Journal of Systems Architecture 55 (2009) 421–433

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Analysis of network processing workloads

Ramaswamy Ramaswamy [a], Ning Weng [b], Tilman Wolf [c,*]

[a] Cisco Systems Inc., San Jose, CA, USA
[b] Department of Electrical and Computer Engineering, Southern Illinois University, Carbondale, IL, USA
[c] Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA, USA

ARTICLE INFO

Article history:
Received 17 March 2009
Received in revised form 10 August 2009
Accepted 3 September 2009
Available online 15 September 2009

Keywords:
Network processor
Workload characterization
Router design

ABSTRACT

Network processing is becoming an increasingly important paradigm as the Internet moves towards an architecture with more complex functionality in the data path. 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. For statistics collection, PacketBench provides the ability to derive a number of microarchitectural and networking related metrics. We show a range of workload results that focus on individual packets and the variation between them. The understanding of workload details of network processing has many practical applications. We discuss how PacketBench results can be used to estimate network processing delay that are very close to those obtained from measurement.

1. Introduction

The Internet has progressed from a simple store-and-forward network to a more complex communication infrastructure. To meet demands for security, flexibility, and performance in today's networks data packets not only need to be forwarded from router to router, but also processed inside the network [1]. This trend towards more complex data path processing continues in next-generation Internet architectures that are being considered [2]. Such network 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 requests).

To handle the constantly varying functional requirements of the networking domain, router designs have moved away from hardwired ASIC forwarding engines. Instead, software-programmable "network processors" (NPs) have been developed in recent years [3]. These NPs are typically multiprocessor systems on a chip (MPSoC) 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 Fig. 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 processing workload characteristics of this domain in more detail.

The processing workload on network nodes is unique and 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 [4]), are not necessarily applicable to the NP domain. Good examples are the dominance of I/O and the requirements for the memory hierarchy, where smaller on-chip memories suffice due to the nature of packet processing.

To explore and understand network processing workloads in more detail, we present in this paper 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

* Corresponding author. Address: Department of Electrical and Computer Engineering, University of Massachusetts, 151 Holdsworth Way, Amherst, MA 01003, USA. Tel.: +1 413 545 0757.
E-mail addresses: ramramas@cisco.com (R. Ramaswamy), nweng@siu.edu (N. Weng), wolf@ecs.umass.edu (T. Wolf).

1383-7621/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2009.09.001

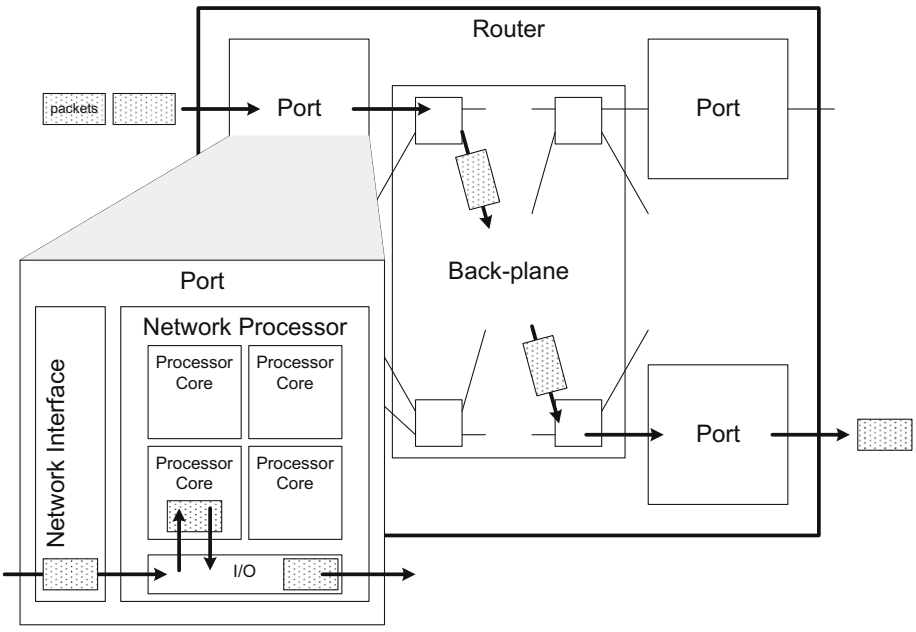


Fig. 1. Router system with network processor. Packets are processed by one of multiple processing cores in the network processor.

traces. The simulation environment is set up to collect statistics only 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 traditional microarchitectural statistics (since PacketBench uses SimpleScalar [5] 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 and different in several ways. First, PacketBench applications can be programmed easily with 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 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. Finally, note that *PacketBench is not a benchmark suite*. Instead, PacketBench is a tool to implement any packet processing applications. The user may choose which are considered representative.

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 run-time 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. Due to a lack of operating system support on NPs, applications are typically fine-tuned offline for a given system.
- **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). 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.

The remainder of this paper is organized as follows. Section 2 discusses related work. We present an overview of PacketBench in Section 3. We introduce several sample applications for PacketBench in Section 4. Workload characteristics of these applications are presented and discussed in Section 5. Section 6 describes how the results obtained from PacketBench can be used in a practical scenario. A summary and conclusions are presented in Section 7.

2. Related work

There are numerous examples of processing packets on network nodes that extend the basic packet forwarding paradigm. Routers can perform firewalling [6], network address translation (NAT) [7], web switching [8], IP traceback [9], 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 A few examples are the Intel IXP2400 [10], AMCC np7300 [11], Cisco QuantumFlow [12], and EZchip NP-3 [13]. The design spaces of such network processors have been explored in several ways. Crowley et al. have evaluated different processor architectures for their performance under networking workloads [14]. This work mostly focuses on the tradeoffs between RISC, superscalar, and multithreaded architectures. A modeling framework is proposed that considers the data flow through the system [15]. Thiele et al. have proposed a performance model for network processors [16] that considers the effects of the queuing system. In our previous work, we have developed a quantitative performance and power consumption model for a range of design parameters [17]. These models require detailed workload parameters to obtain realistic results.

Several network processor benchmarks have captured various characteristics of network processing. Crowley et al. have defined simple programmable network interface workloads in [18]. In our previous work, we have defined a network processor benchmark called CommBench [19]. Memik et al. have proposed a similar benchmark more recently [20]. Lee and John have extended these benchmarks to also consider control plane applications [21]. Embedded systems benchmarks from the Embedded Microprocessor Benchmark Consortium [22] and the MiBench suite [23] contain some typical network applications. 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 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. PacketBench is based on SimpleScalar [5], a tried and tested microarchitectural simulator. Other network processor simulators exist, but these are not programmable in C (e.g., NePSim [24], Intel workbench [25]).

The results obtained from PacketBench can be used to parameterize performance models used in the early stages of network processor design (e.g., [26]). It is also conceivable that the workload is used in models of programmable processor systems that support automatic generation of simulators and compilers (e.g., Tensilica [27], CoWare [28]). The use of multi-core embedded processors in the networking domain is an example of a broader trend towards throughput-oriented computing (e.g., graphics processing, data analysis).

3. 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 this section.[1]

3.1. PacketBench system

The goal of PacketBench is to emulate the functionality of a network processor. The conceptual outline of the tool is shown in Fig. 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 4. 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 [5], 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 5).

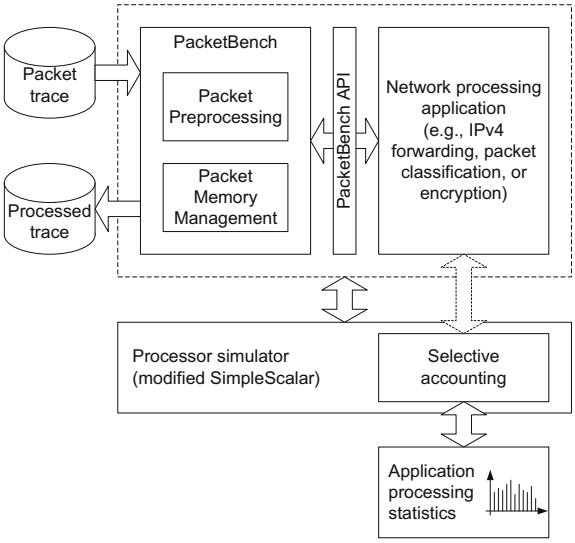


Fig. 2. PacketBench architecture. The selective accounting component ensures that only statistics from the application are collected. Instructions that are executed as part of the PacketBench framework are ignored.

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.

[1] The PacketBench source code and installation instructions can be found at: http://www.ecs.umass.edu/ece/wolf/nsl/software/pb/.

3.2. 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).

3.3. PacketBench prototype

PacketBench is simulated on a typical processor simulator to obtain processing statistics. For this purpose, we use the ARM [29] target of the SimpleScalar [5] 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 [10]. Also, SimpleScalar is freely available with source code and can easily be modified. PacketBench supports packet traces in the *tcpdump* [30] format and the Time Sequenced Header (TSH) format from NLANR [31]. PacketBench generates a very small overhead and does not significantly reduce the performance of SimpleScalar.

4. 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 5.

4.1. Network processing applications

We have chosen eight applications for gathering workload statistics using PacketBench. An overview of the applications is shown in Table 1. Applications are implemented using custom code and open source libraries. Applications are classified as either *header processing applications* (HPA) or *payload processing applications* (PPA) as defined in [19]. HPAs process a limited amount of data in the packet headers and their processing requirements are independent of packet size. PPAs perform computations over the payload portion of the packet and are more demanding in terms of computational power as well as memory bandwidth.

The specific applications are:

- **IPv4-radix.** IPv4-radix is an application that performs RFC1812-compliant packet forwarding [32] 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 [33].
- **IPv4-trie.** IPv4-trie is similar to IPv4-radix and also performs RFC1812-based packet forwarding and was derived from [34]. 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) [35] 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 [36]. In addition to anonymizing the IP addresses, layer 3 and layer 4 headers are collected for each packet that is encountered in the trace.
- **IPSec-AES.** IPSec-AES is an implementation of the IP Security Protocol [37], where the packet payload is encrypted using the Rijndael Advanced Encryption Standard (AES) [38] algorithm.
- **IPSec-DES.** IPSec-DES is also an implementation of the IP Security Protocol, but uses the Data Encryption Standard (DES) [39] algorithm for packet payload encryption. The DES algorithm is employed in several VPN devices.
- **String matching.** String matching is an application where packet payloads are searched for occurrences of patterns. This application contains the implementation of the search algorithm used in the SNORT [40] intrusion detection system.
- **Fingerprinting.** Fingerprinting is an application that identifies commonly occurring patterns in packet payloads using Rabin fingerprints [41]. This kind of processing is performed in worm signature generation systems [42][43] and caching systems [44].

This selection of applications covers a broad space of typical network processing. They vary in terms of both processing complexity and data memory requirements as the results in Section 5 show.

Table 1
Applications analyzed using PacketBench.

Application name	Function	Processing type
IPv4-radix	Table lookup	Header
IPv4-trie	Table lookup	Header
Flow Classification	Table lookup and update	Header
TSA	IP address anonymization	Header
IPSec-AES	Payload encryption with AES	Payload
IPSec-DES	Payload encryption with DES	Payload
String matching	IDS pattern detection	Payload
Fingerprinting	Worm signature generation	Payload

4.2. 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 2 shows the packet traces that we used to evaluate the applications. Traces MRA, COS, and ODU are obtained from the NLANR repository [31] 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 1000 to 100,000 packets extracted from the traces shown in Table 2. 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

Table 2
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)	904,668
LAN	100 Mbps (Ethernet)	100,000

possible execution paths of the application are considered (as shown in Section 5). 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 [45] 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. Additionally, the NLANR traces do not contain packet payloads. Random packet payloads are generated for these traces in the packet preprocessing stage of PacketBench.

5. 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 highlight 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 [19] as well as by other related work [18,20,21]. 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, and
- 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. 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.

5.1. Average statistics

5.1.1. 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 [19]. We show the processing complexity for header processing applications in Table 3 and the processing complexity for payload processing applications in Table 4. The following observations can be made:

- The average number of instructions executed by payload processing applications exceeds the average number of instructions executed by header processing applications by several orders of magnitude. This is an expected result since payload processing involves more complex computation than header processing.
- Payload processing applications show a lot more variation in the number of instructions executed per packet due to the fact that they are dependent on the size of the packet payload.
- There is little variation in the number of instructions per packet for header processing applications. This indicates that header processing applications incur a fixed processing cost irrespective of the size of the packet.
- Any variation in the number of instructions is application specific. For example, in 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). The variation in Flow Classification is caused by packets hashing to different locations in the flow table.
- Different implementations of the same application have varying complexities depending on the data structures and algorithms that are used. For example, 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. Similarly, IPSec-AES uses a newer more computationally efficient algorithm than IPSec-DES and executes fewer instructions per packet.

Table 3
Average number of instructions per packet executed for header processing applications.

Trace name	IPv4-radix	IPv4-trie	Flow Classification	TSA
MRA	4438	206	162	903
COS	4388	206	164	906
ODU	4378	207	161	906
LAN	4972	200	152	900
Average	4544	205	160	904

Table 4
Average number of instructions per packet executed for payload processing applications.

Trace name	IPSec-AES	IPSec-DES	String matching	Fingerprinting
MRA	40,533	191,960	7597	48,148
COS	39,865	188,723	7470	47,233
ODU	39,001	184,584	7312	46,242
LAN	23,990	111,922	5411	25,519
Average	35,847	169,297	6948	41,786

5.1.2. 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 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 using 32-bit wide memory access. The results are summarized in Table 5 for header processing applications and Table 6 for payload processing applications. The following observations can be made:

- Header processing applications (Table 5) require only a few (18–32) accesses to packet memory. Non-packet memory is used much more heavily (17–842) and shows accesses to large data structures maintained in memory.
- The number of memory accesses for payload processing applications (Table 6) is several orders of magnitude more than header processing applications. This indicates that payload processing applications access both packet data and program data more frequently than header processing applications.
- Similar to the processing complexity, header processing applications show little variation in the amount of memory accesses made between packets of different traces. Payload processing applications exhibit a lot more variation in memory accesses made (both to packet and non-packet memory) due to their data dependent nature.

5.1.3. 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 1000 packets of the MRA trace and for all eight sample applications. The results are shown in Table 7 for header and payload processing applications.

The data memory size is quite large for most applications except IPv4-trie (we use a small routing table for this particular application) and TSA. This is again due to large data structures 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 57 non-packet memory accesses per packet.)

The instruction memory sizes in Table 7 imply a small program core which is executed several times, particularly for payload processing applications. For example, in IPSec-AES, the instruction memory size is only 2800 bytes (approximately 700 instructions assuming 32 bit instructions), but the average number of instructions executed per packet is 35,847. This re-emphasizes 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 kilobytes.

Memory optimization is a crucial task in the design of any application that runs on a network system since the time taken to access memory is the dominant factor in the overall processing time. The data shown in Table 7 can be used to optimize an application for a particular network processor or to design more efficient memory hierarchies for next generation network processors.

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.

5.2. 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 off-line 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 eight applications presented in Section 4.1 – IPv4-radix and IPSec-AES.

5.2.1. Processing complexity

Fig. 3a and b show the processing complexity for the two applications. This analysis was performed for the first 500 packets of the MRA trace. IPv4-radix shows some 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 IPSec-AES, the variation is even more significant since the application is data dependent and hence varies with packet size. The data

Table 5
Average number of accesses to packet memory and non-packet memory for header processing applications.

Trace name	IPv4-radix		IPv4-trie		Flow classification		TSA	
	Pkt.	Non-pkt.	Pkt.	Non-pkt.	Pkt.	Non-pkt.	Pkt.	Non-pkt.
MRA	32	842	32	19	23	58	19	89
COS	32	836	32	19	24	60	18	88
ODU	32	841	32	19	24	58	18	88
LAN	32	898	32	17	23	52	20	89
Average	32	854	32	19	24	57	19	89

Table 6
Average Number of accesses to packet memory and non-packet memory for payload processing applications.

Trace name	IPSec-AES		IPSec-DES		String matching		Fingerprinting	
	Pkt.	Non-pkt.	Pkt.	Non-pkt.	Pkt.	Non-pkt.	Pkt.	Non-pkt.
MRA	841	23,844	1652	66,545	662	2860	1221	9740
COS	828	23,447	1625	65,422	650	2812	1198	9557
ODU	810	22,935	1590	63,986	636	2753	1173	9357
LAN	508	14,038	974	38,776	387	1988	648	5162
Average	747	21,066	1460	58,682	584	2603	1060	8454

Table 7
Instruction and data memory sizes (in bytes) for header and payload processing applications.

Application	Instr. memory size	Data memory size
IPv4-radix	4420	18,004
IPv4-trie	584	2908
Flow classification	1584	43,344
TSA	836	2668
IPSec-AES	2800	9428
IPSec-DES	2444	8676
String Matching	1608	105,772
Fingerprinting	820	8016

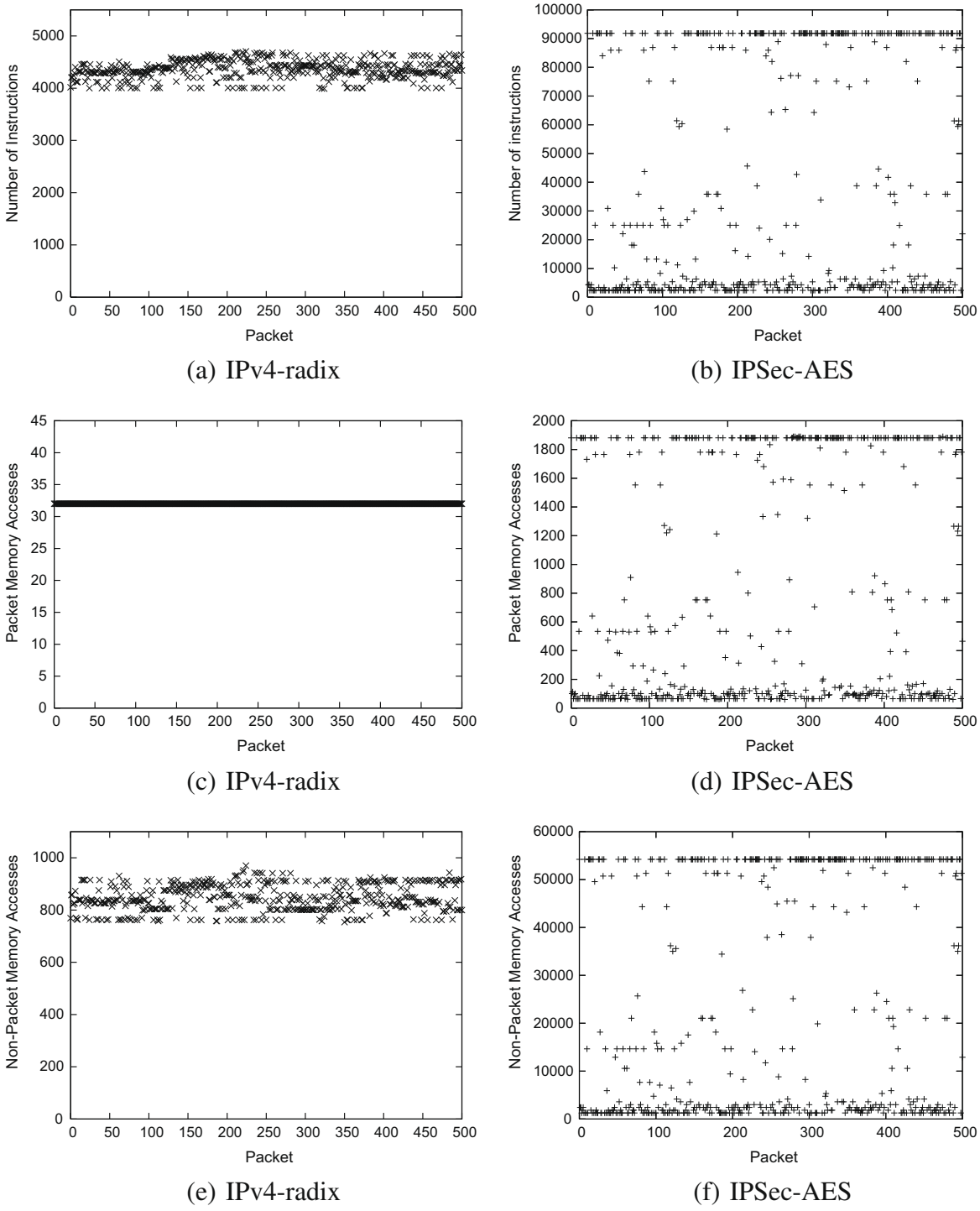


Fig. 3. Packet processing variation. The variation over different packets in the trace for instructions executed, packet memory accesses, and non-packet memory accesses are shown for two applications (IPv4-radix and IPSec-AES).

points at the bottom of the graph indicate the processing for small packets while the data points near the top of graph indicate the processing for larger sized packets.

5.2.2. Memory accesses

The variation in the number of packet memory accesses is shown in Fig. 3c and d for the IPv4-radix and IPSec-AES applications respectively. Since IPv4-radix is a header processing application, the variation in the number of packet memory accesses is very small. In most cases, the same number of header fields need to be extracted or overwritten and this number is almost constant. Fig. 3c shows this for IPv4-radix. For IPSec-AES, a trend similar to the one seen for instructions is exhibited. The number of packet memory accesses made depends on the size of packet payload and shows a significant variation across packets.

When looking at non-packet memory (Fig. 3e and f), the variation follows roughly the variations in the number of instructions executed. This is to be expected as most applications have a fairly constant ratio of memory accesses to overall instructions executed.

Fig. 3 shows that there are differences in the total number of instructions executed and memory accesses made 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.

5.3. 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. Due to space constraints, detailed packet analysis is performed only for the IPv4-radix and Flow Classification applications.

5.3.1. Instruction pattern

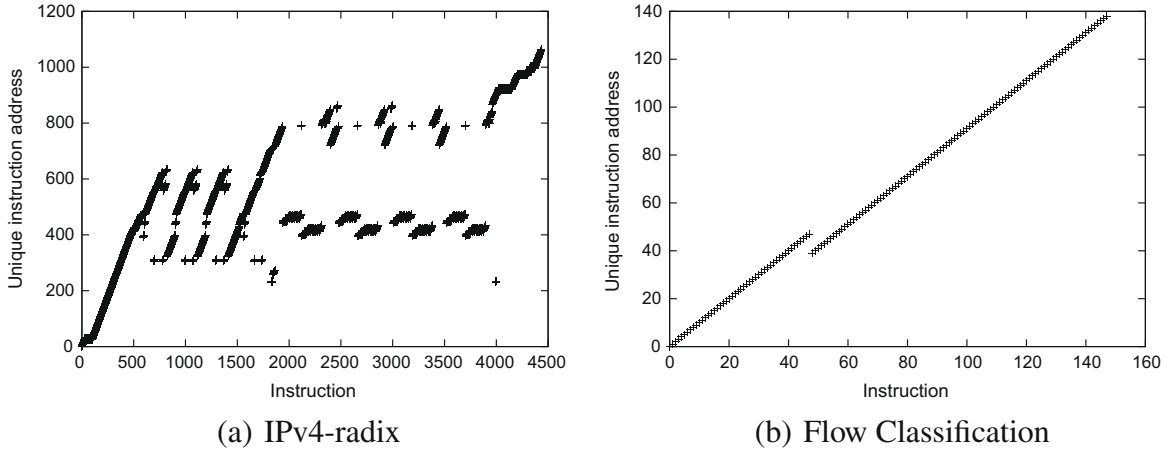
Fig. 4 shows the instruction access patterns for the IPv4-radix and flow classification applications while processing a packet. In order to view the instruction patterns more clearly, the instruction addresses were assigned a unique index depending on the order in which they were executed (shown on the *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). Flow classification exhibits very linear program behavior with very little repetition of instructions. IPv4-radix 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.

5.3.2. Basic block access frequency

To further illustrate the differences in the execution path for different packets, Fig. 5 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 similar pattern can be observed with most blocks being executed frequently and a few blocks with almost no usage.

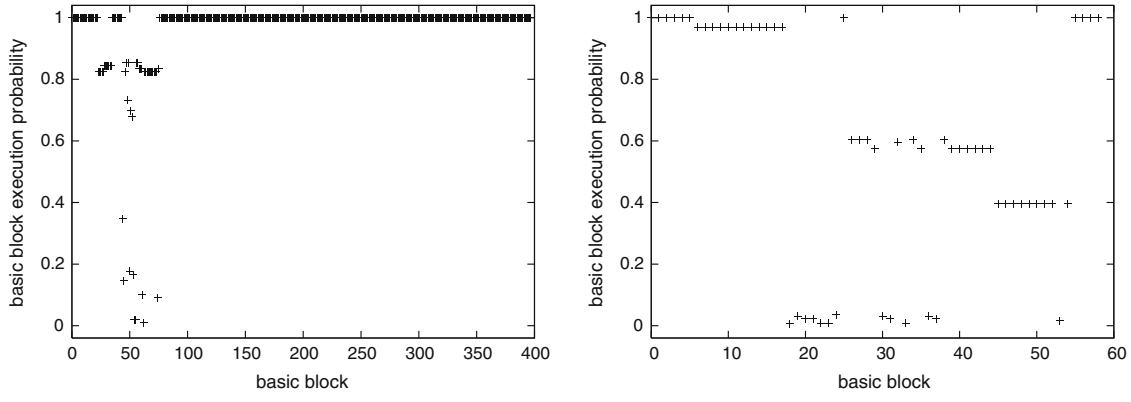
5.3.3. Basic block coverage

The observation that some pieces of code are executed in 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



(a) IPv4-radix (b) Flow Classification

Fig. 4. Detailed packet processing of single packet. The *y*-axis shows unique instruction addresses. Overlaps indicate loops.



(a) IPv4-radix (b) Flow Classification

Fig. 5. Basic block access frequency. A probability of 1 indicates the basic block is executed for every packet.

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.

Fig. 6 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.

5.3.4. Memory access sequence

Memory access patterns while processing a single packet are shown in Fig. 7, for 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, both packet data and program state are accessed continuously. This kind of insight into application behavior can be useful for optimizing applications and designing efficient memory hierarchies for network systems.

6. Analysis of network processing delay

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. In this section, we present how we have used the results obtained from PacketBench to obtain an analytical model for estimating network processing delay.

A packet traversing the network incurs the following delays: (1) transmission delay (the time it takes to send the packet onto the wire), (2) propagation delay (the time it takes to transmit the packet via the wire), (3) processing delay (the time it takes to handle the packet on the network system), and (4) queuing delay (the time the packet is buffered before it can be sent). In most cases, the key contributors of delay are (2) and (4) and are therefore considered in simulations and measurements. The transmission delay (1) is usually small for fast links and small packets and is therefore not considered. Traditionally, the processing delay (3) has also been negligible. However, this is not the case anymore as packet processing on routers becomes more complex. Our measurements and simulations have shown that packet processing can take considerable time especially when payload modifications are involved. This processing cost needs to be considered in network simulations to provide results that are a closer match to real measurements.

Using the processing complexity and memory access characteristics shown in Section 5, it is possible to derive an analytic model to estimate the processing delay of a packet given an application. This analytic model was originally presented in our previous work [46] and is briefly discussed here. This work extends [46] by presenting model parameters for a wider range of applications.

6.1. Analytical model

Our analytic model describes processing cost as a function of a few parameters and is based on results that can be easily derived from PacketBench. To estimate processing cost, we use two parameters, α_a and β_a, which are specific to each network processing application a:

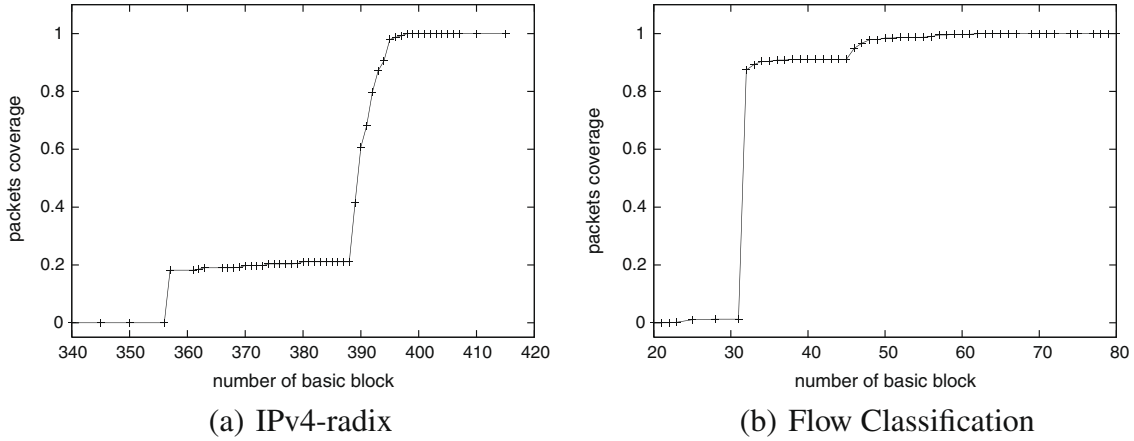


Fig. 6. Packet coverage. The coverage indicates what fraction of the packet trace can be processed with a given number of basic blocks.

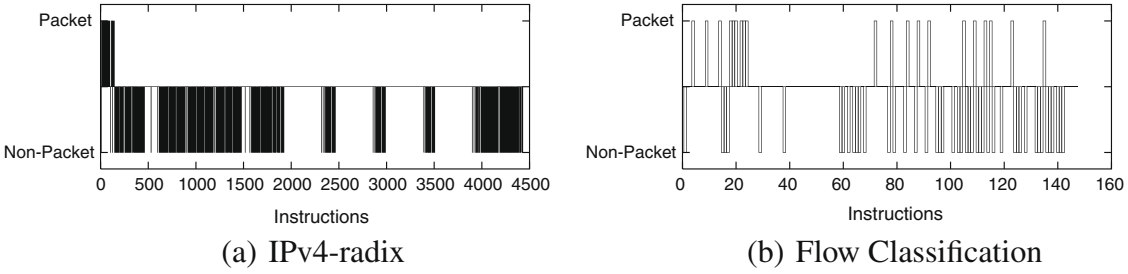


Fig. 7. Data memory access pattern. The graph indicates the memory regions to which accesses are made.

- **Per-packet processing cost** α_a**.** This parameter reflects the instructions that need to be executed for each packet independent of its size.
- **Per-byte processing cost** β_a**.** This parameter reflects the processing cost that depends on the packet size.

The total instructions processed, $i_{a,l}$, by an application a for a packet payload of length l can then be approximated by

$$i_a(l) = \alpha_a + \beta_a \cdot l. \quad (1)$$

Using PacketBench simulation results for the eight applications considered here, we obtain the parameters shown in Table 8. Similarly, we use two parameters to estimate the memory access cost for each application a.

- **Per-packet memory accesses** γ_a.
- **Per-byte memory accesses** δ_a.

The total number of memory accesses for an application a and a packet payload of size l is

$$m_a(l) = \gamma_a + \delta_a \cdot l. \quad (2)$$

The memory parameters that we derived from simulation are also shown in Table 8. In order to derive an overall processing delay as an expression of time, rather than instructions or memory accesses, it is necessary to consider the network system on which the processing is performed. The processing speed of the core and the memory access speed determine the overall processing time.

To capture the processing performance of a RISC core, we use the processor clock frequency, f. This metric is easily obtained and is a good approximation for processing speed. The main problem with using processor clock speeds as performance indicators is that the overall processing time also depends on the processor architecture and other system components. Since basically all network processors today use RISC processor cores, the architecture across systems is very similar. In normal operation, a RISC processor executes one instruction per clock cycle. This yields a processing time, $t_{p,a}$, for application a and a packet payload of length l of

$$t_{p,a}(l) = \frac{i_a(l)}{f}. \quad (3)$$

However, the processing performance of a RISC processor can be reduced due to pipeline stalls, which occur during memory accesses. There are also other causes for memory stalls, like control hazards, but the impact on the overall performance is less severe than stalls due to memory accesses and are therefore neglected. To integrate the effect of memory delay into our model, we assume an average memory access time of t_{mem} and determine the additional memory access delay, $t_{m,a}$, as

$$t_{m,a}(l) = m_a(l) \cdot t_{mem}. \quad (4)$$

Table 8
Application statistics for delay model.

Application, a	Type	Instructions		Memory	
		α_a	β_a	γ_a	δ_a
IPv4-radix	Header	4493	0	868	0
IPv4-trie	Header	205	0	50	0
Flow Classification	Header	153	0	79	0
TSA	Header	902	0	88	0
IPSec-AES	Payload	1272	61	591	36
IPSec-DES	Payload	3517	294	1212	104
String matching	Payload	433	11	155	4
Fingerprinting	Payload	52	78	16	16

The total packet processing time, t_a, is the sum of both delays:

$$t_a(l) = t_{p,a}(l) + t_{m,a}(l). \quad (5)$$

An example of these system parameters for a network processor (Intel IXP2400 [10]) are: $f_{IXP} = 600$ MHz and $t_{mem} = 4$–170 ns depending on the type of memory used (on-chip registers vs. off-chip SDRAM). The workload characteristics shown in Table 8 can also be used in other performance models of network processor systems [47].

6.2. Model verification

We present more detailed measurements on a network system that show the quantitative impact of processing delay. The results are compared to the simulation and model results from previous sections. We use a commercial off-the-shelf router for this measurement. We use the network setup shown in Fig. 8.

Traffic is sent from the source to the sink over 10 Mbps Ethernet. The VPN router is a Linksys BEFVP41 system. The measurement node operates both network interfaces in promiscuous mode and can therefore observe the packets that are transmitted on both sides of the VPN router using *tcpdump* [48]. Since the hubs do not buffer the packets, they do not introduce any delay between the VPN router and the measurement node.

The delay from the VPN router is measured by timestamping packets on both sides of the router. The difference in the timestamp is the delay. It is crucial that both links are measured by the same computer so that differences in system clocks do not bias the measurement.

The traffic that is sent for the delay measurement is a stream of UDP packets of varying size at a low data rate (a few kbps). For the round trip time measurement, a TCP connection is used. This keeps queues empty and ensures that we measure the processing delay on the router and not the queuing delay.

We consider two different cases in our measurement:

(1) Simple packet forwarding. In this case processing is limited to simple IP forwarding. The IPSec tunnel shown in Fig. 8 is not used.
(2) VPN termination. In this case all packets require cryptographic processing when moving into and out of the IPSec tunnel.

The results for both applications are shown in Fig. 9. The plot shows the processing time in microseconds over the packet size. We can observe that there is an increase in processing time with larger packets – as expected. However, even packet forwarding, which is a simple header processing application, shows this behavior. This is probably due to the fact that the Linksys BEFVP41 operates on the store-and-forward paradigm while handling packets. This causes larger packets to take additional time, which is not something that is captured in PacketBench. Nevertheless, the overall trend regarding increasing processing delay is clearly visible.

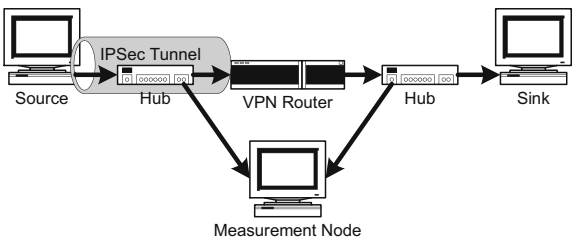


Fig. 8. Measurement setup. The VPN router is a Linksys BEFVP41 and all network connections are 10 Mbps Ethernet.

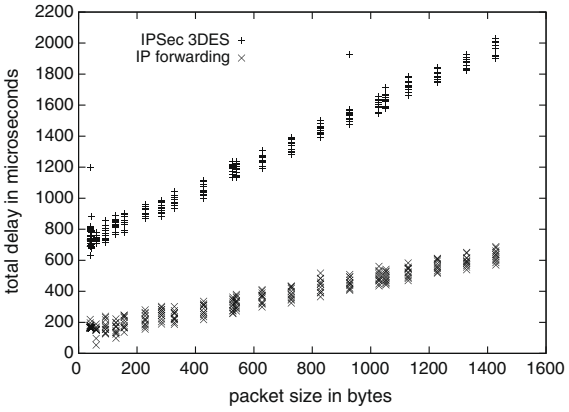


Fig. 9. Measurement results.

The model that we have developed for processing cost can be used easily in network simulations. We show this by adding an estimated delay as derived in Eq. (5) to a networking simulation and comparing the results to the measurements we have performed.

We integrated the delay model into the Network Simulator *ns-2* [49] and simulated the topology shown in Fig. 8 (excluding the measurement node). The metric that we are interested in is the round-trip time (RTT) for a TCP connection that traverses the VPN router. RTT is a good measure as it directly expresses the end-to-end delay and has a significant impact on the performance of TCP connections.

Fig. 10a shows the measured RTT for both IPSec and plain forwarding. The RTT for IPSec averages 56.2 ms and is about 12 ms larger than the 44.1 ms RTT for IP forwarding. This is an expected result and confirms the observations from Fig. 9.

Fig. 10b shows the *ns-2* simulation result for the same setup. The *x*-axis shows time instead of TCP sequence number due to the way *ns-2* reports RTT values. The average RTT for baseline IP forwarding is 42.0 ms and thus very close to the measured value of 44.1 ms. The RTT increases to 56.9 ms for IPSec. This value is also very close to the RTT observed in the measurement.

These results clearly show that:

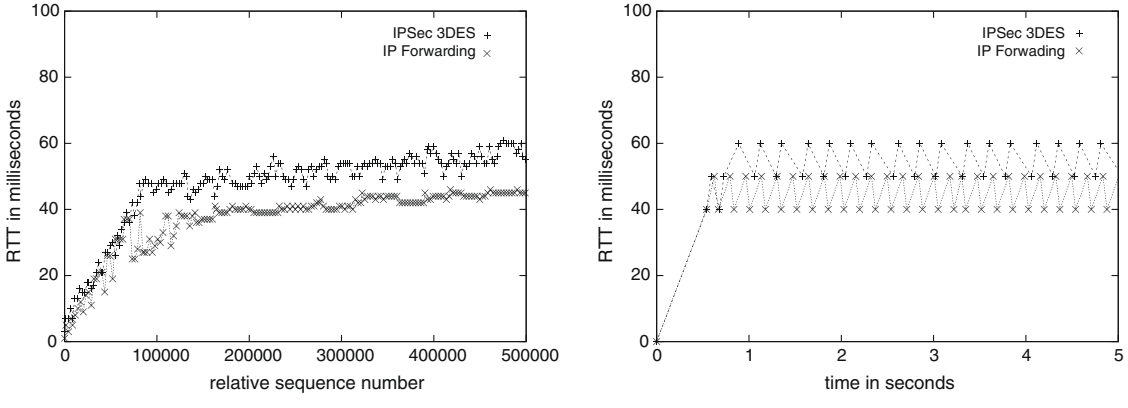
- The processing delay on a router has a direct impact on the performance of TCP connections. The processing delay increases the overall RTT and decreases the throughput.
- By extending the network simulator with our simple model for determining processing delay, we can achieve results that capture the measured network behavior.

On the average, a 2–5% error is introduced by using a simple linear model for the instruction counts and memory accesses. There are some limitations to this model that we wish to point out. First, the performance model in Section 6.1 is only one of many ways of approximating processing cost. Not all applications match the linear behavior that is observed in IP forwarding and IPSec. Examples of this are flows where processing is unevenly distributed among packets (e.g., HTTP load balancers or web switches [8] where most processing is performed on transmission of the initial URL). However, our model captures two key factors that are characteristic for packet processing: the per-packet delay, and the per-byte delay. Many applications process the headers, which incurs a fixed cost, and some process the payload, which incurs a packet length dependent cost. Therefore we expect that a large number of processing applications fall into the category that can be estimated by our model. Second, the derivation of system parameters is difficult for network systems where detailed specifications are not available. Also, the use of co-processors and other hardware accelerators leads to heterogeneous architectures that cannot be easily calibrated with a few metrics.

The processing cost model that we have derived is very simple and only requires two parameters. There is a tradeoff between simplicity and accuracy. We feel that it is important to derive a simple model that can easily be integrated into network simulations (as shown above). If the model requires a large number of parameters that are hard to derive and understand, it is unlikely that it will find broad usage. The results from Fig. 10a–b also show that even this simple model can improve the accuracy of simulations significantly.

7. 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 presented results for eight 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.,



(a) Measurement (b) Simulation

Fig. 10. RTT for TCP connection. The simulation, which uses the processing delay model, matches closely the measured network behavior.

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. We have also presented an analytical model for the estimation of network processing delay which uses workload characteristics derived from PacketBench. Use of this delay model improves the quality of network simulation results by providing accurate parameters. We believe that the availability of such a tool to analyze network processing workloads is an important aspect of designing and operating packet processing systems.

References

[1] W. Eatherton, The push of network processing to the top of the pyramid, in: Keynote Presentation at ACM/IEEE Symposium on Architectures for Networking and Communication Systems (ANCS), Princeton, NJ, 2005.
[2] A. Feldmann, Internet clean-slate design: what and why?, SIGCOMM Computer Communication Review 37 (3) (2007) 59–64.
[3] T. Wolf, Challenges and applications for network-processor-based programmable routers, in: Proceedings of the IEEE Sarnoff Symposium, Princeton, NJ, 2006.
[4] Standard Performance Evaluation Corporation, SPEC CPU2000 – Version 1.2, December 2001.
[5] D. Burger, T. Austin, The SimpleScalar tool set version 2.0, Computer Architecture News 25 (3) (1997) 13–25.
[6] J.C. Mogul, Simple and flexible datagram access controls for UNIX-based gateways, in: USENIX Conference Proceedings, Baltimore, MD, 1989, pp. 203–221.
[7] K.B. Egevang, P. Francis, The IP Network Address Translator (NAT), RFC 1631, Network Working Group, May 1994.
[8] G. Apostolopoulos, D. Aubespin, V. Peris, P. Pradhan, D. Saha, Design, implementation and performance of a content-based switch, in: Proceedings of the IEEE INFOCOM 2000, Tel Aviv, Israel, 2000, pp. 1117–1126.
[9] A.S. Snoeren, C. Partridge, L.A. Sanchez, C.E. Jones, F. Tchakountio, S.T. Kent, W. T. Strayer, Hash-based IP traceback, in: Proceedings of the ACM SIGCOMM 2001, San Diego, CA, 2001, pp. 3–14.
[10] Intel Corporation, Intel Second Generation Network Processor, 2002, <http://www.intel.com/design/network/products/npfamily/ixp2400.htm>.
[11] AMCC, np7300 10 Gbps Network Processor, 2006, <http://www.amcc.com>.
[12] Cisco Systems Inc., San Jose, CA, The Cisco QuantumFlow Processor: Cisco's Next Generation Network Processor, Febuary 2008.
[13] EZchip Technologies Ltd., Yokneam, Israel, NP-3 – 30-Gigabit Network Processor with Integrated Traffic Management, May 2007, <http://www.ezchip.com/>.
[14] P. Crowley, M.E. Fiuczynski, J.-L. Baer, B.N. Bershad, Characterizing processor architectures for programmable network interfaces, in: Proceedings of the 2000 International Conference on Supercomputing, Santa Fe, NM, 2000, pp. 54–65.
[15] P. Crowley, J.-L. Baer, A modelling framework for network processor systems, in: Proceedings of the First Network Processor Workshop (NP-1) in Conjunction with Eighth IEEE International Symposium on High Performance Computer Architecture (HPCA-8), Cambridge, MA, 2002, pp. 86–96.
[16] L. Thiele, S. Chakraborty, M. Gries, S. Künzli, Design space exploration of network processor architectures, in: Proceedings of the First Network Processor Workshop (NP-1) in Conjunction with Eighth IEEE International Symposium on High Performance Computer Architecture (HPCA-8), Cambridge, MA, 2002, pp. 30–41.
[17] T. Wolf, M. Franklin, Performance models for network processor design, IEEE Transactions on Parallel and Distributed Systems 17 (6) (2006) 548–561.
[18] P. Crowley, M. E. Fiuczynski, J.-L. Baer, B. N. Bershad, Workloads for programmable network interfaces, in: IEEE Second Annual Workshop on Workload Characterization, Austin, TX, 1999.
[19] T. Wolf, M. A. Franklin, CommBench – a telecommunications benchmark for network processors, in: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Austin, TX, 2000, pp. 154–162.
[20] G. Memik, W.H. Mangione-Smith, W. Hu, NetBench: a benchmarking suite for network processors, in: Proceedings of the International Conference on Computer-Aided Design, San Jose, CA, 2001, pp. 39–42.
[21] B.K. Lee, L.K. John, NpBench: a benchmark suite for control plane and data plane applications for network processors, in: Proceedings of the IEEE International Conference on Computer Design (ICCD '03), San Jose, CA, 2003, pp. 226–233.
[22] Embedded Microprocessor Benchmark Consortium, <http://www.eembc.org>.
[23] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, R.B. Brown, MiBench: a free, commercially representative embedded benchmark suite, in: Proceedings of the IEEE 4th Annual Workshop on Workload Characterization, Austin, TX, 2001.
[24] Y. Luo, J. Yang, L.N. Bhuyan, L. Zhao, NePSim: a network processor simulator with a power evaluation framework, IEEE Micro 24 (5) (2004) 34–44.
[25] S.D. Goglin, D. Hooper, A. Kumar, R. Yavatkar, Advanced software framework, tools, and languages for the IXP family, Intel Technology Journal 7 (4) (2003) 64–76.
[26] N. Weng, T. Wolf, Analytic modeling of network processors for parallel workload mapping, ACM Transactions on Embedded Computing Systems 8 (3) (2009) 1–29.
[27] Tensilica Inc., Santa Clara, CA, Xtensa LX2 Product Brief, April 2007.
[28] CoWare Inc., San Jose, CA, CoWare Processor Designer, 2006.
[29] ARM Ltd., ARM7 Datasheet, 2003.
[30] TCPDUMP, TCPDUMP Public Repository, 2003 <http://www.tcpdump.org>.
[31] National Laboratory for Applied Network Research – Passive Measurement and Analysis, Passive Measurement and Analysis, 2003, http://pma.nlanr.net/PMA/.
[32] F. Baker, Requirements for IP version 4 routers, RFC 1812, Network Working Group, June 1995.
[33] NetBSD Project, NetBSD release 1.3.1, <http://www.netbsd.org/>.
[34] S. Nilsson, G. Karlsson, IP-address lookup using LC-tries, IEEE Journal on Selected Areas in Communications 17 (6) (1999) 1083–1092.
[35] R. Ramaswamy, T. Wolf, High-speed prefix-preserving IP address anonymization for passive measurement systems, IEEE/ACM Transactions on Networking 15 (1) (2007) 26–39.
[36] J. Xu, J. Fan, M.H. Ammar, S.B. Moon, Prefix-preserving IP address anonymization: measurement-based security evaluation and a new cryptography-based scheme, in: Proceedings of the 10th IEEE International Conference on Network Protocols (ICNP'02), Paris, France, 2002, pp. 280–289.
[37] S. Kent, R. Atkinson, IP Encapsulating Security Payload (ESP), RFC 2406, Network Working Group, November 1998.
[38] National Institute of Standards and Technology, Advanced Encryption Standard (AES), fIPS 197, November 2001.
[39] National Institute of Standards and Technology, Data Encryption Standard (DES), fIPS 46-3, October 1999.
[40] Snort, The Open Source Network Intrusion Detection System, 2004, <http://www.snort.org>.
[41] M.O. Rabin, Fingerprinting by Random Polynomials, Tech. Rep. TR-15-81, Harvard University, Department of Computer Science, 1981.
[42] H.-A. Kim, B. Karp, Autograph: toward automated, distributed worm signature detection, in: Proceedings of the 13th Usenix Security Symposium (Security 2004), San Diego, CA, 2004.
[43] S. Singh, C. Estan, G. Varghese, S. Savage, Automated worm fingerprinting, in: Proceedings of the ACM/USENIX Symposium on Operating System Design and Implementation, San Francisco, CA, 2004.
[44] N.T. Spring, D. Wetherall, A protocol-independent technique for eliminating redundant network traffic, SIGCOMM Computer Communication Review 30 (4) (2000) 87–95.
[45] Network Processor Forum, Benchmarking Implementation Agreements, 2003, <http://www.npforum.org/benchmarking/bia.shtml>.
[46] R. Ramaswamy, N. Weng, T. Wolf, Characterizing network processing delay, in: Proceedings of the IEEE Global Communications Conference (GLOBECOM), Dallas, TX, 2004, pp. 1629–1634.
[47] M.A. Franklin, T. Wolf, A network processor performance and design model with benchmark parameterization, in: Proceedings of the First Network Processor Workshop (NP-1) in Conjunction with Eighth IEEE International Symposium on High Performance Computer Architecture (HPCA-8), Cambridge, MA, 2002, pp. 63–74.
[48] S. McCanne, V. Jacobson, The BSD packet filter: a new architecture for user-level packet capture, in: Proceedings of the USENIX Technical Conference, San Diego, CA, 1993, pp. 259–270.
[49] LBNL, Xerox PARC, UCB, and USC/ISI, The Network Simulator – ns-2, <http://www.isi.edu/nsnam/ns/>.

Ramaswamy Ramaswamy received the B.E. degree in computer science and engineering from the University of Madras, India, in 1999, and M.S. and Ph.D. degrees in computer engineering from the University of Massachusetts, Amherst, in 2001 and 2006, respectively. He is currently with Cisco Systems Inc. in San Jose, CA. His research interests include network systems design and network processor analysis.

Ning Weng received an M.S. degree in electrical and computer engineering from the University of Central Florida in 2000. He received a Ph.D. degree in electrical and computer engineering from the University of Massachusetts, Amherst in 2005. He is currently an assistant professor in the Department of Electrical and Computer Engineering at Southern Illinois University, Carbondale. His research interests are system integration, network processing system design, and network security.

Tilman Wolf is an associate professor in the Department of Electrical and Computer Engineering at the University of Massachusetts Amherst. He received his D.Sc. in computer science from Washington University in St. Louis in 2002. His research interests are next-generation Internet architecture, programmable routers, network processors, and embedded system security.