Evaluating Network Processors in IP Forwarding

Tammo Spalink, Scott Karlin, Larry Peterson
{tspalink,scott,llp}@cs.princeton.edu

Department of Computer Science
Princeton University

Technical Report TR–626–00
November 15, 2000 (Updated January 31, 2001)

Abstract

This paper evaluates the performance of emerging network processors—in particular, designs that employ multiple hardware contexts to hide memory latency—in constructing IP routers. Such processors are designed to forward minimum-sized IP packets at line speeds, with the advantage (over ASIC-based solutions) of being programmable. However, programming such network processors involves two challenges. The first is how to effectively employ the multiple contexts in a way that fully utilizes the memory bandwidth. The second is how to allow the network processor to be programmed dynamically (so it can support new functionality) without violating the processor's tight timing constraints. This paper addresses both of these challenges on a prototype board that uses the IXP1200 network processor. We demonstrate that it is possible to support 8×100 Mbps ports with enough headroom to access up to 224 bytes of state information for each minimum-sized IP packet.

1 Introduction

There is significant interest in *network processors* that can handle packets arriving at multi-gigabit line speeds, with several commercial products recently coming onto the market [4, 11]. Designed primarily to implement forwarding engines on IP routers, these network processors offer one clear advantage over ASIC-based solutions: they are programmable. The performance requirements, however, are quite severe. For example, a network processor assigned to an OC–192 link has to be able to process up to 25 M minimum-sized packets-per-second (pps). Such rates are outside the reach of existing products, however, with current technology able to support line speeds approaching OC–48 (6.1 Mpps).

Network processors are typically memory-limited. For example, copying an OC–48 bit stream into and out of memory requires 2×2.5 Gbps $= 5$ Gbps of memory bandwidth. Sustaining this rate is not trivial due to memory latency. Network processors commonly employ parallelism to hide this latency. For example, the Intel IXP1200 includes six micro-Engines (μEngines), each supporting four hardware contexts. The IXP1200 automatically switches to a new context when the current context stalls on a memory operation. The Vitesse IQ2000 uses a similar design, employing 16 hardware contexts implemented across four processing elements. This approach is reminiscent of the memory latency-hiding strategy used in the Tera multiprocessor [2].

Programming such a network processor involves two challenges. First, assuming one can identify a single function that the network processor has to support—e.g., forwarding vanilla IP packets—the parallel hardware contexts must be programmed in a way that fully utilizes the available memory bandwidth, and hence sustain the maximum required packet rate.[1] The second challenge is to allow the network processor to be programmed more dynamically. Instead of running a single fixed function, it should be possible to load a new function—or perhaps an extension to the existing function—into the network processor, ideally at runtime. The challenge is to structure the program in such a way that new code fragments can be added without violating the processor's tight timing constraints.

This paper addresses both of these challenges on a prototype board that uses the IXP1200 network pro-

[1]It is a market requirement that each line card in a router be able to forward minimum-sized packets at line speed. One reason is that not doing so makes the router susceptible to denial of service attacks.

cessor. It makes two contributions. First, it describes how to program the IXP1200 architecture with a static IP forwarding function (Section 2) and evaluates the performance of this system (Section 3). Second, it demonstrates how excess capacity can be used to dynamically extend the functionality supported by the IXP1200 without violating its ability to process minimum-sized packets at line speed (Section 4).

2 Architecture

This section describes the prototype router we implemented and evaluated. Our design focuses on the machinery needed to forward vanilla IP packets.

2.1 IXP1200 Hardware

The experiments reported in this paper were performed on the IXP1200 evaluation system illustrated in Figure 1. The board consists of a IXP1200 network processor chip (shaded area), 32 MB of DRAM, 2 MB of SRAM, a proprietary 64-bit, 66 MHz IX bus, 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 [6] bus interface.

The IXP1200 chip itself contains a general-purpose StrongARM processor core and six special-purpose micro-Engines running at 177 MHz. Each of the six μEngines supports four hardware contexts, for a total of 24 contexts. Not shown in the figure is a 4 KB instruction store associated with each μEngine. The StrongARM is responsible for loading these μEngine instruction stores; actual StrongARM instructions are fetched from DRAM. A 4 KB on-chip scratch memory is used for synchronization and control of the μEngines.

The chip also has a pair of FIFOs used to send/receive packets to/from the network ports on 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 $\times$ 88 MHz data path, implying a potential to move packets into and out of DRAM at 5.6 Gbps. In theory, this is sufficient to support the $2 \times (8 \times 100\,\text{Mbps} + 2 \times 1\,\text{Gbps}) = 5.6\,\text{Gbps}$ 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 $\times$ 88 MHz = 2.8 Gbps.

The common unit of data transferred among components is a 64-byte MAC-Packet (MP), with the data path traversed by a typical packet defined as follows. As an incoming IP packet is received, the MAC breaks the packet into separate MPs, tags the MP as being the the first, an intermediate, the last, or the only MP of the packet, and stores the MP in an input FIFO slot. A μEngine context moves the MPs of each packet from the input FIFO to a DRAM buffer. Later, after the packet is ready to be forwarded, a (possibly different) μEngine context reads the MPs of the packet from the DRAM buffer, tags them with the output MAC name, and stores each MP into an output FIFO slot. Finally, the named MAC chip reads the MPs from the output FIFO and transmits them.

The key to performance is the latency involved in each MP transfer. Table 1 gives the measured latency for each of six μEngine transfer instructions related to packet movement. The latency is given in μEngine cycles, which are approximately 5.6 ns. (Recall that the processor runs at 177 MHz.) Also note that each transfer moves at most 32 bytes, meaning that moving a minimum-sized Ethernet packet into DRAM takes two FIFO-to-DRAM transfers, and moving the same packet out of DRAM takes two DRAM-to-FIFO instructions, for a total of four transfers to move a 64-byte MP from the input FIFO to the output FIFO. For completeness, the last four rows of Table 1 give the latency for SRAM and scratch memory.

Thus, at a minimum, the four transfers into and out of DRAM take $2 \times 45 + 2 \times 55 = 200\,\text{cycles} = 1.12\,\mu\text{s}$. This corresponds to a forwarding rate of 893 Kpps for a single context. If we could effectively employ all 24 hardware contexts in this process, we could expect to forward at most 21.4 Mpps, but memory bandwidth eventually becomes the bottleneck, limiting the packet forwarding rate to a best case 5.6 Gbps / 64 bytes-per-packet / 2 = 5.45 Mpps, considering that each packet must (at a minimum) be copied both into and out of DRAM.[2] The remainder of this section discusses some of the challenges and implementation details in

[2]This discussion ignores the limits of the IX bus and specific MAC ports on the evaluation board.

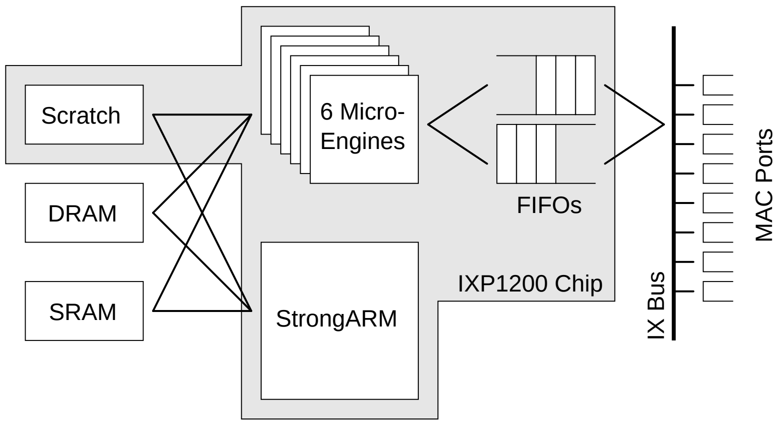

Figure 1: Block Diagram of an IXP1200 Evaluation System

Instruction	Cycles
FIFO-to-DRAM	45
DRAM-to-FIFO	55
FIFO-to-μEngine	30
μEngine-to-FIFO	40
DRAM-to-μEngine	40
μEngine-to-DRAM	50
SRAM-to-μEngine	30
μEngine-to-SRAM	30
Scratch-to-μEngine	25
μEngine-to-Scratch	25

Table 1: Cycle times to transfer 32 bytes of data between components

forwarding IP packets.

2.2 Software

There are two complications to programming the IXP1200 to forward vanilla IP packets. The first challenge is managing two sets of parallel resources: a set of 24 contexts, and a set of 32 FIFO slots (16 slots in the input FIFO and 16 slots in the output FIFO). The second complication is that 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, one context must first read the packet and queue it in DRAM, with a second context later moving the packet from DRAM to an output port.

The design space for programming the IXP1200 is very rich. For example, work can be assigned to a context dynamically, or each context can be statically bound to a particular task. Similarly, FIFO slots can be either dynamically or statically bound to ports. Our approach was to statically allocate all resources.[3] Doing so obviously simplified the implementation, but in retrospect, it was the right thing to do in terms of yielding predictable performance numbers. Predictable performance, in turn, facilitates the extensibility described in Section 4.

Figure 2 illustrates the key components of the implementation; to make the discussion concrete, we focus on the version of the software designed to support just the 8×100 Mbps Ethernet ports. First, the 24 contexts are divided into two groups: 16 service the input FIFO, and 8 service the output FIFO. This 2-to-1 allocation is justified for three reasons: (1) there is an integral number of contexts for each port, (2) there is an integral number of μEngines bound to input processing and output processing, and (3) there is roughly twice as much work to do on input as on output.

Second, we make a static assignment of ports to FIFO slots to μEngine contexts. For both the input and output paths, the 16 FIFO slots are partitioned into 8 slot-pairs. On the output side, each output context is assigned to an output FIFO slot-pair which is assigned to an output port. On the input side, each port is assigned to an input FIFO slot-pair which is assigned to a pair of input contexts. (That is, each input process services a specific input FIFO slot.)

Third, we implemented a shared buffer pool for packet data in the DRAM. In the SRAM we implemented 8 queues (one per port) of to-be-transmitted packet addresses. These addresses point to packets in the DRAM buffer pool. In addition, the SRAM also holds the routing table; each entry in the table con-

[3]As opposed to the Intel reference software for the IXP1200, which uses centralized dynamic schedulers.

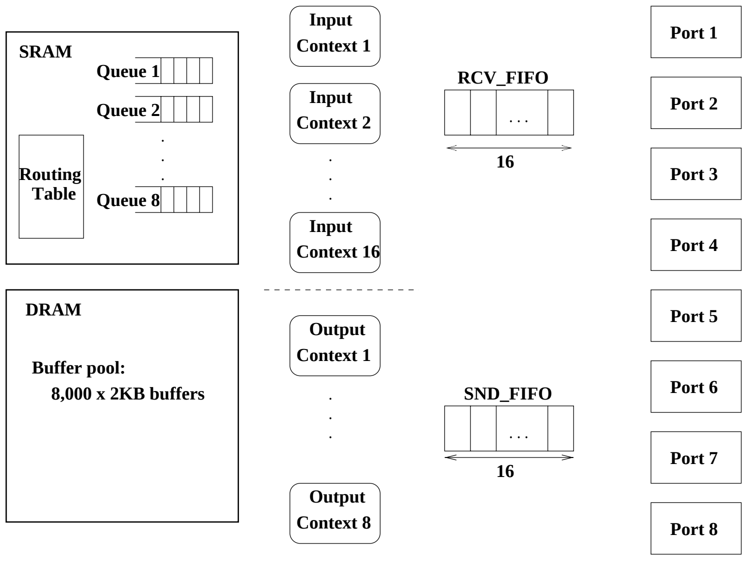

Figure 2: Major components involved in IP forwarding.

tains 16 bytes of information: SrcIPAddr (4 bytes), DstIPAddr (4 bytes), DstEthAddr (6 bytes), Queue (1 byte), and Func (1 byte). The Func field selects the function to be applied to the packet; for now, assume all packets receive the same vanilla IP processing. Finally, each hardware context includes a set of registers, which we statically allocate to hold packet headers (Reg_Hdr) and an entry from the routing table (Reg_Entry).

Figure 3 gives pseudo-code for the loop executed by each context *c* assigned to input processing (top) and output processing (bottom) on behalf of port *p*. Each loop executes once per MP received or sent. In the figure, Reg_Entry.Queue denotes the Queue field of the routing table entry stored in the context registers allocated to routing table entries, Reg_Hdr.DstIPAddr denotes the destination address of the IP header held in registers, and so on. Also, DRAM[*a*] denotes memory address *a* in DRAM.

We start by examining the input process. The first instruction (until_ready_rcv) is the only step that hides a significant amount of detail. In effect, it involves a busy loop that queries a state machine (not shown) to determine when data is available on port *p*, and instructs the state machine to move the next MP that arrives on that port to FIFO slot *c*. (Recall that there is a 1-to-1 mapping between input contexts and slots in the input FIFO.) The other detail we are obscuring is where the input context decides to store the incoming packet. We maintain a simple cyclic buffer in DRAM, with the buffer for a given packet selected as a function of the input port *p*, hence the address *p_buf*.

```
input:
  until_ready_rcv(p, c)
  move IN_FIFO[c] to DRAM[p_buf + offset]
  move DRAM[p_buf] to Reg_Hdr
  h = hash(Reg_Hdr.DstIPAddr)
  move SRAM[h] to Reg_Entry
  Reg_Entry.Func(Reg_Hdr)
  if End_Of_Packet
      enqueue(p_buf, Reg_Entry.Queue)

output:
  until_ready_snd(p, c)
  if Start_Of_Packet
      buf = dequeue(p)
  move DRAM[buf + offset] to OUT_FIFO[c]
  start_transfer(p, c)
```

Figure 3: Pseudo-code running in each context assigned to input and output processing.

The move instructions transfer data between μEngine registers, DRAM, SRAM, and FIFOs, as described earlier in this section. The hash function is implemented in hardware, and so involves no memory accesses. The enqueue operation inserts the address of the buffer holding the received packet into the output port's packet queue.

The "function" Reg_Entry.Func(), which is actually performed in-line, includes all protocol-specific packet header or content modifications. In our experiments this is the vanilla IP forwarding function: decrement the time-to-live (TTL) field, recompute the checksum, set the destination MAC address to the one found in the routing table, set the source MAC address to that of the output port, and write these changed values back to DRAM.

Note that these manipulations need to be performed only once per packet although the function is called once for every MP, and thus many times for large packets. At first it may seem that loading the packet header, performing a forwarding table lookup, and executing this function for each MP is wasteful—when processing MPs that comprise the body of large packets, there is probably no need to have the packet headers or forwarding table entry information on hand. However, resources are allocated statically, so we have enough resources to treat each MP as if it is a minimum-sized packet. Performing these operations even when they are not strictly necessary does not affect other system components. The reason to structure the code the way we do will become apparent in Section 4.

Finally, the check for End_Of_Packet tells the context that the whole packet has now been received, and that the time has come to insert the packet in the transmit queue. Note that the move from FIFO to DRAM happens for all MPs; the conditional statements happen only once per packet. The worst-case scenario, therefore, is when each MP contains a complete (minimum-sized) IP packet.

The output process is relatively straightforward. Basically, the MPs which comprise a packet are moved from DRAM to FIFO. When all of the MPs have been moved, the address of the next outbound packet is removed from the port's queue.

This description has focused on how we programmed the μEngines to handle common-case traffic. Exceptional packets, for example those that incur a miss in the routing table or involve additional processing (e.g., IP options) receive most, but not all, of the same processing. They are placed in DRAM, but responsibility for forwarding them is passed onto the StrongARM processor. We return to the role played by this processor, as well as a Pentium attached to the IXP1200's PCI bus, in Section 4.

3 Performance

This section presents a set of experiments designed to understand the performance limits of the router as it forwards vanilla IP packets. For the experiments reported in this and the next section, we drive the 8×100 Mbps Ethernet ports on the IXP1200 board with eight Kingston KNE100TX PCI Ethernet cards based on the 21143 Tulip chipset. A pair of these cards are plugged into each of four 450 MHz Pentium IIs running an IP packet generator. When configured to generate minimum-sized IP packets, each card produces 141 Kpps, which is a bit less than the theoretical maximum of 148 Kpps.

Given this traffic source, the IXP1200-based router is capable of sustaining 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.

3.1 Maximum Forwarding Rate

Calculating the maximum performance of the router, independent of the MAC ports that populate the board, is the more interesting question. To determine the maximum possible forwarding rate, we modified the input process to move only a single packet from a port to each FIFO slot. Future iterations of the input process see this same packet without any delay or port interaction. This lets us measure the maximum system performance from FIFO to FIFO, effectively emulating infinitely fast network ports. An alternative would have been to use the 2×1 Gbps ports, but we were unable to create stable packet sources for this configuration; we also do not believe that these ports would have saturated the IXP1200, so we would still have to perform the same experiment.

The first step is to measure how many packets a single context can process using either the input or output loops. Instrumenting the code to use the IXP1200's 64-bit cycle counter, we measured these rates to be 375 Kpps and 596 Kpps, respectively. Of course we cannot simply multiply these rates by the number of available contexts, since the contexts will contend for DRAM.

To determine the affects of parallel contexts on DRAM, we increased the number of contexts running the input loop to 4, 8, and 16, and similarly, we measured the impact of 4 and 8 contexts running the output loop. These tests were run independently, so the input and output loops do not affect each other. The results are shown in Figure 4, where we see the rate for the input loop dropping from 297 Kpps per context (4 contexts) to 237 Kpps per context (16 contexts), and the rate for the output loop dropping from 464 Kpps (4 contexts) to 427 Kpps (8 contexts).

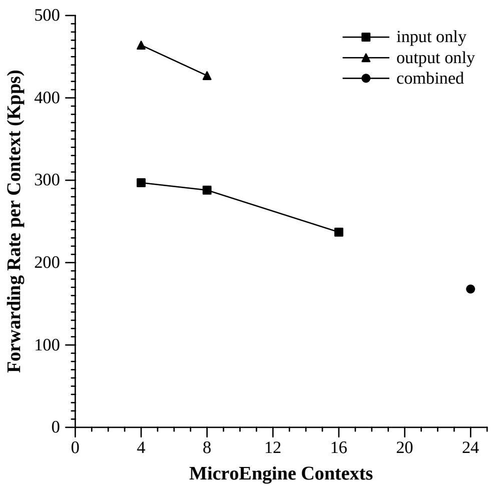

Figure 4: Per-context forwarding rates as the number of parallel contexts increases.

When we combined 16 input contexts with 8 output contexts, each input context was able to forward 168 Kpps (shown by the single dot in Figure 4). The output contexts were paced by the input contexts, and so forwarded 336 Kpps each. Thus, the IXP1200 is able to forward minimum-sized IP packets at a rate of 16×168 Kpps $= 2.69$ Mpps. This is roughly half of the 5.45 Mpps predicted by the 5.6 Gbps DRAM bandwidth, but is easily explained by the fact that every IP packet header needs to be copied from DRAM into the μEngine registers and then back to DRAM.

This data highlights the negative effects of memory contention. If only input processing is performed, (using 16 contexts) a rate of 3.79 Mpps is achived. Only output (8 contexts) results in 3.41 Mpps. The actual forwarding rate of 2.69 Mpps reflects running both together. The input and output processing occur on different μEngines. They are independent from one another except for sharing memory. Without memory contention (e.g. if the system had sufficiently fast and wide DRAM) performance would be at least limited by processor cycles at 3.41 Mpps, a 26% improvement.

3.2 Code Inspection

To better understand this 2.69 Mpps limit, we examined the code run by each of the contexts. Figure 5 depicts the same input/output loops presented in Figure 3, but this time annotated with instruction counts. In this figure, I denotes the count of actual μEngine instructions executed at each step, while D, S, and L denote the number of DRAM, SRAM, and scratch (local) memory accesses, respectively.

Code	I	D	S	L
until_ready_rcv(*p, c*)	44			4
move IN_FIFO[*c*] to				
DRAM[*p_buf* + *offset*]	24	2		3
move DRAM[*p_buf*] to Reg_Hdr	7	1		
h = hash(Reg_Hdr.DstIPAddr)	2			1
move SRAM[*h*] to Reg_Entry	4		1	
Reg_Entry.Func(Reg_Hdr)	32	2		
if End_Of_Packet	2			
enqueue(*p_buf*, Reg_Entry.Queue)	23		1	4
until_ready_snd(*p, c*)	7			1
if Start_Of_Packet	2			
buf = dequeue(*p*)	38		1	2
move DRAM[*buf* + *offset*] to				
OUT_FIFO[*c*]	20	2		
start_transfer(*p, c*)	10			1
Total	215	7	3	16

Figure 5: The per-packet pseudo-code annotated with the number of actual instructions (I), DRAM accesses (D), SRAM accesses (S), and scratch (local) memory accesses (L).

The main thing to note is that 7 DRAM accesses are required for each 64-byte MP, not the best-case 4 accesses suggested in Section 2.1. This is because of the additional processing done by Reg_Entry.Func(). For each packet, the header has to be brought into μEngine registers, modified, and then copied back out to DRAM. The asymmetry—one access to copy into registers and two accesses to copy out—is caused by the fact that the Ethernet + IP header at the front of each packet is 34 bytes long and only 32 bytes worth of registers are available for DRAM transfers. We do not need to copy the Ethernet header into the registers (hence one DRAM access), while we do need to copy the Ethernet header out (hence two DRAM accesses) since the Ethernet addresses in the packet need to be modified.

We independently measured the achievable DRAM bandwidth to be 5.01 Gbps for 64-byte transfers, and 4.16 Gbps for 32-byte transfers. We transfer 3×64 bytes $= 1536$ bits at 5.01 Gbps, and 1×32 bytes $=$ 256 bits at 4.16 Gbps, for a total expected forwarding rate of 2.71 Mpps. This is almost exactly the same as the measured 2.69 Mpps forwarding rate, convincing us that DRAM is the bottleneck.

The 16 scratch memory accesses may come as somewhat of a surprise. These implement either mutex operations used to lock shared queues, or operations to get and set configuration registers that control the state machine that moves data between the ports and the FIFOs. Our implementation, in fact, is very close to saturating the scratch memory. However, we could easily move the mutex operations to SRAM, where we have plenty of head room.

4 Computing on Packets

The previous section explored how all of the IXP1200's resources could be brought to bear on forwarding vanilla IP packets. An equally viable alternative is to assume packets arrive at less than the maximum sustainable line speed, where excess compute capacity is applied to running more sophisticated forwarding functions. Considering this scenario makes sense for two reasons. First, using a programmable line card to implement vanilla IP forwarding is not a cost-effective solution as there are plenty of ASIC-based solutions available. Second, there are a wealth of interesting computations that routers are being asked to perform on packets. For example, routers are programmed to filter packets, translate addresses, make level-*n* routing decisions, broker quality of service (QoS) reservations, thin data streams, run proxies, support computationally-weak home electronic devices, serve as the front-end to scalable clusters, and support application-specific virtual networks [1, 3, 5, 7, 8, 10, 12].

The question, then, is how much computation can we expect to perform on each packet, given some fixed packet rate. This section considers this question for the 8×100 Mbps Ethernet ports on the evaluation board.

4.1 Processor Hierarchy

In revisiting the block diagram for the IXP1200 presented in Figure 1, we observe that the architecture can be viewed as a three-level processor hierarchy. Packets follow switching paths that traverse different levels of the hierarchy. As illustrated in Figure 6, one switching path traverses just the μEngine, a second traverses the StrongARM processor, and a third traverses the Pentium processor. 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 consume resources at lower levels of the hierarchy to access them.

It is also the case that the lowest level of the hierarchy must be able to process packets at line speed; only "excess" capacity can be used to do additional processing. In other words, the packet rate that must be sustained is known in advance, so we can calculate a fixed cycle budget for each packet. In contrast, the higher levels of the hierarchy see a packet arrival rate that is a function of the workload; e.g., a router can be engineered to process IP options in at most 1% of the packets it forwards. It is therefore possible to trade the number of cycles available for each packet against the rate at which packets are expected to arrive.

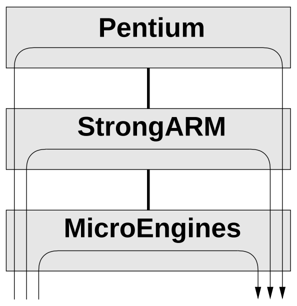

Figure 6: Three switching paths through the IXP1200 processor hierarchy.

The rest of this section considers each of the three processors in Figure 6 in more detail, starting at the bottom of the hierarchy.

4.2 MicroEngines

Because the μEngines are committed to running at line speeds, they are best viewed as being divided into two components: (1) a *router infrastructure* (RI) that is able to forward minimum-sized vanilla packets at line speed, and (2) a *virtual router processor* (VRP) that runs additional code on behalf of each packet. In effect, Section 2 defines the RI, while this section defines the VRP. In terms of the pseudo-code in Figure 3, everything except Reg_Entry.Func(Reg_Hdr) is part of the RI.

The first question is how to characterize the capability of the VRP, so we can understand what code it is allowed to run. Starting with just the RI, we calculated what fraction of the available processor cycles and memory accesses are *not* consumed forwarding minimum-sized packets arriving on the eight 100 Mbit Ethernet ports. We then added the extra instructions—no-ops, SRAM reads/writes, and DRAM reads/writes—needed to consume this extra capacity to the input loop, and verified that we could still forward packets at line speed. Finally, we experimented with various permutations of these instructions, that is, different mixes of register and memory operations. In the end, we opted for a conservative set of extra instructions that consumed roughly 65% of the head room we predicted.

Although there is some head room available in the output context, we elected not to exploit it, so as to concentrate the additional processing steps to a single control point.

Based on these experiments, and taking into consideration the state of the μEngine context when the packet-specific function is allowed to run, we characterize the VRP as follows:

- The first 32 bytes of the IP header are available in registers.
- The function has access to 8 general purpose 32-bit registers. Values stored here do not last across invocations of the VRP, thus should be used for temporary state (e.g. intermediate computational results, packet payload loaded from DRAM, or state loaded from SRAM).
- The function can execute up to 400 cycles worth of instructions.
- The function can perform up to 4 DRAM transfers (reads or writes) of 32 bytes each (128 bytes), with each read costing 40 cycles and each write costing 50 cycles against the 400 cycle total.
- The function can perform up to 7 SRAM transfers (reads or writes) of 32 bytes each (224 bytes), with each read or write costing 30 cycles against the 400 cycle total. Unlike DRAM, the cost of SRAM access scales with transfer size, allowing a greater number of smaller (4, 8, 16 byte) accesses if desired.
- The function can perform 3 hardware hashes.

Keep in mind that this budget is available for each 64-byte MP processed by the μEngine.

We conclude that the DRAM budget is actually sufficient to re-write the entire packet since we have 4 DRAM transfers available to process each 64-byte MP: two to read the MP into the μEngine and two to write it back to DRAM. Furthermore, there are sufficient SRAM accesses to load and store up to 224 bytes of state that persists across packets and packet flows. This is important because the parallelism of multiple μEngine contexts working together on a single input port may prevent a specific μEngine from processing all packets on a specific flow.

Beyond understanding the resource budget, there are several additional issues that need attention. First, we should revisit the issue of exactly what the RI provides at the expense of the VRP budget. For example, we could maintain 32 bytes of state for each flow rather than 16 bytes for each route. In this way the RI would already be equipped to differentiate packets on a flow basis rather than a route basis.

Second, we need a mechanism to extend the μEngine program to include new per-flow functions. This turns out to be easy. Each μEngine has its own 1024-instruction control store (ISTORE) that can be programmed independently of the others. The store can be modified at almost any time. At boot time, before the μEngines are enabled, the StrongARM initialized this with 350 instructions of router infrastructure code. This leaves a 650-instruction VRP store, which can also be initialized at boot time by the StrongARM, or modified later at runtime. To modify a μEngine control store once the router is in operation, the specific μEngine is disabled by the StrongARM. Once disabled, its control store can be read and written with instruction level granularity. It takes two scratch memory accesses to modify one instruction in the ISTORE. Finally, the μEngine is re-enabled with the new VRP functionality. Modifying the store takes two memory accesses for each instruction, meaning that adding a 10-instruction function to the store takes 800 cycles (4.48μs).

Third, we need to allocate the 650 instruction slots in the ISTORE among competing extensions. For example, we may want to allow (1) many small functions associated with equally many fine-grain packet flows, (2) many small functions that affect all packets, (3) a single extension that uses most of the store but benefits only a limited number of flows, and so on. In general, we need a mechanism (and policy) for assigning “pages” of the ISTORE to different VRP programs.

Finally, we need to verify that a given extension adheres to the VRP’s resource budget. This is easy if we disallow loops, which is not unreasonable since all operations are implicitly on 64-byte chunks of each packet. That is, the loop has already been unrolled for us.

4.3 StrongARM

The StrongARM is in a unique position. On the one hand, it is able to directly access DRAM, so the packet is available for it to compute on with minimal additional overhead. The only latency is the cost of a μEngine signalling the StrongARM to inform it that a packet is available. This can be implemented in two different ways. One involves having a μEngine and the StrongARM communicate via a shared memory location. This can be done in two memory accesses—one in each direction—meaning that control over a packet can be passed to the StrongARM and back in 60 cycles if we use SRAM. Of course, we cannot afford to have the StrongARM spinning on memory shared with the μEngines, but the StrongARM could poll this location at some coarser interval. The second involves a μEngine delivering an interrupt to the StrongARM when a packet is available for processing, and the StrongARM writing to a memory location when the packet is ready to be transmitted.

On the other hand, because the StrongARM shares memory with the μEngines, any memory accesses it performs steals accesses from the μEngines. Said another way, the StrongARM must be viewed as a 7th μEngine competing for a limited number of memory accesses. If the StrongARM were to limit itself to only register operations, it could perform approximately 100 such instructions for every one of the 1.128 Mpps arriving on the eight 100 Mbps ports. This number is calculated based on the inter-packet arrival time of 8.86μs (based on the arrival) minus the signalling overhead of 3.36μs = 5.5μs = 98 cycles. However, the StrongARM must limit itself to the same memory budget as the μEngines: 4 DRAM accesses and 7 SRAM accesses per MP.

Considering that the μEngines are equally capable of consuming the excess memory capacity, we envision the StrongARM playing three limited roles. First, it manages the μEngines. This means updating the routing table and loading new code into the μEngines. We expect these to be relatively rare events, so it seems possible to account for them by simply being conservative in how close we come to fully utilizing SRAM—even ten-thousand route updates per-second consume less than 1% of the SRAM bandwidth.

Second, the StrongARM processes a limited number of exceptional packets, but ones for which the forwarding function is known *a priori*. For example, the StrongARM has sufficient capacity to process packets containing IP options. However, the StrongARM must be scheduled to perform this operation for a limited workload (e.g., 1% of the maximum arrival rate) so as to not render the μEngines vulnerable to denial of service attacks.

Third, the StrongARM serves as a go-between for the Pentium and the μEngines. This involves fetching each MP-sized chunk of the packet from DRAM and writing it to the PCI bus (two DRAM accesses) and later reading each MP-sized chunk from the PCI bus and storing it to DRAM (two DRAM accesses). If we were to pass every packet to the Pentium then these four DRAM transfers would consume the entire DRAM budget, but as we will see in the next subsection (not surprisingly) the PCI bus cannot transfer data at this rate. Whatever percentage of packets require processing on the Pentium, the StrongARM will have consumed the DRAM budget for these packets. This is not a problem because it makes no sense for the μEngines to make additional DRAM accesses for each packet, only to pass the packet off to the Pentium.

4.4 Pentium

At the next level of the processor hierarchy, a full Pentium processor is available, connected to the rest of the architecture by a PCI bus. There is clearly a non-trivial overhead involved in getting packets into the Pentium, but significant compute capacity is available once they are there. Also, like the StrongARM but unlike the μEngines, we have to make a judgment call about the expected workload; i.e., how many packets of various types we expect to see. The following discussion is limited to an evaluation of overhead and the remaining capacity.

4.4.1 Experimental Setup

The system we are using is configured to treat the IXP1200 evaluation board as the motherboard, making it impossible to also plug a Pentium motherboard in the PCI bus. This means we cannot directly measure the StrongARM-to-Pentium performance across the PCI bus.[4] Instead, we measure the Pentium/PCI performance using another programmable line card: the RAMiX PMC694 [9]. The RAMiX card interacts with the PCI bus in a way that is functionally identical to the StrongARM on the IXP1200 card, and so it gives us an accurate account of both how many packets per second we can push across the PCI bus, and how much impact this has on the Pentium. This setup cannot tell us what impact these transfers have on the StrongARM, but to a first approximation we already accounted for this overhead in the previous subsection.

[4]This is a limitation of the current system, not the IXP1200 architecture.

Figure 7 shows the experimental setup. We are using an Intel CA810E motherboard with a 733 MHz Pentium III CPU and a 133 MHz system bus. The RAMiX PMC694 card has a 266 MHz PowerPC processor. The primary PCI bus is isolated from the secondary PCI bus (local to the card) with an Intel 21554 non-transparent PCI-to-PCI bridge. Both the primary PCI bus and the secondary PCI bus (on the RAMiX card) run at 33 MHz and are 32-bits wide. The Pentium processor runs a Linux 2.2.16 kernel while the PowerPC runs our "raw" code with no operating system.

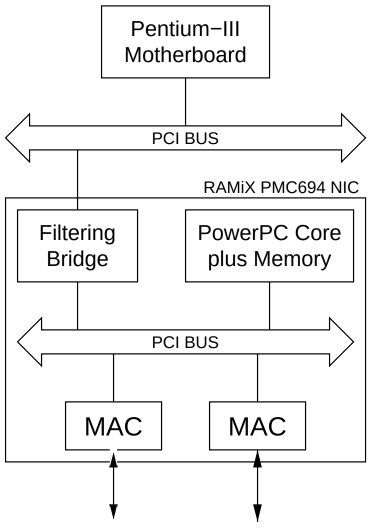

Figure 7: Block Diagram of our Network Processor-to-Pentium Testbed.

4.4.2 PCI Bandwidth

Before measuring packet turn-around time from the card to the host and back, we made several raw bandwidth measurements to determine the best method of moving the data. Both the host processor and the card processor can be the PCI bus master and initiate transfers. Using read or write instructions a processor can pull or push data across the bus. In addition, the card has a direct memory access (DMA) engine that is optimized for moving data. (These same options are are also available on the IXP1200.)

Table 2 summarizes the results of our experiments. Under programmed I/O (PIO), we measured several different data movement techniques. These differ in the number of bits transferred per loop iteration (Transfer Size). We measured transfer size of 64 bits and 128 bits by unrolling the 32-bit loop two times and four times, respectively. We obtained our best times by using the ANSI C memcpy library routine (from egcs-2.91.66) which uses tuned assembly routines. Our code was written in ANSI C.

4.4.3 Available Cycles

From Table 2, we see that the fastest way to move a packet from the card to the host and back to the card is for the card to push the packet to the host using its DMA controller and then have the host push the packet back to the card using memcpy-based PIO.

Table 3 shows the estimated maximum forwarding rate and available per-packet processor cycles for 64-byte and 1500-byte packets calculated from the best "push" rates in Table 2. The round trip rate is calculated as one-half the harmonic mean of the memcpy-based host push rate and the DMA-based card push rate. We estimate the available Pentium III cycles as the number of cycles that pass while the card is transferring the data to the host.

Packet Bytes	Round Trip Kpps	Available Pentium III Cycles @ 733 MHz
64	356.6	1372
1500	12.6	41700

Table 3: Estimated Maximum Forwarding Rate and Available Per-Packet Processor Cycles.

Packet Bytes	Round Trip Kpps	Available Pentium III Cycles @ 733 MHz
64	183.3	2706
128	139.6	2975
1500	24.6	10637

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

Table 4 shows the measured maximum forwarding rate and available per-packet processor cycles for 64-byte, 128-byte, and 1500-byte using the memcpy-based host push and DMA-based card push methods. Using this information, we can make predictions as to the number of packets which can reach the Pentium and the number of Pentium cycles available to those packets. Of course, this will depend on the packet rate as well as the packet-size mix.

Here we explore the boundaries for different workloads for the eight 100 Mbps links. We vary the packet size as well as the *message size*. The message size is the number of initial bytes of the packet which are sent

Bus Master	Transfer Size	Mode	Direction	64-Byte Packets		1500-Byte Packets	
				Kpps	MByte/sec	Kpps	MByte/sec
host	32 bits	PIO	push	919.9	58.88	39.37	59.06
host	64 bits	PIO	push	975.0	62.40	41.58	62.37
host	128 bits	PIO	push	962.4	61.60	41.53	62.30
host	(memcpy)	PIO	push	1073.4	68.70	44.29	66.44
host	32 bits	PIO	pull	61.1	3.91	2.61	3.91
host	64 bits	PIO	pull	61.8	3.95	2.63	3.95
host	128 bits	PIO	pull	61.9	3.96	2.64	3.95
host	(memcpy)	PIO	pull	62.2	3.98	2.66	3.99
card	32 bits	PIO	push	365.5	23.39	14.84	22.26
card	64 bits	PIO	push	363.4	23.26	14.81	22.21
card	128 bits	PIO	push	363.9	23.29	14.81	22.21
card	—	DMA	push	534.2	34.19	17.57	26.35
card	32 bits	PIO	pull	53.7	3.44	2.29	3.43
card	64 bits	PIO	pull	53.8	3.44	2.28	3.41
card	128 bits	PIO	pull	53.8	3.44	2.28	3.41
card	—	DMA	pull	354.1	22.66	15.74	23.61

Table 2: Raw PCI Transfer Rates

to the Pentium and back for processing. For our experiment, we used packet sizes of 64, 128, and 1500 bytes and message sizes of either the packet size or 64 bytes. (The 64-byte case is interesting because most router functions only require reading and/or modifying the first 64 bytes of the packet.) We used the measured 1.128 Mpps rate for the 64-byte packet case. From this we calculated a 564 Kpps rate for the 128-byte case and a 48.1 Kpps rate for the 1500-byte case. (This calculation is slightly conservative as the relative size of the Ethernet preamble is smaller for the larger packets.) Table 5 summarizes the results. The "Pentium Reach" column is the fraction of packets which could be sent to the Pentium (from Table 4) out of the packet rate given in Table 5. The "Pentium Cycles" column is the maximum number of processing cycles available for each of the messages which reach the Pentium. The values in the first four rows of the this column are from Table 4. The value in the last row is calculated as follows: From Table 4, we determine that the total time for one 64-byte packet is $1\,\text{pkt}/183.3\,\text{Kpps} = 5.46\,\mu\text{s}$. From the same table, we see that the Pentium was idle for $2706\,\text{cyc}/733\,\text{Mcps} = 3.69\,\mu\text{s}$. Therefore, the Pentium was busy for $5.46\,\mu\text{s} - 3.69\,\mu\text{s} = 1.77\,\mu\text{s}$ for that 64-byte packet. If we assume that the Pentium will be busy for the same amount of time in our third scenario, we calculate the idle time as $1\,\text{pkt}/48.1\,\text{Kpps} - 1.77\,\mu\text{s} = 19.0\,\mu\text{s}$. At 733 MHz, this corresponds to 13900 cycles. Of course, if fewer packets are handled by the Pentium, more cycles are available to the remainder. For example, in the case of 64-byte packets, if the Pentium only needs to process 10.3% of the 1.128 Mpps stream, then there are 5000 cycles available per packet.

Packet Size	Packet Rate	Msg Size	Pentium III	
			Reach	Cycles
Bytes	Kpps	Bytes	%	@733 MHz
64	1128.	64	16.3	2706
128	564.	128	24.8	2975
128	564.	64	32.5	2706
1500	48.1	1500	51.1	10637
1500	48.1	64	100.0	13900

Table 5: Available Per-Packet Cycles Under Various Scenarios. Message size indicates how many bytes from the packet are sent through (round-trip) the Pentium for processing.

In [7] the authors report forwarding rates on a 450 MHz Pentium II for a TCP proxy as 85.5 Kpps (11.7 μs per packet). If we subtract the measured 3.1 μs non-proxy overhead, we see that the per-packet cycle costs for the core TCP proxy code is 3900 cycles on the 450 MHz Pentium II. On a 733 MHz Pentium III, this corresponds to between 3900 and 6300 cycles (depending on architectural differences between the two machines). Given that Table 5 shows that we have between 2706 and 13900 cycles available, we conclude that there are practical router applications which could take advantage of the Pentium processing cycles.

5 Conclusions

We have evaluated the performance of the IXP1200 network processor, which uses parallel computing contexts to hide memory latency. We implemented a prototype IP router on this processor, and demonstrated that it can easily sustain line speeds for $8 \times$ 100 Mbps Ethernet ports. Moreover, by emulating infinitely fast network ports, we show that the IXP1200 is capable of forwarding minimum-sized Ethernet packets at a rate of 2.69 Mpps. Our experiments indicate that DRAM is the bottleneck. There is sufficient processing capacity available to get at least 26% improvement with faster memory.

Static allocation of resources plays a key role in our router design. The IXP1200 has seven processing units which run concurrently but share many resources. Dynamically scheduling and synchronizing different tasks in such systems is notoriously difficult to get right. By assigning all resources a fixed schedule, we made the system very predictable, and thereby easy to debug, measure, and optimize.

Rather than focus solely on maximizing packet rates, we also evaluate how such a system might be used to perform additional computation on each packet. This is an important consideration since routers are being programmed to provide increasingly sophisticated services. We demonstrate that at a 1.128 Mpps packet arrival rate (corresponding to 8×100 Mbps Ethernet ports), the IXP1200 has considerable head room. Static scheduling results in a fixed resource budget being available for processing each packet. These resources can be viewed as a virtual router processor (VRP) that, given a conservative measurement of our head room, can perform up to 4 DRAM accesses, 7 SRAM accesses, and 400 register instructions for each 64-byte chunk of a packet. Higher levels of the processor hierarchy—StrongARM and Pentium—can apply additional cycles to each packet.

While we have demonstrated that it is possible to apply non-trivial computational resources to packets, much work remains in specifying how new code is added to the forwarding path. This includes careful thought about where to draw the line between the fixed forwarding infrastructure and the variable VRP, verifying that extensions live within the VRP's resource constraints, and allocating chucks of the instruction space to different programs. We also hope that by precisely defining a "form factor" for such extensions, we will be able to squeeze more resources out of the μEngines; we currently use only 65% of the available spare capacity. These questions are all a focus of current research.

References

[1] D. S. Alexander, M. Shaw, S. M. Nettles, and J. M. Smith. Active Bridging. In *Proceedings of the ACM SIGCOMM '97 Conference*, pages 101–111, September 1997.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Poterfield, and B. Smith. The Tera Computer System. In *Proceedings of 1990 International Conference on Supercomputing*, pages 1–6, June 1990.

[3] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router Plugins: A Software Architecture for Next Generation Routers. In *Proceedings of the ACM SIGCOMM '98 Conference*, pages 229–240, September 1998.

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

[5] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 217–231, December 1999.

[6] PCI Special Interest Group, Hillsboro, Oregon. *PCI Local Bus Specification, Revision 2.2*, December 1998.

[7] L. Peterson, Y. Gottlieb, S. Schwab, S. Rho, M. Hibler, P. Tullmann, J. Lepreau, and J. Hartman. An OS Interface for Active Routers. *IEEE Journal on Selected Areas in Communications*, 2001. To appear.

[8] L. L. Peterson, S. C. Karlin, and K. Li. OS Support for General-Purpose Routers. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS–VII)*, March 1999.

[9] RAMiX Incorporated, Ventura, California. *PMC/CompactPCI Ethernet Controllers Product Family Data Sheet*, 1999.

[10] J. M. Smith, K. L. Calvert, S. L. Murphy, H. K. Orman, and L. L. Peterson. Activating Networks: A Progress Report. *IEEE Computer*, 32(4):32–41, April 1999.

[11] Vitesse Semiconductor Corporation, Longmont, Colorado. *IQ2000 Network Processor Product Brief*, 2000.

[12] D. Wetherall. Active network vision and reality: lessons from a capsule-based system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 64–79, December 1999.