Journal of Instruction-Level Parallelism 10 (2008) 1-24 Submitted 10/06; published 6/08

Exploiting Locality to Ameliorate Packet Queue Contention and Serialization

Patrick Crowley PCROWLEY@WUSTL.EDU
Sailesh Kumar SAILESH@ARL.WUSTL.EDU
John Maschmeyer JRM5@CEC.WUSTL.EDU
Department of Computer Science and Engineering
Applied Research Laboratory
Washington University
One Brookings Drive
St. Louis, MO 63130-4899 USA

Abstract

Packet processing systems maintain high throughput despite relatively high memory latencies by exploiting the coarse-grained parallelism available between packets. In particular, multiple processors are used to overlap the processing of multiple packets. Packet queuing—the fundamental mechanism enabling packet scheduling, differentiated services, and traffic isolation—requires a read-modify-write operation on a linked list data structure to enqueue and dequeue packets; this operation represents a potential serializing bottleneck. If all packets awaiting service are destined for different queues, these read-modify-write cycles can proceed in parallel. However, if all or many of the incoming packets are destined for the same queue, or for a small number of queues, then system throughput will be serialized by these sequential external memory operations. For this reason, low latency SRAMs are used to implement the queue data structures. This reduces the absolute cost of serialization but does not eliminate it; SRAM latencies determine system throughput.

In this paper *we observe that the worst-case scenario for packet queuing coincides with the best-case scenario for caches*: i.e., when locality exists and the majority of packets are destined for a small number of queues. The main contribution of this work is the queuing cache, which consists of a hardware cache and a closely coupled queuing engine that implements queue operations. The queuing cache improves performance dramatically by moving the bottleneck from external memory onto the packet processor, where clock rates are higher and latencies are lower. We compare the queuing cache to a number of alternatives, specifically, SRAM controllers with: no queuing support, a software-controlled cache plus a queuing engine (like that used on Intel's IXP network processor), and a hardware cache. Relative to these models, we show that a queuing cache improves worst-case throughput by factors of 3.1, 1.5, and 2.1 and the throughput of real-world traffic traces by factors of 2.6, 1.3, and 1.75, respectively. We also show that the queuing cache decreases external memory bandwidth usage, on-chip communication, and the number of queuing instructions executed under best-case, worst-case and real-world traffic workloads. Based on our VHDL models, we conclude that a queuing cache could be implemented at a low cost relative to the resulting performance and efficiency benefits.

---

* An early version of this manuscript appeared in ACM Computing Frontiers 2006.

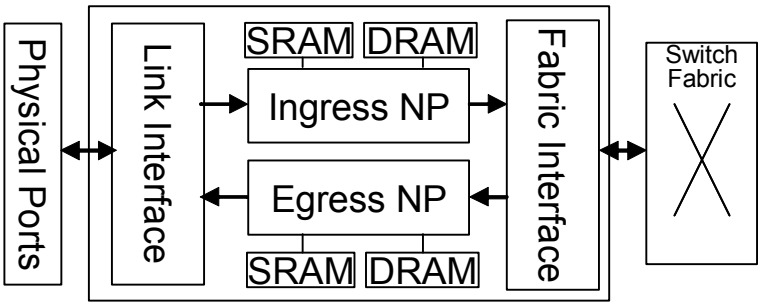

Figure 1: A Network Processor-based router line card.

1. Introduction

Packet queues represent a critical serialization point in packet processing systems. An arriving packet is inserted into a queue based on the router's classification policy, with the assigned queue representing the type of service the packet should receive. Similarly, when a packet is scheduled for transmission, the scheduler removes it from its queue. Queues are implemented as linked lists, and these operations require read-modify-write operations on the queue descriptor that keeps track of the start, end and size of the queue. When multiple queues are simultaneously active, the read-modify-write operations can be carried out in parallel. Modern packet processing systems use coarse-grained parallelism, in the form of multiple on-chip processors and memory controllers, to exploit this situation. However, the most challenging requirement is to provide good performance for consecutive operations on the same queue due to the lack of opportunity for parallelism. The worst-case, performance-limiting scenario in modern packet processing systems arises when there is high contention for a small number of queues.

High-performance systems must support a large number of such queues; therefore queue descriptors are kept in off-chip memory. In high-speed networks, packet inter-arrival times rival memory access times, so SRAMs are used for this purpose. For example, QDR SRAM has an effective access latency of at least 15 to 20 ns, when chip to chip interconnects are accounted for, and the minimum packet arrival time in an OC-768 link is 8 ns. Therefore, when serialization occurs in a small number of queues, the latency to perform read-modify-write operations through off-chip memory will determine performance.

In this paper, we observe that contention for a small number of queues is a form of locality, and is therefore ideal for a cache. We propose the queuing cache, an on-chip, hardware-based cache for chip-multiprocessors that supports queue operations directly. In our evaluation, we compare queuing cache performance to: 1) an unmodified system with no queuing or cache support, 2) a base system augmented with a hardware data cache, and 3) a system with a software-managed queuing cache. We show that the queuing cache provides superior throughput over a wide range of synthetic and real-world workloads, while increasing efficiency by reducing on-chip communication, reducing memory bandwidth, and reducing the number of instructions executed in software.

The rest of this paper is organized as follows. Section 2 provides background on network processor (NP)-based packet processing systems. Section 3 describes the queuing cache as well as other traditional memory system models for packet queuing; the section also introduces an analytical performance model. Section 4 presents our experimental evaluation of the queuing

cache and the alternate models. Section 5 provides an analysis of the results and elaborates queuing cache implementation details. Finally, the paper concludes in Section 6.

2. Packet Processing Systems

The organization of an NP-based router line card is shown in Figure 1. There are variations among line cards, including those that augment NPs with queue management chipsets [17], but the overall organization and use of SRAM and DRAM for queuing, as described below, are common to all variations. In both the ingress and egress directions, an NP sits between the switch fabric and the physical interface. The switch fabric carries traffic between line cards. The physical interface may consist of a single link, such as 10 Gb Ethernet or SONET, or a collection of slower links, such as 10/100 Mb Ethernet or DSL.

To increase the number of instructions and memory operations that can be applied to each packet while meeting a target line rate, NPs are typically organized as highly-integrated chip-multiprocessors. For example, Intel's IXP2800 [1] 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 queuing. 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 one or more processors.

2.1. Packet Queues

Packet queues are used to provide packet scheduling, QoS, and other types of differentiated services to packet aggregates. Many routers use a three-level queue hierarchy, where the first represents physical ports, the second represents classes of traffic and the last level consists of virtual output queues. Each ingress NP maintains a queue for each output port to eliminate head-of-line blocking; each of these output port queues has a number of class queues associated with it in order to enable service differentiation and QoS; each of these class queues consists of per-flow virtual output queues which allow individual flows to be shaped, e.g., by throttling unresponsive flows that are causing congestion.

Each incoming packet is enqueued into some virtual queue, and the status for the corresponding class and physical queues are updated to record the activity. A similar 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 during each packet arrival/departure period. Since both operations involve updates to shared queues, serialization can occur.

In order to provide good memory utilization, virtual queues are typically kept in a linked list data structure [17]. Port and class queues, however, are only kept in a linked list data structure when the selection policy for class and virtual queue is ring-based. Round robin or weighted deficit round robin [18] are examples of ring based selection policies, where the 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. For example, many packet scheduling algorithms use

queue occupancies as inputs. For this reason, some architectures pass every enqueue and dequeue command 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.

2.2. A Packet Processing Pipeline

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 maintain good throughput despite relatively high memory latencies. The processing pipeline generally consists of the following tasks, each typically mapped to its own processor(s).

1. **Packet assembly**. Several interfaces deliver packets in multiplexed frames or cells across different physical ports.
2. **Packet classification**. Incoming packets are mapped to a queue.
3. **Admission control**. Based on the QoS attribute of the queues, such as a maximum size, packets are either admitted or dropped.
4. **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.
5. **Scheduling and dequeue**. The scheduler selects the queues based on the QoS configuration, and then a packet is dequeued and transmitted.
6. **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.

2.3. Queue Operations and Parallelism

Both the queue descriptors, consisting of head and tail pointers and the queue length, and the linked lists (i.e., 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 implicitly 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. 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 and all operations are serialized, 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.

2.4. Related Work

The importance of packet queuing in routers and switches has been motivated and discussed in the literature [14],[15],[16]. The majority of research on high-performance memory systems for networking has focused on optimizing packet buffer memory bandwidth. Researchers from Stanford University have proposed several schemes to buffer packets into DRAM at very high speeds [5],[6],[7],[10]. A group at ICS FORTH has proposed several ASIC based architectures to

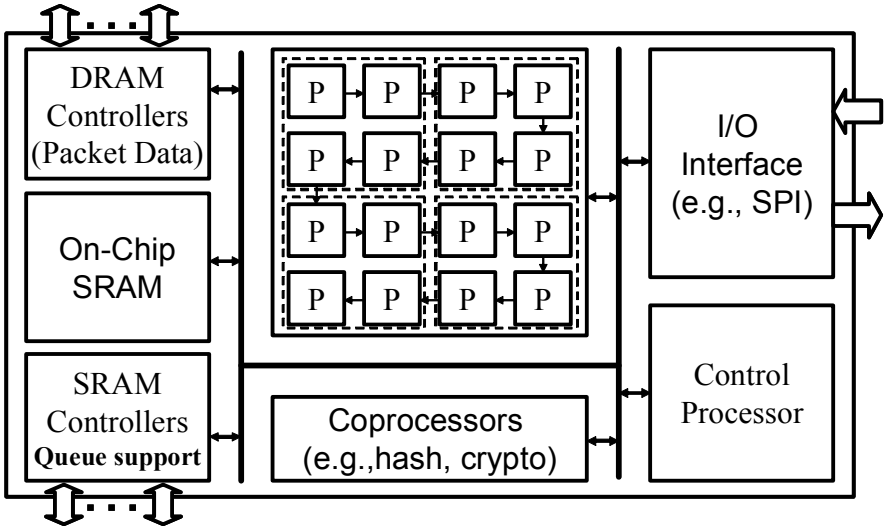

Figure 2: Structure of the NP architecture with shared interconnect for memory and other resources. Each processor (P) is assumed to be multithreaded. The queuing subsystem involves processors, DRAM, and SRAM. HW-based queuing support would be integrated at the SRAM controllers.

efficiently perform the packet buffering and manage large number of queues at very high speeds [13].

Another branch of related work has focused on maximizing the utilization of available DRAM technologies. McKee et al. have proposed a number of mechanisms to maximize DRAM bandwidth in streaming computations [11]. Rixner et al. have proposed a related performance enhancement via efficient memory access scheduling [8]. Techniques to effectively reduce DRAM latency have also been proposed [9]. Hasan et al. have shown an approach to efficiently utilize the available DRAM bandwidth on a network processor [4].

The challenges of packet buffer design and DRAM utilization are important, but orthogonal issues. In this work, we directly improve the performance of packet queue data structures whose worst-case performance limits the performance of existing programmable packet processing systems.

The software-controlled Q array structure in the Intel IXP 2XXX family of NPs is similar to the software-managed queuing cache discussed in Section 3.3. In the IXP, the software cache is managed and kept coherent by using a CAM in a processor (i.e., a micro-engine). The CAM keeps track of the mapping between queue descriptor ID and cache entry, as well as the location of the LRU entry for eviction purposes. As shown in this paper, using a hardware-managed queuing cache in the memory controller greatly increases performance while also improving the overall efficiency.

3. Memory System Models

In this section, we will describe the structure of the queuing cache and its alternatives—a base system with no cache or queuing support, a data cache, and a software-controlled queuing cache—as well as how queue operations are carried out on each organization. We conclude this presentation of memory models with a brief analysis of the alternatives.

System type	System parameter	Value
Base CMP parameters	Interconnect clock frequency	400 MHz
	Memory access time (round trip)	40 ns
	Interconnect bus width	8 words
	Processor clock frequency	1 GHz
	Interconnect delay (round trip)	40 ns
	Synchronization server delay	0 ns
	Memory clock frequency	200 MHz
	Total instruction cycle time	10 ns
	Cache access time	2.5 ns
Software cache	CAM access latency	5 ns
	Total instruction cycle time	35 ns
Queuing cache	Total instruction cycle time	5 ns

Table 1: Parameters chosen for the base multithreaded CMP queuing system model.

3.1. Architecture and Memory Organization

Figure 2 shows the structure of our basic system. It is a highly integrated system, featuring multiple processors, each multithreaded, and memory channels all sharing a bus-based interconnect. This general organization is common among chip multiprocessors and network processors [2].

Since we are modeling a shared memory in a multiprocessor system, memory consistency is a critical issue. Specifically, our requirement is that requests for a particular queue occur in the correct order without corrupting the queue descriptor data structure.

The queuing cache synchronizes access to a given queue automatically. For the other memory organizations, we model a zero-cost synchronization mechanism that assures ordered, coherent access to all queues. A real system would require some sort of software synchronization to maintain consistency, but these costs are not accounted for in the non-queuing cache models. Adding a realistic synchronization mechanism would greatly increase the complexity of the alternate models. As we will see, the queuing cache provides superior performance despite this disadvantage.

An enqueue operation in the base model involves three memory references, a) initially the queue descriptor is read, b) then the arriving packet is linked to the queue's tail, and c) finally the updated queue descriptor (tail) is written back. The last two references can be carried out concurrently, as they are independent writes. A dequeue operation also involves three memory references, a) initially the queue descriptor is read, b) then the next node of the head is read (which will become the new head), and c) finally the updated queue descriptor (head) is written back. All three references, in this instance, must be carried out sequentially.

The time needed to complete a given memory reference depends on two factors: 1) the external memory latency and 2) the latency of the shared interconnect between the processor and the memory interface. This second factor is considerable and can rival or exceed external memory latency in highly-integrated CMPs, even in unloaded systems (i.e., those without interconnect contention).

3.2. General-Purpose Data Cache

A data cache can be placed between the queue descriptor memory hierarchy and the processors. It provides no direct support for queuing operations, but can shorten the load/store latency for operations applied to cached queue descriptors. Figure 4 illustrates this effect, as well as the corresponding operations for all the implementations. Since there is little re-use in the linked list memory, the cache only needs to hold the recently accessed queue descriptors. In addition, for good performance it must be able to service multiple requests concurrently, and hence the cache must be lock-up free [3]. The data cache improves queuing performance as long as there is some locality in the queue descriptor accesses. Since a hit does not require an external memory access, the cost of serialization is reduced. However, the interconnect latency persists.

3.3. Software-controlled Queuing Cache

The cost of serialization can also be reduced with the addition of a small linked list processor (also referred to as a queuing engine) and a software-controlled cache between the queue descriptor memory hierarchy and the processors. This is the approach used in Intel's NPs. The linked list reads and writes will be handled internally by the cache, resulting in interconnect latency savings. Each thread sends an enqueue or dequeue command to the cache and the cache performs the link list read or write followed by the queue descriptor update. However, since the cache is software controlled, before issuing an enqueue or a dequeue command, threads must ensure that the queue descriptor is cached. If it is not cached, then threads need to issue commands in order to evict an entry and bring in the appropriate queue descriptor to the cache. Thus, threads issue two commands on a miss and one command on a hit, in order to service an enqueue or a dequeue request. In addition, the synchronization must be carried out by the threads.

3.4. Queuing Cache

A queuing cache is a small fully associative cache with tightly coupled queuing engines. It reduces the interconnect latency by enabling the threads to issue only one command for an enqueue or dequeue operation. The Queuing cache internally manages the queue descriptor cache and performs the appropriate queuing operations on a hit or miss. Furthermore, a queuing cache operates in parallel and accepts commands from multiple threads (i.e., it is also lock-up free). When multiple requests are destined to a single queue, it ensures that all requests are serviced without any contention and in the correct order. Thus, individual threads need not maintain synchronization or packet ordering. We consider a particular queuing cache design and its implementation costs in Section 5.

3.5. Summarizing the Queuing Sub-systems

The four alternatives are best differentiated by considering how they 1) reduce the number of off-chip memory accesses, and 2) reduce the number of operations sent across the shared on-chip interconnect.

The three *cache-based schemes* all *reduce external memory accesses*, and thereby reduce latency, on a hit—this is the traditional benefit of caching. The two schemes that *employ queuing engines*, the queuing cache and software managed queuing cache, *reduce the number of memory commands sent* by sending queue operations rather than the memory references that implement them. The software managed cache, however, must send cache management operations on misses, so the benefit is reduced.

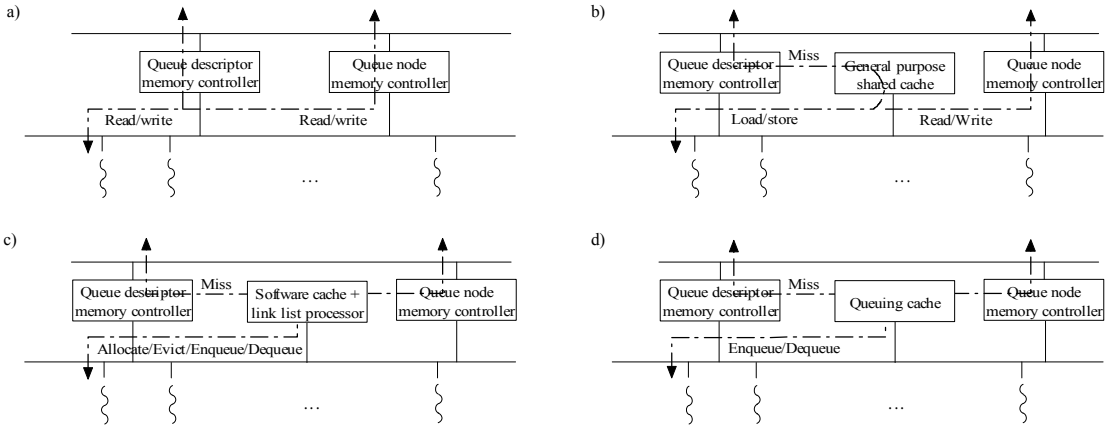

Figure 3: High level schematic diagram and communications involved in the queuing subsystem a) without any cache, b) with data cache, c) with software-controlled cache and linked list processor, and d) queuing cache. The communication across the shared interconnect in order to perform an enqueue or dequeue is also highlighted.

System	Enqueue operation	Dequeue operation

System with /without data cache	enqueue(pkt_ptr, queue) {
	// First grab the lock
	grab_lock();
	// Read in the queue tail
	tail = load_tail(queue);
	// Link old tail to new pkt
	wr_link_mem(tail, pkt_ptr);
	// Update and store tail
	tail = pkt_ptr;
	store_tail(queue, pkt_ptr);
	}
		pkt_ptr dequeue(queue) {
		// First grab the lock
		grab_lock();
		// Read in the queue head
		head = load_head(queue);
		// Read next pkt from head
		new_head = rd_link_mem(head);
		// Store the new head
		tail = pkt_ptr;
		store_head(queue, new_head);
		return head;
		}

System with software controlled cache	enqueue(pkt_ptr, queue) {
	//wait until queue is free via CAM
	grab_lock();
	//hit => my_entry is cache location
	my_entry = lookup_cache();
	if (my_entry = -1) { // Miss
	// Evict LRU entry from cache
	my_entry = lru_cache;
	evict_from_cache(my_entry);
	// Set the in use bit
	set_inuse_bit(my_entry);
	// Load tail in LRU location
	load_tail(queue, my_entry);
	}
	enqueue(pkt_ptr, my_entry);
	}
		pkt_ptr dequeue(queue) {
		//wait until queue is free via CAM
		grab_lock();
		//hit => my_entry is cache location
		my_entry = lookup_cache();
		If (my_entry = -1) { // Miss
		// Evict LRU entry from cache
		my_entry = lru_cache;
		evict_from_cache(my_entry);
		// Set the in use bit
		set_inuse_bit(my_entry);
		// Load head in LRU location
		load_head(queue, my_entry);
		}
		head = dequeue(pkt_ptr, my_entry);
		return head;
		}

System with queuing cache	enqueue(pkt_ptr, queue) {
	// Implemented in queuing cache
	enqueue(pkt_ptr, queue);
	}
		pkt_ptr dequeue(queue) {
		// Implemented in queuing cache
		return dequeue(queue);
		}

Table 2: Pseudo-code of the operations performed at each thread in the four queuing subsystem models.

The differences in on-chip and off-chip communication can be seen in Figure 3, where the queue operations and their resulting memory interactions are illustrated for each implementation. In order to compare the operational differences in concrete terms, Table 2 provides pseudo-code for each subsystem implementation, for both enqueue and dequeue operations. The operational timelines shown in Figure 4 illustrates the individual contributors to enqueue and enequeue latency.

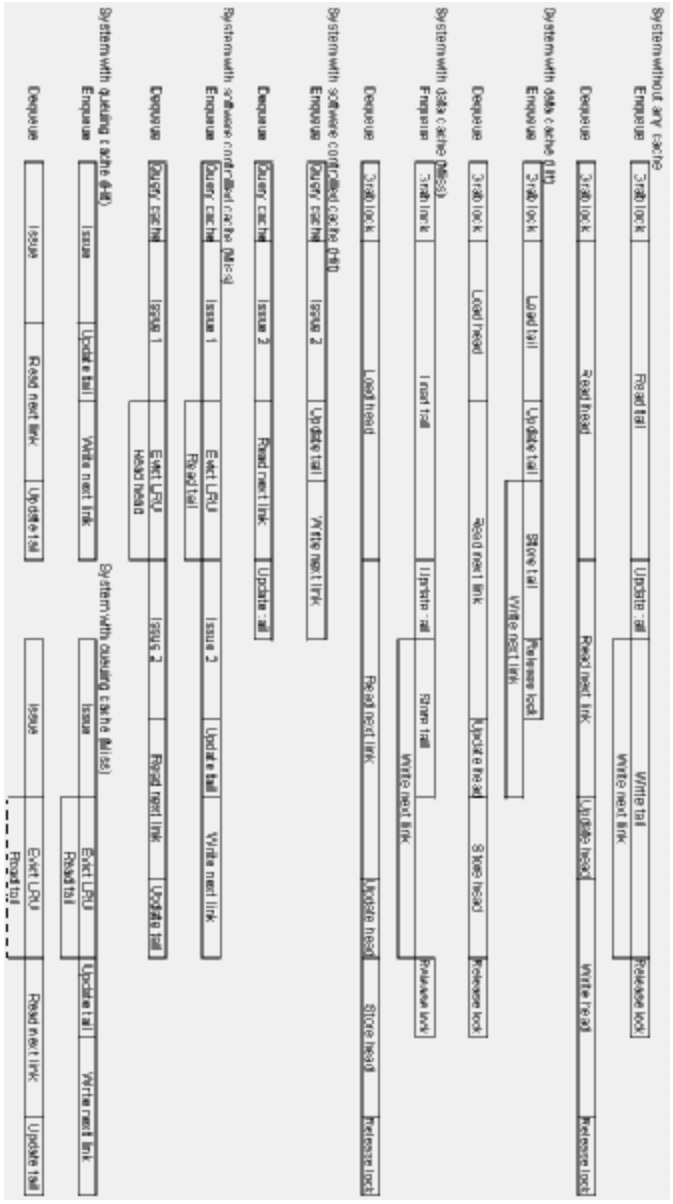

Figure 4: Illustration of the time line for servicing enqueue and dequeue requests in, a) system without any cache or explicit support for queuing, b) system with general purpose data cache for queue descriptors, c) system with software controlled cache and linked list processing capability, and d) system with queuing cache.

Characteristics	No caching	Data cache	Software-controlled cache	Queuing cache
Queue contention	Threads resolve cooperatively. May need hardware support for synchronization	Same as previous	Threads resolve contention with the aid of fully associative cache array	Internally resolved by the queuing cache
Packet order	Threads must ensure correct packet service order. Adds complexity and thread stalls	Same as previous	Same as previous	Packet service order is always ensured by the queuing cache
Instruction count for queuing	~30. May vary based on ISA	~30. May vary based on the ISA	~60. May vary based on the ISA	3-4
Communication at the shared inter-connect	Read QD Write QD Read/Write link list	Same as previous	Issue 1 (evict and allocate) Issue 2 (enqueue/dequeue)	A single Issue (enqueue/ dequeue)
QD bandwidth	No reduction	Reduction = hit rate	Same as previous	Same as previous

Table 3: Table summarizing the characteristics of each of the four queuing sub-system models.

3.6. Initial Comparative Analysis

Table 3 compares each of the sub-system implementations according to their treatment of a selection of key characteristics: queue contention resolution, maintaining packet order, instruction overhead for queue management, on-chip communication bandwidth, and off-chip bandwidth.

We have also developed an intuitive model that can be used to analytically determine the peak throughput of the various queuing sub-system models. In this model, peak throughput is achieved when enough threads are used to saturate memory bandwidth, and, hence, additional threads will offer no benefit. Our analytical method is based on the concept of time gained and lost. Time refers to the real processing time of a single thread. Time lost refers to the total real processing time lost by a thread due to stalls. Time gained implies the total real processing time saved by a thread due to a cache hit. We will now quantify the time lost and gained.

Theoretically, the highest throughput is achieved when every thread accesses distinct queues and the time required to perform a single enqueue or dequeue is T_e/n or T_d/n, respectively. T_e and T_d, respectively, are the individual enqueue and dequeue service times, and n is the total number of threads. Note that these times will vary across different queuing sub-systems. When there is complete contention, all but one contending thread get stalled. The time lost during such stalls will be, $T_l = \sum_{i \in \text{all packets}} (T_{op} - t(i)) * f(t(i))$, where T_{op} is the time required to service the current enqueue or dequeue operation, $t(i)$ is the time since the last operation on the same queue, and, $f(x)$ is a unit step function with a step at T_{op}. Thus $f(x)$ is one when x is greater than T_{op}, and zero elsewhere. That is, if the next operation for queue arrives before the current one is complete, stalls will result.

A cache hit saves the processing time of a single thread and the savings amounts to $T_{e(miss)} - T_{e(hit)}$ for an enqueue and $T_{d(miss)} - T_{d(hit)}$ for a dequeue. Note that the time saved during enqueues and dequeues is the same in our context. The time saved, T_s, in a cache based system with a hit

rate of h is given by $T_s = \sum_{i \in \text{all packets}} (T_{miss} - T_{hit}) * h$. Thus, the time needed to service N_e enqueues and N_d dequeues in a system is given by $\frac{N_e * T_e + N_d * T_d}{n} + T_l - T_s$.

In the next sections, we will show how the time lost due to serialization and the time gained due to cache hits influence the system throughput. In Section 5, we will revisit this model with insights gained from the experimental results.

4. Experimental Evaluation

In this section, we evaluate the performance of each queuing sub-system under a variety of synthetic and real-world workloads. Since our goal is to measure queuing performance, queuing subsystem remains the bottleneck. Our aim is to show which queuing organization provides the best performance and efficiency.

4.1. Simulation Methodology

We use a mix of behavioral and RTL-style VHDL to simulate our system and memory models. RTL was used to model the control and logic details of the queuing cache and other units, and behavioral VHDL was used to model processor activity, interconnect paths and arbitration, and memory structures. We have built a base CMP model with hardware support for multithreading in behavioral VHDL and developed three variants of queuing sub-systems around it.

4.1.1. Base Multithreaded CMP model

The parameters chosen for our base CMP system are shown in Table 1. These parameters closely approximate those found in Intel's IXP NPs. In the data cache model, we add a cache with support for the following parameters: associativity, line size, capacity, number of miss status holding registers (MSHRs), optional write-allocation, write-through or write-back, and LRU or random replacement [20]. In the software controlled cache model we have attached the cache array and queuing engine at the memory controller. In the queuing cache model, we have added the RTL design of the queuing cache at the memory controller.

In most experiments, the queuing cache and the software-controlled cache were configured with 32 fully associative entries and 32 queuing engines. However, the data cache was configured to be 2-way set associative with 128 words. This is because this data cache configuration had a) the best performance to complexity ratio, and b) an area footprint equivalent to the queuing cache and software controlled cache [12].

4.1.2. Synchronization

As described in Section 3, synchronization between queues is necessary when performing parallel operations in order to prevent packet reordering and to keep queue data structures coherent. Of the structures modeled in this paper, only the queuing cache accounts for the overhead costs of synchronization, since the associated logic was implemented inside it. The other models would require some kind of software-based synchronization involving locks. Our other models do not implement these software mechanisms, but if they did the net effect would be greater operation latency and communication bandwidth. By not including the costs of synchronization in the non-queuing cache solutions, we over-estimate the performance and efficiency of these models. As we will see, the queuing cache gives strong relative performance despite this disadvantage.

Workload type	Notation	Distribution of λ across queues	Description
Logarithmic	Lk	Exponential	The weights of each queue were exponentially distributed. Thus, $\lambda_i = k^* \lambda_{i-1}$
Uniformly random	R	Uniform	Weights of all queues were same, λ.
Strictly uniform	U	n.a.	The arrival process, instead of being Poisson, is a round robin one, in which queues send packets in a round robin order. It results in no temporal locality.
Set dominating	SDn_k	Two sets of uniform	A set of n equal priority queues had captured k% of the bandwidth and remaining (N-n) queues took the remaining bandwidth. Thus, $n^* \lambda_1 = k/(100-k)^* (N-n)^* \lambda_2$, where λ_1 and λ_2 are the mean inter-arrival time of first and second set of queues respectively.
Single dominating	SD1_k	Two sets of uniform	This is the limiting case of the previous workload, wherein the value of n is one.

Table 4: Description of the synthetic workloads, their notation and the distribution of parameters.

4.1.3. Workloads

When performed collectively by multiple threads, queuing throughput can be highly sensitive to the workload. As we have noted, system throughput is determined by the fraction of current operations that can proceed in parallel. To evaluate a range of possibilities, we have gathered a collection of synthetic and real-world packet traces. Each trace has a different concentration of packets to queues.

For real-world workloads, we use two traces: a network-edge trace from NLANR [22] that consists of traffic recorded on an OC-3 link between a University and the Internet, and a core Internet trace from CAIDA [21] that consists of traffic taken from an OC-48 link connecting two backbone routers. Traffic is mapped to the queues by applying a hash function on the packet header.

Our synthetic traces were designed to capture a wide range of temporal locality in queue references. In order to generate various synthetic traces, we modeled a packet arrival and departure server. At the arrival server, packets arrived for each queue according to a Poisson process [19]. In order to model the varying rates of each queue, we assign each queue a different Poisson process parameter[1], λ. The departure server modeled an approximate deficit round robin

[1] $1/\lambda$ is the mean of the exponential random variable of the Poisson process.

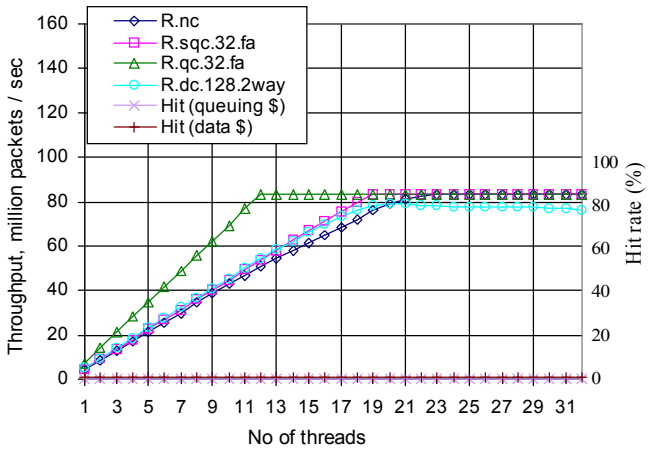

Figure 5: Plots throughput versus number of threads under a uniformly random workload. The hit rate for cache-based systems is also shown.

selection[1] [18], wherein a queue with a higher rate and backlog sends not only more packets but is also likely to send them in larger bursts. Our sets of synthetic workloads are described in Table 4.

4.1.4. Description of the Notation in Graphs and Tables

We use the following notation for each experimental curve: *Workload_type.System_type.Cache_size.Cache_type*. The notation for *Workload_type* is described in Table 4. *System_type* is, a) **sqc** for software-controlled cache, b) **nc** for base system, c) **dc** for data cache, and d) **qc** for queuing cache. Thus, *SD1_70.sqc.32.fa* indicates that a) the traffic type is single dominating queue taking 70% of the link capacity, b) system is software-controlled cache, c) the cache array size is 32, and d) it is fully associative.

4.2. Results with Uniformly Random Traffic

Figure 5 shows the queuing throughput, in millions of packets per second, of each queuing sub-system organization on a uniformly random workload. This situation corresponds to the best-case scenario in a multithreaded system without caches, in which all operations can operate in parallel. In this workload, there are a large number of active queues, and packets are spread equally. Thus, there is no queue locality or contention, as reflected by the low, near-zero hit rates. All models saturate at slightly over 80M operations per second. The queuing cache has a greater slope, however, and achieves peak performance with only 12 threads as compared to the 18-20 threads required by the others.

This result shows that the queuing cache is more efficient, i.e. needs fewer concurrent operations, in achieving peak performance. We also note that this performance level is peak performance achievable in a system without a cache.

[1] Deficit round robin (DRR) is a class of fair queuing algorithm and is widely used commercial routers.

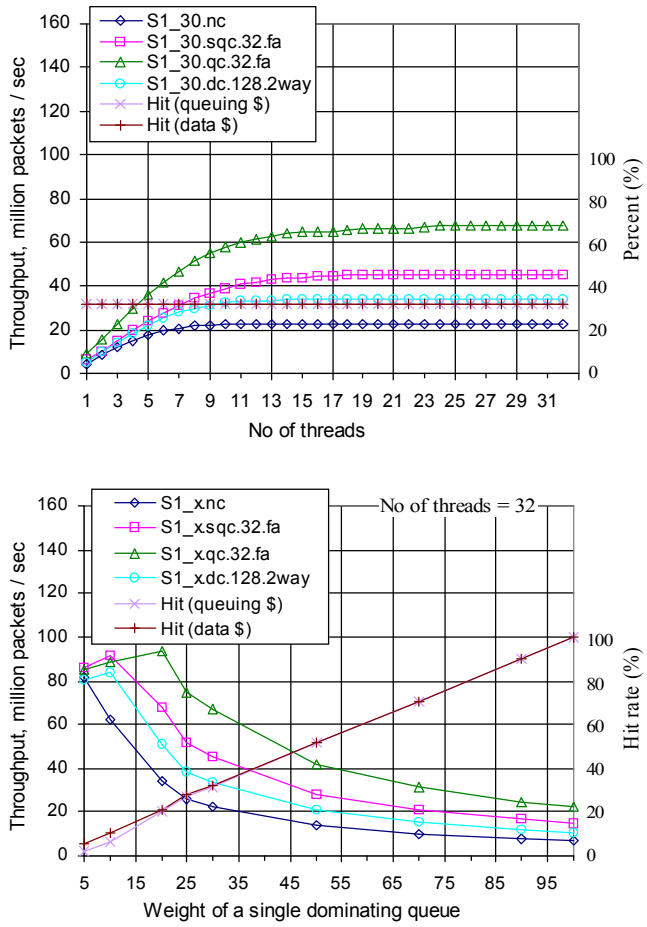

Figure 6: a) Top. Plots throughput versus number of threads under traffic with a single dominating queue utilizing 30% of the link capacity. b) Bottom. Plots throughput versus weight of a single dominating queue for systems with 32 threads. This simulates gradually increasing locality. We note that small weights improve the performance of cache-based systems and as locality increases, serialization degrades the performance.

4.3. Results with Weighted Random Traffic

4.3.1. A Single Dominating Queue

Figure 6a reports throughput for a trace consisting of traffic dominated by a single queue that carries 30% of all packets, with the remaining traffic spread uniformly over a large number of queues. This scenario presents a moderate amount of *narrow* locality, since 30% of all packets target a single queue. Therefore we see higher hit rates, around 30%, and more contention. Clearly, this locality is dominated by the costs of serialization, since performance for all models is worse than that of the uniformly random case. Here, the queuing cache achieves superior performance due to its shorter serialization path.

Figure 6b reports the throughput with 32 threads as the weight of the single dominating queue varies from 5% to 100% of the total traffic. As can be seen, the cache-based models see initial

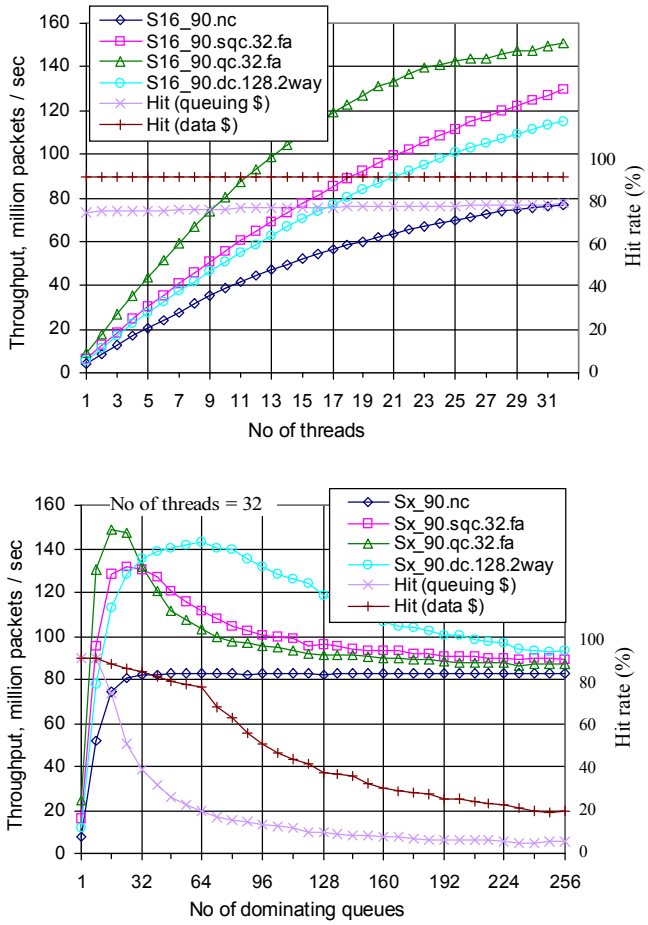

Figure 7: a) Top. Plots throughput versus number of threads under traffic with a set of 16 dominating queues, which take 90% of the link capacity. b) Bottom. Plots throughput versus number of dominating queues for systems with 32 threads and the dominant queues take 90% of the link capacity. This simulates gradually decreasing locality. We see that as the number of dominant queues approaches the cache size, performance gets maximized and afterwards, as the hit rates fall, the performance approaches to the memory bandwidth.

benefit as the weight increases, rise to a peak, and then eventually descend to the worst-case performance levels. The initial benefits are due to the increased hit rates while the eventual decline is due to serialization of traffic at a single queue. The queuing cache has the highest peak, and maintains it over the widest range of weights. But locality ceases to be a benefit for the queuing cache beyond a weight of 20%, whereas the other cache-based models lose the benefit of locality at around 10%. As expected, hit rates increase with the queue weight.

4.3.2. A Set of Dominating Queues

We now consider workloads that extend the locality from one queue to a set of queues. Figure 7a shows throughput when a set of 16 dominating queues accounts for 90% of all traffic. As expected, hit rates are high, between 75% and 90%, and the cache-based models achieve performance greatly in excess of the memory bandwidth limit. This scenario represents a broader

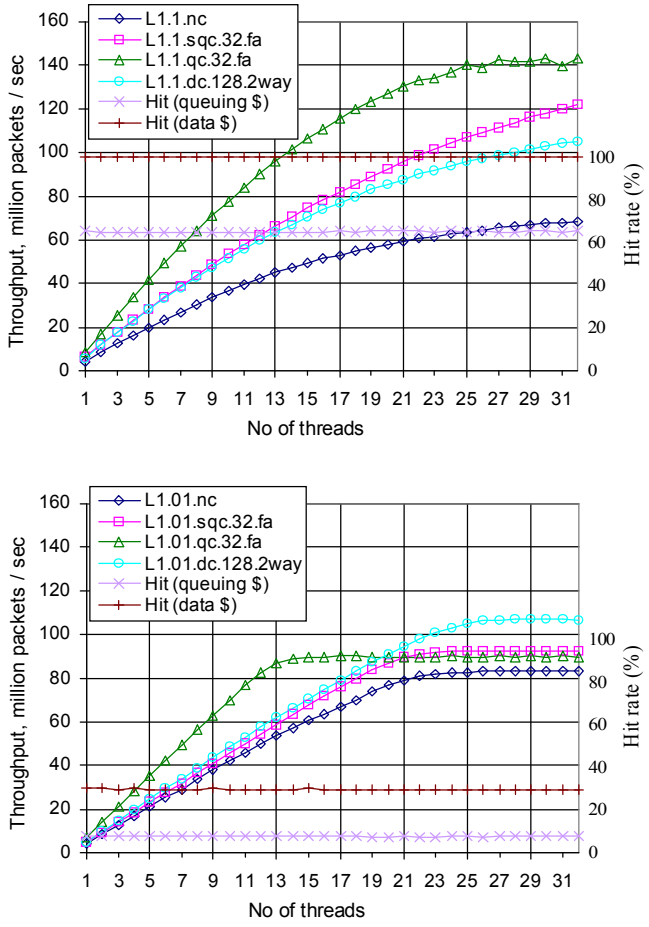

Figure 8: a) Top. Plots throughput versus number of threads under traffic with exponentially weighted queues and exponent of 1.1. b) Bottom. Plots throughput versus number of threads under traffic with exponentially weighted queues and exponent of 1.01. It is evident that traffic with an exponent of 1.1 results in higher hit rates and hence higher performance, because the weight is distributed such that there is broader locality.

form of locality, where 16 different operations are concurrently active on average. The queuing cache achieves greater performance with fewer threads due to its greater efficiency and lower operation latency. It achieves a throughput of 150 M operations per second, representing improvement factors of 1.15 and 1.30 over the software queuing cache and data cache, respectively.

Figure 7b reports results for 32 threads as the number of queues in the dominating set (90% traffic) ranges from 1 to 256. The graph shows that all cache-based models increase with dominating set size until the cache capacity is met: 32 for the queuing cache and software queuing cache, whereas the larger data cache begins its decline at 64. By examining the hit rates, it can be seen that throughput tracks hit rate precisely. The larger data cache maintains a higher hit rate, and thus maintains higher throughput longer. All models eventually converge on their uniformly random performance levels.

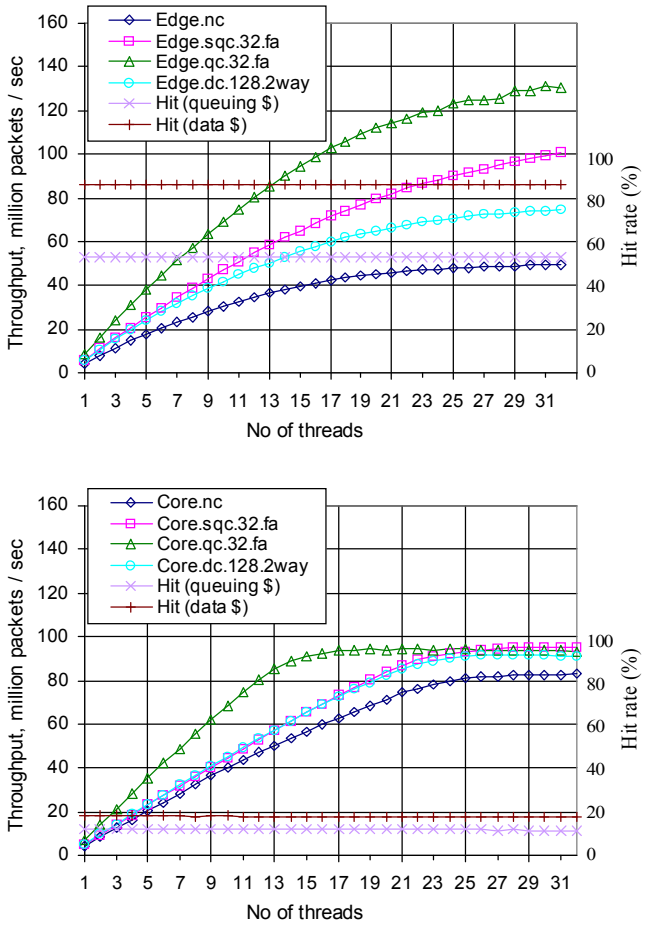

Figure 9: a) Top. Plots throughput versus number of threads under real-world traffic collected at an edge router. b) Bottom. Plots throughput versus number of threads under real-world traffic collected at the core router. It is evident that the edge traffic has a higher degree of temporal locality and hence better hit rates and performance.

4.3.3. Exponentially Distributed Queue Weights

Figure 8 reports throughput when queues have exponentially distributed weights. We use exponentially distributed weights because they provide a means of weighting queues more realistically than previous traces. As can be seen from the performance plots, the narrower exponential distribution shown in Figure 8a results in higher hit rates and throughput than does the wider distribution shown in Figure 8b. The first distribution has a moderate amount of broad locality; the second spreads the weight over more queues, therefore the larger data cache sees better performance by capturing a slightly larger set of active queues.

4.4. Real-World Workload Results

Figure 9 shows throughput for the edge and core traces respectively. As expected, the edge trace shows moderate locality and hit rates, and hence high-performance in cache-based systems. In fact, the throughput results are similar to those seen in the narrow L1.1 exponential synthetic

trace. On edge traffic with 32 threads, the queuing cache achieves throughput of 130 M operations per second, representing improvement factors of 1.3 and 1.75 relative to the software queuing cache and data cache, respectively.

The core trace has little locality, low hit rates and, hence, low throughput. As expected, this trace features many active queues. Therefore, performance is similar to, but slightly better than, the uniformly random case. Under this workload, the cache-based models all converge between 90-95M operations per second. Once again, the queuing cache is more efficient in achieving peak performance. It saturates at 17 threads, compared to 25 for the other cache-based models.

4.5. Tuning Cache Parameters

We have performed experiments on several configurations of the data cache. We have changed a) the eviction policy from LRU to random, b) the allocation policy from write allocate to no-allocate, c) the write policy from write back to write through, and d) line size of the cache. We didn't notice more than a single percent point change in the overall system performance with the first two changes. The third change, namely the write policy actually deteriorated the performance in the setup where the memory had a single shared read and write bus. This is because write through policy performs redundant writes, which if not performed, could have been used for reads. We have also observed that overall performance deteriorated as we increased the cache line size. This is due to the lack of spatial locality between queues. Thus, an increased line size results in extraneous memory accesses that fetch entries that are not likely to be used. Therefore, we conclude that the best cache configuration for the queuing operations under network workloads is a single word per line and write back policy. Allocation and eviction policy don't have significant impact on performance.

4.6. Summary

In the preceding sections, the benefit of caching queue descriptor data has been shown. To summarize these caching benefits, we make the following points.

When contention for a queue is high, keeping the queue descriptor in an on-chip cache shortens the serialization path. Therefore, worst-case performance is improved. Of the models we considered, the queuing cache had the shortest serialization and therefore the best worst-case performance. It improves throughput by factors of 3.1, 1.5, and 2.1 over the system with no cache, with a software-controlled cache and with a data cache, respectively.

When the number of active queues is much greater than the number of queue entries, then caching provides little benefit. However, the queuing cache improves efficiency since it achieves peak performance with the fewest number of threads, reaching peak performance with only 12 threads, as opposed to the 18-20 required by the other organizations.

When the number of active queues is greater than one but not much greater than the cache capacity, then this broad locality will greatly increase throughput. In effect, each cache entry can support a concurrent operation; this provides operation throughput above that which is achievable based on external memory bandwidth. As the number of active queues exceeds the cache capacity, the request sequence appears more random and misses begin to dominate. When the number of active queues is close to the cache size, the performance of the queuing cache peaks at approximately 150M packets/second. As the number of active queues increases, the throughput approaches the 90 M packets/second value seen in uniformly random traffic.

Model	Enqueue Latency	Dequeue Latency	Off-chip Bandwidth	On-chip Bandwidth	Instructions
Queuing Cache	$T_{e(hit)}$ = 80 ns $T_{e(miss)}$ = 120 ns	$T_{d(hit)}$ = 80 ns $T_{d(miss)}$ = 120 ns	(1-*h*)*2*T*+*T*	T	3-4
Software Queuing Cache	$T_{e(hit)}$ = 120 ns $T_{e(miss)}$ = 200 ns	$T_{d(hit)}$ = 120 ns $T_{d(miss)}$ = 200 ns	(1-*h*)*2*T*+*T*	3*T*	60
Data Cache	$T_{e(hit)}$ = 130 ns $T_{e(miss)}$ = 170 ns	$T_{d(hit)}$ = 170 ns $T_{d(miss)}$ = 210 ns	(1-*h*)*2*T*+*T*	3*T*	30
No Cache	T_e = 170 ns	T_d =250 ns	3*T*	3*T*	30

Table 5: Quantitative summary of the characteristics of the four queuing sub-systems.

5. Analysis and Discussion

In this section, we discuss additional quantitative metrics that can be helpful in evaluating a packet queuing memory system, including: workload characterization, operation latency, external memory bandwidth, packet queuing instruction count, and on-chip communication bandwidth. Table 5 summarizes many of these metrics.

5.1. Input Characterization

The input traces can be characterized by the inter-arrival times, where we refer to the inter-arrival time as the time elapsed to service all other requests between two requests for the same queue. Serialization occurs only when the inter-arrival time is less than the enqueue or dequeue service time of a single thread, which we will call the *loss threshold*. In a fully associative cache, hits are guaranteed to occur when the number of inter-arriving packets is less than the cache size, which we will call the *gain threshold*. Above this threshold, misses can occur resulting in no time gain. Furthermore, misses are more likely to occur as the inter-arrival time increases.

By plotting the distribution of inter-arrival times for a given input trace, the performance of a queuing system can be estimated. For example in Figure 10, for the single weighted queue trace, ninety percent of the inter-arrival times are less than the service time of a single packet, indicating that serialization is occurring. On the other hand, the majority of inter-arrival times for the random and core traces are above the gain threshold, and therefore a cache gives little benefit. A large percentage of the inter-arrival times for the trace in which 8 queues are heavily weighted lie between the loss and gain thresholds, resulting in large time gains and small losses. The exponential and edge traces have about 50 percent of inter-arrival times below the gain threshold (considering a cache size of 32, and therefore gain threshold of 32 packets serviced) and a small fraction below the loss threshold. These trends are reflected in the throughput plots in the experiments.

5.2. Off-Chip Bandwidth

With no cache, every queue operation results in three off-chip references. Each enqueue operation requires one read and two writes to memory. A dequeue requires two reads and one write. Thus

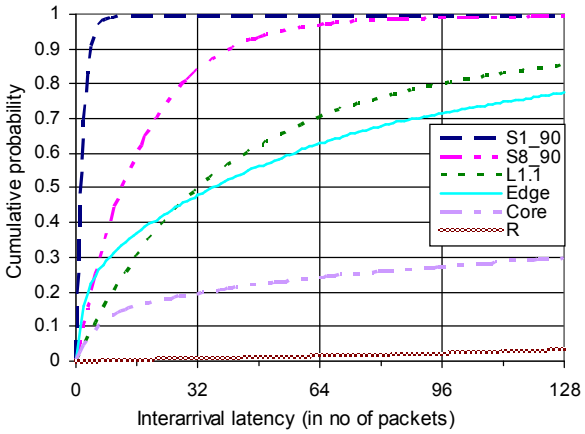

Figure 10: Plots the probability distribution of the average inter-arrival time of all packets within each queue. It is clear that as we move from a single dominant queue to a set of dominant queues to exponentially distributed weighted queues and real-world traces, the inter-arrival time distribution becomes wider. In the limit, random distributions have the widest range of inter-arrival times.

on average, if the system throughput is T, the required off-chip bandwidth will be $1.5*T$ for both reads and writes. For cache based systems with a hit rate of h, the effective bandwidth will get reduced by h, and will be $(1.5-h)*T$ for both reads and writes.

5.3. On-Chip Bandwidth

A cache-less system without any hardware support for queuing uses the same amount of bandwidth on-chip as off. With the addition of a data cache, while the off-chip bandwidth is reduced, the on-chip bandwidth stays the same. The software queuing cache adds a small amount of on-chip bandwidth due to cache management. A queuing cache significantly reduces the on-chip bandwidth because an enqueue or dequeue requires only a single instruction.

5.4. Efficiency

On the Intel IXP, the queue manager software that controls the Q array structure in the SRAM controller (which is similar to our software queuing cache model) requires approximately 60 instructions. The data cache and cache-less systems each use a smaller number of instructions since only the instructions to read and write the queue descriptors, update the queue descriptor, and modify the links are required. This requires approximately 30 instructions. A queuing cache requires only 3-4 instructions for either an enqueue or dequeue. Thus, a queuing cache simplifies the programming of the packet processor. In fact, it frees up resources which are otherwise used for queue management.

5.5. Scalability and Flexibility

One of the primary advantages of the queuing cache model is its scalability. As has been shown, a system with a queuing cache is often able to achieve higher throughput with about half to two-thirds as many threads. In addition, a queuing cache eliminates the synchronization overhead

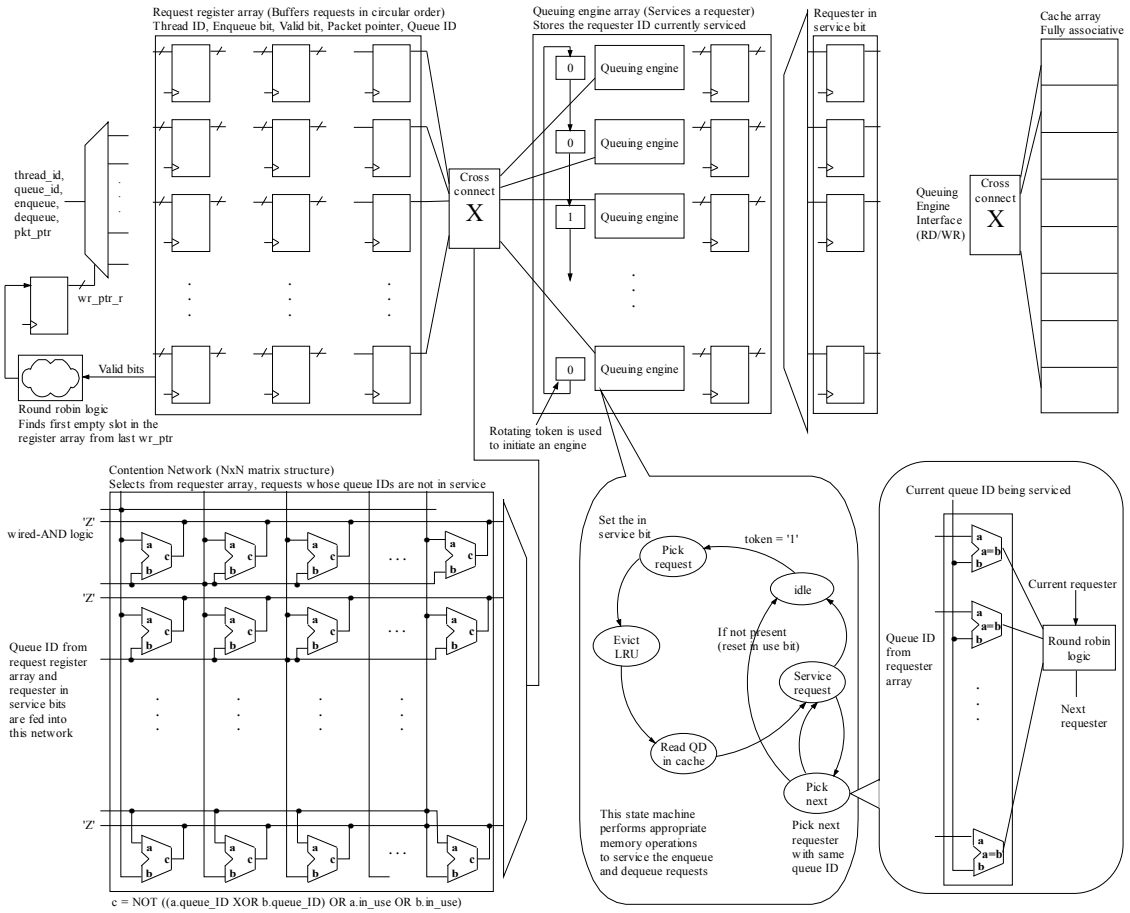

Figure 11: High level organization and circuit diagram of the queuing cache implemented for this experimental evaluation.

which is needed in the other models. The software queuing cache model requires additional on-chip bandwidth for synchronization. For example, the Intel IXP Q array is kept coherent with a CAM at a single micro-engine; this restricts queue operations and management to a single processor. Consequently, there is a limit to the number of threads that can be used and, hence, the maximum throughput of the system. In a queuing cache, there is no such limitation on either the number of threads used for queuing or the location of these threads. Thus, throughput can be improved by increasing the number of threads and increasing the size of the queuing cache without a large increase in the on-chip bandwidth utilization. Moreover, in the data cache and cache-less models, a synchronization mechanism is needed to resolve contention. As we have noted, we have not accounted for the synchronization overheads of these systems, and therefore a performance loss is likely in a real implementation.

5.6. Implementation Complexity

A queuing cache implementation consists of: a fully associative cache to hold the recently accessed queue descriptors, logic maintaining the LRU entry for eviction purposes, and an array of queuing engines. In order to ensure correct service order within each queue, each arriving

Block name	Flip Flop Count	Combinatorial Logic Gate count
Request register array	1760	2K
Contention Network	-	18K
Queuing engine array	512	5K
Cache array	1K	4.5K
Cross connects	-	7.5K, 1K

Table 6: Gate count estimate of the example queuing cache implementation.

request is placed into a circular buffer. A queuing engine is allowed to choose a request from the buffer every clock cycle. This is achieved by using a rotating token method. In order to resolve contention involving multiple requests for the same queue, an *NxN* contention network, where *N* is the maximum number of requesting threads, selects the queues which are not currently in service. One of them is chosen by the queuing engine currently holding the token. Subsequently, all requests with the same queue ID are automatically masked by the contention network. Thus, every engine services a unique queue sequentially. An eviction and an allocation are performed on a miss. Since queuing engines are initiated sequentially, only one eviction and allocation in the cache can occur during a given clock.

Once a request is serviced, the engine checks the circular buffer for jobs pending for the queue it has just served. A linear search is performed beginning from the position from where the last request was picked. This ensures the correct service order within each queue. The high level organization and circuit diagram of the queuing cache is shown in Figure 11.

Table 6 summarizes the combinatorial gate count and flip-flop count estimates of the queuing cache, for a configuration in which there are 32 parallel queuing engines and a fully associative 32 word cache array. The queue ID is assumed to be 16-bits wide, which in turn implies a maximum of 64K queues, and packet pointers are 32-bits.

As we have determined from our (non-optimized) sample implementation, the complexity of a queuing cache with 32 entries is comparable to the complexity of a 128-entry, 2-way set associative, lock-up free cache that can support 32 concurrent requests. This indicates that a queuing cache is a cost-effective means of improving queuing performance in a robust way.

6. Conclusion

This paper describes and evaluates the queuing cache, a hardware cache with a tightly-coupled queuing engine. We have shown that a queuing cache can improve packet queuing performance over a wide variety of synthetic and real-world workloads relative to a number of alternative memory models. The queuing cache improves performance by reducing the number of on- and off-chip requests generated, and thereby reducing operation latency. We compare the queuing cache to: a cache-less system with no support for queuing, a system with a software-controlled

queuing cache (similar to the mechanism used in the IXP 2XXX network processors), and a system with a data cache. Under worst-case conditions, when all packets belong to a single queue, the queuing cache provides throughput improvement factors of 3.1, 1.5, and 2.1 over these models, respectively. Under uniformly random traffic, all organizations saturate external memory bandwidth and achieve the same performance, but the queuing cache shows greater efficiency by requiring 33%-40% fewer threads to achieve maximum performance. As expected, when locality exists in a workload, all cache-based models see improved performance. In a real-world trace of edge Internet traffic, the queuing cache improved throughput by factors of 2.6, 1.3 and 1.75, respectively. The queuing cache also results in gains in on-chip bandwidth, off-chip memory bandwidth, and programmer ease-of-use. Finally, our RTL-style VHDL implementation demonstrates that the footprint of a queuing cache is not too large, making it a highly cost-effective means of improving queuing performance.

Acknowledgments

This work was supported in part by NSF grants CCF-0430012 and CNS-0325298 and by a gift from Intel Corp.

References

[1] M. Adiletta, et al. "The Next Generation of Intel IXP Network Processors," *Intel Technology Journal*, pp. 6-18, Aug. 2002.

[2] M. Chiang, G. Sohi, "Evaluating Design Choices for Shared Bus Multiprocessors in a Throughput-Oriented Environment," *IEEE Transactions on Computers*, pp.297-317*,* Mar. 1992.

[3] C. Scheurich, M. Dubois, "Lockup-free Caches in High-Performance Multiprocessors," *J. Parallel Distrib. Comput.*, pp. 25-36, Jan. 1991.

[4] J. Hasan, S. Chandra, T. N. Vijaykumar, "Efficient Use of Memory Bandwidth to Improve Network Processor Throughput," *Proc. 30th Int'l. Symp. on Computer Architecture,* (ISCA 03), ACM Press, pp. 300-313, 2003.

[5] S. Iyer, R. R. Compella, N. McKeown, *Designing Buffers for Router Line Cards*, Technical Report TR02-HPNG-031001, Computer Science Dept., Stanford University, 2002.

[6] S. Iyer, R. R. Kompella, N. McKeown, "Analysis of a Memory Architecture for Fast Packet Buffers," *Proc. 2001 IEEE Workshop on High Performance Switching and Routing*, pp. 368-373, May 2001.

[7] G. Shrimali, N. McKeown, *Statistical Guarantees for Packet Buffers: The Monolithic DRAM Case*, Technical Report TR04-HPNG-020603, Computer Science Dept., Stanford University, 2004.

[8] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory Access Scheduling," *Proc. 27th Int'l. Symp. on Computer Architecture,* (ISCA 03), ACM Press, pp. 128-138, 2000.

[9] W. Lin, S. Reinhardt, D. Burger, "Reducing DRAM Latencies with an Integrated Memory Hierarchy Design," *Proc. 7th Int'l Symp. on High-Perf. Computer Architecture*, (HCPA 01). IEEE Press, pp. 301-312, 2001.

[10] Y. Joo, N. McKeown, "Doubling Memory Bandwidth for Network Buffers," *Proc. 17th Ann. Joint Conf. of the IEEE Computer and Communications Societies*, (INFOCOM 98), IEEE Press, pp. 808-815, 1998.

[11] S.A. McKee, et al., "Dynamic Access Ordering for Streamed Computations," *IEEE Trans. on Computers*, pp. 1255-1271, Nov. 2000.

[12] R.L. Lee , P.C. Yew , D. H. Lawrie, "Multiprocessor Cache Design Considerations," *Proc. 14th Int'l. Symp. on Computer Architecture,* (ISCA 87), ACM Press, pp.253-262, 1987.

[13] A. Nikologiannis, M. Katevenis, "Efficient Per-Flow Queuing in DRAM at OC-192 Line Rate Using Out-of-Order Execution Techniques," *Proc. 2001 IEEE Int'l Conf. on Communications*, (ICC 2001), IEEE Press, pp. 2048-2052, 2001.

[14] S.-T. Chuang, et al., "Matching Output Queuing with a Combined Input and Output Queued Switch," *Proc. 18th Ann. Joint Conf. of the IEEE Computer and Communications Societies*, (INFOCOM 99), IEEE Press, pp. 1169-1178, 1999.

[15] M.G. Hluchyj, M.J. Karol, "Queuing in High-Performance Packet Switching," *IEEE J. Sel. Areas Comm.*, pp. 1587-1597*,* Dec. 1998.

[16] S.N. Bhatti, J. Crowcroft. "QoS-Sensitive Flows: Issues in IP Packet Handling," *IEEE Internet Computing*, pp. 48-57, July 2000.

[17] J-G Chen, et al. "Chapter 14—Implementing High-Performance, High-Value Traffic Management Using Agere Network Processor Solutions", *Network Processor Design*, volume 2, Morgan Kaufmann, 2004.

[18] M. Shreedhar, George Varghese. "Efficient Fair Queuing Using Deficit Round-Robin," *IEEE Trans. on Networking*, pp. 375-385, June 1996.

[19] V. Frost, B. Melamed, "Traffic Modeling for Telecommunications Networks," *IEEE Communications Magazine*, pp. 70-80, Mar. 1994.

[20] A.J. Smith, "Cache Memories," *ACM Computing Surveys*, pp.473-530, Sep. 1982.

[21] Backbone packet header traces at OC192 and OC48, collected at Cooperative Association for Internet Data Analysis (CAIDA), http://www.caida.org/projects/trends/data/

[22] Backbone and edge packet header traces collected from the Internet Measurement, Internet Analysis, National Laboratory for Applied Network Research (NLANR), http://moat.nlanr.net/