Parallel Programmable Ethernet Controllers: Performance and Security

Derek L. Schuff and Vijay S. Pai, Purdue University
Paul Willmann and Scott Rixner, Rice University

Abstract

Programmable network interfaces can provide network servers with a flexible interface to high-bandwidth Ethernet links, but they face critical software and architectural challenges. This article explores architectural and software support for an efficient programmable 10 Gigabit Ethernet controller. The design is then extended to support a self-securing Gigabit Ethernet controller that performs intrusion detection on all network data frames. Both raw performance and security require high-bitrate frame data transfer, low-latency metadata access, and intensive computational capacity while still operating under the area, cost, and power budget of a peripheral device. These goals are achieved using a combination of parallel lightweight processing cores, an explicitly-partitioned memory system, and dedicated hardware assists. The firmware on the network interface is designed to utilize these resources efficiently by exploiting frame-level, flow-level, and task-level concurrency.

Modern computer systems increasingly rely on almost constant communication with the outside world. Ethernet bandwidth to individual computers has continued to increase at an exponential rate to accommodate this demand. Today, 1 Gb/s Ethernet is becoming common for desktop systems, and 10 Gb/s Ethernet is available for server systems. Nevertheless, actually accessing the network at these rates poses serious hardware and software challenges on the network interface that stem from high computational requirements, large volumes of data traffic, security concerns, and the need for system responsiveness. Furthermore, unlike a network interface in a router, a network interface card (NIC) in a server also must tolerate high latency communication with the system host processor for all incoming and outgoing frames. All of these challenges must be met subject to the constraints of a peripheral within the server, limiting the area and power consumption of the NIC.

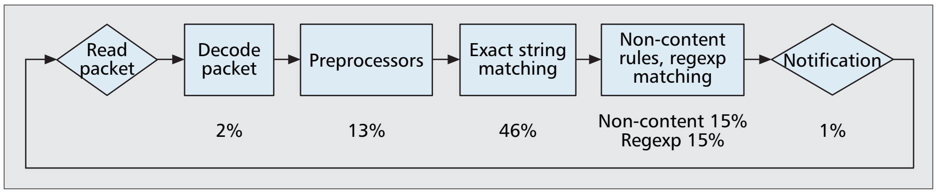
The NIC of the future must provide not only high-bandwidth access to the network, but also must support extensible services, most notably for security. Programmable NICs can serve as a flexible and extensible interface to such high-bandwidth Ethernet links.

This article presents the design of an efficient programmable Ethernet network interface and extensions to enable such an interface to support network intrusion detection. This article presents a hardware architecture for programmable network interfaces that splits protocol processing work across low-frequency parallel reduced instruction set computer (RISC) cores (rather than a single power-hungry high-frequency core). This architecture can easily achieve full-duplex 10 Gb/s throughput. This article then presents and evaluates a strategy for integrating security into this programmable network interface architecture by re-targeting the popular Snort network intrusion detection system to exploit parallelism across independent network flows. The proposed architecture and firmware achieves gigabit packet inspection rates. This draws together the authors' previous work in network interface design and network intrusion detection [1, 2].

Background

Network Interfaces

The host operating system of a network server uses the network interface to send and receive packets. The operating system stores and retrieves data directly to or from the main memory, and the NIC transfers this data to or from its own local transmit and receive buffers. Sending and receiving data is handled cooperatively by the NIC and the device driver in the operating system. These two components notify each other when data is ready to be sent or has just been received, and they must work together as only the device driver has access to host OS structures, and only the NIC has direct access to the network. This coordination requires both the NIC and the device driver to maintain state information about frames that currently are being processed. Managing this state to enable the NIC to send and receive frames requires several different types of communication between the device driver and the network interface. The device driver uses programmed input/output (I/O) to notify the NIC that there is new data to transmit or new buffer space for received data. The NIC uses direct memory accesses (DMA) to host memory to transfer frames between the host and the NIC. The NIC also uses interrupts to notify the host when frames have been sent or received. Finally, the NIC also must manage access to the actual network via the media access control (MAC). The NIC-side tasks may be performed using either a programmable or hardwired application-specific integrated circuit (ASIC) net-



■ Figure 1. *The Snort packet processing loop with percentage of time spent in each phase.*

work interface, but programmable interfaces are attractive because they may be used to offload various services from the host, such as TCP/IP processing [3], iSCSI [4], or network interface data caching [5].

Detailed analysis of the previous tasks on a programmable network interface shows that on average, sending a full-sized Ethernet frame requires 281 RISC instructions and 404 bytes of metadata access, and receiving an Ethernet frame requires 253 instructions and 340 bytes of metadata accesses [1]. A full-duplex 10 Gb/s link can deliver maximum-sized 1518-byte frames at the rate of 812,744 frames per second in each direction. Therefore, a full-duplex 10 gigabit Ethernet controller must be able to sustain 435 MIPS and 4.8 Gb/s of data bandwidth for protocol processing. Additionally, the frame data must be touched twice on every transmission: once for a transfer between the NIC memory and the host and once for a transfer between the NIC memory and the network. Thus, sending and receiving maximum-sized frames at 10 Gb/s requires an additional 39.5 Gb/s of bandwidth for frame data. This is slightly less than the overall link bandwidth would suggest (2*2*10 Gb/s), because data cannot be sent during the Ethernet interframe gap.

Traditional solutions for achieving high performance in conventional processors are not applicable to network interfaces. Instruction-level parallelism requires high complexity (such as wide issue windows, register renaming hardware, and complex branch prediction), all of which add area, delay, and power consumption. High-frequency processors are also not an option because power consumption grows superlinearly as frequency increases, leading to a less-efficient peripheral. Further, techniques commonly used in network processors designed for routers (such as multithreading) are not applicable for NICs; unlike router interfaces, server NICs must tolerate the long latencies of DMA communications with the host and thus must use more aggressive latency-tolerance techniques (such as event-driven firmware).

Intrusion Detection

Network intrusion detection systems (NIDS) aim to detect (and in some cases prevent) attempted intrusions and vulnerability exploits by inspecting the content of every packet destined for a system or network. Snort is the most popular NIDS available. The system and its intrusion-detection rule set are freely available, and both are regularly updated to account for the latest threats [6]. Snort rules detect attacks based on traffic characteristics such as the protocol type (TCP, UDP, ICMP, or general IP), the port number, the size of the packets, the packet contents, and the position of the suspicious content. Packet contents can be examined for exact string matches and regular expression matches. Snort can perform thousands of exact string matches in parallel using one of several different multi-string pattern matching algorithms, with the Aho-Corasick algorithm as the default [7]. Snort also includes *preprocessors* that perform certain operations on the data stream, such as TCP stream reassembly (to avoid missing attacks that are split across packets) or rewriting uniform resource locators (URLs) in HTTP canonical format (e.g., replacing "`%7e`" with a tilde or eliminating "../" directory traversals).

Common rules include checks for buffer-overflow attacks, cross-site scripting, or distributed denial-of-service attacks. Each of these rules includes multiple test conditions and uses the logical AND of those conditions to confirm an attack. The Snort ruleset language includes 15 tests based on packet payload and 20 based on headers. Over 95 percent of the rules in recent Snort rulesets specify string content to match, but no rule specifies *only* string content matching. Some rules specify multiple string matches; about 30 percent use regular expressions; and all are augmented with other tests, such as specific ports, IP addresses, or URL.

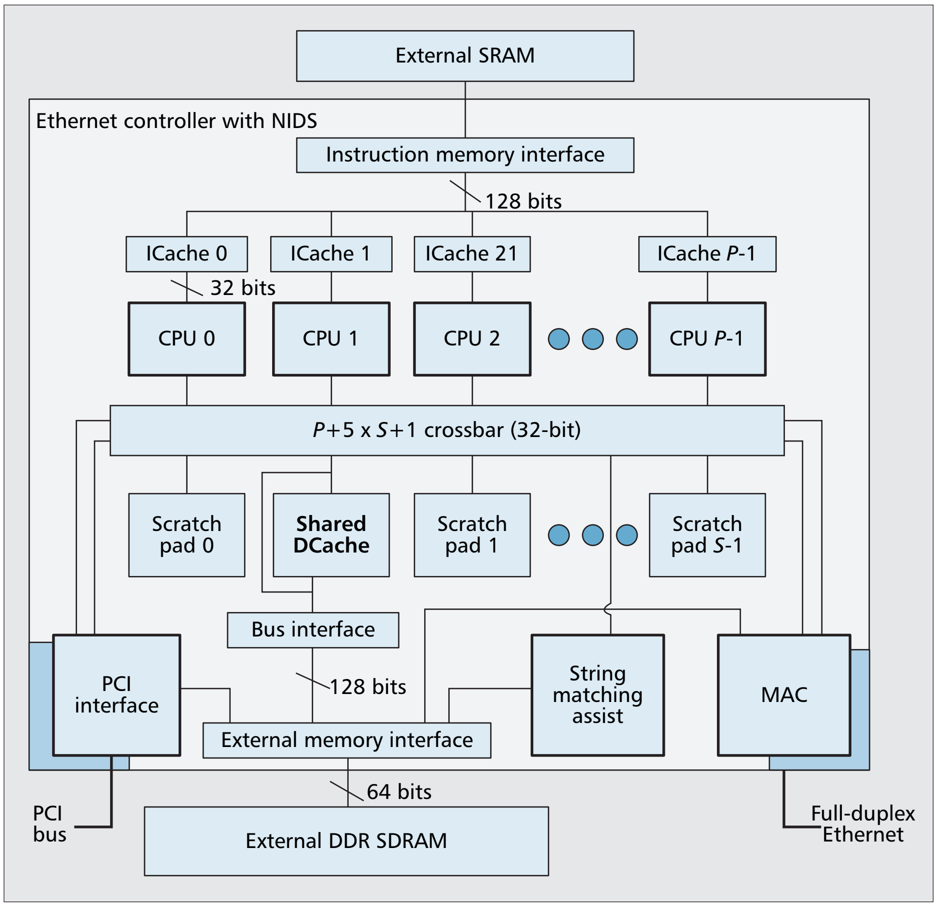
Figure 1 depicts the stages of the Snort packet processing loop along with the percentage of execution time spent on each one when invoked with a representative network trace [8]. The profile shown here was gathered using the `oprofile` full-system profiling utility on Linux, running on a Sun Fire v20z with two 2.0 GHz dual-core Opterons (four processors total). Snort runs on only one processor. The Snort configuration used a full recent ruleset and included the preprocessors described previously. The Snort code profiled here is modified to read all of its packets sequentially from an in-memory buffer to allow playback of a previously-captured network trace and avoid the overhead of capturing packets from the operating system (OS). The system achieves 463 Mb/s inspection throughput for this trace, with packet inspection consuming 92 percent of execution time (the remaining 8 percent is consumed by tasks that are not part of Snort packet processing, such as miscellaneous library calls, operating system activity, the profiler, and other applications). Because the processors of an actual running server machine also must be shared with the server applications and the OS, it is not feasible to deploy Snort directly on high-end network servers that must serve data at gigabit rates or higher. These observations tend to limit the deployment of NIDS to edge appliances, as in commercial developments by F5, Radware, and other companies. Edge-based NIDS is effective at detecting intrusions from the Internet, but provides no coverage for attacks within the local area network (LAN) that might occur as follow-on attacks after one external attack slips through the perimeter.

Figure 1 indicates that string content matching is a major component of intrusion detection, constituting 46 percent of execution time. Similar observations have led others to design dedicated hardware specifically aimed at string content matching; various solutions based on ASICs, field-programmable gate arrays (FPGAs), and ternary content addressable memories (TCAMs) have been shown to match content at line rate (e.g., [9–11]). However, the other stages of NIDS are also equally important. This is not unexpected since the rules invariably include multiple types of tests, not just string content matching. Thus, any performance optimization strategy should target the full intrusion detection system.

Programmable Network Interface Architecture

This article proposes a self-securing Ethernet controller that runs event-driven protocol processing firmware on multiple RISC processors augmented with special-purpose hardware to accelerate the string-matching portion of intrusion detection. Figure 2 shows the architecture of the proposed Ethernet controller. The specific portions of the architecture target the following uses:

- Programmable processors: control-intensive computation; require low latency
- Memory transfer assists (DMA and MAC): data transfers to external interfaces; require high bandwidth



■ Figure 2. *Block diagram of proposed Ethernet controller architecture.*

- String-matching hardware: data processing; requires high bandwidth
- Graphics double-data-rate (GDDR) synchronous dynamic random access memory (SDRAM): provides high bandwidth and high capacity for network data and dynamically allocated data structures
- Banked static random access memory (SRAM) scratchpads: provides low latency for fixed control data access
- Instruction caches: provides low latency instruction access
- Data cache: provides low latency on repeated accesses of dynamic NIDS data structures

The following discusses the architectural components in greater detail.

Programmable Processing Elements

Ethernet protocol processing and the portions of Snort other than string matching are executed on RISC processors that use a single-issue five-stage pipeline and an instruction set based on a subset of the MIPS architecture. Parallelism is a natural approach to achieving performance in this environment because packets from independent flows have no semantic dependencies between them. The processor count is varied from 1–8 in this study, and the frequency is varied from 100–500 MHz.

Memory Transfer Assists

This system includes hardware engines to transfer data between the NIC memory and the host memory or network efficiently. These assists, called the DMA and MAC assists, operate on command queues that specify the addresses and lengths of the memory regions to be transferred to the host by DMA or to the network following the Ethernet medium-access control policy. The DMA assist also offloads TCP/IP checksumming from the host operating system.

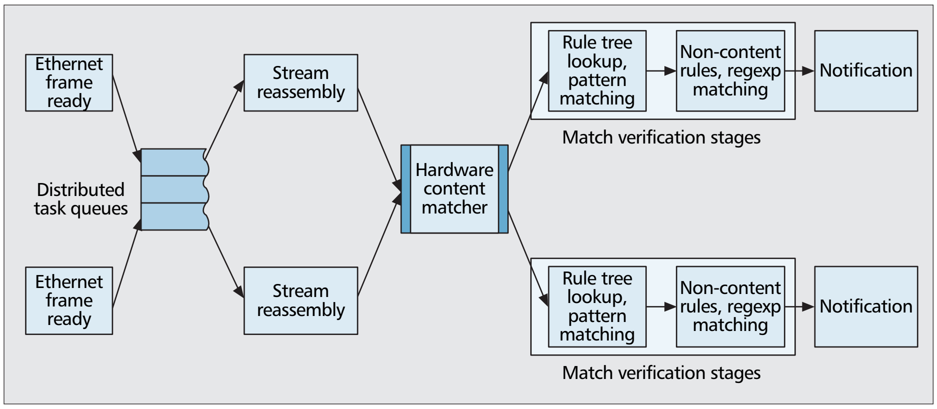
String Matching Hardware

High-performance network intrusion detection requires the ability to match string content patterns efficiently, so the controller integrates string-matching-assist hardware. The string-matcher operates by reading from a dedicated task queue describing regions of memory to process and then reading those regions of memory (just like the DMA/MAC assists). It then feeds that data to a simulated hardware implementation of the Aho-Corasick algorithm. The actual implementation could be based on many previously proposed ASIC, TCAM, or FPGA-based designs, many of which have been shown to match content at line rate (e.g., [9–11]). The only potential slowdown would be reading the input data from memory; fortunately, the memory hierarchy provides sufficient bandwidth.

Memory System

The memory system consists of small instruction caches for low-latency access to the instruction stream, banked SRAM scratchpads for access to protocol processing and Snort control data, and an external GDDR SDRAM memory system for high-bandwidth access to high-capacity frame data. The instruction caches, though very small, have high enough hit rates that instruction access is not a bottleneck in this system. The scratchpads used in this study have four banks and run at the processor clock frequency. Two 500 MHz Micron GDDR SDRAM chips provide 64 Mbytes of memory with a 64-bit interface yielding a peak bandwidth of 64 Gb/s. Additionally, for the NIDS firmware, the memory system includes a shared data cache to enable the programmable processors to use dynamic random access memory-based (DRAM) frame data when required without incurring row activation overhead on every access. The cache is not used for scratchpad accesses or by the assists. For simplicity, the cache is write-through; consequently, it is important to avoid allocating heavily-updated performance-critical data in the DRAM. The cache also does not maintain hardware coherence with respect to the assists; instead, the code must explicitly flush the relevant data from the cache any time it must communicate with the assists.

The processors and each of the five hardware assists connect to the scratchpads through a crossbar as in a dancehall architecture. Arbitration for a given scratchpad requires a cycle, and actual scratchpad access requires another cycle. Although this arbitration cost could be tolerated by extending the processor pipeline, our work makes no changes to the processors and thus experiences an additional cycle delay on each data load from the scratchpad. There also is a crossbar connection to allow the processors to connect to the shared data cache or the external memory interface; the assists access the external memory interface directly to facilitate longer burst transfers.



Ethernet frame ready
Stream reassembly
Rule tree lookup, pattern matching
Non-content rules, regexp matching
Notification
Match verification stages
Distributed task queues
Hardware content matcher
Ethernet frame ready
Stream reassembly
Rule tree lookup, pattern matching
Non-content rules, regexp matching
Notification
Match verification stages

■ Figure 3. *Firmware parallelization strategy used for NIC-embedded Snort.*

Network Interface Firmware

The firmware of a programmable network interface is responsible for using the architecture provided to perform protocol processing and other extended services. The key challenge in this system is to exploit parallelism efficiently without violating network ordering constraints.

Protocol Processing

To tolerate the long latencies of interactions with the host, NIC firmware uses an event-based processing model in which the various steps of data communication map to separate events. When an event is triggered, the firmware runs a specific *event handler* function for that type of event. Events may be triggered by hardware completion notifications (e.g., packet arrival, DMA completion) or by other event handler functions that wish to trigger a software event (e.g., processing a newly arrived packet descriptor to fetch the actual packet contents). The protocol processing firmware used in this study exploits both frame-level and task-level parallelism, dividing work into bundles of specific frames that require a certain type of processing.

The firmware is a parallel program that executes identically on all processors in the system. Each processor executes the work-discovery dispatch loop, which inspects a distributed task queue and several hardware-maintained pointers. The distributed task queue contains software-generated events, and the hardware-maintained pointers indicate the completion of hardware events such as packet arrivals. After work units are discovered (subject to synchronization to ensure correctness), they can be executed in parallel, regardless of the type of processing required. This frame-level parallel organization enables high levels of concurrency but requires additional overhead for work discovery and to maintain frame ordering. Whereas the firmware work-discovery and task-management orchestration is entirely new, the core task-processing functions are based on those used in Revision 12.4.13 of the Alteon Websystems Tigon-II firmware. However, that task handling code has been extended to make the task processing functions re-entrant and to apply synchronization to all data shared among different tasks.

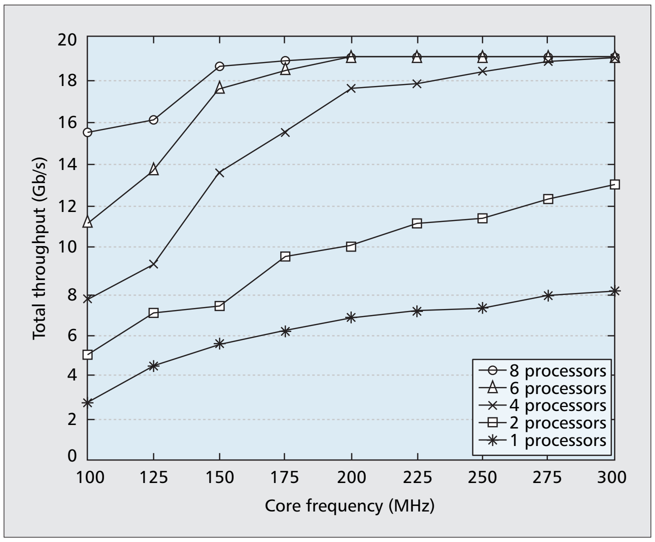
A side effect of frame-parallel event processing is that frames may complete their processing out-of-order with respect to their arrival order. However, in-order frame delivery must be ensured to avoid the performance degradation associated with out-of-order TCP packet delivery, such as duplicate acknowledgments (ACKs) and the fast retransmit algorithm. To facilitate this, the firmware maintains several status buffers where the intermediate results of network interface processing may be written. The firmware's work-discovery loop inspects the final stage results in order for a "done" status and commits all subsequent, consecutive completed frames. The task of committing a frame may not be run concurrently but committing a frame requires only a pointer update.

Adding NIDS

Recall the Snort processing loop depicted in Fig. 1. A NIC-embedded Snort does not require a separate stage to read packets, as Ethernet processing already will have brought them into memory, and Snort can be added as follow-on phases to the existing parallel firmware. However, the parallelization of these stages presents obstacles. In particular, the data structures used by stream reassembly must be shared by different packets, making packet-level parallelization impractical. However, there are no data dependencies between different TCP sessions, so a parallelization that ensures that each session would be handled by only one processor could access the different sessions without any need for synchronization. Whereas the Ethernet-processing portion of the firmware uses frame parallelism, the lack of dependencies among TCP sessions for Snort analysis motivates session-level parallelism for intrusion-detection stages of the firmware.

Figure 3 shows the stages of Snort operating using this parallelization strategy. The top and bottom rows represent separate threads of execution, with shared elements in the middle. The parallel software uses distributed task queues, with one per processor. After a packet has been committed for completion by the Ethernet-layer firmware processing, it is ready for intrusion-detection processing. However, the packet must first be placed into the queue corresponding to its flow. The source and destination IP addresses are looked up in a global hash table. If the stream has an entry, the queue listed in the table is used. Otherwise the stream is assigned to whichever queue is currently shortest, and the entry is added to the table, ensuring subsequent packets from the stream will go into the same queue. Hence, after assignment, the flow is entered into the firmware's distributed task queue, where it will be processed by the assigned processor's work-discovery loop. Stream reassembly is then performed for each session using the steps described earlier. The processor assigned to the queue places a pointer to the incoming packet data in its stream reassembly tree. Eventually, enough of the stream is gathered that Snort decides to flush it. This processor then finds a free stream reassembly buffer and copies the packet data from DRAM into the free buffer. These stream reassembly buffers are allocated in the scratchpad for fast access; then the buffers are passed to the hardware pattern matching assist by enqueueing a command descriptor.

The string content-matching assist dequeues the descriptor passed in from stream reassembly, fetches the reassembled stream data, scans it in hardware, and notes any observed rule matches. For each rule matched by the stream, it writes a descriptor into the global match queue, containing a pointer to the rule data. The next processor to dequeue from the match queue must verify each match reported by the content matcher. Each Snort rule specifies several conditions (that may include multiple strings), but the matcher checks only for the longest exact content string related to each rule. The verification stages check each condition for the rule and handle alerting, if necessary. The HTTP inspect and verification stages are essentially unmodified from the original Snort. The notification stage shares common alerting mechanisms; no



■ Figure 4. *Scaling core frequency and the number of processors for 10 Gb/s Ethernet.*

attempt was made to privatize this stage as it represents only 1.5 percent of the total computation even with complete serialization.

The firmware resides in the external instruction memory and requires approximately 415 kbytes, but its working set is small enough to allow a high hit rate in the instruction caches. The firmware uses the scratchpads for all statically-allocated data (e.g., Ethernet processing control data, interstage queues, and target buffers for stream reassembly and HTTP inspect). DRAM is used for frame contents and all dynamically-allocated data (e.g., the stream reassembly trees, the session assignment table, and the per-port rule tree data). This data allocation does not suffer as a result of the simple write-through cache in the system because very little data in DRAM is actually updated by the processors during operation.

Performance Evaluation

The proposed Ethernet controller architectures are evaluated using Spinach, a cycle-accurate toolkit for simulating programmable network interfaces [12]. Spinach allows various simulators to be composed from a library of modules that can be composed hierarchically at various levels of abstraction. Spinach features modules that are applicable to general-purpose programmable systems and modules that are specific to embedded systems and network interfaces, including host and network harness modules capable of playing back a tcpdump-formatted trace file. Spinach has been validated against the Tigon-II programmable multiprocessor NIC.

Ethernet Processing

The proposed architecture is first evaluated by examining raw performance when performing only Ethernet protocol processing. Performance is benchmarked by using streams of fixed-length user datagram protocol (UDP) packets. Figure 4 shows UDP throughput when processing bi-directional streams of maximum-sized UDP packets (1472 bytes) that lead to maximum-sized Ethernet frames (1518 bytes). The figure shows the achieved UDP throughput as the processor frequency and number of processors in the architecture are varied. All configurations use four scratchpad banks, an 8-kbyte two-way set associative instruction cache with 32-byte lines/processor, external SDRAM operating at 500 MHz, and a physical network link operating at 10 Gb/s for both transmit and receive traffic. The figure shows that the architectural and software mechanisms enable parallel cores to achieve the physical network maximum UDP bandwidth of almost 19.2 Gb/s.

At 175 MHz, six cores achieve 96.3 percent of line rate, and eight cores achieve 98.7 percent of line rate. At 200 MHz, both six and eight cores achieve within 1 percent of 10 Gb/s Ethernet bi-directional line rate. In contrast, simulation data (not shown in the figure) shows that a single core must operate at 800 MHz to achieve line rate [1]. Given the superlinear relationship between processor frequency and power dissipation, even eight 200-MHz processors would be far more power-efficient than a single 800-MHz processor.

When using six 200-MHz cores, the proposed memory and computation architecture is operating at peak performance as the system is able to provide full-duplex 10 Gb/s line rate. However, the two-dimensional scalability of the proposed architecture affords the flexibility to improve overall capability through either parallel processing or frequency scaling. This enables the programmer to implement advanced services, including intrusion detection, in exchange for a portion of the achievable Ethernet-processing bandwidth of the architecture.

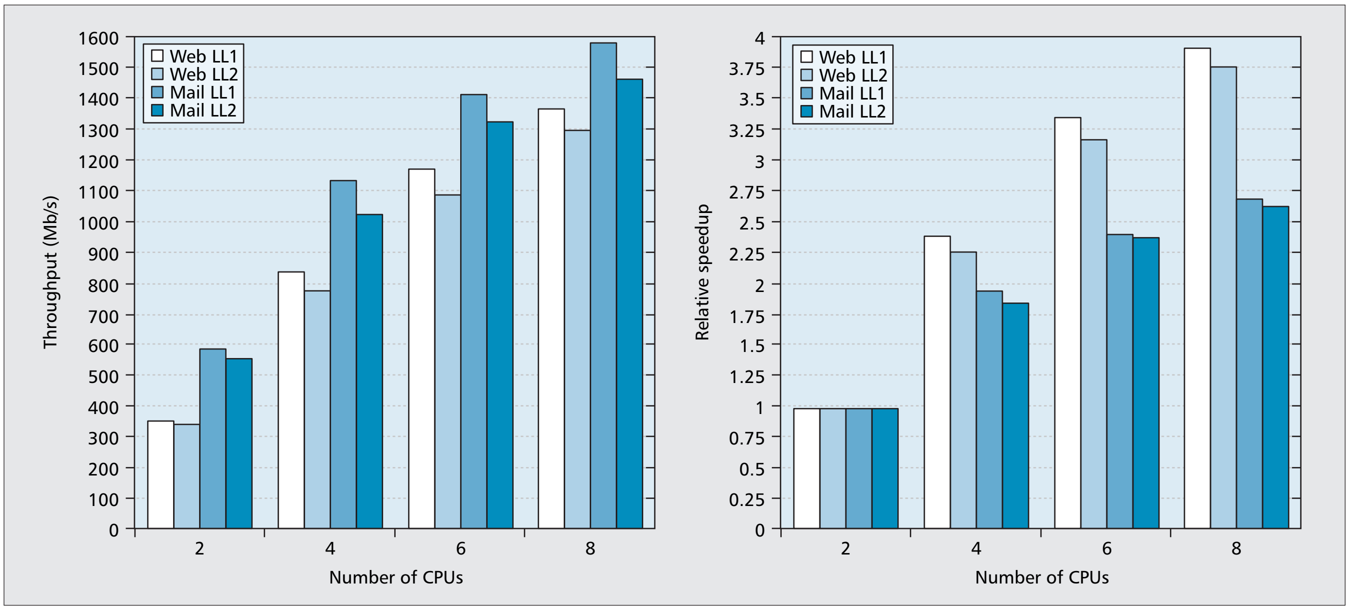
Intrusion Detection

Benchmarking for NIDS is more complicated than for protocol processing because NIDS is highly sensitive to the content of the packets being inspected. Thus, actual packet traces must be used. The traces used here come from the 1998-1999 Defense Advanced Research Projects Agency (DARPA) intrusion detection evaluation at the MIT Lincoln Lab, which simulates a large military network [8]. Because they were generated specifically for NIDS testing, the traces have a good collection of traffic and contain attacks that were known at the time. However, because they were designed for testing completeness rather than performance, the packet sizes and flow concurrency levels are not realistic. To address this problem, a variety of flows were taken from several DARPA traces and reassembled to more closely match average packet sizes (≈ 778 bytes) seen in publicly available header traces from the National Laboratory for Applied Network Research (NLANR) Passive Measurement and Analysis Web site (pma.nlanr.net). Two traces, called LL1 and LL2, were created in this manner, using different ordering and interleaving among the flows. The NIDS ruleset is taken from the official Snort ruleset released by its authors. Running an NIDS on an individual system enables the ruleset to be customized; in particular, rules that do not apply to the system (e.g., Windows NetBIOS rules on a Unix system) can be removed to increase performance and reduce false positives. Two different rulesets are tested, one containing rules that might apply to a mail server and another containing rules that might apply to a Web server. Both rulesets contain rules for common services such as ssh.

Figure 5a shows the base results for the LL traces using the Web and mail rulesets for varying numbers of CPU, running at 500 MHz. There are many factors, both in hardware and software, that affect performance. The following analyzes the impact of the ruleset used, the number of processors, and the clock frequency.

Rulesets

Rulesets have an important effect on any intrusion detection system (IDS), and the choice of rules always involves performance trade-offs. In Snort, the ruleset affects how much work the firmware must do in verifying content matches. Since the content matcher has only one string table, many matches are for rules that ultimately do not lead to alerts; these *false positives* will be filtered out by the verification stage. A false posi-



■ Figure 5. *Snort throughput results achieved with 2–8 CPUs for mail and Web rulesets with LL1 and LL2 input traces: a) inspection throughput; b) normalized speedup relative to 2 CPUs.*

tive can arise because the rule does not apply to the TCP/UDP ports in use or because it specifies additional conditions that are not met by the packet. If the ports do not match, the verification will complete quickly. However, if additional string or regular expression matching is required, verification will be much slower as the processor must read the packet and process the data using the matching algorithms. The mail ruleset actually generates more false positives at the string-matching hardware, because this ruleset contains many short strings that appear in many packets; however, the traces used contain far more Web flows than email flows, so verification for the mail ruleset typically completes quickly after a simple port mismatch. In contrast, many Web rules continue to the more expensive checks, causing the Web ruleset to spend nearly twice as much of its time on verification as the mail ruleset. Within the verification stage, the largest fraction of CPU time is consumed by regular expression matching.

Processor and Frequency Scaling

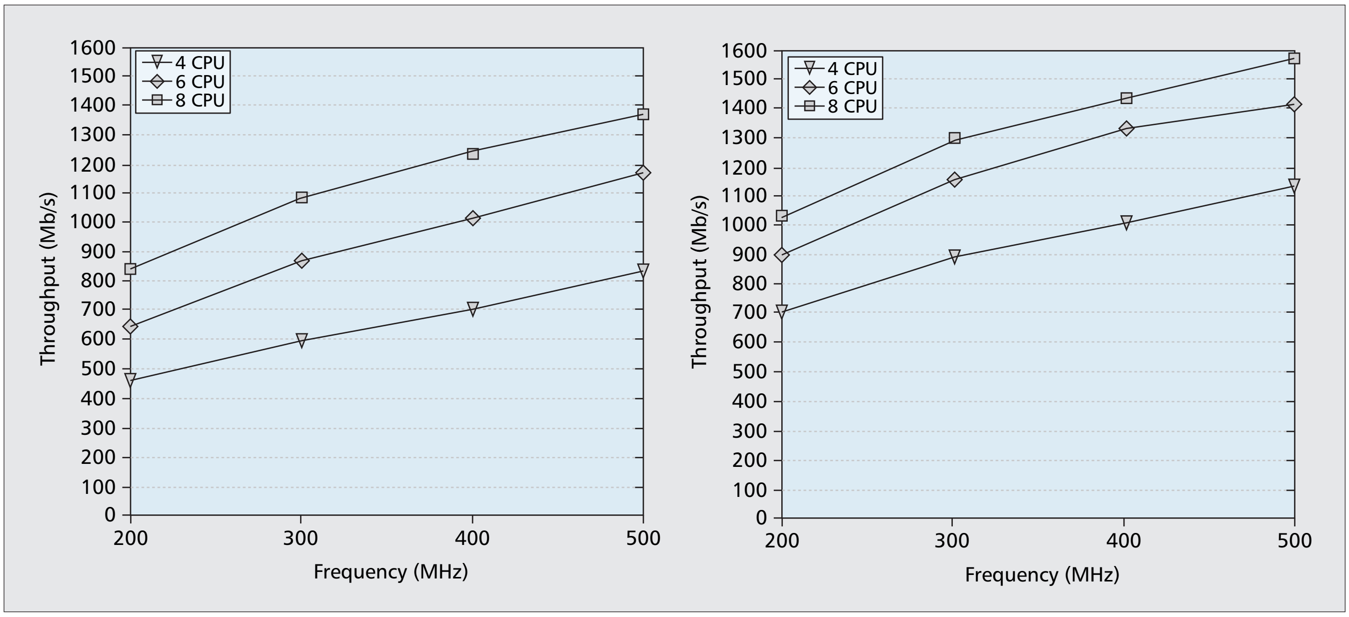
Figure 5b shows the speedup obtained by increasing the number of processors, normalized to the two-processor throughput, for each trace and ruleset. Although the mail ruleset has higher raw throughput in all cases, the Web ruleset scales much better with increasing numbers of processors. For the mail ruleset, two main factors contributed to the limitation in scalability. Because the mail ruleset requires a comparatively small amount of verification work per packet, its throughput is more limited by the rate at which the processors can perform the TCP stream reassembly and copy the reassembled data from the DRAM into the scratchpad for inspection. Thus, there is more contention for the shared cache with the mail ruleset even with four processors than there is with the Web ruleset with eight, limiting the scalability of the throughput with more processors. The mail ruleset also triggers a larger increase in lock contention as the number of processors increases. Conversely, the Web ruleset requires much more verification work per packet and as a result, has lower overall throughput. The verification process reads the reassembled packet data from scratchpad rather than DRAM so for the Web ruleset, the overall workload is more balanced between the scratchpad and the DRAM, making increased contention for the DRAM less important. Moreover, because the scratchpad is banked, the increase in processors causes less contention there than for the cache.

Figure 6 shows the relationships between processor frequency and throughput for the LL1 trace with the Web and mail rulesets. Scaling the processor frequency gives less than linear speedups because the DRAM latency and bandwidth is unchanged, so processors at higher frequencies spend more cycles waiting for DRAM accesses. The Web ruleset scales better with frequency for the same reason it scales better with additional processors—it is slower overall, and DRAM access makes up a smaller fraction of its time. These results indicate several paths to achieve near-gigabit speeds; for simpler rulesets, such as the mail ruleset, four processors can be used at 400 MHz, six at between 200 MHz and 300 MHz, or eight at 200 MHz, depending on costs and power budgets. Likewise, a workload such as the more demanding Web ruleset would require six processors at 400 MHz or eight at between 200 MHz and 300 MHz. Scaling the architecture beyond eight processors would probably require bandwidth improvements in the cache. Alternatively, adding further hardware assistance for stream reassembly (such as adding a gather capability to the string matcher) could substantially reduce the overall workload by eliminating copy overhead for the reassembly but would make the overall system less flexible.

Additional results, including the effect of flow assignment and reassignment, hardware-assisted string matching, and caches are discussed elsewhere [2]. To summarize them, flow reassignment provides benefits of up to 16 percent, with an average of 7 percent across all traces and configurations tested. Hardware-assisted string matching was found to be essential to approach gigabit speeds, as was at least a small amount of cache. However the DRAM footprint and amount of reuse are small enough that increasing cache size beyond 4 kbytes provides only limited benefit. In addition, a shared cache configuration performed better than private caches as coherence-related flushes were not required for interprocessor sharing.

Conclusions

This article proposes and evaluates a design for an efficient programmable Ethernet controller based on parallel processing cores, an explicitly-partitioned memory system, and firmware that exploits concurrency in the network data stream. This architecture is shown to scale to 10 Gb/s full-duplex rates using six simple pipelined cores at 200 MHz. This



■ Figure 6. *Impact of CPU frequency for LL1 trace: a) Web ruleset; b) Mail ruleset.*

architecture provides a solid base for implementing extended services, such as network intrusion detection. This network interface architecture can directly execute the Snort network intrusion detection system at gigabit Ethernet network throughputs by extending the NIC to eight cores operating at 300 MHz and adding a hardware string-matching assist. Such a network interface can support all standard Snort rule features, reassemble TCP streams, and transform HTTP URL. In contrast, achieving similar performance on the host would require multiple 2 GHz general-purpose processors. Using a programmable NIC enhanced with Snort, a single host can effectively be protected from both LAN-based and Internet-based attacks, unlike edge-based NIDS which only guards against the latter.

Acknowledgments

This work is supported in part by the National Science Foundation (NSF) under Grant Nos. CCR-0209174, CCF-0532448, and CNS-0532452 and by donations from Advanced Micro Devices (AMD).

References

[1] P. Willmann *et al.*, "An Efficient Programmable 10 Gigabit Ethernet Network Interface Card," *Proc. 11th Int'l. Symp. High-Performance Comp. Architecture*, Feb. 2005, pp. 96–107.
[2] D. L. Schuff and V. S. Pai, "Design Alternatives for a High-Performance Self-Securing Ethernet Network Interface," *Proc. 21st IEEE Int'l. Parallel and Distrib. Processing Symp.*, Mar. 2007.
[3] H.-Y. Kim and S. Rixner, "TCP Offload through Connection Handoff," *Proc. EuroSys Conf.*, Apr. 2006, pp. 279–90.
[4] K. Z. Meth and J. Satran, "Design of the iSCSI Protocol," *Proc. Conf. Mass Storage Sys. and Technologies*, Apr. 2003, pp. 116–22.
[5] H.-Y. Kim, V. S. Pai, and S. Rixner, "Improving Web Server Throughput with Network Interface Data Caching," *Proc. 10th Int'l. Conf. Architectural Support for Programming Languages and Op. Sys.*, Oct. 2002, pp. 239–50.
[6] M. Roesch, "Snort — Lightweight Intrusion Detection for Networks," *Proc. 13th USENIX Conf. Sys. Admin.*, 1999, pp. 229–38.
[7] A. V. Aho and M. J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Commun. ACM*, vol. 18, no. 6, 1975, pp. 333–40.
[8] J. W. Haines *et al.*, "1999 DARPA Intrusion Detection System Evaluation: Design and Procedures," MIT Lincoln Lab., tech. rep. 1062, 2001.
[9] L. Tan and T. Sherwood, "A High Throughput String Matching Architecture for Intrusion Detection and Prevention," *Proc. 32nd Annual Int'l. Symp. Comp. Architecture*, June 2005, pp. 112–22.
[10] F. Yu, R. H. Katz, and T. V. Lakshman, "Gigabit Rate Packet Pattern-Matching Using TCAM," *Proc. 12th IEEE Int'l. Conf. Network Protocols*, Oct. 2004, pp. 174–83.
[11] J. Moscola *et al.*, "Implementation of a Content-Scanning Module for an Internet Firewall," *Proc. 11th Annual IEEE Symp. Field-Programmable Custom Computing Machines*, Apr. 2003, pp. 31–38.
[12] P. Willmann, M. Brogioli, and V. S. Pai, "Spinach: A Liberty-based Simulator for Programmable Network Interface Architectures," *Proc. ACM SIGPLAN/SIGBED 2004 Conf. Languages, Compilers, and Tools for Embedded Syst.*, ACM Press, July 2004, pp. 20–29.

Biographies

DEREK L. SCHUFF (dschuff@purdue.edu) received an M.S. in electrical and computer engineering from Purdue University and is a Ph.D. candidate in electrical and computer engineering at Purdue. His research interests include multiprocessor computer architecture, parallel programming, and high-performance intrusion detection.

VIJAY S. PAI received a Ph.D. in electrical engineering from Rice University. He is an assistant professor of electrical and computer engineering at Purdue University. His research interests include parallel programming tools and technologies, multiprocessor computer architecture, network server software, and performance evaluation.

PAUL WILLMANN received an M.S. in electrical and computer engineering from Rice University. He is a Ph.D. candidate in electrical and computer engineering at Rice. His research interests include operating systems, virtual machine monitors, programmable embedded systems, and software/hardware interfaces for efficient I/O.

SCOTT RIXNER received a Ph.D. in electrical engineering from the Massachusetts Institute of Technology. He is an associate professor of computer science and electrical and computer engineering at Rice University. His research interests include media, network, and communications processing; memory systems architecture; and the interaction between operating systems and computer architectures.