PacketBench: A Tool for Workload Characterization of Network Processing

Ramaswamy Ramaswamy and Tilman Wolf
Department of Electrical and Computer Engineering
University of Massachusetts
Amherst, MA 01003
{rramaswa,wolf}@ecs.umass.edu

***Abstract*— Network processing is becoming an increasingly important paradigm as the Internet moves towards an architecture with more complex functionality inside the network. Modern routers not only forward packets, but also process headers and payloads to implement a variety of functions related to security, performance, and customization. It is important to get a detailed understanding of the workloads associated with this processing in order to be able to develop efficient network processing engines. We present a tool called PacketBench, which provides a framework for implementing network processing applications and obtaining an extensive set of workload characteristics. PacketBench provides the support functions to handle various packet traces and manage packet memory. For statistics collection, PacketBench provides the ability to derive a number of microarchitectural and networking related metrics. We present the results of such measurements for four different networking applications ranging from simple packet forwarding to complex packet payload encryption. The results show that such workload analysis has a range of uses from network processor design to application optimization.**

I. Introduction

The Internet has progressed from a simple store-and-forward network to a more complex communication infrastructure. In order to meet demands on security, flexibility, and performance, network traffic not only needs to be forwarded, but also processed inside the network. Such processing is performed on routers, where port processors can be programmed to implement a range of functions from simple packet classification (e.g., for firewalls) to complex payload modifications (e.g., encryption, content adaptation for wireless clients, or ad insertion in web page request).

To handle the functional and performance requirement of the networking domain, routers designs have moved away from the hard-wired ASIC forwarding engines. Instead, software-programmable "network processors" (NPs) have been developed in recent years. These NPs are typically single-chip multiprocessors with high-performance I/O components. A network processor is usually located on a physical port of a router. Packet processing tasks are performed on the network processor before the packets are passed on through the router switching fabric and through the next network link. This is illustrated in Figure 1. It is a current area of research to explore the design space of such NP architectures as well as developing novel protocols and applications that can further benefit from network processing. A crucial step in this process is to understand the workload characteristics of this domain in more detail.

The processing workload on network nodes is unique and particularly different from traditional workstation or server workloads, which are dominated by a few large processing tasks. Network processing is entirely limited to a large number of very simple tasks that operate on small chunks of data (i.e., packets). This implies that many results derived from analyzing workstation or server benchmarks (e.g. SPEC [28] or TPC [32]), are not necessarily applicable to the NP domain. A good example is the memory hierarchy, where smaller on-chip memories suffice due to the nature of packet processing.

In order to explore and understand network processing workloads in more detail, we present a novel tool, called "PacketBench" (a contraction of "packet workbench"). PacketBench provides a programming and simulation environment, where packet processing functions can be implemented easily and quickly. These applications can then be simulated using a variety of real packet traces. The simulation environment is set up to only collect statistics for the packet processing application and not for the supporting PacketBench framework. Thus, numerous workload characteristics that reflect the processing on the network processor can be derived. These include the traditional micro-architectural statistics (as PacketBench uses SimpleScalar [4] for simulating processor cores) as well as statistics that are very specific to the networking environment (e.g., number of memory accesses per packet).

PacketBench is novel in several ways. First, PacketBench applications can be programmed easily with a just a bit of background in networking. Real network processors (which occasionally provide similar system simulators) require in-depth knowledge of the system architecture and are extremely difficult to use. Second, it allows applications to operate on actual packets in the same fashion as it is done inside the network processor. Third, the simulation environment is able to hide the overhead for packet preprocessing in the PacketBench framework (which in real systems is done by specialized hardware components). This provides the basis for realistic program behavior and accurate workload characterization.

The workload statistics that are derived from PacketBench can be used in a number of ways. A few examples are:

- **Application Optimization.** A detailed analysis of the runtime behavior of an application is useful for applica-

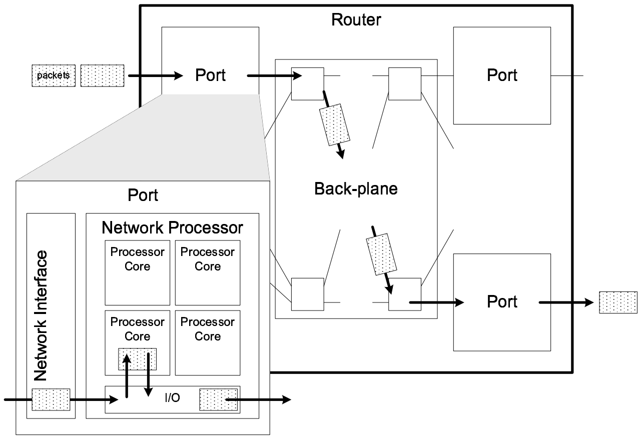


Fig. 1. Network Processing.

tion developers to optimize its performance. Particularly in the NP domain there are many real-time constraints that require a clear understanding of application run-time statistics.

- **Allocation of Processing Tasks.** On a network router, there are several levels of processing resources (data-path processors and co-processors, port control processors, and system control processors as defined similarly in [24]). Processing tasks often can be allocated to any of these levels. Understanding the performance requirements of each task allows system designers make correct choices.
- **Developing Novel NP Architectures.** NP architectures are based on exploiting the inherent packet-level parallelism in the networking domain. Understanding the processing and memory access statistics is important when developing novel designs.

While we are not exploring the actual use of workload statistics in this paper, it does illustrate the usefulness of such information.

The remainder of this paper is organized as follows. Section II discusses related work. We present an overview of PacketBench in Section III. We introduce several sample applications for PacketBench in Section IV. Workload characteristics of these applications are presented and discussed in Section V. A summary and conclusions are presented in Section VI.

II. Related Work

There are numerous examples of processing packets on network nodes that extend the basic packet forwarding paradigm. Routers can perform firewalling [19], network address translation (NAT) [10], web switching [1], IP traceback [27], and many other functions. With increasingly heterogeneous end-systems (e.g., mobile devices and "thin" clients), computationally demanding services have been moved into the network. Examples for these are content transcoding, advertisement insertion, and cryptographic processing. It can be expected that this trend towards more functionality on the router continues. A few programmable network platforms that have been developed by the active networking community [30], [5], [23] have reached the necessary level of maturity. Also, there are standardization efforts in progress by the IETF OPES (Open Pluggable Edge Services) working group [15] to define such networking service platforms.

In practice, these processing functions can be implemented in a variety of ways, ranging from software-based routers (workstation acting as a specialized router) to specialized hardware (ASIC implementation on router line card). In recent years, programmable network processors have become available for performing general-purpose processing on high-bandwidth data links. These network processors are system-on-a-chip multiprocessors that are optimized for high-bandwidth I/O and highly parallel processing of packets. A few of the numerous examples are the Intel IXP1200 [16], IBM PowerNP [14], and EZchip NP-1 [11]. The design spaces of such network processors has been explored in several ways. Crowley *et al.* have evaluated different processor architectures for their performance under networking workloads [8]. This work mostly focuses on the tradeoffs between RISC, superscalar, and multithreaded architectures. In more recent work, a modelling framework is proposed that considers the data flow through the system [6]. Thiele *et al.* have proposed a performance model for network processors [31] that considers the effects of the queueing system. In our previous work, we have developing a quantitative performance and power consumption model for a range of design parameters [12] [13]. All these models require detailed workload parameters to obtain realistic results.

Several network processor benchmarks have captured various characteristics of network processing. Crowley *et al.* has defined simple programmable network interface workloads in [7]. In our previous work, we have defined a network processor benchmark called CommBench [33]. Memik *et al.* have proposed a similar benchmark more recently [18]. All these benchmarks are useful in that they define a realistic set of applications, but are limited in how much detail of workload characteristics can be derived. We try to address this shortcoming with PacketBench. PacketBench is a novel tool that allows a very detailed and network-processor-specific analysis of such benchmark applications that goes beyond simple microarchitectural metrics.

III. PacketBench

PacketBench is a tool with which packet processing applications can easily be implemented. It provides the support functions to read and write packets from and to packet traces, manage packet memory, and implement a simple API. The details of PacketBench are discussed in more detail in this section.

A. PacketBench System

The goal of PacketBench is to emulate the functionality of a network processor. The conceptual outline of the tool is shown in Figure 2. The main components are:

- **PacketBench Framework.** The framework provides functions that are necessary to read and write packets,

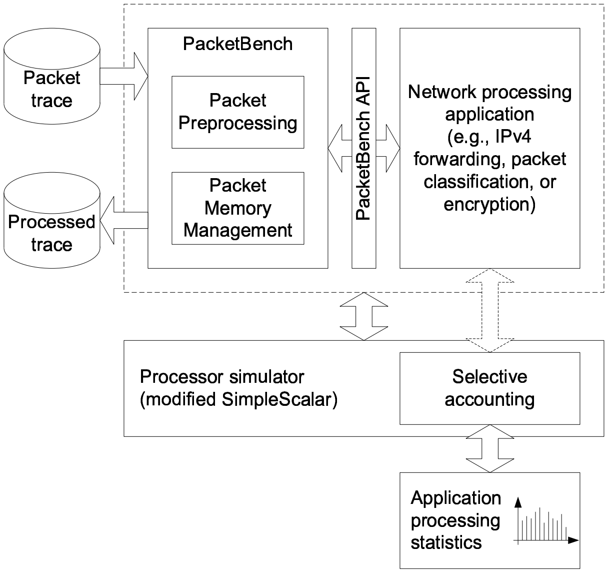

Packet trace
Processed trace
PacketBench
Packet Preprocessing
Packet Memory Management
PacketBench API
Network processing application (e.g., IPv4 forwarding, packet classification, or encryption)
Processor simulator (modified SimpleScalar)
Selective accounting
Application processing statistics

Fig. 2. PacketBench Architecture.

and manage memory. This involves reading and writing trace files and placing packets into the memory data structures used internally by PacketBench. On a network processor, many of these functions are implemented by specialized hardware components and therefore should not be considered part of the application.

- **PacketBench API.** PacketBench provides an interface for applications to receive, send, or drop packets as well as doing other high-level operations. Using this clearly defined interface makes it possible to distinguish between PacketBench and application operations during simulation.
- **Network Processing Application.** The application implements the actual processing of the packets. This can range from simple forwarding to complex payload processing. Several such applications are discussed in Section IV. The workload characteristics of these applications are most relevant and need to be collected separately from workload generated by the PacketBench framework.
- **Processor Simulator.** To get instruction-level workload statistics, we use a full processor simulator. In our current prototype we use SimpleScalar [4], but in principle any processor simulator could be used. Since we want to limit the workload statistics to the application and not the framework, we modified the simulator to distinguish operations accordingly. The Selective Accounting component does that and thereby generates workload statistics as if the application had run by itself on the processor. This corresponds to the actual operation of a network processor, where the application runs by itself on one of the processor cores. Additionally, it is possible to distinguish between access to various types of memory (instruction, packet data, and application state, see Section V).

The key point about this system design is that the application and the framework can be clearly distinguished – even though both components need to be compiled into a single executable in order to be simulated. This is done by analyzing the instruction addresses and sequence of API calls. This separation allows us to adjust the simulator to generate statistics for the application processing and ignore the framework functions. Another key benefit of PacketBench is the ease of implementing new applications. The architecture is modular and the interface between the application and the framework is well defined. New applications can be developed in C, plugged into the framework, and run on the simulator to obtain processing characteristics.

B. PacketBench API

The PacketBench API defines how applications can receive, send, or drop packets. PacketBench makes no restrictions on the application other than that it needs to adhere to the API. The three main functions that are defined in the API are:

- **void *init()** – This function is implemented by the application and called by the framework before any packets are processed. It allows the application to initialize any data structures that are required for packet processing (e.g., routing table). The processing that occurs as part of *init()* is not counted towards packet processing.
- **void (*process_packet_function)(packet *)** – This function is the packet handler that is implemented by the application. The variable *process_packet_function* contains the pointer to the actual packet handler. It is called once for each packet that is processed by the framework. A pointer to the packet is passed as an argument. The packet memory is managed by the framework. The packet processing function has access to the contents of the packet from the layer 3 header onwards. This function call uses a function pointer to allow the application to specify the packet processing function dynamically.
- **void write_packet_to_file(packet *, int)** – This function is implemented by the framework and called by the application when processing is complete. It writes the packet to the trace file (specified by the second parameter).

Similar to *write_packet_to_file()* there is a function to drop the packet. We are planning on extending the API further to implement an interface to packet scheduling and other control-level operations.

C. PacketBench Prototype

PacketBench is simulated on a typical processor simulator to obtain processing statistics such as the number of instructions executed and the number of memory accesses made. For this purpose, we use the ARM [2] target of the SimpleScalar [4] simulator (obtained from [26]), to analyze our applications. We chose this simulator because the ARM architecture is very similar to the architecture of the core processor and the microengines found in the Intel IXP 1200 network processor [16]. Also, SimpleScalar is freely available with source code and can easily be modified. The tools were setup to work on an

Intel x86 workstation running RedHat Linux 7.3. PacketBench supports packet traces in the *tcpdump* [29] format and the Time Sequenced Header (TSH) format from NLANR [20].

The PacketBench application and framework are compiled with a cross compilation toolchain for the GNU C compiler and an ARM backend. The executable is run on SimpleScalar (*sim-profile*) and the simulator output is analyzed. SimpleScalar was augmented to evaluate the address of each simulated instruction and check if it is in the address range of the application or the framework. We can obtain these address ranges easily from the compiled object code with *nm* and *objdump*. These addresses are then passed to *sim-profile* as input parameters. The simulator was further modified to provide more detailed statistics about program execution, such as cumulative instruction counts for the packet processing function, and the number of memory accesses made to a particular variable in the packet handler. The output of the simulator is post-processed using *perl* scripts to obtain the required results.

IV. Example Workload

We illustrate the capabilities of PacketBench by using it with an example set of application and network traces. The results of the workload evaluation are presented in Section V.

A. Applications

We have chosen four applications for gathering workload statistics using PacketBench. Two of these applications are IP forwarding according to current Internet standards using two different implementations for the routing table lookup. The third application implements packet classification, which is commonly used in firewalls and monitoring systems. And the final application implements encryption, which is a function that actually modifies the entire packet payload. The specific applications are:

- **IPv4-radix.** IPv4-radix is an application that performs RFC1812-compliant packet forwarding [3] and uses a radix tree structure to store entries of the routing table. The routing table is accessed to find the interface to which the packet must be sent, depending on its destination IP address. The radix tree data structure is based on an implementation in the BSD operating system [21].
- **IPv4-trie.** IPv4-trie is similar to IPv4-radix and also performs RFC1812-based packet forwarding. This implementation is derived from an implementation for the Intel IXP1200 network processor [16]. This application uses a multibit trie data structure to store the routing table, which is more efficient in terms of storage space and lookup complexity.
- **Flow Classification.** Flow classification is a common part of various applications such as firewalling, NAT, and network monitoring. The packets passing through the network processor are classified into flows which are defined by a 5-tuple consisting of the IP source and destination addresses, source and destination port numbers, and transport protocol identifier. The 5-tuple is used to compute a hash index into a hash data structure that uses link lists to resolve collisions.
- **IPSec Encryption.** IPSec is an implementation of the IP Security Protocol [17], where the packet payload is encrypted using the Rijndael Advanced Encryption Standard (AES) algorithm [9]. This is the only application where the packet payload is read and modified.

Trace Name	Type	Packets
MRA	OC-12c (PoS)	4,643,333
COS	OC-3c (ATM)	2,183,310
ODU	OC-3c (ATM)	784,278
LAN	100Mbps (Ethernet)	100,000

TABLE I

Packet Traces Used to Evaluate Applications.

This selection of applications cover a broad space of typical network processing. First of all, IPv4-radix and IPv4-trie are realistic, full-fledged packet forwarding applications, which perform all required IP forwarding steps (header checksum verification, decrementing TTL, etc.). IPv4-radix represents a straight-forward unoptimized implementation, while IPv4-trie is more efficient. Second, the applications can be distinguished between header processing applications (HPA) and payload processing applications (as defined in [33]). HPA process a limited amount of data in the packet headers and their processing requirements are independent of packet size. PPA perform computations over the payload portion of the packet and are therefore more demanding in terms of computational power as well as memory bandwidth. IPSec is a payload processing applications and the others are header processing applications. Third, the applications vary significantly in the amount of data memory that is required. Encryption only needs to store a key and small amounts of state, but the routing tables of the IP forwarding applications are very large.

Altogether, the four applications chosen in this work give are good representatives of different types of network processing. They display a variety of workload characteristics as the results in Section V show.

B. Network Traces and Routing Tables

To characterize workloads accurately, it is important to have realistic packet traces that are representative of the traffic that would occur in a real network. Table I shows the packet traces that we used to evaluate the applications. Traces MRA, COS, and ODU are obtained from the NLANR repository [20] and were collected on different access and backbone links. The LAN trace was collected on our local intranet.

Due to the nature of the NLANR TSH trace format, the packet payload is not available. Since the IPSec application requires packet payload to be present, a dummy payload was inserted in the packet preprocessing step. Since the AES algorithm is not data-dependent, the workload statistics will be the same as for "real" packet payloads.

Another limitation of the NLANR traces is the way IP addresses are anonymized. To provide privacy, IP addresses are numbered incrementally starting at 10.0.0.1 in the order of their occurrence. This leads to a non-uniform coverage of destination addresses in the address space. As a result, lookups into typical routing tables (e.g., MAE-WEST [22] as we use for IPv4-radix) lead almost always to the same prefix. To avoid this bias, we scrambled the IP address in the packet preprocessing stage to achieve more uniform coverage.

V. Results

There are a number of workload characteristics that can be derived from PacketBench. In general, there are basically two classes of results that can be derived:

- **Microarchitectural Results.** Most processor simulators provide a range of statistics that are related to the simulated processor core. Examples are instruction mix, branch misprediction rates, and instruction-level parallelism.
- **Network Processing Results.** In the context of network processing there are a number of statistics that can be gathered, which combine microarchitectural metrics (e.g., instruction count and memory bandwidth) with packet metrics (e.g., packet size). This leads to novel metrics that are specific to the network processing environment (e.g., packet processing complexity and packet memory access pattern).

Microarchitectural results for network processing have been covered in our own previous work [33] as well as by other related work [7] [18]. Gathering similar workload characteristics is a straightforward exercise and we are not considering it further in this work. Instead we are exploring novel network processing statistics. These statistics include:

- Processing Complexity
- Memory Access Rates and Patterns
- Memory Coverage
- Instruction Grouping

Processing complexity puts the number of instructions executed per packet in context with different applications and packet sizes. The memory access and coverage statistics distinguish not only between instruction and data memory, but further separate packet data and program data. This is an important distinction as packet data is handled differently in network processors. Such a distinction has not been made in previous work. Finally, instruction grouping is a useful tool to identify instruction blocks that form semantic entities. For network processor design, it is important to identify processing intensive instruction groups as they can be implemented on co-processors or on programmable hardware.

The results of each network processing metric are presented and discussed in the following subsections.

A. Processing Complexity

The average number of instructions executed per packet can be expressed as the *application complexity* as we have defined in our previous work [33]. We show the processing complexity for the various applications discussed in the previous section in Table II. The number of instructions executed per packet for encryption (IPSec in column 5) is several orders of magnitude higher than the number of instructions required for the other applications because encryption is a payload processing application and processes the entire packet. The variation in instruction counts between different traces can be attributed to the fact that payload lengths across different traces are not uniform. Additionally, the following observations can be made:

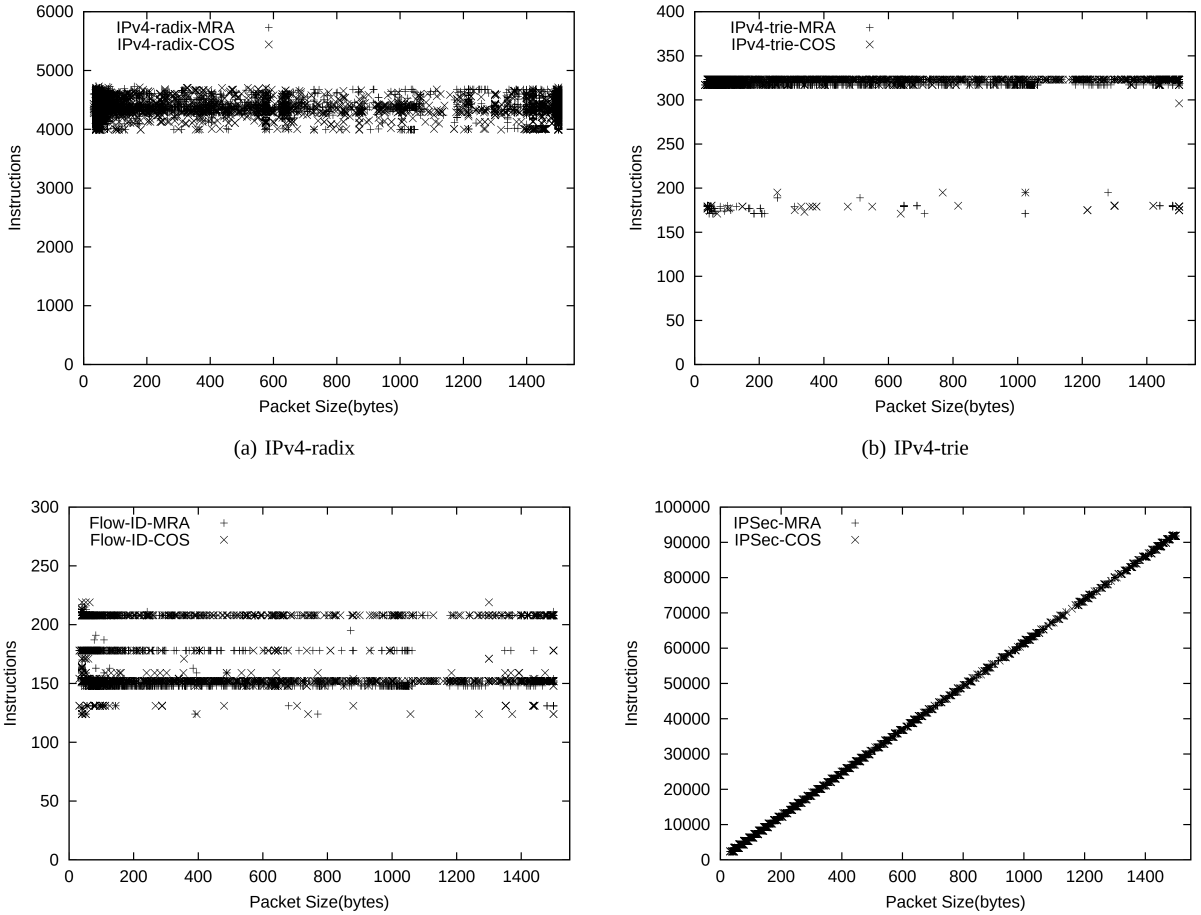
Trace Name	IPv4-radix	IPv4-trie	Flow Classification	IPSec
MRA	4,438	320	139	43,470
COS	4,388	321	138	38,938
ODU	4,324	320	140	32,839
LAN	4,820	437	135	34,417
Average	4,493	350	138	37,416

TABLE II

Average Number of Instructions per Packet Executed for Various Applications.

- For IPv4 forwarding (IPv4-radix), the number of instructions can vary depending on the destination address of the packet (which may be at different locations in the routing table). This behavior is not exhibited by IPv4-trie due to the fact that we use a very small routing table.
- IPv4-radix forwarding requires more instructions to execute than IPv4-trie forwarding. Most of the instruction difference can be attributed to the overhead of maintaining and traversing the radix tree. Moreover the implementation of IPv4-radix is a not particularly optimized as compared to IPv4-trie.
- Flow classification is a strictly linear (i.e., not data-dependent) application which takes approximately the same amount of instructions irrespective of the trace.
- In column 3 of Table II packets of the Internet traces (MRA, COS and ODU) have destination IP addresses that match a different entry in the routing table than the destination IP addresses of the LAN trace packets. This accounts for the difference in instruction counts between the traces.

To illustrate the impact of packet size, the number of instructions executed per packet against packet size is plotted in Figure 3 for two traces (MRA and COS) and for the four applications discussed previously. For IPSec encryption, the number of instructions executed increases linearly with packet size. For the three header processing applications, the number of instructions executed remains more or less constant over all packet sizes. Any change in the number of instructions is due to data-dependent operations such as accessing the routing table for IPv4 forwarding, or updating the flow table for flow classification. For example, the packets in the IPv4-radix graph that require 4,800 instructions represent those whose destination IP prefixes are longer than those packets which require 4,000 instructions. As a result the routing table has to be searched deeper, which causes the extra operations.



(a) IPv4-radix

(b) IPv4-trie

(c) Flow Classification

(d) Encryption

Fig. 3. Processing Complexity over a Range of Packet Sizes.

B. Memory Characteristics

As discussed above, PacketBench can distinguish between different memory regions which are in the same address space but are semantically different. For the memory statistics, we distinguish between instructions, packet data, and program data (i.e., program state).

1) Memory Accesses: We have analyzed the four test applications in terms of accesses to packet memory (which contains the packet header and the payload), and accesses to non-packet memory. The analysis was performed for the first 10,000 packets of each trace. The results are summarized in Table III. The following observations can be made:

- Header processing applications require fewer accesses to packet and non-packet memory than payload processing applications. Moreover, for a given application, header processing requires about the same number of accesses to both packet and non-packet memory for all the traces.
- Payload processing requires a lot of memory accesses (both packet and non-packet) since the entire payload is encrypted (and hence has to be read from memory). The high number of non-packet memory accesses is due to a large amount of state being manipulated. Variations in the number of memory accesses is due to variations in packet payload size.

For IPSec, the number of memory accesses exhibits the same linear relationship with increasing packet size as for instruction complexity (not shown).

2) Memory Coverage: We have estimated the size of the active memory regions for both instructions and data by post-processing the instruction and data traces returned by SimpleScalar. The analysis was performed on the first 1,000 packets of the MRA trace. The results are shown in Table IV. A comparison of Table II and Table IV indicates that smaller instruction memory sizes implies a small program core which is executed several times. For example, in IPSec encryption, the average number of instructions per packet is 43,470, but the instruction memory size is only 2,800 bytes (approximately 700 instructions, assuming 32 bit instructions). Similar observations can be made for IPv4-radix and IPv4-trie forwarding. This reemphasizes one key characteristic of network processing, which is the simplicity of the code executed on network nodes. As a result, network processors can operate efficiently with instruction stores of only a few

Trace Name	IPv4-radix		IPv4-trie		Flow classification		IPSec	
	Packet	Non-packet	Packet	Non-packet	Packet	Non-packet	Packet	Non-packet
MRA	32	842	12	71	23	58	841	23,844
COS	32	836	12	71	24	60	828	23,447
ODU	32	833	12	71	23	54	725	20,427
LAN	32	831	17	98	23	51	571	15,926
Average	32	836	13	78	23	56	741	20,911

TABLE III

Average Number of Accesses to Packet Memory and Non-Packet Memory for Various Applications (10,000 Packets).

Application	Instr. memory size	Data memory size
IPv4-radix	4,420	18,004
IPv4-trie	848	264
Flow classification	1,584	43,344
IPSec encryption	2,800	9,428

TABLE IV

Instruction and Data Memory Sizes(in bytes) for Various Applications.

kilobytes.

In terms of data memory (counting both packet and program data), flow classification has the largest data memory size since a lot of state is maintained and manipulated on a per packet basis. Similar conclusions can be drawn for IPv4-radix forwarding. The results in column 3 of Table IV supports the general trend that header processing applications are dominated by the data stored in the various routing and lookup data structures.

C. *Detailed Packet Processing*

In addition to the cumulative analysis performed in the previous sections, we have analyzed program behavior while focusing on a single packet. This gives us a more detailed insight on the different processing steps that occur for one packet. We chose a packet from the MRA trace such that the number of instructions required to process the packet are as close as possible to the average values shown in Table II.

1) Memory Requests: Figure 4 shows the memory access patterns for the program while processing a packet. Reads and writes to packet memory are plotted on the positive y-axis while accesses to non-packet memory are plotted on the negative y-axis.

Both IPv4-trie and IPv4-radix forwarding have similar memory access patterns where the packet header contents are accessed initially to extract the destination IP address, decrement the time-to-live (TTL) and update the header checksum. The remainder of the memory accesses are to the routing table (which is non-packet memory, hence the points on the negative y-axis) to lookup the interface for the destination IP addresses. Flow classification consists of a set of initial memory accesses to the packet header to determine which flow the packet belongs to. The next set of memory accesses are to non-packet memory during which the flow record for that particular flow is obtained. The final set of memory accesses are to both packet and non-packet memory which involves updating the flow record (non-packet memory) with the values from the packet header (packet memory). For IPSec encryption, distinct blocks of non-packet memory accesses can be seen. These correspond to encryption processing. The packet accesses before and after represent reading and writing the payload from and to the packet memory.

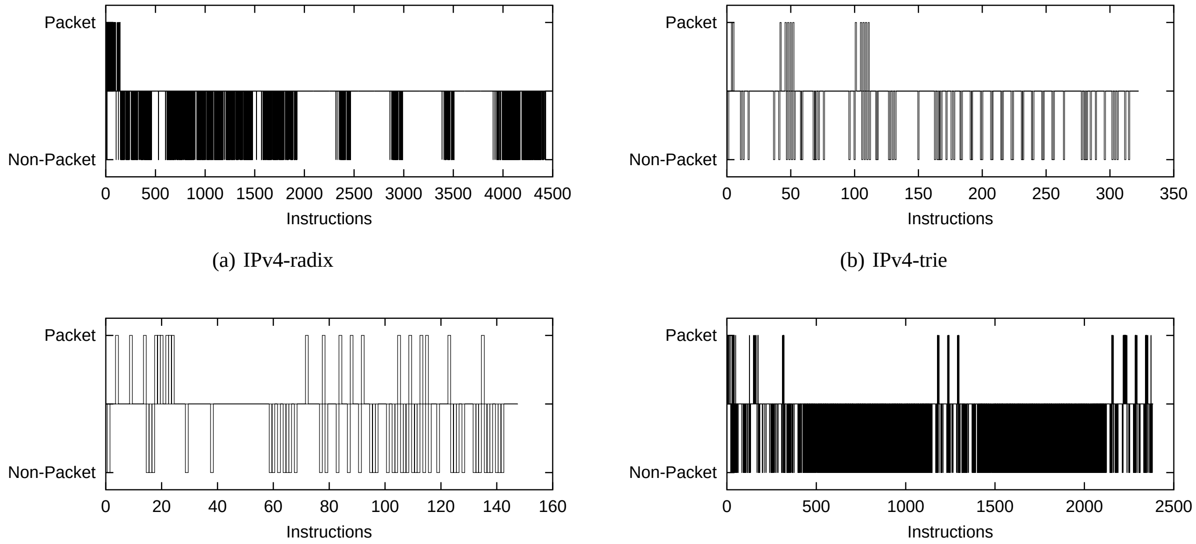
2) Instruction Grouping: Finally, we show the instruction access patterns for the program while processing a packet. In order to view the instruction patterns more clearly, the instruction addresses were assigned a unique index depending on the order in which they were executed. Results are shown in Figure 5. Overlaps of the graph on the y-axis represent sequences of instructions which are repeatedly executed (such as loops). Flow classification exhibits very linear program behavior with very little repetition of instructions. The other 3 applications show very regular patterns in which the instructions are accessed. For example, in IPSec encryption, a block of instructions approximately 70 instructions long is executed 8 times between instructions 500 and 1,000 and instructions 1,500 and 2,000.

D. *Impact of Results*

The results shown above give some interesting insights into packet processing workloads. While the set of applications used does not consider all possible types of packet processing, a good coverage of basic applications is achieved.

The workload characteristics can be used in many ways. We are discussing here a few usages that pertain to our current research. Using the processing complexity and memory access characteristics, it is possible to derive an analytic model to estimate the processing delay of a packet given an application. This is useful in the context of network simulations, where processing delay is currently not or only superficially considered. Preliminary results can be found in [25].

The detailed instruction grouping is useful to identify semantic groups of instructions. In network processors, it is desirable to speed up operations with co-processors. It is however difficult to generally identify operations that are used frequently enough to justify the hardware expense of a co-processor. Sets of instructions that are repeatedly executed can easily be identified with the instruction grouping plots shown in Figure 5. For example, IPSec clearly shows such a sequence of repeated operations.



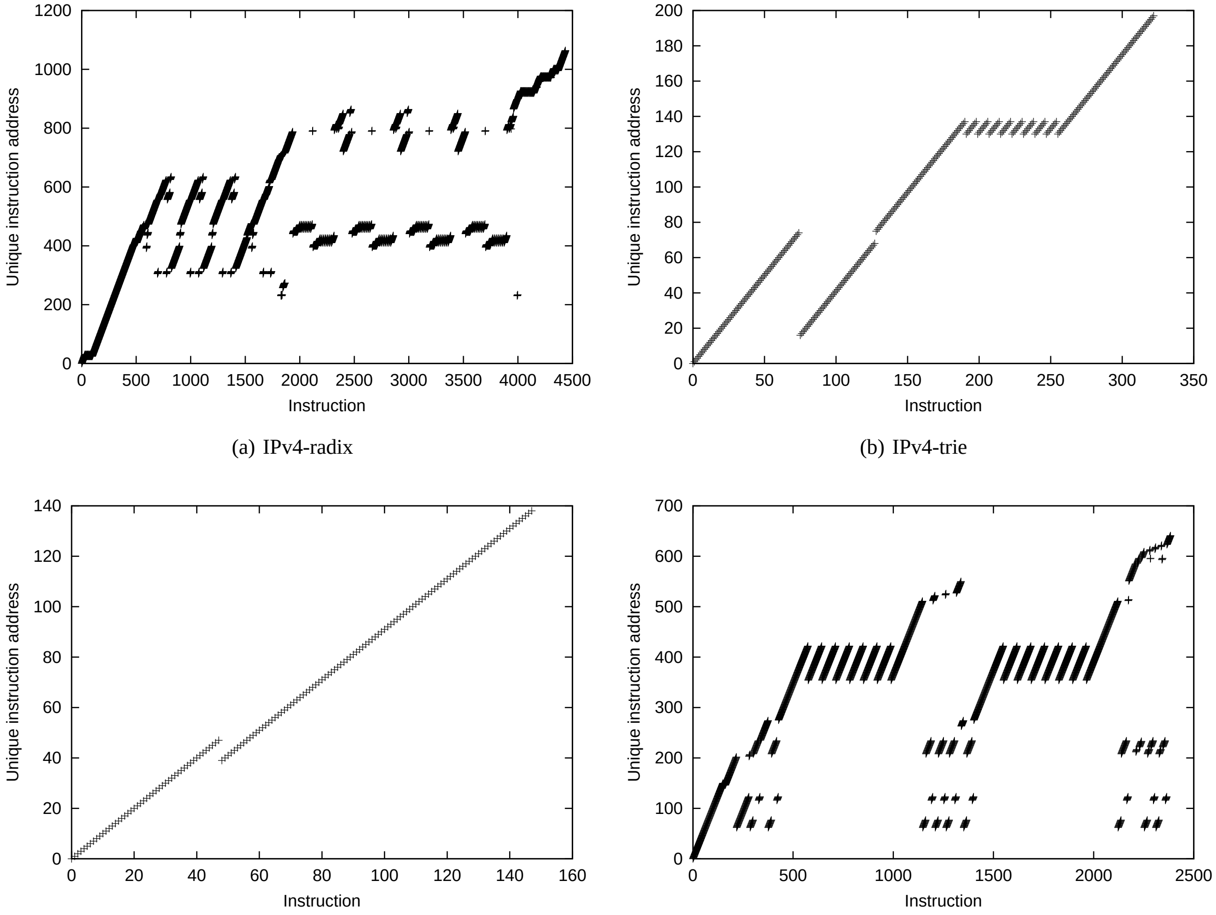
(a) IPv4-radix

(b) IPv4-trie

(c) Flow Classification

(d) Encryption

Fig. 4. Detailed Memory Access Patterns for a Single Packet from the MRA Trace.



(a) IPv4-radix

(b) IPv4-trie

(c) Flow Classification

(d) Encryption

Fig. 5. Detailed Instruction Access Patterns for a Single Packet from the MRA Trace.

Finally, the results are useful for the workload characteristics themselves. We have developed several analytic performance models for network processors [12] [13], which all rely on accurate workload parameters. PacketBench can provide such statistics and also allows to generate them easily for new applications.

VI. Conclusion

In this paper, we have presented PacketBench, a tool for analyzing network processing workloads. PacketBench provides a simple platform for developing network processing applications and simulating them in a realistic way using real packet traces. We present results for four different networking applications – ranging from simple IP forwarding to complex encryption processing.

The workload characteristics derived with PacketBench focus mostly on novel, packet-processing related characteristics. In particular, we are able to combine microarchitectural statistics (e.g., instruction count) with networking metrics (e.g., packet size) to gain a better understanding of network processing workloads.

References

[1] G. Apostolopoulos, D. Aubespin, V. Peris, P. Pradhan, and D. Saha. Design, implementation and performance of a content-based switch. In *Proc. of IEEE INFOCOM 2000*, Tel Aviv, Israel, Mar. 2000.

[2] ARM Ltd. *ARM7 Datasheet*, 2003.

[3] F. Baker. Requirements for IP version 4 routers. RFC 1812, Network Working Group, June 1995.

[4] D. Burger and T. Austin. The SimpleScalar tool set version 2.0. *Computer Architecture News*, 25(3):13–25, June 1997.

[5] A. T. Campbell, H. G. De Meer, M. E. Kounavis, K. Miki, J. B. Vincente, and D. Villela. A survey of programmable networks. *Computer Communication Review*, 29(2):7–23, Apr. 1999.

[6] P. Crowley and J.-L. Baer. A modelling framework for network processor systems. In *Network Processor Workshop in conjunction with Eighth International Symposium on High Performance Computer Architecture (HPCA-8)*, pages 86–96, Cambridge, MA, Feb. 2002.

[7] P. Crowley, M. E. Fiuczynski, J.-L. Baer, and B. N. Bershad. Workloads for programmable network interfaces. In *IEEE Second Annual Workshop on Workload Characterization*, Austin, TX, Oct. 1999.

[8] P. Crowley, M. E. Fiuczynski, J.-L. Baer, and B. N. Bershad. Characterizing processor architectures for programmable network interfaces. In *Proc. of 2000 International Conference on Supercomputing*, pages 54–65, Santa Fe, NM, May 2000.

[9] J. Daemen and V. Rijmen. The block cipher Rijndael. In *Lecture Notes in Computer Science*, volume 1820, pages 288–296. Springer-Verlag, 2000.

[10] K. B. Egevang and P. Francis. The IP network address translator (NAT). RFC 1631, Network Working Group, May 1994.

[11] EZchip Technologies Ltd., Yokneam, Israel. *NP-1 10-Gigabit 7-Layer Network Processor*, 2002. http://www.ezchip.com/html/pr_np-1.html.

[12] M. A. Franklin and T. Wolf. A network processor performance and design model with benchmark parameterization. In P. Crowley, M. A. Franklin, H. Hadimioglu, and P. Z. Onufryk, editors, *Network Processor Design: Issues and Practices I*, chapter 6, pages 117–138. Morgan Kaufmann Publishers, Oct. 2002.

[13] M. A. Franklin and T. Wolf. Power considerations in network processor design. In *Network Processor Workshop in conjunction with Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*, pages 10–22, Anaheim, CA, Feb. 2003.

[14] IBM Corp. *IBM Power Network Processors*, 2000. http://www.chips.ibm.com/products/wired/communications/network_processors.html.

[15] IETF. *Open Pluggable Edge Services*, 2003. http://www.ietf-opes.org/.

[16] Intel Corp. *Intel IXP1200 Network Processor*, 2000. http://www.intel.com/design/network/products/npfamily/ixp1200.htm.

[17] S. Kent and R. Atkinson. Security architecture for the internet protocol. RFC 2401, Network Working Group, Nov. 1998.

[18] G. Memik, W. H. Mangione-Smith, and W. Hu. NetBench: A benchmarking suite for network processors. In *Proc. of International Conference on Computer-Aided Design*, San Jose, CA, Nov. 2001.

[19] J. C. Mogul. Simple and flexible datagram access controls for UNIX-based gateways. In *USENIX Conference Proceedings*, pages 203–221, Baltimore, MD, June 1989.

[20] National Laboratory for Applied Network Research - Passive Measurement and Analysis. *Passive Measurement and Analysis*, 2003. http://pma.nlanr.net/PMA/.

[21] NetBSD Project. *NetBSD release 1.3.1*. http://www.netbsd.org/.

[22] Network Processor Forum. *Benchmarking Implementation Agreements*, 2003. http://www.npforum.org/benchmarking/bia.shtml.

[23] K. Psounis. Active networks: Applications, security, safety, and architectures. *IEEE Communications Surveys*, 2(1), Q1 1999.

[24] X. Qie, A. Bavier, L. Peterson, and S. Karlin. Scheduling computations on a software-based router. In *Proc. IEEE Joint International Conference on Measurement & Modeling of Computer Systems (SIGMETRICS)*, pages 13–24, Cambridge, MA, June 2001.

[25] R. Ramaswamy, N. Weng, and T. Wolf. Considering processing cost in network simulations. In *Proc. of Workshop on Models, Methods and Tools for Reproducible Network Research (MoMeTools) in conjunction with ACM SIGCOMM*, Karlsruhe, Germany, Aug. 2003.

[26] SimpleScalar LLC. *http://www.simplescalar.com*, 2003.

[27] A. S. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, S. T. Kent, and W. T. Strayer. Hash-based ip traceback. In *Proc. of ACM SIGCOMM 2001*, pages 3–14, San Diego, CA, Aug. 2001.

[28] Standard Performance Evaluation Corporation. *SPEC CPU2000 - Version 1.2*, Dec. 2001.

[29] TCPDUMP Repository. *http://www.tcpdump.org*, 2003.

[30] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, Jan. 1997.

[31] L. Thiele, S. Chakraborty, M. Gries, and S. Künzli. Design space exploration of network processor architectures. In *Network Processor Workshop in conjunction with Eighth International Symposium on High Performance Computer Architecture (HPCA-8)*, pages 30–41, Cambridge, MA, Feb. 2002.

[32] Transaction Processing Performance Council. *TPC Benchmark C, Revision 5.1*, Dec. 2002.

[33] T. Wolf and M. A. Franklin. CommBench - a telecommunications benchmark for network processors. In *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 154–162, Austin, TX, Apr. 2000.