Efficient Implementation of Packet Scheduling Algorithm on High-Speed Programmable Network Processors

Weidong Shi Xiaotong Zhuang *Indrani Paul Karsten Schwan

Georgia Institute of Technology, College of Computing, 801 Atlantic Drive
Atlanta, GA, 30332-0280
{Shiw, xt2000, schwan }@cc.gatech.edu
*indrani@ece.gatech.edu

Abstract. This paper describes the design and implementation of the Dynamic Window-Constrained Scheduling (DWCS)[1][2][3] algorithm to schedule packets on network processors. The DWCS algorithm characterizes multimedia streams with diverse Quality of Service (QoS) requirements. Earlier implementations of DWCS on Linux and Solaris machines use a heap-based implementation, which requires O(n) time to find the next packet and send it out, and which frequently moves heap elements. For speed improvements and conservation of memory bandwidth, our design uses a Hierarchically Indexed Linear Queue (HILQ). The HILQ substantially reduces the number of memory accesses by scattering packets sparsely into the queue. Experimental results demonstrate improved scalability compared to a heap-based implementation in supporting thousands of streams with strict real-time constraints, while causing no loss in accuracy compared to the standard DWCS algorithm.

1 Introduction

Network technology is being pushed forward to satisfy the ever-increasing requirements of future applications. Real-time media servers need to serve hundreds to thousands of clients, each with its own Quality of Service (QoS) requirements. Critical QoS properties like packet loss rate, deadline and delay variance need to be maintained without compromising the processing speed of incoming data streams. Many papers [1][4][5] have addressed the problem of scheduling priority-ordered input packets that packets from different flows are processed in an appropriate order, the order being determined by the QoS requirement of each stream. Packets of less priority not able to meet their QoS may be dropped to save computing power or bandwidth for more urgent packets.

This paper presents an efficient implementation of the Dynamic Window-Constrained Scheduling Algorithm (DWCS) [1][2][3] on high-speed programmable network processors. DWCS is designed to guarantee the QoS requirements of media streams with different performance objectives. Using only two attributes, deadline and loss-tolerance, DWCS can maximize the network bandwidth usage and limit the number of packets that have missed deadlines over a finite window of consecutive packets.

DWCS is also able to share bandwidth among competing streams in strict proportion to their deadline and loss-tolerance. Moreover, several traditional scheduling algorithms like Earliest Deadline First (EDF) [4], Static Priority, and Fair Queuing [5] can be simulated with DWCS.

Previous implementations of DWCS have been limited to Linux or cluster machines where computation resources are relatively plentiful compared to packet arriving rates. An earlier implementation of DWCS [3] uses a heap structure to store and select packets according to their deadline and loss-tolerance order. The computational complexity of this implementation is proved to be O(n), where n is the number of the simultaneously active streams in the system.

This paper considers the efficient implementation of stream scheduling on high-speed programmable network processors. The processors attain high processing speed by exploiting parallelism to hide memory latency. In the Intel IXP1200 network processor, for example, two Gigabit ports are supported by six microengines running in parallel. Each microengine contains four hardware contexts or threads. Memory operations and on-board computation can be performed concurrently by context switching. However, memory operations are time- consuming. Roughly, if the network processor is serving two Gigabit ports, one packet should be sent out every 500 – 600 microengine cycles (each microengine runs at 232 MHz), while each off-chip memory access takes about 20 cycles to complete. Although the latency can be somewhat hidden by context switching, memory intensive data structures like heaps or linked lists make it difficult to meet these requirements. An outcome is the need for new designs that address the limitation of memory-intensive data structures.

This is particularly important for packet scheduling algorithms [1][4][5], which rely on ordered queues or lists that must be updated, as packets arrive or are delivered. Specifically, to keep the queue or list ordered, with O(n) operations, where n is the number of elements in the queue, it becomes impossible to maintain an ordered queue (list) on the IXP network processor without major modifications in the packet scheduling algorithm while operating at Gigabit rates.

This paper addresses memory bandwidth limitations by holding the fact that memory space is of less concern than memory speed. The "Hierarchically Indexed Linear Queue" (HILQ) introduced in this paper reduces the time needed for packet ordering and priority adjustments by reducing the cost of both packet insertion and selection operations. It also eliminates the need to move data within memory by directly inserting packets into the queue that is intentionally kept sparse.

This paper is organized as follows. Section 2 discusses some of the related work. Section 3 presents the DWCS algorithm and discusses the previous implementations of DWCS. Section 4 introduces the IXP architecture. Section 5 describes the data structures and the implementation of HILQ. Section 6 presents the performance results and section 7 concludes the paper.

2 Related Work

There are many approaches to schedule a large number of streams. For instance, rate-monotonic [4] scheduling is based on the period of each task. The task with the shortest period gets the highest priority among all queued packets. Earliest Deadline First (EDF) [4] and Minimum Laxity First (MLF) are two variations of RM.

D.C.Stephens, C.R.Bernnett, H.Zhang[13] proposed the fast implementation of PFQ (Packet Fair Queueing) on ATM networks. They used a grouping architecture to reduce the cost of basic operations. Only a limited number of guaranteed rates are

supported by the server. Flows with similar rates are grouped together. Inaccuracy of flow rates is introduced when approximating flows with fixed number of rates. Their implementation uses a hierarchical calendar queue for intragroup scheduling, which is somewhat similar to the HILQ structure used by us. However, they add packets to the list of the same slot (when a conflict happens), and only two-level hierarchical structures are supported. As a result, the scan operation on the calendar queue is costly when the number of slots increases. Since they implement of PFQ, no garbage collection mechanism is needed to improve concurrency.

In [14], a heap is implemented with hardware (ASICs), where the hardware supports operations at each layer of the heap. The three basic operations (read-compare-write) can be quickly performed by pipelined processing.

In [15], the authors design a *hash with chaining* data structure to accelerate flow table management for high-speed routers. The hash value is provided by a hardware hash function, and the entries competing for the same hash value are chained together. Their hash function can achieve uniform hashing without considering the stream properties of packets, while in our case, uniform hashing will cause the packets to cluster around certain values. *Purge when convenient* is used to reduce the overhead of deleting entries from the flow table.

3 DWCS Algorithm and Earlier Implementations

The DWCS algorithm is detailed in [1] [2] [3]. We briefly reiterate some of the main features of DWCS in this section. We also discuss a previous implementation of DWCS.

The DWCS algorithm can work as a network packet scheduler. It schedules packets from multiple streams by limiting the number of late or lost packets over a finite number of consecutive packets. This is most favorable for video or audio streams, where a limited amount of packet loss is tolerable over a fixed transfer window. DWCS can also be applied to the scheduling of processes for time-sensitive real-time tasks. The DWCS algorithm uses two parameters for each stream:

Deadline -- Deadline is the latest time a packet can commence service.

Loss-tolerance -- This is specified as a value x_i / y_i, where x_i is the number of packets that can be lost or transmitted late for every window, y_i, of consecutive packet arrivals in the same stream i.

DWCS transmits the packets in an order according to their loss-tolerances and deadlines. Packets of the same stream have the same original and current loss-tolerances and will be scheduled in the order of their arrival. When scheduled by DWCS, the priorities of each stream can be adjusted dynamically. Whenever a packet misses its deadline, the loss tolerance for all packets in the same stream is reduced to reflect the increased importance of transmitting a packet from this stream. This approach prevents starvation and tries to keep the stream from violating its original loss-tolerance. Table 1 lists the complete packet-ordering scheme according to the priority.

Table 1: Precedence amongst pairs of packets (from [1])

x/y=0	x/y!=0
EDF	EDF
Same deadline, lowest y first	Same deadline, lowest x/y first
Same deadline and y, FIFO	Same deadline and x/y, lowest y first
	Same deadline, x/y, y, FIFO

Every time a packet in a stream is transmitted, the loss-tolerance is adjusted for that stream. Also, every time a packet misses its deadline, the loss-tolerance is adjusted to give

the stream a higher priority when the next packet comes. Formally, the pseudo-code for adjusting the loss-tolerance value is listed below in Figure 1.

```
        For packets transmitted before deadline:
if (y'i > x'i) then y'i =x'i-1;
if (y'i = x'i=0) then x'i =xi; y'i =yi;
        For any packet misses its deadline:
If(x'i>0) then
        x'i =x'i-1; y'i =y'i -1;
        if(x'i = y'i=0) then x'i = xi; y'i = yi;
else if(x'i = 0) then
        if(xi > 0) then y'i = y'i + ⌈(yi -xi)/ xi ⌉;
        if(xi = 0) then y'i = y'i + yi;
```

Figure 1: Pseudo -code for packet processing (from [1])

R.West, Chris. Poellabauer [3] fully implemented DWCS on Linux machines. This implementation uses a circular queue for each stream. Packets of the same stream are added to the tail of the queue and removed from the head of the queue. Two heaps are set up for deadline and loss-tolerance. With the heaps, packet insertion can be done using heap sorting. Packet precedence comparison will follow the rules of Table 1.

DWCS checks streams for packets that have missed their deadlines every time a packet has been serviced. In the worst case, every stream can have late packets when the scheduler completes the service of one packet. The deletion of these packets from the loss-tolerance heap and the heap adjustment will cost O(nlogn). Also, O(logn) time is required to find the next packet to be sent from the n streams. Thus, the current implementation on Linux with heap structure is not scalable as the number of active streams in the system increases. When the stream rate is low (more active streams co exist in the system) or the bandwidth is high compared to the stream rate, the two heaps will be adjusted frequently. Since more time is spent on insertion and selection, there will be further degradation in the throughput as the system is overloaded.

4 IXP Architecture

4.1 IXP hardware

The IXP (Internet Exchange Processor) 1200 in Figure 2 is a network processor from Intel. The processor consists of a StrongArm core, which functions as a traditional microprocessor. Connected to this core are six microengines (RISCs), which are responsible for managing the network data. Both the Strong Arm core and the six microengines run at the same clock rate (232 MHz). Furthermore, all microengines connect to a high-speed bus (IX bus, 80 MHz) for receiving and transmitting packet data. On the other side of the IX bus are external MAC devices (media access controllers). The MAC devices can be either multi-port 10/100BT Ethernet MAC, Gigabit Ethernet MAC, or ATM MAC. Both the StrongArm core and the microengines can access all of the address space of the SRAM and the SDRAM.

4.2 Pipelined processing

There are two choices in assigning microengines (µEs) to tasks. One is to duplicate the functionality of each µE, so that all µEs work in the same way to service the ports in a round robin fashion. The second choice is to assign the microengines with different tasks,

so that each of them can work on one job. The second choice allows threads in different microengines to be chained to form a processing pipeline.

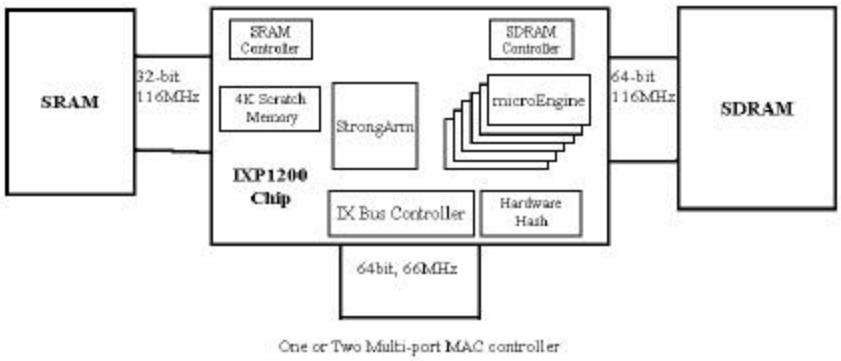

Figure 2: Block Diagram of IXP network processor

5 Hierarchically Indexed Linear Queue (HILQ) - Data Structure and System Design

5.1 Drawbacks of the heap structure

The current Linux implementation of DWCS has many drawbacks that make it unsuitable for network processors:
(i) The heap needs many memory operations. Selection and insertion each require O(log(n)) time. In the worst case, when every stream may have packets that missed deadlines, O(nlog(n)) time will be needed to delete and rearrange the heap.
(ii) The loss-tolerance heap will potentially become a bottleneck when packet insertion, packet dropping and packet selection happen concurrently. Introducing locks on the loss-tolerance heap will further degrade the performance.
(iii) A centralized scheduler will surely become a bottleneck even with a small number of streams (5-10). To maintain Gigabit speed, a centralized scheduler should have a scheduling period within 500 – 600 microengine cycles (microengine runs at about 232MHz).

5.2 Assumptions

Several assumptions influence the HILQ data structure's design and implementation. They are as follows:
(i) The IXP network processor can have very large off-chip memories, and a well-established memory hierarchy from scratchpad to SRAM, to SDRAM and it offers specific hardware for hash operations, freelists, memory locks, etc. So, a well-established data structure may benefit from this architecture.
(ii) We choose a video/audio unified traffic, where the intervals between frames are in the range of tens to at most one hundred milli-seconds. In other words, the packet rate is as low as 20 – 60 packets/second for each stream. However, the number of streams can be extremely large (hundreds to thousands of streams). This paper addresses scalability in number of streams, but current implementation avoids the need of frame fragmentation and reassembly by scheduling MAC-layer packets.

(iii) Our implementation deals with systems that are not overloaded, where the average stay time or the average delay time for each packet (including queuing time and processing time) is much less than the packet inter-arrival time, i.e. when the input and output packet rates are balanced.

5.3 Data structures

Previous researches on priority queues [13][14][16][17] used *heap* or *calendar queue* data structures. Calendar queues have a good average performance, especially when the queue is sparse. Our design can be thought of as calendar queue-like, since the "hash function" is not arbitrarily defined but based on the deadline and loss-tolerance. To search for the next packet to be sent, we traverse the queue linearly.

Figure 3 shows the overall data structure—the Hierarchically Indexed Linear Queue (HILQ). Level 0 is a continuous queue divided equally into 100 segments. Each segment represents one milli-second, i.e. all the packets with deadline in this milli-second are stored into the segment accordingly. Currently, 100 milli-seconds (segments) are included, which means we only look forward for this amount of time. 100 milli-second is sufficient for stream data, where inter-arrival times are typically much smaller. Also notice that the segments are organized as a circular queue, in a sense that it will loop back after segment 99. The transmission microengine will work on the current milli-second segment and then go forward to the next milli-second segment when all packets in the current segment have been serviced. The receiving engine will put the packets directly into the queue according to their deadlines. For packets with deadline 100 milli-seconds in the future, this structure will cause problems. For example, if the current time is 100.050s, the transmission engine will work on segment 50. For a packet with a deadline of 101.020s, it will be put into segment 20. The transmission engine proceeds linearly till segment 99 and then loops back to segment 0. It will reach segment 20 at time 101.20s. If we insert a packet with a deadline of 101.60s, then it will be erroneously put into segment 60, i.e. it will be processed at time 100.60s. Thus, our implementation does not support arbitrarily disordered streams.

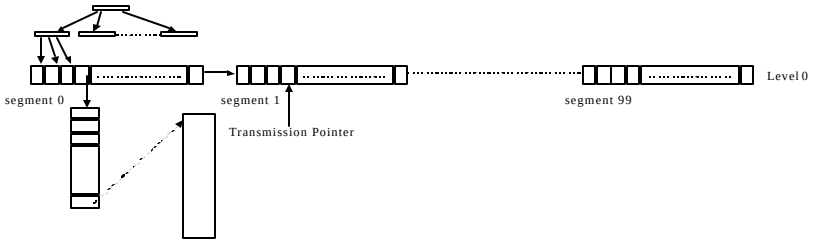

Figure 3: Hierarchically Indexed Linear Queue

Within each segment, a number of slots are created which is actually a mapping from the values of x and y. Figure 4 shows the structure inside each segment. First, to avoid the calculation of x/y, an x/y table is used to retrieve the value of x/y via a single memory operation. The size of the table depends on the ranges of permitted x and y values. The current implementation uses 50 distinct values in the x/y table. These 50 values are from 0 to 1 with a step of 0.02. Each x/y value is rounded to the nearest one among these 50 predetermined values. In other words, we may have up to 0.01 inaccuracy during the round off. We consider this amount of inaccuracy tolerable in practice. In our

experiments, x can have an integer value between 0 and 19; y can have an integer value from 0 to 29.

Figure 4 shows the 1000 slots and the interpretation of each slot. Slots are ordered based on Table 1. The slots with smaller indices will be serviced earlier since they have higher precedence. If x/y values are identical, slots are ordered again by the value of x, the numerator. So, 20 slots for each of the 50 x/y values are added to the queue. Therefore, a total of 1000 slots are needed for each milli-second. Till this point, we have considered the precedence of x/y and x.

However, it is still possible that packets with the same values of x/y and x fall into the same milli-second interval. In reality, this happens when streams are quite regular. We also observe that streams tend to cluster around several x/y values as x/y values are adjusted during the transmission of streams. This requests the existence of an element queue (a FIFO queue) for each slot. Actually, each slot currently contains information (header pointer and element number) about the element queue if there are packets in this slot. Multiple packets belonging to the same slot will be ordered by their arrival times and placed into appropriate queue position.

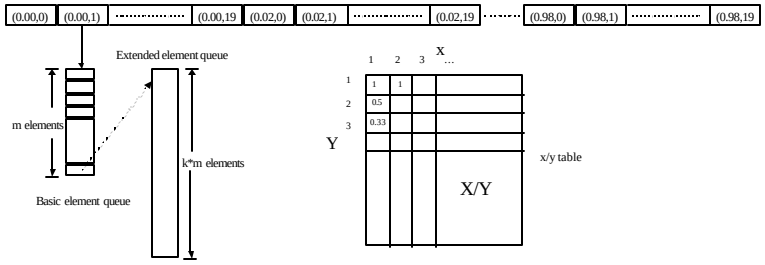

Figure 4: Elements inside a Segment

All three precedences have been taken into account in HILQ. To avoid frequent memory reallocation, there are two kinds of element queues for each slot, a basic element queue and an extended element queue. The basic element queue is created for a fixed number of packets, while the extended element queue is used only when the basic element queue is not large enough. Statistically, the extended element queue is rarely used. The two element queue structures can reduce memory needs while still providing enough support in the worst case. Memory usage is partly static and partly dynamic. The segments for each milli-second are allocated statically, whereas element queues (basic and extended) are dynamic data structures, which can be allocated on demand and released by the garbage collector.

5.4 Pipeline stages

The microengines are assigned three kinds of functionality—receiving, scheduling and garbage collection. They all operate on the shared HILQ data structure, while the element queues reside in the IXP SRAM for low memory access latency. We develop microcode component for each functionality and assign them to different threads to form a processing pipeline.

5.4.1 Receiving threads

When the network processor receives a packet, admission control will first check whether to accept it. Dropping packets by admission control is a violation of the DWCS rules. It

can only happen when the system is running out of memory, which may occur when the system is overloaded.

Receiving threads also do some common processing on the packet header like modification to the protocol fields, etc. After that, the pointer to the packet is inserted into a proper place in the queue. In DWCS, three property values are used to direct the packet insertion—D(deadline), x and y. Suppose the current time is T_{now}(in milli-seconds), then the packet should be put into the segment with number (T_{now}+D)%100. Then we will look up the value of x/y in the x/y table. In fact, the table stores the index number to the position inside the segment. Let the function LOOKUP do the mapping of the x/y table. We will insert the packet pointer into the element queue at the place LOOKUP(x/y)+x inside the segment.

If the packet is the first one in a slot, a new basic element queue is allocated from the freelist. If the basic element queue is full at the time a new packet is inserted, an extended element queue is allocated from the freelist.

5.4.2 Scheduling thread

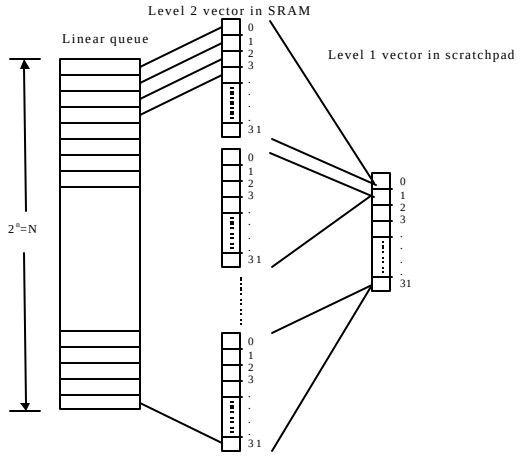

Figure 5: The Hierarchical Index Queue

The scheduling thread first works on the current milli-second segment and tries to send out all the waiting packets in this segment. The scheduling thread traverses the segment linearly. Certainly, there will not be any new packets put into the segment when the scheduling thread is working on it. Therefore, the search is unidirectional. We can also make use of the hierarchical index to skip empty positions quickly. When a packet is encountered, the scheduling thread will remove it from the queue, which includes resetting the pointer and releasing the element queue if needed. The x, y values of that stream should also be adjusted.

After servicing the packets in the current milli-second segment, according to the DWCS algorithm, we should go forward to send the packets in the future segments (we should not wait in the current segment, otherwise the computation power is wasted). However, the future segments are dynamically changing, when new packets are inserted into their deadline slots. We cannot simply search forward. If a new packet belongs to an

earlier place than the current search position, the scheduling thread has to go back and send it. Note that this will not degrade performance because when time for the next segment comes, the schedulingthread will move to the next segment immediately. All the packets before the next segment will be dropped.

Figure 5 shows a two level hierarchical indexed queue that can further reduce search time (n=10, $N=2^n=1024$). The level-one vector is a longword (4 bytes) in the scratchpad (on-chip) memory. The level two vectors are 32 longwords in the SRAM. Each bit of the level-two vector corresponds to an element in the queue. Each bit in the level-one vector corresponds to a longword in the level-two vector. To search out the next element with a packet, it requires only two memory reads.

5.4.3 Garbage collection thread

The garbage collector is called every time the scheduler has moved to the next milli-second segment. It searches the processed milli-second segment, drops all packets in the segment, and adjusts the x/y values for the streams with dropped packets. Separate threads are used for garbage collection. Since scheduling and receiving threads will not work on the same data location as the garbage collector, we are sacrificing the accuracy of the DWCS algorithm. This is because, at the time garbage collector begins to work but packet-dropping has not been completed, the receiving threads are still using the old x/y values of the streams (those x, y values may be modified if packets are dropped by the garbage collector). The garbage collector can work extremely fast under light overload (a small number of packets are dropped), thus limiting the inaccuracy to a small range. When the system is under heavy load, packets will not be admitted by the receiving threads, automatically constraining potential inaccuracies.

6 Performance evaluation and analysis

We have implemented the HILQ algorithm with microcode. All experiments are conducted on the Intel-provided simulation environment—IXP1200 Developer Benchmark 2.0. The IXP1200 workbench supports two microcode testing modes, simulation and real hardware. The user's microcode program can be tested in either of these two modes. It is capable of simulating most IXP devices and microengines with high fidelity. It also has a cycle accurate IX bus model that can simulate receiving and sending traffic data. We want to demonstrate that our HILQ based DWCS performs well with high-speed connections like Gigabit Ethernet. All the experiments are simulated in the workbench with a SRAM configuration of 8MB and SDRAM configuration of 64MB.

To get an idea of how costly the SRAM operations are, we collected the data in Table 2 to show the distribution of one SRAM operation. We can observe from the figure that most of the SRAM operations need about 20 cycles to complete.

Table 2: Latency Distribution for SRAM Access (cycles)

Cycles of SRAM Access	19	21	23	25	27	29	31	33
Percentage of SRAM Access	64.7	10.3	8.7	7.5	2.9	2.4	1.9	1.0

In Table 3, we compare the number of memory accesses required in our algorithm with that of a simple heap-based implementation. It takes only one SRAM access for a heap-based scheduler to find out the next packet to be sent. However, maintaining the heap requires O(logn) steps, and there are two heaps in the original DWCS implementation. Each step of heap reorganization requires six SRAM accesses. HILQ

minimizes the number of SRAM accesses as compared to the heap-based DWCS implementation. The number of SRAM accesses per stream is close to a constant for HILQ, while it increases logarithmically with the number of active streams for the heap-based DWCS. As shown in Table 3, SRAM access is expensive. Combining results in Table 3, we can conclude that a simple heap-based DWCS would not scale as well as the HILQ for a large number of streams.

Table 3: Memory Operation Comparison for Different Number of Streams

No. of streams		10	50	100	200	500	1000	2000
Memory access#	Heap	45.86	73.73	85.73	97.73	113.59	125.59	137.58
	HILQ	19.8	14.36	13.68	13.34	13.135	13.068	13.034

We test the scalability of the HILQ-based scheduler in simulation. There are two sets of tests, one without input packet traffic, and the other with input/output packet traffic. The objective of the first testing is to chart the actual scheduling overhead under different numbers of active streams. To obtain a large number of active streams, we use a separate microengine to generate a variety of pseudo-streams. In total, there are two microengines in use: one sends stream scheduling requests and the other runs the HILQ scheduler. Figure 6 shows the scheduling delay per active stream as we vary the number of active streams. It indicates that our HILQ-based DWCS scales well for a large number of active streams. As the number of streams increases, the average scheduling delay per active stream approaches 500 cycles/per stream. Readers may notice the high scheduling delay when the number of active streams is small (below 20). This is caused by a fixed amount of overhead in our HILQ scheduler. The scheduler has to read and check all indexes during a segment sweep even when there are only few active streams. Secondly, we want to show that the HILQ scheduler can actually produce high data throughput. Traffic is fed into the Developer Benchmark through a packet generator. We can adjust the packet speed and the number of streams through the packet generator and the configuration option in the simulator.

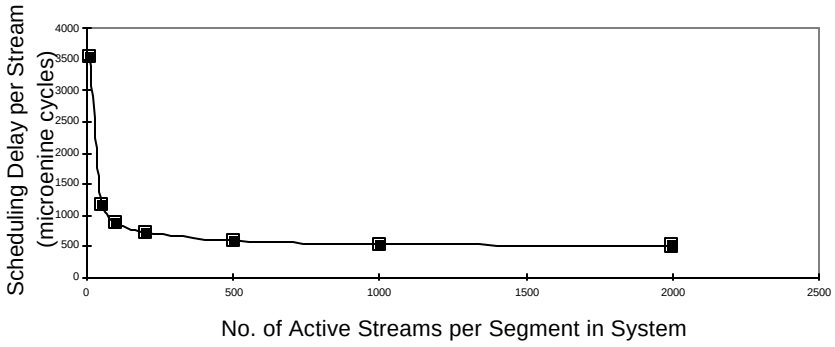

Figure 6: Scalability - No. of Active Streams vs. Average Scheduling Delay

We linked our HILQ scheduler with microcode that performs packet receiving and transmission. There are two microengines, which run the receiver code that pulls packets from a Gigabit Ethernet port, and two other microengines run the transmission code that sends packets to another Gigabit Ethernet port. One more microengine is used for the HILQ scheduler. The receiver thread pulls each packet from the wire and puts it into the scheduler queue. The HILQ scheduler scans each segment linearly and sends out the

waiting packets to an output queue. The transmission thread reads from the output queue and puts the packet on the wire. To sustain a certain data rate, the scheduler has to complete a packet scheduling cycle within a time limit. Figure 7 shows the throughput vs. the scheduling cycle for different sizes of packets. The figure indicates that large packets allow a higher time budget for a packet scheduler. When the packet size is 1K, a packet scheduler with about 900 cycle scheduling overhead is still capable of supporting a Gigabit data link. However, for small packet size like 128 bytes and 256 bytes, throughput starts to drop even with small scheduling overhead.

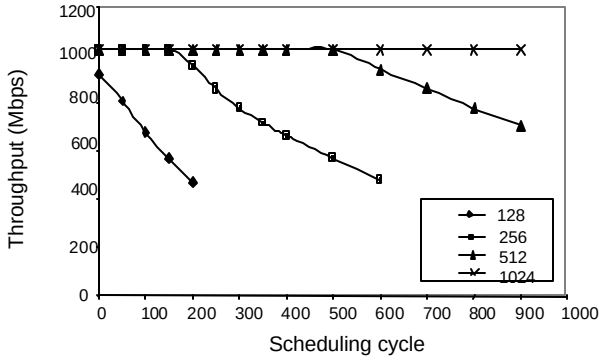

Figure 7: Throughput vs. Scheduling Cycles for Different Packet Sizes

One interesting result concerns the 512-byte packet size. There is about a 600 cycles time-budget for a packet scheduler to sustain a Gigabit link with a packet size of 512 bytes. This means that any packet scheduler that can schedule a packet in less than 600 cycles will not have a negative impact on the data rate. Since our HILQ has a limit of about 500 cycles, we anticipate that it should be capable of supporting Gigabit throughput for streams whose packet size are equal to or above 512 bytes. In Figure 8, the results indicate that with our HILQ implementation, the DWCS QoS scheduler is capable of supporting a Gigabit data link when the packet size is 512 bytes. For smaller size packets such as 256 bytes, the throughput is about 650MB/ps. This result is consistent with the scheduler time budget shown in Figure 7. For 512-byte packets, the scheduler is able to schedule about 260,000 packets per second. If each stream has only about 40 – 60 packets per second, our HILQ scheduler would be able to support about 4,000 – 6,000 streams. Note: Figures 6 and 7 refer to the number of active streams.

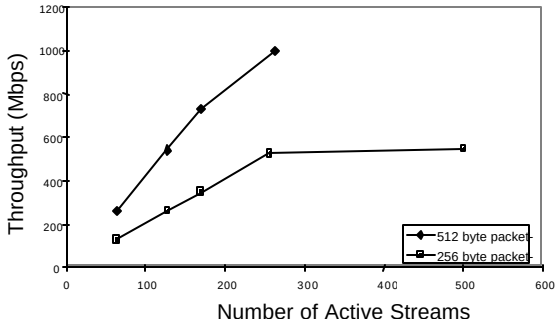

Figure 8: Throughput vs. No. of Active Streams for Different Sizes of Packets

The last result compares our DWCS approximation with the original DWCS. The single most important metric for evaluation is window constraint violation. Theoretically, DWCS does not have any loss tolerance violation when the bandwidth utilization factor is below 1.0 [2]. According to [2], the bandwidth utilization factor is computed as U = sum((1-Wi)Ci/Ti), i = 1,2,…, n, where Wi = xi/yi, and Ci is the amount of time required to serve a packet of stream i and Ti is the request period of stream i. However, when U exceeds 1.0, there will be window constraint violations. Results in Figure 9 show that our DWCS approximation does not create more window constraint violations than the heap-based implementation when U exceeds 1.0. The results are collected from a simulation having 200 streams. Each stream starts with a random Wi = xi/yi lying between 0.2 and 0.8. The range of x is 0 to 19 and the range of y is 0 to 29.

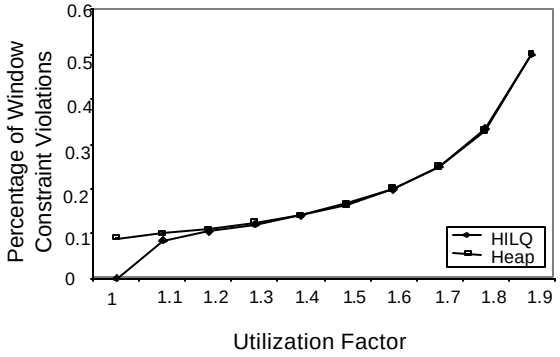

Figure 9: Comparison between HILQ and Heap Based Implementation under Different Utilization Factors

7 Conclusion

This paper proposes a fast algorithm to implement a multimedia stream scheduler on the IXP Network Processor. A data structure called Hierarchically Indexed Linear Queue meets the requirements of Gigabit speeds by exploiting certain architectural attributes. High performance is gained by integrating send/receive and garbage collection threads.

Our implementation can achieve O(1) packet insertion and selection time by hierarchically indexing a sparse queue and separation of the garbage collection threads. Although our design incurs a small amount of inaccuracy with regard to the standard DWCS specification, the inaccuracies are shown to be tolerable. Our performance results strongly support the benefits of HILQ by serving up to thousands of streams at Gigabit speeds.

REFERENCES

[1] Richard West and Karsten Schwan, "Dynamic Window-Constrained Scheduling for Multimedia Applications," *in Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS)*, 1999.

[2] Richard West and Christian Poellabauer, "Analysis of a Window-Constrained Scheduler for Real-Time and Best-Effort Packet Streams," *in Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS)*, 2000.

[3] Richard West, Karsten Schwan and Christian Poellabauer, "Scalable Scheduling Support for Loss and Delay Constrained Media Streams", *in Proceedings of the 5th IEEE Real-Time Technology and Applications Symposium (RTAS)*, 1999.
[4] C. Liu and J. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of ACM, 20(1)*, January 1973.
[5] P.Goyal, H.Vin, and H.Cheng, "Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks," in *IEEE SIGCOMM'96*, IEEE,1996.
[6] K. Parekh. "A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks," *PhDthesis*, Department of Electrical Engineering and Computer Science, MIT, February 1992.
[7] "IXP 1200 Network Processor: Software Reference Manual", Part No. 278306-005. Sep. 2000.
[8] "IXP 1200 Network Processor: Programmer's Reference Manual", Part No. 278304-006. Sep. 2000.
[9] "IXP 1200 Network Processor: Development Tools User's Guide", Part No. 278302-005. Oct. 2000.
[10] P.Crowley, M.E.Fiuczynski, J.Baer, B.N.Bershad, "Characterizing processor architectures for Programmable Network Interfacess," *Proceedings of the 2000 International Conference on Supercomputing*, pp.54-65, May. 2000.
[11] Tammo Spalink, Scott Karlin, Larry Peterson, Yitzchak GottliebIn, "Building a Robust Software-Based Router Using Network Processors," *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)* . pages 216--229, Chateau Lake Louise, Banff, Alberta, Canada, October 2001.
[12] Xiaohu Qie, Andy Bavier, Larry Peterson, Scott Karlin "Scheduling Computations on a Software-Based Router," *Proceedings of SIMETRICS 2001*, Jun 2001.
[13] D.C. Stephen, J.C.R.Bennett, H.Zhang, "Implementing scheduling algorithms in high-speed networks", *IEEE Selected Areas in Communications*, pp.1145-1158, Vol 17, No.6, Jun.1999
[14] A. Ioannou, M. Katevenis: "Pipelined Heap (Priority Queue) Management for Advanced Scheduling in High Speed Networks," *Proc. IEEE Int. Conf. on Communications (ICC'2001)*, Helsinki, Finland, June 2001, pp. 2043-2047.
[15] Jun Xu and Mukesh Singhal, "Cost-Effective Flow Table Designs for High-Speed Internet Routers: Architecture and Performance Evaluation," to appear in *IEEE Transactions on Computers*.
[16] Ranjita Bhagwan, Bill Lin "Fast and Scalable Priority Queue Architecture for High-Speed Network Switches," *IEEE INFOCOMM'00*, Tel Aviv, Vol. 2, pages 538-547, March 2000.
[17] R. Brown, "Calendar queues: a fast 0(1) priority queue implementation for the simulation event set problem," *Communications of the ACM*, vol. 31, pp. 1220--1227, October 1988.
[18] K. Bruce Erickson, Richard E. Ladner, Anthony LaMarca, "Optimizing Static Calendar Queues," *IEEE Symposium on* Foundations of Computer Science,1998.
[19] Rajamar Krishnamurthy, Karsten Schwan, and Marcel Rosu, "A Network Co-Processor-Based Approach to Scalable Media Streaming in Servers", *International Conference on Parallel Processing (ICPP)*, August 2000.