID uniquely identify an AAL5 frame in the network. A mapping from *(VPI, VCI, Frame ID)* to the actual AAL5 frame is added to a global hash table. Thus at any node in the simulator, the first 5 bytes of the new representation (containing VPI, VCI and Frame ID) can be used to perform a hash table lookup to access the original AAL5 frame. The original AAL5 frame is stored by the simulator until all cells associated with it are delivered to the destination.

To facilitate delivery, three attributes *numCells, numInactive and dropFlag* are associated with an AAL5 frame. *numCells* contains total number of cells that are associated with this frame. *numInactive* is the total number of cells that are inactive i.e. have either been received by the destination or have been dropped at some switch. *dropFlag* indicates if any cell has been corrupted or dropped. If all cells are successfully received at the destination, the stored AAL5 frame is delivered as an IP packet to the IP layer above. The mapping in the global hash table that maps *(VPI, VCI, Frame ID)* to the frame is then removed. As the Frame ID has a size of 16 bits, the simulator can thus handle upto 65535 active[6] AAL5 frames per host[7].

As far as simulation of network links is concerned, the new representation *behaves* exactly as if it had a size of 53 bytes (the size of the actual ATM cell). Further, 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 made possible by defining a dummy 8th byte in the data type definition for the *x*-kernel message structure. This byte was set to the value 255 (thus setting all bits) for the *x*-kernel messages. In the new representation, all the bits of the 8th byte can never be set as the PTI field only takes on values of 0 and 1. Thus the existing code in the simulator can easily distinguish between an ATM cell and an *x*-kernel message.

The new cell representation has the following advantages:

1. It avoids the overhead involved with manipulating data structures during segmentation and reassembly.
2. It requires less memory – 8 bytes as compared to at least 37 bytes of overhead required by the *x*-kernel messages (an empty *x*-kernel message requires 32 bytes and the original ATM header requires 5 bytes; the 48 byte payload is however shared by the original AAL5 frame).
3. It can easily fit in a single cache block (32 bytes on the alpha architecture). The representation using *x*-kernel messages required at least two cache blocks and had

[6] We consider an AAL5 frame to be active if any cell associated with it is active.

[7] If more active AAL5 frames are required, then some bits from *Node Handle* field can be added to the *Frame Id* field.

a greater chance of incurring a cache miss. Further, the x-kernel messages can also occupy more than one cache block on the second-level cache due to the dis-contiguous nature of the memory allocated to tree-based data structures. This is particularly important for ATM as each IP packet can get broken up into a large number of cells[8].

4. The individual fields in the 8 byte representation can be efficiently accessed on the 64 bit alpha architecture.

We've mentioned in Section 3 that the combination of VPI and VCI uniquely identify a virtual circuit in our simulator and the switches do not modify the VPI and VCI fields in cells. In a more general implementation, the ATM switches can change the VPI and VCI fields and a virtual circuit is uniquely identified by the tuple *(VPI, VCI, switch)*. For incorporating this in the method we describe above, the hashing will be be done on the first 7 bytes (instead of the 5 bytes) of the new cell representation. This would include the *Node Handle* and thus would uniquely identify a frame within a virtual circuit.

4.2 Detecting Corrupt Cells and Frames

Before an ATM cell is transmitted, an 8-bit CRC checksum is computed over its header and stored as part of the header (in the HEC field in Figure 3). Intermediate switches can thus detect if the cell header is corrupted by recomputing the CRC checksum and checking it against the stored vaue (a cell with a corrupted header is discarded). If any of the fields in the header is modified by the switches, then a new CRC checksum is computed and stored in the header. Thus for each cell, the 8-bit CRC checksum is computed at least once through each host/switch that it passes through.

An AAL5 frame is considered corrupted by the destination host if any cell associated with the frame is either corrupted or lost (dropped due to switch buffer overflows). To enable the receiver to detect corrupt frames, the sender computes a 32-bit CRC checksum and appends it as a trailer in the AAL5 frame. After the destination host receives the last cell in the frame, a CRC check is done on the complete frame and if it fails, the frame is considered corrupt and is thrown away. The original implementation of AAL5 thus computes the CRC checksum twice per frame – once at the source and once at the destination.

The above mentioned CRC computations impose a huge computational overhead in the simulator. Our new representation avoids the overhead associated with these CRC computations. Whenever a cell is dropped due to switch-buffer overflow, the original AAL5 frame is located by looking up the hash table (as explained in Section 4.1). The *dropFlag* attribute is set and the *numInactive* attribute is incremented. Similarly,

[8]The default MTU of IP over ATM is 9K bytes [1] which would be transmitted over the network in about 193 ATM cells.

when an uncorrupted cell is received by the AAL5 at the destination, the *numInactive* is incremented. As soon as *numInactive* equals the *numCells* attribute, the mapping is removed from the hash table and the frame is discarded or else is delivered to the protocol layer above depending on whether *dropFlag* is set or unset respectively.

Our simulator does not yet model corrupting individual cells, but it can be easily extended to do so. As soon as a cell is corrupted, the *dropFlag* associated with the corresponding AAL5 frame can be set. This will ensure that the AAL5 frame will be discarded. Discarding the corrupt cell depends on whether its header was corrupted or not. If it is, then *numInactive* can be incremented and the cell is discarded. If only the payload of the cell is corrupted, then it can simply be delivered to the next switch on the path as for an uncorrupted cell.

The improvements proposed in Section 4.1 and Section 4.2 result in a deviation from the earlier faithful implementation of the AAL5 and ATM protocols in the simulator. However, they capture all the essential features in addition to being less computationally expensive.

4.3 Event Handling

Another cause for the decrease in simulator efficiency can be attributed to the huge increase in the number of events being handled by the simulator while simulating ATM. As *x*-sim is an event driven simulator, transmission and propagation delays over network links are modelled by scheduling events in the simulator. Thus, if the number of events generated due to transmissions and propagation delays in a packet network simulation are M, and if N cells are contained within a packet, then an ATM simulation generates N*M times more events than the corresponding packet network simulation.

The primary operations implemented by the event management algorithms in an event driven simulator are:

- *schedule_event(ev, tm)*: Schedule an event *ev* to fire at time *tm*.
- *get_next_event()*: Returns the earliest event that has been scheduled. This function also increments the variable maintaining the simulated time to the time at which this event is to be fired.

The above two operations can be modelled by a priority queue algorithm [4]. The operations *schedule_event* and *get_next_event* are modelled by the *INSERT* and *EXTRACT_MIN* operations in the priority queue respectively.

The priority queue implementation being originally used in the simulator was *Timing Wheels* based on Scheme 5 described in [13]. In Section 4.3.1 we describe this implementation and the problems associated with it. In Section 4.3.2, we describe a Calendar Queue [3] based priority queue. In Section 4.3.3, we present our improvements to the Calendar Queue algorithm.

4.3.1 Timing Wheels

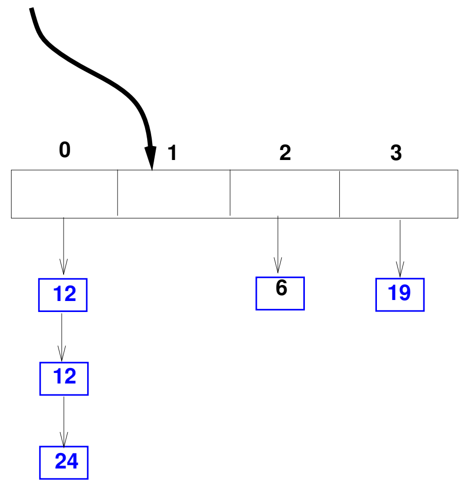

Figure 5: **Timing Wheels**

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

- *schedule_event(ev, tm)*: insert *ev* in sorted order in the list at the bucket given by the hashing function *(tm mod num_buckets)*.
- *get_next_event*: keep advancing *curr_time* by 1 until an event that needs to be fired at *curr_time* is found in bucket *(curr_time mod num_buckets)*. This is equivalent to circularly examining the array slots sequentially until an event is found.

An optimization while searching for the next event to fire in *get_next_event* involves keeping track of the event with the least value of time encountered. Thus, if no event matching *curr_time* is found during the first scan of all buckets, the *curr_time* (and hence the pointer) can be directly advanced to the event with the least time.

The number of buckets is dynamically varied depending on the number of events currently scheduled in the simulator. Thus, if the number of events grows greater than twice the number of buckets, the number of buckets are doubled and all the events are rehashed. Similarly, the number of buckets is halved if the number of events become less than half the number of buckets. This is done in an attempt to prevent the lengths of the event lists in buckets from becoming too large.

Even when the hashing function distributes events uniformly, the *get_next_event* operation can scan a large number of buckets before finding an event that can be fired. Consider the situation in Figure 5 when the value of *curr_time* is 1. Even though almost all buckets contain events, all buckets will be scanned before the next event to be fired (at time 6) is detected. This problem is specially acute when the number of events handled by the simulator are large, as the number of buckets are proportional to the number of events. Thus, *get_next_event* can impose a huge overhead when the average time separation between consecutive events is large. This effect will be quantified in Section 5.

4.3.2 Calendar Queues

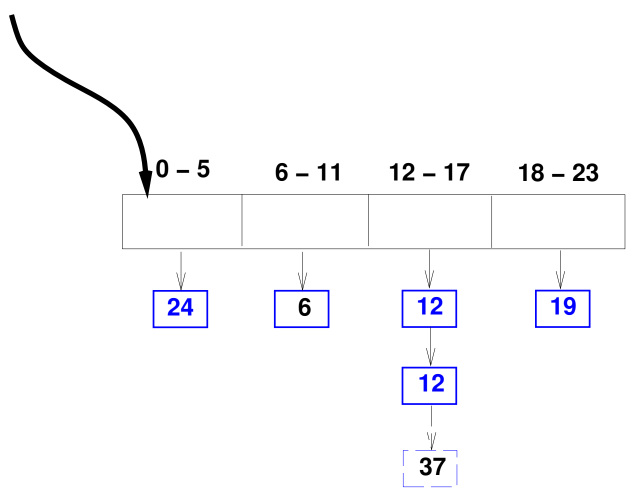

Figure 6: **Calendar Queue**

Calendar queues [3] aim towards providing O(1) *schedule_event* and *get_next_event* operations by associating a width with each bucket[9]. This *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 given as follows:

- *schedule_event(ev, tm)*: insert *ev* in sorted order in the list at the bucket given by the hashing function *((tm div bucket_width) mod num_buckets)*.
- *get_next_event*: keep advancing *curr_time* by *bucket_width* until an event is found in bucket *((curr_time div bucket_width) mod num_buckets)* which is to be fired at a time less than *curr_time* + *bucket_width*.

The calendar queue corresponding to the timing wheels in Figure 5 is given in Figure 6 (where the *bucket_width is 6*). As the number of buckets is proportional to the number of events, and the *bucket_width* is computed as the average separation between

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

the events, it is to be expected that each bucket would have constant number of events and the next event to be fired would be found in the next consecutive bucket. This makes the expected time of both *schedule_event* and *get_next_event* to be O(1). Our simulations indicate that the less than 0.001% of the *schedule_event* and *get_next_event* operations result in a recomputation of the bucket_width and the number of buckets.

4.3.3 Calendar Queues with Lateral Insertion

In a network simulation, many network nodes receive packets from the network at any simulation time instant. The reception of network packets is modelled by events in discrete event simulation. Hence, it is expected that the event handling algorithm has to handle many events that fire at the same simulation time instant. *x*-sim ensures that if two events are scheduled to occur at the same time, they are fired in FIFO order. As such events will be inserted in the same bucket in a Calendar Queue, thus any such event will be inserted at the end of the chain of other events that are scheduled to occur at the same time. These chains can become large when the network topology being simulated is large. Inserting events at the ends of such chains can then affect the speed of the operation *schedule_event*. Furthermore, even though such events get inserted in the same bucket, yet they affect the number of buckets (as it is proportional to the number of events). This can affect the speed of the *get_next_event* operation as it might unnecessarily scan empty buckets.

Figure 6 shows a chain of two events scheduled to occur at time 12. The speed of the *schedule_event* operation for inserting event with time 37 gets affected as it has to traverse the chain of events scheduled to occur at time 12. Similarly, any other insertions of events at time 12 would get similarly affected (for the purpose of maintaining the FIFO order).

To improve the speed of *schedule_event*, we introduce a technique which we call *Lateral Insertion*. This technique involves using an extra pointer, called the *lateral* pointer, in the event structure. After the insertion of the first event, all other events scheduled to occur at the same time are inserted *laterally* in a chain which is attached to the *lateral* pointer of the last event inserted. The free pointer (that was originally used to maintain the list) in the second such event inserted points to the last such event inserted. Thus, events can be inserted quickly in FIFO order. Further, the calender queue structure now keeps track of the total number of *distinct* events (events scheduled to occur at different times) for making a decision about changing the number of buckets.

Figure 7 shows the calendar queue with lateral insertion. The events scheduled to occur at time 12 are attached in a *lateral* chain. The second event in the chain has its free pointer pointing to the last event in the chain so that any further insertions can be made at the end quickly. There are 5 *distinct* events while the total number of events are 9. Thus the number of buckets is 4, even though the total number of events is more

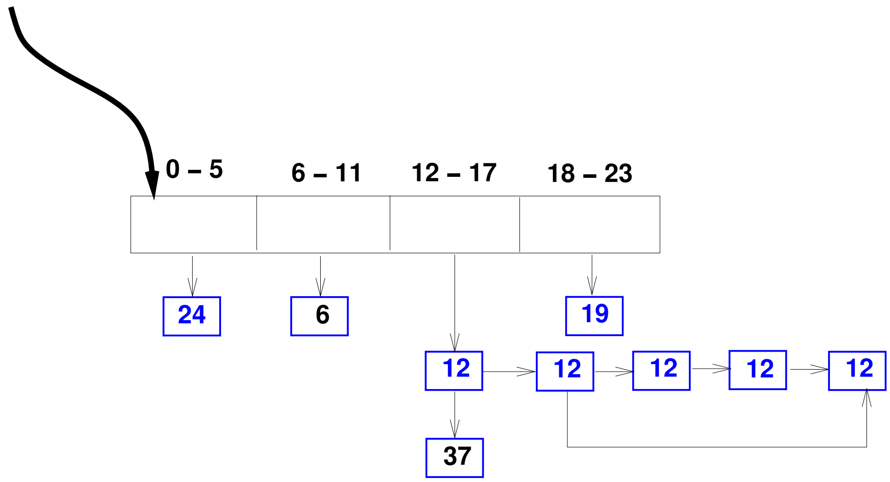

Figure 7: **Calendar Queue with Lateral Insertion**

than double the number of buckets.

5 Performance Evaluation

In this section we quantify the effects of the improvements proposed in Section 4. We first present a brief description of the profiling tools that we used to measure the performance of the simulator. We then describe the network configuration that we 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

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 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. The bottleneck link connects two switches R1 and R2 where R1 is the bottleneck switch. The round trip propagation delay is 60ms and all links have the same bandwidth. The parameter n,

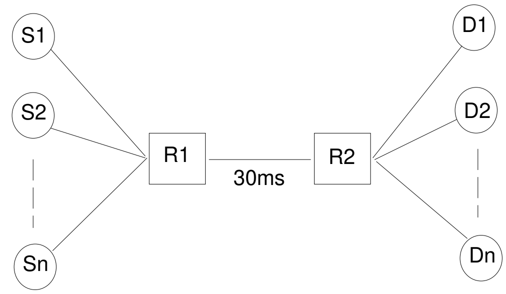

Figure 8: **Network configuration for simulations**

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

This network configuration resembles the ones 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. Our simulator does support these enhancements, but as this paper does not focus on TCP performance aspects, all simulations reported use switches with the FCFS queuing discipline. The TCP segment size was chosen based on the default MTU of IP over ATM which is 9180 bytes [1].

5.3 Results

In this subsection, we present results showing the impact of our improvements on the efficiency of the simulator. We present some of our experiments on the network configuration shown in Figure 8. We have also performed experiments with other network parameters and on other configurations with similar results.

5.3.1 Bulk Transfer Experiments

We begin by studying the performance of the simulator with bulk TCP transfers over ATM and packet networks. We configured the network configuration in Figure 8 so as to have 8 senders sending 60MB of data to 8 destinations (the parameter n is 8). The simulated time was limited to a value of 15s. The simulations were performed with two different values for the link bandwidths – 10.09Mb/s and 143.33Mb/s. These values were chosen so as to make the transmission of an ATM cell (53 bytes) an integral number of microseconds – $42\mu s$ and $3\mu s$ respectively. We consider these link bandwidths to be representative of typical low and high bandwidth networks found today. For simulations using link bandwidth of 143.33Mb/s, the TCP advertised window size was set to 512KB to accomodate the large bandwidth-delay product path.

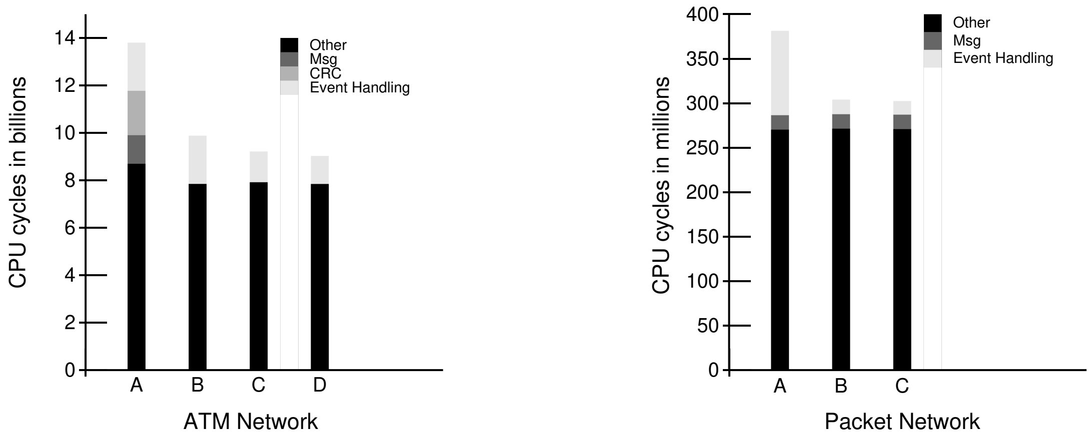

Figure 9: **Bulk Transfer Experiment with 10.09Mb/s links**

Figure 9 shows the distribution of CPU cycles for both ATM and packet simulations for the link bandwidth of 10.09Mb/s. Figure 10 shows similar distributions for link bandwidths of 141.33Mb/s. Each distribution shows the 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 4 distributions (marked A, B, C and D). Each of these quantify the results of successive improvements to the simulator as follows:

A: the distribution for the original faithful ATM implementation with Timing Wheels being used for event management.

B: for the new ATM implementation in the simulator which avoids CRC and *x*-kernel message overheads as mentioned in Section 4.1 and Section 4.2. The event management is handled using Timing Wheels.

C: for the improved ATM implementation (as in B) but uses Calendar Queues for event management.

D: as in C but also applies the *Lateral Insertion* technique to Calendar Queues as mentioned in Section 4.3.3.

The graph for the packet network shows 3 distributions. These stand for:

A: the distribution using Timing Wheels as the event management algorithm.

B: the distribution using Calendar Queues for event management.

C: shows the distribution when Calendar Queues are enhanced with the *Lateral Insertion* technique.

The results in 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

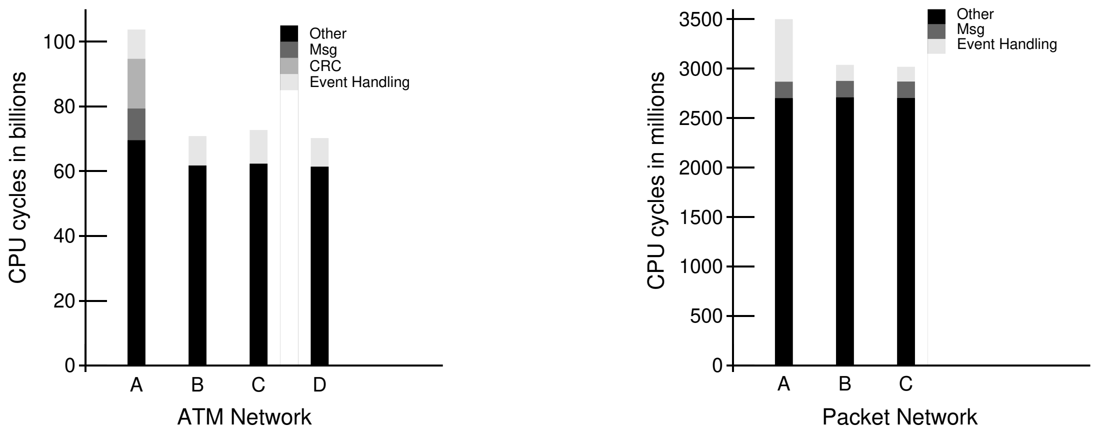

Figure 10: **Bulk Transfer Experiment with 141.33Mb/s links**

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 results show that the *x*-kernel message overhead decreases by about 99% by using the improved ATM implementation. The decrease in event overhead by replacing Timing Wheels with Calendar Queues depends both on the bandwidth as well as on the unit of transmission on the network. Thus, for the 10.09Mb/s links, the event handling overhead decreases by 36% for ATM and by 82% for packet network. For the topology using 141.33Mb/s links, this overhead actually increased by 14% for ATM while it reduced by 74% for the corresponding packet network. The overhead increases for ATM because with high bandwidths, the average separation between consecutive events is small ($3\mu s$ due to cell transmission time) and thus Timing Wheels perform quite well while the higher overhead of computing the *bucket_width* in Calendar Queues affects its performance adversely. For the 10.09Mb/s links, the *Lateral Insertion* technique decreases the event overhead by 9% and 7% for the ATM and packet networks respectively. The corresponding numbers for the 141.33Mb/s links are 15% and 8%.

The distributions for ATM also show that the CPU cycles spent in *other* parts of the program also decrease by about 10% when the improved ATM implementation is used. This reduction is a direct result of the better cache locality and the reduction in the overhead of memory allocation routines (malloc and free) due to the improved ATM implementation.

5.3.2 Background Traffic Experiments

We next performed simulations by running a single bulk transfer when the network is loaded with traffic generated from *tcplib* [5]. One of the application protocols available in x-sim is called TRAFFIC – it implements TCP Internet traffic based on tcplib. The configuration in Figure 8 was configured to have 1 sender doing a bulk data transfer (as in the Bulk Transfer experiments). The parameter n was set so as to have the remaining $(n-1)$ senders running the TRAFFIC protocol such that the combined tcplib traffic occupies 80% of the link bandwidth. We again report results for two different values of link bandwidths – 10.09Mb/s and 141.33Mb/s. For the former, the parameter n was set to 10, and for the latter it was set to 80. The higher bandwidth path needs more TRAFFIC hosts to occupy 80% of the bandwidth as each TRAFFIC host generates network traffic at a fixed frequency. The simulated time was limited to 15s as in the Bulk Transfer experiments.

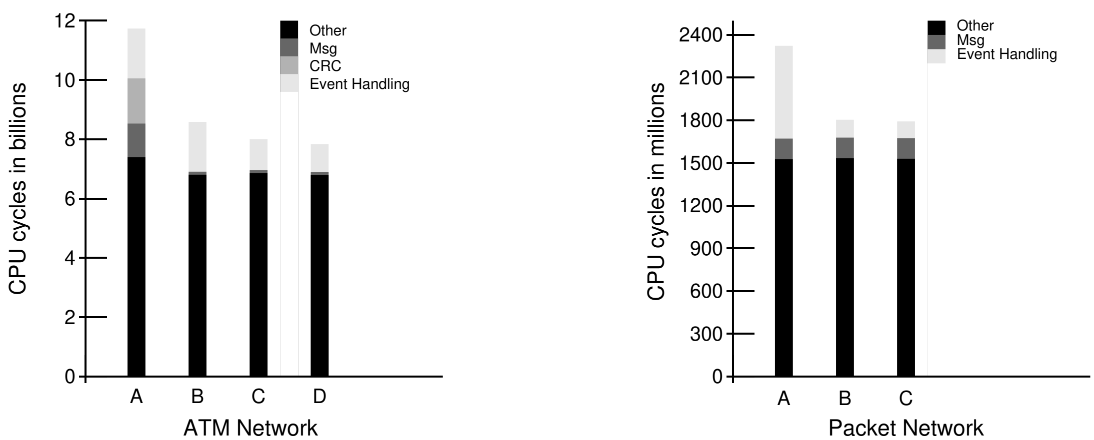

Figure 11: **Background Traffic Experiment with 10.09Mb/s links**

The results are presented in Figure 11 and Figure 12. The individual distributions have the same meaning as given in Section 5.3.1.

For the configuration with 10.09Mb/s links, the overall improvement for the ATM simulation was 33% and for the packet simulation it was 23%. 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% in packet simulation. The corresponding numbers for the simulations with 141.33Mb/s links are 14% and 8%.

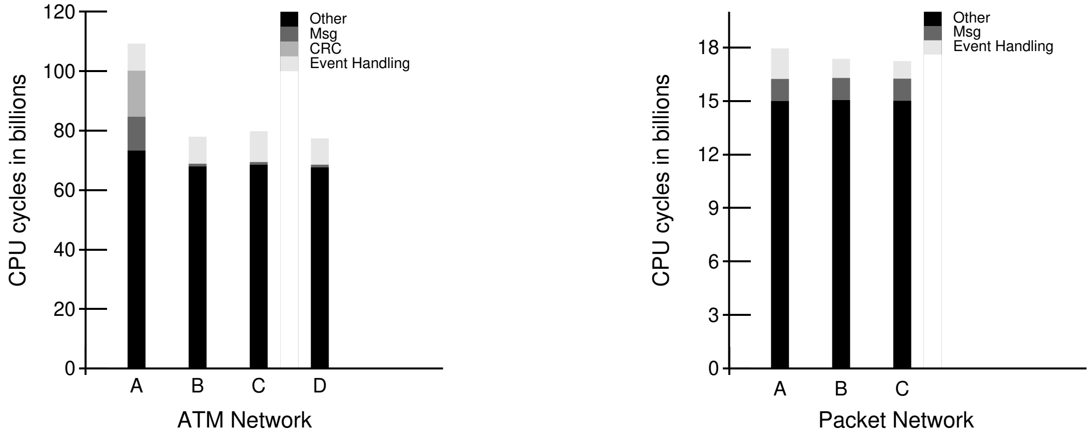

Figure 12: **Background Traffic Experiment with 141.33Mb/s links**

5.4 Discussion

The results presented in Section 5.3 show that our improvements increase the efficiency of the ATM simulations by 29-34% and the packet simulations by 4-23%. Our improvements eliminate the computational overhead associated with CRC and reduce the overhead associated with the *x*-kernel messages by more than 90% for the ATM simulations. Or results show that the Calendar Queues afford a significant performance advantage over Timing Wheels for packet simulations and for low bandwidth ATM simulations. For high bandwidth ATM simulations, the average separation between consecutive events is small, enabling the Timing Wheels to afford good performance. The *Lateral Insertion* technique affords a consistent improvement in event handling of about 15% for ATM simulations and 8% for packet simulations. Our results show that these improvements are mainly due to the increase in speed of the *schedule_event* operation.

A comparison of the CPU cycles taken by corresponding ATM and packet simulations show that the ATM simulations still take more cycles than the packet simulations. This is primarily because the processing done for switching each cell in ATM is almost the same as the processing needed for switching a packet in the packet network. As each IP packet is sent as several cells in an ATM network, therefore even when the event handling is quite efficient the ATM simulations have a higher processing overhead than a corresponding packet simulation.

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. Second, we have shown that unnecessary CRC computation can be avoided in the simulator. Finally, we have presented better event handling algorithms for efficient management of the large number of discrete events typical in an ATM simulation. We have shown that our techniques improve the efficiency of the simulator from 29-34%. We have also shown that the better event handling algorithms also improve the packet simulations by 4-23%.

While we introduced these techniques in the framework of our simulator *x*-sim, we believe that these techniques are universally applicable and can be easily used in other network simulators.

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.