

A Scalable, Cache-Based Queue Management Subsystem for Network Processors

Sailesh Kumar and Patrick Crowley

Applied Research Laboratory
Department of Computer Science and Engineering
Washington University in St. Louis, MO, 63130

Abstract— Queues are a fundamental data structure in packet processing systems. In this short paper, we propose and discuss a scalable queue management (QM) building block for network processors (NPs). We make two main contributions: 1) we argue qualitatively and quantitatively that caching can be used to improve both best- and worst-case queuing performance, and 2) we describe and discuss our proposal and show that it avoids the failings of existing approaches. We show that our cache-based approach improves worst-case queue operation throughput by a factor of 4 as compared to a cache-less system. We also argue that our approach is more scalable and more efficient than the cache-based mechanism used in Intel’s second-generation network processors.

I. INTRODUCTION

When packets enter or prepare to leave a router, they are typically placed in a queue depending on its source, destination, or type (including protocol or application). The use of packet queues enables systematic allocation of resources such as buffer space and link and switch bandwidth.

Queues allow packets to be buffered together according to various criteria. Queuing applications are numerous, and include: per-flow queuing to rate-limit unresponsive flows that do not participate in congestion control [5]; hierarchical queues and scheduling algorithms that provide integrated link sharing, real-time traffic management, and best-effort traffic management [2]; and, the use of multiple queues to approximate fair interconnect scheduling [3].

Since packets must be enqueued and dequeued upon arrival and departure, respectively, these operations must occur at lines rates. This is a challenge since packet transmission is limited by speed-of-light constraints, while queue operations are typically constrained by external memory speeds.

In this paper, we: describe packet queuing in NP-based line cards (Section II); discuss a scalable, cache-based queue management building block that is compatible with existing NP-based SRAM and DRAM architectures (Section III); present a case for using cache-based approaches in queue management (Section IV); and contrast our proposal with the mechanism used in the second generation of Intel IXP [1] network processors (Section V).

Our proposal is not the first cache-based queue management approach (e.g., the Intel IXP uses one), but we believe that showing how and why caches are

beneficial under best and worst-case queuing conditions is a novel contribution.

II. PACKET QUEUES IN NETWORK PROCESSORS

The organization of an NP-based router line card is shown in Figure 1. There are, of course, variations among line cards, including those that do not use NPs [4], but we do not consider these in this short paper. In both the ingress and egress directions, an NP sits between the switch fabric and the physical interface. NPs are typically organized as highly-integrated chip-multiprocessors. For concreteness, we will use the IXP2800 as an example throughout this paper. It features 16 pipelined processors, called micro-engines (MEs), each of which support 8 thread contexts and zero cycle context switches in hardware. The chip also integrates 4 QDR SRAM controllers and 3 Rambus DRAM controllers, along with many other hardware units unrelated to packet queues. In line cards like this, both SRAM and DRAM are used to implement packet queues. Queues and their descriptors are kept in SRAM, while the packets are kept in DRAM. The scheduling discipline is implemented in software on an ME.

The framer can be viewed as an integrated device that interfaces one high-bandwidth port (e.g., 10 Gb Ethernet) or many lower-bandwidth ones (e.g., 10x1Gb Ethernet). The switch fabric is the interconnection path between line cards and, hence, router ports.

A. Queue Hierarchies

Queue hierarchies are often used to provide packet scheduling and QoS. Many routers use a three-level hierarchy, where the first represents physical ports, the second represents classes of traffic and the last level consists of virtual (output) queues. Note that both the port and class levels are meta-queues in the sense that they contain queues (only the virtual queues contain packets). Each ingress NP maintains a queue for each output port (this avoids head-of-line blocking); each of these output port queues has a number of class queues associated with it (this enables QoS); each of these class queues consists of per-flow virtual output queues (this allows individual flows to be shaped, esp. unresponsive ones causing congestion).

Each incoming packet is enqueued into some virtual queue, and the corresponding class and physical queue statuses are updated to record the activity. A similar

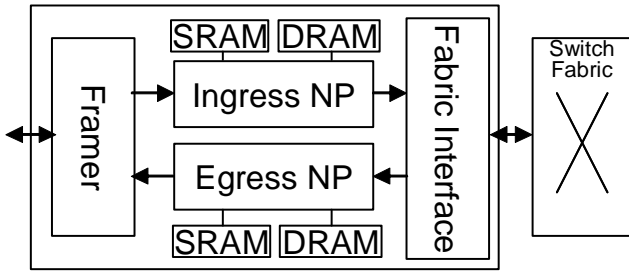


Figure 1: NP-based router line card

sequence occurs when a packet is dequeued from a virtual queue by the scheduler. Scheduling is typically carried out from root to leaf; i.e., first, the port is selected according to the port selection policy, then a class from the selected port is chosen which is followed by a virtual queue selection. It is important to note that one enqueue and one dequeue are expected each packet arrival/departure period. Moreover, since both involve updates to shared queues, serialization can occur.

Virtual queues are generally kept in a linked list data structure because packets are generally enqueued and dequeued sequentially from the virtual queue. Port and class queues, however, are kept in a linked list data structure only if the selection policy of class and virtual queue is ring based (round robin or deficit weighted round robin are examples of ring based selection policies, where next selection is the next link in the ring of active queues).

A queue's status needs to be updated for every incoming and outgoing packet, so that scheduling can be carried out efficiently (e.g., a queue's occupancy can influence the schedule). In some architectures, every enqueue and dequeue command, along with the respective queue addresses, are passed to the scheduler, which manages its own local queue status database. This keeps the scheduler from either using stale information or making frequent queue descriptor read requests.

B. A Packet Processing Pipeline

NPs incorporate multiple processors on a single chip to allow parallel packet processing. Packet processing is typically implemented as a pipeline consisting of multiple processor stages. Whenever a stage makes heavy use of memory (e.g., queue operations), multiple threads are used to hide the latency. The processing pipeline generally consists of the following tasks.

Packet assembly- Several interfaces deliver packets in multiplexed frames or cells across different physical ports.

Packet classification- Incoming packets are mapped to a queue in each hierarchy.

Admission control- Based on the QoS attribute of the queues, such as a maximum size, packets are either admitted or dropped.

Packet enqueue- Upon admission, the packet is buffered in the DRAM, and the packet pointers are enqueued to the associated queues. Most architectures buffer the packet in DRAM at the first stage and then deallocate the buffer later if the packet is not admitted.

Scheduling and dequeue- The scheduler selects the queues based on the QoS configuration, then a packet is dequeued and transmitted.

Data Manipulation, Statistics- A module may perform statistics collection and data manipulation based on the configuration. Packet reordering, segmentation and reassembly may also be performed.

C. Queue Operations and Parallelism

Both the queue descriptors, consisting of head and tail pointers and the queue length, and the linked-lists (implementing the queues) are stored in SRAM. SRAM and DRAM buffers are allocated in pairs, so that the address of the linked-list node in SRAM indicates the packet address in DRAM. Thus, the linked-lists in SRAM only contain next-pointer addresses.

With this structure, every enqueue and dequeue operation involves an external memory read followed by a write operation. Several read followed by write operations may need to be carried out in the case of hierarchical queuing. Recall that since the access time of external memory requires many processor cycles, multiple threads are used to hide the memory latency. A system can be balanced in this way by adding processors and threads, so long as each thread accesses a different queue. As soon as threads start accessing the same queue, the parallelism is lost, since every queuing operation involves a read followed by write, and the write back is always based on the data that was read. In the worst case, all threads compete for the same queue and progress is serialized, regardless of the number of processors or threads.

As we will show, using an on-chip cache for queue descriptors can improve this worst-case performance.

III. PROPOSED QUEUING ARCHITECTURE

We propose a queuing cache architecture for NPs that scales well with increased numbers of threads and processors. The proposed architecture uses a shared on-chip cache of queue descriptors accessible by every ME. The cache ensures that every thread operates on the most recent copy of data and internally manages queue descriptors and the read and write-back operations from external memories. We will later show that our scheme also simplifies the programming of network processors.

Our proposal shares two important features with the Intel IXP: 1) it presumes hardware support in the SRAM controller for queue operations, and 2) it uses an on-chip cache to hold in-use queue descriptors. In Section V, we describe Intel's approach in detail and compare it to our

own. First we describe our proposal and how it handles packet enqueue and dequeue operations. Then, we discuss the benefits of caching in this context and the instructions needed to implement it.

A. Managing Queues with the Queuing Cache

Due to the speed gap between the NP and its external memory, multiple processors and thread contexts are used to provide higher throughput despite long-latency memory operations. In our architecture, groups of m threads are used on each of n processors. Thus, a total of $m*n$ threads are used to handle enqueues and dequeues. For a given application, these parameters can be chosen as follows.

1. The number of threads on a processor, m , is determined by the ratio of time the program spends waiting for memory references to the time spent using the ALU. For example, a ratio of one implies that two threads could completely hide the latency.
2. The number of processors, n , is determined by the aggregate throughput requirement and the throughput supported by a single processor. However, when the external memory interface bandwidth saturates, adding further processors will have no effect.

When threads access different queues, multithreading automatically pipelines the queuing operations at the memory interface and, hence, throughput is only constrained by memory bandwidth and the number of threads. Each thread increases the throughput linearly until the bandwidth at the memory interface saturates. However, when threads access the same queue throughput drops dramatically because a) the queue accesses are all serialized and no longer pipelined, and b) queue descriptors are stored in external memory and hence every enqueue/dequeue incurs the latency of multiple memory references. Our proposed queuing cache eliminates this worst-case problem.

We use an on-chip cache which moves the most recent copies of queue descriptors to and from external memory. Every enqueue/dequeue command for an external memory-based queue passes through the cache, which sends the request to external memory only if the required queue descriptors are not present in the cache (a miss). Upon a hit, the queue descriptor in the cache is updated and the associated links of the queue are updated in the external memory. Queue descriptors are evicted only if a miss occurs while the cache is full. The mapping of a queue ID (i.e., the address of the queue descriptor) to a cache location is handled internally by the queuing cache. A fully associative mapping is feasible in this context (evict LRU entry), since request arrival is rate-limited by the chip-interconnect (i.e., rather than a processor pipeline stage). The top level architecture using such a scheme is shown in Figure 2.

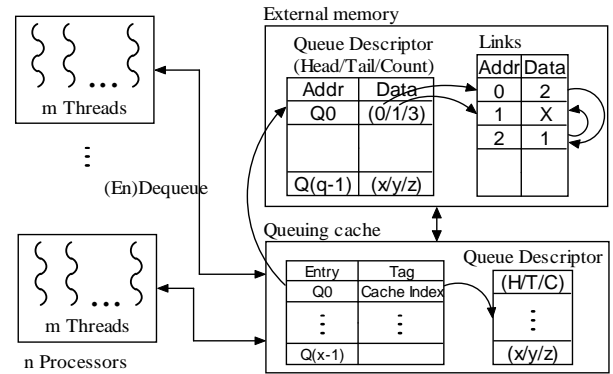


Figure 2: Top-level queuing cache architecture

With a queuing cache, queue descriptors need not be accessed from external memory multiple times when many threads access few queues in a small window of time. Thus in a scenario of multiple threads accessing a queue, the queuing operation itself is accelerated. Therefore, even if queue operations from multiple threads are serialized at one queue, the throughput is improved since the serialized operations (queue descriptor read and write) happen on-chip.

B. Benefits of Caching

The worst-case condition now occurs when an operation misses in the cache and an eviction must be carried out (due either to a conflict or a full cache). Thus the queue descriptor needs to be read, along with an eviction and a write back.

To illustrate these ideas, Figure 3 shows a series of queue operations from multiple threads on a single processor (with a single ALU) both with and without a queuing cache. Two scenarios have been shown, a) one when the queuing operations are performed across distinct queues and b) another in which all threads contend for a single queue. It is clear from the figure that, as threads hit a single queue, the performance without a queuing cache degrades rapidly. With a queuing cache, performance remains the same, whether queuing is done on a single queue or on distinct queues.

C. Queuing Instructions

Given a queue ID (address of the queue descriptor) and a packet ID (head and tail address of the packet to be enqueued), the queuing and linking operations can be performed entirely by the queuing cache. If the ISA provides enqueue and dequeue operations, then the queuing cache can hide the implementation of these operations from the software. This provides a simple programming interface, and saves cycles on the processor. Our proposed enqueue and dequeue instructions with their arguments and return values are shown in Table 1. It may also be desirable to add an instruction that pins a queue descriptor in the cache.

Table 1: Enqueue and Dequeue instructions

Instruction	Arguments	Returns
Enqueue	Queue ID, Head, Tail, Count	Null
Dequeue	Queue ID	Buffer/cell ID

IV. EFFECT OF QUEUING CACHE

Some of the benefits of the queuing cache have been discussed qualitatively in previous sections. In this section, we consider these benefits in greater detail, beginning with a quantitative analysis of performance.

A. Significantly higher throughput

We now construct an analytical model of throughput. In addition to our previous notation of m threads on each of n processors, we assume the following notation.

Total number of threads	$= T = m*n$
Read and write latency to cache	$= c$
Read latency of external memory	$= r$
Burst read throughput	$= 1/br$
Write latency to external memory	$= w$
Burst write throughput	$= 1/bw$
Burst read/write throughput	$= 1/bwr$

Thus a single write takes time w and x parallel writes take $w+(x-1)*bw$ units of time. Replace w with r for reads and wr for alternate reads and writes.

1. Throughput without Queuing Cache

Consider a multithreaded packet queuing system without any caching. An enqueue operation involves the following steps.

1. Read queue descriptor (head, tail and count).
2. Update tail and increment the count.
3. Post a write to the external memory to make the old tail point to the new tail.
4. Write back the updated queue.

Step 1 takes r units of time, and assume that processing time (in step 2) is P . Steps 3 and 4 can be carried out in parallel, therefore will take time $(w+bw)$. Thus, throughput of a single thread will be $1/(r+P+w+bw)$.

The best case will be when every thread accesses distinct queues. Throughput will increase linearly with the number of threads and will be $T/(r+P+w+bw)$. The worst case arises when all threads operate on the same queue. In this scenario, threads need to operate sequentially and they also need to use some inter thread mechanism to ensure coherent sequential access. In this scenario, the throughput of a single thread will be the

total throughput of system and hence will be $1/(r+P+w+bw)$ regardless of the number of threads.

A dequeue operation involves

1. Read queue descriptor (Head, tail and count).
2. Post a read to external memory to get the head node.
3. Update the head and decrement the count.
4. Write back the queue descriptor to memory.

Note that here, none of the above steps can be parallelized; hence, worst case throughput when all threads operate on same queue is $1/(r+r+P+w)$. This is valid regardless of the number of threads employed. Again the best case throughput will be $T/(r+r+P+w)$.

2. Throughput with Queuing Cache

When a queuing cache is used, an enqueue involves

1. Check if queue descriptor is present in cache. Consider worst case, i.e. miss and cache is full
2. Select an entry (LRU) in cache, and post a write to memory (this entry is getting evicted)
3. Post a read request from cache to memory to get the queue descriptor into cache
4. Update tail and increment count in cache
5. Post a write to external memory to make the old tail point to the new tail

Step 1 takes time c . Steps 2 and 3 can be carried out in parallel, therefore take time $\text{Max}(w,r)+bwr$. Step 4 takes time P and Step 5 takes time w . Note that when N threads access the same queue, Steps 2 and 3 can be skipped $N-1$ times, because the queue descriptor stays in the cache after the first queue access. Hence throughput when N threads access the same queue is $X(N) = N/(N*c+\text{Max}(w,r)+bwr+N*P+N*bw)$. Note bw is present instead of w because every thread can perform Step 5 independently and thus will achieve the burst access throughput. Now, when threads access distinct queues, the throughput of a single thread will be $1/(c+\text{Max}(w,r)+bwr+P+bw)$, and since each thread operates independently, for M threads, throughput will be $Y(M) = M/(c+\text{Max}(w,r)+bwr+P+bw)$, assuming that M is not enough to saturate the memory and cache bus.

A dequeue involves

1. Check if queue descriptor is present in cache. Consider worst case, i.e. miss and cache is full.
2. Select an entry in cache, and post a write to memory
3. Post a read request from cache to the memory to get the queue descriptor
4. Post a read to the memory to get the next pointer of the head
5. Update head and decrement count in cache

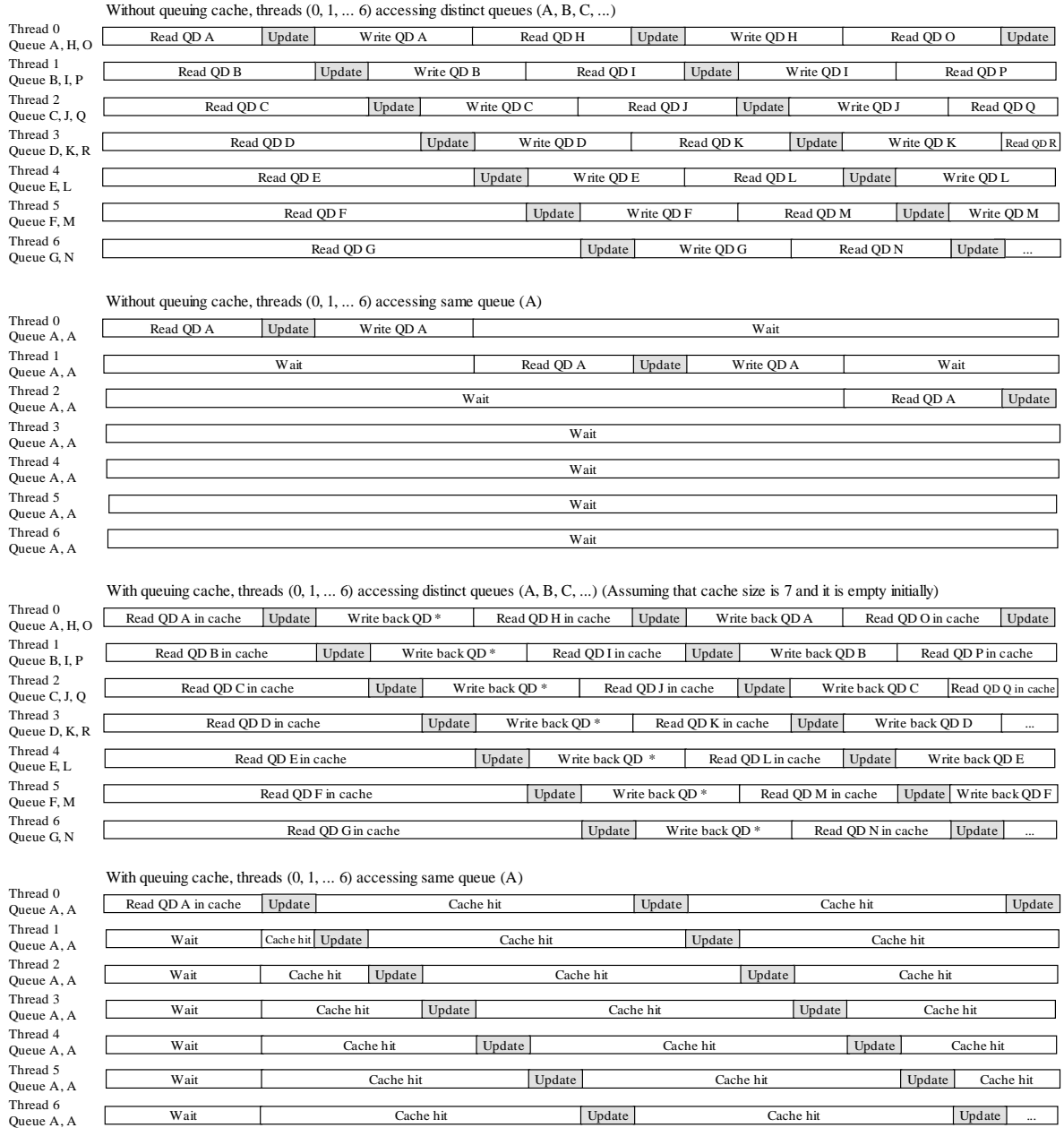


Figure 3: Time line of queuing with and without queuing cache for same queues and different queues

Step 1 takes time c . Steps 2 and 3 can be carried out in parallel and take $\text{Max}(w,r)+bwr$ units of time. Step 4 takes r time units. Step 5 takes time P . With N threads accessing same queue, the throughput will be $X(N) = N/(N*c+\text{Max}(w,r)+bwr+N*r+N*P)$. $N*r$ has been used instead of $N*br$ because Steps 4 and 5 are carried out sequentially and burst read performance can't be achieved because threads are parsing the same queue (link list). When threads access distinct queues, the throughput of a single thread will be $1/(c+\text{Max}(w,r)+bwr+br+P)$, and for M threads, throughput will be $Y(M) = M/(c+\text{Max}(w,r)+bwr+br+P)$, assuming that M is not enough to saturate the memory.

For a total T threads and $M+N=T$, consider the vector $\langle t_1, t_2, t_3, \dots, t_T \rangle$, with constraint that sum of the t_i is T , and t_i is the number of threads operating on each distinct queue. A vector like $\langle 1, 1, \dots, 1 \rangle$ means that all threads are operating on distinct queues. The aggregate throughput for any given vector will be $\sum_{i=1}^T X(t_i)$. The minimum of this function will give the worst case throughput. We have plotted this function and found that its minimum is achieved for any vector of type $\langle 0, \dots, T, \dots, 0 \rangle$ and is equal to $X(T)$. $Y(T)$ will give the upper bound on throughput.

Consider following realistic values for the parameters, $w = r = 80$ ns, $bwr = bw = br = 7.5$ ns, $c = 20$ ns, and $P =$

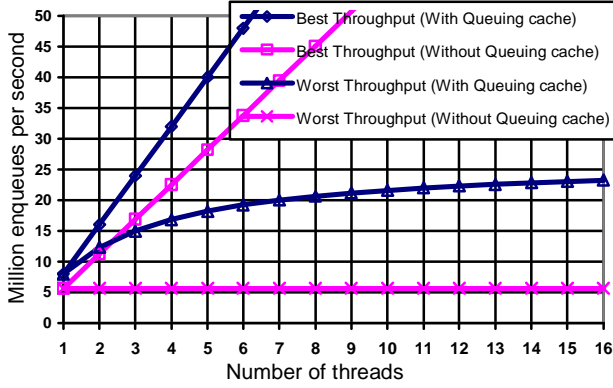


Figure 4: Throughput of a processor with and without queuing cache versus number of threads

10 ns. Figure 4 graphs best and worst-case enqueue throughput for these parameters, both with and without caching, over a range of thread counts. The results in the figure assume that memory bandwidth is not exhausted.

When $T=16$ (i.e., 16 threads), the worst case enqueue throughput without any cache is 5.6 million enqueues per second while the worst case enqueue throughput with queuing cache is 23.3 million enqueues per second, an increase by a factor of more than 4.

It should be noted that this analysis is based on a static view of the system. Precise analyses of cache-based systems must be benchmarked with traffic for long time periods since the history of the cache state affects performance. However, in our analysis we have considered the worst possible scenario, thus we believe that our worst case results are pessimistic.

B. Improved Efficiency

In addition to the throughput benefits discussed, our queuing cache proposal improves efficiency in several ways.

1. Reduced On-Chip Communication

Since the queuing cache handles linking, the processors do not ever fetch the queue descriptors. Without this support at the memory interface, all active queuing threads would require substantially more on-chip bandwidth to access to queue descriptors and fetch link nodes.

2. Reduced Instruction Count on Each Processor

Since the cache implements the queue operations, the processors do not; this is a direct reduction in the time spent utilizing the ALU. As discussed in Section III.A, reducing the ALU utilization, and hence the ratio of compute to I/O times, increases the utility of additional threads. This leads to better utilization of the processor, and, perhaps, fewer processors needed to handle queue management.

3. Centralized Arbitration for Shared Queues

With multithreading, protections must be applied to keep multiple threads from corrupting shared queue state. A queue cache is a logical and efficient mechanism for providing this coherency; far more efficient than using shared locks in memory.

V. COMPARISON TO INTEL'S IXP NP

Intel's second-generation IXP network processors have support for queuing in both the SRAM controller, which holds queue descriptors and implements queue operations, and in the MEs, which support enqueue and dequeue operations. In this section, we describe how queuing works in the Intel IXP2X00 architecture and compare it with our scheme.

The IXP architecture ensures high performance queuing in following ways.

1. It implements enqueue and dequeue operations for cell/packet queues in the SRAM controller, and the ME ISA provides enqueue and dequeue I/O instructions. The SRAM controller internally manages the links in the queues and updates the queue length counters.

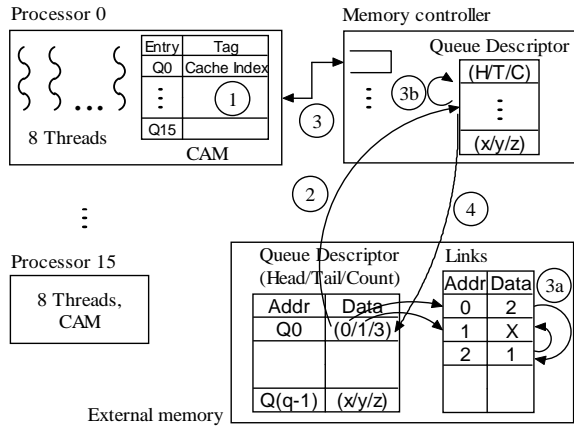
2. The IXP SRAM controller provides on-chip memory to store queue descriptors, called the Q-array. Descriptors are loaded and evicted by threads; that is, the controller provides memory, but cache management is left to the threads. Individual threads manage evictions and write backs.

3. To keep track of which queue descriptors are stored in the memory controller, and where, each ME is equipped with a 16-entry content-addressable memory (CAM). The queue descriptor address (QID) is used to store a 4-bit Q-array tag entry in the cache. A thread performs a lookup into the CAM, using the QID as a key. A hit returns the desired Q-array entry, while a miss returns the LRU entry which can then be evicted to make room for the desired queue descriptor. Thus, the IXP mechanism keeps the data entries in the SRAM controller and the tag entries in the ME. Since the CAM has 16 entries and each processor has 8 threads, a given ME can support two queue operations at a time in all threads, yielding 16 concurrently active queues.

The IXP queuing architecture and the steps involved for an optimized queuing operation are shown in Figure 5.

Thus, three mechanisms in the IXP – the CAM in every ME, the SRAM Q-array, and the queuing specific instructions – together provide effective queuing with good worst-case throughput (when all threads access same queue). This architecture can support enqueues and dequeues at an OC192 rate for a single hierarchy of queues in a single ME (all 8 threads manage two queues each). However, this approach has the following drawbacks not shared by our proposed architecture.

A. Unnecessary Instructions and Software Complexity



- Step 1: Select cache ID (queue descriptor location in memory controller) and store the cache ID and queue ID in CAM.
 Step 2: Cache the queue descriptor (Specific instructions provided)
 Step 3: Perform enqueue/dequeue (Specific instructions provided)
 3c: Links are either made (for enqueue) or parsed (for dequeue)
 3b: Cashed queue descriptor (head/tail/count) is updated internally
 Step 4: Write back queue descriptor to memory (Specific instructions)

Figure 5: IXP queuing architecture

In our scheme, cache management is implemented at the memory interface. The IXP requires software on the ME to implement the cache. Transparent cache management is desirable since it simplifies the program and reduces the number of instructions executed when queuing. Software management holds the promise of flexibility, but a fully associative cache, plus the ability to pin queue descriptors to avoid eviction, is ideal in most cases. In our view, it is better to reduce the ALU resources for each thread and improve throughput via additional multithreading.

B. Unnecessary Communication between Processors and the Memory Controller

For every enqueue and dequeue operation, three distinct commands at three distinct times need to be sent to the memory controller from the ME, namely, a) cache command with queue ID, b) enqueue/dequeue command, and c) write back command. Since the interconnect between the memory controller and the MEs is a bus shared by many units across the chip, reducing such communication is always a benefit. Our proposed scheme, as explained earlier, achieves such a reduction by requiring only one command for each enqueue and dequeue operation.

C. Lack of scalability

One consequence of using an ME CAM to implement the cache index is that *only one ME can access the queues*. There is no way to keep distinct CAMs coherent, so on the IXP all queue operations must be generated by a single ME. This limits the number of threads that can be used for queuing, and, hence, the ultimate throughput of the system. The IXP mechanism

can only scale to faster line rates by increasing both the number of threads per ME, and the size of the CAM. However, as noted earlier, increasing the number of threads is problematic since the IXP offloads queue management to the MEs, keeping the ratio of instructions to I/O cycles high. Our scheme, on the other hand, scales naturally by increasing the cache size and utilizing more threads, from any number of processors.

Our scheme would, in fact, remove the main need for a CAM in each ME. While the CAM can be used for other cache-related purposes, we suspect it would be dropped from the ME microarchitecture if it were not needed for queuing since only one ME can be used for queuing anyway. As the number of MEs per IXP increases, there may well be increasing pressure to economize.

VI. CONCLUSION AND FUTURE WORK

In this short paper, we have proposed a cache-based queue management system for network processors. We described the role of queues in packet processing systems, argued qualitatively and quantitatively that caching queue descriptors can provide a significant benefit to queue management, and contrasted our proposal with the queuing mechanism used in Intel’s second-generation IXP network processors.

In future work, we plan to explore this idea and a few variations in greater detail. For example, in order to support queuing in line cards supporting many 10s of gigabits per second or more, we plan to investigate a cache hierarchy of queuing caches in which each of cluster of MEs shares a first-level queuing cache, which is backed by a shared second-level cache at the memory controller. Additionally, we plan to explore the benefits of similar queue management techniques in high-performance end-hosts.

References

- [1] M. Adiletta, et al. “The Next Generation of Intel IXP Network Processors,” Intel Technology Journal, vol. 6, no 3, pp. 6-18, Aug 2002.
- [2] J. Bennett and H. Zhang, “Hierarchical packet fair queuing algorithms,” IEEE/ACM Transactions on Networking, vol. 5, no. 5, pp. 675–689, Oct. 1997.
- [3] Saleem N. Bhatti and Jon Crowcroft. “QoS-Sensitive Flows: Issues in IP Packet Handling,” IEEE Internet Computing, vol 4, no. 4, pp. 48-57, July 2000.
- [4] J-G Chen, et al. Chapter 14—Implementing High-Performance, High-Value Traffic Management Using Agere Network Processor Solutions, in Network Processor Design, volume 2, Morgan Kaufmann, 2004.
- [5] B. Suter, et al. “Buffer management schemes for supporting TCP in gigabit routers with per-flow queuing,” IEEE Journal on Selected Areas in Communications, vol 17, pp 1159-1169, June 1999.