Overcoming the Memory Wall in Packet Processing: Hammers or Ladders?

Jayaram Mudigonda
Dept. of Computer Scienes
University of Texas at Austin
jram@cs.utexas.edu

Harrick M. Vin
Dept. of Computer Scienes
University of Texas at Austin
vin@cs.utexas.edu

Raj Yavatkar
Intel Corporation
raj.yavatkar@intel.com

ABSTRACT

Overhead of memory accesses limits the performance of packet processing applications. To overcome this bottleneck, today's network processors can utilize a wide-range of mechanisms—such as multi-level memory hierarchy, wide-word accesses, special-purpose result-caches, asynchronous memory, and hardware multi-threading. However, supporting all of these mechanisms complicates programmability and hardware design, and wastes system resources. In this paper, we address the following fundamental question: what *minimal* set of hardware mechanisms must a network processor support to achieve the twin goals of simplified programmability and high packet throughput? We show that no single mechanism suffices; the minimal set must include *data-caches* and *multi-threading*. Data-caches and multi-threading are complementary; whereas data-caches exploit locality to reduce the number of context-switches and the off-chip memory bandwidth requirement, multi-threading exploits parallelism to hide long cache-miss latencies.

Categories and Subject Descriptors: C.2.6 [Internetworking]: Routers C.2.6 [Internetworking]: Routers C.2.m [Computer-Communication Networks]: Miscellaneous C.5.m Comuter Systems Implementation Miscellaneous, D.4.4 [Communication management]: Network communication

General Terms: Measurement, Performance, Design.

Keywords: Network processors, Data-caches, Multithreading

1. INTRODUCTION

The gap between processor and memory performance—often referred to as the *memory wall* [39]—has been a major source of concern for all of computing; this problem is exacerbated in packet processing systems. Over the past decade, the link bandwidths supported by these systems have doubled every year; processor performance has doubled every 18 months (Moore's law); and memory latencies have improved only by about 10% a year. To put this argument in perspective, consider a system supporting OC-192 (or 10Gbps) links. In this case, minimum size SONET packets can arrive at the system every 32*ns*, which is roughly the latency for only a few memory accesses. Today, packet processing systems support a wide-range of *header-processing applications* such as network address translation (NAT), protocol conversion (e.g., IPv4/v6 interoperation gateway) and firewall; as well as *payload-processing applications* such as Secure Socket Layer (SSL), intrusion detection, content-based load balancing, and virus scanning. Many of these applications perform hundreds of memory accesses per packet.

To overcome this memory wall, *network processors (NPs)* [12], the building blocks of today's packet processing systems, support two types of mechanisms: (1) *hammers*—that exploit *locality* to reduce the overhead of memory accesses (i.e., *reduce the height* of the memory wall), and (2) *ladders*—that exploit *parallelism* to hide memory access latencies (i.e., *climb over* the memory wall). Hammers include such mechanisms as wide-word accesses [18, 31], special-purpose caches (e.g., route-caches) [11, 41], and multi-level memory hierarchy [19]. Ladders, on the other hand, include asynchronous (*non-blocking*) memory accesses and hardware multi-threading [19]. Wide-word accesses exploit spatial locality to reduce the number of memory accesses performed per packet. Special-purpose caches—such as route-caches—exploit the locality inherent in packet streams by caching and reusing results of repeated computations (e.g., route lookups); a hit in such *result-caches* eliminates the repeated computation and the corresponding memory accesses. Multi-level memory hierarchies exploit temporal locality of data accesses to reduce average latency of memory access. Asynchronous memory accesses and multi-threading, respectively, exploit the intra/inter-packet parallelism to overlap the execution of independent instructions/packets with memory accesses.

Observe that unlike general-purpose processors that rely primarily on *data-caches* (a *hammer*) to overcome the memory wall, NPs can utilize a much wider-range of mechanisms. This is facilitated by three characteristics of the packet processing domain. First, the domain offers significant packet-level parallelism. Second, network traffic, consisting of interleaved streams of related packets, exhibits locality [15, 21, 27, 28]. Finally, for packet processing systems, throughput—and not latency—is the primary optimization criterion.

Although the availability of wider-range of mechanisms offers greater hope in overcoming the memory wall, supporting all of these mechanisms in hardware without clear guidelines for their usage has two limitations. First, judicious use of these mechanisms is crucial for achieving high packet throughput; today, this task is often left to the programmers, which makes NPs very difficult to program. Second, provisioning, in hardware, mechanisms that yield only marginal benefits leads to unnecessary hardware complexity and wasted system resources (e.g., chip-area and memory bandwidth).

ANCS'05, October 26–28, 2005, Princeton, New Jersey, USA.

Much of the prior work on overcoming the memory wall in packet processing has focused on demonstrating the benefits of specific mechanisms and only for specific applications (e.g., IP forwarding) [10, 11, 34, 41]. For instance, the benefit of prefetching packet data is demonstrated in [18, 20]; a high-performance memory system based on wide-word accesses is described in [31]; special-purpose route-caches are proposed in [11, 41]; and the effectiveness of data-caches is shown in [24]. The literature, however, does not contain any comparative evaluation of these mechanisms or guidelines for their usage in a broader class of applications.

In this paper, we ask the following fundamental question: what *minimal* set of hardware mechanisms must a network processor support to achieve the twin goals of simplified programmability and high packet throughput? We conduct a thorough analysis of data accesses performed by a wide-range of modern packet processing applications and show that no single mechanism suffices; the minimal set must include data-caches and multi-threading. We demonstrate that:

- From the set of hammers, data-caches—generally not supported in today's NPs—outperform such mechanisms as wide-word accesses, special-purpose result-caches, and exposed multi-level memory hierarchy. However, because packet processing applications are memory-access-intensive, data-cache misses can cause significant processor stalls. Hence, data-caches alone are insufficient to achieve high throughput.

- Because of the significant packet-level parallelism available in workloads, hardware multi-threading is effective at hiding memory access latency; its effectiveness, however, is limited severely by constraints on off-chip memory bandwidth, context-switch overhead, and serialization required to protect access to shared read-write data.

- Data-caches and multi-threading are complementary. Data-caches exploit locality in the workload to reduce the number of context-switches and the off-chip memory bandwidth requirement. Multi-threading exploits parallelism in the workload to hide long cache-miss latencies and to reduce processor stalls. Hence, a hybrid architecture that supports data-caches and multi-threading is highly effective in achieving high throughput.

We also make two important observations. First, unlike general-purpose processors where data-caches are used to reduce average memory access latency, the primary benefit of supporting data-caches in network processors is in reducing *number of context-switches* and *off-chip memory bandwidth* requirement. Second, data-caches are transparent to programmers and compilers. The presence of data-caches also simplifies the usage of hardware multi-threading. In particular, in the presence of data-caches, NPs can simply switch context from one thread to another on a cache miss. This eliminates the need to schedule hardware threads explicitly (by the programmer or the compiler). In such a case, a programmer is *only* required to develop thread-safe applications suitable for execution in a multi-processor environment; the existence and usage of hardware multi-threading becomes transparent. Thus, a hybrid architecture that supports both data-caches and multi-threading in hardware simplifies programmability of NPs significantly.

The rest of the paper is organized as follows. In Section 2, we discuss our experimental methodology. We quantify the relative benefits of using hammers and ladders to overcome the memory wall and derive the minimal set of hardware mechanisms in Section 3. We discuss the implications of our findings in Section 4. Section 5 discusses related work, and finally, Section 6 summarizes our contributions.

2. METHODOLOGY

To compare the effectiveness of mechanisms for overcoming the memory wall, we analyze data accesses performed by a wide-range of packet processing applications. In what follows, we describe the set of applications, the traffic traces and control data used in our experiments, and the simulation environment.

2.1 Packet Processing Applications

We select packet processing applications based on the following *semantic* characterization. All packet processing systems perform the following four functions (in addition to the *receive* and *transmit* functions): (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 (i.e., scheduling). Whereas only a small number of implementation variants exist for integrity verification, classification, and scheduling, significant diversity exists for packet processing functions.

We consider two canonical integrity verification implementations based on the well-known IP-Checksum (RFC-1071) and MD5 (RFC-1321) algorithms. Classification, in its most general form may involve range, prefix or exact matching and hence can be very complex [17]. We consider a simpler but an important case wherein the five-tuple (source and destination addresses, ports and protocol) are hashed to find the *FlowID*. To determine the relative order for transmitting packets, we consider an implementation of the Deficit Round Robin (DRR) scheduling algorithm [32] used in many commercial routers.

To cover a reasonable spectrum of packet processing functions, we study several header- and payload-processing applications (see Table 1).

Header-processing applications:

1. *IP forwarding* (RFC-1812), which involves validating the header, decrementing the time-to-live (TTL) field, route lookup (longest-prefix match (LPM) for the destination IP address), and processing of any IP-options. For the LPM functionality, we consider four different implementations: (1) Patricia trie (denoted by `patricia`), (2) bitmap trie (`bitmap`), (3) binary search on prefix lengths (`bsol`), and (4) dual trie from IXA SDK (`ixp`).

   Many Free-BSD-based implementations use a uni-bit trie, referred to as *Patricia trie* [33]. Bitmap trie [14] is a variant of multi-bit trie that uses bitmaps to compress the child and data pointers in the trie nodes. This is used in many commercial routers. Unlike the other schemes, Binary search on prefix lengths [35] does not use a tree-based lookup data structure; instead, it uses a set of hash tables (one per prefix length) and does a binary search on the prefix lengths to find the longest prefix match. It has the best known average-case complexity. Finally, the reference design in the Intel®'s SDK for the IXP series of processors (IXA SDK 3.0) uses two multi-bit tries simultaneously and bounds both lookup and insertion cost.

2. *Metering*, which involves marking packets based on the packet and byte count of the flow (the FlowID derived during classification is used to access and update the FlowRecord). We consider three implementations: (1) `srtcm`: the Single-Rate-Three-Color Marker (RFC-2697), (2) `trtcm`: the Two-Rate-Three-Color Marker (RFC-2698), and (3) `tswtcm`: the Time-Sliding-Window-Three-Color Marker (RFC-2859).

Functionality	Application		Source
Integrity verification	IP-Checksum	`checksum`	Free BSD
	MD5	`md5`	R.S.A Inc.
Classification	Hash based	`classify`	UT
Prefix match	Patricia Trie	`patricia`	Free BSD
	Bitmap Trie	`bitmap`	UT
	Binary Search	`bsol`	UT
	IXA Dual Trie	`ixp`	IXA SDK 3.0
Metering	Sliding Window	`tswtcm`	UT
	Single Rate	`srtcm`	IXA SDK 3.0
	Two Rate	`trtcm`	IXA SDK 3.0
Header processing	Stream-4	`stream`	Snort 2.0
	PortScan	`portscan`	Snort 2.0
Payload processing	CAST	`cast`	SSLeay Lib
	Pattern matcher	`vscan`	Snort 2.0
Scheduler	DRR	`drr`	UT

Table 1: Packet Processing Applications (UT = We developed)

3. *Stream-4*: Stream-4 is a module of the Snort-2.0 intrusion detection system [5]. It emulates the TCP state machine for each ongoing session and reconstructs the byte stream out of the packets. It performs most of the the TCP processing generally performed in end-systems. The primary data structures are the per-session data and a session table that maintains pointers to individual session data items. It uses splay trees for the session table and also to keep track of the partially re-assembled byte streams.

4. *Portscan*: This is also a module of Snort [5]. It is used to detect suspicious port-scanning activity. The main data structures are splay trees. First, a splay tree that is keyed on the source address is looked up. This lookup yields a pointer to another splay tree that keeps track of the recent ports the source address in question has accessed. Port-scanner declares an attack if a source accesses too many ports during a short period.

Payload processing applications: We consider two applications: (1) `vscan`, *a pattern matcher* [38, 5] that matches packet content against a set of pre-defined patterns (e.g., virus signatures)—we consider the default and preferred pattern matching implementation [38] from Snort-2.0 [5]; and (2) CAST (RFC-2612), which is used to encrypt and decrypt the packet payload to implement Virtual Private Networks (VPN).

These applications subsume NP benchmarks [23, 36].

2.2 Packet Traces and Control Data

To study data accesses, we execute the applications with three inputs: *packet traces*, a *route table*, and *virus signatures*.

We use packet traces collected from four locations in the Internet. Three of them (ANL, FRG and MRA) are provided by NLANR [26] while the fourth one (UNC) is obtained from the University of North Carolina. The ANL trace represents traffic on a link that connects an enterprise (the Argonne National Lab) to its service provider. The UNC trace represents traffic on a link connecting a large university to its ISP. The FRG trace represents the aggregation of traffic from several universities in the Denver area to the high-speed Abilene network. Finally, MRA traces represent traffic on a link connecting Merit and Abilene—two large networks. The qualitative conclusions remain same across all these traces, with the ANL and MRA traces often forming the extremes quantitatively. Hence, for brevity, we present results only for the ANL and MRA traces. All the traces are of 90 seconds duration[1]. Each ANL trace (collected from an OC-3 (i.e., 155 Mbps) link) contains about 0.5 million packets while each MRA trace (collected from an OC-12 link) has about 5 million packets.

We construct our route table using the data obtained from the *RouteViews* project [8]. We utilize the database of *virus signatures* published by Snort [5].

Utilizing these traces and control data for our experiments presents two challenges. First, the publicly available packet traces are always *anonymized*. An unfortunate side-effect of this anonymization process is that IP addresses contained in these traces do not match any IP address prefixes available from the RouteViews. Hence, before using these traces with our route table, we *de-anonymize* them; we substitute every occurrence of IP address in the trace with another randomly selected address for which the route table contains a prefix. This de-anonymization process preserves the traffic pattern and flow characteristics; further, it conforms to the traffic generation guidelines recommended by the network processor forum [2]. Second, for the virus scan application, the memory access profile depends not only on the signatures, but also on the packet content. Unfortunately, none of the publicly available traces contain valid packet content. Hence, for our experiments, we consider two scenarios: (1) packets with random payload; and (2) packets with payload containing web pages from popular web sites (containing valid English text). By doing so, we characterize the *common case* where the packets do not match any signature in the database.

2.3 Simulation Environment

Single-threaded Processor Environment: To profile data accesses in a single-threaded processor environment, we execute our applications within the *SimpleScalar* [6] simulation environment. In particular, we use the `sim-safe` CPU simulator. `Sim-safe` simulates a very simple RISC instruction set similar to the ones supported by today's NPs. We enhanced `sim-safe` to produce an instruction execution trace that is partitioned into *blocks*, where each block represents the execution of a packet. Each block consists of (1) the arrival time (relative to the the first packet in the trace) of the packet; and (2) the sequence of all the ALU instructions, memory accesses, and mutex operations performed while processing the packet. Each memory access is described by the tuple: ⟨memory address, the number of bytes accessed, the data structure id⟩. Each mutex acquire/release event describes the identifier of the lock. We also use a utility based on the `cheetah` cache simulator library of the Simplescalar toolkit.

Multi-threaded Processor Environment: Most network processors today support processor cores with multiple hardware threads. To study such environments, we designed a discrete-event simulator. The simulator takes as input the instruction trace collected using our enhanced version of `sim-safe`. Upon scheduling a memory access, the simulator switches context to the thread at the head of the *ready* queue. When the memory access completes, 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; each thread is assumed to process one packet at a time. Our 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 [19]).

[1]About 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.

3. RESULTS

Packet processing applications access three types of data: (1) *packet-data*—that include packet header, payload, and any packet-specific meta-data such as packet arrival time and interfaceID; (2) *temporary-data*—that include data structures allocated on the stack; and (3) *application-data*—that refer to any persistent data structures used by the packet processing application (e.g., a route table, per-flow metering counters, and a virus signature database). In this paper, we consider the overhead resulting only from application-data accesses. This is because of three reasons [24].

- For all the applications, packet-data is relatively small in size (44bytes meta-data, and 704/736bytes of payload on an average for ANL/MRA traces, respectively). Further, packet-data is generally accessed sequentially. Hence, prefetching [18, 20] is effective in minimizing the overhead of accessing packet-data.
- Temporary-data is relatively small in size; for 12 out of 16 applications, temporary-data is smaller than 108bytes, with the maximum being only 496bytes. Further, accesses to temporary-data exhibit considerable temporal locality. Thus, by making use of registers, and where available, the small fast memories close to the processor [22] to store temporary-data, the overhead for temporary-data accesses can be eliminated.
- Application-data sizes are significantly larger than packet-data and temporary-data. For our applications, the application-data sizes range from 135KB in `bsol` to as high as 10MB in `portscan`. These sizes are significantly larger than the fast local-memories supported in today's network processors. Further, application-data accesses account for a large fraction of the total number of accesses in most applications; even payload processing applications, such as `vscan`, make far greater number of accesses to application-data than packet-data. In many cases, application-data accesses constitute as high as 90% of the non-temporary-data accesses.

Today's NPs rely on multiple processor cores to achieve high packet throughput. However, since most application-data are either read-only (e.g., route table) or flow-specific (e.g., metering counters), we ensure the coherence of application-data by pinning each flow to a processor core (i.e., by processing all packets of a flow on the same processor core).

In what follows, we study the effectiveness of reducing the overhead of application-data accesses using two types of mechanisms: (1) *hammers*—that exploit locality to reduce the overhead of memory accesses, and (2) *ladders*—that exploit parallelism to hide memory access latencies. We demonstrate that, to overcome the memory wall, network processors should support in hardware a *hammer* (namely, a *data-cache*) and a *ladder* (namely, *multi-threading*). No single mechanism in isolation suffices. Together, these two mechanisms enable network processors to achieve the twin goals of simplified programmability and high packet throughput.

3.1 Hammers: Reducing Overhead

Hammers exploit locality in the workload to do one of two things: (1) *Reduce the number of memory accesses*, or (2) *reduce average access latency*. We consider two mechanisms each as representatives of the two categories: *wide-word accesses* and *result-caches* for reducing the number of memory accesses, and *exposed multi-level memory hierarchy* and *data-caches* for reducing average latency. We first evaluate the effectiveness of each of these mechanisms in isolation, and then compare their relative benefits.

3.1.1 Wide-word Accesses

The mechanism for issuing wide-word accesses to reduce memory access overhead exploits two observations. First, for many applications, application-data is organized into semantic units that are larger than one word (4bytes). For instance, for the `patricia`, each trie node is 16bytes and when a trie node is accessed, most of the fields within the node are accessed. Second, wide-word accesses amortize fixed but significant parts of the access latency, such as the bus arbitration overhead, over a larger number of bytes. For instance, accessing a double-word requires only marginally more cycles than accessing a single-word, and substantially less than accessing two single-words in succession.

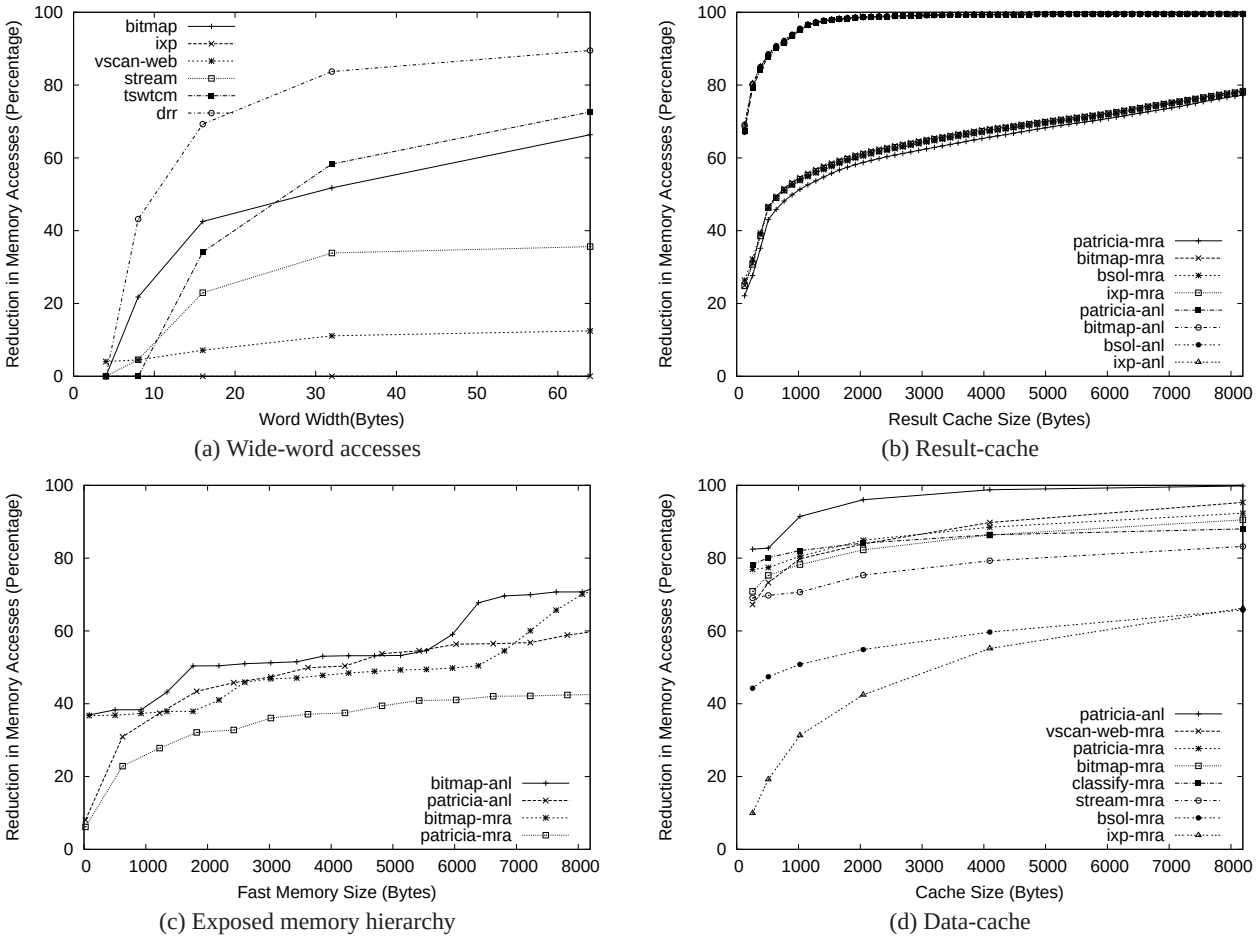
Programmers can use wide-word accesses explicitly or compilers, through data dependence analysis, can in some cases, issue wide-word accesses to fetch entire data items in one go. The effectiveness of wide-word accesses depends on the *spatial locality* of data accesses exhibited by applications and the *size* of the wide-word. Figure 1(a) shows the observed percentage reductions in memory accesses as a function of wide-word size; the applications shown here cover the entire range of observed values. It shows for a wide-word size of 32bytes, the number of memory accesses performed by applications while processing a packet reduces by about 10% for `vscan` application to 80% for `DRR`.

3.1.2 Result Caches

In some applications, the result of a computation depends solely on the input packet details (e.g., destination IP address). Such applications are amenable to caching the *results of computation*; cached results can be reused when another packet with the same header fields appears. This eliminates the entire computation as well as the corresponding memory accesses. An example of such a scheme is a *route-cache* that caches the results of route lookups. When a packet destined to the same IP address arrives again, the result of the previous lookup is re-used. Observe that while wide-word accesses exploit intra-packet spatial locality, result-caching exploits temporal locality in the input packet stream.

Figure 1(b) evaluates the effectiveness of result-caches in terms of the percentage reduction in the memory accesses as a function of the result-cache size for different longest-prefix-match (LPM) implementations. Figure 1(b) shows two sets of lines—for the ANL and MRA traces. It demonstrates that the percentage reduction in memory accesses is *only* a function of the trace, and not of the different LPM implementations. Further, it shows that the percentage reduction in memory accesses is higher for the ANL traces than the MRA traces. This is because, result-cache reduces memory accesses only when packets of the same flow are processed. 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. Hence, MRA traces contain traffic aggregated from a larger number of sources and contain larger number of simultaneous flows. This results in larger working sets and smaller hit-rates for a given result-cache size.

Note that a serious drawback of such result-caches is that many applications do not lend themselves to result-caching; in particular, result-caches can't be used in applications where the processing of a packet updates the persistent application state. For instance, in our metering applications, processing a packet updates the state (namely, byte and packet counters) maintained for the flow; further, the color of the mark a packet receives depends on this flow state. Thus, the color of the mark cannot be cached since the next packet in the flow may receive a different colored mark.



(a) Wide-word accesses

(b) Result-cache

(c) Exposed memory hierarchy

(d) Data-cache

Figure 1: Effectiveness of hammers

3.1.3 *Exposed Memory Hierarchy*

Today's NPs expose the memory hierarchy to the programmers; programmers *map* data structures to different levels of the memory hierarchy explicitly to reduce average latency.

In the simplest case, to map a data structure to a certain level of the memory hierarchy, the memory size should be at least as large as the maximum data structure size. This "entire" data structure mapping scheme is essential for most of the application data structures (e.g., a hash table or the virus-signature table). However, for certain data structures—such as the trie used in LPM—it is possible to partition the data structure such that the most frequently accessed portions of the data structure (namely, the top-levels of the trie) are placed in fast memory and the remaining structure is stored in a larger, slower memory [9, 22]. This approach approximates the behavior of a cache through static partitioning of the data structures. Figure 1(c) shows the percentage reduction in the number of memory accesses made to slower memory levels as a result of partitioning the trie across multiple levels of the hierarchy. Note that the data structures used in `bsol` and `ixp` have a very wide fanout at the first level; hence, they can't take advantage of small, fast memories using static mapping.

3.1.4 *Data Caches*

To study the effectiveness of data-caches, we consider a 4-way set-associative cache with 16-byte (4-word) wide lines[2]. Figure 1(d) depicts the percentage reduction in accesses to slow memory as a function of data-cache size for various applications under the MRA and ANL traces; the results shown cover the entire range of observed percentage reductions. It shows that even for a small data-cache size (8*KB*), the number of accesses to memory reduces by 65-99%. Further, as was the case for result-caches, because of the greater locality present in ANL traces, the percentage reduction in memory accesses is greater for ANL traces than MRA traces.

3.1.5 *Comparative Evaluation of Hammers*

Figure 2 compares, for all the applications and MRA traces, the percentage reduction in memory accesses resulting from the four different hammers: wide-word accesses, result-caches, exposed memory hierarchies, and data-caches. It illustrates that data-caches dominate wide-word accesses and exposed memory hierarchy in all cases; further, data-caches perform better than result-caches in all applications except `bsol` and `ixp`. In both of these applications, the data structure constructed for LPM has a very wide fanout at the first level; hence, there is little reuse of data across packets with different destination IP addresses. Further, because

[2]We have experimented with a range of associativity and line widths; all of these cases yield the same set of qualitative conclusions we report in this paper.

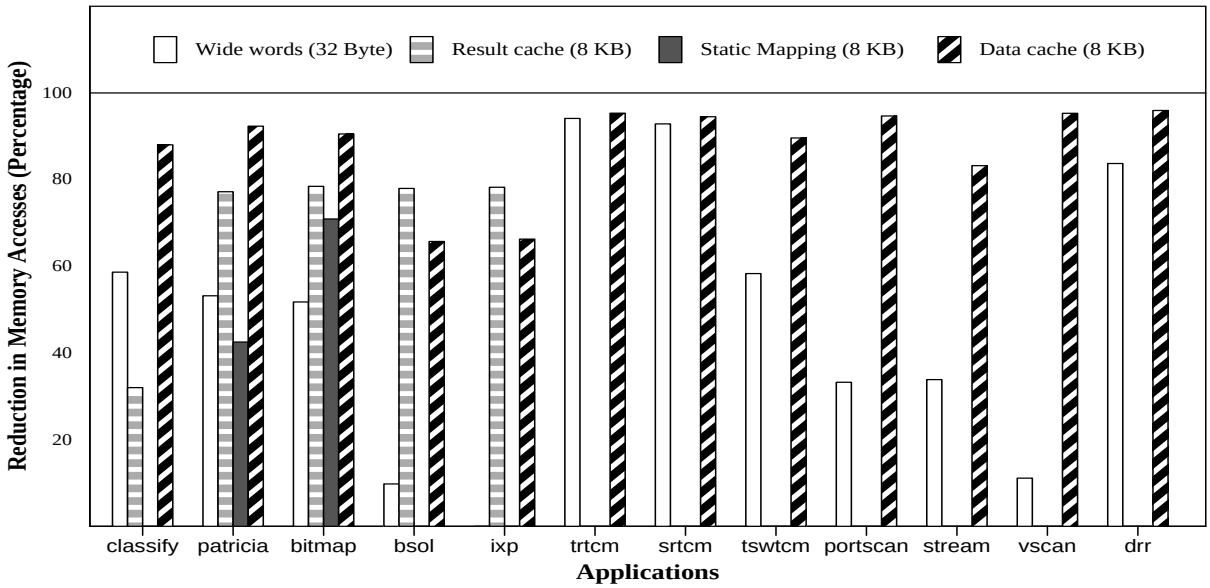


Figure 2: Comparison of Hammers

data-caches maintain individual data items (as opposed to only the result of the lookup), the working set size for data-caches in these applications is larger than result-caches.

From the above discussion, we conclude that data-caches provide an attractive alternative to wide-word accesses and exposed memory hierarchies. Unlike result-caches, data-caches are more general and can be used for all data structures. Further, data-caches are transparent to programmers and compilers, and thus simplify programmability. Hence, data-cache is an effective hammer for packet processing.

In Figure 3, we demonstrate the benefits and limits of using data-caches with respect to two performance metrics: (1) *reduction in the processing time* of a single packet, and (2) *processor utilization*, defined as 1 minus the percentage of the time a processor stalls waiting for memory accesses. For our experiments, we use a miss penalty of 150 cycles, which represents the latency in accessing off-chip QDR SRAM in IXP2800 [22].

For the MRA traces, Figure 3(a) shows the reduction in packet processing time as a function of cache size for six applications (that capture the full range of reductions observed for all the applications). Note that the application `bitmap` that benefits the least is in fact the one with higher hit-rates than many other applications. This is because, `bitmap` examines long bitmaps to determine the next trie node to visit and hence executes a larger number of compute operations per memory access. Thus, the impact of reducing average memory latency is small. On the other hand, applications such as `DRR` exhibit significantly lower amount of locality; yet, because of their memory-access-intensive nature, even a small data-cache of 8*KB* can reduce the per-packet processing time by as much as 85%.

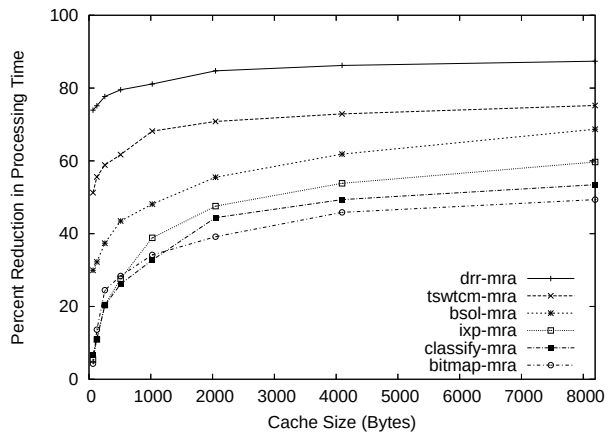
Figure 3(b) depicts processor utilization as a function of cache size for the same six applications. The compute-bound `bitmap` achieve nearly 80% utilization for small cache sizes. `ixp` performs as well as `bitmap`, but it needs a larger cache (about 32*KB*). For all other applications, processor utilization does not increase much beyond 30% even for very large cache sizes (64*KB*). Because of the memory-access-intensive nature of these applications, even a small number of data-cache misses cause significant processor stalls. Thus, we conclude that although data-caches can reduce the processing time of a single packet significantly, they alone are insufficient to achieve acceptable processor utilization and high packet processing throughput in many applications.

3.2 Ladders: Hiding Overhead

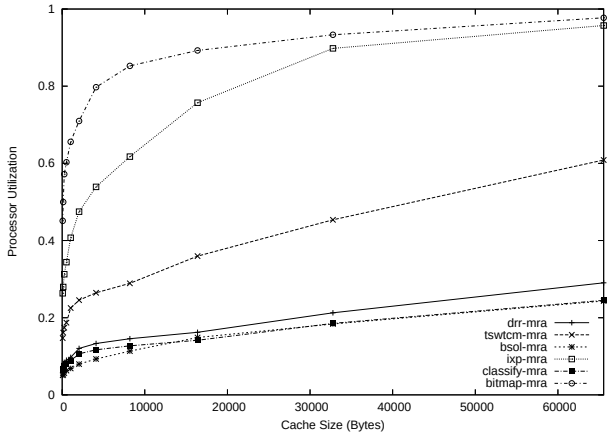
Since throughput is often the primary performance metric for packet processing systems, NPs utilize ladders–such as *hardware multi-threading* and *asynchronous memory*–that exploit inter- and intra-packet parallelism, respectively, to *hide* memory access latencies and to improve processor utilization. In this section, we demonstrate that systems that use ladders alone can be severely limited by the available off-chip memory bandwidth.

Figure 4 shows, for various applications, the processor utilization attainable using hardware multi-threading; with multi-threading, processor switches context to a different thread upon scheduling a memory access. We show only a subset of applications due to space constraints. However, the results for each of the remaining applications resemble closely to one of these four and our conclusions remain valid. For this and other experiments in this section, we use a context switch overhead of two cycles (the minimum mandatory in IXP2800) and a miss-penalty of 150 cycles [22]. Figure 4 illustrates that, for all applications, processor utilization improves linearly with increase in the number of threads supported. However, the rate of improvement as well as the saturation levels are different for different applications. The following application charateristics influence the rate of improvement and saturation levels.

- The number of computation instructions executed per context-switch (C). Higher values of C amortize the context-switch overhead better; the higher the C, the greater is the rate of improvement in processor utilization with increase in number of threads and higher is the saturation level. For instance, the values of C are 26.71 and 1.8 for the `ixp` and `patricia`, respectively. Thus, for a context switch overhead of 2 cycles, while the `ixp` is able to achieve a utilization of $26.71/(26.71+2) = 0.93$, the `patricia` is limited to $1.8/(1.8+2) = 0.47$.



(a) Reduction in packet processing time



(b) Processor utilization

Figure 3: Benefits and limits of data-caches

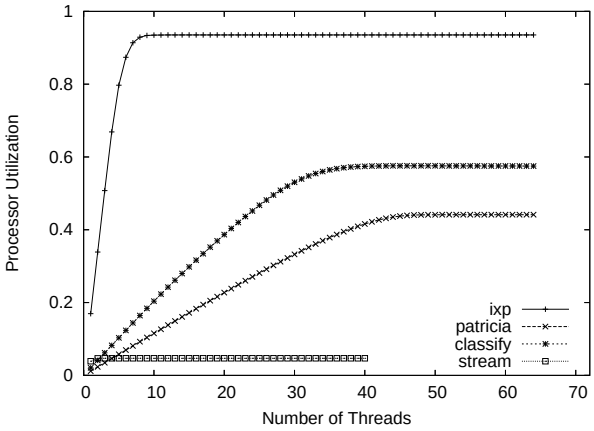


Figure 4: Effectiveness of Hardware Threads

- Access to shared read/write data: The need to serialize access to shared read/write data limits the amount of parallelism, and hence the effectiveness of multi-threading. This effect is quantified in Figure 4 for the two applications `stream` and `classify`. `stream` involves accessing and updating a shared splay tree for every packet; hence, increasing the number of threads yields little improvement in processor utilization. The `portscan` and the `DRR` applications exhibit similar behavior. The `classify` application, on the other hand, involves a small critical section that serializes processing of packets when a flow is either added or deleted. Hence, the impact of serialization is marginal. The metering schemes we consider exhibit behavior similar to `classify`.

Limitations of multi-threading: We argued above that the performance of multi-threading is limited by *C* and the serailization considerations resulting from access to shared read/write data. NPs can utilize another ladder—namely, *asynchronous memory accesses*—to switch context only after issuing multiple memory accesses; this increases the value of *C* and improves the achievable processor utilization. Today, however, a programmer is required to utilize asynchronous memory explicitly; this complicates programming.

A more significant limitation of multi-threading is that the memory bandwidth requirement of a multi-threaded processor grows linearly with increase in number of threads. This is because, with multi-threading, the number of memory accesses made during the time taken to serve one grows linearly with the number of threads in use and saturates at $Penalty/(C+ContextSwitchOverhead)$ where *C* denotes the number of compute operations performed per context-switch, and *Penalty* denotes the memory access latency. Thus, processor utilization achieved by multi-threading is limited not only by *C* but also by the available memory bandwidth.

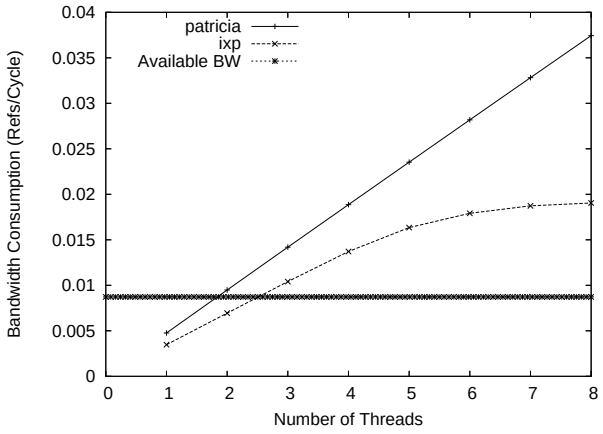


Figure 5: Multi-threading under bandwidth constraints

Figure 5 shows the bandwidth usage for the two applications `ixp` and `patricia` in references/cycle. The memory bandwidth available for IXP2800 is computed based on the aggregate bandwidth of the four QDR SDRAM banks and the processor clock of 1.4GHz. Figure 5 illustrates that, if all the 8 threads on an IXP2800 core execute `patricia`, then the core would consume almost 4 times the fair bandwidth share of the core. In other terms, running `patricia` on *only* 4 out of the 16 cores available on IXP2800 would saturate the available memory bandwidth. `ixp`, the hand-tuned implementation of LPM for IXP2800 does better. However, even in this case, `ixp` can utilize only 7 of the 16 cores before saturating the available memory bandwidth (since running `ixp` on 8 threads on a core requires almost twice the fair bandwidth share of the core). Thus, we conclude that ladders that hide latency to improve processor utilization are necessary, but not sufficient to achieve high throughput.

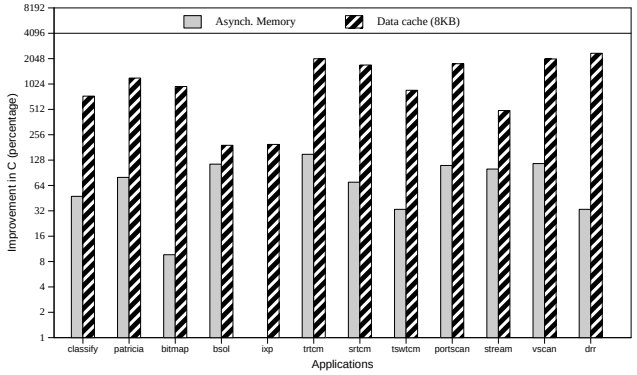


Figure 6: Comparing the effectiveness of data-caches and asynchronous memory in increasing C

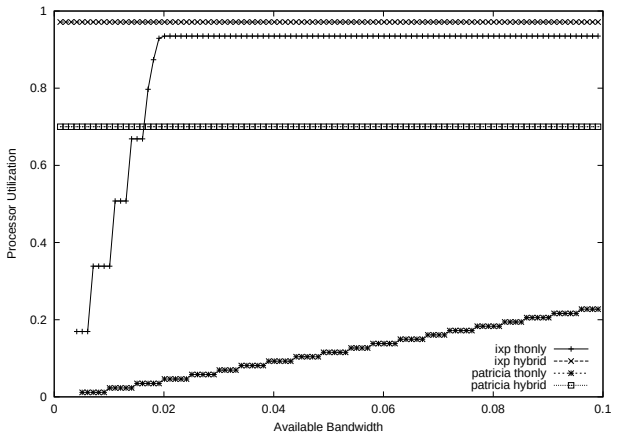


Figure 7: Hybrid vs. Threads-only architecture

3.3 Hybrid Network Processor Architecture

In the previous subsections, we have demonstrated that hammers and ladders are both necessary to achieve high throughput; however, no single mechanism in isolation is sufficient. In this section, we consider a hybrid NP architecture that combines a hammer (namely, data-cache) and a ladder (namely, multi-threading). We consider this combination for three reasons. First, data-caches and multi-threading are complementary. Data-caches exploit locality in the workload to reduce the number of context-switches and the off-chip memory bandwidth requirement. Multi-threading exploits parallelism in the workload to hide long cache-miss latencies and to reduce processor stalls. Second, data-caches are the best-of-breed in hammers, and multi-threading is the most effective ladder. Third, with respect to reducing the number of context-switches and increasing the value of C, data-caches outperform asynchronous memory accesses significantly (see Figure 6); further, asynchronous memory accesses do not reduce the off-chip memory requirement of multi-threaded processors.

To demostrate the effectiveness of this hybrid architecture, we consider two alternative processor architectures that are identical in terms of the functional units and the overall chip-area, but different with respect to their multi-threading and caching configurations. The first alternative (denoted as *threads-only*) uses the available chip-area (i.e., the area remaining after including appropriate functional units and pipeline stages) to support as many hardware threads as possible; this is representative of today's NP designs [19]. The second alternative (denoted as *hybrid*) splits the available chip-area between cache and threads. For this case, we determine the optimal configuration by exhaustively simulating all possible combinations of cache sizes and number of threads. We allocate chip-area to hardware threads and cache in units of *thread-equivalents*, which refers to the chip-area required to support a single hardware thread. We estimate a thread-equivalent using information about Intel's IXP2800 [19]; we use the Cacti toolkit [3] to estimate the number of cache lines that can be accommodated in a certain chip-area.

Figure 7 compares the processor utilization achieved by threads-only and hybrid architectures as a function of available off-chip memory bandwidth. We consider the two LPM implementations, `ixp` and `patricia`, from the previous plots. For this experiment, we fix the chip-space available to the two processor configurations at 64 thread-equivalents. As Figure 7 demonstrates, for `patrica`, the hybrid processor achieves almost 3 times the utilization (and hence throughput) of the threads-only processor. For `ixp`, threads-only and hybrid systems achieve similar peak utilizations; however, the hybrid system uses only a small fraction of the memory bandwidth (0.006 instead of 0.02 references/cycle). The horizontal lines for the hybrid system indicate that its throughput is not constrained by memory bandwidth at all; thus, the hybrid architecture scales better with increase in chip area than the threads-only architecture.

Based on our analyses, we make the following three observations. First, hybrid architecture that combines data-caches and multi-threading is effective in overcoming the memory wall in packet processing. Second, unlike general-purpose processors where data-caches are used to reduce average memory access latency, the primary benefit of supporting data-caches in network processors is in reducing *number of context-switches* and *off-chip memory bandwidth* requirement. Third, data-caches are transparent to the programmers and compilers. The presence of data-caches also simplifies the usage of hardware multi-threading. In particular, in the presence of data-caches, NPs can simply switch context from one thread to another on a cache miss. This eliminates the need to schedule hardware threads explicitly (by the programmer or the compiler). In such a case, a programmer is *only* required to develop thread-safe applications suitable for execution on a multi-processor; the existence and usage of multi-threading becomes transparent. Thus, a hybrid architecture that supports both data-caches and multi-threading in hardware simplifies programmability of NPs significantly.

4. DISCUSSION

In this paper, we have argued that NPs should include data-caches to address the shortcomings of hardware multi-threading. However, none of today's commercial NPs support data-caches. This is often attributed to two main hypotheses. First, since throughput is the primary performance metric for packet processing systems, mechanisms–such as hardware multi-threading–that *hide* memory access latency are sufficient; data-caches–that *reduce* average memory access latency–are not necessary. Second, packet processing systems must be (and are) provisioned with sufficient resources to meet the *worst-case* traffic demands; since caches only improve the average-case, they are not beneficial.

In the previous sections, we showed that the first hypothesis does not hold. In what follows, we argue that most modern packet processing systems are provisioned with sufficient resources *only* to process an *expected* traffic mix; hence, including caches in NPs to improve the average-case is valuable.

Traditionally, vendors of IP routers (Layer-2 and Layer-3 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.

Worst-case provisioning advertised in IP routers, however, is somewhat misleading. This is because, benchmarks defined by IETF and used by most vendors [7] define worst-case only in terms of the arrival rate of packets (namely, arrival of smallest size packets at the line rate) that request the basic IP forwarding service (RFC-1812). An IP router, in reality, processes multiple types of packets (e.g., IP packets with and without options). Processing IP packets with options, for instance, takes considerably larger number of processor cycles than basic IP forwarding; provisioning sufficient resources in an IP router to service worst-case arrival pattern of packets that request IP options processing is prohibitively expensive. The well-known 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 mix*. The practice of optimizing for the average-case is also evident in current commercial IP routers that use a *route-cache* to expedite route lookup in the average-case.

The engineering practice of provisioning sufficient processors to meet the demands of *expected* traffic mix is even more pronounced in packet processing systems supporting complex applications (e.g., SSL, NAT (RFC-1631), firewall, IPv4/IPv6 Interop (RFC-2766)). These systems are generally deployed in edge and enterprise networks and constitute 93.6% of all of the network processor deployments today [4]. Most of these applications involve multiple types of packets; each of these packet types contributes a reasonable percentage of the total traffic; and the arrival rate of packet types 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 not only can improve processor utilization and throughput, but can also simplify programmability.

5. RELATED WORK

In this paper, we considered the problem of overcoming the memory wall resulting from data accesses in packet processing applications. Much of the prior work on overcoming the memory wall has focused either on accesses resulting from instruction fetch, or on demonstrating the benefits of specific mechanisms for handling data accesses in the context of specific applications (e.g., IP forwarding).

Nahum et al. [25] study the instruction cache locality of a TCP implementation on a general-purpose processor. Xie et al. [40] analyze the instruction cache behavior with the goal of designing an efficient ISA. Wolf et al. [37] propose a multiprocessor scheduler that avoids cold instruction caches when assigning packets to processors.

As for the data accesses, a significant amount of prior work considers the problem of efficiently transferring the packet-data (header and payload) from the ingress to the egress interface [18, 20]. The problem of application-data accesses is well-studied in the context of route lookup. Many techniques have been proposed for reducing the number of memory accesses performed during lookup [30]. While lookup schemes that can exploit data-caches are proposed in [10, 34], memory hierarchy designs specifically meant for particular lookup schemes are considered in [9, 31]. Locality in destination IP addresses has also been well-characterized [15, 21, 27, 28]. This observation forms the basis of several proposals for *result-caches*. For instance, special-purpose *route-cache* hardware has been proposed for route lookup and for a similar but more general problem of Layer-4 classification in [11] and [41], respectively. Techniques for improving performance of such route-caches are explored in [16].

Memory access behavior of some packet processing applications is analyzed in [13, 23, 24, 29, 36, 42]. However, none of these studies compare the relative benefits of the various techniques for addressing memory bottleneck.

6. CONCLUSION

In this paper, we address the question: what *minimal* set of hardware mechanisms must a network processor support to achieve the twin goals of simplified programmability and high packet throughput? We show that no single mechanism suffices; the minimal set must include *data-caches* and *multi-threading*. Data-caches—generally not supported in today's network processors—dominate such mechanisms as wide-word accesses, exposed multi-level memory hierarchy and special-purpose caches. However, because packet processing applications are memory-access-intensive, cache misses can cause significant processor stalls. Hence, data-caches alone are insufficient to achieve high throughput. Hardware multi-threading is effective in hiding memory access latency; its effectiveness, however, is limited severely by constraints on off-chip memory bandwidth, context-switch overhead, and serialization required to protect access to shared read-write data.

Data-caches and multi-threading are complementary; whereas data-caches exploit locality to reduce the number of context-switches and the off-chip memory bandwidth requirement, multi-threading exploits parallelism to hide long cache-miss latencies. We also argued that a hybrid architecture that supports both data-caches and multi-threading in hardware simplifies programmability of NPs significantly.

This study raises several interesting questions. First, given a chip-area and off-chip memory bandwidth, what combination of number of cores, threads-per-core and cache-per-core should an NP provision to maximize throughput? Second, since the balance among these three dimensions may depend upon application, system, and trace characteristics, is it possible to design a fixed NP architecture that achieves performance similar to the architecture best-suited for each deployment scenario? If not, how should one design versatile NP architectures? Addressing these questions will impact the design and use of future generations of NPs.

7. REFERENCES

[1] http://www.caida.org/analysis/workload.

[2] Benchmarks, Network Processing Forum. http://www.npforum.org/benchmarking/index.shtml.

[3] Cacti3.2 research.compaq.com/wrl/people/jouppi/cacti.html.

[4] The reincarnation of network processor market. In-Stat MDR, Dec.2003.

[5] Snort: The Open Source Network Intrusion Detection System. http://www.snort.org/.

[6] The Simplescalar Tool Set Version 3.0. http://www.simplescalar.com/.

[7] The Tolly Group: Information Technology Testing, Research and Certification . http://www.tolly.com.

[8] University of Oregon Route Views Project. http://www.routeviews.org/.

[9] J.-L. Baer, D. Low, P. Crowley, and N. Sidhwaney. Memory hierarchy design for a multiprocessor look-up engine. In *Proc. of the 12th International Conference on Parallel Architectures and Compilation Techniques*, 2003.

[10] T.-C. Chiueh and P. Pradhan. High-Performance IP Routing Table Lookup using CPU Caching. In *Proc. of IEEE INFOCOMM'99*, 1999.

[11] T.-C. Chiueh and P. Pradhan. Cache Memory Design for Network Processors. In *Proc. of the 6th HPCA*, Jan 2000.

[12] D. Comer. *Network Systems Design Using Network Processors*. Prentice Hall, ISBN 0-13-141792-4, 2003.

[13] P. Crowley, M. E. Fiuczynski, and J.-L. Baer. Characterizing Processor Architectures for Programmable Network Interfaces. In *Proc. of the 2000 International Conference on Supercomputing*, May 2000.

[14] W. Eatherton, G. Varghese, and Z. Dittia. Tree bitmap: hardware/software IP lookups with incremental updates. *SIGCOMM Comput. Commun. Rev.*, 34(2), 2004.

[15] D. C. Feldemeir. Improving Gateway Performance with a Routing-table Cache. In *Proceedings of IEEE INFOCOMM'88*, March 1988.

[16] K. Gopalan and T.-C. Chiueh. Improving Route Lookup Performance Using Network Processor Cache. In *IEEE/ACM SC Conf.*, 2002.

[17] P. Gupta and N. McKeown. Algorithms for Packet Classification. In *IEEE Network*, March/April 2001.

[18] J. Hasan, S. Chandra, and T. N. Vijaykumar. Efficient Use of Memory Bandwidth to Improve Network Processor Throughput. In *Proc. of the 30th ISCA*, June 2003.

[19] *Intel IXP2800 Hw. Ref. Manual*, Nov 2002.

[20] S. Iyer, R. R. Kompella, and N. McKeown. Techniques for Fast Packet Buffers. In *Gigabit Networking Workshop*, April 2001.

[21] R. Jain. Characteristics of Destination Address Locality in Computer Networks: A Comparison of Caching Schemes. *Computer Networks and ISDN Systems*, 18(4), May 1990.

[22] E. J. Johnson and A. Kunze. *IXP 2xxx Programming*. Intel Press, 2003.

[23] G. Memik, W. H. Mangione-Smith, and W. Hu. NetBench: A Benchmarking Suite for Network Processors. In *ICCAD*, 2001.

[24] J. Mudigonda, H. M. Vin, and R. Yavatkar. Managing Memory Access Latency in Packet Processing. In *Proc. of ACM SIGMETRICS*, Banff, Canada, June 2005.

[25] E. Nahum, D. Yates, J. Kurose, and D. Towsley. Cache behavior of network protocols. *SIGMETRICS*, 25(1):169–180, 1997.

[26] NLANR Network Traffic Packet Header Traces. http://pma.nlanr.net/Traces/.

[27] C. Partridge. A Fifty Gigabit Per Second IP Router. *IEEE/ACM Transactions on Networking*, 6(3), June 1998.

[28] C. Partridge. Locality and Route Caches. In *NSF Workshop, Internet Statistics Measurement and Analysis*, February 1999.

[29] R. Ramaswamy, N. Weng, and T. Wolf. Analysis of network processing workloads. In *Proc. of IEEE Intnl. Symp. on Perf. Analysis of Systems and Software*, Austin, TX, Mar. 2005.

[30] M. Ruiz-Sanchez, E. Biersack, and W. Dabbous. Survey and Taxonomy of IP Address Lookup Algorithms. *IEEE Network*, March 2001.

[31] T. Sherwood, G. Varghese, and B. Calder. A Pipelined Memory Architecture for High Throughput Network Processors. In *ISCA*, 2003.

[32] M. Shreedhar and G. Varghese. Efficient Fair Queuing Using Deficit Round Robin. In *Proceedings of ACM SIGCOMM*, August 1995.

[33] K. Sklower. A Tree-Based Packet Routing Table for Berkely Unix. In *Proceedings of the Winter 1991 USENIX Conference*, January 1991.

[34] V. Srinivasan and G. Varghese. Fast Address Lookups using Controlled Prefix Expansion. *ACM Trans. on Computer Systems*, 17(1), 1999.

[35] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed ip routing lookups. In *Proc. of the ACM SIGCOMM*, 1997.

[36] 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.

[37] T. Wolf and M. A. Franklin. Locality-aware Predictive Scheduling of Network Processors. In *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software*, Nov 2001.

[38] S. Wu and U. Manber. A Fast Algorithm for Multi-pattern Searching. Technical Report TR-94-17, 1994.

[39] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, 1995.

[40] H. Xie, L. Zhao, and L. Bhuyan. Architectural analysis and instruction-set optimization for design of network protocol processors. In *Proc. of the 1st IEEE/ACM Intl. Conf. HW/SW codesign & synthesis*, 2003.

[41] 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.

[42] L. Zhao, R. Illikkal, S. Makineni, and L. Bhuyan. TCP/IP Cache Characterization in Commercial Server Workloads. In *CAECW 2004, Along with HPCA-10*, 2004.