Building a Robust Software-Based Router Using Network Processors

Tammo Spalink, Scott Karlin, Larry Peterson, Yitzchak Gottlieb

Department of Computer Science
Princeton University
35 Olden Street
Princeton, NJ 08544

{tspalink, scott, llp, zuki}@cs.princeton.edu

ABSTRACT

Recent efforts to add new services to the Internet have increased interest in software-based routers that are easy to extend and evolve. This paper describes our experiences using emerging network processors—in particular, the Intel IXP1200—to implement a router. We show it is possible to combine an IXP1200 development board and a PC to build an inexpensive router that forwards minimum-sized packets at a rate of 3.47 Mpps. This is nearly an order of magnitude faster than existing pure PC-based routers, and sufficient to support 1.77 Gbps of aggregate link bandwidth. At lesser aggregate line speeds, our design also allows the excess resources available on the IXP1200 to be used robustly for extra packet processing. For example, with 8×100 Mbps links, 240 register operations and 96 bytes of state storage are available for each 64-byte packet. Using a hierarchical architecture we can guarantee line-speed forwarding rates for simple packets with the IXP1200, and still have extra capacity to process exceptional packets with the Pentium. Up to 310 Kpps of the traffic can be routed through the Pentium to receive 1510 cycles of extra per-packet processing.

1. INTRODUCTION

Software-based routers have always played a role in the Internet [16], but they are becoming increasingly important as the set of services routers are expected to support—e.g., firewalls, intrusion detection, proxies, level-*n* switching, packet tagging, overlay networks—continues to grow. Although software-based routers have historically been built from PC-class machines with conventional network interface cards (NICs) [13, 19], the emergence of *network processors* [8, 10, 25] makes it possible to significantly improve the performance of software-based routers at a modest increase in cost. For example, this paper describes a router, built from a PC using a 733 MHz Pentium III and an IXP1200 development board, that demonstrates nearly an order of magnitude improvement in performance over a pure PC-based router, at a cost of roughly US$1500, based on an estimated US$700 for a IXP1200 board produced in low volume.

Network processors are designed to operate under severe performance requirements. For example, a network processor assigned to an OC–48 link (2.5 Gbps) has to process up to 6.1 M minimum-sized packets-per-second (pps). Copying an OC–48 bit stream into and out of memory requires $2 \times 2.5\,\text{Gbps} = 5\,\text{Gbps}$ of memory bandwidth. Network processors commonly employ parallelism to hide memory latency. For example, the Intel IXP1200 contains six MicroEngines, each supporting four hardware contexts. The intention is that during regular execution one of these contexts is doing real work while the others are blocked on (hiding) a memory operation. The IBM PowerNP and Vitesse IQ2000 use similar designs [8, 25].

This paper describes the design and implementation of a software-based router that uses the IXP1200 network processor. The router implements both the *data plane* that forwards packets, and the *control plane* where signalling protocols like RSVP, OSPF, and LDP run. On a pure PC-based router, both the data and control planes are implemented on the control processor. With the IXP1200, it is largely possible to separate the two, with the data plane running on the network processor and the control plane running on the Pentium. The full story, however, is a bit more complicated, and is the subject of this paper.

One distinction between the data and control planes is that the former must process packets at line speed, while the latter is expected to receive far fewer packets (e.g., whenever routes change or new connections are established). The requirement that the data plane runs at line speed is based on the need to receive and classify packets as fast as they arrive, so as to avoid the possibility of priority inversion; i.e., not being able to receive important packets due to a high arrival rate of less important packets. The expectation that the control plane sees significantly fewer packets is only an assumption. It is possible to attack a router by sending it a heavier load of control packets than it is engineered to accept.

A second distinction between the data and control planes is how much processing each packet requires. At one extreme, the data plane does minimal processing (e.g., IP validates the header, decrements the TTL, recomputes the checksum, and selects the appropriate output port). At the other extreme, the control plane often runs compute-intensive programs, such as the shortest-path algorithm to compute a new routing table. However, these are just two ends of a spectrum. In between, different packet flows require different amounts of processing, such as evaluating firewall rules, gathering packet statistics, processing IP options, and running proxy code. Note that this processing can happen in the data plane, in the sense that it is applied to every packet in a particular flow.

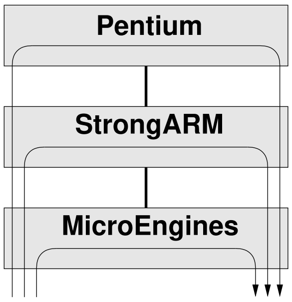

Figure 1: Three switching paths through the Pentium/IXP1200 processor hierarchy.

Taking both packet arrival rates and per-packet processing costs into account, the key is deciding where on the router each processing step should run. Our approach is to treat the router as a *processor hierarchy*, where packets follow switching paths that traverse different levels of the hierarchy. Figure 1 shows the three-level hierarchy corresponding to our prototype hardware. At the lowest level, packets traverse only MicroEngines, while at the highest level, packets are processed by the Pentium. An intermediate level corresponds to a StrongARM processor on the IXP1200 chip. At each level of the hierarchy, the packet has access to some number of cycles, but there is overhead involved in reaching those cycles. Higher levels (e.g., the Pentium) offer more cycles, but packets also consume resources at lower levels of the hierarchy to access them. Lower levels (e.g., the MicroEngines) have enough cycles to perform only certain operations at line speeds.

The main contribution of this paper is to address the resource allocation and scheduling problems of implementing an extensible router on a three-level processor hierarchy. Our approach is guided by three goals:

- **Performance:** The router should be able to forward packets at the highest rate the hardware is able to support. The challenge is to manage the parallel hardware contexts in a way that fully utilizes the available memory bandwidth. This is difficult for two reasons. First, we must assign work to each context so as to effectively exploit the system's parallelism. Second, we must avoid allowing synchronization among the hardware contexts to become the limiting factor.

- **Extensibility:** It should be easy for a trusted entity to inject new functionality into the router, including both new control protocols and code that processes each packet forwarded through the data plane. The challenge in supporting extensibility is defining the interface by which the control program interacts with the code running in the data plane.

- **Robustness:** The router should continue to behave correctly regardless of the offered workload or the extensions it runs. The challenge is to simultaneously support our performance and extensibility goals, or said another way, the system must ensure that the performance of the various components are isolated from each other. For example, it should not be possible to inject code into the data plane that keeps the router from processing packets at line speed, and likewise, it should not be possible for a high packet arrival rate to choke off the delivery of control packets to the control plane.

We describe the design in two stages. First, we show how to program the processor hierarchy with a fixed forwarding function that fully exploits the parallelism available on the IXP1200, as well as the StrongARM and Pentium processors (Section 3). This discussion focuses on the performance limits of the processor hierarchy, that is, how fast each level can forward packets that require no extra processing. Second, we describe how the system can be made extensible without violating the router's ability to process minimum-sized packets at line speed (Section 4). Throughout the paper, we highlight the design decisions that impact the router's ability to provide performance isolation.

2. ARCHITECTURE

This section describes our software and hardware architectures. In the case of the software architecture, our starting point is a communication-oriented OS that runs on a Pentium with non-programmable NICs [12, 19], to which we add a device driver and IXP microcode. This section gives a high-level overview of the original Pentium-based system; later sections focus on those aspects of the architecture that are relevant to a multi-level processor hierarchy (i.e., the driver and microcode components).

2.1 Software

Figure 2 depicts the software architecture for the router. A *classifier* (C) first reads packets from an input port, and based on certain fields in the packet header, selects a *forwarder* (F) to process the packet. Each forwarder then gets packets from its input queue, applies some function to the packet, and sends the modified packet to its output queue. All transformations of packets in the router occur in forwarders. Finally, an *output scheduler* (S) selects one of its non-empty output queues, and transmits the associated packet to the output port. The scheduler performs no processing on the packet.

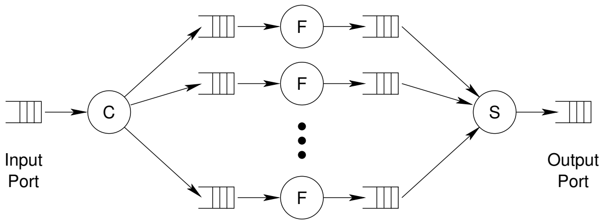

Figure 2: Classifying, forwarding, and scheduling packets.

This architecture has two main attributes. First, it provides explicit support for adding new services to the router. Although the router boots with two default forwarders (one that implements a minimal IP forwarding fast path and one that implements the full IP protocol, including options), additional forwarders can be installed at runtime (e.g., TCP proxies, specialized overlays, and support for virtual LANs). A new forwarder is installed by specifying a demultiplexing key that the classifier is to match and binding that key to the forwarder and some output port. Just to re-emphasize the point, the core architecture supports a generic forwarding infrastructure; even basic IP functionality is treated as an extension.

Second, the architecture does not specify where in the processor hierarchy each forwarder is implemented: some run on the MicroEngines, some on the StrongARM, and some on the Pentium. Note that the architecture does not distinguish between forwarders that implement traditional control protocols and forwarders that would normally be considered on the data plane, although it is likely that the former would be mapped to higher levels of the processor hierarchy and the latter to lower levels of the hierarchy.

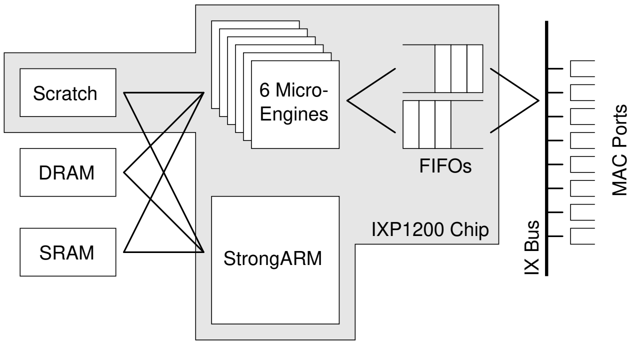

Figure 3: Block Diagram of an IXP1200 Evaluation System

2.2 Hardware

Our router runs on a PC using a 733 MHz Pentium III processor with the IXP1200 evaluation system illustrated in Figure 3 plugged into one PCI slot. The board consists of a IXP1200 network processor chip (shaded area), 32 MB of DRAM, 2 MB of SRAM, 4 KB of on-chip Scratch memory, a proprietary 64-bit 66 MHz IX bus, and a set of media access controller (MAC) chips implementing ten Ethernet ports (8×100 Mbps $+ 2 \times 1$ Gbps). Not shown is a 32-bit 33 MHz PCI bus interface.

The IXP1200 chip itself contains a general-purpose StrongARM processor core and six special-purpose MicroEngine cores all running at 200 MHz (5 ns cycle time).[1] Each of the six MicroEngines supports four hardware contexts for a total of 24 contexts. Not shown in the figure is a 4 KB instruction store (ISTORE) associated with each MicroEngine. The StrongARM is responsible for loading these MicroEngine instruction stores. As for the StrongARM itself, it fetches instructions from a 4 KB I-cache backed by the IXP's DRAM.

The chip also has a pair of FIFOs used to transfer packets to and from the network ports across the IX bus. These are not true hardware FIFOs in the sense that each has a single input, a single output, and no address lines; rather, each "FIFO" is an *addressable* 16 slot $\times$ 64 byte register file. It is up to the programmer to use these register files so that they behave as FIFOs.

Although not explicitly prescribed by the architecture, the most natural use of the DRAM is to buffer packets. This is a function of size (32 MB), but also of speed. The DRAM is connected to the processor by a 64-bit $\times$ 100 MHz data path, implying a potential to move packets into and out of DRAM at 6.4 Gbps. In theory, this is sufficient to support the $2 \times (8 \times 100\,\text{Mbps} + 2 \times 1\,\text{Gbps}) =$ 5.6 Gbps total send/receive bandwidth of the network ports available on the evaluation board, although this rate exceeds the 4 Gbps peak capacity of the IX bus. Similarly, SRAM is a natural place to store the routing table, along with any necessary per-flow state. The SRAM data path has a peak transfer rate of 32-bit $\times$ 100 MHz = 3.2 Gbps.

3. FIXED INFRASTRUCTURE

This section describes and evaluates the fixed infrastructure needed to forward minimal-sized packets through the system. It assumes no packet processing, that is, we run only a null forwarder. Because we do not consider actual forwarders (including the forwarder that implements IP) until Section 4, this discussion is largely independent of IP, and so applies equally well to a router that supports, for example, MPLS [4].

[1] Actual speed is 199.066 MHz.

This section has two goals. One is to establish a performance envelope for each level of the processor hierarchy. The second is to describe enough of the implementation to establish the validity of the performance numbers. This is easy for the StrongARM and Pentium, which have familiar architectures. However, the MicroEngines offer a unique challenge, so we begin by describing how we managed their parallel contexts. Although the description is necessarily tied to the details of the IXP1200, we believe the engineering decisions we made apply generally to any *parallel, software-based switch*. It came as a surprise (but should not have) that many of the issues we faced have direct analogs in managing hardware switching fabrics, which are inherently parallel.

3.1 Forwarding Pipeline

The common unit of data transferred through the IXP1200 is a 64-byte MAC-Packet (MP). As each packet is received, the MAC breaks it into separate MPs; tags each MP as being the first, an intermediate, the last, or the only MP of the packet; and stores the MP in an input FIFO slot. Similarly, the individual MPs that make up a packet must be loaded into output FIFO slots to be transmitted by the MAC. Since only a fixed number of input and output FIFO slots are available (16 of each), it is necessary to allocate slots to MAC ports, and it is the responsibility of the forwarding code running on the MicroEngines to drain the input slots and fill the output slot at a rate that keeps pace with each port's line speed.

While one might naively think that the MicroEngines could move MPs from input FIFO slots directly to output FIFO slots in a single step, forwarding actually requires a two-stage pipeline. This is because port contention—two or more incoming packets destined for the same output port—makes it impractical for a single context to forward a packet. Instead, packets are placed into queues, and these queues are serviced asynchronously [15]. Using different contexts for each stage prevents MicroEngines from being idle during the time a packet is queued. The two pipeline stages are implemented using disjoint sets of MicroEngine contexts, with MPs transfered between the stages via DRAM.

Figure 4 summarizes the forwarding pipeline. It shows the queues used to pass packets between the contexts that perform input processing and the contexts that perform output processing as being implemented using a combination of SRAM and DRAM. This is because packet contents are buffered in DRAM, while SRAM holds the actual queue data structure (each element in a queue is the address in DRAM where the packet is buffered).

3.2 Input Processing

Figure 5 gives pseudo-code for the loop executed once for each MP received. In the figure, *p* denotes the port number on which the MP arrived, *c* is an index in the input FIFO, *mp_addr* is the address in memory where the contents of the MP is stored, *reg_mp_data* denotes the MicroEngine registers that hold the MP, and *state* is a data structure containing information about how the MP should be processed.

The first set of operations (lines 1–4) determine whether port *p* has a new MP available. If so, the load operation instructs a DMA state-machine to copy the MP from the off-chip port memory into the on-chip input FIFO. There is only one DMA state-machine on the IXP1200 and requests to it are not hardware-serialized. Thus, the mutex operations are needed to allow multiple MicroEngine contexts to safely execute input loops in parallel.

Once the MP is in the FIFO, the MicroEngine copies the MP into its registers for protocol_processing, which is performed in-line and includes all protocol-specific packet header or content modifications. In terms of the software architecture described in Sec-

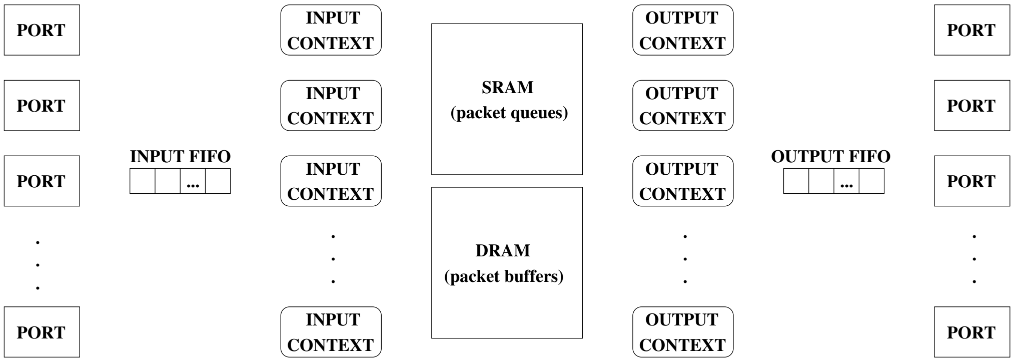

Figure 4: Forwarding Pipeline.

```
    INPUT_LOOP:
1       acquire_input_mutex()
2       if (! port_rdy(p)) goto INPUT_LOOP
3       load IN_FIFO[c]
4       release_input_mutex()
5       mp_addr = calculate_mp_addr()
6       copy reg_mp_data ← IN_FIFO[c]
7       state = protocol_processing(reg_mp_data)
8       copy reg_mp_data → DRAM[mp_addr]
9       if (at_start_of_packet(state))
10          enqueue(state, state.queue)
11      goto INPUT_LOOP
```

Figure 5: Pseudo-code running in each context assigned to input processing.

tion 2.1, this includes both the classifier and the forwarder. In terms of IP, as a specific example, this involves validating the header, decrementing the time-to-live (TTL) field, recomputing the checksum, setting the destination MAC address to the one found in the routing table, and setting the source MAC address to that of the output port. For the purpose of this section, the protocol processing step includes a trivial classifier (it selects the output port based on the destination IP address), and the simplest possible forwarder (it only modifies the destination MAC address). It does no other work.

Although protocol_processing is called once for every MP, and thus many times for large packets, the processing of the first MP in a packet must determine the destination of the packet. This results from the pipeline structure of our router. Once the input stage has produced the first MP of a packet, the output stage may start processing it immediately and must have complete destination information available. Protocol processing is performed for each MP to facilitate operations that modify the entire packet, or that manipulate packet headers lying deeper into the packet.

After protocol processing, the (possibly modified) MP is copied from registers to DRAM. If the MP is the first or only MP of a packet, it is assumed to include the packet headers and the results of packet processing must specify the destination queue of the packet. For these MPs, the packet processing results and some identification information for the packet are then enqueued in the destination queue.

Exceptional packets, for example those that incur a miss in the routing table or involve additional processing (e.g., IP options), receive all of the same processing, but they can be placed in a queue that is serviced by the StrongARM instead of the usual output process. Responsibility for eventually passing these packets to output processing is also passed on to the StrongARM. We discuss this case in more detail in Section 3.6.

3.2.1 Context Scheduling

Since it is impossible to fully predict packet traffic or arrival times, for the sake of robustness we must assume that packets arrive at line speed. This means we must be able to execute the input loop once for each MP at the maximal rate the system is being designed to support.

For simplicity, our design uses only one version of the input forwarding stage and this version is run for each MP. This means that the resources used for input processing are shared evenly across arriving packets and that each packet has the same functions applied to it, that is, it has the same processing options available to it. We do this simply by statically allocating a set of MicroEngine contexts to run only the input loop. This set of contexts must be sufficiently large to meet line speed requirements.

Contrast this approach with the alternative of dynamically allocating MicroEngine contexts to various processing steps on an as-needed basis, possibly with a number of different input stages customized to specific protocols. One downside of this dynamic approach is that additional resources are needed to make the scheduling decision; i.e., decide which context will perform what work next. On the IXP1200 where most inter-process communication involves memory access, our experience is that this rapidly results in memory delays that degrade performance. In general, this approach can be viewed as adding forwarding pipeline stages that perform some scheduling or other classification operations.

Once a static assignment of contexts to input processing is used, scheduling the 16 input FIFO slots becomes straightforward. The pseudo-code statically uses c as a FIFO address (slot number). By constraining input processing to use at most 16 of the 24 available contexts, we have a simple assignment of FIFO slots to contexts. Since there are only 24 contexts, and some still need to be available for output processing, this restriction has not been a problem in our experience.

3.2.2 Mutual Exclusion

The mutex operations in pseudo-code lines 1–4 are implemented using token passing. This token passing uses an inter-thread signaling mechanism provided in hardware by the IXP1200. Importantly, this signalling mechanism is on-chip, takes a single cycle, and is disjoint from the memory. This means that it neither introduces much overhead nor interferes with the already significant contention for memory.

Token passing can be viewed as a simple scheduler that serializes contexts accessing the input DMA. The order of DMA access is made explicit by the order in which the token is passed to maximize the possibility for useful concurrent work, thereby minimizing contention delays. Specifically, we rotate the token so that a context on one MicroEngine always hands the token to a context on another MicroEngine. Passing the token to another context on the same MicroEngine potentially results in two contexts on the same MicroEngine having useful work to do, but only one of them gets to run at a time. Similarly, we assign ports to contexts in such a way that the two contexts servicing the same port are as far apart as possible in the token rotation, thereby maximizing the time one context has to service the port before the next context gets the token.

3.2.3 *Buffer Allocation*

The pseudo-code calculate_mp_addr operation hides the complexity of packet buffer allocation. Because a relatively large amount of DRAM is available on our development board, we elected to use a very simple allocation scheme. 16 MB of DRAM are divided into 8192 buffers of 2 KB each, making each buffer large enough to accommodate a maximally sized (1518 octet frame) Ethernet packet. These buffers are then consumed by input processing contexts in a circular fashion as packets arrive.

The state variable that tracks the next available buffer is shared among the input contexts, and in principle needs to be protected from unsynchronized concurrent access. In our implementation, however, the serialization created by the token passing mechanism allows us to avoid explicitly protecting this operation.

Our allocation strategy has one interesting property: Any given packet buffer remains valid for only one pass though the circular buffer list. Since input processing takes a fixed amount of time per packet, the lifetime of a buffer can be calculated precisely. If a packet is not transmitted by the output process before its buffer is reused, the packet is effectively lost.

At some additional cost, this timing behavior could be eliminated by using hardware support on the IXP1200 for stack operations to implement a buffer pool. To prevent contention from causing shortages, it would be necessary to have a different stack of available buffers for each output port. Since this is not strictly necessary and adds overhead, we chose not to implement this feature.

3.3 Output Processing

Figure 6 gives pseudo-code for packet transmission. For each output port, the select_queue operation chooses a non-empty queue from among the set of queues associated with that port. In terms of the software architecture described in Section 2.1, this is the output scheduler. A packet descriptor is then dequeued from the chosen queue. For each MP of the packet, the DRAM address of the MP is calculated, an available output FIFO slot is selected, the MP is copied from DRAM to the FIFO, and the FIFO slot is activated to schedule a DMA from the on-chip FIFO memory to the actual off-chip port memory.

The hardware interface to the output and input FIFOs are not identical. Unlike the input FIFO, the slots of the output FIFO are strictly ordered and the DMA machine that moves data from the FIFO to network device memory consumes the slots in a circular fashion.

This means that if more than one context is running the output loop concurrently, they need to cooperate to obey the FIFO ordering. This can be done in several ways. We have chosen to statically allocate FIFO slots to output contexts in a very similar manner in which input FIFO slots were allocated to input contexts. Thus, the calculate_fifo_addr operation always returns the same value (on a per-context basis).

To serialize output contexts, we use a token passing loop identical to that used by the input process. This is reflected in the first two lines of the pseudo-code. It is necessary to make sure each context activates its FIFO slots each time around the loop to match the FIFO slot ordering. This happens even if no data is available, resulting in garbage data sent to a non-existent port.

```
OUTPUT_LOOP:
    acquire_output_mutex()
    release_output_mutex()
    if (finished_last_packet)
        qid = select_queue()
        state = dequeue(qid)
        mp_addr = first_mp(state)
    else
        mp_addr = next_mp(state)
    fifo_addr = calculate_fifo_addr()
    copy DRAM[mp_addr] → OUT_FIFO[fifo_addr]
    enable IN_FIFO[fifo_addr]
    finished_last_packet = at_end_of_packet(state)
    goto OUTPUT_LOOP
```

Figure 6: Pseudo-code running in each context assigned to output processing.

3.4 Queuing Discipline

For each packet, protocol processing on the first MP chooses a destination queue for the whole packet. In our system, queues are contiguous circular arrays of 32-bit entries in SRAM. Head and tail pointers are simply indexes into the array, and they are stored in Scratch memory. Buffer pointers are inserted into the queue at the head and removed at the tail.

3.4.1 *Packet Scheduling*

Each output port must have one or more queues associated with it. During each iteration of the output loop for some port, all queues associated with that port must be examined for ready packets. This involves reading the current queue head pointer and comparing it with the tail.

To avoid both additional synchronization costs and reading the tail pointer from memory, queues are assigned statically to output contexts. This allows the output contexts to keep the queue tail pointers in registers and saves multiple memory operations on each loop iteration. However, this restricts the number of queues that each context can service to a maximum of 16, the number of available registers.

If multiple queues are assigned to a single output context, the context may occasionally have multiple packets available for transmission. Choosing which packet to service next is a policy decision that determines the overall packet schedule. This decision is reflected by the pseudo-code select_queue operation. When multiple queues are used, our implementation prioritizes the queues, such that each context drains its queues in priority order. However, any other policy implementable with little computation, which does not require looking deeper into the queues, can be used (e.g., round robin).

If more complex packet scheduling policies are needed, they must be implemented by the input contexts. When multiple queues are available at each output context and when these have fixed priority levels, the larger computing capacity available in input-side protocol processing could be used to select the appropriate priority queue and thereby approximate more complex schemes, such as weighted fair queuing. We have not evaluated this in detail.

3.4.2 Queue Contention

There are two simple approaches to manage contention among input contexts for accessing output queues. The first approach is to protect each output queue with a mutex lock. Since such locks are normally implemented using the memory subsystem, great care must be taken to avoid unpredictable behavior during lock contention. For example, the MicroEngines have a test-and-set instruction that can be used to implement a lock using a tight test-until-acquired loop. However, our experiments with this strategy reveal performance-crippling memory contention when many contexts attempt to acquire the lock at the same time. Fortunately, the IXP1200 also has hardware mutex support for mutually exclusive access to special SRAM regions. Because these operations are blocking, they do not suffer from the same problem.

A second approach is to avoid having input contexts share queues, thereby entirely avoiding the need to synchronize. This is achieved by statically assigning queues to input contexts, effectively giving each input context a private set of queues for each port. The downside of this approach is that each output context must now service many more queues: one per priority level, per input context. Since we have 16 input contexts and each output context has only 16 registers to hold queue pointers, this effectively limits us to a single priority level for each output port.

3.4.3 Optimizations

When output contexts service multiple queues, the latency introduced by the memory accesses needed to check the head pointers to determine which queues have new packets can quickly make output performance unacceptable. This can be mitigated by adding a level of indirection that summarizes queue readiness information. Instead of checking many queues, the output process checks a single bit-array of queue readiness flags. These can be set without extra synchronization by the input contexts using a special bit-set memory access mode provided in the MicroEngine instruction set.

Alternatively, if each output context services only a single queue, memory accesses to the queue head pointer might be avoided by batching packet transmissions. To be more specific, if the queue head pointer is checked and if there are many ready packets in the queue, all of the ready packets can be sent before it is necessary to look at the head pointer again. This can avoid a number of memory accesses during periods of high load.

3.4.4 Queue Port Mapping

If multiple MicroEngine output contexts are servicing queues for the same output port, additional synchronization is required to ensure that all of the MPs for one packet are sent before those of the next packet. Our development board has enough ports that our prototype router can statically allocate ports to contexts. However, if fewer ports were used, the problem could be avoided by more complex and careful allocation of the output FIFO slots. We have not evaluated this in more detail since it is not a problem for our hardware configuration.

3.5 MicroEngine Evaluation

This section reports the results of several experiments designed to evaluate how well our implementation takes advantage of the parallel resources. It also reports some of our experiences working with with IXP1200.

3.5.1 Performance

We initially measured the system using the 8×100 Mbps Ethernet ports on the evaluation board using eight Kingston KNE100TX PCI Ethernet cards based on the 21143 Tulip chip-set as traffic sources. (A pair of these cards are plugged into each of four 450 MHz Pentium IIs running a packet generator.) When configured to generate minimum-sized (64-byte) packets, each card transmits 141 Kpps, which is 95% of the theoretical maximum of 148.8 Kpps (calculated from [9]). Given this traffic source, the MicroEngines are able to sustain line speed across all eight ports, resulting in a forwarding rate of 1.128 Mpps. This is an expected result as the theoretical forwarding capacity of the processing and memory resources in the IXP1200 is much greater than the 800 Mbps of testbed traffic.

To determine the maximum forwarding rate of the IXP1200, independent of the ports configured onto the board, we modified the input process to move a single packet from a port to each FIFO slot. Future iterations of the input process see this same packet without port interaction, although we do still incur the overhead of acquiring the mutual exclusion lock. This lets us measure the maximum system performance from FIFO to FIFO, emulating infinitely fast network ports.

We need to qualify this experiment in two ways. First, although no forwarder is run in this experiment, the protocol_processing step in Figure 5 does perform packet classification based on the destination IP address. It does this using a one-cycle hardware hash of this address, and we assume a hit in a route cache. One consequence of this implementation is that the performance we report is what one would expect in the common case for a virtual circuit-based switch, such as one that supports MPLS. Second, we elected to not include any device interaction in the experiments because each device is different; e.g., the 1 Gbps ports behave differently than the 100 Mbps ports, and the next version of the board promises still different behavior. However, omitting the device interaction does not have a significant impact on the performance numbers presented in this section as even the worst case device (the 100 Mbps ports) accounted for less than 10% of the total per-packet delay.

Because queueing packets is the primary complexity in the router, and the router's performance is greatly influenced by the queueing discipline selected, we measured several combinations of queueing strategies. Table 1 lists the options that we analyzed. For these experiments, the system was configured with 4 MicroEngines (16 contexts) running the input loop and 2 MicroEngines (8 contexts) running the output loop. All 24 contexts were executing their assigned loop for all the measurements.

Input Processing (4 MicroEngines)	
(I.1) private queues in regs	3.75 Mpps
(I.2) protected public queues no contention	3.47 Mpps
(I.3) protected public queues max. contention	1.67 Mpps
Output Processing (2 MicroEngines)	
(O.1) single queue with batching	3.78 Mpps
(O.2) single queue without batching	3.41 Mpps
(O.3) multiple queues with indirection	3.29 Mpps

Table 1: Maximum packet rates broken down by input and output process, and by queueing discipline.

The results for input and output are presented separately to highlight which stage becomes the bottleneck for each option. For example, a configuration with a single queue at every output port with batching (O.1) must be combined with a protected queue access mechanism on the input side (I.2 or I.3), and the system will then be paced by the slower input process. However, it might be reasonable to chose private queues for each input context (I.1), although this forces use of the multiple queueing support on the output side (O.3), which runs at a slower rate.

The fastest feasible system (I.2 + O.1) is able to forward packets at a rate of 3.47 Mpps. This result corresponds to the situation where no two packets are destine for the same queue at the same time, and so represents an upper bound on performance. Row I.3 corresponds to the same configuration, but this time with all packets destine for the same output queue. Note that this configuration (independent of the workload) does not support QoS since a single queue is associated with each output port. In contrast, configuration I.2 + O.3 corresponds to a system that supports up to 16 queues for each output port, providing significant flexibility in differentiating service. It is able to forward packets at a maximum rate of 3.29 Mpps.

	Reg-only	DRAM 32 Byte	SRAM 4 Byte	Scratch 4 Byte
Input	171	(0 / 2)	(2 / 1)	(2 / 4)
Output	109	(2 / 0)	(0 / 1)	(2 / 2)
Total	280	(2 / 2)	(2 / 2)	(2 / 6)

Table 2: Instruction counts for processing one MP, broken down by input and output processing and by type of memory involved. Memory operations are further broken down (read / write).

Memory	Transfer Size (bytes)	Read (cycles)	Write (cycles)
DRAM	32	52	40
SRAM	4	22	22
Scratch	4	16	20

Table 3: MicroEngine cycle times to transfer common-sized data blocks into and out of various memories from the MicroEngines.

Table 2 provides detailed counts of what operations are performed by the input and output stages for configuration I.2 + O.1. Operations are broken down into simple register-based operations that generally take a single cycle to execute, and memory operations that take much longer. Table 3 gives the measured latency for reading/writing each of the three memories. The latency is given in MicroEngine cycles, each of which is approximately 5 ns. Note that each transfer moves a different number of bytes; Table 3 gives the most common transfer size for each case.

Given these instruction counts, each packet requires 280 cycles of registers instructions, plus 180 (DRAM) + 90 (SRAM) + 160 (Scratch) = 430 cycles of memory delay, which totals to 710 cycles. This means that a given packet experiences 3550 ns of delay as it is forwarded by one or more contexts running at 200 MHz. Since the system as a whole is able to forward 3.47 Mpps—that is, it outputs a packet every 288 ns—the system is able to forward a little over 12 packets in parallel.

Looking at the numbers another way, suppose we ignore memory access times and assume that all memory operations complete in one cycle. In reality this is unreasonable because it assumes the hardware memory hiding techniques work perfectly; contention for buses and other synchronization cost make this unlikely. We calculate that one MicroEngine can process 200 MHz / 280 cycles = 714 Kpps for a system total of 4.29 Mpps. Our actual rate of 3.47 Mpps is 80% of this optimistic upper bound. In other words, we are within 20% of the maximal possible performance (for a system with our instruction counts).

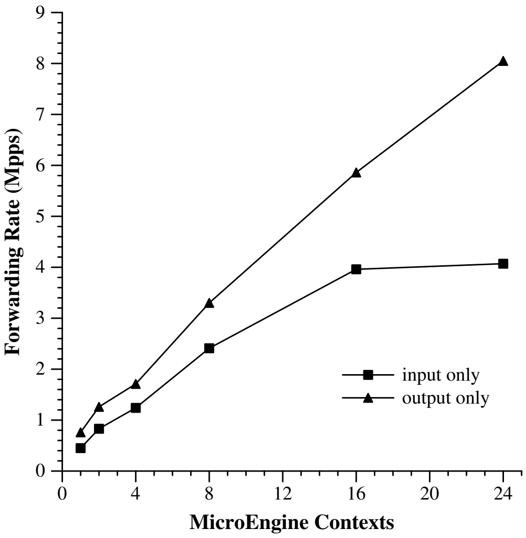

Figure 7: Maximum packet rates achievable by the output and input processes when running independently. For each datapoint only the minimum number of MicroEngine are used, hence the dip in the graph.

Finally, Figure 7 provides some insight into how a system that chooses not to use our 4/2 MicroEngine breakdown might function. This figure illustrates that output processing scales almost perfectly with the number of MicroEngines added to this stage. However, input processing benefits very little from more than 16 contexts. This is because of the serialized access to the DMA state machine, which dominates the performance of input processing once there are enough threads to keep it busy. Note that to prevent interference in these experiments, input or output contexts were running exclusively at any time, never both together. To run the output process without input, a single additional instruction was added to fool the process into believing data was always available. Also, only the minimum number of MicroEngines to support the listed number of contexts were used, e.g., 1 MicroEngine for 1-4 contexts, 2 for 8, 4 for 16, and 6 for 24. This explains the small "dent" in the bottom of the graphs.

3.5.2 Experiences

The IXP1200 is not an easy system to program. We had several false starts in the overall approach to managing the parallel resources, and all the code is written in assembly language—there is no compiler for the MicroEngines. Regarding the overall approach, our experience strongly suggests that a static allocation of resources (i.e., statically assigning tasks to contexts and contexts to ports), coupled with implicit scheduling through the token passing mechanism (which plays the dual role of protecting the shared DMA state machine) yields the most effective design for a router. Regarding the lack of a compiler, while we are aware of several efforts to address this limitation, we are not convinced that they will prove all that helpful, at least for specialized applications like packet forwarding. This is because our ability to achieve good performance depends on having complete knowledge of how registers are allocated. It is also the case that the most error-prone piece of code is the sequence of instructions that deals with the FIFOs and DMA state machine, which is necessarily written in assembly.

Unfortunately, our static approach means that the software needs to be re-designed for boards configured with different ports and port speeds. This is especially problematic for a non-homogeneous set of ports. Having to write this code in assembly language only complicates the situation. We see at least three possible solutions to this problem. One would be to create a hardware indirection stage between the actual ports and the IX bus that connects the ports to the IXP1200. This indirection stage would present a single "virtual" port to the IXP1200. Of course this does not solve the problem, it just moves it from software into hardware, which seems like the wrong thing to do considering the solution must deal with a heterogeneous and varying number of ports.

A second solution would be to have the ports transfer packets directly to and from DRAM, bypassing the FIFOs. Being able to buffer packets in memory would free the MicroEngines from having to service the FIFO slots at varying line speeds; they would only have to keep pace with the aggregate rate. The problem with this solution is that it forces four memory accesses for each byte of a minimal-sized packet: port-to-DRAM, DRAM-to-registers, registers-to-DRAM, and DRAM-to-port. This is not a problem for longer packets, where only the header has to be brought into MicroEngine registers for processing, but it does halve the maximum achievable throughput rate for 64-byte packets. One of our early implementations used this general strategy, and saturated DRAM while forwarding 2.69 Mpps. An alternative would be to provide a larger register bank as a staging area for packets (recall that each FIFO is really just a 16-register bank), but it is not clear that doing so offers a full solution, or only makes the problematic case less likely.

The third solution would be to construct the software for a new port configuration from a collection of building block components. This could ultimately result in a domain-specific compiler. Our current implementation takes the first step in this direction by defining a set of macros that can be used in different combinations. The hard part is knowing how to partition the resources (contexts and FIFO slots) in the most effective way for a given configuration. We are currently developing a resource model that supports this third approach.

3.6 StrongARM

The StrongARM is able to directly access DRAM, so packets are available for it to compute on with minimal additional overhead. The only latency is the cost of a MicroEngine signalling the StrongARM to inform it that a packet is available.

An input context processes the packet as usual, but upon detecting that the packet requires service by the StrongARM (e.g., there is a miss in the route cache or the packet contains IP options), it enqueues the packet in a StrongARM-specific queue instead of a queue assigned to an output port. At this point, we have two options: interrupt the StrongARM or let the StrongARM poll to see if any packets have arrived. In both cases, the StrongARM dequeues the next packet from this queue, performs whatever processing is required, and places the packet on the appropriate output output queue.

We measured the maximum rate that the StrongARM can process packets by having it run a null forwarder, with the input contexts programmed to pass all their packets to the StrongARM. With this configuration, we achieve a maximum forwarding rate of 526 Kpps using polling; interrupts where significantly slower. By adding a delay loop that counts to a pre-determined value, we conclude that the StrongARM has no additional cycles available to compute on packets when receiving them at this rate.

3.7 Pentium

We move packets between the IXP1200 and the Pentium over the PCI bus. Our implementation uses the IXP1200's DMA engine, plus queue management hardware registers supporting the Intelligent I/O (I_2O) standard [11]. For each logical queue from the IXP1200 to the Pentium—where each logical queue corresponds to the use of the term "queue" throughout the rest of this section—the implementation uses a pair of I_2O hardware queues. One queue contains pointers to empty buffers in Pentium memory, and the other contains pointers to full buffers in Pentium memory. On the IXP1200 side, putting a packet onto a logical queue involves first pulling a pointer to a free buffer from the free I_2O queue, filling the buffer using the DMA, and then placing the pointer on the full I_2O queue. On the Pentium side, getting a packet from a logical queue involves first retrieving a pointer from the full I_2O queue, processing the packet, and then returning the block to the free pool by placing its pointer on the free I_2O queue. Moving packets from the Pentium to the IXP1200 works in an analogous way, and involves a second pair of I_2O queues.

We measured the maximum rate that the Pentium can process packets by having it run a loop that reads packets of various sizes from the IXP1200, and then writes the packet back onto the IXP1200. (Due to a silicon error, the I_2O mechanism does not work. We therefore had to simulate it in software.) The StrongARM is programmed to feed packets to the Pentium as fast as possible. We also inserted a delay loop on both sides to determine the number of spare cycles available, that is, cycles not involved in the data transfer. The results are given in Table 4, which shows that the router is able to forward up to 534 Kpps through the Pentium. This rate saturates the StrongARM, but leaves 500 cycles per packet available on the Pentium.

Packet Size (Bytes)	Rate (Kpps)	Pentium (Cycles)	StrongARM (Cycles)
64	534.0	500	0
1500	43.6	800	4200

Table 4: Measured Maximum Forwarding Rate and Excess Per-Packet Processor Cycles.

Note that up to this point we have focused on 64-byte packets. This is because processing minimal-sized packets is the worst-case scenario. It is also the case that forwarding larger packets scales linearly on the MicroEngines: forwarding a 1500-byte packet involves forwarding twenty-four 64-byte MPs. Crossing the PCI bus is different, however, since the DMA engine runs concurrently with the StrongARM. Also note that even if 1500-byte packets arrive, we do not necessarily need to move them across the PCI bus, as many forwarders just need to inspect the packet header. To account for this likelihood, we move just the first 64-bytes across the PCI bus, along with an 8-byte internal routing header that informs the Pentium of (1) the classification decision made on the IXP1200, and (2) how to retrieve the rest of the message (lazily) should the forwarder running on the Pentium need to access the packet body.

4. EXTENSIBILITY

The previous section establishes the maximum rate that each of the three processors can forward packets with a null forwarder. This section evaluates adding more complex forwarders to the data plane and integrating the control plane into the system. It also demonstrates the robustness of the system, that is, how the extent to which the performance of the different layers of the processor hierarchy are isolated from each other.

4.1 Design Issues

We start by considering the processor hierarchy as an integrated whole. Figure 8 depicts three possible switching paths. Path A includes the MicroEngines and the IXP memory. Path B includes the MicroEngines, the IXP memory, and the StrongARM. Path C includes the MicroEngines, the IXP memory, the StrongARM, the Pentium memory, and the Pentium. The shaded boxes correspond to the three processors, while the non-shaded boxes represent memory that implements packet queues and buffers. We know from the previous section that path A can forward packets at a maximum rate of 3.47 Mpps, path B at 526 Kpps, and path C at 534 Kpps, but there are three caveats that affect our design.

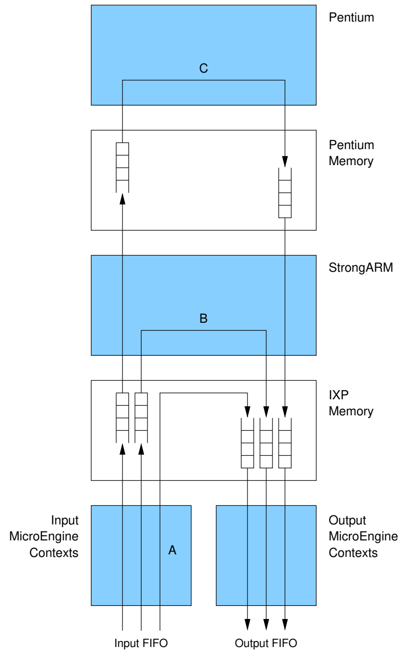

Figure 8: Three switching paths through the Pentium/IXP1200 processor hierarchy.

First, we cannot simultaneously support paths B and C at their maximum rates since the StrongARM is involved in both. Our design gives priority to packets destined for the Pentium, with the StrongARM primarily serving as a bridge between the MicroEngines and the Pentium. We limit the forwarders that run locally (corresponding to path B) to those that fit within the remaining capacity. Similar interference between path A and paths B and/or C is possible, except that in our design the work the MicroEngine has to do to pass a packet up to the StrongARM is the same as the work it has to do to implement path A; no additional cycles are required.

Second, more complicated forwarders require more cycles-per-packet (cpp), possibly reducing the maximum forwarding rate. In the case of the Pentium, we have 500 cpp available at the maximum 534 Kpps rate; more expensive forwarders will obviously lessen the sustainable forwarding rate. In the case of the MicroEngines, all of the available capacity is needed to achieve the 3.47 Mpps forwarding rate. If this matches the line speed, then only the minimal forwarder can run here. However, if we assume a lesser line speed, then there may be excess capacity that can be used to run additional forwarders. We return to this issue in Section 4.2.

Deciding what forwarders to run on the StrongARM is complicated by the fact that the StrongARM must support the Pentium (as described above) and because it shares SRAM and DRAM bandwidth with the MicroEngines. This means an arbitrary forwarder running on the StrongARM has the potential to interfere with the MicroEngine's ability to forward packets at line speed. As a consequence, the StrongARM must run within the same resource budget as the MicroEngines. It is for this reason that we elect to not run a general-purpose OS like Linux on this processor. Instead, the StrongARM runs a minimal OS that does two things: (1) acts as a bridge that forwards packets to the Pentium, and (2) supports a small collection of local forwarders.

Third, even though we know the maximum rates that can be supported by the Pentium and StrongARM, in the worst case all arriving packets require more processing than the MicroEngines can provide, and so have to be passed up the processor hierarchy. This means the higher levels of the processor hierarchy must differentiate among packets based on classification done at the MicroEngine level and then schedule their available capacity in some meaningful way.

Regarding classification, this means that (although not shown in Figure 8) multiple queues feed the higher levels of the processor hierarchy allowing, for example, the router to isolate OSPF updates from ping packets. Regarding scheduling, we run a proportional share scheduler on the Pentium, where deciding what share to allocate to each flow is a policy issue. For example, we allocate sufficient cycles to the OSPF control protocol to ensure that it is able to update the routing table at an acceptable rate, and we allow forwarders that implement per-flow services to reserve both a packet rate and a cycle rate [19]. We eventually plan to run a proportional share scheduler on the StrongARM, since in general it might also run arbitrary forwarders, but we currently implement a simple priority scheme that gives packets being passed up to the Pentium precedence over packets that are to be processed locally.

4.2 Virtual Router Processor

Our approach to installing useful packet processing at the MicroEngine level of the processor hierarchy is to statically allocate the MicroEngines to two tasks: (1) a *router infrastructure* (RI) that is able to forward minimum-sized packets at line speed, and (2) a *virtual router processor* (VRP) that runs additional code on behalf of each packet. In effect, Section 3 defines the RI, while this section defines the VRP. In terms of the pseudo-code in Figure 5, everything except protocol_processing is part of the RI, and it is useful to think of the protocol_processing step as running on an abstract machine (called the VRP) that supports a fixed number of cycles for each MP.

The next question is how to characterize the capacity of the VRP so we can understand what code it is allowed to run. Figure 9 illustrates the effects of adding instructions to the null VRP of the 3.47 Mpps system. The three lines in the graph represent adding three different basic blocks of "VRP code". Blocks are either sets of 10 register-based instructions, a single 4-byte SRAM access, or a combination block with both 10 register instructions and the 4-byte SRAM operation. Effectively, the graph shows the relationship between supportable line speed and VRP budget. By fixing either of these variables, the graph can be used to determine the availability of the other. For example, at an aggregate forwarding rate of 1 Mpps, the VRP has a budget of 32 blocks, each consisting of 10 register operations and a 4-byte read from SRAM.

However, Figure 9 is based on measurements of traffic without any contention for output queues. Since contention is common in practice, it is important to measure its effects on the VRP. Figure 10

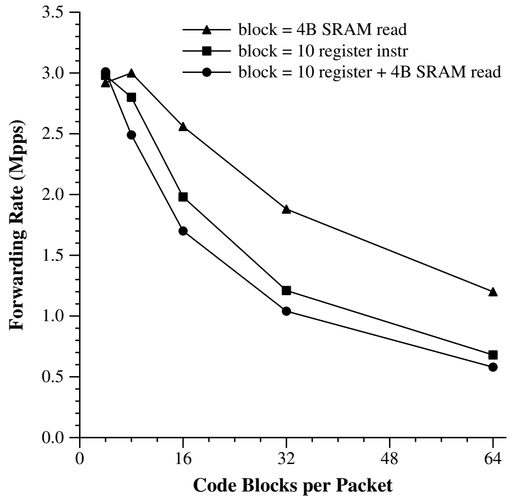

Figure 9: Number of blocks of VRP code that can run at different line speeds.

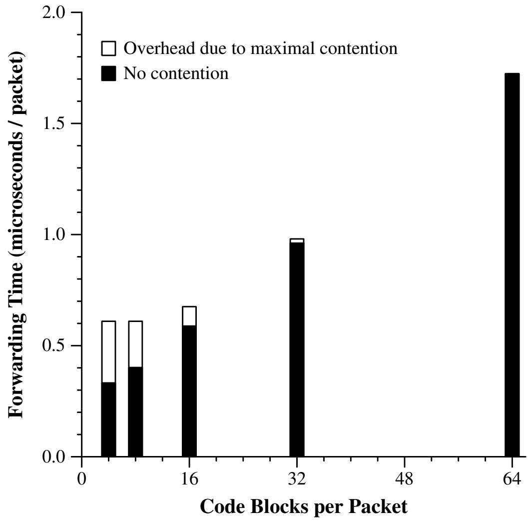

Figure 10: Forwarding time breakdown under maximal output port contention for the case of 10 register instructions and a 4-byte SRAM read per code block. (The "no contention" portion corresponds to the circle points of Figure 9.)

shows the same experiment in the face of maximal contention—all incoming traffic bound for the same protected queue. The graph shows that the time otherwise lost to contention delay can be used for VRP processing. (Note that when we apply 64 blocks of VRP code to each packet, there is no measurable contention overhead.) This is very important because it means that one does not need to worry that large fraction of the system resources will be wasted due to contention delays. In fact, these resources can be reclaimed by increasing the VRP budget to pace input processing to not overrun the anticipated level of contention. This is double a benefit in that the extra VRP cycles can be used to analyze which packets actually deserve to be sent out the contended port.

4.3 Prototype Configuration

We now turn our attention to the actual line speed available on our development board: 8×100 Mbps Ethernet ports. This means that the MicroEngines are required to forward at most 1.128 Mpps, leaving a significant VRP budget. Based on the experiment reported in previous subsection and taking into consideration the state of the MicroEngine context when the packet-specific function is allowed to run, we characterize the VRP on our prototype as follows:

- The packet is fragmented into 64-byte pieces, which become accessible to the VRP in registers one fragment at a time. The first fragment holds both the TCP and IP headers.
- In addition to the 16 registers that hold packet data, the forwarder has access to 8 general purpose 32-bit registers. Values stored here do not last across invocations of the VRP, and so these registers can only be used for temporary state (e.g. intermediate computational results or state loaded from SRAM). An additional register contains the SRAM address of the flow-specific state.
- The forwarder can execute up to 240 cycles worth of instructions.
- The forwarder can perform up to 24 SRAM transfers (reads or writes) of 4 bytes each.
- The forwarder can perform 3 hashes with support of the hardware hashing unit.

Keep in mind that this budget is available for each 64-byte MP processed by the MicroEngine. In addition, there are 650 instruction slots in the ISTORE that must be allocated to the competing extensions. (The next version of the chip will support 1024 additional instructions giving the VRP room for 1674 instructions.) Finally, since those packets passed to the StrongARM will not have yet consumed VRP resources on the MicroEngines—in particular, the available memory references—all this capacity is also available on the StrongARM.

An important consequence of this analysis is that there is sufficient SRAM capacity to load and store up to 96 bytes of state that persists across packets and packet flows. This is critical because the parallelism of multiple MicroEngine contexts working on a single input port may prevent a specific MicroEngine from processing all packets on a particular flow.

Note that this evaluation was done in the context of the worst-case load we can put on the router—forwarding minimum-sized packets arriving at line speed. We do not currently take advantage of cycles that are available when we handle MPs other than the first in the packet. One could imagine using the cycles available before the input worker has to return to the top of the loop to work on some "background" process, but we have not yet attempted to exploit this capacity.

4.4 Extension Requirements

A number of router services have been proposed over the past few years: performance monitoring, intrusion detection, application-level proxies, application-dependent packet dropping, packet tagging, denial-of-service detection and other assorted filters and firewalls. A characteristic of many of these services is that they have separable control and data components. A pure PC-based router is not able to take advantage of this separation since the entire service must be implemented on the main processor(s). A conventional hardware-intensive router with distinct data and control planes is not able to take advantage of this separation because it's data plane is not programmable. Our architecture, however, can exploit this separation. Specifically, these services may be implemented by a pair of forwarders—a data forwarder running on the

IXP1200 that processes every packet and a control forwarder on the Pentium that initializes and manages the data forwarder.

Performance monitoring is a typical example [20]. The data forwarder increments one or more counters based on some property of the packet (e.g., the input or output port, the source or destination address, the packet's protocol number, the TCP ACK or SYN flag). The control forwarder periodically aggregates these counters and sends summaries to a global coordinator. Based on high-level analysis, it is possible that the control forwarder then elects to install new counters in the data forwarder. Intrusion detection often works in a similar way: the data forwarder records events; the control forwarder analyzes them and in turn installs filters in the data forwarder.

TCP splicing, a technique for optimizing proxy performance, also illustrates the separation of the control and data components [21]. A proxy running on the router that connects a private corporate network to the public Internet first inspects the data received on a TCP connection to the external network—e.g., authenticates the entity making the connection request and performs access control on the requested resource—but assuming the proxy is satisfied with what it sees, it then simply forwards data between the external and internal connections. The optimization is to splice the two TCP connections together once the authentication phase is complete, thereby eliminating the need for two TCP state machines and the proxy. However, splicing requires modifying fields in the TCP header of every packet being forwarded between the two connections. In this case, the full TCPs and proxy run in a control forwarder (they operate on only a few packets per connection), while the splicing code that patches the TCP headers runs in a data forwarder (it operates on all subsequent packets).

It is also possible to install a smart packet dropping service that exploits knowledge of a particular kind of application data. For example, wavelet encoded video divides the video stream into multiple layers [3]. Depending on the level of congestion experienced at a router, packets carrying low-frequency layers are forwarded and packets carrying high-frequency layers are dropped. In this case, the data forwarder records the number of packets successfully forwarded for this flow, while the control forwarder uses this information to determine the available forwarding rate, and from this, the cutoff layer for forwarding. The control forwarder then informs the data forwarder of this cutoff, and the data forwarder decides what packets to drop based on this value.

Forwarder	SRAM Read/Write (bytes)	Register Operations (instructions)	Registers Needed
TCP Splicer	24	45	7
Wavelet Dropper	8	28	4
ACK Monitor	12	15	4
SYN Monitor	4	5	0
Port Filter	20	26	2
IP−−	24	32	2

Table 5: Cycle, Memory and Register Requirements of Example Data Forwarders

We have implemented five example data forwarders. Table 5 gives the memory and cycle requirements for each. The first two correspond to the forwarders just described. The third (ACK Monitor) watches a TCP connection for repeat ACKs in an effort to determine the connection's behavior [17]. The fourth (SYN Monitor) counts the rate of SYN packets in an effort to detect a SYN attack. Port Filter is a simple filter that drops packets addressed to a set of up to five port ranges. The last is minimal IP processing, which consists of decrementing the TTL, recomputing the checksum and replacing the Ethernet header. (Note that the IP header also needs to be validated—the checksum verified and the version and length fields checked—but this is done as part of the classifier rather than the forwarder.) Although far from a comprehensive study, this simple list demonstrates that it is easy to write data forwarders that can live within the VRP budget.

In contrast, we have measured more complicated forwarders such as TCP proxies and full IP to require at least 800 and 660 cycles per packet, respectively. Also, the prefix matching algorithm we use [22] requires on average 236 cycles per packet. These forwarders clearly need to run on the StrongARM or Pentium.

4.5 Interface and Implementation

Taking these examples into consideration, we have defined the following interface for a control forwarder running on the Pentium to install and share state with a data forwarder running on the IXP. The IXP exports this interface to the Pentium, and the operations are implemented on the StrongARM. The StrongARM interacts with the MicroEngines to implement the operations, but this interaction is hidden from the Pentium. The interface consists of four operations:

```
fid = install(key, fwdr, size, where)
remove(fid)
data = getdata(fid)
setdata(fid, data)
```

The first operation installs forwarder fwdr on behalf of all packets that match the specified key, with size bytes of associated flow state. The where argument indicates the processor on which the forwarder is to run.

The key is a $\langle$*src_addr, src_port, dst_addr, dst_port*$\rangle$ 4-tuple. As a special case, key can have the value ALL, indicating that the corresponding forwarder is to be applied to all incoming packets. The 4-tuple is used to bind a forwarder that implements something like TCP splicing or wavelet video dropping to a specific end-to-end flow, while ALL is generally used by forwarders that count various packet events or filter certain addresses or ports. We call the former a *per-flow forwarder* and the latter a *general forwarder*. Note that this discussion assumes a fixed, IP-centric classifier. In general, the classifier could itself be replaced with one that also understands, say, MPLS labels. The current implementation does not support incremental changes to the classification code; this would require re-loading the entire MicroEngine ISTORE.

The where argument takes one of three values, which selects how the fwdr argument is to be interpreted and referenced.

- ME: the fwdr argument is an executable fragment of MicroEngine code; it is loaded into the ISTORE of the input contexts and is subsequently referenced by its offset in the I-STORE.
- SA: the fwdr argument is an executable StrongARM function; it is loaded into the DRAM and subsequently referenced by an index into a jump table.[2]
- PE: the fwdr argument is an index into a jump table that is available on the Pentium; subsequently passing that address to the Pentium causes the Pentium to jump to that function.

[2]The current implementation does not allow new forwarders to be dynamically added to the StrongARM. Instead, the StrongARM boots with a fixed set of forwarders, and the install function simply binds one of them to a flow.

The StrongARM maintains a table of all the forwarders it has installed; the return value fid is an index into this table. For each forwarder, the table records the SRAM address that holds the flow state, the function address, and the key. The getdata and setdata operations use the fid to access the flow state. It is by manipulating this shared state that a control forwarder is able to communicate with its partner data forwarder. The remove operation uses the fid to locate information about an installed forwarder, allowing it to remove the key from the hash table and free the memory (DRAM or ISTORE) holding the function.

The install operation is implemented on the StrongARM as follows. First, based upon the where argument, it copies the code block passed in the fwdr argument into DRAM (where=SA) or the ISTORE of all the input contexts (where=ME). Next, the StrongARM allocates size of SRAM memory to hold the flow state, and initializes it to zero. Finally, it updates the hash table used by the packet classifier to map the key to the forwarder and the address of the flow's state.

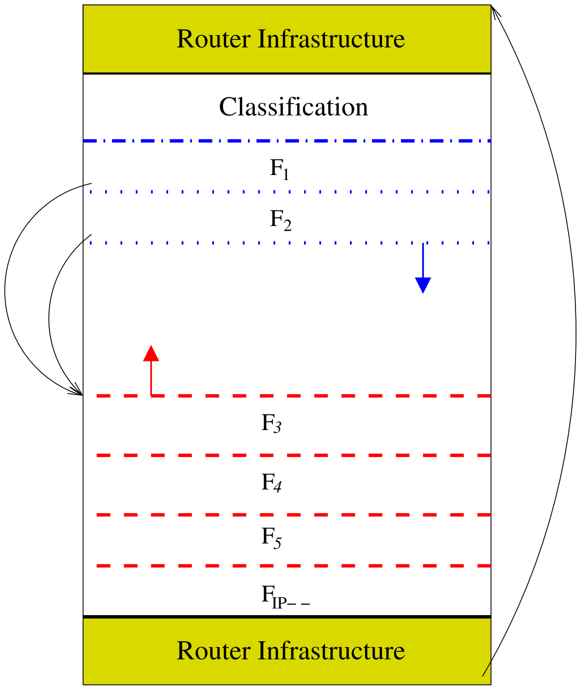

Figure 11: Layout of extensions in MicroEngine ISTORE for input contexts.

Focusing on extensions that run on the MicroEngines, Figure 11 shows the layout of the ISTORE on each context running the input loop. The shaded areas at the top and bottom of the figure correspond to RI component of the loop, while the clear area in the center represents the protocol_processing step in Figure 5. The latter area of the ISTORE is further divided into three segments: the code block that implement packet classification, zero or more code blocks that implement per-flow forwarders, and one or more code blocks that implement general forwarders. The last general forwarder (denoted F_{IP--}) is always present, and implements minimal IP processing.

To modify a MicroEngine ISTORE once the router is in operation, the StrongARM first disables the MicroEngine and writes its ISTORE with instruction level granularity; it then re-enables the MicroEngine. Modifying the MicroEngine program takes two memory accesses for each instruction, meaning that adding a 10-instruction forwarder to the ISTORE takes 800 cycles, while re-writing the entire ISTORE takes over 80,000 cycles. Because we want to be able to compile forwarders separately and install them incrementally—i.e., without having to re-write the whole ISTORE—we are not able to hard-code jump addresses into the forwarders. Instead, general forwarders are stored in reverse order from the end of the ISTORE, thereby allowing control to just fall from one to the next. Also, the last instruction of each per-flow forwarder is an indirect jump to an address maintained in a MicroEngine register.

Revisiting what happens for each packet that arrives, the classification code in the protocol_processing step of the input context first validates the headers, then hashes the IP and TCP headers separately. The two hashed values are combined to index into a table that contains metadata for the flow: the key, where the forwarder is to run, a reference to the forwarder (this is the address of the forwarder's code in the ISTORE if the forwarder is to run on the MicroEngine), and the addresses of the forwarder's state in SRAM. This classification process requires 56 instructions and accesses 20 bytes of SRAM; this code is counted against the VRP budget. Given this metadata, and assuming the forwarder is available on the MicroEngine, the context jumps to the specified offset in its ISTORE. If the forwarder runs on the StrongARM or the Pentium, the context instead enqueues the packet on the corresponding queue and signals the StrongARM that a new packet has arrived. The address of the flow metadata is also passed to the StrongARM, so that it does not have to re-classify the packet.

The StrongARM manages two sets of queues; the MicroEngines insert packets into these queues and the StrongARM services them. The first set contains packets that are to be processed locally. The second set contains queues for each flow that is to be passed up to the Pentium. For packets bound for the StrongARM, it jumps to the corresponding local function. For packets on the Pentium bound queue, it initiates the process of copying the packet to the Pentium, as outlined in Section 3.7. As part of the packet the StrongARM passes the metadata along to the Pentium, so it knows what function to apply to the packet.

4.6 Admission Control

Our design depends on an admission control mechanism that decides what forwarders to install. We have not yet implemented this mechanism, but envision it running on the Pentium.

For any forwarder to be installed on the MicroEngines, the admission control mechanism must inspect the code to determine the number of cycles and memory accesses it requires. (The number of cycles required is slightly larger than the instruction counts reported in Table 5 since branch delays must be taken into consideration.) If the VRP budget allows, and there is room in the ISTORE, the forwarder is approved and the install operation called. Verifying that the forwarder lives within the available VRP budget is trivial since there is no reason for the forwarder to contain a loop, and hence, a backwards jump. This is because the MicroEngines operate on 64-byte chunks, any processing loop that a forwarder might want to employ is already effectively unrolled. Note that general forwarders that operate on all packets run in serial (that is, the sum of their cycle/memory requirements is bounded by the VRP since they are all applied to each packet), while per-flow forwarders logically run in parallel (that is, only the most expensive per-flow forwarder counts against the VRP budget since only the forwarder that matches the given packet is run). We limit the number of per-flow forwarders that can be applied to any packet to one.

For any forwarder to be installed on the StrongARM, the admission control mechanism must both verify that the code does not violate the VRP budget, and enough of the StrongARM capacity is set aside to meet its obligations to move data to/from the Pentium. Based on our experience to date, we do not believe that trying to squeeze additional forwarders onto the StrongARM is justified. Therefore, our current implementation allocates all of the capacity on the StrongARM to passing messages up to the Pentium.

Although it is beyond the scope of this paper, the admission control mechanism must also decided how many forwarders to allow on the Pentium. For each such forwarder, the requester specifies the expected packet rate and the expected number of cycles expended on each packet. From these two values, the mechanism determines the forwarder's total cycle rate. The forwarder can be admitted only if the processor has sufficient cycles-per-second are available and the total packet rate remains below the maximum that the Pentium can sustain. Admission control to the Pentium, as well as the strategy for scheduling the Pentium's cycles, are discussed elsewhere [19].

4.7 Robustness Experiments

To validate the performance of the complete system, we configured the MicroEngines to run a synthetic suite of forwarders based on the examples given in Section 4.4. The suite utilizes the full VRP budget. We then programmed the VRP to forward a variable number of packets to the Pentium. We found that the system was able to forward up to 310 Kpps (out of the 1.128 Mpps offered load) through the Pentium without dropping any packets at any level of the processor hierarchy. Each of the 310 Kpps routed through the Pentium, in turn, receives 1510 cycles of service.

In a second experiment, we ran the base infrastructure described in Section 3 without any VRP, and treated an increasing percentage of the packets as exceptional, thereby simulating a flood of control packets. These exceptional packets had no effect on the router's ability to forward regular packets, and in fact, up to the point that a processor higher in the hierarchy (e.g., the StrongARM) was unable to service the stream of exceptional packets, the router was able to sustain the full rate of 3.47 Mpps. This is because the MicroEngines budget enough resources to classify and enqueue every packet arriving at line speeds, and once enqueued for a particular forwarder, a given flow receives whatever level of service the scheduling policy dictates.

5. RELATED WORK

Programmable network cards have been used for a number of purposes over the years, including to provide access to high-speed links [6, 24, 26], improve handling of multimedia streams [7], and implement distributed shared memory [1]. All of these prior efforts have been limited to end hosts, and the NICs support a single network port. The IXP1200 is also unique in the level of parallelism it applies to packet processing.

Several recent projects have also focused on the problem of making it easier to extend router functionality [5, 13, 18], but to-date these have been limited to Pentium-based implementations. The exception is a recent effort at Washington University to study the feasibility of implementing router extensions in FPGAs [23]. Perhaps the work closest to our own is an ongoing effort to port the Genesis kernel [2, 14] to the IXP1200. Genesis is designed to support virtual networks by dynamically loading routelets (similar to our forwarders) onto the IXP1200. The main difference is that our approach runs all forwarders for a given packet in a single thread, which is critical to our ability to isolate performance under varying loads.

6. CONCLUSIONS

This paper addresses the resource allocation and scheduling problems of implementing an extensible router on a three-level processor hierarchy. We demonstrate our design on prototype hardware consisting of a Pentium augmented with an IXP network processor. Our design results in two specific contributions. First, we describe how to program the processor hierarchy with a fixed forwarding infrastructure that fully exploits the parallelism available on the IXP1200 MicroEngines. Our approach is able to achieve a forwarding rate of 3.47 Mpps. Second, we demonstrate how new functionality can be injected into all three levels of the processor hierarchy without jeopardizing the router's robustness in the face of different workloads. The key innovation is to statically partition the processing capacity of the MicroEngines into a fixed routing infrastructure and a programmable VRP, and to ensure that any extensions programming into the MicroEngines live within the VRP's budget.

While we have focused on a router configuration that includes a single Pentium/IXP pair, we believe our basic architecture applies equally well to richer configurations. For example, we next plan to construct a router from four Pentium/IXP pairs connected by a Gigabit Ethernet switch. The main difference from the configuration described in this paper is that we will need to budget RI capacity to service packets arriving on the "internal" link (i.e., some fraction of the 1 Gbps Ethernet link connecting the IXP to the switch), leaving fewer cycles for the VRP. In general, as network processors become more prevalent in high-end routers, we expect our techniques to also apply there as well. In the end, we expect the distinction between "hardware-based" and "software-based" routers to become less meaningful.

Acknowlegements

We are indebted to Intel's Dirk Brandewie for debugging our understanding of the IXP1200. We would also like to thank the anonymous reviewers and Hari Balakrishnan, our shepherd, for helping us improve the clarity and focus of the paper. This work supported in part by NSF grant ANI-9906704, DARPA contract F30602–00–2–0561, and Intel Corporation.

7. REFERENCES

[1] M. A. Blumrich, C. Dubnicki, E. W. Felten, K. Li, and M. R. Mesarina. Virtual-Memory-Mapped Network Interfaces. *IEEE Micro*, 15(1):21–28, 1995.

[2] A. T. Campbell, S. Chou, M. E. Kounavis, and V. D. Stachtos. Implementing Routelets: Virtual Router Support for the IXP1200 Network Processor. In *IXA Univeristy Program Workshop*, Portland, Oregon, June 2001.

[3] M. Dasen, G. Fankhauser, and B. Plattner. An Error Tolerant, Scalable Video Stream Encoding and Compression for Mobile Computing. In *Proceedings of ACTS Mobile Summit 96*, pages 762–771, November 1996.

[4] B. Davie and Y. Rekhter. *MPLS: Technology and Applications*. Morgan Kaufmann Publishers, Inc., 2000.

[5] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router Plugins: A Software Architecture for Next Generation Routers. *IEEE/ACM Transactions on Networking*, 8(1):2–15, February 2000.

[6] P. Druschel, L. L. Peterson, and B. S. Davie. Experiences with a High-Speed Network Adaptor: A Software Perspective. In *Proceedings of SIGCOMM '94 Conference*, pages 2–13, October 1994.

[7] M. E. Fiuczynski, R. P. Martin, T. Owa, and B. N. Bershad. On Using Intelligent Network Interface Cards to support Multimedia Applications. In *Proceedings of the 8th International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 95–98, July 1998.

[8] IBM Microelectronics Division. *IBM PowerNP NP4GS3 Network Processor Solutions Product Overview*, April 2001.

[9] IEEE. *Standard 802.3*, October 2000.

[10] Intel Corporation. *IXP1200 Network Processor Datasheet*, September 2000.

[11] Intelligent I/O (I_2O) Special Interest Group. *Intelligent I/O (I_2O) Architecture Specification, Version 2.0*, March 1999.

[12] S. Karlin and L. Peterson. VERA: An Extensible Router Architecture. In *Proceedings of the 4th International Conference on Open Architectures and Network Programming (OPENARCH)*, pages 3–14, April 2001.

[13] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[14] M. E. Kounavis, A. T. Campell, S. Chou, F. Modoux, J. Vicente, and H. Zhuang. The Genesis Kernel: A Programming System for Spawning Network Architectures. *IEEE Journal on Selected Areas in Communications*, 19(3):511–526, March 2001.

[15] H. C. Lauer and R. M. Needham. On the Duality of Operating System Structures. *Operating Systems Review*, 13(2):3–19, April 1979.

[16] D. L. Mills. The Fuzzball. In *Proceedings of the SIGCOMM '88 Symposium*, pages 115–122, August 1988.

[17] V. Paxson. Automated Packet Trace Analysis of TCP Implementations. In *Proceedings of the ACM SIGCOMM '97 Conference*, pages 167–179, September 1997.

[18] P. Pradhan and T.-C. Chiueh. Operating System Support for Programmable Cluster-Based Internet Routers. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS–VII)*, pages 76–81, March 1999.

[19] X. Qie, A. Bavier, L. Peterson, and S. Karlin. Scheduling Computations on a Programmable Router. In *Proceedings of the ACM SIGMETRICS 2001 Conference*, pages 13–24, June 2001.

[20] M. J. Ranum, K. Landfield, M. Stolarchuk, M. Sienkiewicz, A. Lambeth, and E. Wall. Implementing a Generalized Tool For Network Monitoring. In *Proceedings of the Eleventh Systems Administration Conference (LISA '97)*, pages 1–8, San Diego, CA, October 1997.

[21] O. Spatscheck, J. Hansen, J. Hartman, and L. Peterson. Optimizing TCP Forwarder Performance. *IEEE/ACM Transactions on Networking*, 8(2):146–157, April 2000.

[22] V. Srinivasan and G. Varghese. Fast address lookups using controlled prefix expansion. *ACM Transactions on Computer Systems*, 17(1):1–40, February 1999.

[23] D. E. Taylor, J. S. Turner, and J. W. Lockwood. Dynamic Hardware Plugins (DHP): Exploiting Reconfigurable Hardware for High-Performance Programmable Routers. In *Proceedings of the 4th International Conference on Open Architectures and Network Programming (OPENARCH)*, pages 25–34, April 2001.

[24] C. B. S. Traw and J. M. Smith. Hardware/Software Organization of a High-Performance ATM Host Interface. *IEEE Journal on Selected Areas in Communications (Special Issue on High Speed Computer/Network Interfaces)*, 11(2):240–253, 1993.

[25] Vitesse Semiconductor Corporation. *IQ2000 Network Processor Product Brief*, 2000.

[26] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 40–53, December 1995.