

A Case for Data Caching in Network Processors

Jayaram Mudigonda[†], Harrick M. Vin[†]
[†]Laboratory for Advanced Systems Research
University of Texas at Austin
{jram,vin}@cs.utexas.edu

Raj Yavatkar[‡]
[‡]Network Processor Division
Intel
raj.yavatkar@intel.com

Abstract: Today’s network processors (NPs) support mechanisms to *hide* long memory access latencies; however, they often do not support data caches that are effective in *reducing* average memory access latency. In this paper, we study a wide-range of packet processing applications and demonstrate that accesses to many data structures used in these applications exhibit considerable temporal locality; further, these accesses constitute a significant fraction of the total number of memory accesses made while processing a packet. Consequently, utilizing a cache for these data structures can (1) speedup packet processing, and (2) reduce the total off-chip memory bandwidth requirement considerably.

1 Introduction

Packet processing systems are designed to process network packets efficiently. The design of these systems is governed by two trends. First, over the past decade, the link bandwidth supported by these systems has doubled every year. Second, the diversity and complexity of applications supported by these systems have increased dramatically. Today, packet processing systems support a wide-range of *header-processing applications* such as network address translation (NAT) [20], protocol conversion (e.g., IPv4/v6 inter-operation gateway) [44] and firewall [3]; as well as *payload-processing applications* such as Secure Socket Layer (SSL) [22], intrusion detection [5], content-based load balancing, and virus scanning [5].

To meet simultaneously the demands of high-performance and flexibility, a new breed of programmable processors—referred to as *network processors (NPs)*, has emerged [18]. To achieve high packet throughput, NPs support several architectural features. For instance, most NPs support multiple processor cores to process packets concurrently. Further, each core supports hardware multi-threading; this enables NPs to hide memory access latencies by switching context from one hardware thread to another on memory accesses. However, unlike conventional general-purpose processors that rely extensively on caching to reduce average memory access latencies, NPs often do not support *data caching*; they expose the memory hierarchy to the programmers and expect programmers to *map* data structures to different levels of the memory hierarchy explicitly.

In this paper, we make a two-fold argument for supporting data caching in network processors.

Data caching can be beneficial : The lack of caching in NPs is often attributed to a hypothesis that packet processing systems are *required* to be configured with sufficient resources to meet the *worst-case* traffic demands; since caching can only improve the average-case (but not the worst-case), caches are not beneficial. We first argue that this hypothesis is false.

We observe that, today, 93.6% of NP-based systems are used in edge and enterprise networks [4]. These systems support complex applications—such as enterprise firewalls, virus-scanning, and storage-over-IP—that involve multiple packet types with different processing requirements. Further, the arrival rate of each packet type varies widely over time. Hence, provisioning such systems to meet the worst-case processing demands of all packet types is often pro-

hibitively expensive. Consequently, these systems are routinely provisioned to service only an *expected mix* of complex packet types, while ensuring that the worst-case processing requirements for only the basic IP-forwarding benchmark [13] are met. In such systems, data caching—if effective—can reduce the average time required to process each packet (which, in turn, can reduce the resource provisioning level). Further, by improving the efficiency of utilizing system resources (e.g., memory access bandwidth, hardware threads), caching enables a packet processing system to accommodate transient deviations from the expected traffic mix—thereby leading to system designs that are robust to traffic fluctuations.

Data caching is effective : Much of the prior research on illustrating the benefits of caching in the context of Internet applications has focused only on basic IP forwarding [13]. Further, much of this work exploits the frequent re-occurrence of IP addresses in packet traces by caching the *result* of the IP address lookup (as opposed to caching the data structures used to perform the lookup) [17, 21, 30, 36]. In this paper, we analyze the locality properties exhibited by a wide-range of data structures used in modern packet processing applications, and demonstrate that data caching can be highly effective. We argue that packet processing applications access two types of data structures: *packet data structures* and *application data structures*. Packet data structures (that include packet header, payload, and packet metadata) exhibit considerable spatial locality, but little temporal locality. On the other hand, application data structures (e.g., a trie used for route lookup, a hash table used for classifying packets as belonging to flows, a pattern table used by virus scanner, etc.) exhibit considerable temporal locality. We demonstrate that accesses to application data structures constitute a significant percentage of the non-stack memory accesses made while processing each packet. Consequently, utilizing a cache for application data structures is highly effective.

We demonstrate using two packet processing applications—a Quality of Service (QoS) router and a virus scanner—and several representative packet traces collected from the Internet [35] that a moderate-size (40KB for the applications and traces considered) cache can achieve high (80%) hit rate for application data structures in both single- and multi-threaded processor environments. We also demonstrate that: (1) the techniques for mapping data structures to different levels of a memory hierarchy used in today’s NPs require significantly larger amount of on-chip fast memory to match the performance of a system that uses data caching (e.g., for the virus scanner application and for the traces we consider, to match the performance of a system with 5KB cache, a mapping scheme requires over 500KB of on-chip fast memory); and (2) for the same amount of chip-area, a system with data caches outperforms solutions that only cache *results* of IP address lookups [17, 21, 30, 36]. Finally, we show that by using moderate size data caches, one can reduce packet processing times by 30-70% (depending on the miss penalty), and reduce off-chip memory bandwidth requirement by 80-90%.

The rest of the paper is organized as follows. In Section 2, we argue that support for data caching in network processors can be beneficial. We demonstrate the effectiveness of data caching and quan-

tify its benefits by studying the locality exhibited by a wide-range of packet processing applications (Sections 3 and 4). Section 5 discusses related work, and finally, Section 6 summarizes our contributions.

2 Is Caching Beneficial?

Traditionally, vendors of Internet routers (Layer-2 and Layer-3 packet processing systems) have advertised resource provisioning levels needed to meet the demands of the *worst-case traffic*. This has led to a commonly held belief that packet processing systems must be (and are) provisioned with sufficient processing resources to ensure that even a worst-case stream of packet arrivals can be serviced by the system without dropping any packets. Further, since caching can only improve the average-case (but not the worst-case), caches are not beneficial.

Worst-case provisioning advertised in conventional IP routers, however, is somewhat misleading. This is because, most vendors define worst-case provisioning by considering only the worst-case arrival pattern (namely, back-to-back arrival of smallest size packets at the line rate) of packets that request only the basic IP forwarding service [13]. The benchmarks defined by IETF and used by most vendors focus on this worst-case arrival pattern [8]. An IP router, in reality, processes a wide range of packet types (e.g., IP packets with options). Processing IP packets with options, for instance, takes considerably greater number of processor cycles than basic IP forwarding; provisioning sufficient processor resources in an IP router to service worst-case arrival pattern of packets that request IP options processing is prohibitively expensive. Thus, IP routers generally include sufficient resources to meet the processing demands only of the *expected traffic mix* (consisting of all the packet types that the router may receive), while ensuring that the worst-case processing requirements for the basic IP-forwarding benchmark are met. The well-known “Christmas Tree Packet” attack (in which every packet sent to a router requests IP options processing) has exposed the vulnerability of existing routers to such *worst-case* traffic mixes [38].

The practice of provisioning sufficient processor resources to meet the demands of *expected* traffic mix is even more pronounced in packet processing systems supporting complex applications (e.g., Secure Sockets Layer [22], Network Address Translation [20], firewalls [3], IPv4/IPv6 Interoperability [44], and TCP/IP header compression and decompression [29]). These systems are generally deployed in edge and enterprise networks, and constitute 93.6% of all of the NP-deployments today [4]. Most of these applications involve multiple types of packets; applications are specified as graphs of functions and the specific sequence of functions invoked for a packet depends on the packet’s type (determined based on the packet header and/or payload). For example, a Secure Socket Layer (SSL) [6] application processes three packet types—setup packets (that create per-flow state in the system), outgoing packets (that involve encryption), and incoming packets (that require decryption). In the IPv4/v6 interop system [2], an input packet can either be an IPv4 or an IPv6 packet; further, the processing requirement varies depending on whether the packet is an ICMP, TCP, or a UDP packet. The processing cycles (measured using the Performance Counters Library on a 930MHz, Intel® Pentium® III system) required to service different packet types for the SSL and IPv4/v6 interop applications are summarized in Tables 1 and 2.

As the tables indicate, modern packet processing applications often involve multiple packet types with significantly different processing times¹. Further, each packet type constitutes a reasonable percent-

¹Here, we want to emphasize the relative differences in packet processing times; the absolute values of packet processing times are platform-dependent.

| Packet type | Cycles |
|----------------------------|----------------|
| Setup | 20.913 million |
| Outgoing data (encryption) | 325 per byte. |
| Incoming data (decryption) | 325 per byte. |

Table 1: Processing requirements of SSL [6].

| Pkt. type | Cycles | Pkt. type | Cycles |
|------------------|--------|------------------|--------|
| IPv4 ICMP | 4317 | IPv6 ICMP | 1656 |
| IPv4 TCP FTP | 16950 | IPv6 TCP FTP | 9387 |
| IPv4 TCP (other) | 12541 | IPv6 TCP (other) | 2949 |
| IPv4 UDP DNS | 20430 | IPv6 UDP DNS | 9042 |
| IPv4 UDP (other) | 12346 | IPv6 UDP (other) | 2837 |

Table 2: Processing requirements for IPv4/v6 interop application [2]; packets carrying FTP and DNS payload require more processing than all other TCP and UDP packets, respectively.

age of the total traffic received by a packet processing system, and the arrival rate for different types of packets can vary widely over time. Hence, most of these systems are designed with sufficient resources to service an *expected* mix of packet types, while ensuring that the worst-case performance requirements for only the basic IP-forwarding benchmark are met.

In such systems, data caching—if effective—promises to reduce average memory access latency and the bandwidth contention for higher memory levels, and thereby reduce the time to process individual packets. The processing resources thus released enable the system to accommodate and service transient deviations from the expected traffic mix—thereby leading to system designs that are robust to traffic fluctuations. Further, since the amount of resources (e.g., processors, memory bandwidth) provisioned in the system is a function of the expected arrival rate and the time required to process packets of each type, reducing the average time to process individual packets reduces the resource provisioning levels (and hence reduces system cost). Thus, contrary to the popular belief, support for caching in network processors *can* offer significant benefits. In what follows, we derive the locality properties of packet processing applications, and thereby address the question: *are the potential benefits of supporting caching realizable?*

3 Experimental Methodology

Our objective is to characterize the locality properties of data accesses made by packet processing applications. Our methodology for deriving these characteristics involves three steps. First, we identify a set of *kernels* found in a wide-range of packet processing applications. Using these kernels, we construct two significant packet processing applications. Second, we utilize traffic traces and control data used by packet processing applications from today’s Internet; our approach ensures that the conclusions derived through our study are representative of what is observed in real deployments. Finally, we build tools that allow us to extract and analyze memory access profiles for individual data structures under representative traffic conditions in single- and multi-threaded processor environments.

3.1 Packet Processing Applications

Although the task of creating standard benchmarks for packet processing applications has received much attention lately [34, 45], there does not exist any publicly available suite of packet processing kernels. Hence, we identify a set of commonly used kernels and construct two significant real-world applications for our analysis.

3.1.1 Selection of Packet Processing Kernels

We select our packet processing kernels based on the following *semantic* characterization of packet processing applications. All packet processing applications perform the following four functions (in addition to the basic *receive* and *transmit* functionality): (1) verify the integrity of packets; (2) classify each packet as belonging to a flow; (3) process packets; and (4) determine the relative order of transmitting packets.

Integrity Verification Checksums are commonly used to verify integrity of packets. The two canonical implementations for checksum computation are IP-Checksum [14] and MD5 [39], which are used, respectively, to verify the integrity of packet header and payload.

Flow Classification Flow classification is the process of splitting a stream of incoming packets into sequences of *related* packets. Flow classification involves matching one or more fields (e.g., source and destination addresses and port numbers, protocol ID) contained in the packet against a set of rules that define a flow. Multi-field matching, in general, is quite complex; it may involve range, prefix, or exact matching on field values [24]. We consider a simpler but an important case of this multi-field matching problem wherein a flow is identified using an *exact* match on each of the fields. Further, we consider hash-based classification as a specific implementation of this exact matching. The hash-based matching scheme derives a hash on multiple packet header fields and uses it as the *FlowID*.

Packet Processing This involves accessing and updating packet header or payload. To cover a reasonable spectrum of packet processing functions, we include the following four kernels:

1. IP Forwarding [13]: This involves validating the IP source and destination addresses contained in the packet, determining the next-hop address (through route lookup), decrementing the time-to-live (TTL) field in the packet header, and processing any IP options marked in the packet header.
The route table is the most interesting data structure used by the IP forwarding kernel. Since IP addresses are hierarchically allocated in the Internet, route tables generally maintain next-hop information for IP address prefixes (that represent a collection of hosts with the same IP address prefix). Hence, route lookup involves determining the *longest-prefix match (LPM)* in the routing table for the destination host IP address. Several trie-based schemes [40] proposed in the literature are well-suited for this function. For our experiments, we include the uni-bit trie implementation for route lookup extracted from the BSD kernel.
2. Metering [26]: This involves maintaining accounting information for each flow. We consider a simple implementation of metering where each FlowRecord contains two fields: packet count (number of packets sent on a flow) and byte count (amount of information transmitted on the flow). The FlowID derived during classification is used to access and update the FlowRecord.
3. CAST (encryption) [10]: Applications such as SSL involve encrypting and decrypting packet payload. CAST is the kernel that performs this task. CAST scans the packet payload in a perfect sequence and does not use any significant data structures.
4. Pattern matching [47]: A large number of payload processing applications (e.g., XML firewall, virus scan, etc.) involve matching packet content against a set of pre-defined patterns. A pattern matcher requires two inputs: a set of patterns and the packet payload. For our experiments, we include the pattern matcher from Snort-2.0 [5]. The process of matching starts by looking for prefixes of patterns in the input text. If found, a hash

| Functionality | Kernel | Source |
|------------------------|------------------------|------------------|
| Integrity verification | IP-Checksum | Free BSD Project |
| | MD5 | R.S.A Inc. |
| Flow classification | Hash-based exact match | Self |
| Packet processing | IP forwarder | Free BSD Project |
| | Meter | Self |
| | CAST | SSLeay Lib |
| | Pattern matcher | Snort |
| Scheduler | DRR | Self |

Table 3: Packet Processing Kernels (Self = kernels we developed).

table, which is constructed while preprocessing patterns, is consulted to determine a set of candidate patterns. The exact match is then determined by using a reverse string match.

Whereas the IP forwarding and metering are examples of header-processing functionality, CAST and pattern matcher are instances of payload-processing functionality.

Scheduling Packet scheduling algorithms determine the relative order of transmitting packets on an outgoing network link and thereby provide delay, jitter, and throughput guarantees to flows. The literature contains descriptions of a wide-range of scheduling algorithms and efficient sorting techniques [32]. For our experiments, we select an implementation of the Deficit Round Robin (DRR) scheduling algorithm [42]. DRR is used in many commercial routers.

3.1.2 Building Applications from Kernels

Table 3 summarizes the selected packet processing kernels. Using these kernels, we construct the following two packet processing applications².

- *A Quality-of-Service (QoS) Router*: This application involves in-order (1) IP-Checksum computation; (2) hash-based 5-tuple flow classification (based on the source and destination IP addresses, port numbers, and protocol identifier); (3) Meter; (4) IP forwarder; and (5) the DRR scheduler.
- *A Virus Scanner*: This application involves (1) IP-Checksum computation; (2) pattern matcher for matching packet payload against a set of virus signatures; and (3) IP forwarder that forwards packets without any virus.

QoS router is a classic header-processing application, while virus scanner is a payload-processing application. QoS router is an application at the OSI Layer3-4 (deployed in the core and edge networks), whereas virus scanner represents Layer 5 and higher functionality (generally deployed in edge networks).

3.2 Packet Traces and Control Data

To collect a profile of data structure accesses, we execute our kernels and applications with three types of inputs: *packet traces* (representing realistic workload), a *route table*, and a set of *virus signatures*.

- **Packet Traces**: We utilize Internet packet traces collected by NLANR [35]. NLANR collects and publishes traces collected from several points in the Internet. For our study, we experimented with traces collected from two sites: ANL and MRA. The ANL trace is collected from an OC-3 (155 Mbits/second) link that connects an enterprise to its ISP; the MRA trace is collected from an OC-12 (620 Mbits/second) link that is closer to

²Most of the kernels have hundreds of lines of optimized C code (excluding comments); further, our applications have more than 1000 lines of C code.

the Internet backbone. Thus, the ANL traces represent traffic pattern at the edge of the network, while the MRA traces represent traffic in the network core. All the traces are of 90 seconds duration³, and are in the `tsh` format. Each ANL trace contains about 0.5 million packets, while each MRA trace contains as many as 5 million packets.

- **Route Table:** To determine the the next-hop router address for each packet, we need a route table. We construct our route table using the data obtained from the *RouteViews* project [9], which publishes routing information collected from a large number of routers deployed in the Internet.
- **Content Signatures:** To provide a realistic set of patterns to the virus scanner, we utilize the database of *content signatures* published by Snort [5], an open-source network intrusion detection system deployed in the Internet. We chose 100 signatures that represent various viruses and content-based attacks observed in the Internet over the past few months.

To utilize these traces and control data for our experiments, we need to address two challenges.

First, the packet traces published by NLANR are *anonymized*; to maintain anonymity, NLANR substitutes the source and the destination IP addresses contained in each packet by addresses selected from a well-known range. The anonymization process retains traffic patterns and flow information; in particular, once a source/destination IP address *A* is substituted by address *B* in a packet, then the occurrence of IP address *A* in any subsequent packet within the trace is also substituted by address *B*. An unfortunate side-effect of this anonymization process is that the source and destination IP addresses contained in these packet traces do not match any IP address prefixes maintained in the route table we collect from *RouteViews*. Hence, prior to using these traces in conjunction with our route table, we need to de-anonymize these packet traces. In particular, we substitute every occurrence of IP address *B* in the trace with IP address *C*, a randomly selected address for which the route table contains a prefix. This de-anonymization process, just as the original anonymization process, preserves the traffic patterns and flow characteristics.

Second, for the virus scan application, the memory access profile depends not only on the signatures, but also on the packet content. Unfortunately, for privacy reasons, the traces available in the public domain do not contain valid packet content. Hence, for our experiments, we supply random content for each packet sent to the virus scanner. By doing so, we attempt to characterize memory accesses in the case where the packet content does not match the signatures maintained in the database. Note that for a virus scanner, this is, in fact, the common case—a virus scanner is expected to detect packets that match any of the signatures in the database relatively infrequently (e.g., once in several million packets).

3.3 Profiling Data Structure Accesses

We characterize the locality properties of data structures defined in packet processing applications in two steps. First, we derive a data structure access trace in a single-threaded processor environment. Second, we utilize the trace to measure cache hit rates in a single-threaded and a multi-threaded, multi-processor environment (that better represents today’s NPs).

3.3.1 Single-threaded Processor Environment

To profile data structure accesses in a single-threaded processor environment, we use *SimpleScalar* [7] as our simulation environment.

³Several Internet measurement studies showed that 90% of the traffic consists of short-lived (a few seconds) TCP flows [1]. In our experiments, the hit-rates reach steady-state for traces longer than 50-seconds.

We use the simplest of the CPU simulators—namely, *sim-safe*—supported in *SimpleScalar*. Using *sim-safe*, it is easy to create a trace of all the memory accesses made by the application; however, at the time we began this study, the *SimpleScalar* environment did not support any mechanism to associate memory accesses to individual data structures defined in the application⁴.

In our context, the design of a tool for profiling data structure accesses is challenging because of two reasons. First, since most packet processing applications allocate and deallocate memory dynamically, the profiling tool must keep track of the mapping between each memory location and the data structures to which it belongs. Second, to ensure that the data structure access trace we collect is not cluttered by the memory accesses resulting from the execution of various “administrative” parts of the applications (e.g., reading the next packet information from a trace file), the tool must support a mechanism by which the logging of data structure accesses can be turned on or off.

Observe that these requirements can be met if applications can provide *directives* to the simulator when a data structure is dynamically allocated/deallocated or when the logging needs to be turned on/off. We designed and implemented a generic mechanism—referred to as *Simcall*—that meets this requirement. The *Simcall* mechanism allows applications to *trap* the *SimpleScalar* simulator by accessing a special memory location, as well as pass any information to the simulator using the `simcall_struct` data structure. The `simcall_struct` contains a *directive code* and any *directive-specific data*. For each *directive code*, the simulator contains a *directive-handler*.

We illustrate our use of the *Simcall* interface using an example. Consider the problem of notifying the simulator of the mapping from a memory address to a data structure. To communicate this information, upon the dynamic allocation of the data structure, the application initializes a `simcall_struct` with the starting address and the length of the newly allocated segment, the data structure id and the appropriate directive code (`ADD_MEM_SEG`). It then writes the address of this `simcall_struct` into the trap variable. The simulator on being trapped retrieves the directive code and calls the handler function for `ADD_MEM_SEG`. The handler updates the mappings from memory addresses to data structures.

Our enhanced version of *sim-safe* produces a data structure access trace. The trace is partitioned into *blocks*, where each block represents the execution of a single packet. The block consists of (1) the arrival time (relative to the the first packet in the trace) of the packet; and (2) a sequence of entries capturing the memory accesses. Each entry contains the memory address, the number of bytes accessed, the data structure id, and the number of non-memory access instructions executed since the previous memory access.

Once a trace of accesses to data structures is collected, we use *cheetah*, a utility library included in *SimpleScalar* to derive cache hit rates under the LRU and the OPT cache replacement policies in a single-threaded processor environment.

3.3.2 Multi-threaded Processor Environment

Most network processors today support processor cores with multiple hardware threads. To evaluate the performance of caches in such environments, we designed a discrete-event simulator. Our discrete-event simulator models a processor with multiple hardware threads and a data cache shared among all the threads. Each thread is assumed to process one packet at a time. The discrete-event simulator takes as input the data structure trace collected using *sim-safe*. The packet arrival time stored in each block is used to generate a new packet arrival event. Further, the number of non-memory access instructions executed between successive memory access instructions

⁴More recently, a tool with similar capability has become available [11].

(stored with each trace entry) is used to simulate computation (and hence advance the simulation time). The memory address, number of bytes accessed, and the data structure id are used to perform a cache lookup. On a cache hit, the thread continues execution. On a cache miss⁵, the simulator switches context to a thread on the *ready* queue. Further, the simulator schedules a memory access completion event after *miss-penalty* cycles from the current time; on completion of the memory access, the blocked thread is placed on the *ready* queue. On completing the processing of a packet, the simulator assigns to the thread a new packet (if one is available) or adds the thread to the *idle* queue. Our discrete-event simulator captures several details—such as context switch overhead, memory contention and queuing, etc.—of a multi-threaded network processor (e.g., Intel’s IXP2800 [27]).

4 Experimental Results

Using the tools described in Section 3.3, we derived the data structure profiles for each of the packet processing kernels and our two applications—QoS router and virus scanner—under several ANL and MRA traffic traces. In Section 4.1, we analyze these profiles and demonstrate that a moderate size cache can achieve significant hit rates in both single-threaded and multi-threaded processor environments. In Section 4.2, we demonstrate that moderate size caches can reduce the average packet processing times and the off-chip memory bandwidth requirement significantly. Unless otherwise stated, we assume a fully-associative LRU cache with a line length of one-word.

4.1 Effectiveness of Caching

4.1.1 Analysis of Packet Processing Kernels

Packet processing kernels (and applications) access two types of data structures: (1) *packet data structures*—that include packet header, payload, and any packet-specific meta-data generated while processing packets; and (2) *application data structures*—that maintain flow- and application-specific data. Observe that packet data structures exist only while the packet is being processed; application data structures persist across the processing of multiple packets.

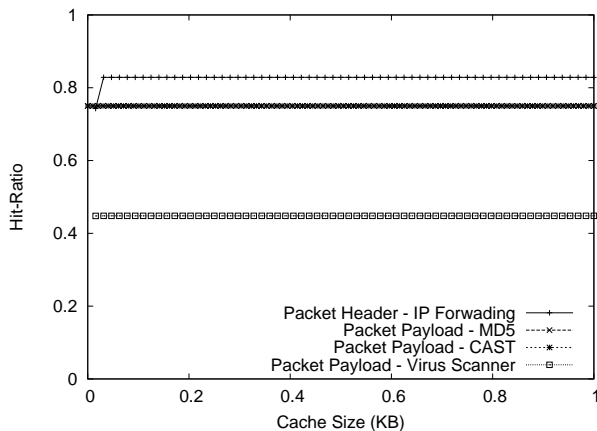


Figure 1: Locality of packet data structures

Figure 1 depicts the locality properties of the packet data structures accessed by various kernels. It demonstrates that the hit rate for packet data structures is *independent* of the cache size. This indicates that packet data structures exhibit mostly spatial locality, but little or no temporal locality. MD5 and CAST perform a sequential

⁵This is a simplified model; many NPs allow applications to issue multiple memory accesses prior to switching context.

scan (a byte at a time) of the packet payload; since for this experiment, we have assumed cache line size to be one word (4 bytes), the hit rate for both MD5 and CAST is 0.75. IP forwarder, on the other hand, performs several functions on the packet header, including verifying the integrity of the header, accessing the destination IP address for route lookup, verifying the IP version number, reducing the TTL, etc. Thus, accesses to the packet header made by the IP forwarder exhibit both spatial and temporal locality of access, and hence the hit rate observed is higher than MD5 and CAST. Finally, the pattern matcher, on realizing that the part of the payload accessed thus far does not match any of the patterns, attempts to minimize unnecessary search and memory accesses by skipping over parts of the payload. Although this optimization reduces the number of memory accesses, it also reduces the spatial locality of access.

Figure 2 depicts the locality properties of the application data structures used in our kernels. We make the following observations.

1. All of the application data structures used in our kernels exhibit considerable amount of temporal locality. Further, relatively small cache sizes (5-10KB) are sufficient to achieve the peak hit rates for all of the data structures.
2. For both the ANL and the MRA traces, the pattern table yields the same hit rate. This is because, the pattern table is accessed for every packet in the trace. For a given cache size, the hit rate observed for this data structure is dependent only on the patterns maintained in the pattern table and the content of the packets.
3. For the hash table, meter, routing table, and the DRR data structures, lower hit rates are observed with MRA traces than ANL traces. This is because of two reasons. First, unlike the pattern table, the temporal reuse of these data structures results primarily when multiple packets belonging to the *same* flow are processed. Consider, for instance, the hash table; a specific bucket in the hash table is accessed only when the application receives a packet with a flowID that matches the bucket ID. Second, recall that MRA traces are collected from a link closer to the core of the Internet, while ANL traces are collected at the edge of the network. The MRA trace supports a much larger number of flows simultaneously (and aggregates traffic from a larger number of sources). Hence, two packets belonging to the same flow are separated by a larger number of packets of other flows.

From the above discussion, we conclude that the packet data structures and the application data structures exhibit considerably different locality properties. To prevent packet data structures from interfering with the locality properties of application data structures, the two types of data structures must be managed *separately*. Whereas packet data structures can be managed using pre-fetching [28] or using stream buffers [15, 33], application data structures can benefit from caching. For the remainder of this paper, we will assume that packet and application data structures are, in fact, managed separately and hence they do not interfere with each other; further, we will focus only on application data structures.

4.1.2 Analysis of Packet Processing Applications

Tables 4 and 5 shows the sizes and the frequencies of accesses made to application data structures for QoS router and virus scanner. QoS router can be deployed in the network core or at the network edge; hence, we present results obtained using both MRA and ANL traces. Virus scanner, on the other hand, is often deployed at network edges; hence, we evaluate virus scanner using ANL traces. We make the following observations.

1. Stack accesses constitute a large percentage (58% in QoS router and 81% in virus scanner) of the total number of memory ac-

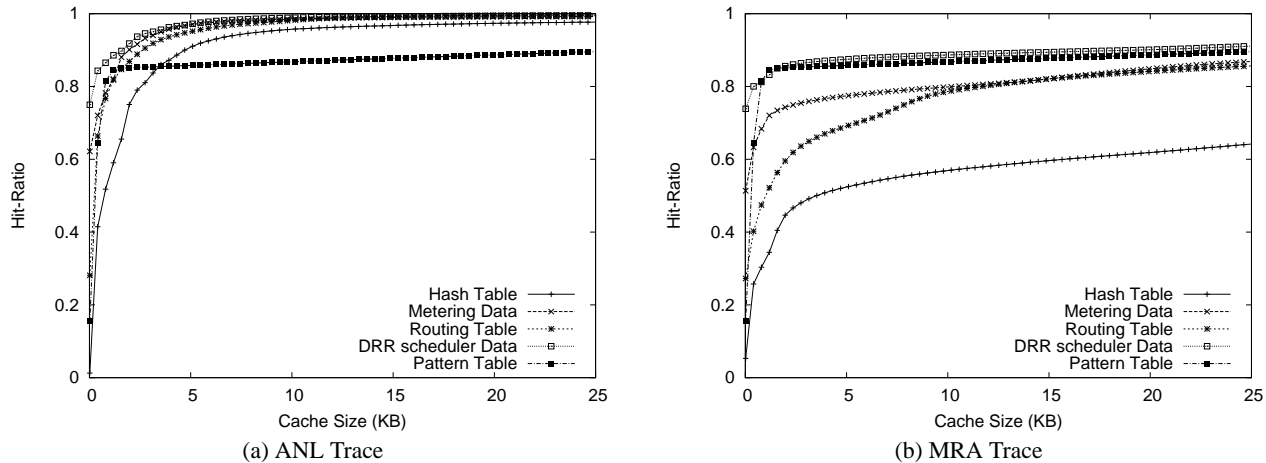


Figure 2: Locality properties of application data structures

| Data Structure | Size (bytes) | Refs/Pkt | % of accesses |
|----------------|--------------|----------|---------------|
| Pkt Hdr | 44 | 34.98 | 8.11 |
| Hash Table | 2915652 | 7.66 | 1.77 |
| Metering Data | 1599984 | 7.86 | 1.82 |
| Prefix Trie | 2974304 | 86.78 | 20.13 |
| DRR Data | 2569000 | 22.74 | 5.28 |
| Stack | 352 | 253.98 | 58.91 |
| Globals | 120 | 17.08 | 3.96 |

(a) QoS Router (MRA trace)

| Data Structure | Size | Refs/Pkt | % of accesses |
|----------------|--------|----------|---------------|
| Pkt Hdr | 44 | 34.99 | 8.77 |
| Hash Table | 395916 | 5.17 | 1.30 |
| Metering Data | 89184 | 8.00 | 2.00 |
| Prefix Trie | 220544 | 66.60 | 16.68 |
| DRR Data | 133776 | 23.00 | 5.76 |
| Stack | 280 | 241.44 | 60.47 |
| Globals | 116 | 17.05 | 4.27 |

(b) QoS Router (ANL trace)

Table 4: Data structure access frequencies for QoS Router

| Data Structure | Size (bytes) | Refs/Pkt | % of accesses |
|----------------|--------------|----------|---------------|
| Pkt Hdr | 44 | 22.00 | 1.05 |
| Prefix Trie | 220544 | 66.60 | 3.16 |
| Packet Payload | 1500 | 147.44 | 7.00 |
| Pattern Tables | 145852 | 152.40 | 7.24 |
| Stack | 188 | 1713.20 | 81.36 |
| Globals | 108 | 4.00 | 0.19 |

Virus Scanner (ANL trace)

Table 5: Data structure access frequencies for virus scanner

cesses performed per packet. Further, they exhibit considerable temporal locality. Hence, if included, they make caches look very attractive. In reality, however, stacks in these applications are relatively small (less than 400 bytes for both applications). In fact, compilers for today’s NPs allocate stack onto registers and fast local memory (e.g., 640-word local memory on IXP2800 [27]), and thereby eliminate memory access overhead for stack without requiring a cache. Thus, the case for caching should be made only based on locality properties of non-stack data. Hence, for the remainder of this paper, we will assume that stack accesses do not incur any overhead, and hence do not affect the packet processing throughput.

2. In the QoS router application, 29% of the accesses are to application data structures and only 8% of the accesses are to packet data structures. For the virus scanner application, about 10% of the accesses are to application data structures and 8% of the accesses are to packet data structures. Observe that our QoS router and virus scanner applications cover the spectrum of header- and payload-processing applications. For both classes of ap-

plications, accesses to application data structures constitute a significant portion of the non-stack accesses.

3. The sizes of application data structures are significantly larger than L1 caches supported in today’s processors. In the QoS router application, the application data structures—hash table, metering data, prefix trie, and DRR data—occupy 10MB of space, while for the virus scanner application, the application data—prefix trie and the pattern tables—occupy 365KB of space. Hence, it is infeasible to map these data structures in their entirety to local, fast memories (or L1 caches).

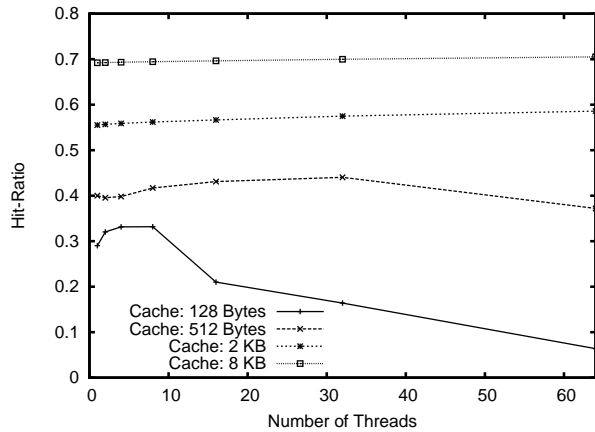
Figures 3 depicts the variation in the hit rates observed for the QoS router and the virus scanner applications as a function of cache size. For the QoS router application (with the MRA trace), a 40KB cache is sufficient to attain an 80% hit rate. With the ANL trace, QoS router and virus scanner can achieve an 80% hit rate with only a 2KB cache.

Throughout these experiments, we assumed a fully associative LRU cache with one-word line width. This choice was governed by our intent to study the fundamental locality properties exhibited by packet processing applications. We have repeated our experiments with caches supporting different levels of set-associativity and line widths; these experiments confirm that the hit rates we observed earlier can be realized using several practical cache configurations.

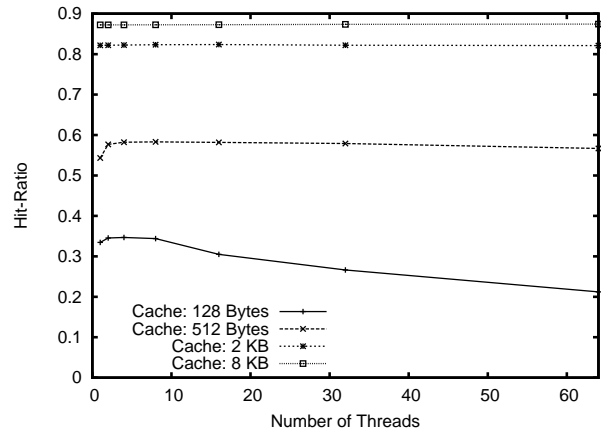
4.1.3 Multi-threaded processor

We evaluate the performance of caching in multi-threaded environments using our discrete-event simulator. Figure 4 plots the variation in hit rate with increase in the number of threads supported by a processor. We make the following observations.

1. Multi-threaded processor environments do not adversely affect the effectiveness of caching for packet processing applications;



(a) QoS Router (MRA trace)



(b) Virus Scanner (ANL trace)

Figure 4: Hit rates for the application data structures in multi-threaded processor environment

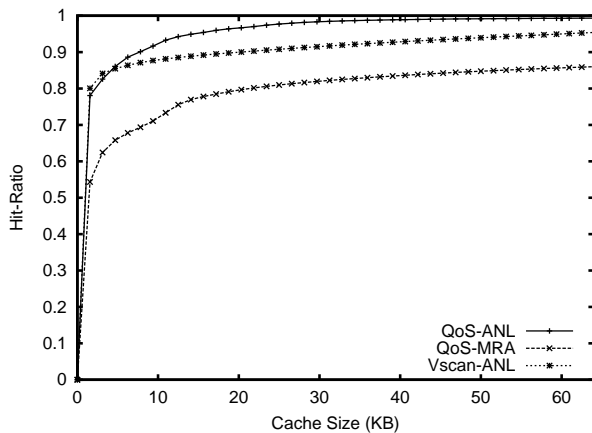


Figure 3: Cache hit rates observed for the QoS router and Virus scanner applications under MRA and ANL traces

moderate cache sizes continue to yield high hit rates for application data structures.

- At very small cache sizes ($< 1KB$), with increase in the number of threads, the hit rate first increases and then decreases. Increasing the number of threads supported by a processor overlaps the execution of greater number of packets. The resulting interleaving of memory accesses has two effects: (1) it increases hit rate because of improved inter-packet locality (resulting from temporal reuse of data structures across packets); and (2) it reduces the hit rate because of the increase in working set size. At small cache sizes, the hit rate improves because of increased inter-packet locality at first; but eventually increase in the working set sizes results in reduction in hit rates.
- For moderate cache sizes (2-10KB), increasing the number of threads has little impact on the hit rate. With moderate size caches, application data structures yield high hit rates (0.55-0.7 for the QoS router with the MRA trace, and 0.8-0.9 for the virus scanner with the ANL trace). At these hit rates, assuming a miss penalty of 50 cycles and a zero-cycle context-switch penalty, only a small number of threads are needed to achieve 100% utilization of the processor core (see Figure 5). Thus, increasing the number of threads per processor beyond this point has little impact on the hit rate.

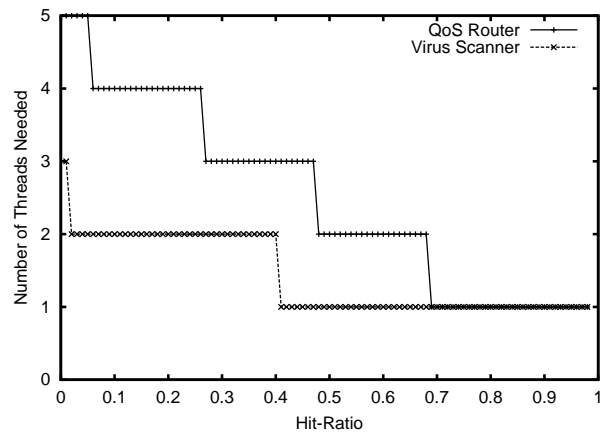


Figure 5: Variation in number of threads needed to achieve maximum processor utilization with hit rate

4.1.4 Mapping vs. Caching

Today's network processors expose the memory hierarchy to the programmers; programmers *map* data structures to different levels of the memory hierarchy explicitly. In this section, we compare the performance of such mapping techniques to caching.

To map a data structure to a certain level of the memory hierarchy at design time, the memory size must be at least as large as the maximum data structure size. Hence, with mapping, the hit rate (i.e., the percentage of memory accesses that can be serviced from fast memory) would change only when the fast memory size increases sufficiently to accommodate a new data structure. Hence, the hit rate changes in steps (where the length of the step corresponds to data structure size and the height of the step indicates the percentage of accesses made to the data structure). Figure 6 illustrates this behavior for both the QoS router and the virus scanner. It shows that to achieve a hit rate of 90% or higher, mapping requires two orders of magnitude greater amount of fast memory as compared to caching.

Observe that the above scheme of “entire” data structure mapping is essential for most of the application data structures (e.g., a hash table, metering data, and the pattern table). However, for data structures—such as the trie—it is possible to partition the data structure such that the most frequently accessed portion of the data structure (namely, the top-levels of the trie) can be placed in fast memory

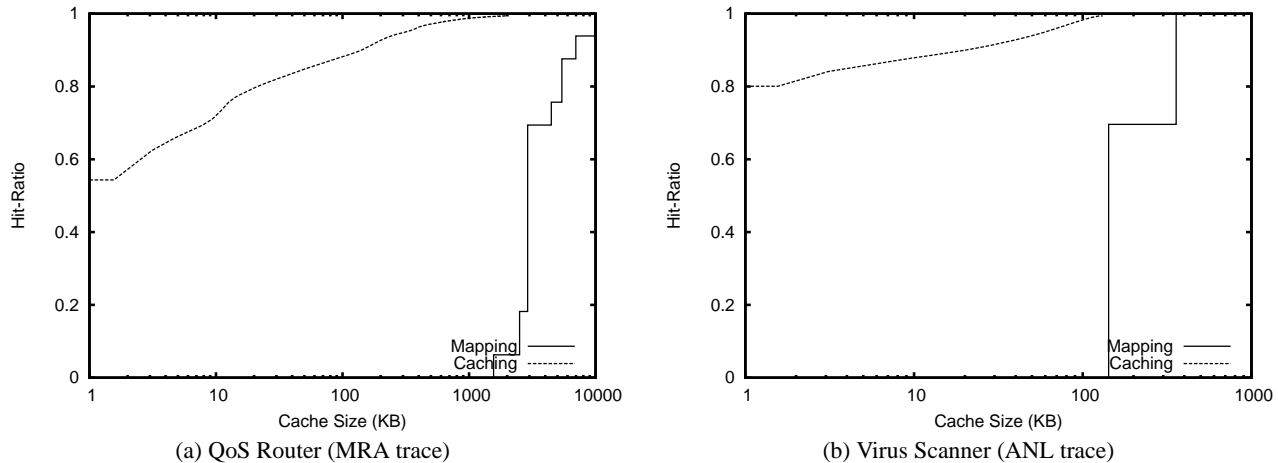


Figure 6: Comparison of mapping and caching techniques

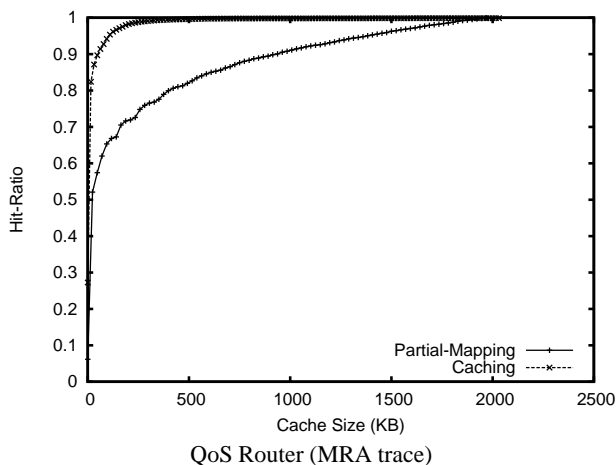


Figure 7: Comparison of partial mapping of trie with caching

and the remaining structure can be stored in a larger, slower memory [31, 12]. This approach attempts to approximate the behavior of a cache through static partitioning of the data structures. We evaluate the relative performance of such a mapping scheme with caching for the trie data structure. Figure 7 illustrates that this partial data structure mapping technique falls significantly short of the hit rate attained by caching.

4.2 Benefits of Caching

In Section 4.1, we have shown that (1) a significant fraction of the non-stack accesses are to application data structures, and (2) a moderate size cache is sufficient in achieving high hit rates (80% and higher) for these data structures. Thus, provisioning caches in NPs for use by application data structures offers the following benefits.

- By absorbing a significant percentage of accesses to application data structures as cache hits, an NP with appropriately provisioned cache can significantly reduce (by 30-70%) the time required to process each packet (see Figure 8). This, in turn, reduces the number of processors required to meet the packet throughput requirements. In particular, if λ is the rate at which packets arrive at a packet processing system, and if T is the average time required to service each packet, then the total num-

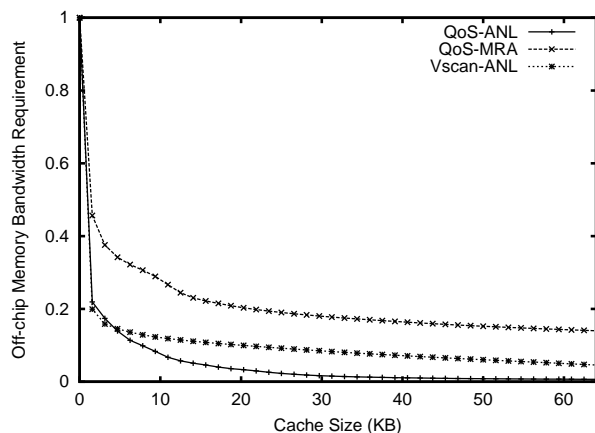


Figure 9: Normalized off-chip memory bandwidth requirement (with respect to a system without cache)

ber of processors N that the system should provision is given by $N * \frac{1}{T} = \lambda \Rightarrow N = \lambda T$. Thus, smaller the T , the smaller the required level of processor provisioning.

- An appropriately-provisioned cache for application data structures can reduce (by 80-90%) the off-chip memory bandwidth requirement (see Figure 9). This improves the scalability of packet processing systems.

5 Related Work

The literature contains several approaches for improving the throughput for accessing packet data structures (either by exploiting the sequential nature of accesses or by designing high-throughput pipelined memory subsystems) [25, 28, 41]. A significant body of research also addresses the problem of reducing the number of memory accesses performed for route lookup [40]. Route lookup schemes that exploit caches available in general purpose CPUs have been proposed in [16, 43]. A memory hierarchy optimized for a route lookup algorithm is proposed in [12]. A comprehensive survey of lookup algorithms can be found in [40]. These efforts are complementary to our work.

As for caching, several researchers have observed that, because of the frequent re-occurrence of IP addresses in packet traces, caching the *results* of previous route lookup operations can improve router

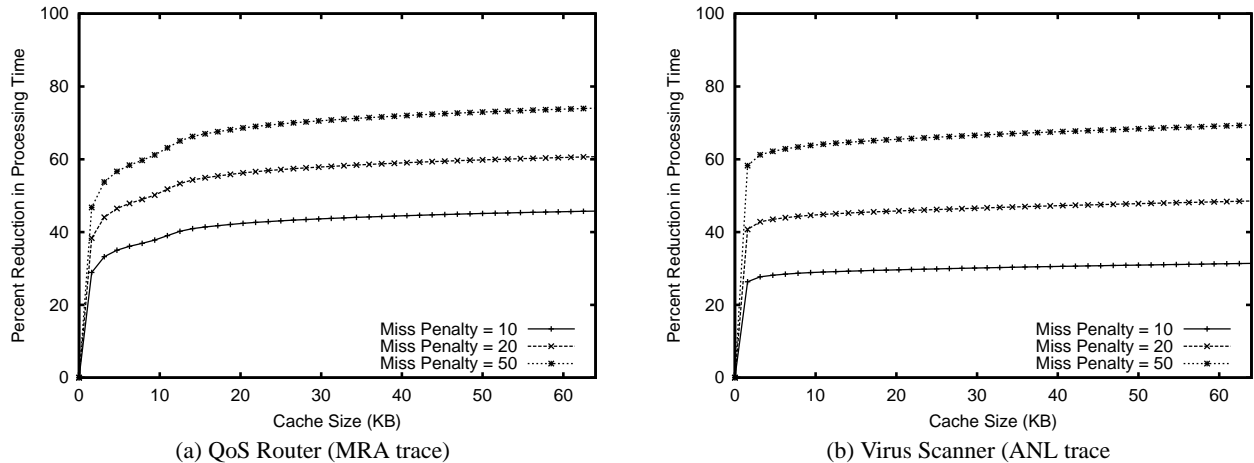


Figure 8: Reduction in average packet processing times (as compared a system without cache)

throughput [21, 30, 36, 37]. Special-purpose *result-cache* hardware has been proposed for the route lookup and a similar but more general problem of Layer-4 classification in [17] and [48] respectively. Techniques for improving performance of such result-caches have been proposed in [23]. Our study differs from all of these efforts in two ways. First, we evaluate the locality properties of data structures used in a wide-range of packet processing applications and kernels (not just route lookup). Second, in the context of route lookup, prior work has demonstrated the benefits of caching *results* of route lookup; instead, we evaluate the effectiveness and benefits of caching data structures (in particular, trie) used in performing route lookups. This is a subtle, and yet an important difference.

In Figure 10, we compare the reduction (with respect to a system without any type of a cache) in the average packet processing times observed in a system configured with a *result-cache* used only for caching route lookup results to a system that utilizes a data cache for route-table trie and other application data structures. It shows that, for the same cache size (in particular, the same chip-area devoted to implementing result-cache and data cache), the system with the data cache achieves a greater reduction in the average packet processing times. This demonstrates that, because of the inherent temporal locality of reference exhibited by application data structures, a data cache can outperform special-purpose result-caches.

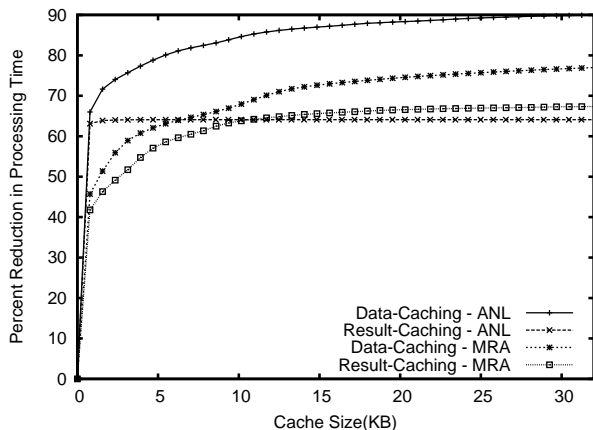


Figure 10: Comparison of average packet processing times in systems with a result cache and a data cache

Crowley et al. [19] evaluate data cache miss rates for a few packet

processing applications. Their primary focus is the comparison of processor architectures in exploiting the instruction-level and thread-level parallelism; they do not quantify the effectiveness of data caching. Wolf et al. [46] study instruction cache locality with the aim of avoiding cold instruction caches when assigning packets to processors. Baer et al. [12] study route lookup in the presence of packet traces with little or no temporal locality in the destination IP addresses; in this case, the top few levels of the trie are brought into the cache (much like the partial mapping scheme we studied in Section 4.1.4). Our study shows that with more realistic traces, the benefits of caching are even more pronounced.

CommBench[45] and NetBench [34] present a preliminary analysis of the memory access behavior of some benchmark applications. These studies, however, analyze the benchmark programs as a whole, without isolating data structures. As we have demonstrated, the miss rates for entire applications can be biased significantly with the inclusion of stack accesses.

To the best of our knowledge, we are the first to systematically explore and quantify the benefits data caching across a wide range of packet processing applications.

6 Concluding Remarks

Today’s network processors (NPs) support mechanisms to *hide* long memory access latencies; however, they often do not support data caches that are effective in *reducing* average memory access latency. In this paper, we make a two-fold argument for supporting data caches in network processors.

First, we argue that to support complex applications in the edge and enterprise networks, NP-based systems are provisioned with sufficient resources to process an *expected* traffic mix, while meeting the worst-case demands only for basic IP forwarding. In such systems, data caching—if effective—can reduce the average time to process packets (and hence the resource provisioning level) as well as lead to system designs that are robust to traffic fluctuations.

Second, we study the locality properties exhibited by a wide-range of packet processing applications. We show that packet processing applications access two types of data structures: packet data structures and application data structures. Whereas packet data structures exhibit considerable spatial locality but little temporal locality, application data structures exhibit considerable temporal locality. We demonstrate that application data structure accesses constitute a significant fraction of the total number of memory accesses. Consequently, utilizing a cache for application data structures can (1) re-

duce packet processing times, and (2) reduce the total off-chip memory bandwidth requirement considerably. We demonstrate using two packet processing applications—a Quality of Service (QoS) router and a virus scanner—and several traces collected from the Internet that a moderate-size (40KB for the applications and traces considered) cache can achieve high (80%) hit rate for application data structures. Further, it can reduce packet processing times by 30-70% (depending on the miss penalty), and reduce off-chip memory bandwidth requirement by 80-90%.

References

- [1] <http://www.caida.org/analysis/workload>.
- [2] Linux-based user-space NAT-PT. <http://www.ipv6.or.kr/english/natpt-overview.htm>.
- [3] Netfilter and iptables. <http://www.netfilter.org>.
- [4] The reincarnation of network processor market. In-Stat MDR, December 2003.
- [5] Snort: The Open Source Network Intrusion Detection System. <http://www.snort.org/>.
- [6] SSL: OpenSSL 0.9.6b. <http://www.openssl.org>.
- [7] The SimpleScalar Tool Set Version 3.0. <http://www.simplescalar.com/>.
- [8] The Tolly Group: Information Technology Testing, Research and Certification. <http://www.tolly.com>.
- [9] University of Oregon Route Views Project. <http://www.routeviews.org/>.
- [10] C. Adams and J. Gilchrist. The CAST-256 Encryption Algorithm. IETF RFC 2612.
- [11] K. K. Agaram, S. W. Keckler, C. Lin, and K. McKinley. Phase Analysis of Program Memory Behavior. Technical Report UTCS TR2002-67, University of Texas at Austin, 2002.
- [12] J.-L. Baer, D. Low, P. Crowley, and N. Sidhwany. Memory hierarchy design for a multiprocessor look-up engine. In *Proc. of the 12th International Conference on Parallel Architectures and Compilation Techniques*, 2003.
- [13] F. Baker. Requirements for IP Version 4 Routers. IETF RFC 1812, June 1995.
- [14] R. Braden, D. Borman, and C. Partridge. Computing the Internet Checksum. IETF RFC 1071, September 1988.
- [15] T. Chen and J. Baer. Effective Hardware-based Data Prefetching for High-performance Processors. *IEEE Transactions on Computers*, 44(5), May 1995.
- [16] T.-C. Chiueh and P. Pradhan. High-Performance IP Routing Table Lookup using CPU Caching. In *Proceedings of IEEE INFOCOMM'99*, 1999.
- [17] T.-C. Chiueh and P. Pradhan. Cache Memory Design for Network Processors. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture (HPCA)*, January 2000.
- [18] D. Comer. *Network Systems Design Using Network Processors*. Prentice Hall, ISBN 0-13-141792-4, 2002.
- [19] P. Crowley, M. E. Fiuczynski, and J.-L. Baer. Characterizing Processor Architectures for Programmable Network Interfaces. In *Proceedings of the 2000 International Conference on Supercomputing Santa Fe, N.M.*, May 2000.
- [20] K. Egevang and P. Francis. The IP Network Address Translator (NAT). IETF RFC 1631, May 1994.
- [21] D. C. Feldemeir. Improving Gateway Performance with a Routing-table Cache. In *Proceedings of IEEE INFOCOMM'88*, March 1988.
- [22] A. O. Freier, P. Karlton, and P. C. Kocher. The SSL Protocol Version 3.0. Internet Draft, November 1996.
- [23] K. Gopalan and T.-C. Chiueh. Improving Route Lookup Performance Using Network Processor Cache. In *IEEE/ACM SC2002 Conference*, November 2002.
- [24] P. Gupta and N. McKeown. Algorithms for Packet Classification. In *IEEE Network*, March/April 2001.
- [25] J. Hasan, S. Chandra, and T. N. Vijaykumar. Efficient Use of Memory Bandwidth to Improve Network Processor Throughput. In *Proceedings of the 30th International Symposium on Computer Architecture*, June 2003.
- [26] J. Heinanen, T. Finland, and R. Guerin. A Two Rate Three Color Marker. IETF RFC 2698.
- [27] *Intel IXP2800 Network Processor Hardware Reference Manual*, November 2002.
- [28] S. Iyer, R. R. Kompella, and N. McKeown. Techniques for Fast Packet Buffers. In *Gigabit Networking Workshop*, April 2001.
- [29] V. Jacobson. Compressing TCP/IP Headers for Low-speed Serial Links. IETF RFC 1144, February 1990.
- [30] R. Jain. Characteristics of Destination Address Locality in Computer Networks: A Comparison of Caching Schemes. *Computer Networks and ISDN Systems*, 18(4), May 1990.
- [31] E. J. Johnson and A. Kunze. *IXP1200 Programming*. Intel Press, 2002.
- [32] S. Keshav. *An Engineering Approach to Computer Networking*. Addison Wesley, 1997.
- [33] S. McKee, R. Klenke, K. Wright, W. Wulf, M. Salinas, J. Aylor, and A. Batson. Smarter Memory: Improving Bandwidth for Streamed References. *IEEE Computer*, July 1998.
- [34] G. Memik, W. H. Mangione-Smith, and W. Hu. NetBench: A Benchmarking Suite for Network Processors. In *ICCAD*, 2001.
- [35] NLANR Network Traffic Packet Header Traces. <http://pma.nlanr.net/Traces/>.
- [36] C. Partridge. A Fifty Gigabit Per Second IP Router. *IEEE/ACM Transactions on Networking*, 6(3), June 1998.
- [37] C. Partridge. Locality and Route Caches. In *NSF Workshop, Internet Statistics Measurement and Analysis*, February 1999.
- [38] X. Qie, R. Pang, and L. Peterson. Defensive Programming: Using an Annotation Toolkit to Build DoS-Resistant Software. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, December 2002.
- [39] R. Rivest. The MD5 Message-Digest Algorithm. IETF RFC 1321.
- [40] M. Ruiz-Sanchez, E. Biersack, and W. Dabbous. Survey and Taxonomy of IP Address Lookup Algorithms. *IEEE Network Magazine*, March 2001.
- [41] T. Sherwood, G. Varghese, and B. Calder. A Pipelined Memory Architecture for High Throughput Network Processors. In *30th International Symposium on Computer Architecture*, June 2003.
- [42] M. Shreedhar and G. Varghese. Efficient Fair Queuing Using Deficit Round Robin. In *Proceedings of ACM SIGCOMM*, August 1995.
- [43] V. Srinivasan and G. Varghese. Fast Address Lookups using Controlled Prefix Expansion. *ACM Transactions on Computer Systems*, 17(1), 1999.
- [44] G. Tsirtsis and P. Srisuresh. Network Address Translation - Protocol Translation (NAT-PT). IETF RFC 2766, February 2000.
- [45] T. Wolf and M. Franklin. CommBench - A Telecommunications Benchmark for Network Processors. In *IEEE International Symposium on Performance Analysis of Systems and Software*, April 2000.
- [46] T. Wolf and M. A. Franklin. Locality-aware Predictive Scheduling of Network Processors. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, November 2001.
- [47] S. Wu and U. Manber. A Fast Algorithm for Multi-pattern Searching. Technical Report TR-94-17, 1994.
- [48] J. Xu, M. Singhal, and J. Degroat. A Novel Cache Architecture to Support Layer-Four Packet Classification at Memory Access Speeds. In *IEEE INFOCOM*, 2000.