Analysis of Network Processing Workloads

Ramaswamy Ramaswamy, Ning Weng, and Tilman Wolf
Department of Electrical and Computer Engineering
University of Massachusetts
Amherst, MA 01003
{rramaswa,nweng,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. The understanding of workload details of network processing has many practical applications. As network processing systems move towards highly parallel embedded systems, it is becoming increasingly important to explore the processing requirements of individual packets rather than averaged statistics. We show a range of workload results that focus on individual packets and the variation between them.**

Keywords: network processors, workload characterization, network processing simulator.

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 constantly varying functional requirements of the networking domain, router designs have moved away from 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. They contain several simple processor cores which are optimized for handling packets along with a control processor, which handles higher level functions. 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. Design space exploration of NP architectures, development of novel protocols and network processing applications, and the creation of suitable programming abstractions for such parallel embedded systems are current areas of research. Therefore it is crucial to understand the workload characteristics of this domain in more detail.

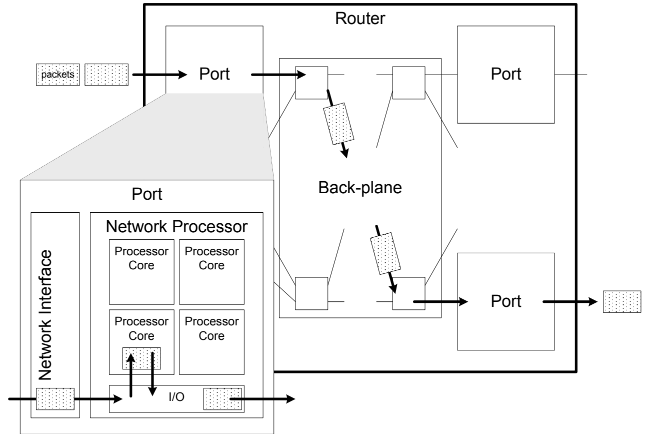


Fig. 1. Router System with Network Processor. Packets are processed by one of multiple processing cores in the network processor.

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 [1]), 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 [2] for simulating processor cores) as well as statistics that are very specific to the networking environment (e.g., number of memory accesses per packet).

Most importantly, PacketBench allows the collection of workload information on a per-packet basis. Rather than examining averaged metrics, we can explore the detailed processing of each packet and explore the differences between individual packets. This is important for network processing environments as most packets exhibit very similar processing demands, but special cases need to be processed on the control processor of the router system.

Compared to other network processing simulators and benchmarks, 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. We do not wish to characterize this processing since it is handled by specialized hardware components in real systems. 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 application 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 [3]). Processing tasks 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.
- **Understanding the Dynamics of Network Processing.** The simplicity and regularity of network processing applications has lead to NP solutions that utilize numerous simple processing cores. Due to a lack of operating system support on these processing engines, applications are typically fine tuned off-line for a given system. Deviations of the workload from the expected behavior can lead to performance degradation in these systems. By comparing the execution path of different packets on the same application, we can develop a weighted flow graph that illustrates the dynamics of packet processing.

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 [4], network address translation (NAT) [5], web switching [6], IP traceback [7], 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.

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 examples are the Intel IXP2400 [8], and EZchip NP-1 [9]. 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 [10]. This work mostly focuses on the tradeoffs between RISC, superscalar, and multithreaded architectures. In more recent work, a modeling framework is proposed that considers the data flow through the system [11]. Thiele *et al.* have proposed a performance model for network processors [12] that considers the effects of the queueing system. In our previous work, we have developed a quantitative performance and power consumption model for a range of design parameters [13] [14]. 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 [15]. In our previous work, we have defined a network processor benchmark called CommBench [16]. Memik *et al.* have proposed a similar benchmark more recently [17]. Lee and John have extended these benchmarks to also consider control plane applications [18]. A commercial benchmark that contains typical networking workloads is available from [19]. All these benchmarks are useful since they define a realistic set of applications, but are limited in the detail of workload characteristics which can be derived. We try to address this

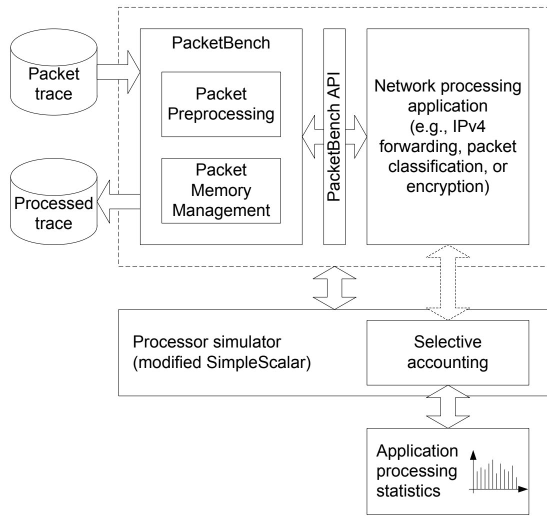


Fig. 2. PacketBench Architecture. The selective accounting component ensures that only statistics from the application are collected. Instructions that are executed as part of the PacketBench framework are ignored.

shortcoming with PacketBench. PacketBench allows a very detailed and network-processor-specific analysis of such benchmark applications that goes beyond simple microarchitectural metrics and yields information on the processing steps for each individual packet.

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, 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 [2], 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 accesses 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. 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 processing function has access to the contents of the packet from the layer 3 header onwards.
- **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).

C. PacketBench Prototype

PacketBench is simulated on a typical processor simulator to obtain processing statistics. For this purpose, we use the ARM [20] target of the SimpleScalar [2] simulator 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 2400 network processor [8]. 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 9.0. PacketBench supports packet traces in the *tcpdump* [21] format and the Time Sequenced Header (TSH) format from NLANR [22]. PacketBench generates a very small overhead and does not significantly reduce the performance of SimpleScalar.

IV. Application 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. Network Processing 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. The fourth implements prefix-preserving IP address anonymization, which is a common function in network measurement and monitoring. The specific applications are:

- **IPv4-radix.** IPv4-radix is an application that performs RFC1812-compliant packet forwarding [23] 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 [24].
- **IPv4-trie.** IPv4-trie is similar to IPv4-radix and also performs RFC1812-based packet forwarding and was derived from [25]. This application uses a trie data structure with both path and level compression 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.
- **TSA.** Top-hashed subtree-replicated anonymization (TSA) [26] is an algorithm to scramble IP addresses in network traces to ensure the privacy of users and is a high-speed optimization to prefix-preserving anonymization [27]. In addition to anonymizing the IP addresses, layer 3 and layer 4 headers are collected for each packet that is encountered in the trace.

TABLE I

PACKET TRACES USED TO EVALUATE APPLICATIONS.

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

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 vary significantly in the amount of data memory that is required. The routing tables for IP forwarding and the TSA data structures are large, but fixed size. The memory requirements for Flow Classification change with the number of flows in the trace. Finally, the processing complexity of these applications range from simple data structure lookups to more complex computations.

Altogether, the four applications chosen in this work are good representatives of different types of network processing. They display a variety of workload characteristics as the results in Section V show. We have also used PacketBench for the analysis of payload processing applications (PPA, as defined in [16]). In this work, we focus on header processing applications (HPA), but it is important to note that PacketBench can be used to analyze both types of applications.

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 [22] and were collected on different access and backbone links. The LAN trace was collected on our local intranet.

We have run our experiments using traces of 1,000 to 100,000 packets extracted from the traces shown in Table I. While very large traces are important for conventional workstation benchmarks, network processing is dominated by small repetitive tasks. Even with traces of only a few thousand packets, almost all possible execution paths of the application are considered (as shown in Section V). For this reason, the results that we have presented based on these traces can be considered representative.

To provide privacy, IP addresses in the NLANR traces 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 [28] which 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 generated with PacketBench. In general, there are three 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).
- **Per-Packet Analysis.** The differences in the execution path of packets highlights the dynamic variations in network processing. While most packets follow the same processing steps, it is important to identify the percentages of other cases in order to decide how to implement them most efficiently.

Microarchitectural results for network processing have been covered in our own previous work [16] as well as by other related work [15] [17] [18]. Gathering similar workload characteristics is a straightforward exercise and is not considered further (although they can be obtained from PacketBench). Instead, we explore novel network processing statistics. These statistics are divided into three categories:

- Averaged statistics
- Variation across packets
- Individual packet analysis

In particular, we focus on processing complexity and memory accesses. The memory accesses and coverage statistics distinguish not only between instruction and data memory, but further separate data memory into 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. The detailed packet processing analysis (complexity variation, basic block coverage, and memory accesses) explores the instruction execution path and memory access differences between packets. These metrics can help us to achieve efficient operation of an application on a network processor.

A. Average Statistics

1) 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 [16]. We show the processing complexity for the various applications discussed in the previous section in Table II. The processing complexity for all four applications is in the range expected for header processing applications. The following observations can be made:

TABLE II
Average Number of Instructions per Packet Executed for Various Applications.

Trace Name	IPv4-radix	IPv4-trie	Flow Classification	TSA
MRA	4,438	206	162	903
COS	4,388	206	164	906
ODU	4,324	206	157	901
LAN	4,820	200	152	904
Average	4,493	205	159	904

- 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).
- 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 and TSA are strictly linear (i.e., not data-dependent) applications which take approximately the same amount of instructions irrespective of the trace.
- 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.

2) Memory Accesses: As discussed previously, 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).

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 (e.g. routing tables, flow tables). The analysis was performed for the first 10,000 packets of each trace. The results are summarized in Table III. The applications require only a few (18–32) accesses to packet memory. Non-packet memory is used much more heavily (17–842). This is due to the fact that all four applications maintain large data structures in memory.

3) 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.

The data memory size is quite large for all applications except IPv4-trie (we use a small routing table for this particular application) and TSA. This is again due to large data structures

TABLE III

AVERAGE NUMBER OF ACCESSES TO PACKET MEMORY AND NON-PACKET MEMORY (10,000 PACKETS).

Trace Name	IPv4-radix		IPv4-trie		Flow classification		TSA	
	Packet	Non-packet	Packet	Non-packet	Packet	Non-packet	Packet	Non-packet
MRA	32	842	32	19	23	58	19	89
COS	32	836	32	19	24	60	18	88
ODU	32	833	32	19	23	54	19	88
LAN	32	831	32	17	23	51	18	88
Average	32	836	32	18	23	56	18	88

TABLE IV

INSTRUCTION AND DATA MEMORY SIZES (IN BYTES) FOR VARIOUS APPLICATIONS.

Application	Instr. memory size	Data memory size
IPv4-radix	4,420	18,004
IPv4-trie	584	2908
Flow classification	1,584	43,344
TSA	836	2668

and shows that the memory regions accessed are very different for individual packets. Otherwise, an application cannot cover such a large region of memory with few memory accesses (e.g. Flow Classification covers 43,344 memory locations with an average of 56 non-packet memory accesses per packet).

These averaged statistics only give a glimpse into the workload characteristics of network processing. We provide further details by comparing the variation in processing behavior between individual packets.

B. Variation Across Packets

The variation in processing across packets is an important characteristic in network processing. Network processors are highly parallel embedded systems with relatively simplistic processor cores and hardly any operating system support. As a result, NP applications are often fine-tuned offline by hand. Variations in processing behavior due to different packet characteristics can impact such optimizations. Due to space constraints, we present detailed plots for only two of the four applications presented in Section IV-A - IPv4-radix and Flow Classification. The observations are applicable to the other two applications.

1) Processing Complexity: Figure 3 shows the processing complexity for the two applications. This analysis was performed for the first 500 packets of the MRA trace. IPv4-radix shows a significant variation in the number of instructions executed for individual packets. This is due to the fact that different locations in the routing table are being searched depending on the address prefixes encountered in the packet. For Flow Classification, the variation is not as significant, but a few cases dominate (e.g. around 156 instructions and 212 instructions). These correspond to the processing required to update the flow table for packets that belong to existing flows or create new flows. To illustrate this better, Table V shows the three most common occurrences of processing cost of all applications for 100,000 packets of the COS trace. The total percentages for the three most common occurrences are close to 90% except for IPv4-radix. This indicates that, despite variations, most applications are very regular and a few common cases dominate processing. This is an important observation that can help when optimizing network processing applications.

2) Memory Accesses: The variation in the number of memory accesses is very small when considering packet memory. In most cases, the same number of header fields need to be extracted or overwritten and this number is almost constant. Figure 4 shows this for IPv4-radix. Flow Classification shows a bit more variation, but is also dominated by a few common cases.

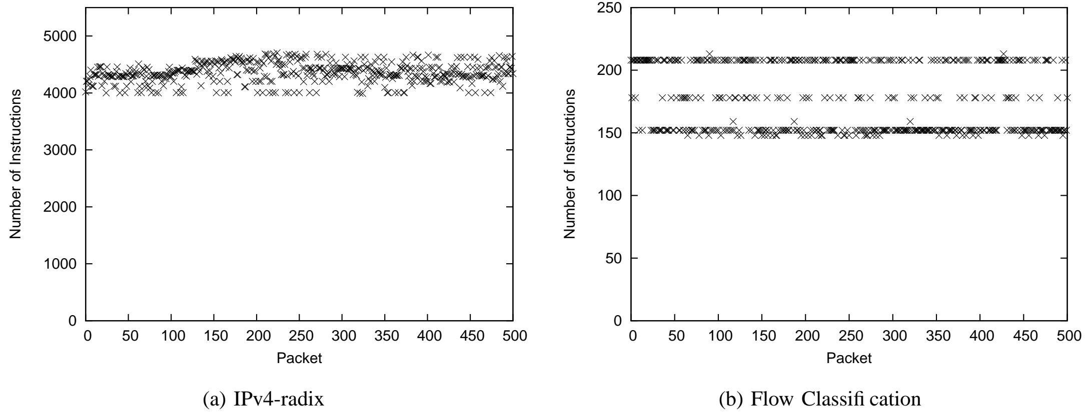
When looking at non-packet memory, the variation follows roughly the variations in the number of instructions executed (see Figure 5). This is to be expected as most applications have a fairly constant ratio of memory accesses to overall instructions executed.

Figure 3 shows that there are differences in the total number of instructions executed for each packet. From a practical point of view, it is also important to explore if these instructions are generated from the same piece of code. If they are, then locality in the instruction store can be exploited, since the amount of instruction store available is limited. To understand this aspect better, we explore the details of processing a single packet.

C. Individual Packet Analysis

In this part of the analysis, we look at two different characteristics of the instructions that are executed while processing a packet - the basic blocks they belong to, and sequences of instructions (such as loops) that are repeatedly executed. This type of analysis is particularly important for network processors as it gives insight into how to optimize application execution. Due to the highly repetitive nature of network processing, any improvement in the implementation will have considerable impact on the entire system. Also, the simplicity of network processing makes such an analysis feasible.

1) Instruction Pattern: Figure 6 shows the instruction access patterns for an application 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 (shown on the y-axis). The x-axis shows the instruction number of the instructions that are required to process that particular packet. Overlaps of the graph on the y-axis represent sequences of instructions which are repeatedly executed (such as loops).

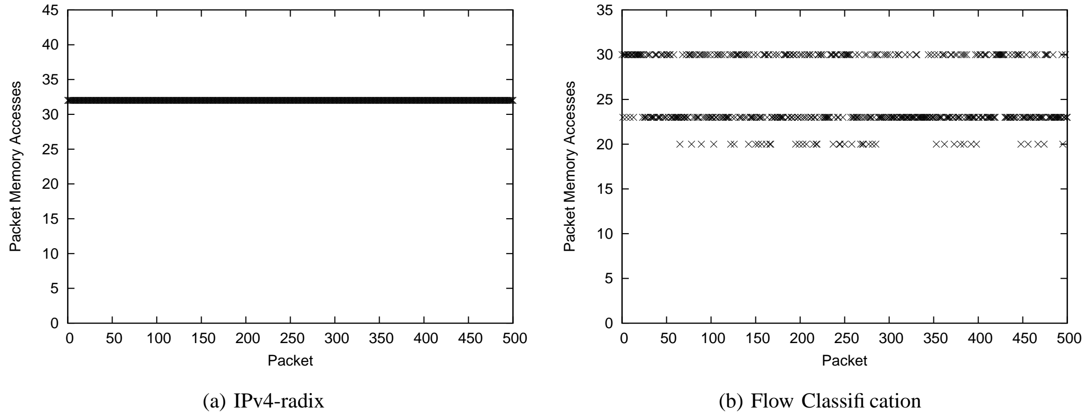


(a) IPv4-radix (b) Flow Classifi cation

Fig. 3. Packet Processing Complexity Variation. The number of total instructions executed varies over different packets in the trace.

TABLE V
VARIATION OF EXECUTED INSTRUCTIONS.

Application	Number of instructions					
	Most frequent occurrences					
	1st	2nd	3rd	Minimum	Maximum	Average
IPv4-radix	3,991 (10.53%)	4,191 (6.04%)	4,292 (3.20%)	3,989 (0.16%)	4,728 (0.02%)	4,313
IPv4-trie	200 (49.00%)	211 (37.23%)	199 (8.02%)	199 (8.02%)	233 (0.09%)	206
Flow Class.	156 (82.60%)	212 (9.58%)	152 (1.79%)	128 (1.00%)	223 (8.0e-5%)	162
TSA	906 (83.95%)	911 (7.62%)	910 (3.28%)	833 (0.64%)	922 (0.13%)	906



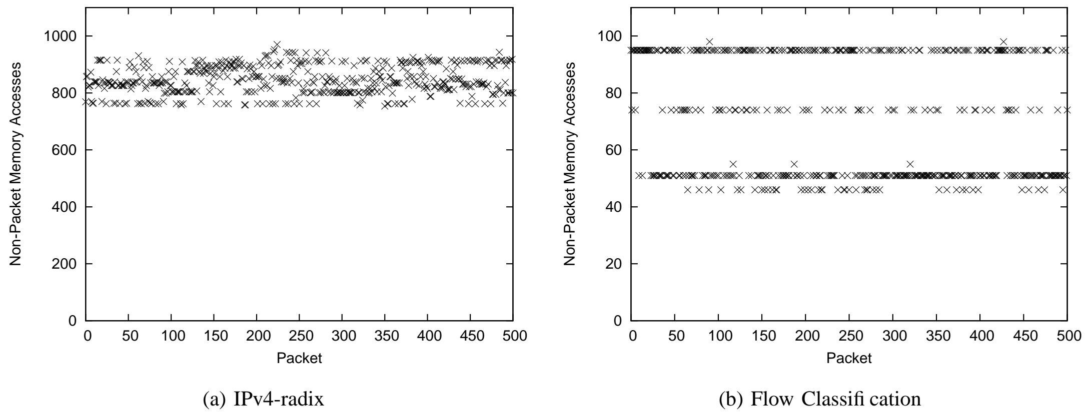
(a) IPv4-radix (b) Flow Classifi cation

Fig. 4. Packet Memory Access Pattern. The variation in accesses to packet memory is very small.

Flow classification exhibits very linear program behavior with very little repetition of instructions. IPv4-radix (similar to the other two applications) shows very regular patterns in which the instructions are accessed. For example, there is a loop that is executed four times between instructions 400 and 1800.

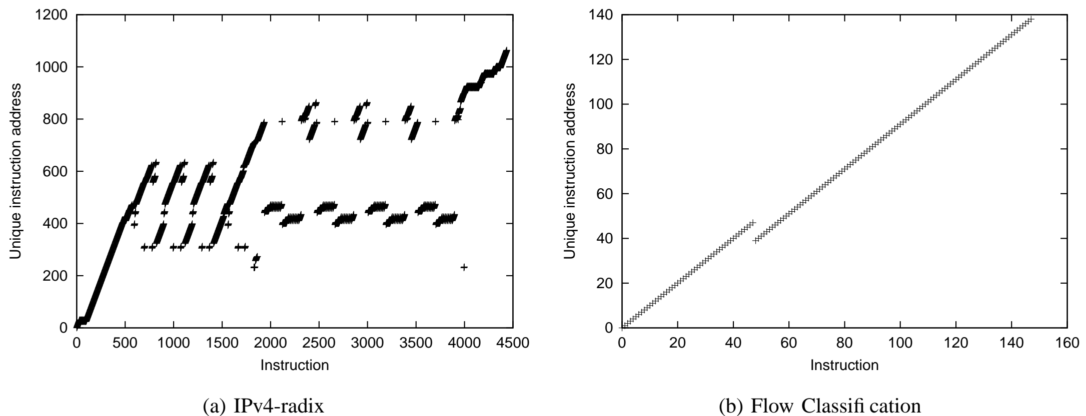
2) Unique Instructions: From the previous result, we can see that the y-axis shows the number of unique instructions that are executed. The variation of the number of unique instructions across packets is shown in detail in Table VI for the four applications. These numbers were computed for the first 100,000 packets of the COS trace. It can be observed that the variation in the number of unique instructions is much less than the variation of total instructions executed in Table V. In terms of how often instructions are repeatedly executed, IPv4-radix and TSA show a factor of about four, whereas IPv4-trie and Flow Classification are almost linear and show only little repetition.

3) Basic Block Access Frequency: To further illustrate the differences in the execution path for different packets, Figure 7 shows the probability that a basic block will be executed while processing a packet for the IPv4-radix and Flow Classification applications. For IPv4-radix most basic blocks are executed for all cases (execution probability equals 1). Some blocks (#30–#70) are executed less frequently (80% probability down to almost 0%). These blocks are handling special cases of packet processing and can potentially be left out of the fast path of a router system thereby saving instruction store space and be handled by the slow path. For Flow Classification,



(a) IPv4-radix (b) Flow Classification

Fig. 5. Non-Packet Memory Access Pattern. The variation in accesses to non-packet data memory is correlated to the variations in instructions executed (see Figure 3).



(a) IPv4-radix (b) Flow Classification

Fig. 6. Detailed Packet Processing. The y-axis shows unique instruction addresses. Overlaps indicate loops.

TABLE VI
VARIATION OF UNIQUE EXECUTED INSTRUCTIONS.

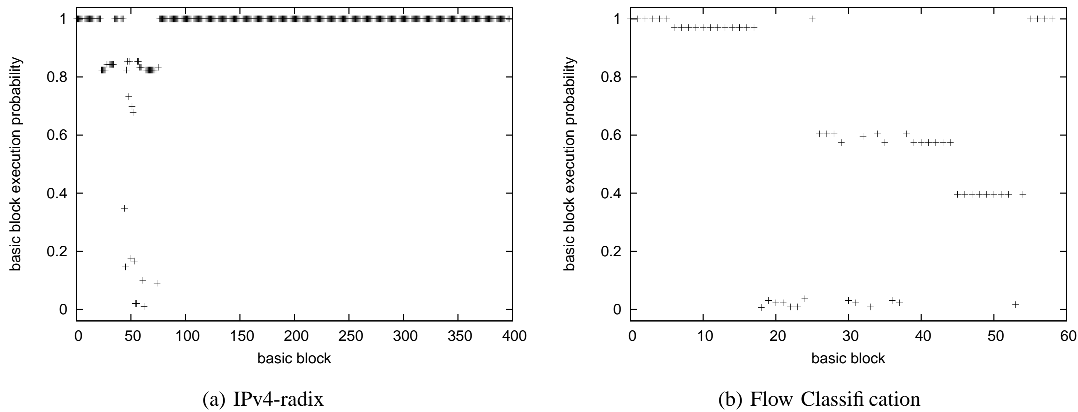
Application	Number of unique instructions					
	Most frequent occurrences					
	1st	2nd	3rd	Minimum	Maximum	Average
IPv4-radix	1,064 (36.06%)	963 (23.57%)	1,068 (16.41%)	963 (23.57%)	1,077 (0.78%)	1,040
IPv4-trie	128 (50.82%)	139 (39.15%)	127 (9.31%)	127 (9.31%)	142 (0.70%)	133
Flow Class.	147 (82.60%)	203 (9.58%)	143 (1.79%)	119 (1.00%)	214 (8.0e-5%)	153
TSA	172 (83.95%)	181 (7.62%)	180 (3.29%)	161 (0.88%)	184 (0.42%)	173

a similar pattern can be observed with most blocks being executed frequently and a few blocks with almost no usage.

4) Basic Block Coverage: The observation that some pieces of code are executed in only rare cases gives rise to the question of the minimum number of basic blocks that need to be installed in the fast path of a network processor to efficiently process most packets. Minimizing this number while being able to process a large percentage of packets allows the system to achieve higher throughput and minimize the amount of instruction store required for a given application.

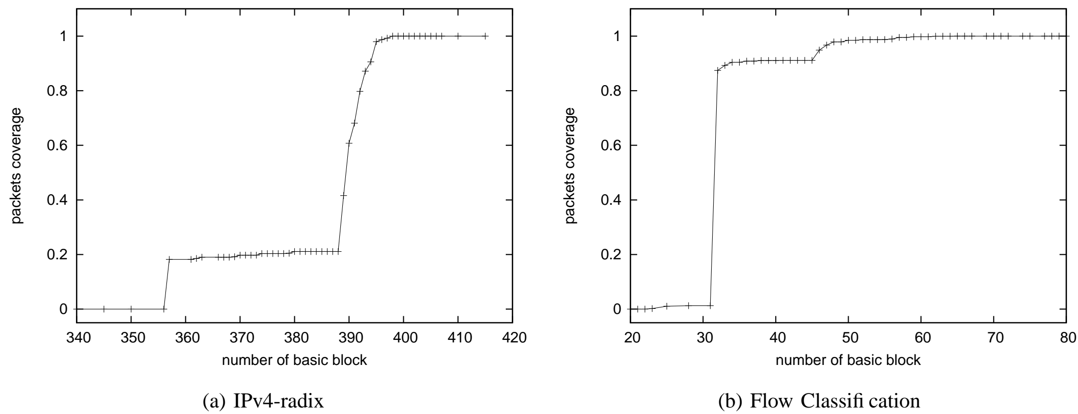
Figure 8 shows the percentage of packets that can be processed (y-axis) with a given number of basic blocks (x-axis). The goal here is to find a tradeoff between the number of basic blocks that can be stored (in an instruction store for example) and the number of packets that can be processed with those basic blocks. If too few basic blocks are present, we risk being unable to process certain types of packets. If more basic blocks are available, more types of packets can be processed, but we risk using up too much storage space. The "sweet spots" of these plots are the steps, where the packet coverage increases by adding on more basic block (e.g., 395 basic blocks for IPv4-radix and 32 for Flow Classification). In both cases over 90% packet coverage can be achieved. Additional basic blocks increase this value incrementally.

5) Memory Access Sequence: Memory access patterns while processing a single packet are shown in Figure 9, for



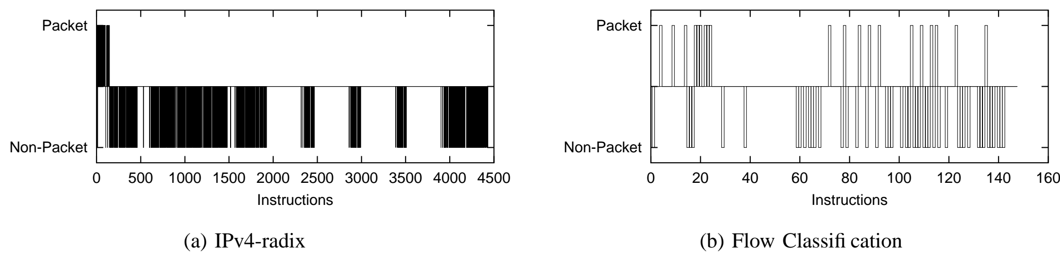
(a) IPv4-radix (b) Flow Classification

Fig. 7. Basic Block Access Frequency. A probability of 1 indicates the basic block is executed for every packet.



(a) IPv4-radix (b) Flow Classification

Fig. 8. Packet Coverage. The coverage indicates what fraction of the packet trace can be processed with a given number of basic blocks.



(a) IPv4-radix (b) Flow Classification

Fig. 9. Data Memory Access Pattern.

the IPv4-radix and Flow Classification applications. 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. IPv4-radix accesses the packet memory (reading IP header fields) initially and then operates entirely on non-packet data memory to search the routing data structure. In Flow Classification, the both packet data and program state are accessed continuously.

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. 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 [29]. This is useful in the context of network simulations, where processing delay is currently not or only superficially considered. These workload characteristics can also be used in other performance models of network processor systems [30] [14].

The detailed instruction analysis is useful to identify se-

mantic 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 6.

One of the key design parameters of network systems is the size of the on-chip instruction store of processing engines. It has to be big enough to accommodate enough instructions to achieve sufficient packet coverage (see Figure 8). Alternatively, applications can be partitioned across multiple processing engines. Preliminary results can be found in [31].

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. It is currently available for download from [32]. We present results for four different networking applications. 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). The detailed analysis of processing characteristics of individual packets and the variation between them helps to gain a better understanding of network processing workloads.

References

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

[2] Doug Burger and Todd Austin, "The SimpleScalar tool set version 2.0," *Computer Architecture News*, vol. 25, no. 3, pp. 13–25, June 1997.

[3] Xiaohu Qie, Andy Bavier, Larry Peterson, and Scott Karlin, "Scheduling computations on a software-based router," in *Proc. IEEE Joint International Conference on Measurement & Modeling of Computer Systems (SIGMETRICS)*, Cambridge, MA, June 2001, pp. 13–24.

[4] Jeffrey C. Mogul, "Simple and flexible datagram access controls for UNIX-based gateways," in *USENIX Conference Proceedings*, Baltimore, MD, June 1989, pp. 203–221.

[5] Kjeld Borch Egevang and Paul Francis, "The IP network address translator (NAT)," RFC 1631, Network Working Group, May 1994.

[6] George Apostolopoulos, David Aubespin, Vinod Peris, Prashant Pradhan, and Debanjan Saha, "Design, implementation and performance of a content-based switch," in *Proc. of IEEE INFOCOM 2000*, Tel Aviv, Israel, Mar. 2000, pp. 1117–1126.

[7] Alex S. Snoeren, Craig Partridge, Luis A. Sanchez, Christine E. Jones, Fabrice Tchakountio, Stephen T. Kent, and W. Timothy Strayer, "Hash-based ip traceback," in *Proc. of ACM SIGCOMM 2001*, San Diego, CA, Aug. 2001, pp. 3–14.

[8] Intel Corp., *Intel Second Generation Network Processor*, 2002, http://www.intel.com/design/network/products/npfamily/ixp2400.htm.

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

[10] Patrick Crowley, Marc E. Fiuczynski, Jean-Loup Baer, and Brian N. Bershad, "Characterizing processor architectures for programmable network interfaces," in *Proc. of 2000 International Conference on Supercomputing*, Santa Fe, NM, May 2000, pp. 54–65.

[11] Patrick Crowley and Jean-Loup Baer, "A modelling framework for network processor systems," in *Proc. of Network Processor Workshop in conjunction with Eighth International Symposium on High Performance Computer Architecture (HPCA-8)*, Cambridge, MA, Feb. 2002, pp. 86–96.

[12] Lothar Thiele, Samarjit Chakraborty, Matthias Gries, and Simon Künzli, "Design space exploration of network processor architectures," in *Proc. of Network Processor Workshop in conjunction with Eighth International Symposium on High Performance Computer Architecture (HPCA-8)*, Cambridge, MA, Feb. 2002, pp. 30–41.

[13] Mark A. Franklin and Tilman Wolf, "A network processor performance and design model with benchmark parameterization," in *Network Processor Design: Issues and Practices, Volume 1*, Patrick Crowley, Mark A. Franklin, Haldun Hadimioglu, and Peter Z. Onufryk, Eds., chapter 6, pp. 117–138. Morgan Kaufmann Publishers, Oct. 2002.

[14] Mark A. Franklin and Tilman Wolf, "Power considerations in network processor design," in *Proc. of Network Processor Workshop in conjunction with Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*, Anaheim, CA, Feb. 2003, pp. 10–22.

[15] Patrick Crowley, Marc E. Fiuczynski, Jean-Loup Baer, and Brian N. Bershad, "Workloads for programmable network interfaces," in *IEEE Second Annual Workshop on Workload Characterization*, Austin, TX, Oct. 1999.

[16] Tilman Wolf and Mark A. Franklin, "CommBench - a telecommunications benchmark for network processors," in *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Austin, TX, Apr. 2000, pp. 154–162.

[17] Gokhan Memik, William H. Mangione-Smith, and Wendong Hu, "NetBench: A benchmarking suite for network processors," in *Proc. of International Conference on Computer-Aided Design*, San Jose, CA, Nov. 2001, pp. 39–42.

[18] Byeong Kil Lee and Lizy Kurian John, "NpBench: A benchmark suite for control plane and data plane applications for network processors," in *Proceedings of IEEE International Conference on Computer Design (ICCD 03)*, San Jose, CA, Oct. 2003, pp. 226–233.

[19] "Embedded microprocessor benchmark consortium," http://www.eembc.org.

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

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

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

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

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

[25] Stefan Nilsson and Gunnar Karlsson, "IP-address lookup using LC-tries," *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 6, pp. 1083–1092, June 1999.

[26] Ramaswamy Ramaswamy and Tilman Wolf, "High-speed prefix-preserving IP address anonymization for passive measurement systems," *under submission*.

[27] Jun Xu, Jinliang Fan, Mostafa H. Ammar, and Sue B. Moon, "Prefix-preserving ip address anonymization: Measurement-based security evaluation and a new cryptography-based scheme," in *Proc. of 10th IEEE International Conference on Network Protocols (ICNP'02)*, Paris, France, Nov. 2002, pp. 280–289.

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

[29] Ramaswamy Ramaswamy, Ning Weng, and Tilman Wolf, "Characterizing network processing delay," in *Proc. of IEEE Global Communications Conference (GLOBECOM)*, Dallas, TX, Nov. 2004.

[30] Mark A. Franklin and Tilman Wolf, "A network processor performance and design model with benchmark parameterization," in *Proc. of Network Processor Workshop in conjunction with Eighth International Symposium on High Performance Computer Architecture (HPCA-8)*, Cambridge, MA, Feb. 2002, pp. 63–74.

[31] Ning Weng and Tilman Wolf, "Pipelining vs. multiprocessors - choosing the right network processor system topology," in *Proc. of Advanced Networking and Communications Hardware Workshop (ANCHOR 2004) in conjunction with The 31st Annual International Symposium on Computer Architecture (ISCA 2004)*, Munich, Germany, June 2004.

[32] "Packetbench - workload characterization," http://www-unix.ecs.umass.edu/ rramaswa/pb/.