

# Balancing Multi-processing, Threading and Caching in Network Processors

Jayaram Mudigonda and Harrick Vin

## Abstract:

The performance of packet processing applications is limited by memory access latencies and bandwidth. Today’s network processors rely upon multi-processing and multi-threading to achieve high packet throughputs. However, they generally don’t utilize caches. Recent studies have shown that moderate-sized data caches can be effective in improving packet throughput. In this paper, we explore the architectural design space defined by multi-processing, multi-threading, and caching to identify optimal configuration for network processors.

We make three primary contributions. First, we present a methodology for exploring the architectural design space for NPs systematically and efficiently. Design space exploration is challenging because: (1) multi-processing, multi-threading, and caching interact in subtle ways, and (2) the vast design space makes straightforward simulations prohibitively expensive. Second, we show that a balanced configuration—derived with the chip-area and memory bandwidth of Intel’s IXP2800—can achieve upto 3 times the packet processing throughput as compared to a processor with IXP2800’s configuration. Third, we show that packet trace characteristics influence the architectural balance significantly; in particular, optimal NPU architectures for the network edge can differ significantly from those for the network core.

## 1 Introduction

Network processors (NPs) are basic building blocks for designing *packet processing systems*. Today, packet processing systems support a wide-range of *header-processing applications* such as network address translation (NAT) [18], protocol conversion (e.g., IPv4/v6 inter-operation gateway) [39] and firewall [5]; as well as *payload-processing applications* such as Secure Socket Layer (SSL) [21], intrusion detection [7], and virus scanning [7]. Maximizing packet processing throughput is a key design criteria for these systems.

Based on Little’s law, the throughput  $\lambda$  of a system is given by:  $\lambda = \frac{N}{T}$ , where  $N$  is the number of packets being processed concurrently and  $T$  is the time required to process each packet. Thus, increasing  $\lambda$  requires either an increase in  $N$ , or a decrease in  $T$ , or both. Observe that the time  $T$  to process a packet is governed primarily by memory access latencies; hence, speeding up the processing of computational instructions reduces  $T$  only marginally. Further, whereas the network link bandwidths (and hence, the throughput requirements) have doubled every year over the past decade, memory access latencies have improved by only about 10% each year during the same period.

To meet throughput demands, today’s NPs include multiple processor cores each with support for hardware multi-threading. For instance, Intel<sup>®</sup>’s IXP2800 network processor supports 16 RISC cores (referred to as *micro-engines*) each with 8 hardware threads as well as an XScale<sup>®</sup> core on-chip. Support for multiple processor cores allow NPs to process packets concurrently. Further, with hardware multi-threading, each processor can switch context from one packet to another when a thread on a memory access; this allows NPs to hide memory access latencies, as well as increase the level of concurrent processing. This approach exploits the packet-level parallelism inherent in applications and workload.

An alternate architectural design is one where *data caches* are used to reduce  $T$ , and thereby increase  $\lambda$ . However, unlike conventional general-purpose processors that rely extensively on caching to reduce average memory access latencies, NPs often do not support data caches. A recent study [12] analyzed the locality properties exhibited by a wide-range of data structures used in modern packet processing applications. IT showed 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 meta-data) 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. It also demonstrated 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 can be highly effective.

In light of this observation, in this paper, we ask the following questions: (1) *what is the optimal balance among multi-processing, multi-threading, and caching in NP architectures?* (2) *how much throughput improvement can such a balanced architecture achieve over today’s NP designs?* and (3) *how does the optimal architectural balance change with application and trace characteristics; system parameters (e.g., miss penalty and context-switch overhead); and technology trends (e.g., increase in chip area)?* Observe that identifying an optimal configuration that balances multi-processing, multi-threading, and caching is challenging. This is because, (1) multi-processing, multi-threading, and caching interact in subtle ways, and (2) the vast design space makes straightforward simulations prohibitively expensive.

We make three primary contributions. First, we present a methodology for exploring the architectural design space systematically and efficiently. Second, we show that a balanced configuration—derived with the chip-area and memory bandwidth of Intel’s IXP2800—can achieve upto 3 times the packet processing throughput as compared to a processor with IXP2800’s configuration. Third, we show that packet trace characteristics influence the architectural balance significantly; in particular, optimal NPU architectures for the network edge can differ significantly from those for the network core.

The rest of the paper is organized as follows. In Section 2, we discuss the viability of caching in NPs and then formulate the problem of identifying optimal architectural balance. Section 3 describes our methodology for exploring the design space efficiently. In Sections 4 and 5, we discuss the applications and traces that we utilize for our experiments, and then describe our results. Section 6 discusses related work, and finally, Section 7 summarizes our contributions.

## 2 Problem Formulation

As we argued in Section 1, to increase throughput  $\lambda$ , a network processor can support multi-processing/threading mechanism to process multiple packets concurrently, or caching to reduce the processing time for each packet. In this section, we argue that neither of these two mechanisms are sufficient in isolation; to maximize packet processing throughput, an NP should utilize a *hybrid* architecture that combines these mechanisms.

### 2.1 Multi-processing and Multi-threading: Benefits and Limitations

**Benefits :** The nature of packet processing applications as well as workload characteristics offer significant opportunities for processing packets in parallel. Multiple processor cores with hardware multi-threading can exploit this parallelism to achieve high throughput. Consequently, most NPs today include support for multi-processing and multi-threading [3, 27].

**Limitations :** The multi-processing/threading mechanism for improving packet throughput has the following limitations.

- Multi-threading allows a processor to switch context to a different thread when the executing thread stalls on a memory access. By overlapping computation with memory accesses, multi-threading hides memory access latency and achieves high utilization for each processor core. The effectiveness of this mechanism, however, is governed by four factors: (1) the average number of computational cycles  $C$  executed between two context-switch operations; (2) the overhead  $O$  of context switch operation; (3) the number of hardware threads  $T$  supported by the processor; and (4) the memory access latency  $M$ . Whereas the peak processor utilization achievable by multi-threading is given by  $\frac{C}{C+O}$ , the number of hardware threads  $T$  required to achieve this utilization level is given by  $T = \frac{M}{C+O}$ . Figure 1(a) illustrates this behavior for two realistic systems—a QoS router and a virus scanner—on a processor with  $O = 4$ . It shows that the processor utilization saturates at about 0.5 (indicating that  $C \approx 4$ ); further, at large memory access latencies, achieving even this level of utilization requires a large number of threads.

It is important to observe that because of pipelined processor architectures,  $O > 0$ . For instance, for Intel<sup>®</sup>’s IXP2800 NP,  $O = 4$ . Masking this context-switch overhead requires programmer/compiler to schedule useful instructions in these *defer slots*. However, since packet processing applications are often not compute-bound, finding sufficient number of instructions to fill differ slots is tricky. An examination of a collection of hand-tuned packet processing applications included in Intel<sup>®</sup>’s IXA SDK3.0 [4] reveals that only about 67% of the two usable defer-slots could be filled on an average (hardware restrictions allow usage of only two of the four slots [26]); further, these applications when executed on IXP2800 achieve only 55-78% processor utilization.

- Increasing the number of packets being processed concurrently increases the total memory bandwidth requirement linearly. Once the concurrency level increases to a point that saturates the available memory bandwidth, any further increase in the number of processors or threads-per-processor does not yield any benefits (and may, in fact, worsen packet throughput because of memory contention). Thus, the multi-processing/threading mechanism becomes ineffective at large die sizes.

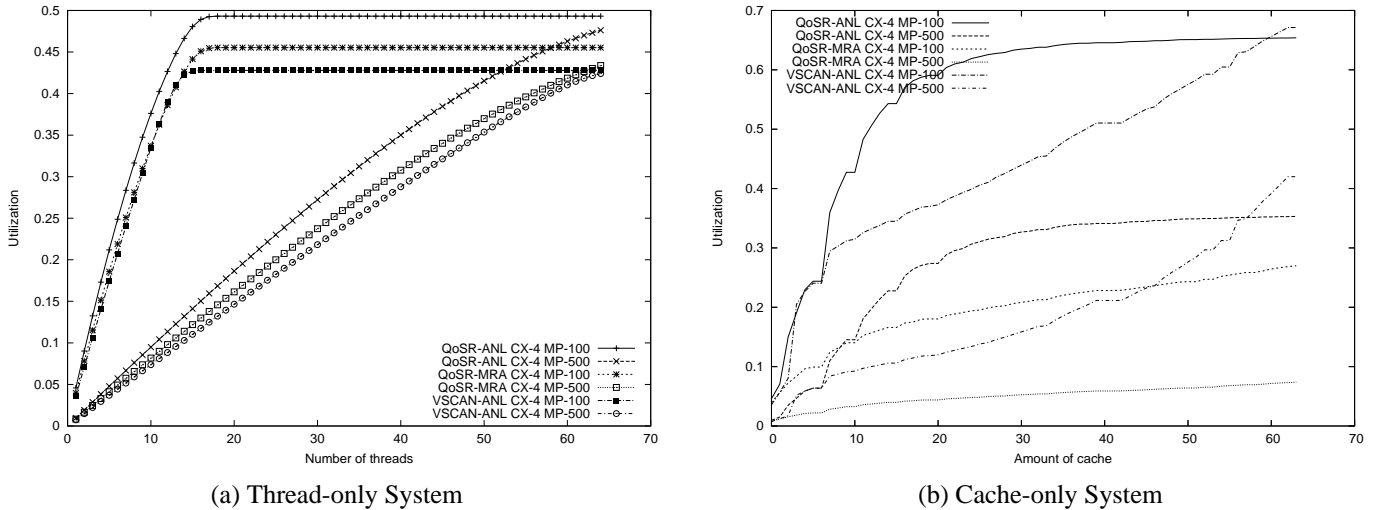


Figure 1: Limitations of Thread-only and Cache-only Architectures; Cache sizes are defined in terms of *thread-equivalent* chip area (see Section 3.1)

## 2.2 Caching: Benefits and Limitations

Caching is an effective technique for reducing *average* memory access times. However, unlike conventional general-purpose processors that rely extensively on caching to reduce average memory access latencies, NPs often do not support data caches. In what follows, we first argue that data caches can indeed be beneficial and effective in NPs. Then, we discuss the limits of caching for improving packet processing throughput.

**Benefits :** 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. We first argue that this hypothesis is false.

Worst-case provisioning advertised in conventional IP routers 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 [9]. 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 [33].

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 [21], Network Address Translation [18], firewalls [5], IPv4/IPv6 Interoperability [39], and TCP/IP header compression and decompression [28]).

These systems are generally deployed in edge and enterprise networks, and constitute 93.6% of all of the NP-deployments today [6]. 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). Further, the arrival rate for each packet type varies widely over time. Hence, provisioning such systems to meet the worst-case processing demands of all packet types is often prohibitively 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, and thereby increase the average packet throughput. 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. Thus, contrary to the popular belief, support for caching in network processors *can* offer significant benefits.

A recent study [12] analyzed the locality properties exhibited by a wide-range of data structures used in modern packet processing applications. It showed 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 meta-data) 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. It also demonstrated 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 can be effective.

**Limitations** : The effectiveness of caches to improve the processor utilization (and hence, the packet throughput) depends on: (1) the hit rate observed at a given cache size; and (2) the miss penalty. Figure 1(b) shows that at moderate to high miss penalties, achieving high utilization level requires very large cache sizes. Further, even when the hit rate saturates, the processor utilization can be relatively small.

### 2.3 A Case for Hybrid Architecture

Whereas the performance of multi-processing/threading is limited by the ratio  $\frac{C}{C+O}$  and the available memory bandwidth, the effectiveness of caching is limited by miss penalty. A *hybrid* NP architecture that combines multi-processing/threading with caching has the potential for maximizing processor utilization and packet throughput. On the one hand, caching increases the average number of instructions executed between context swaps (and thereby increase the ratio  $\frac{C}{C+O}$ ); further, it reduces the number of requests sent to off-chip memory, thereby alleviating memory bandwidth bottleneck. On the other hand, multi-threading allows a system with caches to mask miss penalty (by overlapping the execution of threads with cache misses), and thereby improve processor utilization.

This leads to the following three questions: (1) *what is the optimal balance among multi-processing, multi-threading, and caching in NP architectures?* (2) *how much throughput improvement can such a balanced architecture achieve over today’s NP designs?* and (3) *how does the optimal architectural balance change with application and trace characteristics; system parameters (e.g., miss penalty and context-switch overhead); and technology trends (e.g., increase in chip area)?* We address these questions in the rest of this paper.

## 3 Methodology

In this section, we first discuss the system model and assumptions used in our analysis; then, we describe our methodology for exploring the architectural design space to identify optimal network processor configurations.

### 3.1 System Model and Assumptions

We consider a single-chip, homogeneous multi-processor where the individual processor cores are multi-threaded. Each processor contains a data cache that is shared by all the threads. The data cache is used for only application data structures (that exhibit considerable temporal locality). Packet data structures (such as packet header and content) exhibit only spatial locality—hence,

we assume that they are managed using prefetch- or stream-buffers. This ensures that packet data structures do not corrupt the data cache. Finally, we assume that the stack can be maintained in registers or some fast local memory; hence, all stack accesses can be serviced within a single cycle.

We assume that the code running on the processor cores resides completely within its instruction cache/store; hence, an instruction fetch never stall the processor. We also assume that all computational instructions complete in one cycle.

Each packet arriving in the system is assigned to a thread; the thread processes the packet to completion prior to selecting a new packet for service. A processor switches context from one thread to another when a memory reference results in a miss in data cache. We assume that threads are scheduled in FIFO order from a queue of ready-to-run threads; further, the context switch time is constant (i.e., not a function of the number of threads).

For the purpose of estimating chip area, we represent a single multi-threaded core as consisting of a *thread-independent*, a *thread-specific*, and a *data-cache* region. The thread-independent region includes such components as one or more functional units, pipeline logic, and instruction fetch and decode units. The thread-specific region includes all the context-specific information such as register files, status words, program counter, etc. The data-cache region contains the data cache. We assume that the size of the thread-specific region grows linearly with number of threads supported by the processor core. Thus, the area occupied by a multi-threaded core is estimated as:  $S_I + T \times S_T + S_C$ , where  $T$  is the number of threads per core; and  $S_I$ ,  $S_T$ , and  $S_C$ , respectively, denote the chip area sizes of the thread-independent, thread-specific, data-cache regions.

We derive estimates for  $S_I$  and  $S_T$  using Intel<sup>®</sup>'s IXP2800 NP. IXP2800 is designed using 0.13 micron technology; it consists of 16 processing cores (referred to as micro-engines) that are used for fast-path packet processing<sup>1</sup>. The 16 micro-engines occupy approximately 29% of the  $14.116 \times 18.878 \text{ mm}$  die. 73% of the area of each micro-engine represent thread-independent region; it is taken up by the instruction store, functional units, pipeline and a 640-word local memory (used for storing stack variables). The remaining 27% (i.e.,  $3.526 \text{ mm}^2$ ) of the area of each micro-engine constitutes thread-specific region. Since each micro-engine supports 8 threads, each thread occupies  $0.163 \text{ mm}^2$  chip area. For our analysis, we assume that chip area is allocated in units of per-thread area (namely,  $0.163 \text{ mm}^2$ ); we refer to this as a *thread-equivalent* chip-area. Note that for IXP2800, the thread-independent area used for each micro-engine is about 21 thread-equivalents, while the total area occupied by the 16 micro-engines is approximately 475 thread equivalents.

We estimate the chip area occupied by data cache using Cacti. We initialize Cacti using 0.13 micron fabrication process (the same one used for IXP2800). With this setting, for instance, a single thread-equivalent chip area can accommodate about 128 2-way associative 64-bit lines. Once more than 8 thread-equivalent chip-area is allocated to data cache, every additional thread-equivalent chip-area results in the addition of about 256 cache lines.

Finally, because of our homogeneous processor assumption, we estimate that the chip-area and the memory bandwidth requirements grow linearly with number of processors. This allows us to analyze a single processor architecture first, and subsequently extend the results to a multi-core system.

### 3.2 Architectural Space Exploration

The objective of our architectural space exploration is to identify optimal network processor configurations (consisting of some combination of multi-processing, multi-threading, and caching). The exploration is constrained by the total chip-area availability. For each configuration, we determine *system utilization* and the total *off-chip memory bandwidth requirement*. System utilization is defined as the sum of the utilization of all the constituent processor cores. For a vast majority of packet processing systems, higher system utilization means higher sustainable packet throughput.

Observe that identifying the optimal NP configuration is challenging. This is because, multi-threading and caching interact in subtle ways. On the one hand, the sequence of memory accesses that arrive at a cache depend on the order in which threads are scheduled. On the other hand, because of the context-swap-on-cache-miss policy used by processors, the thread scheduling itself is dependent on the cache size. Consequently, identifying the optimal configuration requires an exhaustive search of the design space defined by the multi-processing, multi-threading, and caching dimensions. Unfortunately, even for a uni-processor setting, the design space—as defined by the number of threads per processor, the cache size, miss penalty, and

<sup>1</sup>IXP2800 also includes an XScale core; XScale is used only for less-frequent control-path operations. Hence, we do not consider the chip-area occupied by XScale in our calculations.

context-switch overhead—can be vast. Consider, for instance, the task of identifying optimal configurations of multi-threading and caching in a uni-processor environment, where the total chip-space available for thread-specific and data-cache regions is in the range [1..64] thread-equivalents. For each chip-space availability  $S$ , there are  $S$  possible architectural configurations [(1 thread,  $S - 1$  thread-equivalent data-cache), ..., ( $S$  threads, no cache)]. For 64 different values of  $S$ , this yields 2080 different configurations. Now, if one considers 10 possible values of miss-penalty and 5 different values of context-switch overhead, the total number of configurations that one has to evaluate is given by  $2080 \times 10 \times 5 = 104,000$ . To derive system utilization value for each configuration using SimpleScalar for realistic packet processing application and packet trace requires 30-45 minutes on a 1.6GHz Pentium-4 system (with 512MB memory); hence, the total design space exploration would take 5.9 years!

Our methodology for making the design space exploration tractable relies on the following observations. computations.

- For a vast majority of packet processing applications, the sequence of computational and memory access instructions performed while processing a packet is only dependent on the application logic and the content of the packets. The execution sequence of instructions is independent of the number of threads or the amount of cache available within a processor<sup>2</sup>.
- In the event that: (1) the system is back-logged (i.e., a thread never idles); (2) the memory subsystem serves requests in FIFO order; and (3) the threads are scheduled from the ready-to-run queue in FIFO order; then miss-penalty and context-switch overhead do not affect the order in which threads are scheduled for execution. The miss-penalty and context-switch overhead values only affect the memory access latencies and processor utilization values<sup>3</sup>.

Based on these two observations, we define a three-step process for exploring the architectural design space. First, we generate the trace of computational and memory access instructions (using a modified version of SimpleScalar simulator [8]) executed while processing a sequence of packets (derived from a real packet trace) using a realistic packet processing application in a single-threaded, uni-processor environment. Second, we use a *discrete-event simulator* to determine the order in which these computational and memory access instructions will be interleaved in a multi-threaded processor environment with a given cache size. Third, we determine the overall system utilization and memory bandwidth requirements for different values of miss-penalty and context-switch overhead.

Observe that this three-step process allows us to reuse results of expensive simulations. The application execution is simulated for each packet trace only once. Using this execution trace, we derive multiple scheduling sequences (one for each processor configuration) using the discrete-event simulator. Finally, we re-use the scheduling sequence for a given configuration to determine system utilization and memory bandwidth requirements at various miss-penalties and context-switch overheads. This reduces the total time required for architectural space exploration for a single application and a single packet trace from 5.9 years to about 9 days! In what follows, we explain each of the three steps in greater detail.

### 3.2.1 Execution Trace Generation

To create an execution trace of computational and memory access instructions, we use *SimpleScalar* [8] as our simulation environment. We use the PISA instruction set, since it resembles the simple instruction set supported in today’s network processors (e.g., the micro-engines of IXP2800). Derivation of the execution trace is challenging because of two reasons. First, each memory access in the execution trace should be marked with the corresponding data structure. Since most packet processing applications allocate and deallocate memory dynamically, the simulator 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.

<sup>2</sup>Exceptions to this general observation are applications such as Random Early Detection (RED) used in routers. In this case, a router randomly drops packets based on some thresholds on the input packet-queue. Since the rate at which the input-queue is drained depends upon the number of processors, threads-per-processor, and cache size, the sequence of instructions executed for a packet (in particular, whether or not to drop a packet) would change when the application is run on a system with different number of threads. However, in reality, only a very small number of applications belong to this class; hence, we ignore these exceptional cases from our analysis.

<sup>3</sup>We have formally proved and empirically verified this property; we omit further discussion of this because of space constraints.

We designed and implemented a generic mechanism—referred to as *Simcall*—that meets this requirement. The Simcall mechanism allows applications to provide directives to the simulator. We used this mechanism to inform the simulator of dynamic memory allocations/deallocation as well as to switch the logging on and off at appropriate times during the simulation.

Our enhanced version of the simulator produces an execution 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.

### 3.2.2 Derivation of Schedule Sequence

To derive the scheduling sequence for a multi-threaded processor with a given amount of data cache, we designed a discrete-event simulator. The discrete-event simulator takes as input the execution-trace collected during the previous step. The number of non-memory access instructions executed between successive memory access instructions (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 miss<sup>4</sup>, the simulator switches in the context at the head of the *ready* queue and places the switched-out thread at the back of the *ready* queue. Also on every context switch, the simulator logs the number of instructions executed since the last context-switch.

### 3.2.3 Derivation of System Utilization and Memory Bandwidth

**System Utilization** : In what follows, we illustrate the process of computing system utilization using the scheduling sequence derived in the previous step for different values of miss-penalty; the process for deriving system utilization values for different context-switch overhead is similar.

Let the sequence  $x_1, x_2, \dots, x_n$  denote the schedule sequence where  $x_i$ 's denote the amount of computation performed after the  $i$ th context-switch. Let us assume, without loss of generality, that the threads are identified by integers  $0, 1, \dots, k$  and are scheduled in the increasing order of their ids, where  $k$  is the total number of threads in the processor. Let  $d_i$  be the latest time at which the thread  $i$  is done computing and is swapped out. Let  $c_i$  be the amount computation thread  $i$  performs on being swapped in next. We step through the schedule sequence and assign the next unread  $x_j$  to  $c_i$ . Let  $p$  denote the miss-penalty. Let  $r_i$  denote time at which the thread  $i$  finishes its latest memory read and becomes ready. That is,  $r_i$  can be computed as:  $d_i + p$ . Let  $b_i$  be the time at which the thread  $i$  is going to be scheduled next. Let  $w_i$  be the time the processor idles just after the schedule of the chunk of computation  $x_i$ . By definition  $w_n$  gets a value of zero.

Let us look at the system just after the thread  $i$  missed the cache and is swapped out. Let the corresponding position in the schedule sequence be  $x_m$ . Let us show how to determine the amount of time the processor idles,  $w_m$  before the next thread  $j$  is scheduled and runs for  $c_j = x_{m+1}$  time units. We apply this algorithm repeatedly to determine the time the processor idles before it executes each of the chunks of computation from the schedule sequence. Then, we determine the total time,  $T$ , as  $\sum_{i=1}^m (w_i + x_i)$  Since the thread  $i$  just finished the current time by definition is  $d_i$ . Let  $j$  be the next thread that runs. The time at which  $j$  can run is either the time at which it become ready  $r_j$  or the time at which the the previous thread,  $i$  finishes. Note that  $r_j$  can be computed as  $d_j + p$ . That is,  $b_j = \max(r_j, d_i)$ . Processor idles if  $b_j > d_i$  and the idle time  $w_m$  can be computed as  $b_j - d_i$ . Since we know  $b_j$  and the amount of computation  $c_j = x_{m+1}$ , we can compute when the thread  $j$  leaves the processor -  $d_j$ . Then we can repeat the steps described above to compute the idle time just after the swap-out.

**Memory Bandwidth** : Since the scheduling sequence precisely identifies the total number of cache misses as well as the average number of computational cycles executed between cache misses (and hence, context-switch operations), we derive the memory bandwidth requirement in terms of average number of memory accesses performed per computational cycle.

Observe that the main objective of our analysis is to derive the memory bandwidth requirement for different NP configurations. Our methodology achieves precisely this. We do not evaluate the impact of bandwidth constraints on the performance of different processor configurations; such a characterization is beyond the scope of this paper.

<sup>4</sup>This is a simplified model; many NPs allow applications to issue multiple memory accesses prior to switching context.

## 4 Experimental Setup

### 4.1 Packet Processing Systems

Although the task of creating standard benchmarks for packet processing systems has received much attention lately [31, 40], there does not exist any publicly available suite of packet processing applications. Hence, we identify a set of commonly used applications and construct two significant real-world systems for our analysis.

#### 4.1.1 Selection of Packet Processing Applications

We select our packet processing applications based on the following *semantic* characterization of packet processing systems. All packet processing systems perform the following 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; and (3) process packets.

**Integrity Verification** Checksums are commonly used to verify integrity of packets. The two canonical implementations for checksum computation are IP-Checksum [15] and MD5 [34], which are used, respectively, to verify the integrity of packet header and payload.

**Flow Classification** Flow classification is the process of splitting the stream of incoming packets into sequences of *related* packets. Flow classification involves matching one or more header fields (e.g., addresses and port numbers) 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 [23]. 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 and/or payload. To cover a reasonable spectrum of packet processing functions, we include the following three applications:

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 application. 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 [35] proposed in the literature are well-suited for this function. For our experiments, we use the trie implementation from the BSD Kernel.

Metering [24]: 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.

Pattern matching [41]: A large number of payload processing systems (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 [7]. The process of matching starts by looking for prefixes of patterns in the input text. If found, a hash 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, pattern matcher is an instances of payload-processing functionality.

#### 4.1.2 Building Systems from Applications

Table 1 summarizes the selected packet processing applications. We note that, in reality, typical packet processing systems often contain more than one of these applications. So in order to make our results applicable to reality as much as possible, we chose to not study the applications in isolation. Instead, we built and use two substantial packet processing systems<sup>5</sup>.

A *Quality-of-Service (QoS) Router*: This system 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; and (4)

---

<sup>5</sup>Most of the applications have hundreds of lines of optimized C code (excluding comments); further, our systems consist of more than 1000 lines of C code.

	IP-Checksum	MD5	Hash-Classifer	IP-Forwarder	Meter	Pattern Matcher
Functionality	Integrity Check	Integrity Check	Classification	Header Processing	Header Processing	Payload Processing
Source	Free BSD	R.S.A Inc.	Self	Free BSD	Self	Snort

Table 1: Packet Processing Applications (Self = applications we developed).

	ANL	FRG	MRA	UNC
Trace Length (Million Pkts)	0.42	3.4	5.6	2.6
Unique IP addresses	3488	19442	58695	11913
Link Type and Speed	OC-3 (155Mbps)	OC-12 (620Mbps)	OC-12 (620Mbps)	GigEth (1Gbps)
Link Location	ArgonneNatLab ↔ STARTAP	FrontRangeGigaPOP ↔ Abilene	Merit ↔ Abilene	UNC ↔ ISP

Table 2: Details of Packet Traces

IP forwarder

A *Virus Scanner*: This system 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 system, while virus scanner is a payload-processing system. QoS router is an system 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).

NP-Forum, a consortium of NPU manufacturers, works on benchmarking the performance of packet processing systems. These benchmarks are based on two systems: IP and MPLS forwarding [2]. Our systems (the QoS router in particular) include a significant portion of the processing from both these systems.

## 4.2 Packet Traces and Control Data

We need three kinds of inputs in order to be able to run the systems and collect their execution traces: *packet traces* (representing realistic workload), a *route table*, and a set of *virus signatures*.

- **Packet Traces:** We use Internet packet traces collected at four different sites. Three of them (ANL, FRG and MRA) are provided by NLANR [32] while the fourth one (UNC) is obtained from the University of North Carolina. For each location we have three different traces of 90 seconds duration<sup>6</sup>. They are collected at different times of day. The details of the traces are shown in table 2.

We chose these four sites since they represent four different locations in the Internet hierarchy. On the one extreme is the link at ANL. It connects the Argonne National Lab, a reactively small community, to its service provider. UNC connects the University of North Carolina campus to its service provider and has a number of college and departmental networks underneath it. FRG is similar to UNC in its position in the Internet hierarchy. It connects various universities in the Denver area to the high-speed Internet Abilene. MRA is the link between Merit and Abilene - two large networks. The number of unique IP addresses seen in the traces gives a hint about how big is the set of hosts that use that link. As expected, ANL sees the smallest number of hosts going through it, while MRA has the highest. UNC and FRG lie in between these two extremes.

- **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 [10], 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 use the database of *content signatures* published by Snort [7], 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.

<sup>6</sup>Several 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.

Utilizing these traces and control data for our experiments presents two challenges.

First, the publicly available packet traces are always *anonymized*; to maintain anonymity, the source and the destination IP addresses are overwritten with randomly selected addresses. However, in order to preserve the traffic patterns and flow characteristics, once a source/destination IP address  $A$  is substituted with address  $B$  in a packet, then all the subsequent occurrences of  $A$  are also replaced with  $B$ . An unfortunate side-effect of this anonymization process is that IP addresses contained in these traces do not match any IP address prefixes we collect from RouteViews. Hence, prior to using these traces in conjunction with our route table, we *de-anonymize* them. In particular, we substitute every occurrence of IP address  $B$  in the trace with another randomly selected address  $C$ , 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 system, 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 the *common case* where the packet content does not match any signature in the database. Note that for a virus scanner, this is, in fact, the common case— since the packets with virus signatures are relatively rare (e.g., one in several million packets).

## 5 Results

We studied the two packet processing systems with 12 different packet traces. As shown later in this section, quantitative results for the traces collected from the same location do not vary much leaving the qualitative conclusions unchanged. However the demarcation between the traces from different locations is very apparent, with the ANL and MRA traces often forming the extremes. So during the rest of this section we limit the detailed discussion to the results for the QoS Router system driven by ANL and MRA traces. However, we do provide a summary of all the results in tables 3 and 4. Similarly, we had experimented with a wide-range of miss-penalties and context-switch overheads. However for space reasons, unless otherwise noted, the results shown are for context-switch overhead of 2 and miss-penalty of 200 cycles. We chose these since these values represent the minimum mandatory context-switch overhead and the approximate latency to the SRAM in IXP2800.

During the rest of this section, we consider three architectural schemes: *threads-only*, *cache-only* and *hybrid*. Hybrid processors include both threads and caching. First we study how these schemes compare in the case of a single processor. We then consider the multi-processors. We quantify the performance gains of the hybrid architectures and show how the optimal configuration changes with increasing chip-space. Next we characterize the bandwidth requirements the optimal configurations place on the memory subsystem. Finally we investigate how the system parameters: miss-penalty and context-switch overhead influence the optimum balance.

### 5.1 Single Hybrid Processor

Figure 2 compares, at various available areas (for the thread-specific region), utilizations achieved by the QoS Router system under the three schemes. Figure 3 shows the hybrid configuration at various values of the available area. From these we draw the following conclusions:

- The benefits of the hybrid architecture are significant; even at very small available areas for both ANL and MRA traces. This indicates that the caches are desirable even for very small processors. This is confirmed in figure 3. For both the traces, the optimal hybrid configuration always includes a cache.
- There is a clear demarcation between ANL and MRA traces. The ANL trace benefits from cache more than the MRA. This is so because the ANL represents the network edge while the MRA represents the network core. Since traffic from a larger portion of Internet transits them, core links see larger number of flows and consequently the packets belonging to the same flow are separated by a larger number of unrelated packets. This means core routers see less locality compared to the edges. The relatively less locality seen at MRA is also reflected in the hybrid configurations for the two traces. Configuration for MRA uses twice as many threads compared to that of ANL (figure 3). It is interesting to note that the benefits of a small cache can be significant even in core routers.

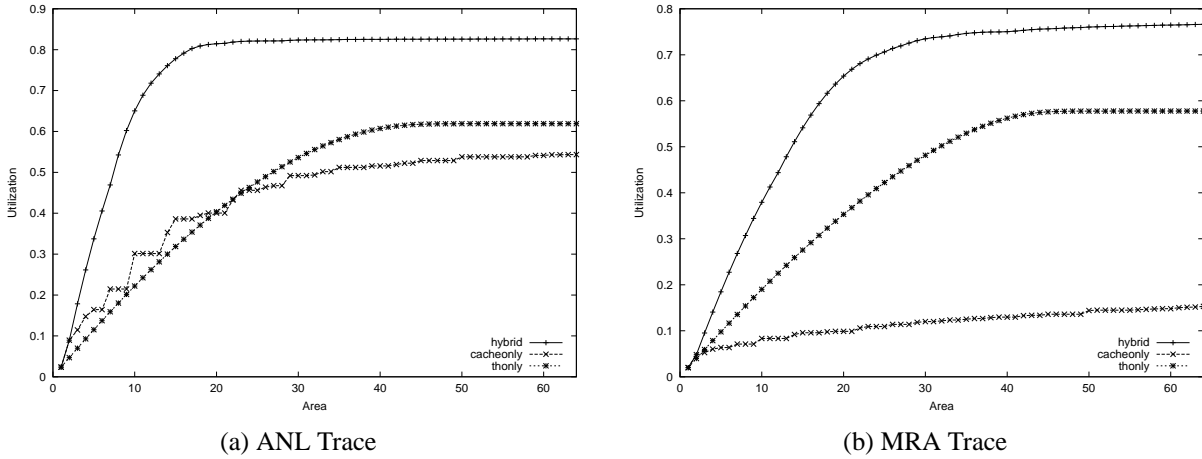


Figure 2: Comparison of utilization achieved by the three architectures

- Note that the available area at which the benefits taper-off for hybrid architecture ( $\approx 15$  and  $24$  for ANL and MRA respectively) is significantly smaller than that of the threads-only scheme ( $\approx 35$  and  $40$  for ANL and MRA respectively). That is, the hybrid processor not only achieves better utilization, but also can be significantly smaller in size. This means, that the hybrid architectures can take better advantage of chip-area and bandwidth when available.
- A micro-engine of IXP2800 has 8 threads and no cache. It can be seen from the plots that, for the same available area of 8, hybrid processor brings up the utilization to 0.54 from 0.18 for the ANL trace. The corresponding increase for MRA is from 0.14 to 0.30. Note that the configuration of the corresponding hybrid processor is not much different from IXP2800. In case of MRA, the hybrid processor has just one less thread which is converted into a small cache.

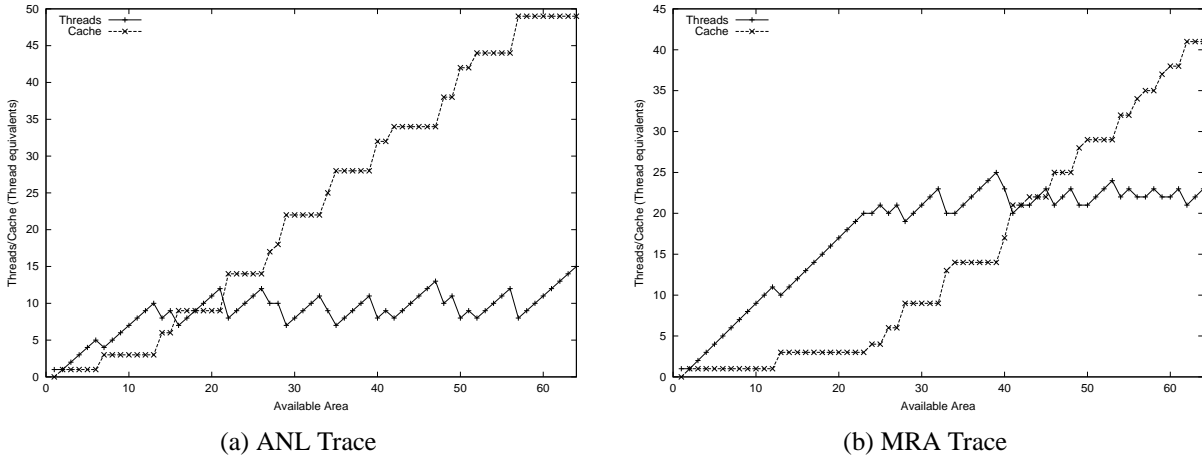
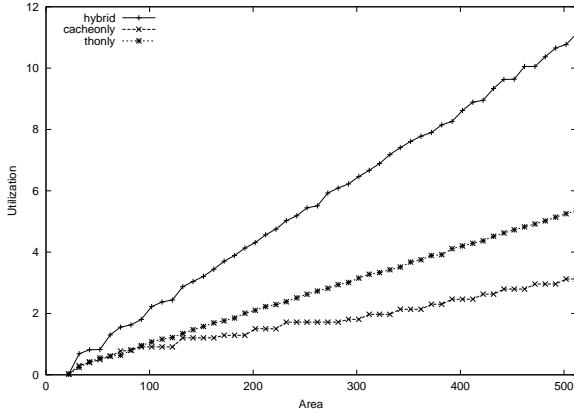


Figure 3: Layout of Hybrid System

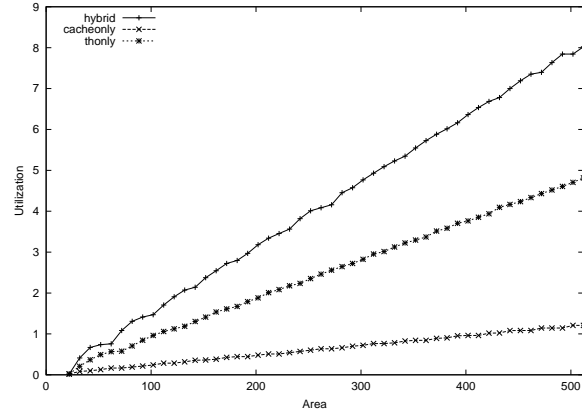
## 5.2 Multi-Processor

In this subsection we consider a multi-processor with identical processors that are based on the same scheme (i.e., threads-only, cache-only or hybrid). We recall from section- 3 that the system utilization is defined as the sum its constituent processors. Figure 4 compares the system utilization of the three types of multi-processors, for the QoS Router system with ANL and MRA traces. The corresponding hybrid configurations (number of processors, number of threads and amount of cache per processor) are shown in figure 5.

As explained in section- 3, the area of the IXP2800 that is devoted to the micro-engines (packet processing cores) is roughly about 455 thread equivalents. At the available space of 455, for ANL trace, the hybrid architecture achieves 100% increase in

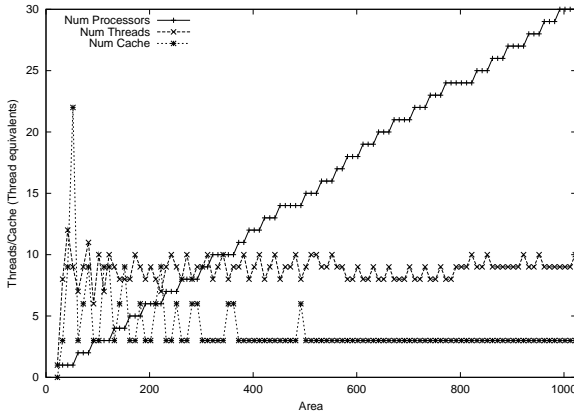


(a) ANL Trace

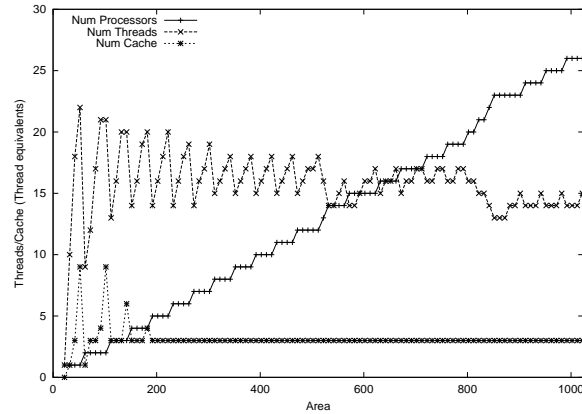


(b) MRA Trace

Figure 4: Comparison of utilization achieved by the three architectures



(a) ANL Trace



(b) MRA Trace

Figure 5: Layout of the Hybrid Multi-processor System

utilization (from 4.7 to 9.6) when compared to an optimal threads-only multi-processor. In case of MRA, the utilization goes up by about 70% from 4.23 to 7.19. Note that these comparisons are with a threads-only processor that has the optimum number of processor and threads per processor which performs significantly better than a processor with the configuration of IXP2800. See the summary of results in Table 3.

From figure 5 it can be seen, that with the increasing chip-area, the optimal configuration for the individual processor converges to the configuration corresponding to the “knee” in the utilization of the single hybrid processor(see figures 2, 3). So in a nutshell, the optimal multi-processor maximizes the number of processors (since the utilization grows linearly), such that the individual processor, in terms of the configuration, is as “close” to the knee as possible.

Tables 3, and 4 show the complete set of optimal configurations and corresponding utilizations for both the systems: QoS Router and Virus Scanner run with all the 12 different traces. From these we make the following observations.

The configuration stays relatively constant within the set of traces collected from the same location. Since these are collected at different points in time it seems to indicate that the trace characteristics that influence the architectural balance such as the locality and the average amount of computation needed per memory reference stays relatively constant. However, there is a clear distinction across traces. This distinction is more pronounced in case of QoS Router than the Virus Scanner. This is so because, the Virus Scanner has a significant portion of accesses that are due to the pattern matching on packet payload and they do not benefit from the locality in the trace.

On the other hand, the QoS Router is influenced by the trace since most of its processing benefits from the locality in the

	ANL			FRG			MRA			UNC		
Processors	14	14	13	13	13	13	11	11	11	13	13	12
Threads	8	8	8	11	10	10	17	17	17	11	11	13
Cache(TEQ)	3	3	6	3	4	4	3	3	3	3	3	3
Utilization(Balanced)	9.64	10.25	9.84	9.30	9.69	8.94	7.47	7.39	7.19	8.15	8.08	7.87
Utilization(CacheOnly)	2.79	3.47	2.73	2.33	2.84	2.04	1.20	1.17	1.08	1.56	1.52	1.42
Utilization(ThrdsOnly)	4.72	4.61	4.66	4.40	4.35	4.34	4.32	4.27	4.23	4.53	4.48	4.49
Utilization(IXP2800-like)	2.88	2.79	2.82	2.60	2.56	2.55	2.53	2.49	2.46	2.71	2.67	2.67

Table 3: Summary of Results for the QoS Router (Miss-penalty: 200 and Context-switch overhead: 2)

	ANL			FRG			MRA			UNC		
Processors	14	14	13	13	13	13	13	13	13	13	13	13
Threads	8	8	8	8	8	10	11	10	11	10	10	10
Cache(TEQ)	3	3	6	6	6	4	3	4	3	4	4	4
Utilization(Balanced)	9.62	9.93	9.70	9.56	9.65	9.41	8.97	8.83	8.60	9.34	9.26	9.23
Utilization(CacheOnly)	2.68	2.88	2.64	2.50	2.611	2.36	1.93	1.76	1.84	1.99	1.97	1.91
Utilization(ThrdsOnly)	4.35	4.34	4.33	4.29	4.33	4.27	4.26	4.27	4.25	4.33	4.32	4.36
Utilization(IXP2800-like)	2.30	2.29	2.27	2.25	2.31	2.22	2.23	2.21	2.28	2.27	2.27	2.27

Table 4: Results Summary for Virus Scanner (Area: 455 Miss-penalty: 200 Context-switch: 2)

traffic and the link’s position in the Internet hierarchy does affect the locality seen. Note, for example, the MRA trace needs twice as many threads compared to ANL. It is interesting to observe the clear trend of moving towards smaller number of bigger processors with more trends as we go from the edge (ANL) to core (MRA) via the intermediate locations (UNC and FRG).

Note that the performance of optimal-threadonly and IXP2800-like processors are fairly independent of the traces. This is so since their performance is solely dictated by the average computation done per memory reference of the application and is independent of the locality in the trace.

### 5.3 Requirements on the Memory Subsystem

In this subsection we characterize the requirements different architectural designs place on the memory subsystem. As discussed in section 3 our goal is to determine how the bandwidth requirements change as we explore the architectural design space. In particular, we ask how well a particular processor architecture utilize the bandwidth if and when available.

Please note that it is not the focus of this study to determine how any particular feature of the memory subsystem (such as the queuing and scheduling of the memory requests) would affect a particular processor configuration. We measure the average bandwidth of a configuration assuming a simple but generic memory subsystem so our results can be applied to wider-variety of circumstances. In particular, we assume that the access latency is constant. We measure the bandwidth in the units of 64-bit(cache line-length) references per processor cycle.

Figure- 6 compares the ability of the three architectures to utilize the memory bandwidth when available. It plots, for each scheme, the utilization of the best possible system configuration that has its average bandwidth requirements below a given value (on the x-axis). The chip-area for all the systems has been fixed at 455 thread equivalents. while the miss-penalty and context-switch overheads are 200 and 2 cycles respectively.

We make the following observations: Cache-only system fails to exploit bandwidth beyond a small value (0.02 for ANL 0.05 for MRA) since cache-only architectures at a given chip-space has a constant amount of bandwidth requirement determined by the miss-rate at that cache size. So to be able to exploit any more of the available bandwidth, cache-only systems need more space than 455. That is, cache-only system becomes “space-limited”. On the other hand, for the threads-only system the utilization continues to grow with the bandwidth indicating its bandwidth-limited nature.

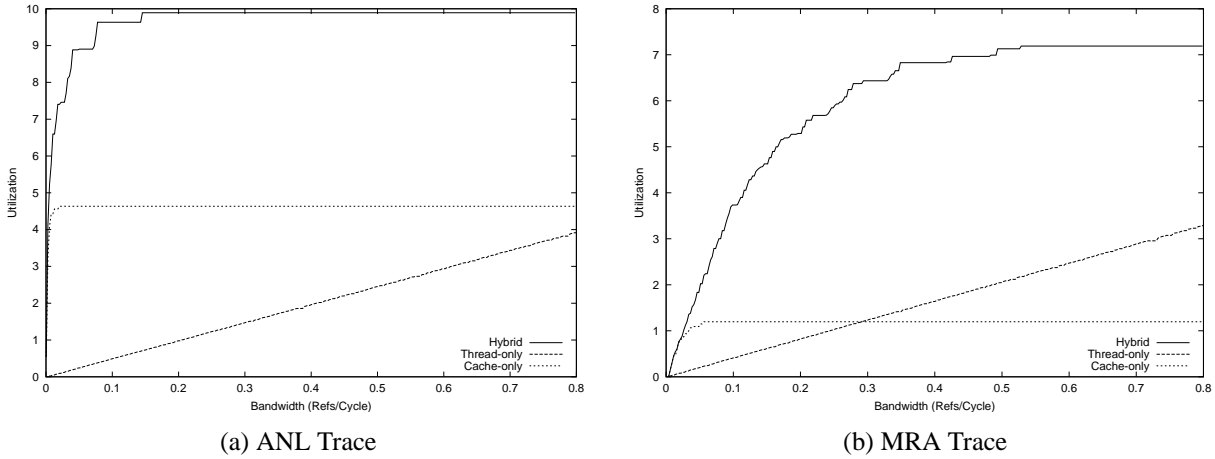


Figure 6: Utilization as a function of Peak Bandwidth Availability

The hybrid system becomes space-limited at a bandwidth value that is smaller than that of the threads-only and larger than that of the cache-only. This illustrates the ability of the hybrid architecture to exploit both bigger chip-spaces and available bandwidth. Note that the larger number of threads for MRA reflects in the hybrid system using up more bandwidth than ANL before becoming space-limited.

Using the cycle accurate simulator available in the IXA SDK-3.5 We measured the peak QDR SRAM bandwidth available to the micro-engines. The aggregate bandwidth of the four parallel memory controllers is 0.5357 references per micro-engine cycle. Since the plots in figure 6 are for the IXP equivalent area of 455, we can see that for the bandwidth available to the micro-engines (0.5357) the hybrid system can achieve as much as 200% percent increase in utilization for the MRA trace.

#### 5.4 Effect of Miss-penalty and Context-switch Overhead

In this subsection let us examine how the optimal balance changes with the two system parameters: miss-penalty and context-switch overhead.

Figure- 7 shows how the optimal configuration varies as the miss-penalty increases. We draw the following conclusions. With increasing miss-penalty, the number of processors goes down, since at higher miss-penalties more threads and/or cache is required to keep each of the functional units busy. It is interesting to note that as the miss-penalty increases, the difference between the optimal configuration further gets amplified. For example the optimal configurations at penalty 1000 ANL consists of 10 processors 10 threads per processor and 14 thread equivalents of cache while that of MRA has 6 processors each with 40 threads and 14 thread equivalents of cache.

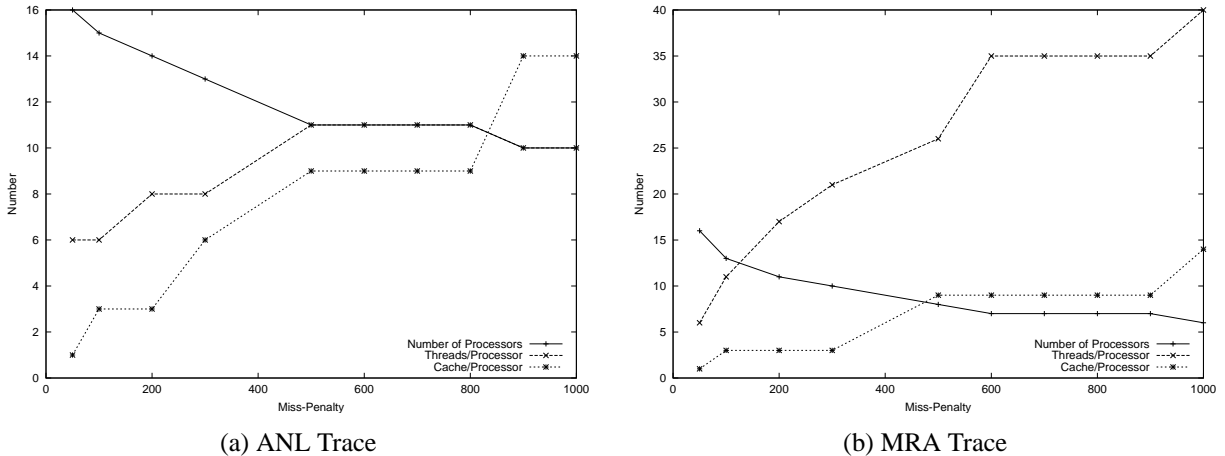


Figure 7: Effect of Miss-Penalty on the Hybrid Layout

Figure- 8 shows how the optimal configuration varies as the context switch overhead increases. Since higher context-switch overheads make the threads less effective in masking the memory latency, the optimal configuration includes smaller number of threads. Since the edge trace from ANL benefits from larger caches, the optimal configuration increases the cache from 3 thread equivalents to 9. However, MRA because of its lack of locality, does not use caches. Instead it uses the freed up space due to smaller number of threads, to layout more processors.

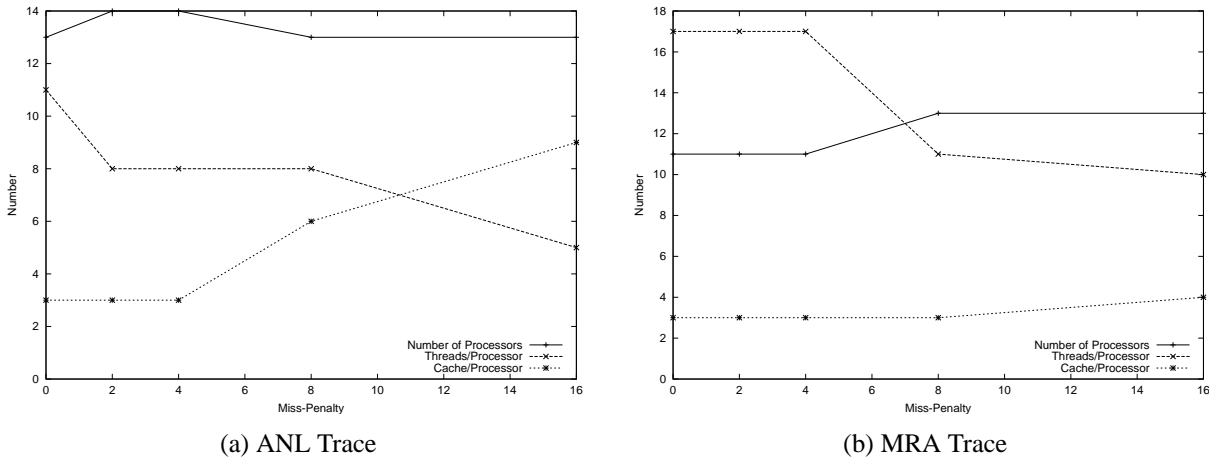


Figure 8: Effect of Context-Switch overhead on the Hybrid Layout

## 6 Related Work

Caching and threading have been well-known techniques for bridging the gap between the CPU and memory speeds. These techniques have been studied thoroughly in isolation for a long time. In the recent past they have been compared to each other in various settings. A quantitative comparison of various latency management techniques including hardware contexts and hardware coherent caches can be found in [22]. The main focus of this work, however, lies in quantifying the benefits of hardware coherence mechanisms in a shared memory multi-processor. The discussion in [14] compares qualitatively the effectiveness of simultaneous multi-threading and caches in improving the processor utilization.

Multi-threading and caching have also been studied analytically in [11] and [36]. In [11] it is assumed that the working-set sizes of individual threads and the amount of data-sharing among threads can be precisely known and hence are modeled as constants. However, in practice it is not very clear as to how these quantities should be determined. A simpler model used in [36] assumes that the threads split the available cache evenly.

Interactions of caches with operating system controlled software threads (e.g., *threads*) running scientific applications have been studied in [30].

None these studies [22, 14, 11, 36, 30] view the techniques of threading and caching as mechanisms that compete for common resource such as chip-area. They focus on explaining the performance based on the interactions. They do not consider the problems associated with either determining the right balance or the exploration of the design space.

Recently many architectural design space explorations dealt with the trade-offs related to CMPs. An exploration along the dimensions of number of processing cores and cache per core can be found in [19]. A more recent study [25] explores the CMP design space along three dimensions: number of processing cores, granularity of each core and the size of on-chip caches. This line of work considers only general purpose computing.

Traditional processor architectures such as SMT, superscalar, and CMP have been evaluated for their suitability for network processing in [16, 17]. Their study uses a fixed amount of cache and focuses on quantifying the ability of the candidate architectures to exploit ILP. An analytical design space exploration for network processors is carried out in [20]. This study, as done in [36] assumes that the threads split the cache evenly among them and hence does not consider the interactions between threading and caching.

The simulation techniques for architectural experiments have received considerable attention. Simpoint [37] does an off-line analysis of the program and simulates only selective phases of the run. Techniques for efficient cache have been proposed in [38, 29]. However these techniques cannot be adapted in a straightforward manner for the multi-threaded execution environment we work with.

## 7 Conclusions

The performance of packet processing applications is limited by memory access latencies and bandwidth. Today's network processors rely upon multi-processing and multi-threading to achieve high packet throughputs. However, they generally don't utilize caches. Recent studies have shown that moderate-sized data caches can be effective in improving packet throughput. In this paper, we explore the architectural design space defined by multi-processing, multi-threading, and caching to identify optimal configuration for network processors.

We make three primary contributions. First, we present a methodology for exploring the architectural design space for NPs systematically and efficiently. Design space exploration is challenging because: (1) multi-processing, multi-threading, and caching interact in subtle ways, and (2) the vast design space makes straightforward simulations prohibitively expensive. Second, we show that a balanced configuration—derived with the chip-area and memory bandwidth of Intel's IXP2800—can achieve upto 3 times the packet processing throughput as compared to a processor with IXP2800's configuration. Third, we show that packet trace characteristics influence the architectural balance significantly; in particular, optimal NPU architectures for the network edge can differ significantly from those for the network core.

## References

- [1] <http://www.caida.org/analysis/workload>.
- [2] Benchmarks, Network Processing Forum. <http://www.npforum.org/benchmarking/index.shtml>.
- [3] IBM PowerNP Network Processors. [http://www-3.ibm.com/chips/techlib/techlib.nsf/products/IBM\\_PowerNP\\_NP4GS3](http://www-3.ibm.com/chips/techlib/techlib.nsf/products/IBM_PowerNP_NP4GS3).
- [4] Intel IXA Software Developers Kit 3.0. <http://www.intel.com/design/network/products/npfamily/sdk3.htm>.
- [5] Netfilter and iptables. <http://www.netfilter.org>.
- [6] The reincarnation of network processor market. In-Stat MDR, December 2003.
- [7] Snort: The Open Source Network Intrusion Detection System. <http://www.snort.org/>.
- [8] The SimpleScalar Tool Set Version 3.0. <http://www.simplescalar.com/>.
- [9] The Tolly Group: Information Technology Testing, Research and Certification . <http://www.tolly.com>.
- [10] University of Oregon Route Views Project. <http://www.routeviews.org/>.
- [11] A. Agarwal. Performance Tradeoffs in Multithreaded Processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, September 1992.
- [12] Anonymous. A Case For Data Caches in Network Processors.
- [13] F. Baker. Requirements for IP Version 4 Routers. IETF RFC 1812, June 1995.
- [14] A. Bakshi, J.-L. Gaudiot, W.-Y. Lin, M. Makhija, V. K. Prasanna, W. Ro, and C. Shin. Memory Latency: to Tolerate or to Reduce? In *12th Symposium on Computer Architecture and High Performance Computing. Invited paper*, June 2000.
- [15] R. Braden, D. Borman, and C. Partridge. Computing the Internet Checksum. IETF RFC 1071, September 1988.
- [16] 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.
- [17] P. Crowley, M. E. Fiuczynski, and J.-L. Baer. On the Performance of Multithreaded Architectures for Network Processors. Technical Report TR2000-10-01, University of Washington, 2000.
- [18] K. Egevang and P. Francis. The IP Network Address Translator (NAT). IETF RFC 1631, May 1994.
- [19] M. Farrens, G. Tyson, and A. R. Pleszkun. A study of single-chip processor/cache organizations for large numbers of transistors. In *Proceedings of the 21ST annual international symposium on Computer architecture*, 1994.
- [20] M. A. Franklin and T. Wolf. A Network Processor Performance and Design Model with Benchmark Parameterization. In *Proceedings of the Network Processor workshop in conjunction with the Eighth International Symposium on High Performance Computer Architecture (HPCA8)*, Cambridge MA, USA, February 2002., February 2002.
- [21] A. O. Freier, P. Karlton, and P. C. Kocher. The SSL Protocol Version 3.0. Internet Draft, November 1996.

- [22] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W.-D. Weber. Comparative evaluation of latency reducing and tolerating techniques. In *Proceedings of the 18th International Symposium on Computer Architecture (ISCA)*, pages 254–265, New York, NY, 1991. ACM Press.
- [23] P. Gupta and N. McKeown. Algorithms for Packet Classification. In *IEEE Network*, March/April 2001.
- [24] J. Heinanen, T. Finland, and R. Guerin. A Two Rate Three Color Marker. IETF RFC 2698.
- [25] J. Huh, S. W. Keckler, and D. Burger. Exploring the Design Space of Future CMPs. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques, PACT 2001*, pages 199–210, September 2001.
- [26] *Intel IXP2800 Network Processor Hardware Reference Manual*, November 2002.
- [27] *Intel IXP2400 Network Processor Hardware Reference Manual*, January 2003.
- [28] V. Jacobson. Compressing TCP/IP Headers for Low-speed Serial Links. IETF RFC 1144, February 1990.
- [29] S. F. Kaplan, Y. Smaragdakis, and P. R. Wilson. Trace reduction for virtual memory simulations. In *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 1999.
- [30] H. Kwak, B. Lee, A. R. Hurson, S.-H. Yoon, and W.-J. Hahn. Effects of Multithreading on Cache Performance. *IEEE Transactions on Computers*, 48(2), February 1999.
- [31] G. Memik, W. H. Mangione-Smith, and W. Hu. NetBench: A Benchmarking Suite for Network Processors. In *ICCAD*, 2001.
- [32] NLANR Network Traffic Packet Header Traces. <http://pma.nlanr.net/Traces/>.
- [33] 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.
- [34] R. Rivest. The MD5 Message-Digest Algorithm. IETF RFC 1321.
- [35] M. Ruiz-Sanchez, E. Biersack, and W. Dabbous. Survey and Taxonomy of IP Address Lookup Algorithms. *IEEE Network Magazine*, March 2001.
- [36] R. H. Saavedra-Barrera, D. E. Culler, and T. V. Eicken. Analysis of Multithreaded Architectures for Parallel Computing. In *Proceedings of the Second Annual ACM Symposium on Parallel Architectures and Algorithms (SPAA)*, Island of crete, Greece, 1990.
- [37] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th international conference on architectural support for programming languages and operating systems*. ACM Press, 2002.
- [38] R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, 1993.
- [39] G. Tsirtsis and P. Srisuresh. Network Address Translation - Protocol Translation (NAT-PT). IETF RFC 2766, February 2000.
- [40] 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.
- [41] S. Wu and U. Manber. A Fast Algorithm for Multi-pattern Searching. Technical Report TR-94-17, 1994.