Impact of Network Protocols on Programmable Router Architectures*

Ben Hardekopf, Taylor Riché, Jasleen Kaur, Jayaram Mudigonda,
Michael Dahlin and Harrick Vin
Laboratory for Advanced Systems Research
Department of Computer Sciences
The University of Texas at Austin

Abstract

This paper focuses on two questions: (1) how do properties of next-generation network services affect programmable router architectures and (2) how do properties of programmable router architectures affect the design of next-generation network services. We address these questions in the context of a programmable router based on Intel's IXP1200 network processor and a set of network services that offer a range of Quality of Service (QoS) guarantees to flows.

We develop a modular software framework consisting of three building blocks—flow classification, route selection, and packet ordering—and construct 12 canonical network services for providing QoS guarantees by selecting appropriate variations of these building blocks. We (1) analyze the cost of each building block, (2) systematically evaluate and directly compare the demands of different network services, and (3) examine how these demands are likely to scale as network speeds increase. We construct a prototype system written in Micro-C and evaluate it on the cycle-accurate Intel IXP 1200 simulator. Our evaluation identifies architectural enhancements that can help instantiation of such services in next-generation programmable routers.

1 Introduction

Two trends drive the design of next-generation router architectures. First, routers are called upon to support an ever increasing set of rich network services. Second, instead of using ASICs, routers are being designed from programmable components—such as network processors (NPUs)—that can support packet processig applications efficiently. This paper focuses on two questions: (1) how do properties of next-generation network services affect programmable router architectures? and (2) how do programmable router architectures affect the design of next-generation network services? We address these questions in the context of a programmable router based on Intel's IXP1200 network processor and a set of network service architectures that offer a range of Quality of Service (QoS) guarantees to flows.[1]

Realizing next generation network services on programmable router architectures is challenging for at least three reasons. First, network workloads are demanding. For example, to keep up with a 1Gbps network link, a router must process a 500 byte packet in less than 4 microseconds. Second, router architectures must be scalable to support multiple physical network links per router, large per-link bandwidths, and large numbers of network flows on each link. In particular, Internet backbone network capacity has grown at about 100% per year over the 30-year history of the Internet [6], and router architectures must scale throughput at a similar rate. Third, programmable router architectures may be called upon to support a wide range of network service protocols with different service guarantees. In some cases, worst-case processing time may be as relevant as average-case processing time, while in others overall average throughput is what matters.

To understand the requirements placed on routers architectures by a range of next-generation rich network services, we develop a modular software framework consisting of three building blocks—flow classification,

*__Draft__ *2002.11.18: Please do not redistribute. Please check http://www.cs.utexas.edu/users/dahlin for a current version of this paper.*

[1] A flow refers to a sequence of packet transmitted from a source to a destination.

route selection, and packet ordering—and construct 12 canonical network protocols for providing QoS guarantees by selecting appropriate variations of these building blocks. This framework allows us to (1) optimize the building blocks of these services in order to understand their fundamental costs, (2) systematically evaluate and directly compare the demands of different services, and (3) examine how these demands are likely to scale as network speeds increase. We construct a prototype system written in Micro-C [9] and evaluate it on the cycle-accurate Intel IXP1200 simulator using a set of microbenchmarks that allow us to stress system components using both expected-case and worst-case workloads.

Our primary conclusion is that the next generation network protocols place heavy demands on programmable network routers' memory systems along the dimensions of throughput, size, and latency. In particular, the IXP-1200 is able to route at 85.6% to TBD 146% of line-speeds using 14.3% to 54.6% of system memory bandwidth and 1.6% to 226% of system SRAM capacity. And, in order to scale to support these protocols in the future, programmable router architectures' memory throughput and memory size need to increase approximately in proportion with rapidly increasing network bandwidths. Furthermore, because memory latencies are unlikely to improve at this rate, the number of concurrent hardware threads will also need to increase in proportion to network bandwidth in order to provide sufficient concurrency to mask memory access latencies. In contrast with existing practice, we find that per-thread local memory or cache could significantly improve router throughputs for a given global memory throughput budget. For example, a 1 KB per-hardware-thread local memory would reduce average per-packet memory consumption for route look-up to approximately one-third of its current demand.

In addition to shedding light on how routers should be designed to support network services, this work can also be viewed as addressing the dual question: which network services are practical to build based on their hardware demands and the capabilities of existing or forseeable hardware? In the past, proponents of different network service architectures that provide per-flow service guarantees (e.g., IntServe, MPLS, and CSGS) have argued qualitatively about the performance or scalability of the alternatives. This paper provides the first apples-to-apples comparison of these protocols using a realistic hardware and software platform. We find that there are significant differences in the demands these protocols place on hardware. For example on the IXP 1200 platform, three protocols for providing per-flow delay guarantees, IntServe [2], Multi-Protocol Layer Switching (MPLS) [20], and Core-Stateless Guaranteed Service [10] have maximum throughputs of 89.8%, 112%, and 113% of network line-speed while using 54.6%, 45.9%, and 40.0% of system memory bandwidth respectively. We also find architectural constraints that impact a range of protocols; for example, this study suggests that network protocols that currently rely on exact sorting of packets by priority in order to meet QoS goals are unlikely to be feasible given router architectures and that they may need to be modified use additional receiver buffering at end points and to relax end-to-end delay guarantees so that interior routers need only provide approximate sorting of packets by priority.

The primary contribution of this paper is to systematically evaluate the architectural demands that a wide range of end-to-end network protocols place on programmable routers to understand the architectural implications for routers and to determine which distributed network protocols are likely to be practical.

This evaluation quantifies how memory bandwidth, memory size, and hardware threads should be scaled to meet likely increases in network throughput demands, and it indicates that architectural features such as per-thread local memories could reduce bandwidth pressure significantly. This evaluation also suggests changes in end-to-end network protocols to accomodate likely hardware limits

The rest of this paper proceeds as follows. In Section 2, we provide an overview of programmable network router hardware architectures as well as the range of QoS protocols we study. In Sections **??**, 4, and 5 we discuss the three key network protocol building blocks: classification, route selection, and output ordering. Then, Section 6 looks at the effects of router architectures on end-to-end network protocols. Finally, Section 7 discusses related work and Section 8 summarizes our conclusions.

2 Background

2.1 Programmable Routers

Traditionally, high performance routers have been designed using fixed-function, Application Specific Integrated Circuits (ASICs) that process packets at high speeds. Generally, these ASICs have long design and development times and are difficult to upgrade. Recently, an alternate approach, which uses programmable network processors to design high-performance routers, is gaining popularity. Network processors are fully programmable; consequently, they facilitate rapid development and deployment of new services. Further, they create an excellent platform for performing apples-to-apples comparison of different network service architectures. Hence, in this paper, we use for our analysis a router based on Intel's IXP1200 network processor. IXP1200 network processor was designed to support routers with OC-3 links (i.e., with aggregate router bandwidths of about 100-500Mbps). The next-generation network processors from Intel (e.g., the IXP2400 and 2800 family of network processors to be available within the next year) will support link bandwidths up to 10Gbps (OC-192 links). These network processors share the same basic architectural features (multiple RISC cores per processor, multiple hardware threads per core, etc.), albeit IXP2400 and IXP2800 network processors support a greater number of and faster processor cores. Hence, in this paper, we use IXP1200 as the basis for analyzing the impact of router architectures on the design of network services and vice versa; and then extrapolate our results to discuss the implications on the next-generation router architectures.

2.1.1 Intel's IXP1200 Network Processor

Intel's IXP1200 network processor contains a StrongARM core processor, six RISC CPUs (known as microengines), a proprietary bus (the 64-bit 66MHz IX bus) controller, a PCI controller, control units for accessing off-chip SRAM and DRAM memory, and a small amount (4KB) of on-chip scratchpad memory (see Figure 1). The StrongARM Core is used for control path processing, such as handling slow path exception packets, managing routing tables, and other network state information. Microengines, on the other hand, are used for data path processing; they process multiple packets in parallel. Each microengine is associated with a private 4KB instruction store. Both the StrongARM and the microengines are clocked at 200MHz.

To enable a network processor to process packets at line speeds, it is essential to hide the latency incurred while accessing memory during packet processing. To achieve this, each microengine supports 4 hardware

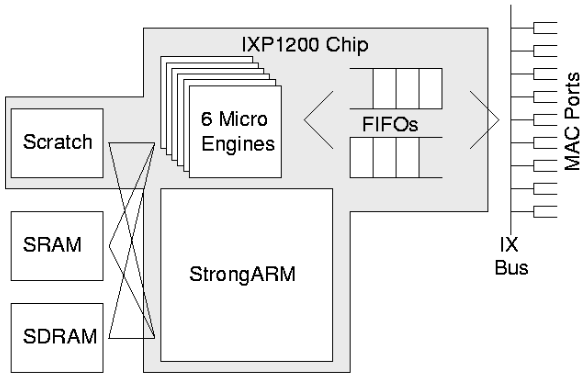


Figure 1: Block Diagram of the IXP1200 System

Feature	IXP1200	IXP2800
Target bandwidth	OC-3 – OC-12 (155-620Mbps)	OC-192 (10Gbps)
Number of microengines	6	16
Number of threads/ME	4	8
Clock rate	200MHz	1.0-1.4GHz
Instruction store	1K instructions	4K instructions
Scratchpad memory	4KB	16KB
Control-path processor	StrongARM	XScale
Off-chip SRAM interfaces	1	4
Off-chip SDRAM interfaces	1	3

Table 1: Comparison of IXP1200 with IXP2800

threads; a microengine can switch context from one hardware thread to another in a single cycle when it issues a memory request.

Given the relative sizes and average access latencies of DRAM and SRAM in the IXP 1200 (256 MB at 45 cycles per access versus 8 MB at 22 cycles per access, respectively) DRAM is most useful in buffering packet data, while SRAM is the natural place to store frequently accessed control information, such as routing tables, per-flow state, etc. SRAM is relatively small in size ($8MB$ in our case) and has a much smaller access time (22 cycles). The on-chip scratchpad (with access time of 16 cycles) is used to read and write short control messages and data that are shared between microengines and the StrongARM.

Bridalveil, the IXP1200-based development systems we used for our implementation and experiments, supports 4 100Mbps Ethernet ports. Thus, at the aggregate arrival rate of 400Mbps, if we conservatively assume packets of size 64 bytes, then packet arrive at the IXP1200 every 1.22 microseconds. Given that the IXP1200 can process 24 packet concurrently (6 microengines each with 4 threads) and the microengines are clocked at 200MHz (5ns/cycle), a processor can work on processing an average packet for at most 5856 cycles. Similarly, given the measured 400MB/s maximum SRAM throughput, to ensure that packets can be processed at their maximum arrival rate, a packet processing application can access at most 122 words of SRAM per packet.

2.1.2 Technology Trends

Over the past three decades, Internet network link bandwidths have approximately doubled each year [6]; this rate of increase is higher than the improvements in traditional processor performance (which doubles every two years) and in memory access latencies (which are improving only by 10-15% a year). Fortunately, network processing workloads provide considerable parallelism across the processing of different packets, so network processor architectures may be able to keep up with network link bandwidths by adding processor cores and memory ports so that system throughput increases even if the latency to process each packet does not improve. Table 1, which compares the architecture of the IXP1200 to Inte's next-generation IXP2800, illustrates this approach.

To evaluate architecture and protocol scalability for systems beyond these existing designs, we assume a simple model for system scaling in the rest of this paper. Given that individual processor core performance and especially memory latency are likely to improve more slowly than network bandwidths, we assume that future architectures will use parallelism to cope with increasing throughput demands. In particular, in comparing future demands of network protocols to resources programmable routers will likely be able to provide, we assume that at best the number of hardware threads, the bandwidth to memory, and the size of memory will increase in proportion to packet processing demands and that this will allow the budget of memory references per packet to remain approximately constant.

Note that although these assumptions are a plausible strategy for scaling programmable router architectures, they are still somewhat aggressive. In particular, while transistor counts have historically grown by a factor of two every 18 months, Internet network bandwidths historically have doubled each year [6]. Thus, scaling the number of hardware threads and memory size with network bandwidth may be challenging. Similarly, although increasing transistor counts can be harnessed to increase memory throughput (e.g., by adding ports, widening busses, or pipelining memories), the need to simultaneously increase memory size and memory throughput also means that a large fraction of the required memory will likely continue to reside off-chip, which means that pin bandwidth could limit improvements in memory bandwidth. Burger et. al [5] find that processor performance outpaced per-package memory bandwidth by about a factor of ten per decade from 1977 to 1997, implying roughly only a 35%/year improvement in per-package memory bandwidth. But, higher rates of improvements may be feasible given that (1) efforts are being made to address the pin bottleneck [14, 15, 28] and (2) the bottleneck is in some sense economic rather than fundamentally technical: a router architecture could, for instance, "buy" bandwidth by using multiple network processor chips. We consider the impact of more conservative technology scaling assumptions in Section 6.

In evaluating the impact of forseeable programmable processor architectures on network protocols we conclude that network protocols that require processing, memory bandwidth, or memory capacity to increase superlinearly with network throughput are unlikely to be scalable. Also, because our architecture assumptions imply that the number of active threads will increase in proportion to network throughput because parallelism will increase while latency improves slowly, we conclude that designs whose lock contention increases as threads are added are unlikely to scale.

For those protocols and building blocks whose demands increase in proportion to bandwidth, we estimate the number of hardware threads that would be needed to sustain a total link throughput of T packets/second assuming that memory bandwidth is the primary constraint by measuring the number of SRAM accesses (in words) per packet (M), and the average SRAM memory access latency per memory request in seconds (L), and caculating ($threads = M \times T \times L$). Similarly, we estimate the required memory bandwidth using the above values and S, the size of a word in SRAM (32 bits for the IXP 1200). The memory bandwidth must be able to sustain at least ($bandwidth = M \times T \times S$) bits per second.

2.2 Network Service Architectures

Traditionally, routers support simple packet forwarding function; for each packet received from an input port, the router determines and transmits the packet onto an output port based on the destination address for the packet. Networks based on these simple IP routers provide best-effort service to packets; the network does not provide any guarantees on when, or if, packets will be delivered to the destination. Over the last decade, the number and the complexity of applications supported by routers have grown considerably. Today, routers support a wide range of applications including firewalls, content filtering, usage auditing, virus scanning, and intrusion detection. Further, routers employ sophisticated techniques for providing bandwidth, delay, and jitter guarantees to flows. The ability of a router architecture to support a network service at line rates depends on the computational complexity and the memory access requirements of the network service. To evaluate the implication of router architectures on the design of such network services, in this paper, we focus on network services that can provide to flows end-to-end guarantees on parameters such as delay, bandwidth, and jitter.

To provide delay, bandwidth, and jitter guarantees, routers employ mechanisms that differentiate among packets belonging to different flows. The literature describes several different network architectures — such as the Integrated Services (IntServ) network [22], Core-stateless Guaranteed Services (CSGS) networks [27], and Multi-protocol Label Switching (MPLS) networks [20]— for providing guarantees to flows. All of these architectures rely of three fundamental *building blocks*: (1) *classification* that can associate each incoming packet as belonging to one of the existing flows; (2) *route selection* that determines the next-hop address for the packet; and (3) *packet ordering* that can enforce an ordering among packets transmitted by the router on a network link. Whereas the flow classification component enables a router to control the amount of network processor resources allocated to process packets of a flow, the packet ordering component allows a router to control the allocation of network bandwidth to flows. In what follows, we discuss briefly the functionality of each of these building blocks, and then describe how they can be combined to construct different network services.

Flow Classification is the process of splitting a stream of incoming packets into a set of flows of *related* packets. Once classified, the router can provide different levels of service to packets of each flow. Generally, flow classification is the very first processing step a packet undergoes upon arriving at a router. The functionality of a classifier is generally spread across both control and data planes. The control plane is responsible for creating and maintaining a *Flow Table* that is used by the data plane to classify packets. When a new flow is established, the control plane *inserts* a new *Flow Record* into the *Flow Table*. On a packet arrival, the data plane *accesses* the *Flow Record* using the information contained in packets.

There are two models for identifying a packet as belonging to a flow: (1) *tagging*, where each packet carries an explit field that identifies the flow (e.g., a field that contains an index into the Flow Table); and (2) *matching*, where flow identification is derived from a combination of fields contained in the packet. Tagging is clearly simpler to implement at the router; however, to implement tagging, the network needs to support an end-to-end signalling protocol that allows routers to agree upon the flow identification tag(s). Matching, on the other hand, is more complex to implement at the router. It often involves processing multiple fields in the packet header–such as the source and destination IP addresses, the source and destination port numbers, and protocol type. Hash-based classification is a specific implementation of matching; routers derive a hash on multiple packet header fields and use that as the flowID to access the flow record. In Section **??**, we compare and contrast the impact of tagging and hash-based matching techniques on router architectures.

Route Selection is the process of selecting the next-hop router based on the IP address of the destination included in the packet. Since IP addresses are hierarchically allocated in the Internet, routing tables at routers generally maintain next-hop information for IP address prefixes (that represent a collection of hosts with the same IP address prefix). Hence, route selection generally involves determining the *longest prefix-match* in the routing table for the destination host IP address. Several Trie-based schemes [26, 4] proposed in the literature are well-suited for this function. The other possibility for route selection is referred to as *source routing* (or a simple lookup); in this scheme, the source of the flow includes in the packet header the next-hop information for all the routers that the packet may pass through. In Section 4, we compare and contrast the impact of using trie-based schemes for determining longest-prefix match in routing table or using source routing on router architectures.

Packet Ordering mechanism determines the relative order of transmitting packets on an outgoing network link. Today's IP routers transmit packet on a network link in the first-in-first-out (FIFO) order. Unfortunately, FIFO scheduling of packets does not isolate the performance of flows; a burst of packet arrival for a flow can affect adversely the delay suffered by packets of other flows at the router. Thus, to provide any performance guarantees —with respect to delay, jitter, or throughput— to flows, it is necessary for routers to employ (1) scheduling algorithms that can determine the relative order of transmitting packets, and (2) packet ordering or sorting techniques that can transmit packets in the order defined by the scheduling algorithm. Most scheduling algorithms compute a deadline for each packet and queue the packets for transmission in the order of their deadlines. Whereas the deadline computation generally involves only a few ALU operations (and hence does not impose much overhead), the task of sorting packets in accordance with their deadlines requires the router to maintain complex data structures and update them upon arrival and departure of packets.

The literature contains perfect sorting and approximate sorting data structures for ordering packet transmissions. Examples of perfect sorting data structures include Priority Queues, balanced search trees and

Service Architecture	Classification	Route Selection	Ordering
Baseline	NULL	Source-routing (lookup)	FIFO
IP router	NULL	Prefix matching	FIFO
IntServ	Matching	Prefix matching/lookup	Sort
CSGS	NULL	Prefix matching/lookup	Sort
MPLS	Tagging	Prefix matching/lookup	Sort

Table 2: Deriving canonical network service architectures using building blocks

heaps. When the range of values in a Priority Queue is bounded, efficient implementations that use ideas similar to Bin-Sort or Calendar Queues [3] are possible. In fact, if the sorting accuracy can be relaxed, it is possible to design even *constant time* implementations of a Priority Queue that trade sorting accuracy for processing complexity [16, 19]. In Section 5, we discuss the implications on router architectures of implementing FIFO, perfect sort, and approximate sorting techniques for ordering packet transmission in routers.

Constructing service architectures from building blocks By composing together different implementations of these three basic building blocks, we can construct several canonical network service architectures for providing QoS guarantees to flows. For instance, today's IP routers use a NULL flow classification step, IP prefix matching for route selection, and FIFO scheduling for packet ordering. The Integrated Services (IntServ) network service architecture uses hash-based classification, either trie-based prefix-matching scheme or direct lookup (if the next-hop information is maintained in the Flow Record) for route section, and the perfect sorting for packet ordering. Similarly, the Multi-Protocol Label Switching (MPLS) network architecture uses an explicit tag (or a label) in packet header for classification, either trie-based prefix-matching scheme or direct lookup (if the next-hop information is maintained in the Flow Record) for route section, and the perfect sorting for packet ordering. Table 2 summarizes how some of the prominent network service architectures can be derived from these building blocks. We will discuss the impact of router architectures on the relative scalability of these services in Section **??**.

Finally, we want to point out that although we attempt to tune our implementations of the building blocks to maximize system performance for a given semantic requirement, we do not claim that the implementations are optimal. It is possible that different implementations could reveal different architectural trade-offs. To make the trade-offs we make clear, in the next three sections, we discuss the design space and our design rationale in detail before describing our results.

3 Flow Classification

In this section, we compare and contrast the implications of using tagging or hash-based matching scheme for classification on router architectures. We find that on our IXP1200-based routers, both tagging and matching can perform classification at line speeds. Whereas tagging can support classification at line speeds while using 2 hardware threads and 9% of memory bandwidth, hash-based classification requires 4 hardware threads and uses 31% of the memory bandwidth to achieve the same performance. We show that the memory size requirements for both the schemes grow linearly with increase in link bandwidth; hence, both schemes scale well with increase in link bandwidth. In what follows, we first describe our implementation of hash-based classification, and then present details of our experiments.

3.1 Tagging and Matching Algorithms

We implement two methods of classification: tagging and matching. In tagging, a field in the packet header explicitly contains the flowID for the packet; a router can utilize this flowID to retrieve the corresponding Flow Record from the Flow Table. To implement matching, we use a hash table with chaining (for resolving collisions). We use the hash of 5-tuple (<SourceIP, DestinationIP, SourcePort, DestinationPort, ProtocolID>); this allows the classifier to identify finest-granularity flows (and thereby present the most difficult case for our scalability analysis). Two reasons motivate the selection of hashing as a flow identification mechanism. First, the basic hashing algorithm is simple and can be implemented efficiently using hardware supported hashing functions on the IXP1200. Second, hashing schemes have a very good average-case performance; this allows hardware resources to be freed up for other tasks.

It is important to realize that a router can differentiate among packets belonging to different flows only after the classification step. Hence, for routers to provide QoS guarantees to packets, the classification step must operate at line speeds. If this condition does not hold, then the task of classifying a low-priority packet can violate the QoS requirements of high-priority packets. Since memory access latencies constitute the predominant cost of implementing these classification schemes, the need to operate at line speeds places a limit on the maximum number of memory accesses that can be performed to classify a packet. This requirement has implication for the hash-based classification scheme. This is because, with hashing, in the worst-case, a large number of flows may hash onto the same slot; in such a case, flow identification would require several memory accesses. Thus, given a bound on the maximum number of memory accesses the flow classifier can make, hash-based classification scheme has to reject flows that hash onto slots for which the number of memory accesses (or collisions) exceed the bound.

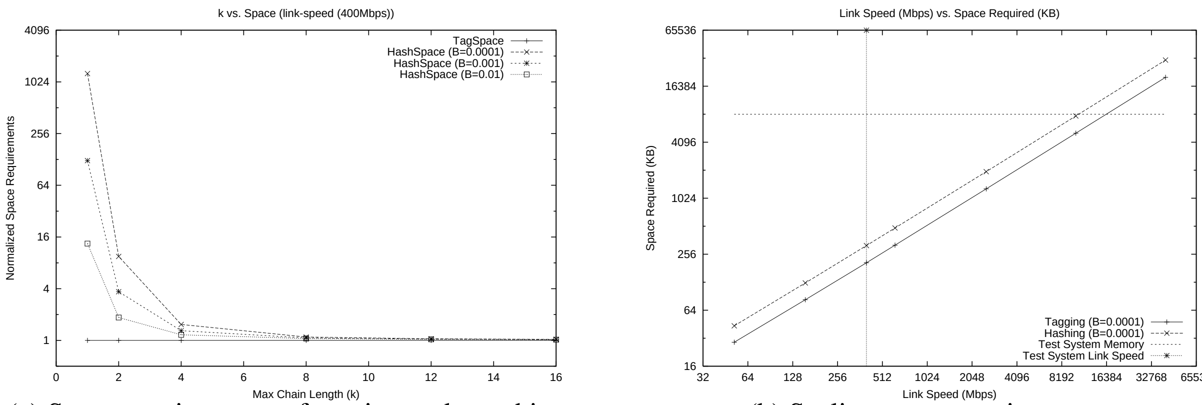
Observe that in the case of tagging, the average-case and the worst-case are the same (a single memory access); hence, in the case of tagging, a flow is rejected only if the flow table is full. In the case of hash-based classification, a flow is rejected even when a flow hashes to a slot with a chain of maximum allowed length. Clearly, increasing the size of the hash table decreases the probability of collisions; this in turn reduces the flow rejection probability. Similarly, increasing the number of threads that perform hash-based classification increases the memory access budget; this in turn also reduces flow rejection probability. Thus, hash-based classification exposes a tradeoff: the greater is the amount of memory or computational resources allocated to classification, the smaller is the flow rejection probability.

3.2 Experiments and Results

In this section, we first compare the memory space requirement for tagging and hash-based matching schemes, and evaluate how these requirements scale with increase in link bandwidth. Second, we compare the number of threads required for tagging and hash-based matching to ensure that classification proceeds at line speeds. Finally, we derive estimates on the memory bandwidth and the number of threads that a router architecture should support to scale tagging and hash-based matching to high bandwidth links.

Figure 2(a) depicts the memory space requirements for tagging and hash-based matching. The memory requirement for the hash-based matching scheme is shown as a multiple of the memory space needed for tagging. Figure 2(a) shows how the memory requirement for hash-based scheme varies with the maximum allowed chain length, k, and the flow rejection probability. It demonstrates that hash-based scheme expose memory space – memory bandwidth (and hence, rejection probability) tradeoff. For a flow rejection rate of 0.0001, restricting $k = 4$ requires the hash-based scheme to use 50% more memory than tagging. With $k = 2$, on the other hand, the memory requirement for hash-based scheme is 8 times that of tagging.

Figure 2(b) illustrates how the space requirement for tagging and hashing scales with link bandwidth. For this experiment, we consider a setting with $k = 4$ and a flow rejection rate of 0.0001. We consider the case where the links are operating at 100% utilization and assume that each flow requires 64Kbps bandwidth. Figure 2(b) shows that the space requirement for both tagging and hashing grows roughly linearly with increase in link bandwidth. Further, it shows that for our IXP1200-based system operating at a cumulative link bandwidth of 400Mbps, tagging and hashing, respectively, need only 210KB and 318KB of memory space. This indicates that for moderate values of k, the memory space requirement of both tagging and hashing will not be a problem for future routers.



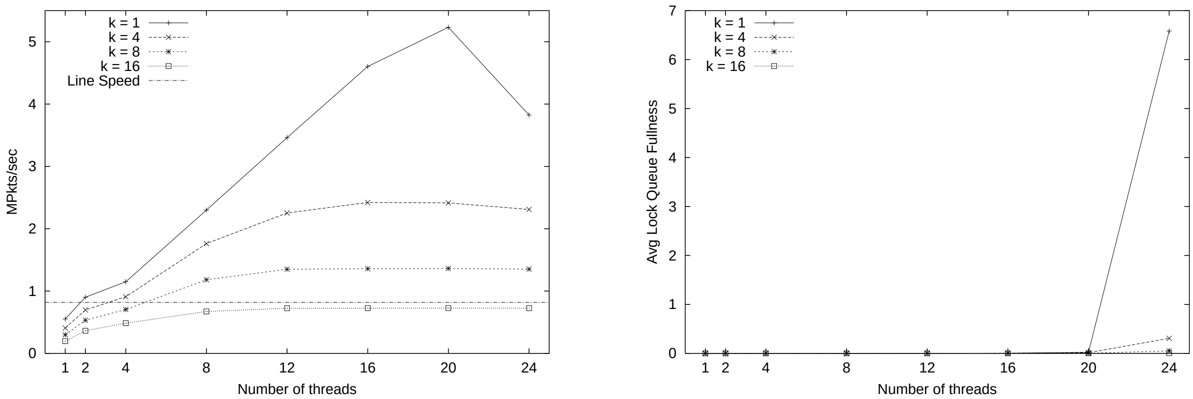
(a) Space requirements of tagging and matching

(b) Scaling space requirements

Figure 2: Comparing space requirements of tagging and hash-based matching.

We now discuss how the number of threads needed to perform classification varies with increase in link bandwidth. Figure 3(a) demonstrates that, for our IXP1200-based system, tagging can operate at line speeds with only 2 threads. Hash-based matching, on the other hand, requires 4 threads (assuming that k is restricted to 4). With $k = 8$, 8 threads need to be allocated to the hash-based classifier. While operating at line speeds, tagging uses up 9% of the memory bandwidth, and hash-based matching with $k = 4$ and $k = 8$, respectively, uses up 31% and 78% of the memory bandwidth.

It is interesting to note that the performance of the hash-based matching scheme with $k = 1$ improves almost linearly with increase in number of threads up to 20 threads. However, after that, the overall throughput drops. This is because, IXP1200 only supports 8 hardware locks. When 24 threads are allocated to hash-based matching, the system converges to a state where 8 locks are held simultaneously almost all the time; hence, lock requests fail because no more hardware locks can be held. Figure 3(b) illustrates this behavior. From this we conclude that to achieve scalability, router architectures should include hardware to support fine-grained locking. Software locking is not very tenable; with software locking, threads poll memory continually to see if a lock can be acquired–this places additional burden on the memory bandwidth.



(a) Throughput (b) Lock queue fullness

Figure 3: Throughput and lock queue fullness of matching for different values of k. $k = 1$ is equivalent to tagging.

From the above discussion, we conclude that classification can scale with increasing line speed as long as the memory space, memory bandwidth and number of threads supported by router architectures scale appropriately. Based on our implementation, we expose the following scalability requirements for memory bandwidth and threads.

- Classification will make an average of $(k + 1)$ memory references per packet. However, the total number of words that are accessed per packet is $(8k + 2)$, as each read of an entry in the chain requests 8 words of memory. The total memory bandwidth required by classification with a total incoming network bandwidth of C and minimum packet size of M is $\left(8 \times \frac{4kC+C}{M}\right)$ or $((.5 \times k \times C) + (.125 \times C))$ if we assume 64-byte minimum packets. If a router supports a throughput of OC-192 (10Gbps), classification would require an effective memory bandwidth of 6.25 Gbps for tagging and 21.25 Gbps for matching with $k = 4$.
- The number of threads required to mask memory latency will increase approximately linearly with network bandwidth. To find the proper flow table entry, one SDRAM read is performed (to retrieve the packet data) and k SRAM reads are performed to traverse the chain. One SRAM write is also performed after a match is found. Classification thus occupies a thread for an average of $(k + 1)$ SRAM references per packet and one SDRAM reference per packet. Looking at only the minimum thread occupancy time for a request caused by memory latencies, a router needs at least $(((k + 1) \times L_{SRAM} + L_{SDRAM}) \times \frac{C}{M})$ active threads. For $L_{SRAM} = 80$ns, $L_{SDRAM} = 160$ns, and a minimum packet size of $M = 64$bytes, OC-192 (10Gbps) would require classification to have at least 7.2 threads for tagging and 10.2 threads

for matching with $k = 4$. For OC-768 (40Gbps) tagging requires at least 29 threads and matching with $k = 4$ requires at least 41 threads.

4 Routing

The routing stage of network processing selects the output port to which each packet should be sent. Today's routers commonly use two methods packet routing: IP shortest prefix matching and packet/flow state lookup in which the correct output port is embedded in the packet header by either a previous processing stage in the router (eg., in MPLS protocols, the classification stage can extract next-hop routing information from the per-flow state and avoid the need for a separate route lookup) or by a node at the edge of the network (eg., source routing protocols require the node that injects a packet into the network to embed each step of routing information in the packet header).

In this section, we examine the demands that route selection places on programmable routers. Overall, we find that the routing phase of network protocols places modest demands on routers. If prefix routing is used, to route at line speed the IXP 1200 needs to dedicate 2 to 4 of its 24 hardware threads and 1.4% to 8.3% of its memory bandwidth to routing, and future routers should be able to accommodate routing demands if they can increase memory bandwidth and the number of hardware threads linearly with line speed. We find that the IXP 1200 does not have sufficient SRAM memory capacity to accommodate full IP routing tables, but because such tables grow slowly, SRAM capacity should not be a concern for future generations of router architectures. We also find that small per-thread private memories could be a valuable addition to programmable router architectures because they could reduce the memory bandwidth needed by prefix routing by 22-66% by storing the top two levels of the prefix routing trie in a 1KB local memory. Finally, turning our attention to the impact of programmable router architectures on end-to-end network protocols, we conclude that due to the modest demands and scalability of prefix routing, the lower processing and memory costs of packet/flow-state lookup is unlikely to be a significant factor in designing or selecting network protocols; other network properties such as robustness and simplicity are more likely to determine routing strategy.

4.1 Prefix matching implementation

We implement IP shortest prefix matching using a multi-bit trie with a 4-bit stride. This is a standard technique well described in the literature[21].

Packets are routed on a network basis. The first bits of the IP address are the network component and are referred to as the prefix. The fact that addresses for hosts on the same network will be routed in the same way is referred to as *address aggregation*, which allows the router to only store prefixes instead of individual addresses. IP shortest prefix matching finds the shortest prefix stored in the router that matches the destination address of the packet and routes that packet to the same port as all other packets whose destination address matches that network prefix.

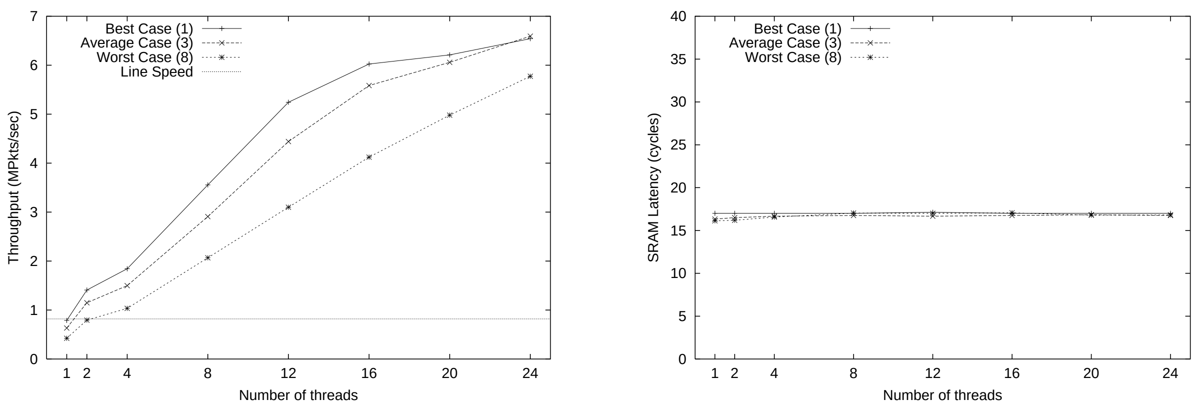
We look at three cases in our analysis of shortest prefix matching. These correspond to the best, average, and worst depth that a thread will have to descend into the trie to find a matching prefix. The best case is a depth of one. The worst case is a depth of eight since we are using a multi-bit trie with a 4-bit stride and 32 bit addresses. In a typical router today average trie lookup depth is 2.76 [21]. A particular router's average trie lookup depth depends on the number of input and output ports it has as well as its location in the network. In particular, edge routers located on small networks with limited connections to the internet may need only a handful of short prefixes to select the correct output port for a packet.

An average core router stores about 100,000 unique prefixes [18]. To store a complete 4-bit stride prefix trie in a core router's SRAM takes approximately 17.6 MB (2^4 bytes per 4-bit trie node $\times$ 100,000 prefixes $\times$ 2.76 nodes per prefix) [21]. The output port for each prefix is stored in the routing table, which is kept in SDRAM. 28 bytes are needed per route table entry. To store the routing table in an average core router takes approximately 2.8 MB (28 bytes per entry $\times$ 100,000 entries). Even though the trie structure is larger, it is accessed more often per packet than the route table and benefits from the lower latency access of SRAM.

Lock contention is not an issue for routing as no locks are necessary to implement shortest prefix matching. The trie structure changes rarely, and when it does, the changes are not made on the data plane by the micro-engines. Therefore, the micro-engines treat the trie structure as read-only.

4.2 Evaluation

The IXP 1200 architecture is sufficiently provisioned for prefix match routing at today's line speed. As Figure 4(a) shows, in all but a few cases, shortest prefix matching achieves a throughput exceeding line speed. In the best and average case, we only need two threads to achieve line speed and in the worst case that number increases to four. Performance with 24 threads is 705% over line speed for the average case. In the worst case, we see a 1263% speedup by going from 1 to 24 threads.



(a) Routing throughput (b) Routing average SRAM access Latency

Figure 4: Routing building block microbenchmark results.

Routing has a peak micro-engine utilization of 85.71% with four threads. In that configuration 7.14% of cycles are stalls due to pipeline flushing on context switches and the microengines are idle the remaining 7.15% of cycles because all threads are waiting for SRAM or SDRAM memory accesses to complete.

The SRAM bandwidth of the current architecture does not limit performance. Two to four threads are sufficient to route at or above line speed for the average and worst cases and these threads use only 3.5%

and 8.3% of the total available bandwidth of the IXP 1200 for average and worst case respectively; 24 threads use 19.8% and 46.3% of the available bandwidth in the average and worst case. As can be seen from Figure 4(b), average SRAM access latency remains constant as we scale the number of threads, indicating that this resource is not oversubscribed.

Adding a local memory to each micro-engine would reduce the number of SRAM references per packet. The compiled code issues one one-word SRAM load to read each level of the tree traversed and makes no other SRAM references, so the average number memory references per packet is 1, 3, and 8 for the best, average, and worst cases respectively. Adding a small (1KB) local memory would be sufficient to store the top two levels of the trie and could reduce SRAM bandwidth required by prefix routing by 22% (worst case) to 67% (average case). Each additional level of the tree requires 16 times more memory, so replicating more than two or three layers of the trie is not likely to be profitable: exponentially larger local memories would be required to linearly reduce worst-case bandwidth, and beyond two or three levels they would provide little improvements to average bandwidth. The read-mostly nature of this data structure makes it amenable to replication to local processor memories; hardware coherent caches do not appear to be required for this optimization.

In addition, one would expect that there is temporal locality across packet destinations, so hardware data caches may be able to reduce global memory bandwith consumption by satisfying lookups for recently accessed paths. In the future, we plan to examine network packet traces to determine whether sufficient locality exists for this technique to provide significant benefits. We speculate that caches shared across threads will provide better hit rates than per-thread caches because packets from a flow to a given destination may be spread across threads.

Since the IXP 1200 only has 8 MB of SRAM, even if we use the entire amount, we do not have enough to hold the average sized trie data structure of a real world router. However, this will not be a significant limitation in the future. On the next family of programmable network processor from Intel, there is 256 MB of SRAM [1]. The number of prefixes in a router is not expected to increase as fast as the size of SRAM on programmable routers.

We conclude that prefix matching can scale with increasing line speeds if the number of hardware threads and memory bandwidth scale proportionally.

Prefix matching in the average case makes an average of 3 memory references per packet. This number may increase only modestly in the future. Given that processing an average packet requires three 1-word memory references in the average case, the memory bandwidth required by prefix matching when total incoming network bandwith is C and the minimum packet size is M average case is $(96 \times \frac{C}{M})$ bits per second or $(.1875 \times C)$ assuming 64-byte minimum packets. For a router that supports throughput at OC-192 (10Gbps), routing would require an effective memory bandwidth of about 1.875 Gbps; OC-768 (40Gbps) routing would require about 7.5 Gbps of effective memory bandwidth.

Similarly, the number of hardware threads required to mask memory latency will increase approximately linearly with network bandwidth. Packet processing occupies a thread for an average of three dependent

SRAM lookups (to traverse the trie) and then one SDRAM lookup (to retrieve the route information) plus additional time for register/register operations. Considering only the minimum thread occupancy time of a request caused by memory latencies, a router needs at least $((3 \times L_{SRAM} + L_{SDRAM}) \times \frac{C}{M})$ active threads to sustain network line rates. For $L_{SRAM} = 80ns$, $L_{SDRAM} = 160ns$, and minimum packet size $M = 64bytes$, OC-192 (10Gbps) route lookup would require at least 7.8 simultaneous threads and OC-768 (40Gbps) would require at least 31 simultaneous threads. The actual number of threads required will be higher due to the additional thread resources consumed by register/register processing; but, this additional cost will fall over time if processor core rates continue to improve more quickly than memory latencies. These "simultaneous threads" can likely be provided with a combination of increasing the number of hardware processors and increaing the number of hardware contexts per microengine. Although the IXP1200's microengines have little idle time and so would benefit little from additional contexts, as noted above they spend about 85% of their time executing thread instructions and only about 7.5% of their time idle (waiting for memory); if register/register operations speed up relative to SRAM and SDRAM latencies, the idle time will increase unless more contexts are added.

5 Packet Ordering

Most routers associate with each output port a buffer that contains packets queued for transmission. This buffer is necessary because multiple input ports may receive packets that are all routed through a single output port. An important characteristic of a router is the order in which it transmits the packets queued at a particular output port. A typical IP router employs a FIFO scheduler that transmits packets in the order they arrive in the buffer. This ordering has the virtue of simplicity and speed, but it is inadequate for providing richer services such as guaranteed bandwidth or delay [10]. For these services, each packet in the router is assigned a deadline by which the packet must be transmitted in order to meet its requirement. It is necessary to be able to sort the outgoing packets by their deadlines to ensure that no packet is transmitted later than its assigned deadline.

In this section we compare the architectural demands and scalability of deadline-aware scheduling with that of simple FIFO scheduling. We conclude that exact sorting is infeasible with today's programmable routers, and will become even less attractive in the future. However, both approximate sorting and FIFO scheduling are feasible on today's routers, and their resource demands scale linearly with increasing packet rates.

5.1 Packet Sorting Algorithm

The ideal packet sorting algorithm would sort the packets in exact order by their deadlines. This ensures that packets with a deadline further in the future are never transmitted before packets with an earlier deadline. However, all of the common exact sorting data structures, such as priority heaps and red-black trees, require $O(\log\ n)$ time and memory references to enqueue or dequeue a packet, where n is the number of packets. This characteristic violates our design constraints, which state that the number of memory references per packet must remain constant as link-speed increases. If such data structures were used, as the link-speed increases and the number of packets simultaneously in the router increases, the router would require more memory references per packet to do exact sorting. Furthermore, these data structures require considerable locking to coordinate thread accesses, which further limits their scalability. Our initial performance evaluation finds that even on current routers, the costs of exact-sorting heap sort are too high: we were able to sustain at most 20% of the IXP1200 line throughput with the various implementations of heap sort we tried.

To meet the requirement imposed by our design constraints that sorting must scale well with increased link speed (and hence scale well with increasing number of packets), we are forced to relax our requirement for exact sorting. We turn to BinSort [17], an approximate sorting algorithm. BinSort does not guarantee that packets will be transmitted in strict order of their deadlines, but it does place an upper bound on the amount of error it introduces.

Given a network description (e.g. number of hops, propagation delay, etc), we can calculate the maximum range of deadlines R that we will see in the router. A packet the router receives at time t is guaranteed to have a deadline which is less than $t+R$. We divide this range into bins, each bin representing a constant time interval. The bins are implemented as FIFO queues. A thread that is enqueuing a packet can calculate which bin contains that packet's deadline, and enqueue the packet directly into the correct bin. A dequeue thread starts at the first bin (i.e. the bin whose time interval contains the earliest times), and successively scans bins until it finds a bin that contains a packet. This process guarantees that packets in an earlier bin are always dequeued before packets from a later bin; however packets within the same bin are unsorted with respect to each other. This algorithm has an enqueue time complexity of $O(1)$ and a dequeue time complexity of $O(m)$, where m is the number of bins. It is important to notice the behavior of this algorithm as related to router utilization. This $O(m)$ term is much less important than it might first appear for two reasons. First, if the router utilization is high—there are constantly packets with deadlines close to the current time ready to be transmitted—the dequeue complexity is actually $O(1)$, since dequeue only examines the first bin to find packets to transmit. The worst case complexity of $O(m)$ is only applicable when the router is lightly utilized; i.e. all the packets ready to be transmitted have deadlines which are a long time in the future (and hence are in bins further away from the base bin). The tradeoff we are making is having a constant dequeue time when it is important to meet packet deadlines, and a linear dequeue time (with respect to the number of bins) when the router is in work-conserving mode (i.e. transmitting packets ahead of their deadlines). Second, we use bitmaps to distinguish full bins from empty ones so a thread can scan multiple bins with each word loaded.

We can bound the error introduced by BinSort to the size of one bin interval per router traversed. This additional delay must be accounted for in the connection set-up phase of end-to-end protocols that provide per-flow guarantees, but it is easy to do so by "pretending" that the network wire delay from one router to the next has been increased by the size of one bin interval. This additional delay motivates us use small bins. On the other hand, the efficiency of dequeue when router utilization is low gets worse as the number of bins increases, which motivates us to make the bins reasonably large. For our experiments with the IXP 1200 we chose a bin size of 1 ms and used 1024 bins.

Each FIFO queue that constitutes a bin is implemented as a linked list. Although an array-based implementation would be faster and consume less memory bandwidth, implementing the bins as arrays would force us to over-provision them to handle the maximum number of packets that could fit in a bin; such arrays would consume 16 MB of SRAM on the IXP 1200, which is twice the total amount of SRAM available. Unfortunately, a linked-list implementation creates additional lock contention compared to an array implementation since both the enqueue and dequeue threads are contending for the same lock to access a particular bin. We are able to avoid scaling this lock contention as the number of threads increases (a requirement of our design constraints) by replicating the data structure. We create one BinSort data structure for every four dequeue threads. The dequeue threads are statically mapped to these structures; the enqueue threads load-balance the work by round-robining among the data structures.

5.2 Experimental Results

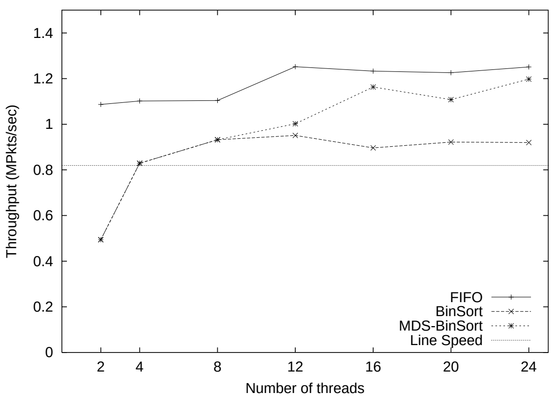
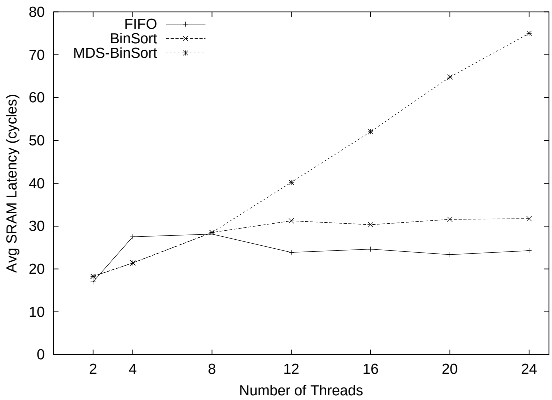
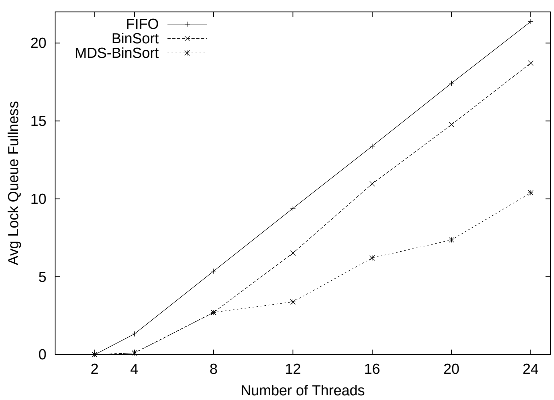


Figure 5: Throughput of packet ordering algorithms.



(a) Memory access latency



(b) Lock Queue Fullness

Figure 6: Memory access latency and lock queue fullness of packet ordering algorithms.

Figure 5 shows the throughput of FIFO, BinSort as a single data structure, and the replicated BinSort data structure (MDS-BinSort) as the number of threads increases. MDS-BinSort is able to achieve a throughput of 1.198 Mpkt/sec (146% of line speed), requiring 0.176 MB of SRAM and 1.5 Gbps of effective SRAM bandwidth. To put these numbers into perspective, this throughput is 96% of FIFO's maximum throughput, using 2.2% of the total SRAM (41% more than FIFO) and 50.5% of the IXP 1200's maximum achievable SRAM bandwidth (219% more than FIFO).

As Figure 6(a) and (b) indicate, FIFO and the unreplicated BinSort are primarily limited by lock contention, and MDS-BinSort is limited by both lock contention and memory bandwidth. The probable cause of MD5-BinSort's lock contention is that the threads are *convoying*: because of the threads' varying speeds, the faster threads catch up to the slower threads and they all start accessing the data structures in lock-step, rather than being distributed evenly among the structures. We plan to attempt to solve this problem by having enqueue threads randomly select instances of the data structure rather than using round robin selection.

FIFO achieves a maximum throughput of 1.252 Mpkt/sec (153% of line speed), requiring 0.125 MB of SRAM (1.6% of the total SRAM available) and 0.47 Gbps of effective SRAM bandwidth (15.8% of the maximum achievable SRAM bandwidth). It is clear that our simple implementation of FIFO as a circular array, while faster than sorting, does not scale well as the number of threads increases. Figure 6(b) shows that the problem is lock contention. Since there is only one array all enqueue threads are contending for one lock. Similarly all dequeue threads are contending for a single lock, causing the linear increase in failed lock requests. This is the same problem encountered in the unreplicated BinSort data structure, and the solution is similar — we must replicate the queue and distribute the threads across these new queues in order to reduce the lock contention. However, we are under a constraint imposed by pre-existing protocols such as TCP that rely on routers not to reorder packets. We can meet this requirement by choosing a queue for a particular packet by hashing its destination address, thereby guaranteeing that packets going to the same destination go into the same queue, and hence are still transmitted in FIFO order. The cost of this approach is that FIFO now has the same order of magnitude of complexity as MDS-BinSort, requiring approximately the same amount of resources.

As we vary the number of threads for MDS-BinSort, the maximum micro-engine utilization (the percentage of time there is a thread running on the micro-engine doing work) peaks at 45%—i.e. even in the best case 55% of the time the micro-engine was not doing any useful work. Of this time 16.5% of the cycles were stall cycles and 38.5% were idle cycles, where all threads on the micro-engine were waiting for memory or lock requests to complete. This result suggests that adding more hardware contexts per micro-engine may be useful, although it appears that memory bandwidth must also be scaled for throughput to increase.

In order to scale approximate sorting or FIFO buffering, SRAM bandwidth, SRAM size, and the number of hardware threads provided by architectures must increase linearly as network speeds increase. For example, Table 3 estimates how the number of threads, SRAM size, and SRAM bandwidth must scale with link-speed in order to meet the requirements of approximate sorting based on our measurements of the average number of memory accesses per packet (14.5) and of words accessed from SRAM per packet (40). Considering only the minimum thread occupancy time of a request caused by memory access latencies, a router needs at least $((14.5 \times L_{SRAM}) \times \frac{C}{M})$ active threads to sustain network line rates. For $L_{SRAM} = 80ns$, and minimum packet size $M = 64bytes$, OC-192 (10Gbps) packet sorting would require at least 22 simultaneous threads and OC-768 (40Gbps) would require at least 91 simultaneous threads. The actual number of

Link-Speed (Gbps)	Min. # Threads	SRAM Size (MB)	SRAM Bandwidth (Gbps)
OC-1 (0.051)	1	0.09	0.127
OC-3 (0.152)	1	0.26	0.380
OC-12 (0.608)	2	1.06	1.52
OC-24 (1.244)	3	2.17	3.11
OC-48 (2.488)	6	4.33	6.22
OC-192 (10)	22	17.41	25
OC-256 (13.271)	30	23.10	33.18
OC-768 (40)	91	69.63	100

Table 3: Link-speed versus resource requirements to scale approximate sorting. The *Min. # Threads* value estimates the number of hardware threads just to tolerate memory latencies and does not include any other processing demands.

threads required will be higher due to the additional thread resources consumed by register/register processing; but, this additional cost will fall over time if processor core rates continue to improve more quickly than memory latencies. These "simultaneous threads" can likely be provided with a combination of increasing the number of hardware processors and increasing the number of hardware contexts per micro-engine.

Similarly, given the average per-packet bandwidth demands of 40 32-bit words, when total incoming network bandwidth is C and the minimum packet size is M average case is $(1280 \times \frac{C}{M})$ bits per second or $(2.5 \times C)$ assuming 64-byte minimum packets. For a router that supports throughput at OC-192 (10 Gbps), ordering would require an effective memory bandwidth of about 25 Gbps; OC-768 (40Gbps) routing would require about 100 Gbps of effective memory bandwidth.

The SRAM size estimates in the table are made using the fact that the sorting data structure must scale with the number of packets simultaneously in the router; this number can be calculated from the characteristics of a deadline-aware packet scheduler and assuming a network of 10 hops with an end-to-end propagation delay of 10ms (the number of packets scales linearly with increasing hops and propagation delay).

The SRAM bandwidth demands of approximate sorting are particularly high. A potential feature that future architectures may want to provide in order to decrease the SRAM bandwidth requirement is a small amount of memory local to the processor, under software control. Less than 1 KB of local memory would be sufficient to hold variables local to a micro-engine that had no need to be stored in shared memory, but are spilled there for lack of register space. 11 words of SRAM access per packet (out of the current 40) could be saved in this manner, reducing the SRAM bandwidth demands by 27%.

Several other features would be useful in improving performance. One such feature is hardware support for linked lists, in the form of atomic enqueue and dequeue operations that the software can hand off to the hardware. MDS-BinSort requires 4 lock operations per packet to manipulate linked-lists, at an average cost of 916 cycles of lock latency each. Reducing this cost would help to increase MDS-BinSort's throughput. Another useful feature would be a functional unit that generates pseudo-random numbers, something that is difficult to do efficiently in software. These numbers have several uses; in terms of sorting they can be used to implement probabilistic load-balancing such as that required for MDS-BinSort. A feature that can help decrease the size of SRAM required is a hardware multiplication unit. The IXP 1200 lacks this feature, forcing any multiplication operation to have at least one operand that is a power of 2 (in order for the compiler to implement the multiplication as a shift). This has the effect of increasing the required amount of memory — arrays are forced to be the next power of 2 over their actual required size in order for the micro-engine to efficiently calculate the required memory address for an array access. The circular array used for the FIFO queue is 164% bigger than it needs to be, simply because of this requirement. The number of bins used by BinSort (which are contained in an array) is also much higher than it needs to be for the same reason.

6 End-to-end

In this section we evaluate the feasibility of each of the network architectures on the IXP 1200. We analyze the hardware requirements of the architectures in terms of number of threads, SRAM bandwidth, and SRAM size. Using current technology trends we predict the scalability of the various architectures.

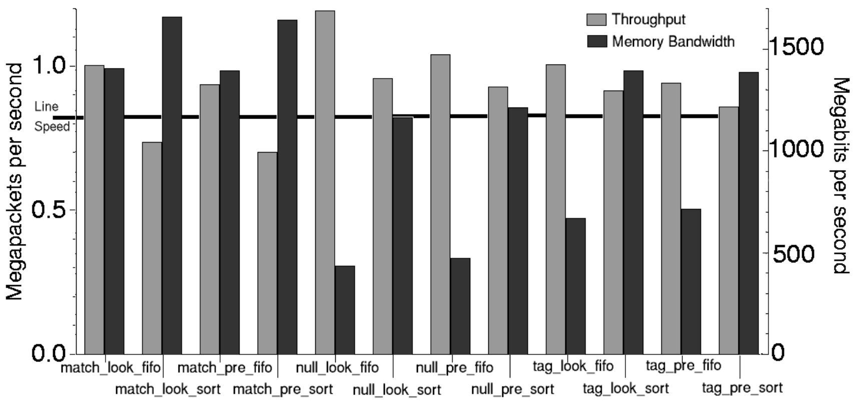


Figure 7: Throughput of different network architectures.

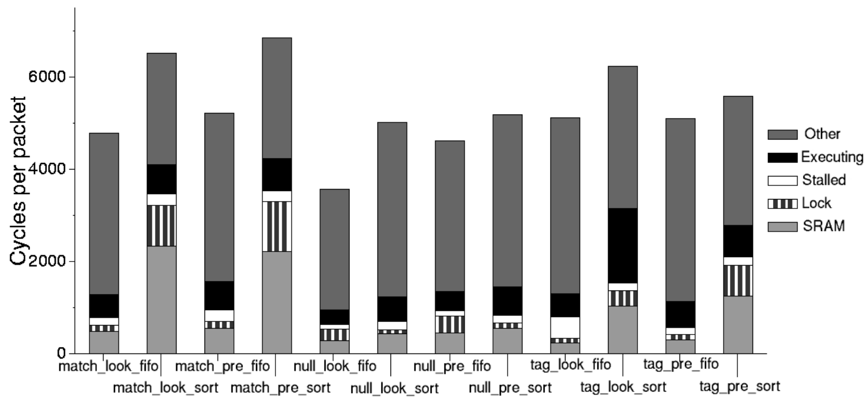


Figure 8: Throughput of different network architectures.

All but two architectures (match_look_sort and match_pre_sort) are able to meet the line speed requirements of the IXP 1200, as shown in figure 7. These two architectures map to the IntServ quality of service

Architecture	# Threads to Tolerate Latency	SRAM Bandwidth (Mbps)
match_look_fifo	$2.46\text{E}{-3} \times C$	$2.88 \times C$
match_look_sort	$3.52\text{E}{-3} \times C$	$4.63 \times C$
match_pre_fifo	$3.28\text{E}{-3} \times C$	$3.06 \times C$
match_pre_sort	$4.34\text{E}{-3} \times C$	$4.81 \times C$
null_look_fifo	$1.31\text{E}{-3} \times C$	$0.75 \times C$
null_look_sort	$2.38\text{E}{-3} \times C$	$2.50 \times C$
null_pre_fifo	$2.13\text{E}{-3} \times C$	$0.94 \times C$
null_pre_sort	$3.19\text{E}{-3} \times C$	$2.69 \times C$
tag_look_fifo	$1.97\text{E}{-3} \times C$	$1.38 \times C$
tag_look_sort	$3.03\text{E}{-3} \times C$	$3.13 \times C$
tag_pre_fifo	$2.79\text{E}{-3} \times C$	$1.56 \times C$
tag_pre_sort	$3.85\text{E}{-3} \times C$	$3.31 \times C$

Table 4: Number of threads needed to tolerate memory access latency and SRAM bandwidth as a function of link-speed (C Mbps) for each architecture.

(QoS) family of protocols. All architectures are able to meet the SRAM bandwidth limitations of the IXP 1200 (a maximum of 2.97 Gbps). All architectures except those that use prefix matching for packet routing meet the SRAM size constraint of 8 MB imposed by the IXP 1200. However, the size requirement of prefix routing does not scale with link-speed, and therefore this does not impose any restrictions on the scalability of architectures that do prefix matching (see section 4).

All 12 architectures are represented by a linear composition of the three basic building blocks. Since all of these blocks scale with increasing link-speed, it follows that all 12 architectures also scale with link-speed. However, they do this with varying costs, as shown by figure 8. The y axis of this graph shows the average number of cycles spent per packet. This value is divided into 5 sections: executing, stalled, waiting for locks, waiting for SRAM, and the remaining cycles. These remaining cycles are a combination of stalls due to SDRAM accesses, control register reads, synchronization using inter-thread signaling, and time a thread spends in the ready state. For the final version of the paper we will be able to break this combined cycle count into distinct categories. Using these numbers we will be able to express the limitations of each architecture in terms of number of microengines, number of hardware contexts per microengine, and SRAM bandwidth.

Table summarizes the memory system requirements needed to scale these protocols to increasing link bandwidths given our prototype implementations. Note that the "# Threads" column quantifies only the number of simultaneously executing threads needed to tolerate memory latency given the number of memory accesses needed per packet in each stage of processing. Because additional processing requires threads to occupy threads for additional time, additional hardware threads will be needed.

7 Related work

Problems related to building routers have been topics of great interest for the past few years. While some of them focus on specific building blocks such as classification [26] others have tried to address higher-level architectural issues [11, 13]. Recently, the problem of implementing router functions on programmable network processors has received considerable attention [23, 24]. However much of this work, has focussed on replicating conventional IP router functionality (namely, packet forwarding and switching functions). In contrast, we explore, in this paper, the problem of implementing rich network services on programmable network processors; with a particular focus on the design of networks that provide to flows end-to-end service guarantees. In what follows, we provide a brief summary of work related to the design and implementation of each of the three building blocks.

Classification Literature contains several proposals for different *dictionary data structures* targeted for solving a particular version of the general classification problem. For instance, the *Grid-Of-Tries* data structure proposed in [26] uses a version trie-of-tries designed for 2-tuple prefix matches. The *Area-Based-Quadtree* proposed in [4] improves upon the Grid-of-Tries and exports to designers the tradeoff between lookup efficiency and insert efficiency by means of a configuration parameter. The tuple space search proposed in [25] decomposes a 5-tuple prefix-matching problem into a series of exact match lookups performed using hashing. All the above algorithms can be implemented either in software or in hardware. The literature also contains several other algorithms specifically designed for hardware implementation (consider, for instance, the *Bitmap-Intersection* scheme presented in [12]). A detailed survey of the classification algorithms can be found in [8].

Route Selection This involves determining the *longest prefix-match* in the routing table for the destination host IP address. Several Trie-based schemes [26, 4] proposed in the literature are well-suited for this function.

Packet Ordering The literature contains several perfect sorting and approximate sorting data structures for ordering packet transmissions. Examples of perfect sorting data structures include Priority Queues, balanced search trees and heaps. Priority Queues, when implemented using search trees, incur $O(h)$ processing time complexity for enqueue and dequeue operations, where h is the height of the tree [7]. Unfortunately, in the worst-case, h can be as large as P, the number of elements in the priority queue. Maintaining balanced search trees or heaps ensure that the worst-case h is no larger than $\log P$. When the range of values in a Priority Queue is bounded, efficient implementations that use ideas similar to Bin-Sort or Calendar Queues [3] are possible. In fact, if the sorting accuracy can be relaxed, it is possible to design even *constant time* implementations of a Priority Queue that trade sorting accuracy for processing complexity [16, 19].

8 Conclusions

Architectures for emerging programmable router chips are complex, parallel, "I/O supercomputers" that may be able to deliver enhanced network QoS guarantees in a scalable manner. In this paper we consider two questions: (1) how do properties of next-generation network services affect programmable router architectures? and (2) how do programmable router architectures affect the design of next-generation network services?

The current IXP 1200 architecture is able to provide a range of network QoS guarantees, from traditional best-effort IP routing to Integrated Services, MPLS, or CSGS that provide per-flow delay, jitter, or bandwidth guarantees at speeds near or exceeding the router's maximum network bandwidth. However, in order to make some of these protocols work on programmable router architectures, we were forced to modify the end-to-end protocols slightly. After these modifications, the differences in costs of providing these different guarantees is relatively small, especially given the architectural challenges to scaling routers to support even traditional best-effort IP routing.

Overall, the scaling strategies for programmable routers is straightforward (but, nonetheless, challenging): add more hardware threads, memory bandwidth, and memory size roughly in proportion to increasing network bandwidth. The additional hardware threads are likely to be a combination of additional execution engines and increasing numbers of threads simultaneously sharing each engine to mask memory latencies. Enhancements such as local memories or additional hardware support for locking or list manipulation may also reduce pressure on global memory bandwidth.

References

[1] Intel®ixp2800 network processor. http://www.intel.com/design/network/products/npfamily/ixp2800.htm.

[2] R Braden, D Clark, and Shenker S. Integrated services in the internet architecture: an overview. *RFC 1363*, June 1994.

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

[4] Milind M. Buddhikot, Subhash Suri, and Marcel Waldvogel. Space decomposition techniques for fast Layer-4 switching. In *Protocols for High Speed Networks IV (Proceedings of PfHSN '99)*, pages 25–41, August 1999.

[5] D Burger and A Goodman, J Kagi. Memory bandwidth limitations of future microprocessors. In *Proceedings of 23rd International Compuer Architecture*, May 1996.

[6] K. G. Coffman and Andrew M. Odlyzko. *Internet growth: Is there a Moores Law for data traffic?*, pages 47–93. Kluwer, 2002.

[7] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. McGraw Hill, 1996.

[8] Pankaj Gupta and Nick McKeown. Algorithms for packet classification. In *IEEE Network*, pages 24–32, March/April 2001.

[9] E. Johnson and A. Kunze. *IXP1200 Programming*. Intel Press, 2002.

[10] J Kaur. *Scalable Network Architectures for Providing Per-flow Service Guarantees*. PhD thesis, Department of Computer Sciences, University of Texas at Austin, August 2002.

[11] S Keshav and R Sharma. Issues and trends in router design. *IEEE Communications Magazine*, 36(5):144–151, May 1998.

[12] T. V. Lakshman and Dimitrios Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *Proceedings of ACM SIGCOMM*, pages 203–214, 1998.

[13] T.V. Lakshman and D. Stiliadis. Beyond best effort: router architectures for differentiated services of tomorrow's Internet. *IEEE Communications Magazine*, 36(5):152–164, May 1998.

[14] Lee, M.E and Dally, W.J and Chiang, S. A 90mw 4gb/s equalized i/o circuit with input offset cancellation. In *Journal of Solid State Circuits*, November 2000.

[15] Lee, M.E and Dally, W.J and Poulton, J.W and Chiang,S and Greenwood, F. An 84-mw 4gb/s clock and data recovery circuit for serial link applications. In *Proceedings of VLSI Circuits Symposium, Kyto, Japan*, June 2001.

[16] J. Liebeherr and D.E. Wrege. Priority queueing schedulers with approximate sorting in output buffered switches. In *IEEE Journal on Selected Areas in Communications*, volume 17, pages 1127–1145, June 1999.

[17] J. Liebeherr and D.E. Wrege. Priority queueing schedulers with approximate sorting in output buffered switches. In *IEEE Journal on Selected Areas in Communications*, pages 1127–1145, June 1999.

[18] University of Oregon Route Views Project. Bgp core routing table size. 2002. http://www.antc.uoregon.edu/route-views/dynamics/.

[19] J. Rexford, A. Greenberg, and F. Bonomi. Hardware-efficient fair queueing architectures for high-speed networks. In *Proceedings of IEEE INFOCOM*, March 1996.

[20] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol label switching architecture. January 2001. Internet RFC 3031.

[21] M.A.R. Sanchez, E.W. Biersack, and W. Dabbous. Survey and taxonomy of ip address lookup algorithms. *IEEE Network*, 15(2):8–23, March 2001.

[22] S. Shenker and C. Partridge. Specification of guaranteed quality of service. Available via anonymous ftp from ftp://ftp.ietf.cnri.reston.va.us/internet-drafts/draft-ietf-intserv-guaranteed-svc-03.txt, November 1995.

[23] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb. Building a robust software-based router using network processors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 216–229, 2001.

[24] Tammo Spalink, Scott Karlin, and Larry Peterson. Evaluating network processors in ip forwarding. In *Princeton University Technical Report TR-626-00*, November 2000.

[25] V. Srinivasan, Subhash Suri, and George Varghese. Packet classification using tuple space search. In *Proceedings of ACM SIGCOMM*, pages 135–146, 1999.

[26] V. Srinivasan, George Varghese, Subhash Suri, and Marcel Waldvogel. Fast and scalable layer four switching. In *Proceedings of ACM SIGCOMM*, pages 191–202, September 1998.

[27] I. Stoica. Stateless core: A scalable approach for quality of service in the internet. *PhD thesis, Carnegie Mellon University, Pittsburgh, PA*, December 2000.

[28] Yeung, E and Horowitz, M. A 2.4 gb/s/pin simultaneous bidirectional parallel link with per-pin skew compensation. In *Journal of Solid State Circuits*, November 2000.