

# Using Network Processors for Packet Filtering

Bruce R. Millard  
Division of  
Computing Studies  
Arizona State University East  
Mesa, AZ, USA

Shyamal Pandya  
Department of  
Computer Science and  
Engineering  
Arizona State University  
Tempe, AZ, USA

Donald S. Miller  
Department of  
Computer Science and  
Engineering  
Arizona State University  
Tempe, AZ, USA

*Abstract - This paper presents a hardware/software design and implementation that uses modern network-processors as packet-filtering devices that can be used for advanced network applications such as firewalls, network address translation, intrusion detection, traffic shaping and others. Included are the motivation and design and implementation tradeoffs of a hierarchical and pipelined, packet-filter software package using the Intel IXP 1200 network processor. The approach adapts Linux netfilter/iptables software and interfaces as a framework for providing wire-speed packet filtering for modern high-speed networks with complex traffic demands. The Intel IXP 1200, contained on a PCI card in a standard off-the-self personal computer, provides a hardware base that allows for a three-level, hierarchical multiprocessor software architecture. Also included in this paper is an attempt to quantify performance improvements that can be expected from the next generation of Intel network processors and other similar devices using an extrapolation of the software parameters included in this packet filter design.*

**Keywords:** network processors, packet filter, multiprocessor, computer network protocols

## 1.0 Introduction

The trends are clear: network bandwidth continues to increase dramatically, network device applications, such as bridging, switching, routing, firewalls, network address translation, virtual private networks, intrusion detection, and prevention, multicast routing, traffic shaping, etc, are continuing to increase in complexity and the market continues to demand that separate applications be merged into one device. Network utilization continues to increase to match the available bandwidth, and single processors mixed with ASICs, and FPGAs do not meet these demands. So the logical solution to these requirements includes multiple processors. Additionally, a clean division of work effort and functionality is needed to use them. This requires

decomposition and repackaging of functions, placing some in hardware and some in software so that the resulting system can scale and adapt more easily to handle increased loads and alternative functionalities and configurations. A way to do this is to move functionality that is simple but required frequently to multiple high-speed parallel computing engines. More complex processing tasks, which happen less often, can then be executed where their complexity is better supported. This approach is what we investigate in this paper.

The usual division of network-application functionality is management, control and data planes. Management happens least frequently and often involves human interaction. This processing can be slow and, frequently requires large long-term storage. Conversely, data plane operations must happen at the speed of packet transfer, require only small amounts of memory in addition to memory for the network packets and the numbers of instructions per data plane operation is small on average. The control plane interactions happen less frequently than data plane operations and frequently require more complex operations such as handling exceptions.

Merging modern network processors and open software environments provides a way to investigate this approach. There are several modern network processor boards based on multiprocessors [1]. We chose the Intel IXP 1200 [2] as the basis for our research because we could construct a hierarchical arrangement of the processors into three layers and because the bottom layer has very simple RISC processor and microengines that we could use to examine how simple the data plane processing could be. With this in mind we employed a widely used packet level application (the Linux *netfilter/iptables* packet filter [3]) that gave us an open source

software framework that could be used to examine packet filtering, a network application that could benefit from multiprocessors. Packet filters are used in all the network applications mentioned above.

In the following section we present the project environment. Section 3 discusses packet filter functionality and design. In section 4 we discuss our design and implementation. Section 5 covers testing and the results we obtained. Section 6 discusses how using the next generation of the IXP Network Processor family would impact the results of our design and implementation. Finally, we present related work and a summary.

## 2.0 Project Environment

### 2.1 Packet Filters

Simplifying, packet filtering is the process of examining a network packet's headers and deciding its fate. Examining a packet involves using bit-map-based comparisons of headers at well-known bit positions for information needed to make the decisions.

Packet operations may also involve changes to the packets for routing or other network operations. After a packet has been looked at or modified, the information gathered leads to additional actions. Examples of actions are: dropping the packet, forwarding to the next stage for further processing, routing to an exit point or dealing with an anomaly. The two most widely used open-software, packet filters are *iptables* [3] and *BPF* [4].

### 2.2 IXP 1200

Figure 1 shows a simplified block diagram of the IXP1200 network processor. The IXP1200 is based on the Intel IXA architecture. Its major components are a StrongARM processor, six programmable multi-threaded microengines with four contexts per microengine, interfaces to SRAM, SDRAM and a PCI bus, some specialized functionality in the FIFO Bus interface unit (FBI) unit and an interface to an Intel bus called the IX Bus [2].

Two external memory stores are available to the IXP1200. This memory can be used by the StrongARM, the microengines and other devices connected to the PCI bus. In addition to SDRAM there is a higher speed SRAM. In normal operation, the SDRAM holds large data structures

and packet data while packets are transitioning from input to output ports. The SRAM typically holds control data, e.g., queue meta-data and routing tables.

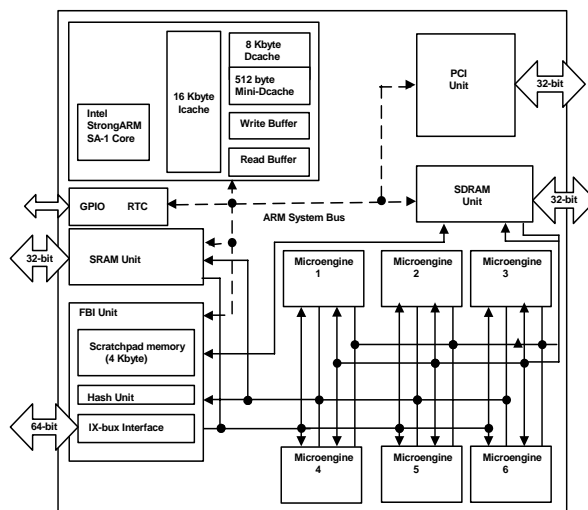


Fig. 1. Intel IXP 1200 Block Diagram (adapted from [2])

The FBI unit provides a collection of hardware resources that are used by the core and microengines to improve packet processing performance. These are scratchpad memory, internal Control and Status Registers, hash creation, IX bus interfacing and an interface to packet MAC unit FIFOs. The 4KB of fast scratchpad memory is used to communicate small control data between the microengines and core processor and to synthesize simple locking primitives.

This project used a RADISYS ENP 2505 [5] network processor board with an IXP 1200, 256 MB of SDRAM, 8 MB of SRAM, 8 MB of flash RAM and 4 10/100 Mbit 802.3 transceivers. The StrongARM core and microengines were clocked at 232 MHz and the memory buses at 116 MHz. The IX bus operated at 66 MHz. Connection to the host processor is through both a PCI bus and a serial line. The flash is used as a RAM-disk and after booting contains Linux which is operated by a root session through the serial line. Host and board PCI drivers were configured so that standard TCP/IP communication was used between host and the ENP 2505 Linux.

### 2.3 Software Environment

The StrongARM core processor runs an ARM version of the Linux kernel. The software

development environment is composed of the following major elements: a cross-compilation tool-chain that runs on Linux; microengine programming using IXP Microcode [6] and MicroC [7]; and a microengine assembler, linker and loader running under Windows NT/2000 (the available support at the time we started).

The setup used during the implementation of this project was a single Pentium IV based host machine with the ENP-2505 board in one of its PCI slots. The host ran the Linux kernel, which made development of StrongARM programs easier. For microengine development it was necessary to have Windows 2000. Our system used VMware [8] software to generate a Windows virtual machine on top of the host Linux operating system.

### 3.0 Packet Filter Design

This section discusses the design issues related to the IPv4 [9] packet filter that we implemented on the IXP1200. The packet filter is based on the Red Hat Linux iptables system [3].

#### 3.1 Rationale

The IXP1200 system is aimed at bandwidth hungry network applications and services, especially on the network access, edge and core. While services like routing and forwarding have been explored in terms of design on the IXP1200 [10], packet filter implementation remains relatively untried.

The choice of iptables as the basis of the packet filter was driven by several factors. The most important was that the design of the iptables system is well modularized so that extending it involves implementing only a small library of functions and a driver module. This has resulted in a clean design that has kept the various packet filtering extensions as isolated and well-defined groups of functions. Since one of our goals was to explore dynamic re-tasking of the microengines during the operation of the packet filter, the functional modularity became a critical factor in its choice. Dynamic re-tasking can be triggered by the addition of a new class or set of rules to the filtering table.

Another factor is that *iptables* is implemented on Linux and the StrongARM core and PC host run Linux. Thus part of the user interface of iptables could be used unchanged.

#### 3.2 Iptables Functionality

The iptables framework actually consists of two distinct parts:

- 1) *Netfilter* is a set of hooks (code that checks for registered procedure calls) at several fixed points within the Linux kernel's networking code. At any of these hooks, a series of callback functions can be registered that can manipulate packets that arrive at the hooks.
- 2) The *iptables* package consists of a set of modules that maintain tables of rules that a packet must match for a corresponding action to take place. A variety of modules exist, such as: ordinary packet filter, Network Address Translation and a mangle module for packet manipulation.

The *netfilter* part of the infrastructure is embedded in the Linux network stack (code). Since the packet filter was implemented for the IXP1200 microengines, the *netfilter* portion of the system was not directly involved. Instead, the *netfilter* hooks were emulated by inserting calls to the filtering microcode at relevant points as the packets traverse a path through the microengines.

#### 3.3 Iptables Design

Iptables is composed of a number of modules implementing a variety of services based on packet inspection and manipulation. In this discussion we focus on one of those services, packet filtering. However, the other network applications (NAT, etc.) mentioned above can, in general, be implemented in similar fashion.

In iptables a packet traverses a set of rules that specify the parts of packets to match with specific values. If a packet passes the match, a corresponding action as specified in the rule is taken. The most common actions include accepting the packet, dropping the packet and continuing to the next rule. At each hook each network application registers a table composed of a set of rules, rules may be organized into chains.

#### 3.4 IXP1200 Specific Design

In terms of a typical network application, the various components of iptables as described in the previous sections are mapped into the planes and IXP 1200 as follows.

The *data plane*, or the fast path portion of iptables, consists of the actual packet filtering

algorithms. These algorithms were implemented on the microengines.

The *control plane* consists of the portion of the application that manipulates the filter table in memory. Also present at this point is the control plane functionality of the forwarding module of the application. The control plane is implemented on the StrongARM core processor.

The *management plane* consists of the user interface of iptables. The iptables user interface enables a user to manipulate the filter tables. The user interface was implemented on the StrongARM core processor. The Management Plane would ideally be implemented on the host processor but time did not permit this implementation.

## 4.0 Packet Filter Implementation

This section points out some of the implementation issues of the packet filter software. In particular, the task partitioning across the microengines and re-tasking of microengines

### 4.1 Packet filter implementation design

The programming environment provided with the IXP 1200, called the Intel IXA Software Development Kit (IXA SDK) [11], supports a programming framework called an Active Computing Element (ACE) that encapsulates the tasks that perform independent packet processing functions. Associated with an ACE may be a piece of code that runs on a microengine, which is called a MicroACE. Code that runs on the microengines in a MicroACE is frequently called a microblock. More than one microblock may be assigned to a microengine.

The packet filter code distributes iptables sub-tasks into a number of software components based on the MicroACE framework. The tasks also extend to the StrongARM.

We distributed much of the data plane functionality across a set of MicroACEs. These are the *Ingress*, *PacketFilter*, *Egress*, *Stack* and *Forwarder* MicroACEs. Each of these is described very briefly below. (See Figure 3 for a pictorial view of MicroACE interconnection.) The *Ingress*, *Egress*, *Forwarder* and *Stack* MicroACEs were provided with the IXA SDK.

The *Ingress* MicroACE handles the tasks associated with packet arrival. The *Egress*

*MicroACE* handles the tasks associated with packet transmission. The *Forwarder* MicroACE performs level 3 (dataplane level) forwarding of packets. The *Forwarder* microblock does lookups for next hops in a forwarding table that is maintained in SRAM. With this microblock there are many possible exception conditions, such as the absence of routes for a particular destination, or the packet is an ARP or ICMP packet. In all such cases the *Forwarder* microblock routes the packets to its core component. The function of the *Stack* MicroACE is to take an incoming packet and present it to the Linux kernel's protocol stack. The *Stack* MicroACE is used for packets that are destined for the StrongARM.

The *Packet Filter* MicroACE encapsulates the packet filtering functionality, including the microblock that has the packet filter algorithms implemented in microblock code and filter table management in the C language, implemented as the core component. The core component of this MicroACE also implements a cross-call API that can be invoked by a non-ACE task. The API is used by the core user interface implementation to manipulate the filter table.

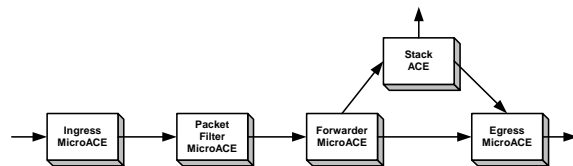


Fig. 3. Possible packet paths through the MicroACEs

### 4.2 IXP 1200 Microengine

Each enabled microengine runs a dispatch loop over its four contexts to control application flow and the flow of packets through the application. The dispatch loop calls the various microblocks inside that microengine. Each microblock works on a single packet. When it is done with the packet processing in one microblock the next microblock is executed, if there are no other microblocks the packet is queued for the next microengine. Upon exception, the packet is queued for the core processor with a tag value that represents the microblock that generated the exception so that the related core component can address it.

### 4.3 Filtering Extensions

Another goal of this research was to extend

the packet filter code to add new filtering capabilities. The objective was to design a way to re-task an already running microengine with the new extension. For the packet filter implementation, we decided to add TCP header matching code.

The introduction of runtime dynamic re-tasking to the packet filter application was accomplished by adding the TCP header matching functionality to an already running packet filter instance in the microengines. If we change just a table we have to swap the old table data structure for a new one. When adding new matching code we also have to change the base microcode (microblock image).

The steps taken to accomplish this ensure that each context interrupts the core only after it is done with any packet processing, and before it starts processing another packet. Thus, the rule change takes place between packets. The steps are similar for table updates except that code re-association and variable patching and thread death are not necessary, only the microengine-thread/core signaling remains.

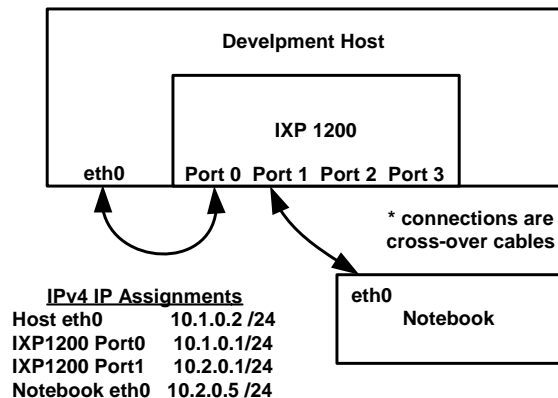


Fig. 4 Computer set-up for experiments

## 5.0 Results

### 5.1 Test Setup

Since the results we were evaluating were not related to overall packet throughput our test set-up was simplified. All we needed was the IXP 1200 board and two hosts acting as packet generator and receiver. To simplify matters even further we used the machine hosting the IXP 1200 as one of the external computers. The experimental setup is shown in Figure 4. The

Libnet [12] open-source library was used to generate TCP, IP and ICMP packets with various headers.

### 5.2 Experiment results

Packet filter operation, task partitioning performance and microengine re-tasking experiments were run. The size of the microcode per implemented functionality, in terms of the number of microengine instructions, relates to the programmability of the microengines.

*Code Sizes* The size of microcode in terms of microwords is shown in Table I. The number of microwords generated for each microengine configuration implemented is shown.

TABLE I.  
NUMBER OF MICROWORDS PER  
MICROENGINE CONFIGURATION

Configuration	Microword Count
Ingress + IP header match (Core packet filtering code)	786
Ingress + IP header match + TCP header match (Extension to core code)	969
Re-tasking code	5
Ingress + IP header match (Core packet filtering code) + Forwarder Microblock	973
Forwarder Microblock	430
Egress Microblock	554

*Effects of task partitioning* The evaluation of the effect of distributing the tasks (filtering and forwarding operations) across two microengines was facilitated by using the TCP header matching code. To accommodate this new code, it was necessary to move the forwarding code to another microengine because of instruction store limitations. To determine the effects on packet throughput the performance was compared using two configurations. In the first configuration only two microengines were used, the first microengine running the Ingress, PacketFilter and Forwarder microblocks and the second microengine running the Egress microblock. The PacketFilter microblock did not contain the TCP header matching code. In the second configuration three microengines were used, one running the Ingress and PacketFilter microblocks,

the second running the forwarder microblock and the third running the Egress microblock.

Ingress to Egress packet delay was averaged over ten thousand packets. The result of adding the third microengine was an average additional 0.3  $\mu$ sec delay per packet or about 12 percent.

Retasking operation This experiment verified the correct functioning of the microengine re-tasking code when the addition of a rule specifying the TCP protocol triggers microengine re-tasking. Also an iptables rule was added to cause a constant flow of traffic from the host port (see Figure 4) into the IXP1200 so that the microengines would be busy processing packets. The microengine was re-tasked successfully with no loss of packets.

## 6.0 Next Generation Extensions

Another goal of this research was to use the experiences gained in the above design and implementation to predict the benefits of a higher functionality/performance network processor. To facilitate this goal we chose to extrapolate the tasks done here to a higher performance more recent network processor within the IXP family, the IXP 2400/2800 series [13] network processors. The IXP1200 and IXP2400 differ in numbers of microengines, contexts per microengine, GP registers per microengine and SRAM and SDRAM transfer registers as well as in instruction store size, microengine frequency, maximum DRAM and SRAM supported and Next Neighbor registers (the IXP1200 has none).

Arguably the most important new feature is the next neighbor registers. Each microengine has a 128 next neighbor register set. The next neighbor registers are used to share data with neighbor microengines. This new functionality dramatically speeds up the transfer of data and notification of the presence of packets between microengines because no memory operations are required. Very briefly our parameterization study indicated the following. The details of this study are in [14]

1. Quadrupling store per microengine enables each microengine to handle about 12 more functional components.
2. Increasing the number of microengines from to 8, doubling the number of contexts from 4 to 8 and increasing the operating frequency from 232 to 600 MHz enables increasing the peak

bandwidth to 19.2 Gbps and enabling the IXP2400 to service 20 ports for the same performance as the IXP1200 with 4 ports.

3. Next neighbor register set reduces packet memory access latencies for queuing and saves considerable memory overhead.
4. The IXP2400 supports up to 8 times as much SRAM and 16 times as much SDRAM as the IXP1200. These larger memories would not affect the packet filter but could be important for more compute intensive applications such as packet encryption, load balancing, QoS and traffic shaping.

## 7.0 Related Work

The IXP 1200 has proved to be a very good platform for fast-path network programming. Several other projects have used the IXP 1200 for similar network related projects. In this section we present a very brief overview of four of them.

Software-based routers on the IXP 1200 were explored in a project at Princeton [10]. This project uses input, output and forwarder concepts similar to the ingress, egress and forwarder concepts used in our project. In the Princeton project routing decisions are split between input and forwarder. No other packet examination was done. This project showed that the environment was able to handle gigabit line speeds with minimally sized network packets; the highest load you can expect a router to encounter.

Fairly Fast packet Filters (FFPF) [15] is an open source packet filter solution that has the IXP 1200 as its initial processor platform. FFPF uses “packet flows” for its underlying packet classification scheme. It runs on top of *netfilter* in the Linux kernel. A prototype ran on the IXP 1200.

Snort-to-IXP1200 (S2I) Compiler [16] generates micro-C code for the IXP 1200 microengines given a set of *snort* [17] signatures. The goal is to enhance packet header analysis for network intrusion detection. The snort filters examined were only those that did no payload searches.

Netbind [18] provides an alternative to the MicroACE programming interface. It promotes fast creation of flexible dynamic routing programs for the IXP 1200. The key idea is microengine-binding flexibility. The study showed better performance than the MicroACE

environment with lower binding overhead. No examination of the IXP 2400/2800 processors was attempted.

## 8.0 Summary

The implementation of the packet filter based on Linux iptables was used to investigate and evaluate the programmability and scalability of the IXP1200 network processor. The architecture of the IXP1200 lends itself to the development of efficient multiprocessor applications. The use of the StrongARM for control plane processing and for data plane exception processing make the IXP 1200 very attractive for complex network applications. However, the scalability is limited without moving to the next generation IXP.

The 1K control store limit of the IXP1200 microengines was sufficient to accommodate the basic packet filtering functionality that examines the IP headers of packets in the data plane. However it is limiting if more functionality is desired. This issue and slow inter-microengine communication and inadequate number of microengines for complex network applications have been rectified in the IXP 2400/2800.

Using the iptables packet filter as the underlying application to structure the data plane has proven to be a very good approach. Combining this with the MicroACE framework proved to be ideal for the implementation of a modular packet filter. This modularization along with the availability of the MicroACE Ingress, Egress and Forwarder components made the implementation process easier and faster.

## Acknowledgment

The work on this project was supported by the Consortium for Embedded and Internetworking Technologies grant CRT 9966, "Operating System and Network Software for Embedded Systems." We are also very appreciative of the efforts of Donald B. White and Alan Skousen who set up most of the initial host target workstation configuration and the development environment.

## References

- [1] D. Comer, *Network Systems Design using Network processors*, Pearson Prentice Hall, January 30, 2003
- [2] Intel Corp., *The IXP1200 Network Processor Datasheet*, Dec. 2001, <http://www.intel.com/design/network/datashts/27829810.pdf>
- [3] B. McCarty, *Red Hat Linux Firewalls*, Wiley, Publishing, November 2003

- [4] S. McCanne and Van Jacobson, "The BSD Packet Filter: A New Architecture for User Level Packet Capture," <http://www.usenix.org/publications/library/proceedings/sd93/mccanne.pdf>
- [5] Radisys, *ENP-2505 Hardware Reference*, March 2002. Radisys, May 2002 [http://www.radisys.com/files/support\\_downloads/007-01266-0002.ENP-2505.pdf](http://www.radisys.com/files/support_downloads/007-01266-0002.ENP-2505.pdf)
- [6] Intel Corp., *The IXP1200 Network Processor Microcode Software Reference Manual*, March 2002, [http://developer.intel.com/design/network/manuals/IXP1200\\_prog.pdf](http://developer.intel.com/design/network/manuals/IXP1200_prog.pdf)
- [7] Intel Corp., *Intel Microengine C Compiler Language Support Reference Manual*, March 2002, Intel
- [8] VMware, "VMware – Enterprise class virtualization software", VMware, May 2002 <http://www.vmware.com/>
- [9] Information Sciences Institute, USC, "RFC 791: Internet Protocol, DARPA Internet Program Protocol Specification." <http://www.faqs.org/rfcs/rfc791.html>, September 1981
- [10] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb, "Building a Robust Software-Based Router Using Network Processors," *Proceedings of the 18th SOSP, October 2001*, Chateau Lake Louise, Banff, Alberta, Canada, October 2001.
- [11] Intel Corp., *Intel Internet Exchange Architecture Software Development Kit Guide for SDK 2.0*, December 2001
- [12] Libnet Home Page, <http://libnet.sourceforge.net>
- [13] Intel Corp., *IXP2400 Network Processor*, 2002 <http://www.intel.com/design/network/products/npfamily/ixp2400.htm>
- [14] Pandya, Shymal H., "Implementation of IXP 1200 Network Processor Packet Filtering Software and Parameterization for Higher Performance Network Processors," MS Thesis, Arizona State University, May 2003
- [15] H. Bos and G. Portokalidis, "FFPF: Fairly Fast Packet Filters," <http://ffpf.sf.net/ffpf.pdf>
- [16] I. Cheritakis, D. Pnevmatikatos, E. Markatos, and K. Anagnostakis, "Code Generation for Packet Header intrusion Analysis on the IXP1200 Network Processor," <http://www.ist-scampi.org/publications/papers/charitakis-s2il.pdf>
- [17] J. Beale, *Snort 2.1 Intrusion Detection*, 2<sup>nd</sup> Ed, Caswell (editor), Syngress, May 2004
- [18] A. Cambell, S. Chou, M. Kounavis, V. Stachtos, and J Vicente, "Netbind: A Binding Tool for Constructing Data paths in Network Processor-Based Routers" <http://comet.ctr.columbia.edu/genesis/netbind/overview/netbind.pdf>