Challenges in Implementing an ESP Service

Kenneth L. Calvert, James Griffioen, Najati Imam, and Jiangbo Li

Laboratory for Advanced Networking
Department of Computer Science
University of Kentucky
Lexington, KY 40506
{calvert, griff, najati, jiangbo}@netlab.uky.edu

Abstract. Although active network services have been widely studied, the task of mapping these services and algorithms onto actual network hardware presents an additional set of challenges. For example, high-end routers and switches are designed to handle as many interfaces delivering packets at wire speed as possible; in such an environment decentralized processing, pipelining, and efficient synchronization are crucial for good performance. At the low end (e.g. small routers/firewalls for the home), cost and flexibility are paramount; such systems are often structured as a general-purpose processor running modular software. Thus, the two environments are different and have different goals and objectives. We present a case study based on a representative active service called Ephemeral State Processing (ESP) that highlights many of the issues that arise when mapping services to real hardware. We discuss engineering considerations for ESP in both low-end uniprocessor and higher-end network processor scenarios, and present performance measurements from both implementations.

1 Introduction

Generally speaking, the active networking community has tackled research problems that are "forward looking", focusing on the development of a new network architecture that is radically different from the status quo. With a few exceptions, active networking research has mostly been conducted using idealized platforms and environments that tend to downplay or even ignore real world constraints. Reasons for this include the need to first understand active networking technology and applications themselves, and the perceived barriers to deployment in the present Internet. Although high-level designs and frameworks are a necessary first step, active networks will not become a reality until the problem of mapping active network technology onto actual network hardware—capable of meeting real-world requirements [19]—is addressed.

Constraints related to performance, cost, management, and operation dictate that networks be composed of distinct types of routers, in different roles and with different design objectives. Even if an active network architecture were deployed tomorrow, the need for different classes of routers would not disappear.

N. Wakamiya et al. (Eds.): IWAN 2003, LNCS 2982, pp. 3–19, 2004.

Consider the class of high-end backbone routers. A key design goal for these systems is performance: the ability to handle as many interfaces as possible, each delivering a highly-multiplexed stream of packets at high speed. In such an environment, decentralized parallel (per-port) processing, pipelining, and efficient synchronization of shared state (e.g. routing state) are critical. Despite the important role this router class plays in the Internet, relatively little attention has been given to the problem of mapping active network services onto such performance-oriented platforms. Indeed, many researchers believe active networking can never be viable at the core of the network, where high degrees of multiplexing are the norm, and performance is paramount.

At the other end of the spectrum are low-end routers, designed for use near the edges of the network. Their design objectives include low cost, flexibility, and ease of maintenance. Performance is less of a priority for these systems, because they may support only a handful of interfaces. Consequently, they may be structured around a general-purpose processor, with routing and forwarding functionality implemented in software. Because the actual computational cost of IP forwarding is modest, as are the line speeds supported, such systems typically have fairly meager processing capacity; this introduces new problems related to performance when they are extended to support active network technology. In addition, integration of active networking functionality into an existing modular software architecture can also be a challenge—particularly when it comes to packets creating and accessing shared state.

In this paper we present a case study of implementing an active network service on two different platforms. One platform represents the design considerations of higher-end routers, emphasizing low-level, "close to the hardware" parallel programming. The second platform represents the more cost-conscious end of the spectrum, providing a higher-level software environment that emphasizes flexibility and code re-use. We used Ephemeral State Processing (ESP) [8] as our representative active service. Like other services, it requires state at routers, executes end-system-specified code, and must handle worst-case conditions (minimum-sized packets arriving at wire speed on all interfaces). In addition to highlighting issues and lessons learned while implementing ESP in both environments, we present performance measurements that illustrate the benefits and drawbacks of each.

The remainder of the paper is organized as follows. Section 2 provides a brief overview of the Ephemeral State Processing (ESP) Service. We then describe our experiences mapping ESP onto two classes of routers in Section 3 and Section 4. Section 5 and Section 6 describe related work and offer some concluding remarks.

2 ESP: A Representative Active Network Service

Ephemeral State Processing (ESP) is a new network-level service that allows packets to create and manipulate small, fixed amounts (64 bits) of state at routers. Each piece of state has an associated *tag*, through which it can be accessed. Tags are chosen at random by users, so ephemeral state can be created

and used in "one pass"—there is no separate allocation step. By using the same tag, different packets can access each others' state in the network. While other active network services (notably ANTS [18]) have had similar features, the difference with ESP is that the state maintained at routers on behalf of users is *ephemeral*: it persists for a short, fixed amount of time, and then disappears. It cannot be refreshed, so there is no need for a deallocation process. Throughout this paper we assume 10 seconds as the fixed "lifetime" of all ephemeral state.

The key consequence of this approach is that creating a piece of ephemeral state involves a *fixed resource requirement*, namely 640 bit-seconds. Given bounds on the amount of state that can be allocated per packet and on packet arrival rate, this makes it possible to determine a lower bound on the size of *Ephemeral State Store* (ESS) needed to ensure that no attempt to create a new piece of ephemeral state will ever fail. As an example, if any packet can create up to two new pieces of ephemeral state, and if at most 10^5 packets arrive per second, an ESS with capacity at least 2×10^6 pieces of ephemeral state would never overflow.

2.1 Manipulating State

To compute with the state, ESP defines a set of *instructions*, each of which defines a small, fixed-length computation that updates information in the ESS based on information in the packet, or vice versa. Each ESP packet carries exactly one instruction, plus descriptions of its operands. When execution of an instruction finishes, the packet that invoked it is either forwarded or silently dropped, depending on the instruction and the result of the computation.

A simple example of an ESP instruction is the *count-with-threshold* instruction, which increments a counter (a value with a known tag, stored in the ESS), and if the resulting value is below a threshold, the packet that invoked it is forwarded; otherwise it is dropped. This instruction is useful in a number of different computations, and can be used to implement a simple form of duplicate filtering.

2.2 Ways to Use ESP

Unlike "heavyweight" active networking services designed for arbitrary processing of packets in the network, ESP is intended mainly to implement functions of an auxiliary nature. Examples of such functions include (i) modulating packet flow based on ephemeral state (e.g. to implement congestion control in multicast applications [16]); (ii) identifying nodes in the topology with specific characteristics (e.g. branch points in an explicitly-constructed multicast tree [17]); and (iii) processing small amounts of user data to improve scalability (e.g. consolidating receiver feedback in a large multicast application [8]).

Most computations take place in two phases, which we refer to generically as *setup* and *collect*. The setup phase generally distributes some information to the nodes involved in the computation; in the collect phase, some function over that information is computed and delivered to a controlling end system.

2.3 Designing for Wire-Speed Processing

If we want active services like ESP to operate at IP-like processing speeds, it is worth considering the techniques that are used in modern routers to achieve high-speed IP forwarding, namely *parallelism* and *pipelining*.

Modern high-speed routers consist of a collection of interface cards connected by a high-speed switching mechanism. The goal for such routers is to maximize aggregate packet-forwarding capacity, i.e. to interconnect as many high-speed interfaces as possible. The basic technique used to achieve this scaling is parallel processing: The forwarding lookup required for each packet is performed on the interface card at which the packet arrives. The result of this lookup determines the output card to which the packet must be switched, and the packet is then switched to that card. In addition, each port card may have multiple packet-processing engines, so that packets are processed in parallel even within a card.

One of the main challenges in fast IP forwarding is *memory latency*. At 10Gbps rates, a minimum-sized packet arrives about every 10–12 nanoseconds—far less than the time required for a *single* memory access for commodity DRAM. Obviously it is necessary to minimize the number of accesses to DRAM required per packet. Because routing tables are large enough to require the use of such relatively slow RAM technology, high-speed routers also rely on techniques such as *pipelining* to hide latency. Parallelism and pipelining are feasible because each IP datagram is handled independently of others; the only constraint is that processing should *preserve the ordering of packets belonging to the same flow*, where "flow" generally refers to packets traveling between the same pair of hosts. Assigning packets to forwarding engines based on a hash of their source and destination addresses is sufficient to ensure that this constraint is satisfied.

For ESP, the only constraints required for correctness of the service are:

1. Packets belonging to the same computation must access the same state store.
2. Packets belonging to the same computation must not be reordered.
3. Instructions accessing the same state store must appear to be executed atomically with respect to that store.

It follows that packets belonging to different computations may be processed in parallel, and indeed, may access different state stores. Thus, per-port implementations are feasible. In fact, we can use parallel implementations with *separate ESS's* on the same port card, so long as we can ensure that packets belonging to the same computation are processed using the same ESS.

Unlike IP, however, source and destination addresses are neither necessary nor sufficient to determine whether packets belong to the same computation. The ESP wire encoding therefore includes two important fields—the *Computation ID* and the *Processing Location Designator*—which are controlled by the user to ensure that the above constraints can be satisfied if ESP is implemented in a per-port, parallel processing architecture. Packets belonging to the same ESP computation must carry the same *Computation ID* value, and must be processed in the same ESS context.

The *Processing Location Designator* is necessary because a packet passes through two different port cards—input and output—on the way through a router. For some computations, processing needs to occur on the *input* side of the router, so that packets going to different destinations, and thus potentially leaving by different output port cards, can share state. (Identification of multicast branch points is one such case.) In other cases, packets going to the same destination from potentially different sources (input ports) need to be processed together, so the processing should happen on the output port. The application code at the user's host sets the location bits to control the processing location of each ESP packet. As long as these constraints are satisfied, implementors are free to subdivide Ephemeral State Stores as needed to improve efficiency.

In the next section, we describe our implementation, and how the parallelization and latency-hiding techniques that work for IP forwarding can also be used with ESP.

3 Case Study 1: Supporting High-Performance Active Services

A key objective of high-performance (backbone) routers is the ability to *handle packets at wire-speeds on as many interfaces as possible*. To achieve this level of performance, current (non-programmable) high-speed routers perform packet processing in hardware using application-specific integrated circuits (ASICs). This approach has been highly successful, producing routers that can switch millions of packets per second. However, the approach relies on immutable, special-purpose logic circuits to achieve high performance. The goal of active networks, on the other hand, is to allow end systems to define the processing on-the-fly (at runtime).

To address this issue, several vendors [10,2] have introduced *network processors* that can be programmed to perform packet processing, often in parallel. In some cases, the processing elements are general-purpose processors [1,6]. In other cases, the processing elements offer instructions specifically-designed to assist in packet processing [13,2]. However, in both cases, the packet processing is software-driven and can be modified on-the-fly (at least in principle). To achieve high-speed packet processing, many network processors support multiple processing elements executing in parallel.

Our case study uses the Intel IXP 1200 network processor as our development platform [10,13]. The IXP 1200 is designed for high-performance, deep packet processing, and has general characteristics similar to those of other network processors such as programmability, multiple processing engines, and a multi-level memory hierarchy. Consequently, we expect that insights derived from our experience with the IXP 1200 can be applied to other network processor platforms as well.

3.1 The IXP 1200 Network Hardware

The Intel IXP 1200 network processor is designed to facilitate the deployment of new value-added services simply by changing the software program. To achieve line-rate routing/switching, the IXP 1200 supports parallel packet processing on a set of six processors called *micro-engines*. The architecture is shown in Figure 1.

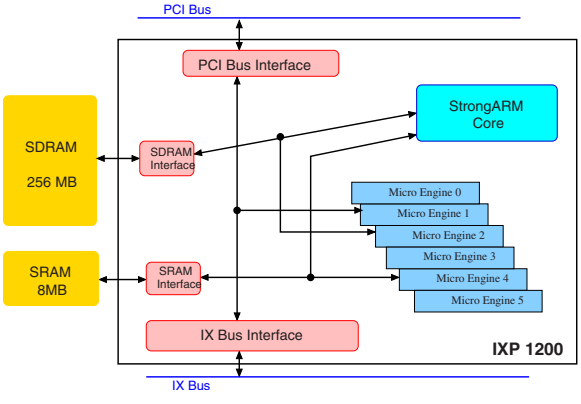

Fig. 1. IXP 1200 architecture

The main processor, called the *core*, is a 233 MHz StrongArm processor. The StrongArm core is not intended to be involved in normal data packet processing. It is primarily used for control functions; we do not discuss it further.

The key feature of the architecture for our purposes is the six on-chip micro-engines each supporting four *hardware threads*. Each of the micro-engines is individually programmable and supports a set of hardware instructions specifically designed for packet processing. The Intel IXP 1200 "Bridalveil" board we used supports four 100 Mbps Ethernet ports connected to the IXP 1200 via a 104 MHz IX bus. Like other network processors, the IXP 1200 uses a multi-level memory hierarchy consisting of registers, SRAM, and SDRAM. Register storage is the least plentiful (32 registers per thread context), but is the fastest. The SRAM is next fastest (and smallest), while SDRAM is slowest and most capacious. Our boards were equipped with 8 MB of SRAM accessed via a 32-bit 233 MHz bus, along with 256 MB of SDRAM accessed via a 64-bit 116 MHz bus.

To help hide the memory access latencies of SRAM and SDRAM, each micro-engine has four hardware threads, each with its own set of context registers. Memory interfaces are pipelined, and switching contexts among these threads is almost free; the result is that when one thread issues a RAM read or write (and thus incurs a long wait), another thread can typically begin executing immediately. To synchronize between threads, each micro-engine also supports a set of CAM locks that can be used to ensure mutual exclusion when necessary. If mutual exclusion or synchronization is needed between micro-engines, it must be implemented using shared memory (SRAM).

3.2 Mapping ESP to the IXP 1200

To achieve the desired level of performance, we defined a set of design goals for our IXP 1200 implementation:

- **Maximize Parallelism/Minimize Synchronization Overhead.**
- **Minimize Memory Accesses.** Map data structures to storage so that ESS information can be retrieved with as few accesses as possible.
- **Maximize ESS Capacity.** Use memory efficiently.
- **Maintain ESP Semantics.** Maintain the ordering and sharing constraints described above.
- **Continuous Operation.** In order to flush expired values from the ESS, a *cleaner* function is required that examines the ESS, compares the current time against the expiry time, and reclaims the location if the timer expired. This process should not interfere with data packet processing.

Parallelizing ESP. To achieve parallelism, we distribute the ESP processing across the set of IXP micro-engines. Intel's standard micro-engine assignments uses micro-engine 0 for ingress processing (from all interfaces), and micro-engine 5 for egress processing (to all interfaces). Although ingress/egress can, in theory, be done on any micro-engine, Intel discourages this—developing one's own code to read from the IX bus is known to be tricky and error prone. As a result, we only distribute the processing across micro-engines 1-4. Micro-engine 0 serves as the ingress processor and packet dispatcher (based on the computation ID as we will see below). Micro-engine 5 collects all outgoing packets and enqueues them for transmission.

Because we want the ESS to be as large as possible, we store the ESS data itself (tags and values) in SDRAM. Although SDRAM has the highest read/write latency of any of the IXP 1200's storage areas, it is only a bit higher than the latency of the SRAM, and it offers significantly larger capacity. Smaller auxiliary data structures (hash tables) are stored in SRAM for speed.

Related to the issue of where the ESS should be located is the question of "how many ESS's should there be". As described earlier, ESP requires at least one ESS per network interface. This is particularly helpful when line-cards are independent from one another with no shared memory between them. There are advantages, however, to having more ESS's per interface. First, partitioning computations into different ESS's reduces the number of tags per ESS, thereby reducing the probability of tag collisions. Second, reclamation of expired data requires synchronization between the cleaning thread and the packet-processing threads; on the IXP1200, synchronization across micro-engines is more expensive than between threads on the same micro-engine. Therefore in our design, each micro-engine has its own ESS, used exclusively for packets processed by that micro-engine. To ensure that packets are processed in the correct ESS context, we use a hash of the computation ID to assign packets to ESSs.

Given our choice of SDRAM for the ESS, the next challenge was masking its read/write access latency, given that each instruction requires at least one

read/write access to memory. Fortunately the IXP 1200 provides good support for this. As noted above, the micro-engine hardware threads allow pipelining to be implemented transparently: When one thread blocks waiting for a memory access another can immediately begin executing until it too blocks and hands off control to a third thread, etc. Although each individual read or write suffers the standard SDRAM latency, when pipelined in this way, the micro-engine is able to process multiple packets simultaneously.

Concurrent processing by multiple threads raises several interesting issues. First, to ensure that packets from the same computation are not processed concurrently by different threads while allowing parallelism, the packet demultiplexing code dispatches all packets having the same computation id to the same one of four input queues (not just the same micro-engine). Each thread can service any input queue; however, we use local thread synchronization techniques to ensure that no two threads work on the same queue at the same time. Second, given the challenges of synchronizing multiple concurrent threads, it might make sense to further subdivide the ESS into subESS's, with one subESS per thread. The downside of this approach is that memory can become fragmented, wasting valuable ESS space. Although it would simplify synchronization, we decided to stick with a single ESS per micro-engine (shared by all threads).

To support continuous operation of the router, we used one thread on each micro-engine as a *cleaner thread*, reclaiming expired (tag,value) pairs from the ESS. Because the cleaner examines and modifies the state of the ESS, there is potential for interference between it and a normal ESP instruction being executed concurrently by another thread. To address this issue we designed an efficient synchronization method that allows ESS accesses by both the cleaner and instruction-processing threads to be interleaved with minimal synchronization. The basic idea is this: every time a thread starts accessing the ESS, it sets a bit in a global "active threads" register indicating that is is "currently using an entry in the store". When a thread finishes an instruction, it unsets the bit. When the cleaner finds an expired entry, it does not remove it. Instead, it simply makes the entry inaccessible to any *new* ESP instructions, by removing its tag from the list of active tags. ESP instructions that are already in progress can continue to read and write the value until they complete (at which point they unset their bit in the "active threads" register). By monitoring the "active threads" register, the cleaner thread knows when an entry is no longer in use and can be safely deallocated. As a result, the cleaner runs in the background, concurrent with the instruction processing threads.

In order to focus as much of the IXP 1200 board's processing power on ESP as possible, we implemented ESP in the form of an external "pass-through" unit, which sits next to a conventional router, processing packets as they enter/exit on two of the router's ports. The idea is that all packets passing in/out of the router pass through the ESP box (just as they would in a port-card-based implementation in the router). This approach allows us to add ESP transparently to existing infrastructure without modification. Among other advantages, this makes it possible to deploy ESP incrementally. The *Location* field in the ESP

packet indicates whether processing should be applied on the input or output side, or both. Because a single micro-engine can support multiple lines running at full-speed, one IXP 1200 board is all that is needed to turn a cisco router with several ports into an ESP-enable router.

3.3 Performance Results

To measure the performance of our ESP implementation, we used the network configuration shown in Figure 2.

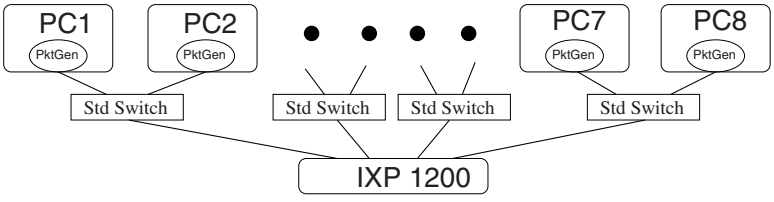

Fig. 2. Test network configuration.

Eight PCs were used to drive the IXP 1200: Two PCs were connected (via a standard switch) to each of the four (full-duplex) Ethernet ports. To generate traffic, we modified the Linux *pktgen* program to send ESP packets at various speeds. The pktgen throughput can be measured both in terms of bandwidth (Mbps) and packets per second. Because we are primarily interested in the number of packets per second that can be handled by the router, we configured pktgen to send minimum-sized packets (64 byte packets) in all our experiments. It should be noted that when using minimum-sized packets, pktgen was not able to saturate a 100 Mbps link to the IXP 1200; 62 Mbps was the maximum attainable rate. Consequently, we used two PCs to drive each IXP line. Even then, we only achieved a maximum rate of 76.25 Mbps per IXP line, because Ethernet cannot run faster than that when using minimum-size packets (due to the interpacket gap and other packet overhead). Thus the total offered load we were able to generate was (76.25 Mbps $\times$ 4 lines) = 305 Mbps.

To measure the overhead of ESP processing, we instrumented the IXP 1200 code to record the number of micro-engine clock cycles needed to perform a **count-with-threshold** ESP instruction. As a baseline, we measured the number of cycles it took to complete a single **count-with-threshold** (creating a new tag) using only a single thread (i.e., *no* pipelining of SDRAM reads/writes via multiple threads). The SDRAM read/write time dominated, consuming approximately 750 cyles out of the total of 1100 needed to complete the instruction. When the single thread also had to perform the cleaning, the achievable throughput was roughly 212K **count-with-threshold** instructions per second, or 108 Mbps.

We then measured the throughput with multiple threads. All threads executed on the same micro-engine, so there was no direct increase in parallelism; however, the use of multiple threads resulted in overlapping SDRAM accesses, hiding (at least some of) the latency of reading and writing. To generate enough traffic to keep the threads busy, all eight PCs simultaneously transmitted minimum-sized packets as fast as possible. Each packet carried a **count-with-threshold** instruction; 50% of the packets created a new tag, and 50% simply incremented an existing tag.

Table 1. Throughput of a single micro-engine, with and without cleaner-assist.

No Cleaner-assist	
No. Threads	Throughput
1	234 Kpps = 120 Mbps
2	420 Kpps = 215 Mbps
3	547 Kpps = 280 Mbps

With Cleaner-assist	
No. Threads	Throughput
1	212 Kpps = 108 Mbps
2	410 Kpps = 210 Mbps
3	507 Kpps = 260 Mbps
4	564 Kpps = 290 Mbps

Table 1 shows the throughput, both in packets per second and megabits per second, for 1, 2, and 3 threads. The fourth thread is the cleaner. The "No Cleaner-assist" table shows performance when the cleaner does nothing but clean. The "With Cleaner-assist" table shows performance when the cleaner uses its spare cycles to assist with packet processing. Clearly, at lower loads, the cleaner has more cycles to assist with packet processing than at high loads, and the throughput with one packet-processing thread plus the cleaner assist is comparable to the throughput with two packet-processing threads. Recall that the maximum achievable throughput (with minimum-size packets) is 305 Mbps due to Ethernet limitations. With three threads (plus the cleaner in its spare time) processing packets, the IXP comes within 5% of the full 305 Mbps. In fact, when we changed the mix of instructions so that all packets carry the same tag, the IXP operated at the optimal 305 Mbps rate. Note that this level of performance is achieved with only *a single micro-engine* doing packet processing.

To measure the impact the size of the ESS has on throughput, we measured the rate of successful completion of instructions under various transmission rates and ESS sizes. This experiment also tested the performance of the cleaner thread—if the cleaner thread is unable to free entries in a timely fashion, throughput will be affected. Figure 3 shows the average throughput (number of completed operations per second) over three runs, for increasing ESS sizes. For small ESS sizes, the ESS fills up quickly causing all subsequent ESP packets to be aborted until space can be reclaimed from expiring entries. The result is a limit on throughput that increases linearly with the size of the store. This allows us to pre-compute the store size needed to support any given line rate.

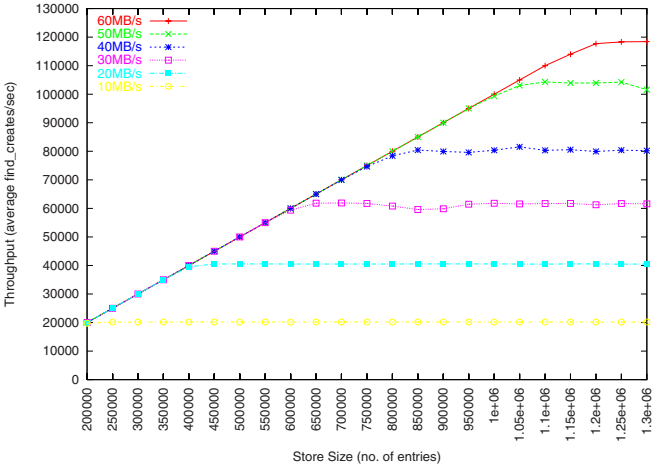

Fig. 3. Avg throughput as a function of store size.

4 Case Study 2: A Modular Software ESP Implementation

Not all routers emphasize performance and scalability as their primary design constraints. Many of the non-core routers in the Internet interconnect only a few networks at modest speeds; for them, wire-speed forwarding is achievable using simpler techniques. The last few years have seen a proliferation of "broadband router/gateway" products that offer bundled functionality in a simple, inexpensive package targetted at homes and small offices [3,5,4]. The design considerations for such routers are quite different from those of the high-end routers; instead of performance and parallelization, these systems focus on robustness, flexibility, and minimizing cost. Moreover, what differentiates these products are their features; the ability to add or change features easily is therefore an important design goal for these systems.

For these routers, a modular software design is of course desirable. By coding the services in software (as opposed to hardware), changes can be made and distributed very easily. Also, designing the software in a modular fashion, with well-defined functional interfaces between components, allows new services to be incorporated quickly, leveraging existing code thereby reducing the size of the code footprint.

In this section we describe an implementation of ESP for *click*, a modular software router.

4.1 The Click Modular Router Platform

Click [11] is a fine-grained modular software approach for implementing routers that need to offer advanced packet-processing features; as such, it is ideal for our purposes.

Click uses the Linux kernel as a base platform, with low-level modifications to handle hardware devices via polling instead of interrupts; this improves click's performance under heavy loads. The basic paradigm of click is to decompose router functionality into small functions that can be performed more or less independently, and encapsulate them in individual modules called *elements*. Elements are organized into a graph according to packet flow; each element consumes packets from *input ports* and emits them on *output ports*. The user controls packet-processing functionality by configuring paths in the graph to include the desired sequences of elements.

Click comes equipped with predefined elements for standard IP processing, including header validation, TTL decrement, packet classification, etc. A standard configuration graph for an IP router with two interfaces is shown (in solid boxes) in Figure 4.

Click does not provide any explicit support for pipelining or parallel processing of packets; the version we used has a single thread and is appropriate for general-purpose uniprocessors (PCs), whose architecture does a reasonable job of hiding memory latency without explicit intervention by the programmer.

4.2 Implementation Considerations for ESP in Click

The first question to be answered in implementing ESP on the click platform is how to modularize it. The click philosophy is for modules to be small and simple; this would suggest separate modules to validate the ESP header, retrieve values from the ESS, execute ESP instructions, and classify packets. This approach has advantages with respect to understandability, maintainability, and reuseability of code. However, a highly-decomposed implementation raises other issues.

Probably the biggest issue is that of support for modules that share state. The usual method of sharing state in click is to *annotate* the packet with the state necessary for its processing; information created by one module travels with the packet. This capability is used, for example, in Click's ICMP implementation to detect and send a redirect message if a packet goes out the same interface it came in on. However, when information needs to be shared among multiple packets *and* multiple modules (as would be the case in a highly-decomposed ESP), additional synchronization mechanisms are needed. Inter-module synchronization can hurt performance, and also complicates the code—thereby offsetting some of the benefits of small modules.

Another possibility is to define an ESS module that exports methods to create, retrieve and update values. However, such methods must be explicitly named in the importing modules; this complicates things when there may be multiple instances of the exporting modules (say, input, output and centralized ESS contexts).

It is worth noting that IP packet processing generally does not require much state-sharing across different functions, and where functions do need to share persistent state, the standard click approach seems to be to encapsulate the functionality in a module (e.g. RoutingLookup and ARPRequest). We took the same approach, and implemented most of the ESP functionality, including instruction execution and state store management, in a single module called *ESPExec*. Some of the benefits of modularity are given up this way, but probably not too many. For example, it is not clear that there is much need for re-use of functional modules within ESP. In addition, the standard composition technique in click has well-known performance costs, and an integrated implementation avoids them.

A second issue is whether and how to implement multiple ephemeral state store contexts per interface. As noted in Section 2, the ESP implementation must ensure that packets with the same computation ID are executed in the same ESS context. To achieve this, it is of course sufficient to implement a single large store and use it for all ESP packets. In a single-threaded uniprocessor implementation with ample memory, there is little performance penalty for doing this, although it slightly increases the probability of tag collision. It avoids any need for synchronization between packets at all.

However, it turns out that separate ESSs *are* necessary for different interfaces. The reason: some global computations associate different semantics with the different processing locations. The best example is when ESP is used to implement *subcasting*, i.e. forwarding packets along a subtree of an existing multicast tree. The idea is that the branches of the tree on which the packets should be forwarded are marked (during the setup phase) with ephemeral state; when a data packet comes along (carrying a *piggybacked* ESP instruction), it checks for the presence of the state, and drops the packet if it is not present. This dropping needs to happen on the *outgoing* interfaces, i.e. after a packet has been duplicated in a router. In a router in which all ESP locations share a single ESS, all branches of the tree leaving that router would get marked if any one of them is marked, and thus subcasting would not be effective.

Thus, we need to implement a separate ESS for each interface on the router. However, there does not seem to be any reason for further subdivision, so our implementation only implements an ESS for each port, plus a "centralized" ESS.

The design of our click ESP implementation is shown in Figure 4.

4.3 Performance Measurements

We report measured performance of our ESP implementation to demonstrate that adding ESP processing to the click path does not result in an unreasonable additional delay. The version we used in our evaluation ran as a Linux Kernel module, executing on a standard Pentium 4 processor with five 10/100 Mbps ethernet ports. (Note that this is a reasonable approximation of the configuration of routers used in low-end home/small business configurations.)

The click authors report the time to process an IP packet is roughly 2900 ns on a PIII running at 700 MHz for a total forwarding rate of 345 Kpps [11]. To baseline our system, a P4 running at 1.8GHz, we reran the same experiment as

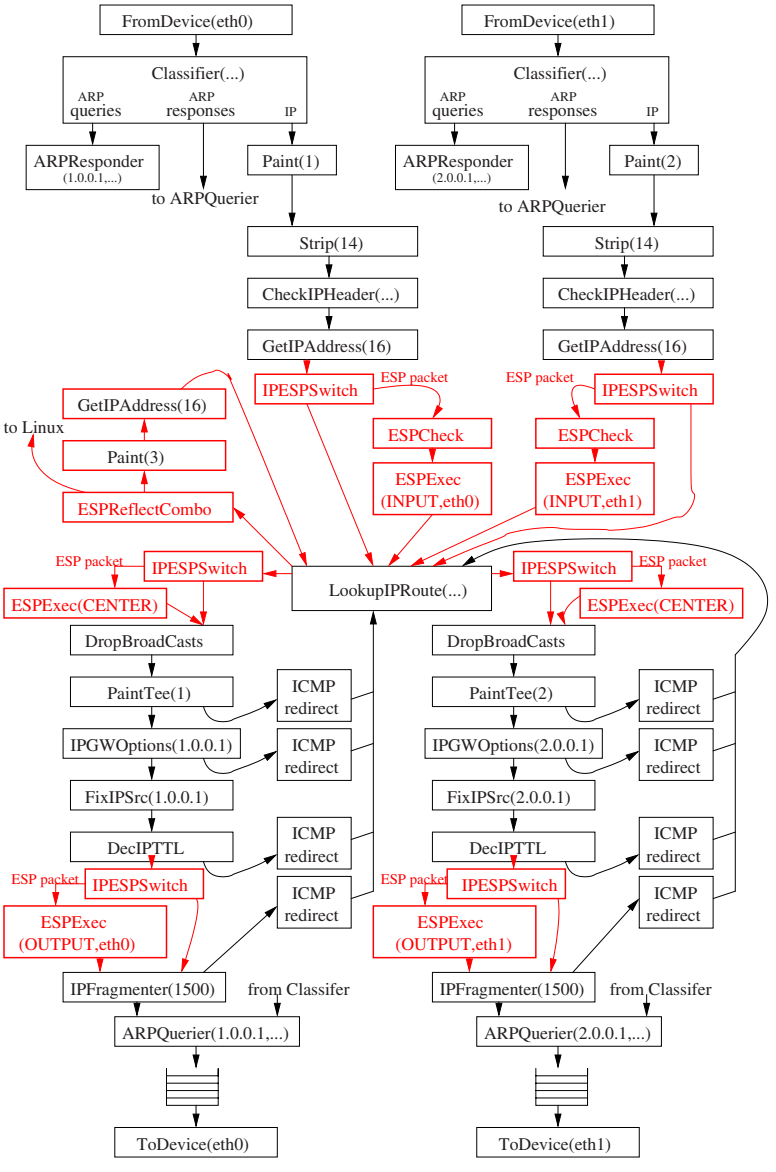

Fig. 4. Router configuration with ESP

the click authors, and found the time to process an IP packet to be roughly 1300 ns for a total forwarding rate of 765 Kpps. Given this base-level performance, we are able to gauge the overhead required by ESP.

The additional ESP processing cost depends on the ESP instruction. The **count-with-threshold** and **compare** instructions are only 55% more costly

Table 2. Added cost of ESP processing.

count	compare	collect	rcollect
660 ns	702 ns	1064 ns	1525 ns

than standard IP processing. The ESP **collect** instuction, on the other hand, involves more tags and thus increases the overhead by 81%. For the more complex **rcollect**, the overhead is about 116%. Although the overhead is substantial, the worst case throughput (with minimum sized packets) is still 354 Kpps or 181 Mbps, which is sufficient to keep all lines on a home router busy.

5 Related Work

Numerous active network platforms have been developed, with varying degrees of (re)programmability. The FPGA-based programmable port card developed at Washington University in St. Louis is an example of a hardware platform whose data-path behavior can be controlled reprogramming the on-board FGPAs [15]. By loading a new bitfile into the FPGA, almost arbitrary processing functions can be defined; the platform includes commonly-used functions and "glue" for connecting modules together. This hardware-based approach provides very high performance for simple functions that need to be applied to packets on the fast path at specific (known) points in the network.

A language-oriented approach that features provably-bounded resource requirements is SNAP [14]. SNAP permits simple, straight-line programs to be carried in packets, allowing users to define the processing applied to their packets. Unlike ESP, SNAP is designed to replace IP as the network layer.

NetBind [9], from the Genesis project at Columbia University, is a tool supporting dynamic construction of datapaths in a network processor. NetBind is based on general ideas and intended to be used on any network processor, though the current design is tailored to the layout of the IXP1200. While NetBind is not an approach to injecting programmability into the network, it addresses one of the most difficult and important parts of programming network processors: dynamic composition of datapaths. By modifying the code store on an IXP1200 microengine at run time, NetBind creates pipelines dynamically according to each packet's needs.

Each of these systems is designed to allow reprogramming on-the-fly, at timescales varying from per-packet (SNAP) to configuration time (DHP). ESP, however, supports a different form of programmability, in which the set of processing options is fixed, but they can be invoked in different orders.

We are aware of few efforts to deploy active network technology on production platforms. The ABone [7], or Active Networks backbone, was a testbed composed of overlay nodes running a software-based active network platform, and supporting a variety of execution environments. Most of the approximately

100 nodes of which the ABone was composed were general-purpose PCs running a variant of the Unix operating system.

Another effort [12] defined an interface on a Nortel router by which processing (e.g. encryption) for particular flows could be configured and turned on and off under user control.

6 Conclusions

We have described two implementations of the Ephemeral State Processing service on real router platforms, together with some of the engineering considerations that went into them.

Our implementation for the Intel IXP1200 was intended to demonstrate that the design goals for ESP were achieved, i.e. that it can be processed at rates comparable to those achievable for IP packets, given suitable hardware support. Our measurements show that store capacity is the limiting factor on throughput of ESP packets when servicing 100 Mbps Ethernet links. Given the modest processing power of the IXP1200, we believe this number can easily scale up to line rates in the gigabits per second for next-generation network processor platforms, which feature both higher-speed engines and greater parallelism. Our results show that ESP can indeed take advantage of additional parallelism to hide memory latency, which decreases much more slowly than processor cycle times.

Our experience with the click software router platform highlights some of the challenges of modularizing active services that involve state-sharing between packets. Our design is well-suited for a uniprocessor router platform. In the future, we will investigate how well it could be mapped onto a multiprocessor architecture. Although the basic click router design seems consistent with the kind of pipelining and parallelism that would be possible on a multiprocessor, as we have noted IP forwarding does not require inter-packet state management. Meanwhile the question of which is better suited for such an environment, a monolithic approach like ours, or a fully-decomposed modular implementation, remains open for now.

It should be noted that measurements of both of our implementations show that achievable throughput rates for ESP processing are comparable to those for IP. This supports our claim that ESP is "IP-like" in terms of its resource requirements. However, because ESP packets are also IP packets, they must undergo *both* types of processing. Thus adding ESP should no more than double the per-packet processing cost. Our click measurements confirm this; it seems a reasonable price to pay for the added flexibility provided by the service.

References

1. BCM4702 airforce wireless network processor, September 2003. http://www.broadcom.com/products/4702.html.
2. IBM PowerNP network processor, August 2003. http://www.zurich.ibm.com/cs/network%5Fproc%5Fhw/index.html.
3. The Linksys WRT54G Wireless-G broadband router, September 2003. http://www.linksys.com/products/product.asp?grid=33&scid=35&prid=508.
4. Linux-based routers and switches, September 2003. http://www.linuxdevices.com/articles/AT2005548492.html.
5. US Robotics model 8200 router, September 2003. http://www.linuxdevices.com/articles/AT4486854045.html.
6. Vitesse eq 2000 network processor, September 2003. http://www.vitesse.com/products/categories.cfm?family%5Fid=5&category%5Fid=16.
7. S. Berson, B. Braden, and S. Dawson. Evolution of an active networks testbed. In *DARPA Active Network Conference and Exhibition (DANCE) 2002*, May 2002.
8. K. Calvert, J. Griffioen, and S. Wen. Lightweight network support for scalable end-to-end services. In *ACM SIGCOMM 2002*, August 2002.
9. A. Campbell, S. Chou, M. Kounavis, V. Stachtos, and J. Vicente. Netbind: A binding tool for constructing data paths in network processor-based routers. http://www.comet.columbia.edu/genesis/netbind/overview/netbind.pdf.
10. Intel Corporation. The IXP1200 Hardware Reference Manual, August 2001.
11. E. Kohler et al. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.
12. T. Lavian et al. Enabling active flow manipulation in silicon-based network forwarding engines. In *DARPA Active Network Conference and Exhibition (DANCE) 2002*, May 2002.
13. Erik J. Johnson and Aaron Kunze. *IXP-1200 Programming*. Intel Press, 2002.
14. Jonathan T. Moore, Michael Hicks, and Scott Nettles. Practical Programmable Packets. In *IEEE INFOCOM*, Anchorage, AK, April 2001.
15. D. Taylor, J. Turner, J. Lockwood, and E. Horta. Dynamic hardware plugins (dhp): Exploiting reconfigurable hardware for high-performance programmable routers. *Computer Networks*, 38(3):295–310, February 2002.
16. S. Wen, J. Griffioen, and K. Calvert. CALM: Congestion-aware layered multicast. In *IEEE Conference on Open Architectures and Network Programming (OpenArch)*, 2002.
17. Su Wen, James Griffioen, and Kenneth Calvert. Building Multicast Services from Unicast Forwarding and Ephemeral State. In *IEEE OpenArch 2001*, Anchorage, AK, April 2001.
18. David J. Wetherall, John V. Guttag, and David L. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols, 1998.
19. Tilman Wolf and Jonathan Turner. Design issues for high performance active routers. *IEEE Journal on Selected Areas of Communications*, 19(3), March 2001.