HIGH-PERFORMANCE NETWORK INTRUSION DETECTION THROUGH PARALLELISM

A Thesis

Submitted to the Faculty

of

Purdue University

by

Derek L. Schuff

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical and Computer Engineering

May 2007

Purdue University

West Lafayette, Indiana

ACKNOWLEDGMENTS

Special thanks are due to my advisor Vijay Pai for guidance, direction, and assistance throughout the process, for continued support in the form of a research assistantship, and for a real commitment to my technical and professional growth. Thanks to Yung Ryn Choe for his joint work on MultiSnort, and to the rest of our research group, Patrick Clough, Jagadeesh Dyaberi, Gautam Upadhyaya, and Stephen Ziemba for their feedback particularly on my conference talk. I look forward to continuing my work with all of you. Thanks also to Sam Midkiff and T. N. Vijaykumar for their valuable feedback on my conference talk, and to my Advisory Committee members William Chappell, Rudolf Eigenmann, and Mithuna Thottethodi for taking the time to consider this work.

Finally, I cannot sufficiently express my gratitude to my family; to my parents Rick and Mary Schuff, whose love, support, guidance, and encouragement through the years have thoroughly equipped me for life, and my wife Mary, who makes it all worthwhile.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Schuff, Derek L. M.S.E.C.E, Purdue University, May, 2007. High-Performance Network Intrusion Detection through Parallelism. Major Professor: Vijay S. Pai.

Network intrusion detection has become increasingly popular in recent years due to the proliferation of Internet-based security attacks. Network intrusion detection systems (NIDS) inspect the content of incoming packets on a network for known attacks and alert the operator when they are found. Intrusion detection is computationally expensive, and this expense limits the inspection throughput for current implementations to substantially less than the Gigabit line rate for modern CPUs. This thesis presents methods for increasing NIDS performance though parallel execution using the open-source Snort NIDS, both on commodity multiprocessor systems and as part of a hardware-accelerated programmable Ethernet network interface. For both types of systems, inspection is parallelized at the level of TCP or UDP flows, as any potential inter-packet dependencies are confined to a single flow. Flows are assigned to processors or threads for all stages of inspection. For commodity multiprocessor systems, this strategy is compared to more optimistic schemes that allow reassignment of flows between threads, and evaluated using several network packet traces.

A strategy for integrating the Snort NIDS into a self-securing Ethernet network interface is also presented, with multiprocessor hardware designs utilizing hardware acceleration for part of the inspection process. The firmware executes a parallel version of Snort, and several hardware and software design alternatives are tested using a simulator.

1. INTRODUCTION

1.1 Network Intrusion Detection

Internet-based security attacks have proliferated in recent years, with buffer overruns, cross-site scripting, and denial-of-service among the most prominent and damaging forms of attack (commonly called *exploits*). A successful attacker can then initiate follow-on exploits, such as compromising the local system to gain administrator-level access or initiating denial of service attacks against other machines. The most popular operating systems regularly publish security updates, but the combination of poorly-administered machines, uninformed users, a vast number of targets, and ever-present software bugs has allowed exploits to remain ahead of patches.

Edge-based firewalls form a traditional approach to network security, based on the assumption that malicious attacks are sent to a target system across the global Internet. However, firewalls do not protect applications running on externally exposed ports, nor do they protect against attacks originating inside the local-area network. Such attacks may arise once a machine inside the network has been compromised by an otherwise undetected attack, virus, or user error (such as opening an email attachment). LAN-based attacks are potentially even more dangerous than external ones as they may propagate at LAN speeds and attack internal services such as NFS.

Network intrusion detection systems (NIDSes) run on a server at the edge of a LAN to identify and log Internet-based attacks against a local network. Unlike firewalls, which work by shutting off external access to certain ports, NIDSes can monitor attacks on externally-exposed ports used for running network services. The most popular NIDS is the open-source Snort, which identifies intrusion attempts by comparing every inbound and outbound packet against a *ruleset* [1]. Rules in the set represent characteristics of known attacks, such as the protocol type, port number, packet size, packet content (both strings and regular expressions), and the position of the suspicious content. Each new type of attack leads to new rules, with rulesets growing rapidly. The most recently-released freely-available Snort rulesets have over 4000 rules.

The processing required by a network intrusion detection system such as Snort is quite high, since the system must decode the data, reassemble network packets into streams, preprocess the data, scan the streams for matches against the specified string content and regular expressions, and log intrusions. These requirements limit Snort to an average packet processing rate of about 699 Mb/s on a modern host machine (1.8 GHz Xeon processor) — well below link-level bandwidths that are already commoditized at 1 Gigabit and are now approaching 10 Gigabits. Consequently, it is not possible to deploy Snort directly at a high-end network access point that requires a data rate of 1 Gb/s or more. NIDSes such as Snort may also run on individual host machines on a LAN. for lightly-loaded desktops or other machines which do not use a large amount of network bandwidth this may be an acceptable option; however, for a server which must serve data at high rates it may still be insufficient. Furthermore, the CPU power on a running server machine would also need to be shared with the server applications and the operating system. Consequently, it is not feasible to deploy Snort directly on such machines.

To address the problem at the network edge, various companies and researchers have introduced high-end single-purpose appliances or proposed solutions based on clustering [2–4]. Clustered NIDS potentially allows high scalability, but requires the use of an expensive load-balancing switch. To allow for the reliable detection and logging of attacks within the LAN, Ganger et al. propose self-securing network interfaces, by which the network interface card executes the NIDS on data as it streams into and out of the host [5]. This mechanism would allow the interface to see all traffic seen by the host, and would also prevent an attack from trivially disabling the NIDS by using root access to kill the NIDS process. Their prototype used a PC placed between the switch and the target PC as the self-securing network interface.

The newly inserted PC ran a limited operating system and several custom-written applications that scanned traffic looking for suspicious behaviors. Although valuable as a proof-of-concept, a more usable self-securing network interface would need to be implemented as an Ethernet network interface card (NIC) and execute a more standard NIDS. This thesis has contributions to address performance issues both at the network edge and within the LAN.

1.2 MultiSnort: High-Performance Parallel Network Intrusion Detection for Commodity Multi-Processor Systems

Chapter 3 presents and evaluates methods to parallelize Snort for PC-based systems with modern multiprocessor architectures. This portion of the thesis draws from joint work with Yung Ryn Choe. Although an NIDS like Snort receives its input on a packet-by-packet basis, it must seek to aggregate distinct packets into TCP streams to prevent an attacker from disguising malicious communications by breaking the data up across several packets. Additionally, an NIDS must process later packets in a given communication based on decisions made when analyzing earlier packets. For example, if a given sequence of characters represents a possible attack in the body of an HTML document but may appear normally in an image, the NIDS should not trigger an alert on that attack if an earlier packet indicated that this data transfer was an image. Such constraints are incorporated into Snort by *stream reassembly* and *flowbits*, respectively. All TCP data is reassembled into streams, and about 36% of rules require flow tracking (90% of which are related to NetBIOS). Although these phases require packets to be processed in-order, a key observation is that any ordering or data sharing between the processing of separate packets only applies to packets in the same IP flow (which include not only TCP streams but also source/destination communication pairs in other protocols). Although this thesis specifically targets the Snort NIDS, the parallelization challenges and strategies discussed here apply to any intrusion detection system that uses TCP stream reassembly to merge packets together for inspection or preserves other state across different packets from the same flow.

The parallelization strategies studied in this chapter take different approaches to splitting the Snort NIDS into threads: two conservative and one optimistic. The primary conservative scheme, called the *flow-concurrent* parallelization, exploits concurrency by parallelizing ruleset processing on a flow-by-flow basis. All packets are initially received by one thread. That thread inspects the IP headers to determine the flow to which the packet belongs and then steers that packet to the appropriate processing thread based on whether or not that flow has already been assigned to a thread. Since each given flow is only processed by one thread at any given time, the dependences required for proper stream reassembly and flow tracking are maintained easily. This scheme works well if there are enough independent flows, but provides no benefits if all packets are from the same flow. The latter case is not a likely situation in a high-bandwidth edge NIDS, but does represent a limitation of this scheme. A variation on this scheme aims to improve load balance by allowing a flow to be dynamically reassigned from one thread to another, but only if there are currently no packets from that flow still waiting to be processed. This reassignment is still conservative because it preserves the invariant that a specific flow is only processed by one thread at any given time.

The alternative parallelization is an optimistic variant on flow concurrency. This scheme starts with the basic flow-concurrent parallelization but then has the ability to dynamically reassign a flow to a different thread even while earlier packets of the flow are still being processed, potentially exploiting parallelism even with just one flow. This optimistic version relies on two key observations. First, TCP stream reassembly will still take place even if a stream is broken at some arbitrary point; reassembly is triggered by various flush conditions, one of which is a timeout. It is also easy to force additional flushes if needed for correctness. Consequently, any unprocessed earlier packets will still go through stream reassembly at their thread even though later packets are being reassembled and processed in another thread. (This property also allows the conservative reassignment described above). Second, most packets do not match rules that use flowbits tracking, so enforcing ordering across all packets in a flow just to deal with a few problematic rules is too restrictive. To precisely deal with the rules that do use flowbits, the optimistic system stalls processing in any packet that sets or checks flowbits unless it is the oldest packet in its flow. This condition is checked by adding per-flow reorder buffers. This system is optimistic in the sense that it reassigns threads under the assumption that the actual use of flowbits is uncommon, but is still conservative in maintaining correct ruleset processing without requiring rollbacks and redundant processing.

All the parallelizations use most of the same packet processing code as the current Snort (version 2.6), with minor modifications to make certain code segments re-entrant and well-synchronized using Pthreads. The resulting NIDS tools are evaluated on two different systems: a 2U rack-mounted x86-64 Linux system with two quad-core Xeon processors (eight processor cores in total) and a similar system with two dual-core Opteron processors. Results for both systems are evaluated and the differences discussed. The flow–concurrent parallelization achieves substantial speedups on 3 of the 5 network packet traces studied, ranging as high as 4.1 speedup on 8 processor cores and processing at speeds up to 3.1 Gb/s. The extra overheads introduced by allowing reassignment actually degrade performance by about 4.5% for the traces that exhibit good flow concurrency, and additional overheads in the optimistic parallelization degrade performance by an additional 16%, limiting speedup to 3.0 on six cores. However, the potential for intra-flow parallelism enabled by the optimistic approach allows one additional trace to see good speedup (2.4 on five cores), with a peak traffic rate over 2.6 Gb/s. All schemes achieve an average traffic rate of over 1.7 Gb/s for the 5 traces, providing several options for substantially increasing the performance of the serial version with only small increases in overall hardware cost and little or no increase in space. Parallelization allows the benefits of high-performance intrusion-detection without relying either on higher clock frequencies (which are reaching a stage of diminishing returns) or costly and space-consuming load balancers.

1.3 LineSnort: A Self-Securing Ethernet Network Interface

Chapter 4 has two key contributions [6]. First, it presents a strategy for integrating the Snort network intrusion detection system into a high-performance programmable Ethernet network interface. The architecture used in this chapter draws from previous work in programmable Ethernet controllers, combining multiple low-frequency pipelined RISC processors, nonprogrammable hardware assists for high-bandwidth memory movement, and an explicitly-managed partitioned memory system [7,8]. The firmware of the resulting Ethernet controller parallelizes Snort at the granularity of TCP sessions, and also exploits opportunities for intra-session concurrency. Previous academic and industrial work on special-purpose architectures for high-speed NIDS have generally only dealt with content matching [9–14]. General Snort-style intrusion detection allows stronger security than content-matching alone by reassembling TCP streams to detect attacks that span multiple packets, by transforming HTTP URLs to canonical formats, and by supporting other types of Snort tests.

The chapter's second contribution is to explore and analyze the performance impact of various hardware and software design alternatives for the resulting self-securing network interface, which is called LineSnort. The paper considers the impact of the Snort rulesets used and scalability with regard to processors and frequency, policies for assigning flows to processors, the benefits of hardware-assisted (as opposed to pure software) string content matching, and the importance of data caches. The results show that throughputs in excess of 1 Gigabit per second can be achieved for fairly large rule sets using hardware-assisted string matching and a small shared data cache. LineSnort can extract performance through increases in processor clock frequency or parallelism, allowing an additional choice for designers to achieve performance within specified area or power budgets.

2. BACKGROUND

2.1 Network Intrusion Detection

Snort is the most popular intrusion-detection system available. The system and its intrusion-detection ruleset are freely available, and both are regularly updated to account for the latest threats [1]. 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 multi-string pattern matching algorithms, including the well-known Aho-Corasick algorithm [15] a modified version of the Wu-Manber algorithm [16], and several others [17,18], which can be selected by the user. Additionally, Snort includes preprocessors that perform certain operations on the data stream. Some important preprocessors include **flow**, **stream4**, and **HTTP Inspect**. The flow preprocessor associates each scanned packet to a specific network traffic flow between a source and destination pair and allows rules to set, clear and check flags (called *flowbits*) associated with the flow based on packet contents. For example, one rule checks for a GIF image header and sets a specific flowbit, and another checks for a heap overflow exploit that may occur in a later packet in flows which have that bit set. The stream4 preprocessor tracks TCP connection states and allows rules to take them into consideration for rule matches (for example, only match the rule if the packet is part of an established TCP connection). Stream4 also performs stream reassembly, taking multiple scanned packets from a given direction of a stream and builds a single conceptual packet by concatenating their payloads, allowing rules to match packet content that spans packet boundaries. This is one of the most important preprocessors because without it, an attacker can trivially hide attacks by simply splitting them across more than one packet. Packets are reassembled into stream buffers and sent into the inspection process after a "flush point" is reached. Flush points are triggered by conditions such as processing a certain randomly-selected amount of data or a timeout. The HTTP Inspect preprocessor converts URLs to a normalized canonical form so that rules can specifically match URLs rather than merely strings or regular expressions. For example, it decodes URLs containing percentage escape symbols (e.g., "`%7e`" instead of the tilde symbol) to their normal forms [19]. It also generates absolute directories instead of the "`../`" forms sometimes used to hide directory traversals. After this decoding, the same pattern matching algorithm that Snort uses for packet data is used on the normalized URL as specified by "URI-content" rules.

The Snort ruleset has been regularly updated over the past 5 years, quadrupling in size from approximately 1000 rules in 2001 to over 4000 in March of 2006, and indicating a roughly constant rate of increase of new attack signatures over that period of time. Although the performance of the multi-string content matching does not depend directly on the number of rules, a greater number of rules does require a greater amount of memory and may require more time to be spent in the stages of intrusion detection other than multi-string matching. Over 86% of the rules are for TCP-based exploits (including over 30% for HTTP rules), with UDP at about 9%, ICMP at 4%, and other IP rules at just over 1%. Nearly all of the higher-level protocol rules (HTTP, TCP, and UDP) check for string matching content, but a large fraction of the ICMP and general IP rules do not. In addition to exact string and regular expression matching, the Snort ruleset language includes 13 tests based on packet payload and 20 based on headers. Most rules specify several conditions on packet content, and all of them must be met for a match to occur. The exact string match (implemented by the multi-pattern matching algorithm) is used as a first-order filter; tests for a specific rule are not performed unless its corresponding pattern is matched first. This is particularly important for rules that specify time-consuming

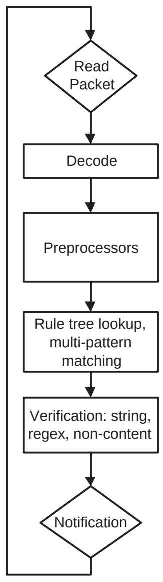

Read Packet
Decode
Preprocessors
Rule tree lookup, multi-pattern matching
Verification: string, regex, non-content
Notification

Fig. 2.1. The Snort packet processing loop

regular expressions. About 10% of HTTP rules, 40% of all TCP rules, and 30% of UDP rules test for regular expression matches.

The following rules demonstrate some of the kinds of traffic characteristics used by Snort to detect attacks:

- SMTP Content-Type buffer overflow: TCP traffic to an SMTP server, established connection to port 25, string “Content-Type:”, regular expression “Content-Type:[ˆ\r\n]300,” (i.e., 300 or more characters after the colon besides carriage return or newline)
- PHP Wiki Cross-site Scripting: TCP traffic to HTTP server set, established connection to HTTP port set, URI contains string “/modules.php?”, URI contains string “name=Wiki”, URI contains string “<script”
- DDOS Trin00 Attacker to Master: TCP traffic to home network, established connection to port 27665, string “betaalmostdone”

Figure 2.1 depicts the packet processing loop used by Snort. Snort first reads a packet from the operating system using the `pcap` library (also used by `tcpdump`

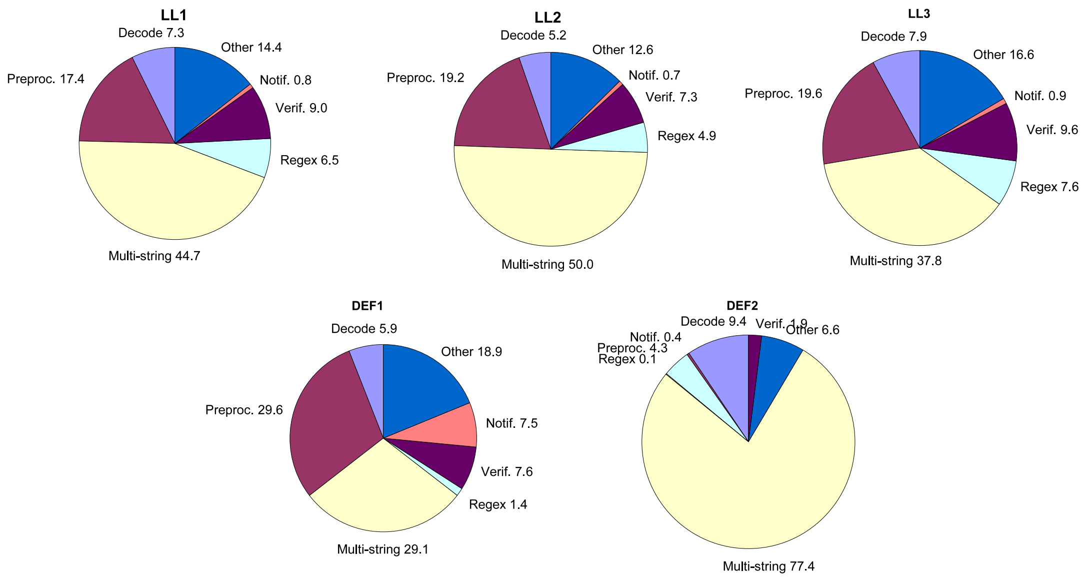


Fig. 2.2. Execution time of Snort categorized into principal components when run using various traces on an Intel Xeon-based system

and other analysis tools). The decode stage interprets the packet's tightly-encoded protocol headers and associated information, storing the results in Snort's loosely-encoded packet data structure. Snort then invokes the preprocessors that use and manipulate packet data in various ways. The rule-tree lookup and pattern matching stage determines which rules are relevant for the packet at hand (based on port number) and checks the packet content for the attack signatures defined in the string rules using the multi-string matching algorithms. Packets may match one or more strings in the multi-string match stage, each of which is associated with a different rule. For each of those rules, all the remaining conditions are checked, including other strings, non-content conditions, and regular expressions. Because each match from the multi-pattern algorithm may or may not result in a match for its rule as a whole, this stage may be thought of as a "verification" stage. This stage also calls detection functions for any rules without exact string matches. The last stage notifies the system owner through alerts related to the specific rule matches.

Figure 2.2 categorizes the execution time of a system running Snort into various components for five different network test patterns. Most of these components correspond to the stages shown in Figure 2.1. The component labeled **Other** includes utility code and library functions shared among several components, the overall packet processing loop and other code between the stages, operating system activity, and other processes running on the system (such as the profiler itself). Most of the time spent in this category consists of shared library calls (`malloc`, `memset`, etc.) and code that calls the other stages and transitions between them. The remaining part is very small, and its effect is not considered further. The Snort code tested here is modified to read all of its packets sequentially from an in-memory buffer to allow the playback of a large network trace representing communication from various hosts to a local network. The network traces used in these tests and their significance are described in more detail in Section 3.2.

The profiles shown here were gathered using the `oprofile` full-system profiling utility running on a Dell Poweredge 2950 with two quad-core Xeons (8 processors total), but Snort only runs on 1 processor. The system has 16 GB of DRAM and uses Linux version 2.6.18. The profile was gathered using the `oprofile` full-system profiling utility, and the Snort configuration included the most important preprocessors: flow, stream4, and HTTP Inspect as described above. The overall performance of this system averages 699 Mb/s for these traces with a peak of 1094 Mb/s.

As Figure 2.2 shows, string content matching is a very important component of intrusion detection, ranging from 29–77% of the execution time of the system with an average of 48%. Similar observations by others have led some researchers to propose custom hardware support for this stage, based on ASICs [13, 20], TCAMs [14], or FPGAs [9, 11, 12]. Some have also proposed support for regular expressions [10, 11]. Such hardware engines increase performance both by eliminating instruction processing overheads and by exploiting concurrency in the computation. Although some of these hardware systems have been tested using strings extracted from Snort rules, all have used unified state machines for all string matching rather than per-port state machines as in the Snort software.

Although Figure 2.1 shows the importance of string content matching, it also shows that no single component of intrusion detection makes up the majority of execution time. This is not unexpected since the rules invariably include multiple types of tests, not just string content matching. For all traces except DEF1, the combination of ruleset processing components (string match, verification, and regular expression) make up over 54% of execution time. Thus, any performance optimization strategy must effectively target those components. At the same time, the other components cannot be ignored since they make up an average of 41% of execution time.

2.2 Programmable Ethernet Controllers

The first widely-used Gigabit Ethernet network interface cards were based on programmable Ethernet controllers such as the Alteon Tigon [7]. Although more recent Gigabit Ethernet controllers have generally abandoned programmability, programmable controllers are again arising for 10 Gigabit Ethernet and for extended services such as TCP/IP offloading, iSCSI, message passing, or network interface data caching [21–25]. Although integrating processing and memory on a network interface card may add additional cost to the NIC, this cost will still likely be a small portion of the overall cost of a high-end PC-based network server. Consequently, such costs are reasonable if programmability can be used to effectively offload, streamline, or secure important portions of the data transfer.

Network interface cards must generally complete a series of steps to send and receive Ethernet frames, which may involve multiple communications with the host using programmed I/O, DMA, and interrupts. Programmable network interfaces implement these steps with a combination of programmable processors and nonprogrammable hardware assists. The assists are used for high-speed data transfers between the NIC and the host or Ethernet, as efficient hardware mechanisms can be used to sequentially transfer data between the NIC local memory and the system I/O bus or network. Willmann et al. studied the hardware and software requirements of a programmable 10 Gigabit Ethernet controller [8]. They found that Ethernet processing firmware does not have sufficient instruction-level parallelism or data reuse for efficient use of multiple-issue, out-of-order processors with large caches; however, a combination of frame-level and task-level concurrency allows the use of parallel low-frequency RISC cores. Additionally, the specific data access characteristics of the network interface firmware allow the use of a software-managed, explicitly-partitioned memory system that includes on-chip SRAM scratchpads for low-latency access to firmware metadata and off-chip graphics DDR (GDDR) DRAM for high-bandwidth access to high-capacity frame contents.

3. CONSERVATIVE VS. OPTIMISTIC PARALLELIZATION OF NETWORK INTRUSION DETECTION

3.1 Parallelization Alternatives

Parallelizing any application requires first identifying the available concurrency. As discussed in Section 2.1, the main loop of the Snort NIDS works on one packet at a time. Consequently, packet-level concurrency seems a natural granularity for parallelizing this application. However, several problems need to be addressed for parallelism to be feasible.

3.1.1 Packet-level Parallelization

Figure 3.1 illustrates the interactions between the processing of two separate packets in the Snort NIDS, following the basic Snort processing loop depicted in Figure 2.1. The center column of Figure 3.1 depicts resources shared by the processing of multiple packets, with dashed lines indicating the accesses to these resources by specific stages in the Snort processing loop. Although this work focuses on Snort, the same basic steps and resources are present in any intrusion detection system that reassembles packets from the same stream and tracks flow-specific state.

For packet-level parallelization to be practical, no resource should actually share common information across separate packets. However, the data structures used by the flow and stream reassembly preprocessors must be shared by different packets. In particular, the flow tracking table must be consulted while processing all packets and must be updated any time a packet arrives from a flow that is not currently being tracked. The flowbits are tested and set in the verification stage. Similarly, each

TCP packet's stream must have its state checked and set (loosely following TCP's state transition table [26], but with provisions for handling missing packets and picking up streams in mid-session) in the stream reassembly preprocessor to decide if it even needs assembly — streams that have not yet completed connection establishment should only be processed packet-by-packet. The reassembly list must be searched and updated on each packet, and the TCP stream state must again be checked in verification to decide if a rule applies. Processing packets completely independently could allow packets to arrive out-of-order at stream reassembly, disrupting the tracking of TCP connection states and possibly causing the system to miss an attack. However, maintaining proper ordering on shared data structures would require multiple expensive synchronization operations on every packet.

The notification stage must also share a common notification interface whenever an alert actually arises. However, the specific ordering of alert reports is not important, so simple mutual exclusion will suffice to enable the parallelization.

Other minor changes are also required for parallelization. For example, stages such as multi-pattern matching currently assume only one packet at a time and thus keep only one common structure for all processing; these structures must now be associated with a specific thread or packet to make the code reentrant.

3.1.2 Flow-level Parallelization

Although packet-level parallelization is impractical because of ordering requirements on shared data structures, any actual information sharing only applies to packets in the same flow. Since packets from one flow will never affect the flow-bits or TCP connection state of another flow, different flows can be processed by independent processing threads with no constraints on ordering. Moreover, the data sharing requirement of the stream preprocessor can be eliminated by simply maintaining separate stream tables for each thread. Then each thread can be responsible for different flows, as long as packets from the same flow are always steered to the

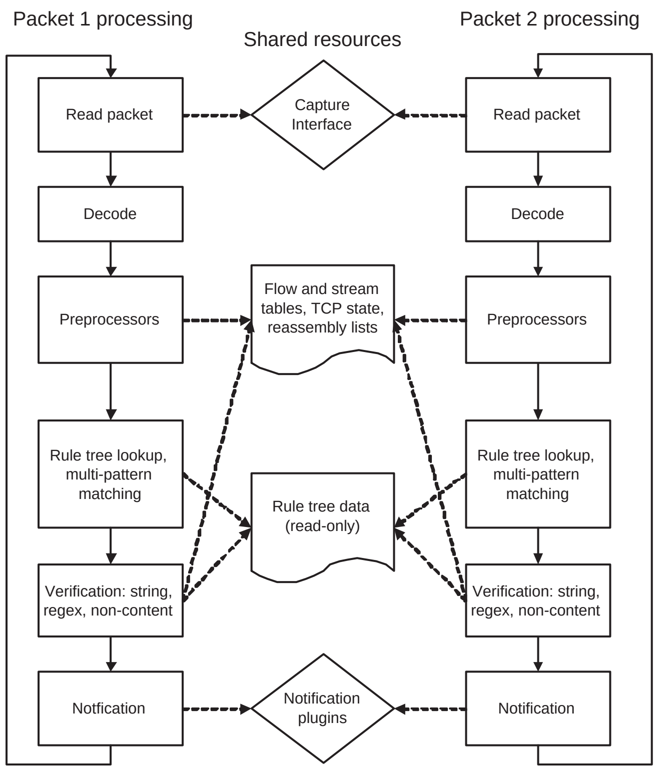

Packet 1 processing
Shared resources
Packet 2 processing
Read packet
Capture Interface
Read packet
Decode
Decode
Preprocessors
Flow and stream tables, TCP state, reassembly lists
Preprocessors
Rule tree lookup, multi-pattern matching
Rule tree data (read-only)
Rule tree lookup, multi-pattern matching
Verification: string, regex, non-content
Verification: string, regex, non-content
Notfication
Notification plugins
Notification

Fig. 3.1. Multiple instances of the Snort packet processing loop, with access to shared resources explicitly identified

same processing thread in-order. This steering process should consist of a minimal amount of code to determine the flow associated with any given packet and then enqueue the packet for processing by the appropriate thread.

In-order processing by each thread guarantees that the packets from each flow will be processed in the same relative sequence as in the serial code. Consequently, this flow-based concurrency model maintains all dependence constraints between packets in a flow. Note that this is conservative because a dependence might not actually exist between successive packets in a flow. For example, if the stream reassembly process encounters a flush point between those packets, they will be reassembled into separate stream buffers. Similarly, if a packet only matches rules that do not use flowbits, it is irrelevant whether earlier packets have set any flowbits. However, these conditions cannot be detected until the system is already deep into packet inspection. Consequently, this parallelization chooses to err on the side of caution by enforcing an ordering between packets in the same flow.

Note that the effectiveness of flow-level parallelization depends on the existence of multiple concurrent flows in the network stream. If there is only one flow on the network, then it can only be inspected by one thread. However, because NIDS sensors usually protect many machines or an entire network, it seems reasonable to assume that many flows will be present at a time, particularly if a large amount of bandwidth is being consumed.

Figure 3.2 shows the stages of the flow-based concurrency model. This model consists of two components: the producer routine and the consumer routine. The producer reads the packet from the interface and assigns the packet to its thread based on its flow. The consumer routine processes the remaining stages of the NIDS just as in the single-threaded Snort. Each thread has its own work queue and consumes packets from it as long as there are packets waiting. If its own queue becomes empty, it then becomes the producer, and begins reading packets and assigning them to their proper work queues. These work queues can be quite large since each entry only includes 3 pointers; consequently, it is unlikely that a work queue will fill up and

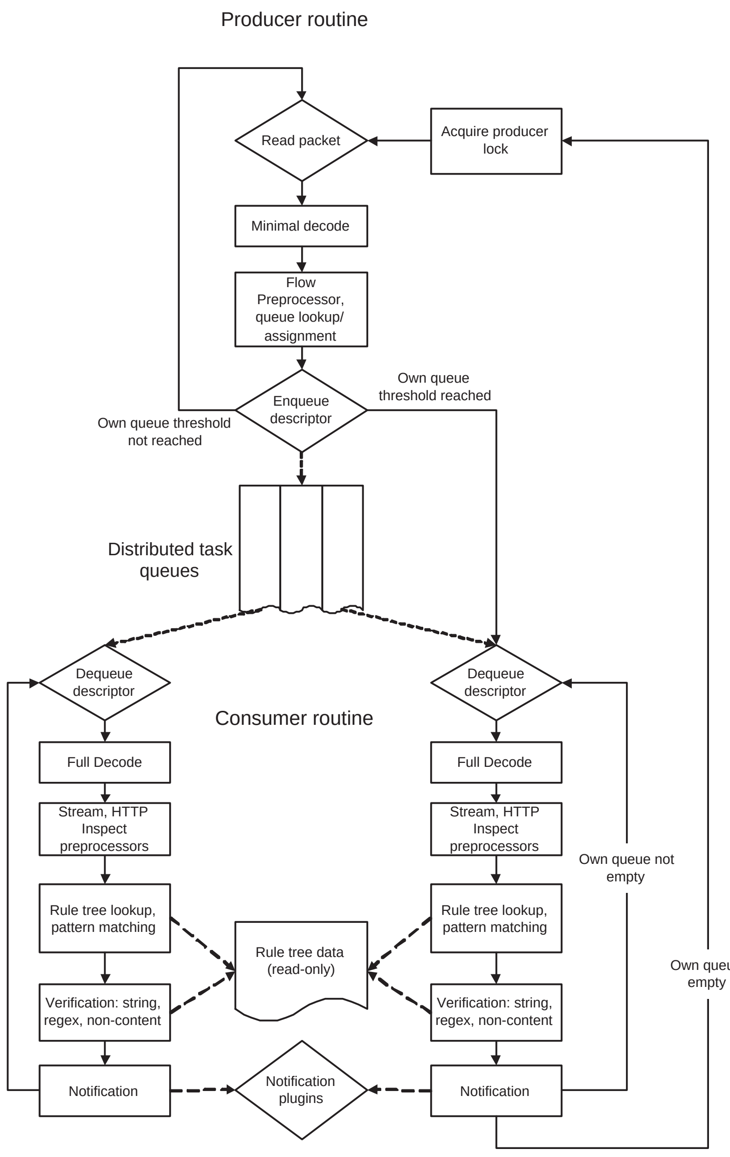

Producer routine
Read packet
Acquire producer lock
Minimal decode
Flow Preprocessor, queue lookup/ assignment
Enqueue descriptor
Own queue threshold reached
Own queue threshold not reached
Distributed task queues
Dequeue descriptor
Consumer routine
Dequeue descriptor
Full Decode
Full Decode
Stream, HTTP Inspect preprocessors
Stream, HTTP Inspect preprocessors
Own queue not empty
Rule tree lookup, pattern matching
Rule tree data (read-only)
Rule tree lookup, pattern matching
Own queue empty
Verification: string, regex, non-content
Verification: string, regex, non-content
Notification
Notification plugins
Notification

Fig. 3.2. Flow-level parallelization strategy for Snort

cause head-of-line blocking by stalling the producer. Any thread can be the producer, but the producer code is protected by a mutex lock so that only one thread may do so at a time. A thread will continue to act as the producer until its own queue size reaches a threshold, at which point it gives up the producer lock and returns to processing from its work queue. The threshold prevents the producer lock and shared data structures from passing back and forth between processors (which causes expensive cache-to-cache transfers) too often.

Completing the process of flow-level parallelization requires a policy for assigning packet flows to threads. At a minimum, the system must never allow packets from the same flow to be queued or processed at multiple consumer threads at the same time. The stream reassembly preprocessor (stream4) separates TCP sessions based on their IP addresses and TCP ports, and the flow preprocessor considers IP addresses, layer 4 protocol, and ports if applicable, meaning that any assignment scheme that satisfies flow's requirements will also satisfy stream4's. The thread assignment scheme must aim to avoid load imbalance. Static hashes based on IP addresses are fast and simple, but are prone to uneven assignments if the flow distribution on the network is unfavorable. A better approach is to dynamically assign flows to threads, based on information available at runtime.

When a new flow arrives at the system, it is assigned to the thread with the shortest work queue and this assignment is entered into a table. The producer routine must steer later packets from the same flow to the appropriate thread. The producer must determine the packet's flow information, consult a hash table to determine if the flow has an existing assignment, or create an assignment otherwise. Conveniently, these are exactly the actions taken by the flow preprocessor to allow its tracking of flowbits. So, the flow preprocessor is simply run before thread assignment, assigning new flows to the thread that is currently least busy. The flow tracking structure is augmented to include the thread assignment for the flow. Flows are identified using the IP addresses of the hosts and the TCP/UDP port numbers, allowing different flows between the same two hosts to be separated. Using this data requires the decoding of the IP and TCP/UDP headers. However, this is the only part of the overall decode stage that is required. Consequently, only a minimal subset of decode is performed before the flow preprocessor, and a full decode is performed by the consumers in parallel. The flow preprocessor itself need not be run again since a pointer to the flow information is passed in the queue along with the pointer to the packet data.

Because the execution of the flow preprocessor is serialized in the producer routine, it is essential that it execute quickly, or speedup will be severely limited by Amdahl's Law. Fortunately, for the LL traces the flow preprocessor consumes less than 4% of execution time according to the profile data used to generate Figure 2.2. Consequently a parallel speedup of at least 6 should be theoretically obtainable using this method. If the workload causes the flow preprocessor to consume significantly more resources, the potential speedup is reduced. Section 3.3.1 discusses such a case.

Deadlock avoidance. Proving that the flow-based parallelization cannot encounter deadlock requires showing that there can be no cycle of dependences that cause stalling. A consumer can be stalled waiting to receive work from the producer (empty queue) or because its thread is currently serving as producer. In no situation can one consumer routine stall for another. Thus, any cycle must include the producer. The producer can only stall if it is trying to insert a work item into a consumer queue that is already full. If this consumer is not the same as the producer thread, it cannot be stalled since such a consumer will only stall for an empty queue. Thus, a cycle can only be formed because the producer is trying to insert into *its own* full consumer queue. The system avoids this case by causing the producer routine to return control to the consumer once its own work queue is sufficiently full.

3.1.3 Conservative Flow Reassignment

The assignment of flows to threads is critical because the load must be well-balanced to obtain high performance. However, even dynamic assignment is not perfect because the only information available when the assignment decision must be made is the current length of each thread queue. Although this provides a coarse form of load-balancing, it does not account for the fact that some flows will require more inspection time than others or that some will end while others continue. To avoid a situation where one thread has too many active flows while another thread has few or none, it would be desirable to allow a flow to be reassigned to another thread if it is safe to do so.

Reassignment is safe when all the packets from a given flow drain out of the system and finish processing. Changing threads is not harmful in this case because there are no packets from the same flow being *simultaneously* processed by different threads. Changing the assignment here has the same effect as reaching a stream flush point in single-threaded Snort; in addition, neither the flush point selection (which is randomized) nor the flow reassignment (which is essentially random because it depends on the processing speed of Snort and the state of all flows in the network) can be predicted (and therefore exploited) by an attacker. Reassignment also has no impact on deadlock avoidance since it introduces no new stall conditions.

Reassignment is implemented by counting the packets in the system for each flow; the producer increments the counter when the packet passes through flow preprocessor and the consumer decrements it when the packet finishes processing; this is essentially a reference count for each flow. When a flow's packet count reaches zero, its assignment is removed, and it is free to be reassigned to the least-busy thread if the flow reappears. This additional flexibility may improve performance by improving the flow workload balance. It does, however, introduce two complications. First, the stream4 session data remains in the stream table of the original thread, not the new one. This data will be flushed from the stream table after a timeout elapses, ensuring that all packets are inspected. Second, flow reassignment introduces locking overheads, since counters for each flow must be incremented and decremented concurrently.

3.1.4 Optimistic Flow Reassignment

The prime limitation of the flow-based parallelization is that it offers no opportunity for speedup on data streams with only a single network flow. As discussed in Section 3.1.1, data sharing between packets in the same flow stems from two key components: stream reassembly and flowbits. However, these subsystems have certain favorable properties that may enable a relaxation of the requirement. First, packets from the same flow that are separated by a stream reassembly flush point actually have no reassembly-related dependences between them since they will be reassembled into separate stream buffers. Second, only 36% of the rules actually test or set the flowbits used in flow-tracking (and more than 90% of these only apply to NetBIOS packets); rules that do not consider the flowbits have no dependences caused by flow-tracking. If few packets have content matches for these rules, there will be no dependence most of the time.

To allow intra-flow parallelization, the producer must be able to steer packets from the same flow to different consumers while also maintaining flowbits dependences when needed. Spreading a single flow across threads is only valuable when the base flow-concurrent version has a load imbalance. Consequently, the approach studied here starts with the flow-concurrent version and opts to reassign a flow to a different thread if the number of packets in the current thread's queue belonging to the flow are over a certain threshold, called the *reassignment threshold* (providing flow affinity to avoid problems in stream reassembly). The flow will then be reassigned to the least-loaded thread. The first packet after reassignment is then marked with a special flush point indicating that all previous packets from this flow should be reassembled and sent to inspection before attempting to process this packet in stream reassembly. This flush insures proper stream reassembly even when a flow is reassigned to a thread to which it has previously been assigned, making sure that the older packets are not reassembled with the newer ones.

Reassignment must not alter the behavior of flow tracking. However, it only needs to enforce flowbits dependences for packets that actually match rules that use flowbits; the flowbits state is irrelevant for other packets. Detailed statistics show that only about 3% of the packets match flowbits rules for the traces shown in Figure 2.2 except DEF1. The new parallelization stalls the actual testing or setting of flowbits until the packet which has actually matched the rule is the oldest packet from that flow in the system. The system determines the oldest packet by maintaining per-flow reorder buffers, which are simply circular arrays of bits representing the completion state of packets in that flow. (Although larger entries may be helpful for storing information besides the completion state, anything larger would increase memory pressure and reduce the number of reorder buffer entries available in the system.) The flow preprocessor adds an incomplete entry bit to the tail of a flow's reorder buffer whenever it processes a packet. A packet's bit is marked complete when the verification stage completes. If the newly completed packet is at the head of the circular array, the head pointer advances through as many complete entries as possible. Only the packet corresponding to the head of the circular array is allowed to test or set flowbits, but any packet that does not require flowbits may simply mark itself complete and then exit the system. (Unlike register renaming in superscalar processors, intrusion detection cannot use the reorder buffers to rename the flowbits because any given operation that sets flowbits only changes some of the bits. Consequently, such an operation must be considered both a read and a write, making renaming useless.)

This parallelization is optimistic because it assumes that intra-flow dependences will not be common. It then uses that assumption to assign flows to multiple threads. If the optimistic assumption is correct, packets from the same flow need not have any ordering imposed on them and will thus achieve intra-flow parallelism. If the optimistic assumption is incorrect, the system will stall until the dependences are met, providing correct detection of attacks.

Deadlock avoidance. Proving the optimistic parallelization deadlock-free is somewhat more complicated than the conservative version. In addition to the stall cases possible in the conservative case, one consumer may now wait for another as a result of a flowbits condition. Further, the producer may stall because it is trying to process a packet from a flow that has a full reorder buffer. Consumer-to-consumer stall cycles are avoided because each queue is processed in-order. Consequently, the processing of the oldest packet still resident in the system will never have to wait for flowbits condition testing; since it is the oldest in the system, there is no older

Table 3.1
Packet traces used to evaluate the system

Trace Name	**Source**	**Date**	**Size (MB)**	**Packets**	**Alerts**
LL1	Lincoln Lab	4/9/99	991	3,393,919	2,074
LL2	Lincoln Lab	4/08/99	740	3,201,382	3,567
LL3	Lincoln Lab	3/24/99	694	2,453,967	142
DEF1	DEFCON	7/14/01	687	3,960,264	127,672
DEF2	DEFCON	7/14/01	842	1,050,364	395

entry in its flow's reorder buffer. The processing of this packet can only stall if its consumer thread is currently acting as the producer. As in the conservative case, the producer will revert control to the consumer if its queue is sufficiently full. The only remaining case is if the producer thread is stalled waiting to insert an entry into a full reorder buffer. If the producer ever encounters a full reorder buffer, it does not immediately know if it is responsible for processing the oldest packet in the flow since the reorder buffer is nothing more than an array of bits. The producer avoids deadlock by checking each entry in its own queue to see if it is responsible for resolving the dependence. If so, the producer puts back the current packet (so that some later producer can handle it) and reverts control to the consumer. Although traversing its own queue is potentially an $O(N)$ operation, it is an extremely unlikely event; additionally, it takes place when the producer would already be stalled waiting for reorder-buffer space, so the cost is not a concern.

3.2 Experimental Methodology

The system studied here is based on Snort version 2.6RC1, downloadable from `www.snort.org`. A few modifications were made to snort that are independent of the parallelization strategy. The most important of these is the use of a large in-memory buffer from which to read the packets while processing, to minimize the system-dependent effects of reading directly from a file or network interface. In practice, this may be an important component of IDS performance, but separate solutions exist to address this problem, such as a version of `libpcap` that uses the `mmap()` system call to map a kernel ring buffer into Snort's address space, thus avoiding the overhead of copying packets to userspace. The measured runtimes for the tests do not include copying the packets from the trace file into the memory buffer, nor printing out statistics data after processing, but only cover reading the packets from the memory buffer, processing them, and generating alerts.

Snort is designed on a plugin architecture for almost all aspects of packet processing. Preprocessors, detection mechanisms, and notification methods are all based on modular plugins that may be mixed and matched according to the specifications of the user and rule writer. The system supports the stream4, flow, and HTTP Inspect preprocessor plugins, all the standard detection plugins (those that are not required to be explicitly enabled in the configuration file), and the "fast" alert method, which consists of writing a line to a text file for each alert generated. Packet logging was disabled. In practice, it is common for large installations to use an external database for collecting alert and log data, which may be on another machine. These and other methods can be supported by making their plugins reentrant. The multi-pattern matching algorithms are also abstracted from the rest of Snort's architecture; the modified Wu-Manber algorithm (the default up until version 2.6) is used in the parallel Snort.

As of Snort version 2.4, the rules and signatures are no longer distributed and released along with Snort itself. Instead, they are updated more often and may be downloaded separately. This paper uses the ruleset released on March 29, 2006. All rules are enabled except those marked as deleted or deprecated. In addition, many rules refer to a variable, such as "home net" or "HTTP servers" (for example, to check for patterns on streams that are only incoming or only bound for a user's web servers), which may be configured to refer to the user's own systems or network. In this study, however, these terms were set to "any" to catch all possible attacks. Other configuration variables exist to define which ports run particular services, and these were left at their defaults. The preprocessors' configurations were also left at their defaults.

Tests were run and analyzed two different types of machines: The first is a 2U Dell Poweredge 2950 server with two 1.8 GHz quad-core Intel Xeon E5320 processors based on the Core 2 microarchitecture (8 processors total). This system has 16 GB of system RAM and 4 MB L2 caches shared between pairs of processors. This system runs Linux kernel 2.6.18 and GNU C library 2.3.6 with the Native POSIX Threads Library in the Debian AMD64 distribution. The second is a 1U rack-mounted Sun Fire X4100 server with two 2.2 GHz dual-core AMD Opteron processors (4 processors total). This system has 4 GB of system RAM and 1 MB private L2 cache per processor core. The system is run using Linux kernel version 2.6.15 and the GNU C library 2.3.6 with the Native POSIX Threads Library in the same Debian AMD64 distribution. The Linux kernel supports affinity scheduling to maintain threads on the same processor whenever possible. Instead of the standard pthread mutex locks, both parallelization methods use the pthread spin locks provided by the GNU C library. Unlike the standard mutex locks, these lock primitives do not suspend the calling thread when they encounter a lock that is already held by another thread; instead, they simply spin-wait until the lock is free. Because the system uses only as many threads as there are processors, the threads do not need to yield the processor, and critical sections are short enough that the overhead of invoking the operating system to block the thread is much higher than simply spinning until the lock is free. On x86 platforms, the spin locks are implemented using an atomic compare-and-swap instruction.

The packet traces used to test the system come from the 1998-1999 DARPA intrusion detection evaluation at MIT Lincoln Lab and from the Defcon 9 Capture the Flag contest [27, 28]. The Lincoln Lab traces are simulations of large military networks generated during an online evaluation of IDSes and are available for download. Because they were generated specifically for IDS testing, (including anomaly-based detection systems, which require realistic traffic models to be useful) the traces have a good collection of ordinary-looking traffic content and also contain attacks that were known at the time. The traces used here are the largest available in the set, and come from the 1999 test. The Defcon traces are logs from a contest in which hackers attempt to attack and defend vulnerable systems. Consequently, these traces contain a huge amount of attacks and anomalous traffic, representing a sort of pathological case for intrusion detection systems. For example, DEF1 generates a very large number of alerts (even compared to the LL traces, which are seeded with real attacks). Table 3.1 shows a summary of the traces used, their source, their capture date, the number and total size of the packets they contain, and the resulting number of alerts.

3.3 Results and Discussion

This section gives experimental results for the parallelization strategies, using the hardware platform and traces described in Section 3.2. Table 3.2 gives the throughput achieved by the standard uniprocessor Snort for each of the traces; the results for parallel speedup are relative to these performance levels. The average throughput of the uniprocessor Snort is 699 Mb/s for the Intel server and 548 Mb/s for the AMD server.

3.3.1 Flow-concurrent Parallelization

Figure 3.3 shows the parallel speedup achieved by the most conservative scheme on the Xeon-based Dell system, and Figure 3.4 shows the speedup for the Opteron-based Sun Fire system. Each line represents one of the packet traces, with parallel speedup plotted against number of threads. The plots show that the flow concurrent scheme achieves good speedup on the three LL traces but not on the DEFCON traces. The conservative parallelization sees an average traffic rate of 2.03 Gb/s across the five traces for the Intel machine, and 1.36 Gb/s for the AMD machine.

Table 3.2
Uniprocessor performance levels achieved by Snort for traces and hardware platform described in Section 3.2.

Trace Name	**Intel Throughput**		**AMD Throughput**	
	$\frac{packets}{sec}$	Mb/s	$\frac{packets}{sec}$	Mb/s
LL1	283,433	694	214,150	525
LL2	241,923	469	190,391	369
LL3	337,854	801	238,644	566
DEF1	298,271	434	236,991	345
DEF2	162,700	1094	138,675	933

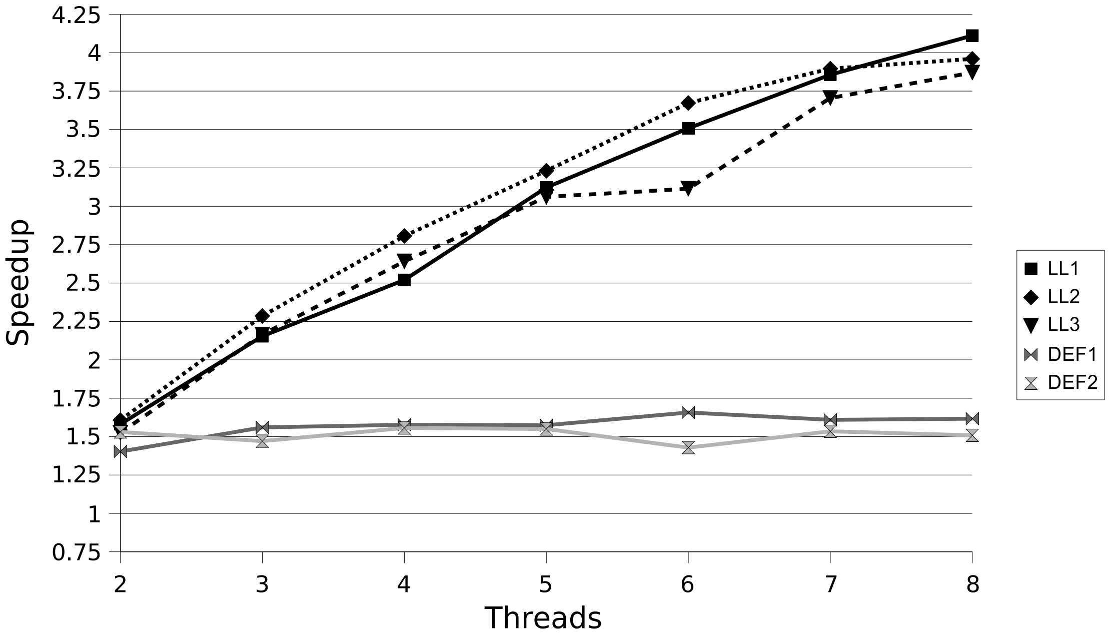


Fig. 3.3. Parallel speedup for pure flow-concurrent parallelism on Dell Poweredge 2950 server with two quad-core Xeon processors (8 processors total)

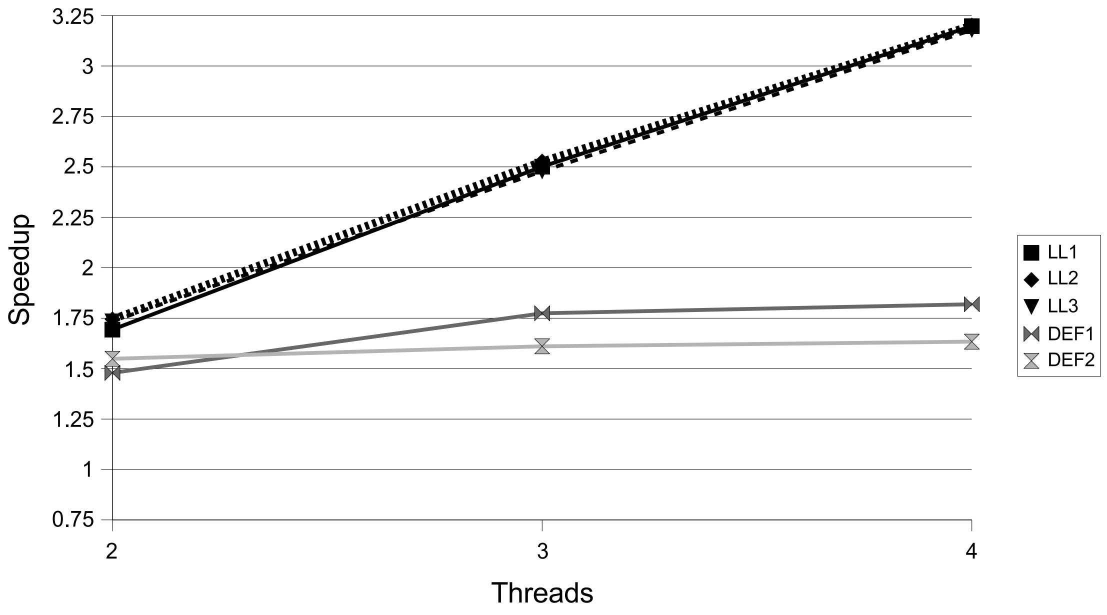


Fig. 3.4. Parallel speedup for pure flow-concurrent parallelism on Sun Fire X4100 server with two dual-core Opteron processors (4 processors total)

The three LL traces have similar speedup characteristics, achieving 63–79% of the theoretical ideal linear speedup for 2–4 threads, but decreasing to as low as 48% for 8 threads, with a peak speedup of 4.1 on the Intel machine. The AMD machine fared better, achieving 79–87% of linear speedup for 2–4 threads with approximately 3.2 speedup at 4 threads. All 3 traces see processing rates in excess of 1 Gb/s with 4 threads even on the slower AMD platform; LL1 and LL3 achieve this rate with 3. The peak processing rate is 3.1 Gb/s for the Intel machine and 1.8 Gb/s for the AMD machine. The two factors that limit performance in these cases are a small amount of imbalance (sometimes more than one thread ran out of work at the same time) and synchronization and data transfer overheads (primarily in the form of cache-to-cache transfers between processors). In particular, the work queues for each thread represent transfer of the packet inspection descriptors from the producer to the consumers and especially the descriptors for the queues themselves; i.e., every time a producer enqueues or a consumer dequeues a packet descriptor, it invalidates the line containing the queue descriptors in the other's cache, which results in a large amount of cache-to-cache transfers. In addition, the amount of coherence traffic on the memory bus may become significant with larger numbers of processors. Although its absolute inspection rate is lower, the parallel speedup seen on the AMD machine is significantly better than that of the Intel machine; on average 20% better across the LL traces for 4 threads. This may be because the HyperTransport bus connecting the Opteron chips allows lower-latency cache transfers than the Xeon's front-side bus.

In contrast, the DEFCON traces, and in particular DEF2, achieves little speedup in any case. As discussed previously, the DEF1 and DEF2 workloads behave very differently from the others. As it turns out, DEF2 has extremely poor flow concurrency; in fact, for much of the trace there is only one active flow, so no purely flow-based parallelization scheme can hope for any significant improvement. DEF1 has several factors which contribute to poor performance. First, it triggers an extreme number of alerts, and because alerts require synchronization, significantly more time is spent waiting for locks with DEF1. Second, DEF1 apparently contains attack attempts which create and abandon huge numbers of flows. This is the cause of the large preprocessing time seen in figure 2.2; in fact, for the single-threaded case, over 17% of the total time is spent in the flow preprocessor searching and updating the hash table containing the flows. This limits the speed of the producer routine, and thus the whole system. Lastly, despite the load on the flow preprocessor, DEF1 also has relatively poor flow concurrency, because the created flows are quickly abandoned and most of the actual traffic is concentrated in a relatively small number of flows. These last two problems exacerbate each other, because threads acting as consumers are more likely to run out of data when the producer is slower. These effects also combine to greatly increase the lock and cache-transfer overhead because multiple threads with empty queues may compete for the lock which protects the producer routine, and the data used by the producer routine is transferred frequently. Of course, the source and nature of the DEFCON workload mean that diminished performance is to be expected; it reflects a very small network and an extremely adversarial environment where nearly all traffic is malicious. A system that detects such a high rate of alerts may respond quickly by more aggressive firewalling to shut off traffic on vulnerable ports or from IP addresses observed to participate in malicious behavior. The other traces are actually more likely to be dangerous since they have a small number of attacks hidden in a larger amount of "normal" traffic. Consequently, such workloads are more important and realistic for IDS testing, and the conservative parallelization performs quite well on 3 of those 4 workloads.

3.3.2 Conservative flow reassignment

Figures 3.5 and 3.6 show the parallel speedup achieved by the Dell and Sun servers when conservative flow reassignment is added. As discussed in Section 3.1, flow assignment is not perfect, and sometimes unlucky assignments can cause threads not to have enough work. Allowing some reassignment should in theory reduce these cases and improve performance, or at least should not have a negative impact because it

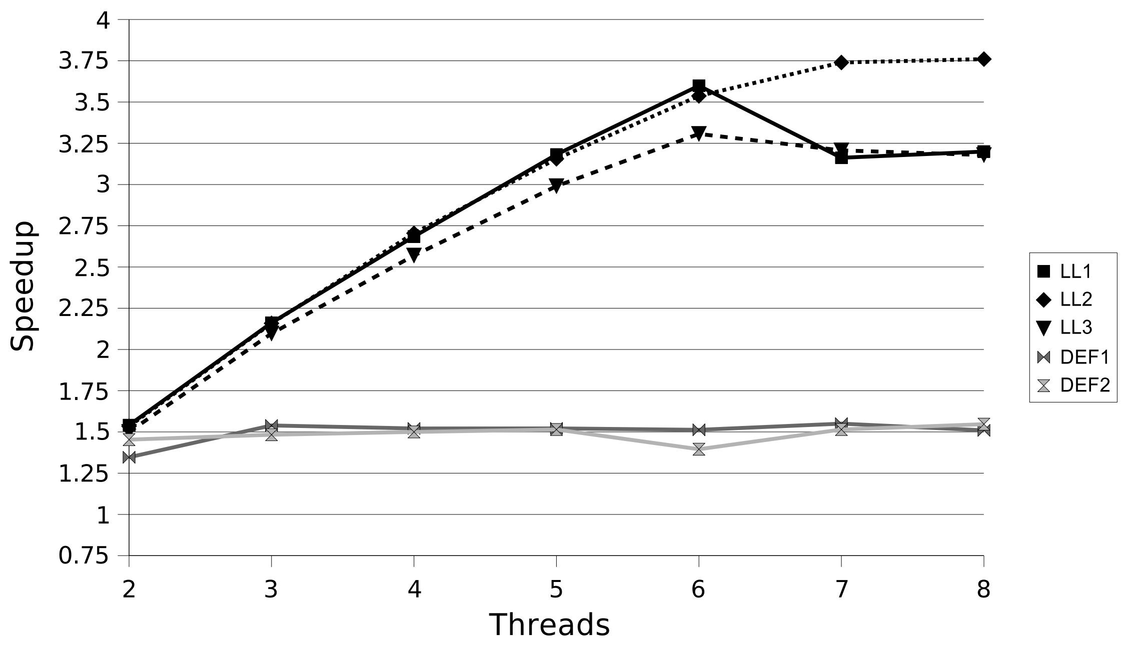


Fig. 3.5. Parallel speedup for conservative flow reassignment on Dell Poweredge 2950 server with two quad-core Xeon processors (8 processors total)

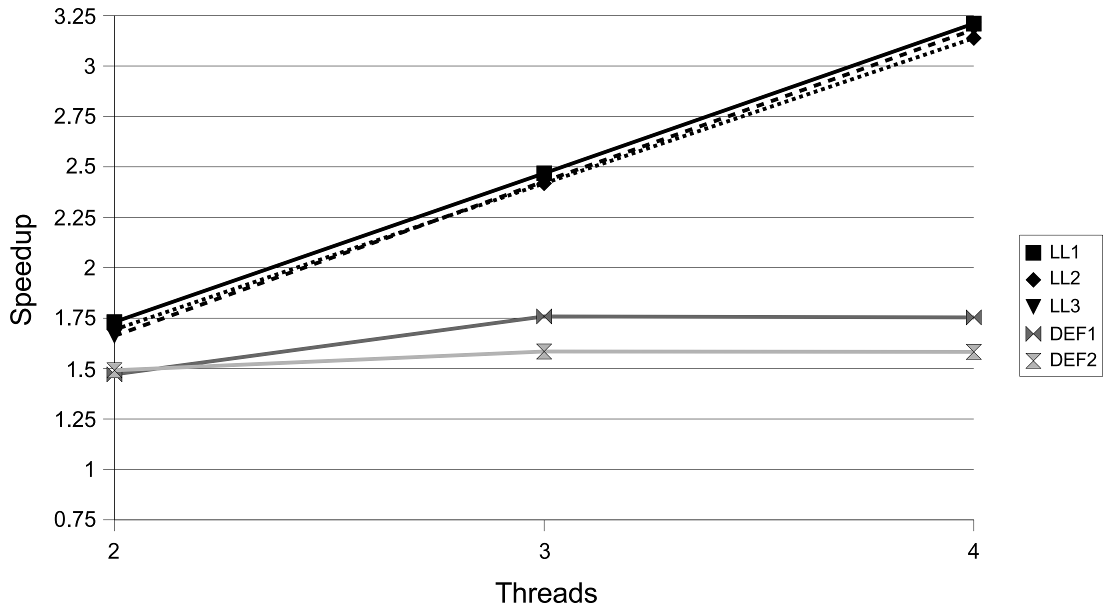


Fig. 3.6. Parallel speedup for conservative flow reassignment on Sun Fire X4100 server with two dual-core Opteron processors (4 processors total)

does not affect the total amount of packet inspection work to be done. However Figure 3.6 is nearly identical to Figure 3.4, indicating that there is no improvement on the AMD platform. Moreover, Figure 3.5 indicates that in most cases adding conservative reassignment actually does reduce performance on the Intel platform. This is particularly noticeable for the 7 and 8 thread cases, for which performance is as much as 22% worse, and which are actually worse than the 6 thread case for the LL1 and LL3 traces. This can be explained with a closer look at the implementation of the conservative flow reassignment with respect to synchronization, remote cache invalidations, and cache-to-cache data transfers. For each packet produced, the producer must acquire the lock on the flow data structure to which the packet belongs, make a new assignment if necessary, and increment the flow's reference count. The consumer must likewise acquire the lock, decrement the reference count, and remove the flow's thread assignment if its reference count has reached zero. This results in more synchronization overhead, remote cache transfers, and coherence traffic for every packet; combined with the overheads discussed in Section 3.3.1, the cost of this strategy is enough to offset any benefits gained by the more flexible flow assignment scheme.

3.3.3 Optimistic flow reassignment

DEF2, as mentioned, has poor flow concurrency, and is thus a good candidate for improvement using the optimistic flow reassignment method. Figures 3.7 and 3.8 show the performance using the optimistic reassignment method for the Intel and AMD platforms, with a reassignment threshold of 100 (about 10% of the queue length). DEF2 indeed shows benefits over the conservative method, improving performance on the Intel machine by as much as 68% for 6 threads, with a 2.4 speedup and a 2.63 Gb/s rate. Likewise the performance on the AMD machine improved by about 45% for 4 threads to achieve a factor of 2.38 parallel speedup and a peak traffic rate over 2.2 Gb/s. The value chosen for the reassignment threshold should be small

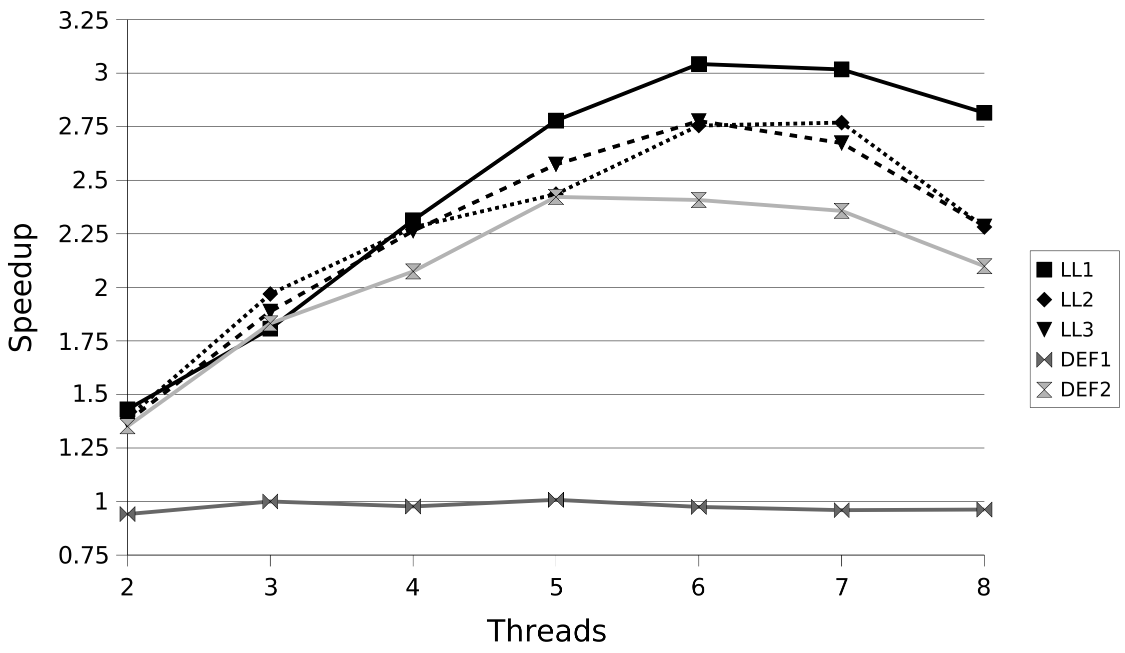


Fig. 3.7. Parallel speedup for optimistic flow reassignment on Dell Poweredge 2950 server with two quad-core Xeon processors (8 processors total)

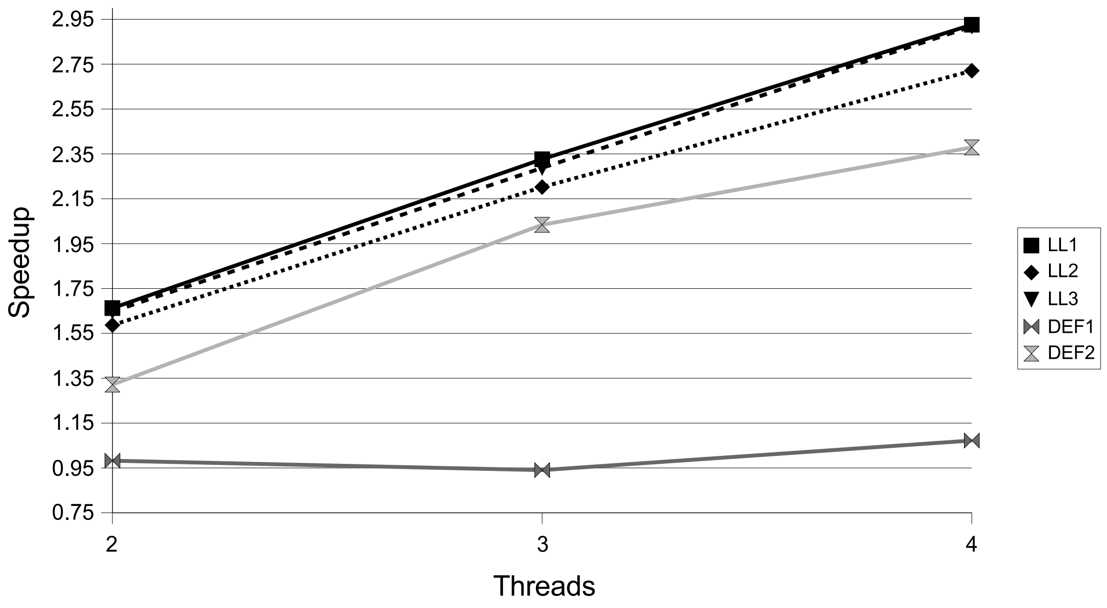


Fig. 3.8. Parallel speedup for optimistic flow reassignment on Sun Fire X4100 server with two dual-core Opteron processors (4 processors total)

enough so that when a packet matches a flowbit rule and must wait for previous packets, it does not have to wait too long (since the number of packets ahead of it can be no more than the threshold multiplied by the number of other queues). However it must be large enough to avoid excessive switching (which causes too much flushing and other overhead in stream reassembly). In practice, 100 is a good balance.

Since the LL traces already have good flow concurrency, optimistic flow reassignment provides no benefit; in fact, their performance is degraded by 26.5% on average across all tests with the LL traces compared to no reassignment because of the overhead of maintaining the reorder buffers. In addition to the overhead of the conservative flow reassignment, updating and checking the bits in the buffer and the head and tail pointers must be done even when packets in a flow are serialized, and this must be done while holding the mutex lock associated with the flow, leading to additional synchronization overhead. DEF1 also does not benefit from reassignment because it matches so many rules, and most of the packets actually trigger setting or checking of flowbits. Thus, any advantage gained by reassigning flows is erased because so many flows must serialize themselves. Further, since the serialized flows are spread across all the threads, they even block flows behind them that might otherwise have been able to pass them. Consequently, the overall rate in DEF1 is reduced to approximately that of the single-threaded case.

As with the non-reassignment scheme, the AMD machine gets better parallel speedups in both reassignment schemes than the Intel machine for the same number of threads. In fact the advantage increases to almost 26% on average for 4 threads with optimistic reassignment. As synchronization overhead and read-write sharing of data increase with the addition of conservative and then optimistic reassignment, so does the parallel speedup advantage of the AMD system over the Intel system, providing more evidence that the AMD system handles this sharing better.

Because the overheads associated with the scheme increase with the number of processors, the throughput achieved actually peaks at 5 or 6 processors and then decreases as more are added. Despite the degradations in processing some of the traces, the average of the best traffic rates achieved for each trace using the optimistic reassignment is roughly the same as that of conservative reassignment at 1.74 Gb/s, but does not reach the 2.03 Gb/s average seen by the non-reassignment scheme. The optimistic parallelization sees good parallel speedup for 4 out of 5 traces, though these are somewhat lower than the conservative version for 3 traces.

3.3.4 Discussion

The results in this section indicate substantial benefits from parallelization in the Snort network intrusion detection system. For most realistic scenarios with many simultaneous packet flows, conservative flow-based parallelism without reassignment is sufficient. Conservative flow reassignment sometimes provided benefits but was ultimately not scalable because of the overhead of maintaining per-flow counters. Networks with poor flow concurrency can see benefits from optimistic reassignment of flows, provided that the number of packets that must check flowbits is limited. All of these parallelization strategies are achieved while using hardware that is increasingly becoming commoditized, allowing for fast single-node edge-based network intrusion detection. As architectures continue to evolve, all expectations are for more multicore and multiprocessor solutions and less potential benefit from ramping up clock frequency. Thus it is essential for an application as important as network intrusion detection to achieve its performance by exploiting fine-grained flow-level and intra-flow parallelism.

The effectiveness of optimistic flow reassignment may vary substantially with the conditions used for reassignment. Experiments with reducing or increasing the flow reassignment threshold suggest that thresholds below 100 suffer from the overhead of additional flushes in stream reassembly while larger thresholds cause excessive stall times when flowbits rules actually match. Additional modifications were tested that would prevent reassignment if the queue length at the reassignment target were at least a certain percentage of the queue length of the origin (indicating some load balance factor). However, this condition had little impact on performance even as this threshold was varied from as high as 50% all the way down to 10%. Further modifications seemed promising, but had little or no impact (for example, disabling reassignment on NetBIOS flows as these account for most flowbits rules, or moving the flowbits tests to the end of rule processing so that they would only be checked if all other conditions matched).

The optimistic system is also still conservative in how it manages rules that actually use flowbits since it stalls when a flowbit is to be set or checked out-of-order. An alternative would be to speculatively perform the flowbit operation and then roll back if there was a violation, but such rollbacks would require the reprocessing of many packets and a great deal of stored state.

4. DESIGN ALTERNATIVES FOR A HIGH-PERFORMANCE SELF-SECURING ETHERNET NETWORK INTERFACE

4.1 A Self-Securing Network Interface Architecture

As discussed in Section 2.2, programmable network interfaces have been proposed as resources for offloading various forms of network-based computation from the host CPU of high-end PC-based network servers. Offloading intrusion detection from the host could be quite valuable because this service potentially requires processing every byte sent across the network. This thesis proposes a self-securing Ethernet controller that uses multiple RISC processors along with event-driven protocol processing firmware and special-purpose hardware for string matching. Figure 4.1 shows the architecture of the proposed Ethernet controller, which is based upon the 10 Gigabit Ethernet design by Willmann et al. [8]. The individual components target the following uses:

- Programmable processors: control-intensive computation, needs low latency
- Memory transfer assists (DMA and MAC): data transfers to external interfaces, need high bandwidth
- String matcher: data processing, needs high bandwidth
- Graphics DDR (GDDR) SDRAM: for high capacity and high bandwidth access to network data and dynamically allocated data structures
- Banked SRAM scratchpads: for low latency on accesses to fixed control data
- Instruction caches: for low latency instruction access
- Data cache: for low latency on repeated data accesses

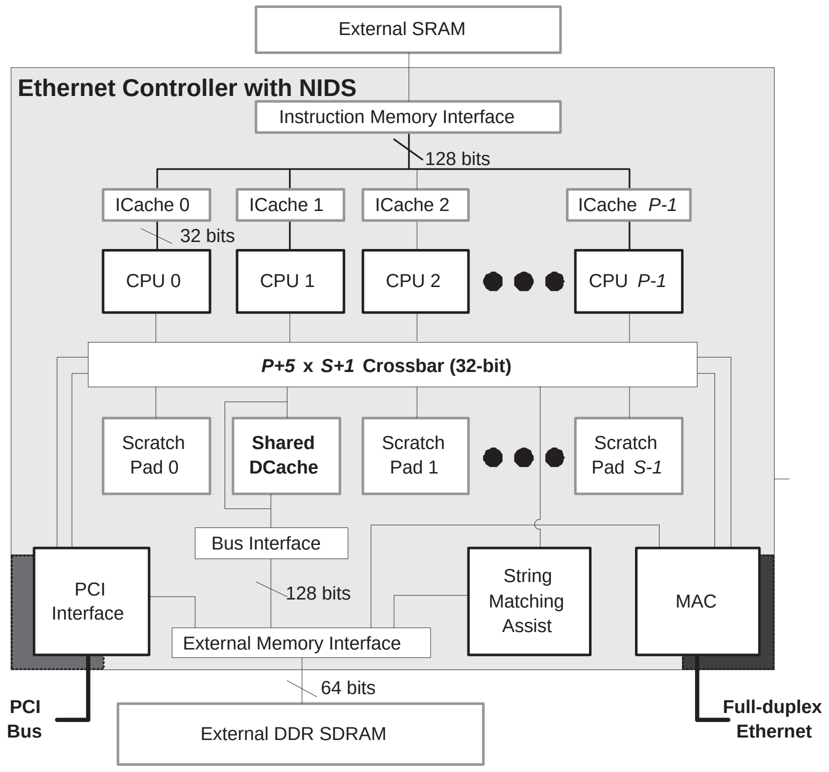


Fig. 4.1. Block diagram of proposed Ethernet controller architecture

Of these components, the data cache did not appear in the previous works discussed. The remainder of this section 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 ISA 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 dependences between them. Additionally, alternatives to parallelism such as higher frequency or superscalar issue are not viable options because network interfaces have a limited power budget and limited cooling space.

Memory Transfer Assists. As in previous programmable network interfaces, this system includes hardware engines to transfer data between the NIC memory and the host memory or network efficiently [7, 8, 29]. 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 is also used to offload TCP/IP checksumming from the host operating system [30].

String Matching Hardware. High-performance network intrusion detection requires the ability to match string content patterns efficiently. Thus, the controller integrates string matching assist hardware. This assist could be based on almost any of the previously-proposed ASIC, TCAM, or FPGA-based designs mentioned in Section 2.1. 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). Since previously-reported string and regular expression matchers have been shown to match content at line-rate (e.g., [9, 11–14, 20]), the only potential slowdown would be reading the input data from memory. This penalty can be managed by allocating data appropriately in the memory hierarchy.

One major difference from conventional Snort string matching, however, is the use of a single table for all strings as described in Section 2.1; in contrast, standard Snort uses separate tables for each set of network ports. Such a solution may not be feasible for embedded hardware string matchers that have limited storage capacity. Consequently, fast multipattern string matching alone will likely generate false positives for rules that only apply to other ports.

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. Two Micron GDDR SDRAM chips can provide a 64 MB memory with a 64-bit interface operating at speeds up to 600 MHz, double-clocked [31]. Additionally, the memory system includes a shared data cache to allow the programmable processors to use DRAM-based frame data when needed 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 does not maintain hardware coherence with respect to uncached accesses from the assists (which are analogous to uncached DMAs from I/O devices in a conventional system). Instead, the code explicitly flushes the relevant data from cache any time it requires communication with the assists. The system adopts a sharing model based on lazy release consistency (LRC), with explicit flushes before loading data that may have been written by the hardware assists [32]; unlike true LRC, this system need not worry about merging updates from multiple writers because the cache is write-through. An alternate design is also evaluated which uses private caches for each processor. Unlike the shared cache, the private caches can provide single-cycle access. However, flushes are also required to maintain coherence between processors. Each time a processor acquires a lock that protects cacheable data, it checks if it was the last processor to own the lock. If not, it must flush its cache before reading because another processor may have written to the data.

Table 4.1 summarizes the default architectural configuration parameters for this controller.

4.2 Integrating IDS into Network Interface Processing

Chapter 2 described the steps in the Snort network intrusion detection system and in programmable Ethernet protocol processing firmware. A self-securing programmable Ethernet NIC integrates these two tasks. Achieving high performance requires analyzing and extracting the concurrency available in these tasks and mapping the computation and data to the specific resources provided by the architecture. Previous work has shown strategies to extract frame-level and task-level concurrency in programmable Ethernet controller firmware [8,29]. Consequently, the following will

Table 4.1
Default architectural parameters

Parameter	Value
Processors	2–8 @ 500 MHz
Scratchpad banks	4
Private I-cache size	8 KB
Shared D-cache size	64 KB
Cache block size	32 bytes
Cache associativity	2-way
GDDR SDRAM chips	2×32 MB
DRAM frequency	500 MHz
Row size	2 KB
Row activation latency	52 ns

focus on executing Snort efficiently for the architecture of Section 4.1. The resulting self-securing Ethernet network interface is called LineSnort.

The architecture of LineSnort requires parallelism to extract performance. Since both Snort and Ethernet protocol firmware process one packet at a time, packet-level concurrency seems a natural granularity for parallelization. Recall the Snort processing loop depicted in Figure 2.1. Unlike conventional processors, LineSnort does not need a separate stage to read packets since this is already done for protocol processing itself. The remaining stages, however, present some obstacles that must be overcome. In particular, the data structures used by stream reassembly must be shared by different packets, making packet-level parallelization impractical.

Stream reassembly is implemented using a tree of TCP session structures that is searched and updated on each packet so that the packet's payload is added to the correct stream of data. A TCP session refers to both directions of data flow in a TCP connection, considered together. The information related to the TCP session must be updated based on each TCP packet's header and can track information such as the current TCP state of the associated connection. Because the session structure tracks the TCP state and data content of its associated connection, packets within each TCP stream must be processed in-order. However, there are no data dependences between different sessions, so a parallelization that ensured that each session would be handled by only one processor could access the different sessions without any need for synchronization. Such a parallelization scheme implies that the Snort processing loop should only use session-level concurrency (rather than packet-level concurrency), at least until stream reassembly completes.

It is worth noting that the IDS itself can be the target of denial-of-service attacks, and the stream reassembly stage is a vulnerable target component, which is doubly true when detection is hardware-assisted. Fortunately, Snort's stream preprocessor already contains code which attempts to detect and mitigate the effects of such attacks, and this code can also be used by LineSnort.

Figure 4.2 shows the stages of Snort operating using this parallelization strategy. The parallel software uses distributed task queues, with one per processor. When a packet arrives in NIC memory and is passed to the intrusion detection code, it must 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. Stream reassembly is then performed for each session using the steps described in Section 2.1. 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; the buffers are then passed to the hardware pattern matching assist by enqueueing a command descriptor. The reassembler also enqueues a descriptor for HTTP inspection if the stream is to or from an HTTP port.

Although the above assignment strategy parallelizes stream reassembly and other portions of Snort using session-level concurrency, it provides no benefit for situations where there are fewer flows than processors (such as a high-bandwidth single-session attack). Such a situation can be supported by exploiting a property of stream reassembly: packets from the same flow that are separated by a stream reassembly flush point actually have no reassembly-related dependences between them since they will be reassembled into separate stream buffers. Consequently, if the processor enqueueing a new packet finds that its destination queue is too full (and that another queue is sufficiently empty), it can change the assignment of the flow to the emptier queue. This works because the stream reassembler is robust enough to handle flows encountered mid-stream (in the new queue), and, after a timeout occurs, will inspect and purge any streams that have not seen any recent packets (in the old queue). It is also possible that a flow may be assigned away from a queue and then later assigned back to it. To prevent improper reassembly of these disjoint stream segments, a flush point is inserted before the new section is added to the reassembly tree. Reassignment achieves better load balance and supports intra-flow concurrency for higher throughput; the cost is additional overhead in stream reassembly. In addition, if reassignment were to be too frequent, (i.e., on almost every packet), the benefits of stream reassembly would be negated; the requirement that the shortest queue be sufficiently empty prevents this.

The string content matching assist dequeues the descriptor passed in from stream reassembly, scans the reassembled stream 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 (which may include multiple strings), but the matcher only checks for the longest exact content string related to each

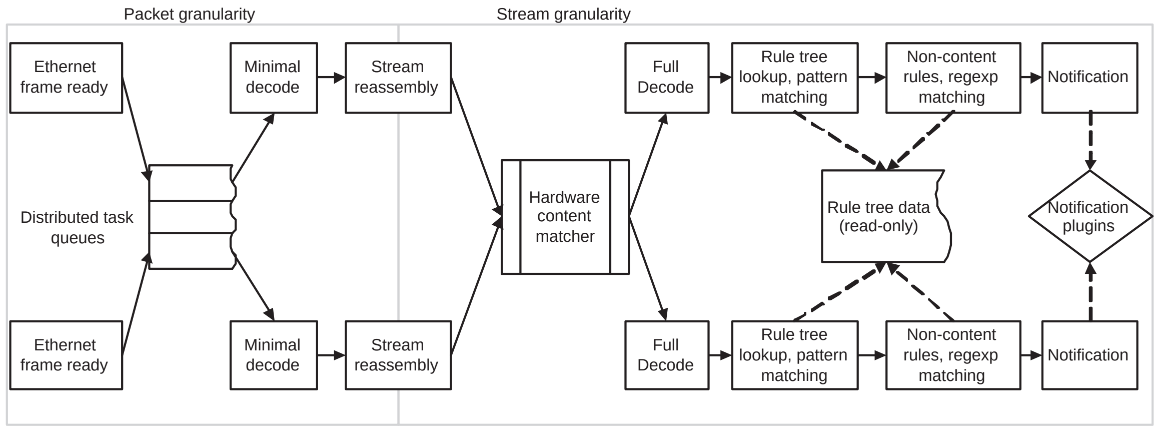


Fig. 4.2. Firmware parallelization strategy used for LineSnort

rule. The verification stage checks each condition for the rule and handles alerting if necessary. At the same time, another processor may also dequeue the HTTP inspect descriptor if one was generated. The HTTP inspect portion of the firmware performs URL normalization and matches “URI-Content” rules to check for HTTP-specific attacks. If any “URI-Content” matches are found, verification is performed in the same manner as for normal content rules. The HTTP Inspect and verification stages are essentially unmodified from the original Snort. The rule data structures are allocated in DRAM since they are not updated while processing individual streams; the HTTP inspection data structures are allocated in the scratchpad, with one per processor. The notification stage shares common alerting mechanisms; no attempt was made to privatize this stage since it represents only 1.5% of the total computation even with complete serialization.

The Snort stages described above integrate into Ethernet protocol processing after new data arrives at the NIC (transmit or receive). Currently, LineSnort only acts to *detect* intrusions, just like the standard Snort; thus, it passes the frame onto the remainder of Ethernet processing regardless of whether or not there was a match. LineSnort could be used to *prevent* intrusions by dropping a suspicious frame or sending a reset on a suspicious connection, but that choice is orthogonal to the basic design of the system.

The firmware uses the scratchpads for all statically-allocated data (e.g., Ethernet processing control data, inter-stage 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.

4.3 Evaluation Methodology

The architecture and firmware described in the previous sections are evaluated using the Spinach simulation infrastructure [33]. Spinach is a toolkit for building network interface simulators and is based on the Liberty Simulation Environment [34]. Spinach includes modules common to general-purpose processors (such as registers and memory) as well as modules specific to network interfaces (such as DMA and MAC assist hardware). Modules are connected through communication queues akin to the structural and hierarchical composition of an actual piece of hardware. Spinach has previously been validated against the Tigon-II programmable Gigabit Ethernet controller, has been used for studying architectural alternatives in achieving 10 Gigabit Ethernet performance, and has been extended to model graphics processing units [8, 33, 35].

Evaluating IDS performance is a difficult problem for several reasons. IDS performance is sensitive to the ruleset used and to packet contents, but there are very few network traces available with packet contents because of size and privacy concerns. In addition, programmable NIC performance is sensitive to packet size. Because of its method of distributing flows to processors, LineSnort is also sensitive to *flow concurrency*, the number of active flows running at one time. Hence, the traces used for evaluation should be realistic in terms of the packet and flow contents and the average packet sizes. The rulesets used should also be reasonable for the target machine.

The rulesets used are taken from those distributed by Sourcefire, the authors of Snort. Sourcefire's ruleset contains over 4000 rules describing vulnerabilities in all kinds of programs and services. Although an edge-based NIDS would need to protect against all possible vulnerabilities, a per-machine NIDS such as LineSnort only needs to protect against the vulnerabilities relevant to that particular server. For example, a Windows machine has no need to protect against vulnerabilities that exist only in Unix, and vice versa. Two rulesets are used in these tests: one with rules for email servers and one with rules for web servers. Both rulesets also include rules for common services such as SSH. The mail ruleset contains approximately 330 rules, and the web ruleset approximately 1050. Tailoring the ruleset to the target machine allows LineSnort to consume less memory and see fewer false positives.

LineSnort is evaluated using a test harness that models the behavior of a host system interacting with its network interface. The test harness plays packets from a trace at a specified rate, and Section 4.4 reports the average rate sustained by LineSnort over the trace. The rules used for testing do not distinguish between sent and received packets, so all traces are tested using only the send side. Although all steps related to performing DMAs and network transmission are included in the LineSnort firmware, the actual PCI and Ethernet link bandwidths are not modeled, as these are constantly evolving and can be set according to the achieved level of performance.

The packet traces used to test LineSnort come from the 1998-1999 DARPA intrusion detection evaluation at MIT Lincoln Lab, which simulates a large military network [27]. Because they were generated specifically for IDS 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 IDS efficacy rather than performance, they are not realistic with respect to packet sizes or flow concurrency. To address this problem, a variety of flows were taken from several traces and reassembled to more closely match average packet sizes ($\approx$ 778 bytes) seen in publicly available header traces from NLANR's Passive Measurement and Analysis website (pma.nlanr.net).

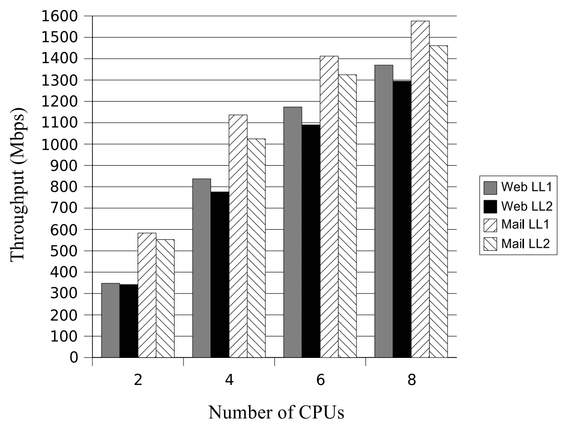


Fig. 4.3. LineSnort throughput results achieved with 2–8 CPUs for mail and web rulesets with LL1 and LL2 input traces

Two traces, called LL1 and LL2, were created in this manner, using different ordering and interleaving among the flows.

4.4 Experimental Results

Figure 4.3 shows the base results for the Lincoln Lab traces using the web and mail rulesets for varying numbers of CPUs. There are many factors, both in hardware and software, that affect performance.

Rulesets. Rulesets have an important effect on any IDS, and the choice of rules will always involve performance tradeoffs. In LineSnort, 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 will be for rules that do not ultimately lead to alerts; these "false positives" will be filtered out by the verification stage. A false positive 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 will need to read the packet and process its data using the matching algorithms. The mail ruleset actually generates more false positives at the string

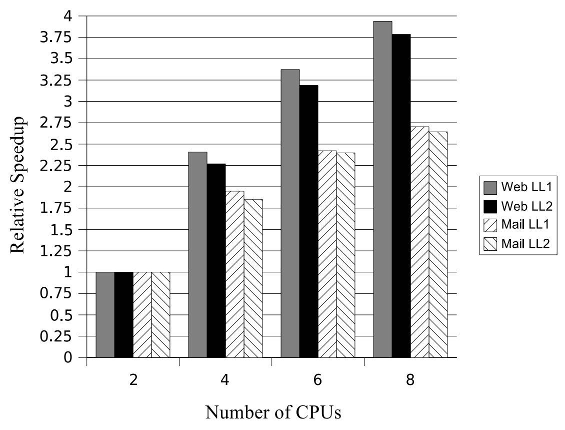
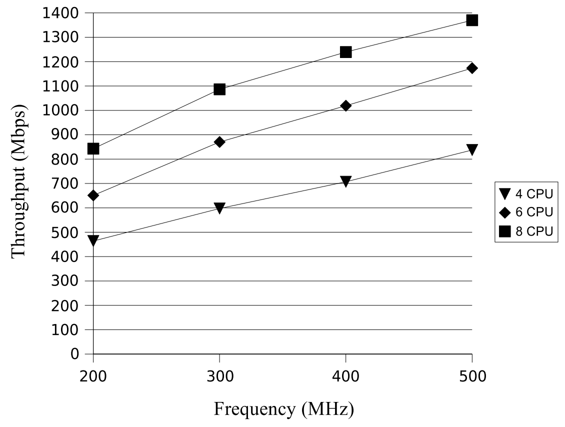
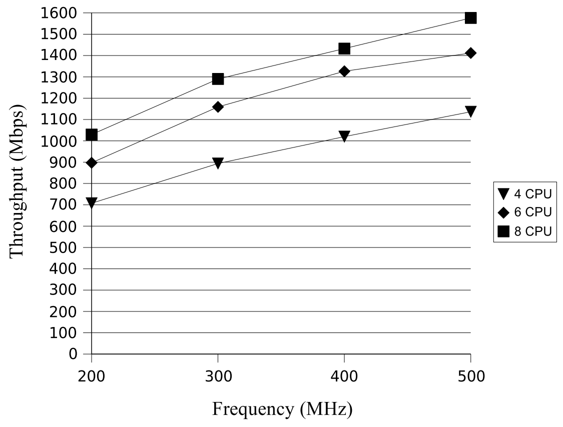


Fig. 4.4. Relative speedup for increasing numbers of processors



(a) Web ruleset



(b) Mail ruleset

Fig. 4.5. Impact of CPU frequency for LL1 trace

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 on 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 4.4 shows the speedup obtained by increasing the number of processors, normalized to the 2-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 4 processors than there is with the web ruleset with 8. 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 thus has lower overall throughput. Verification reads the reassembled packet data from scratchpad rather than DRAM, so the overall workload is more balanced between the scratchpad and the DRAM for the web ruleset, making increased DRAM contention less important. Moreover, because the scratchpad is banked, the increase in processors causes less contention there than for the cache.

Figure 4.5 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.

Flow Assignment. One of the most important influences on overall performance is the assignment of packet flows to processor queues. The assignment must be as balanced as possible to keep the processor load balanced. Otherwise, if the load imbalance is enough to fill up one of the stream reassembly queues, head-of-line blocking can occur, and all incoming packets must wait for the full queue. We evaluate two primary methods for queue assignment. In the *static assignment* method, the source and destination IP addresses are hashed in a symmetric manner, with the

Table 4.2
Speedup for hardware-assisted string matching

	Web Ruleset	Mail Ruleset
LL1 Trace	1.8	2.8
LL2 Trace	1.4	2.4

queue determined directly from the hash, so that packets from the same TCP session are always in the same queue. In *dynamic assignment*, the source and destination IP addresses are looked up in a hash table. If the stream has an entry, the queue stored in the table is used. Otherwise the stream is assigned to whichever queue is currently shortest, and the entry is stored, ensuring subsequent packets from the stream will go into the same queue. Dynamic assignment is then further enhanced with reassignment as discussed in Section 4.

The static assignment method is simpler and faster (which is good because flow assignment is serialized); however, dynamic assignment does a better job balancing the flows, and the reduction in head-of-line blocking compensates for the cost of using the hash table, particularly as more processors are added. The addition of dynamic reassignment provides further improvements for all combinations with 4 or more processors. Reassignment enforces balance and prevents head-of-line blocking, which compensates for the extra stream reassembly overhead; improvements varied from 2-16%, with an overall average of 7%.

Hardware String Matching. One of the main features of the design is the use of a hardware assist for multi-string matching. According to Amdahl's Law, the benefit of accelerating string matching is determined primarily by the fraction of time spent in software string matching. Table 4.4 shows the actual speedups for each ruleset and trace combination when comparing hardware-assisted string matching with Snort's baseline software string matcher. As discussed in Section 3, the hardware string matcher uses only a single rule table, while the software matcher uses separate rule sets

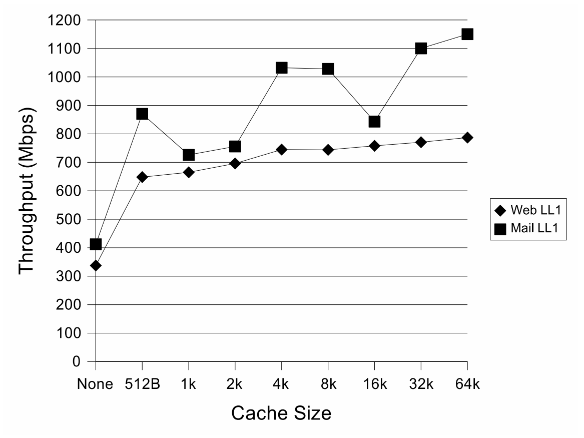


Fig. 4.6. Effect of shared cache size on 4-processor throughput

per target port; consequently, the hardware string matcher sees more false positives for content checks on ports other than the actual target. Nevertheless, hardware string maching provides benefits of 1.4–2.8x for these tests. The mail ruleset benefits much more from the hardware matcher because, as previously discussed, the mail ruleset requires less time in the verification stage for our traces than the web ruleset. Thus, a larger fraction of time is spent in the multi-string matching phase, allowing a larger benefit by accelerating it. A common practice to improve IDS performance is to eliminate unneeded rules, or those that often trigger expensive verification such as regular expressions. Doing so would serve to increase the benefit of the hardware string matcher even more.

Caches. Processor cache sizes were evaluated using the static queue assignment method. Willmann et al. showed that the simple NIC firmware does not benefit much from data caches [35]; however, the same is not true of intrusion detection. Figure 4.6 shows the impact of cache size on throughput for LL1 trace. The working set of the firmware is small; even a 512 byte cache is about twice as fast as a system with no cache, and the largest cache is only 20-30% faster than the smallest. Important data in the DRAM includes the the stream reassembly tree (which is shared for static assignment) and flow assignment table, which are accessed frequently, as well as the packet data itself; the payload is only accessed once but the IP and TCP headers are reused during flow assignment and stream reassembly. The 32-byte cache blocks also help to exploit spatial locality, reducing DRAM row activations by almost 40% with 1 kB of cache. Since the mail ruleset spends less time in verification than the web ruleset, it spends a relatively greater fraction in stream reassembly and thus depends more on DRAM access for its performance. Consequently, it benefits more from the presence of and increased size of the cache.

Private per-processor caches were also tested. For most tests however, the shared cache performed 10-30% faster for the same total amount of cache, despite the fact that the private cache had single-cycle access. The important shared data structures are frequently accessed and updated, and the private cache must often be flushed before accessing them to maintain coherence. By contrast, sharing the data cache gives the benefits of inter-processor prefetching for these structures, and processors using the shared cache need only flush specific lines when reading frame data written by the assists.

The addition of flow reassignment requires that the stream reassembly tree be made private per processor. This means that although the overall number of packets and packet descriptor structures in the system is the same, the number of stream descriptors will be larger when reassignment occurs because one stream can be in multiple reassembly trees at once. This has two potential effects. First, it may increase the demand for cache because of the duplication of stream descriptors. Second, it may reduce the fraction of shared data accessed by the processors, potentially benefitting configurations with private caches, reducing or even reversing the performance gap between shared and private cache configurations.

Summary and Discussion. Performance of any intrusion detection system is heavily dependent on the ruleset and workload. For our traces the mail ruleset requires less time in the verification phase, and is thus more dependent on bandwidth and contention for the cache and DRAM. The web ruleset requires more verification time and so runs slower overall, but benefits more from increased number and frequency of CPUs. Load balancing and assignment of flows to processors for stream reassembly is also of primary importance for our parallelized approach. Dynamic assignment, though less predictable and more variable than static assignment, performs better, particularly as the number of processors increases. 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 provides only limited benefit. In addition, a shared cache configuration performed better than private caches since coherence-related flushes were not needed for inter-processor sharing. Outside of these considerations, several paths can be taken to acheive near-Gigabit speeds; for simpler rulesets, such as the mail ruleset, 4 processors can be used at 400MHz, 6 at 300MHz, or 8 at 200MHz, depending on costs and power budgets. Likewise, a workload like the more demanding web ruleset would require 6 processors at 400MHz or 8 at 300MHz. Scaling the architecture beyond 8 processors would probably require improvements to the cache; a banked architecture for the shared cache, or private caches with hardware coherence could potentially address any contention problems. Alternatively, adding hardware assistance for stream reassembly (such as "gather" support at the string matcher) could substantially reduce the overall workload by eliminating copy overheads.

5. RELATED WORK

Clustered intrusion-detection systems are among the most popularly deployed approaches to high-performance intrusion detection because they exploit multiple low-cost, identically-configured and administered PCs along with a load-balancing switch. Schaelicke et al. have proposed SPANIDS, a system that combines a specially-designed FPGA-based load-balancing switch that considers flow information and system load when redirecting packets to commodity PCs that run intrusion detection software [4]. Commercial offerings by F5 and Radware use the companies' L4–7 load-balancing switches to redirect traffic to a pool of intrusion-detection nodes, allowing high overall throughput scalability [2, 3]. By exploiting current architectural trends toward low-cost multicore and multiprocessor PCs, the parallelized Snort achieves good performance for small to mid-scale deployments without the expense of a load-balancing switch. For larger deployments, the parallel Snort versions presented here could be used in a clustered IDS with greater per-system performance and higher space-efficiency.

The research community has also proposed distributed NIDS, in which nodes at various points in the network track anomalies and collaboratively collect data that may indicate a system-level intrusion even if no specific host triggers an alert [36,37]. For example, Snapp et al. point out that an individual host experiencing a few failed logins may be normal, but a pattern of failed logins across a domain may indicate an intrusion attempt [36]. Collaborative information sharing may thus improve the overall rate of detecting intrusions. However, efforts in distributed NIDS have invariably targeted gathering additional information to *identify* intrusions, rather than processing packets at a faster rate. Thus, distributed NIDS is largely orthogonal of the parallel processing approach.

Other intrusion detection software systems also exist, such as the Bro IDS from Lawrence Berkeley National Labs [38]. Bro rules can detect all standard Snort traffic signatures as well as anomalies such as an excessive number of connections. This work uses a Snort-based system primarily because of its popularity and greater update frequency. Despite the differences among systems, the problem of and need for fine-grained parallelism applies to all NIDS software, and the fundamental challenges and solutions discussed here apply to any system that employs stream reassembly and flow tracking to provide stateful ruleset processing.

Because matching multiple simultaneous strings is such an important component of intrusion detection and because it can potentially exploit extensive hardware concurrency, several works have proposed hardware support for this stage. Several works use reconfigurable FPGAs since the hardware can simply be resynthesized on ruleset updates. Moscola et al. match not only exact strings but also regular expressions, exploiting packet-level parallelism across independent scanning engines [11]. Sourdis and Pnevmatikatos have used independent comparison pipelines in an FPGA to perform exact string matching at a rate of multiple characters per clock cycle [12]. Baker and Prasanna have given an FPGA-based synthesis algorithm that optimizes the set of characters to those that actually exist in matching patterns and then uses several independent matching pipelines for individual bits of those characters [9]. Instead of depending on the reconfigurability of FPGAs for ruleset updates, several works use SRAM for storage of string tables. Aldwairi et al. presented a network processor architecture with string-matching accelerators based on simple FSMs [20]. Tan and Sherwood use a compact SRAM-based representation of the string table along with a special-purpose hardware engine that processes the data by exploiting parallelism on a bit-level granularity [13]. Brodie et al. developed a pipelined FSM representation that allows high-speed matching of regular expressions [10], and can be implemented in an FPGA or an ASIC.

Though there has been much work that can effectively accelerate the multi-string and regular expression matching tasks, it is not sufficient. Figure 2.2 shows an average of 35% of processing time for multi-string and 15% for regular expression matching, for a combined total of about 50% for the LL traces. According to Amdahl's Law, even if it were made infinitely fast, only a factor of 1.5 overall speedup could be obtained with accelerated string-matching, or a factor of 2.0 if regular expressions were accelerated as well. Clearly, though these components are the most important, they cannot be treated in isolation from the rest of the system, and an IDS is not complete if it has only these components. The software approach to parallelizing the various stages of intrusion detection should work synergistically with hardware that speeds up string matching and regular expression matching, increasing its effectiveness.

There has also been some investigation on running intrusion detection software on network processors: Clark et al. used the multithreaded microengines of an Intel IXP1200-based platform to reassemble incoming TCP streams that were then fed to an FPGA-based string content matcher [39]. Bos and Huang implement a rudimentary IDS which uses the IXP architecture and its parallel microengine processor cores to perform Aho-Corasick string matching, stream reconstruction, and I/O operations [40]. Although neither of these NPU-based works include the HTTP preprocessor, they could most probably add it with some performance degradation. Although NPUs can move network data at high rates, their multithreaded latency tolerance features are targeted toward the memory latencies seen in routers rather than the much higher DMA latencies seen when NICs must transfer data to and from the host. Consequently, previous work on using NPUs in NICs has seen performance imbalances related to the cost of DMA transfers [41]. Vermeiren et al. propose several strategies for a multithreaded Snort with the aim of running it on high-end network microprocessors [42] such as the Broadcom BCM1250 [43]. That work does not discuss any solutions for maintaining dependences across the stages of Snort processing or for insuring that packets from the same flow are processed in an appropriate order.

In their work proposing self-securing network interfaces, Ganger et al. suggest integrating the secure network interfaces into switch ports rather than keeping them in individual machines [5]. This strategy aims to protect against physical intruders swapping out the NICs with standard ones. Although LineSnort's design includes host-specific elements such as DMA, it would not need to change substantially to support switch integration.

6. CONCLUSIONS

Chapter 3 presents and evaluates two conservative and one optimistic parallelization strategies for network intrusion detection. Although this thesis specifically targets Snort, the challenges and solutions described here apply to any NIDS that performs stream reassembly and flow-tracking. Both parallelization schemes have their limitations, but both perform quite well for most of the workloads that they target. The most conservative flow–concurrent parallelization achieves substantial speedups on 3 of the 5 network packet traces studied, ranging as high as 4.1 speedup on 8 processor cores and processing at speeds up to 3.1 Gb/s. The extra overheads in the optimistic parallelization degrade performance by about 26% for the traces that exhibit flow concurrency, limiting speedup to 3.0 on six cores. However, the potential for intra-flow parallelism enabled by the optimistic approach allows one additional trace to see good speedup (2.4 on five cores), with a peak traffic rate over 2.6 Gb/s. The most conservative scheme achieves an average traffic rate of 2.03 Gb/s for the 5 traces, and both reassignment-based schemes average over 1.7 Gb/s, providing major improvements in intrusion detection performance using hardware that is cost-effective, space-efficient, and increasingly being commoditized.

Chapter 4 presents the architecture and software design of LineSnort, a programmable network interface card (NIC) that offloads the Snort network intrusion detection system (NIDS) from the host CPU of a high-end PC-based network server. The paper investigates and analyzes design alternatives and workloads that impact the performance of LineSnort, including the Snort rulesets used and the number and frequency of processor cores.

Leveraging previous work in programmable NICs and hardware content matching, LineSnort protects a single host from both LAN-based and Internet-based attacks, unlike edge-based NIDS which only guards against the latter. LineSnort exploits

TCP session-level parallelism using several lightweight processor cores and a dynamic assignment of TCP flows to cores. LineSnort also exploits intra-session concurrency through flow reassignment, providing better load balance and higher throughput as the number of cores increases, or for workloads with poor flow concurrency. Simulation results using the Spinach toolkit and Liberty Simulation Environment show that LineSnort can achieve Gigabit Ethernet network throughputs while supporting all standard Snort rule features, reassembling TCP streams, and transforming HTTP URLs. To achieve these throughput levels, LineSnort requires a small shared cache, a string-matching assist in the hardware, and one of several options for the number and frequency of processors. Lightweight rulesets can achieve Gigabit throughput with 4 CPU cores at 400MHz, 6 at 300MHz, or 8 at 200MHz, while a more demanding ruleset requires 6 CPU cores at 400MHz or 8 at 300MHz.

Substantial prior work has considered the use of network interface cards as a resource for optimizing the flow of communication in a system, with targets ranging from simple checksumming to full protocol offload and customized services [21–25,30]. LineSnort helps to deliver on the promise of programmable network interfaces by demonstrating that efficiently offloading the challenging problem of intrusion detection enables the new service of per-machine NIDS for network servers, providing a higher level of protection against both Internet-based and LAN-based attacks.

LIST OF REFERENCES

LIST OF REFERENCES

[1] M. Roesch, "Snort – Lightweight Intrusion Detection for Networks," in *Proceedings of the 13th USENIX Conference on System Administration*, pp. 229–238, 1999.

[2] F5 Networks, "Securing the Enterprise Perimeter – Using F5's BIG-IP System to Provide Comprehensive Application and Network Security." White paper, Oct. 2004.

[3] Radware Inc., "FireProof Security Activation." White paper, Sept. 2004.

[4] L. Schaelicke, K. Wheeler, and C. Freeland, "SPANIDS: A Scalable Network Intrusion Detection Loadbalancer," in *Proceedings of the 2nd Conference on Computing Frontiers*, pp. 315–322, 2005.

[5] G. R. Ganger, G. Economou, and S. M. Bielski, "Finding and Containing Enemies Within the Walls with Self-securing Network Interfaces," Tech. Rep. CMU-CS-03-109, Carnegie Mellon School of Computer Science, Jan. 2003.

[6] D. L. Schuff and V. S. Pai, "Design alternatives for a high-performance self-seucring ethernet network interface," in *Proceedings. 21st International Parallel and Distributed Processing Symposium*, (Long Beach, CA), March 2007.

[7] Alteon Networks, *Tigon/PCI Ethernet Controller*, Aug. 1997. Revision 1.04.

[8] P. Willmann, H. Kim, S. Rixner, and V. S. Pai, "An Efficient Programmable 10 Gigabit Ethernet Network Interface Card," in *Proc. of the 11th International Symposium on High-Performance Computer Architecture*, pp. 96–107, Feb. 2005.

[9] Z. K. Baker and V. K. Prasanna, "A Methodology for the Synthesis of Efficient Intrusion Detection Systems on FPGAs," in *Proceedings of the Twelfth Annual IEEE Symposium on Field Programmable Custom Computing Machines 2004 (FCCM '04)*, Apr. 2004.

[10] B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A scalable architecture for high-throughput regular-expression pattern matching," in *ISCA '06: Proceedings of the 33rd International Symposium on Computer Architecture*, (Washington, DC, USA), pp. 191–202, IEEE Computer Society, 2006.

[11] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos, "Implementation of a Content-Scanning Module for an Internet Firewall," in *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 31–38, Apr. 2003.

[12] I. Sourdis and D. Pnevmatikatos, "Fast, Large-Scale String Match for a 10Gbps FPGA-based Network Intrusion Detection System," in *Proceedings of the 13th International Conference on Field Programmable Logic and Applications (FPL'03)*, pp. 880–889, Sept. 2003.

[13] L. Tan and T. Sherwood, "A High Throughput String Matching Architecture for Intrusion Detection and Prevention," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pp. 112–122, June 2005.

[14] F. Yu, R. H. Katz, and T. V. Lakshman, "Gigabit Rate Packet Pattern-Matching Using TCAM," in *Proc. of the 12th IEEE International Conference on Network Protocols*, pp. 174–183, Oct. 2004.

[15] A. V. Aho and M. J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Commun. ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[16] S. Wu and U. Manber, "A fast algorithm for multi-pattern searching," Tech. Rep. TR-94-17, Department of Computer Science, University of Arizona, 1994.

[17] R. S. Boyer and J. S. Moore, "A Fast String Search Algorithm," *Communications of the ACM*, vol. 20, pp. 762–772, Oct. 1977.

[18] R. N. Horspool, "Practical Fast Searching in Strings," *Software: Practice and Experience*, vol. 10, no. 6, pp. 501–506, 1980.

[19] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax." IETF RFC 2396, Aug. 1998.

[20] M. Aldwairi, T. Conte, and P. Franzon, "Configurable string matching hardware for speeding up intrusion detection," *SIGARCH Comput. Archit. News*, vol. 33, no. 1, pp. 99–107, 2005.

[21] Adaptec, "ANA-7711 Network Accelerator Card Specification," Mar. 2002.

[22] Y. Hoskote, B. A. Bloechel, G. E. Dermer, V. Erraguntla, D. Finan, J. Howard, D. Klowden, S. G. Narendra, G. Ruhl, J. W. Tschanz, S. Vangal, V. Veeramachaneni, H. Wilson, J. Xu, and N. Borkar, "A TCP Offload Accelerator for 10 Gb/s Ethernet in 90-nm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 38, pp. 1866–1875, Nov. 2003.

[23] H. Kim, V. S. Pai, and S. Rixner, "Improving Web Server Throughput with Network Interface Data Caching," in *Proc. of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 239–250, Oct. 2002.

[24] K. Z. Meth and J. Satran, "Design of the iSCSI Protocol," in *Proceedings of the 20th IEEE Conference on Mass Storage Systems and Technologies*, Apr. 2003.

[25] P. Shivam, P. Wyckoff, and D. Panda, "EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing," in *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (SC2001)*, Nov. 2001.

[26] DARPA Internet Program Protocol Specification, "Transmission Control Protocol." IETF RFC 793, Sept. 1981.

[27] J. W. Haines, R. P. Lippmann, D. J. Fried, E. Tran, S. Boswell, and M. A. Zissman, "1999 DARPA Intrusion Detection System Evaluation: Design and Procedures," Tech. Rep. 1062, MIT Lincoln Laboratory, 2001.

[28] Shmoo Group, "Defcon 9 Capture the Flag Data," Sept. 2001.

[29] H. Kim, V. S. Pai, and S. Rixner, "Exploiting Task-Level Concurrency in a Programmable Network Interface," in *Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2003.

[30] K. Kleinpaste, P. Steenkiste, and B. Zill, "Software Support for Outboard Buffering and Checksumming," in *Proceedings of the ACM SIGCOMM '95 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pp. 87–98, Aug. 1995.

[31] Micron, "256Mb: x32 GDDR3 SDRAM MT44H8M32 data sheet," June 2003. Available from `www.digchip.com`.

[32] P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory," in *Proc. of the 19th Annual International Symposium on Computer Architecture*, pp. 13–21, May 1992.

[33] P. Willmann, M. Brogioli, and V. S. Pai, "Spinach: A Liberty-Based Simulator for Programmable Network Interface Architectures," in *Proc. of the ACM SIGPLAN/SIGBED 2004 Conference on Languages, Compilers, and Tools for Embedded Systems*, pp. 20–29, June 2004.

[34] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August, "Microarchitectural Exploration with Liberty," in *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pp. 271–282, Nov. 2002.

[35] G. S. Johnson, J. Lee, C. A. Burns, and W. R. Mark, "The irregular z-buffer: Hardware acceleration for irregular data structures," *ACM Transactions on Graphics*, vol. 24, no. 4, pp. 1462–1482, 2005.

[36] S. R. Snapp, J. Brentano, G. V. Dias, T. L. Goan, L. T. Heberlein, C. lin Ho, K. N. Levitt, B. Mukherjee, S. E. Smaha, T. Grance, D. M. Teal, and D. Mansur, "DIDS (Distributed Intrusion Detection System) - Motivation, Architecture, and An Early Prototype," in *Proceedings of the 14th National Computer Security Conference*, (Washington, DC), pp. 167–176, Oct. 1991.

[37] R. Gopalakrishna and E. H. Spafford, "A Framework for Distributed Intrusion Detection using Interest Driven Cooperating Agents," in *Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection (RAID 2001)*, Oct. 2001.

[38] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time," *Computer Networks*, vol. 31, pp. 2435–2463, Dec. 1999.

[39] C. Clark, W. Lee, D. Schimmel, D. Contis, M. Koné, and A. Thomas, "A Hardware Platform for Network Intrusion Detection and Prevention," in *Proc. of the Third Workshop on Network Processors and Applications*, Feb. 2004.

[40] H. Bos and K. Huang, "Towards software-based signature detection for intrusion prevention on the network card," *Recent Advances in Intrusion Detection. 8th International Symposium, RAID 2005. Revised Papers (Lecture Notes in Computer Science Vol. 3858)*, pp. 102 – 23, 2005.

[41] K. Mackenzie, W. Shi, A. McDonald, and I. Ganev, "An Intel IXP1200-based Network Interface," in *Proceedings of the 2003 Annual Workshop on Novel Uses of Systems Area Networks (SAN-2)*, Feb. 2003.

[42] T. Vermeiren, E. Borghs, and B. Haagdorens, "Evaluation of software techniques for parallel packet processing on multi-core processors," *IEEE Consumer Communications and Networking Conference, CCNC*, pp. 645 – 647, 2004.

[43] Broadcom, *BCM1250 Product Brief*, 2006.