Packet Scheduling for Deep Packet Inspection on Multi-Core Architectures

Terry Nelms
Georgia Institute of Technology
tnelms@gatech.edu

Mustaque Ahamad
Georgia Institute of Technology
mustaq@cc.gatech.edu

ABSTRACT

Multi-core architectures are commonly used for network applications because the workload is highly parallelizable. Packet scheduling is a critical performance component of these applications and significantly impacts how well they scale. Deep packet inspection (DPI) applications are more complex than most network applications. This makes packet scheduling more difficult, but it can have a larger impact on performance. Also, packet latency and ordering requirements differ depending on whether the DPI application is deployed inline. Therefore, different packet scheduling tradeoffs can be made based on the deployment.

In this paper, we evaluate three packet scheduling algorithms with the Protocol Analysis Module (PAM) as our DPI application using network traces acquired from production networks where intrusion prevention systems (IPS) are deployed. One of the packet scheduling algorithms we evaluate is commonly used in production applications; thus, it is useful for comparison. The other two are of our own design. Our results show that packet scheduling based on cache affinity is more important than trying to balance packets. More specifically, for the three network traces we tested, our cache affinity packet scheduler outperformed the other two schedulers increasing throughput by as much as 38%.

Categories and Subject Descriptors

C.1.2 [**Multiple Data Stream Architectures (Multiprocessors)**]: *Multiple-instruction-stream, multiple-data-stream processors (MIMD);* C.2.0 [**Computer Communication Networks**]: General *–Security and protection (e.g., firewalls)*

General Terms

Design, Measurement, Performance, Security

Keywords

Packet Scheduling, Deep Packet Inspection (DPI), Protocol Analysis Module (PAM), Multi-Core.

1. INTRODUCTION

Deep packet inspection (DPI) is the process of examining the non-header content of a packet by a system that is not an endpoint in the communication. Most DPI systems reconstruct communication streams and maintain state information for large numbers of concurrent connections. When a packet arrives each layer is fully parsed and inspected. State information associated with the communication stream at each layer is updated and stored. All of this work requires many CPU cycles. Fortunately, the work is highly parallelizable; therefore, multi-core architectures can be used to increase the available CPU cycles. So, that raises the question of how packets should be scheduled on a multi-core platform. In this paper we explore this question using the Protocol Analysis Module (PAM) as our DPI system.

PAM [7, 17, 19, 29] is a software library that uses DPI to detect and respond to malicious network traffic. It is a proprietary module used by IBM Internet Security Systems in all products that implement intrusion detection and prevention. PAM fully parses all protocol layers and emulates the state transitions of both the client and server at each layer. PAM detects when a client or server is in a vulnerable state and watches for any attempts to exploit it.

PAM is constantly updated. New protocol parsers are added and old ones are updated so they can detect the latest threat. This continuous addition of new code causes the average number of instructions needed to inspect a packet to slowly increase with each security update. In addition, network speeds continue to increase. Today, 1 Gigabit Ethernet is the standard with 10 Gigabit Ethernet becoming commonplace in the data center. In the next few years, with the 802.3ba standard scheduled to soon be ratified, 40 and 100 Gigabit Ethernet will begin to show up in the data center [6]. This continuous increase in network speeds and average per packet instruction counts results in PAM requiring more and more CPU cycles to inspect traffic at line rate.

In the past PAM relied on Moore's Law to satisfy its demand for more CPU cycles. When a processor manufacturer released a new processor with a higher operating frequency, the performance of PAM increased without any code changes. This worked well for many years because processor manufacturers were able to hide the latency and bandwidth bottlenecks of the other components through caching and instruction level parallelism (ILP). Unfortunately, due to heat and power constraints, this is no longer the case and processor manufacturers have turned to exploiting thread level parallelism (TLP) in order to continue delivering regular performance improvements (e.g., multi-core processors and hardware threads).

PAM is currently being transformed from a single threaded application into a multi-threaded application so it can take advantage of TLP. A PAM thread can process the link, internet, and transport layers of any packet in parallel with any other packet. However, at the application layer not all packets can be processed in parallel. For TCP based protocols, packets on the

ANCS'10, October 25–26, 2010, La Jolla, CA, USA.

same flow must be processed serially. In fact, TCP segments on a flow must be processed in sequence order because of the way PAM emulates the state transitions of client and server applications. A small number of UDP based protocols must be processed serially as well and in application layer sequence order because they implement sessions at the application layer. However, the majority of UDP packets and other protocols can be processed in parallel.

Our contributions in this paper are summarized as follows:

- Designed and implemented two new DPI packet scheduling algorithms. One is designed to maximize work balance and the other cache affinity.
- Compared our two packet scheduling algorithms against a packet scheduling algorithm commonly used in network applications.
- Demonstrated that scheduling packets for cache affinity is more important than balancing the work load. In fact, for one of the network captures we used in our evaluation, scheduling packets for cache affinity improved throughput by 38%.

In the rest of this paper, we explore packet scheduling for DPI with the goal of maximizing throughput and minimizing latency. This paper is organized as follows. The next section presents related work. In section 3, we describe the packet scheduling algorithms we designed, implemented, and evaluated. In section 4, we explain our testing methodology and present the results. In section 5, the paper is concluded and future work is discussed.

2. RELATED WORK

Packet distribution, load balancing, and scheduling are the terms typically used for the process of assigning a packet to a resource that will perform work on it (in this paper these terms are used interchangeably). Packet scheduling occurs at many places on the network. Packets are scheduled on physical links for multi-path routing [16], physical links for link aggregation [11], distributed network processors for high-speed links [5], forwarding engines for parallel forwarding [23-25], transactions for cluster-based Internet services [14], and L7-filter threads for QoS [1, 9, 10]. Although the work being scheduled varies widely, many of the concepts and techniques can be applied across the packet scheduling domain.

2.1 Packet Scheduling Overview

2.1.1 Packet Based Scheduling

In packet based scheduling, packets are assigned to resources on a packet-by-packet basis. Round-robin is an example of a simple packet based scheduling algorithm. More complex algorithms assign packets to the least loaded resource (e.g., the resource with the smallest number of outstanding bytes to process). Packet based scheduling does a good job of balancing the load across resources. However, packet reordering can occur and impact network performance. In addition, packet based scheduling in shared memory systems can be cache inefficient resulting in reduced performance. Packet based scheduling algorithms are commonly found as a scheduling option in network devices and processors.

2.1.2 Flow Based Scheduling

Flow based scheduling [10] maintains a table of active flows where each flow is associated with a resource. Packets are mapped to flows and scheduled on the resource associated with their flow. New flows are assigned to the least loaded resource. Since all packets on the same flow are assigned to the same resource, per flow packet order is maintained. Also, it is cache efficient for the same reason. There are a few drawbacks to this method. First, a table of active flows must be maintained. This table can require a large amount of memory when there are many active flows. In addition, it can take a significant number of clock cycles to perform the lookup. Finally, flows are not equal in the number of packets, bytes, and processing associated with them. Thus, it is difficult to assign new flows to resources so that the work is balanced.

2.1.3 Fixed Hash Scheduling

Direct hash and indirect hash [4] are the two most common fixed hash scheduling algorithms. Direct hash applies a hash function to a subset of the 5-tuple and uses the result modulo the number of resources to determine the resource assignment. Indirect hash uses the result of the hash function modulo the size of the indirection table as an index into it. Each bin in the indirection table is associated with a resource and packets that map to a bin are processed by that resource. Indirect hash allows for unequal resource weights and the indirection table can be tuned for adaptive hash scheduling. For fixed hash scheduling, packets with the same 5-tuple hash to the same value; so, they are assigned to the same resource. Therefore, per flow packet ordering is maintained and it is cache efficient. In addition, it is stateless since there is no table to maintain. The downside to this method is lack of control over resource assignment and this can result in load imbalances. Fixed hash scheduling algorithms are commonly found as a scheduling option in network devices and processors.

2.1.4 Adaptive Hash Scheduling

Adaptive hash scheduling [9, 16, 20, 25] attempts to combine the simplicity of fixed hash scheduling with the ability to change the resource assignment when the load is imbalanced. Receive-side scaling (RSS) [20] is an adaptive hash scheduling algorithm used to balance packets arriving on a network adapter to CPUs. RSS uses indirect hashing to schedule packets. When a load imbalance is detected, the host protocol stack tries to balance the traffic by calculating a new indirection table. The authors of [16] describe and evaluate several adaptive indirect hashing algorithms for multi-path routing. Adaptive hash scheduling can improve fixed hash scheduling load imbalances. However, it adds overhead and the adjustments are reactive typically occurring after the network has been impacted.

2.1.5 Flow Burst Scheduling

Flow burst scheduling [5, 15, 24] tries to combine the workload balance of packet based scheduling with the cache affinity and per flow packet ordering of flow based scheduling. As in flow based scheduling, a flow table is maintained; however, it only needs to contain flows that have packets in the system. A flow entry contains the number of packets currently in the system or a timestamp of the last packet that mapped to that flow. When a packet arrives it is mapped to a flow entry using a subset of the 5-tuple. If flow entry exists and there are packets in the system on that flow, the packet is assigned to the resource processing the other packets. On the other hand, if the flow entry does not exist or there are no packets in the system on that flow, the packet is assigned to the least loaded resource. Since all packets on the same flow in the system at the same time are processed by the same resource, it is cache efficient and per-flow packet ordering is maintained. Also, the balance of the work is better than flow based scheduling because flows are not fixed to a resource and

can be reassigned between packet bursts. However, maintaining the flow entries can be expensive and imbalances can still occur.

2.2 Packet Scheduling For DPI

Most of the packet schedulers in the literature are designed for applications that are not as complex as DPI. The information used to make scheduling decisions for those applications typically does not apply to DPI. For instance, the number of packets or bytes enqueued to a thread. The applications in the literature essentially have a fixed processing time per packet or byte. In contrast, the number of instructions PAM needs to process a packet or byte varies dramatically depending on the protocol and its attributes. For example, a packet with an application layer protocol that PAM does not recognize may require 600 clock cycles to classify; whereas, an identically sized HTTP packet with compressed content may require 30,000 clock cycles to inspect. Thus, it is difficult to determine the amount of work assigned to a PAM thread when making a scheduling decision.

Maintaining per-flow packet order on egress is an important feature of most packet schedulers. This is because they were designed for inline devices and packet reordering within a flow can have a negative impact on the performance of the network. This is due to the fact that reordering packets within a TCP flow can result in duplicate data segment transmissions, a reduced data transmission rate, and burstiness [2, 3]. Requiring per flow packet order on egress does not always mean that packets cannot be processed out-of-order. There are systems that restore per flow packet order on egress [8]. Also, per flow packet ordering at egress is only important for systems that are inline and DPI systems are not always deployed inline. However, inline or not, PAM requires that TCP segments be processed in sequence order. If PAM receives a TCP segment out-of-order it saves a copy until the missing segments arrive. This can impact performance when a large number of TCP segments are received out-of-order. So, maintaining per flow packet ordering for packet processing can improve PAM's performance by reducing the number of TCP segment that must be saved.

As for the multithreaded state of other DPI systems, Snort 3.0 [22, 26] is currently available in beta for download. It supports a multithreaded execution model that allows multiple analysis engines to operate on the same traffic simultaneously. Therefore, unlike PAM it does not utilize packet and connection level parallelism. In the case of Bro [18], a multithreaded version is not currently available. However, in [27] the authors describe an architecture that could be used to create a multithreaded Bro. The architecture uses a flow based scheduling algorithm to distribute packets for analysis. Thus, an improved packet scheduling algorithm could significantly increase performance.

When deployed inline, the goals of a DPI packet scheduler are to maximize throughput, minimize average latency, bound maximum latency (e.g., one millisecond), and minimize the number of packets that are reordered within a flow. When a DPI system is not inline, most of those goals are no longer important. In fact, the goals become maximize throughput and bound maximum latency at a much higher value (e.g., one second).

3. DPI PACKET SCHEDULERS

The goal of packet scheduling for DPI is to maximize the amount of network traffic that can be inspected without noticeably impacting it. The ideal packet scheduler has the following properties:

- Load Balancing: Work is evenly distributed across all threads.
- Low Scheduling Overhead: The cost of scheduling packets (in terms of memory and CPU cycles) is very small in comparison to the work performed on the packet.
- Per-Flow Ordering: Packets on the same flow at egress are in their arrival order.
- Cache Affinity: Packets are scheduled on threads that have their associated data structures already in cache.
- Minimal Packet Delay Variation: Minimal variation in the latency added to packets on the same flow.

In this section we describe the design and implementation of the three packet scheduling algorithms we evaluate with PAM. The first algorithm we describe is Direct Hash (DH). It is an algorithm that is commonly used to schedule packets and we use it in our evaluation for comparison purposes. The other two algorithms are of our own design. We created Packet Handoff (PH) to maximize load balancing. However, to maximize this property, tradeoffs were made such as per-flow ordering and cache affinity. The other algorithm we designed is Last Flow Bundle (LFB). Its goal is to maximize cache affinity. But, like PH it sacrifices other properties (e.g., packet delay variation). No single packet scheduler can have all of the properties of our ideal packet scheduler; so, our goal is to determine the properties that are most important for achieving maximum inspection with minimal network impact.

The DH and LFB packet schedulers only use the source and destination IP addresses instead of the entire 5-tuple for the flow identifier (FID). We chose not to include the transport layer ports because that would require more parsing, they are not included in fragmented packets (except for the first fragment), and it did not have a significant impact on the distribution of packets with our network captures. As for our hash function, we chose a 16-bit CRC because it is known to provide good load distribution [4]. In addition, we always hash the smallest IP address first so that packets in both directions on a connection go to the same thread.

3.1 Direct Hash (DH)

Direct Hash (DH) is a simple fixed hash scheduling algorithm that is widely used; that is why we selected it for comparison. When a packet arrives, the packet scheduler parses the data link and network layers to extract the FID. The FID is then hashed and the result modulo the number of threads determines the PAM thread that will process the packet. DH has several appealing properties:

- Stateless: There is no state to maintain. The information needed to distribute the network traffic is contained in each packet. So, there is no additional memory overhead or table lookups.
- Per-Flow Ordering: By using a subset of the 5-tuple as input to the hash function, packets on the same flow will hash to the same value so they will be processed by the same thread. Thus, per-flow packet order is maintained.
- Cache Affinity: Packets on a flow are always processed by the same thread. So only memory associated with the flows assigned to a thread will be in the processor's cache. This increases the likelihood of a cache hit when processing a packet. Furthermore, packets on a flow often arrive in bursts, referred to as packet trains in [13], resulting in temporal locality that can be used to improve the cache hit rate by assigning the packet train to the same processor.

The drawbacks of using DH are:

- Load Imbalance: There is no control over how packets are distributed. The authors of [23] prove that a direct hash on FIDs cannot balance the workload due to the Zipf-like [30] probability distribution of flows found in real-world network traffic.
- Header Parsing: the packet scheduler must understand how to parse all of the protocol headers that contain FID fields. In addition, it must be able to skip over lower level headers to reach FID headers. Network traffic, even at the lower levels, can be complicated (e.g., contain multiprotocol label switching (MPLS), virtual LAN (VLAN), stacked VLANs, IPv6 extension headers). Therefore, the packet scheduler must be complex enough to extract the FID fields on these networks.
- High Distribution Overhead: The header parsing, hashing, and distribution are in the data plane and executed on every packet. So it is important for them to be efficient. However, esoteric network traffic and the complexity of a good hash function can make the distribution of packets the bottleneck.

3.2 Packet Handoff (PH)

Packet Handoff (PH) is a packet scheduling algorithm we designed with the goal of maximizing the concurrency available in DPI. It is a packet based scheduling algorithm, where the least loaded thread gets the next packet. Therefore, this algorithm is the best at distributing the work evenly across threads. Two types of queues are used to distribute packets. When a packet is received the packet scheduler places the packet in the receive queue (RQ). Threads that are not busy (i.e. not currently processing a packet) go to the RQ to get their next packet. If the RQ is empty they wait (i.e. spin) for a packet to arrive.

After a thread dequeues a packet from the RQ it parses the link, internet, and transport layers of the packet. If the packet's transport protocol is TCP the 5-tuple is extracted by the PAM thread and used as the FID. PAM tracks the transport and application layer state associated with a TCP connection in a connection entry (CE). All CEs are stored in a connection table and the FID is used to map a packet to a CE in the table. The connection table is shared by all threads. If a CE mapping is not found, a new entry is created and the thread sets itself as the owner (this is done by setting a flag in the CE). If the thread finds the CE, it checks to see if another thread owns it. If it is not owned, the thread sets itself as the owner. If the CE is owned, this indicates that another PAM thread is currently processing a packet associated with it. Packet on the same TCP flow must be processed in sequence order. Therefore, instead of waiting for the owning thread to complete its processing and release ownership, the thread enqueues the packet in the connection queue (CQ) associated with the CE and returns to the RQ for its next packet.

Once a thread owns a CE, it processes the application layer of its current packet and updates the CE as needed (no synchronization required). When the thread is finished processing its current packet it checks to see if any packets have been placed in the CQ associated with the CE it owns. If no packets are in the CQ it removes itself as the owner of the CE (by setting a flag in the CE) and proceeds to the RQ to for its next packet. If there is a packet in the CQ, the thread retains CE ownership and dequeues its next packet from the CQ instead of the RQ.

A PAM thread can only own one CE at a time. CE ownership occurs when a packet maps to a CE that is not currently owned. A PAM thread releases CE ownership when it finishes processing its current packet and the CQ associated with the CE is empty. Any PAM thread can enqueue a packet in the CQ associated with a CE, but only the current owner of the CE can dequeue packets. Packets that are enqueued to a CQ receive the cache benefits of having their application layer processed by the same thread. However, when CE ownership is released there may be packets in the system (either being processed or in the RQ) that map to the CE. There is a high probability that those packets will be processed by a different thread and interleaved with packets on other connections; thus, they will not receive cache affinity benefits.

The advantages of PH are:

- Load Balancing: Threads pull packets from a queue (i.e. from the RQ) when they are not busy. The link, internet, and transport layer of all packets are processed in parallel. Packets are only enqueued to a PAM thread when it owns a CE and there are other packets being processed by other threads that map to the owned CE. If n PAM threads are processing packets and there are at least n packets in the receive queue that map to different CEs, no threads will be idle.
- Low Distribution Overhead: The packet scheduler does not need to parse packet headers or compute a hash. All it does is enqueue packets into the RQ. Therefore, the scheduling overhead is low.

The disadvantages of PH are:

- Out-of-Order Packets: Per-flow packet ordering is not preserved. If two packets on the same TCP flow are processed at the same time by different threads, there is a race to acquire ownership of the CE. If the thread with the higher packet number acquires ownership, then a copy of the packet will be saved. In addition, if PAM is inline the packet will be transmitted out-of-order.
- Cache Affinity: There is no mapping of packets that are part of the same flow to the same thread. Therefore, cache lines associated with CEs move from processor to processor as they are looked up and updated. In addition to the cache coherence overhead, the amount of cache lines available to store connections is reduced, increasing memory accesses.
- Queue Overhead: All threads compete to dequeue packets from the RQ. In our implementation, RQ contention was not an issue. This was due to the fact that our average packet size was over 300 bytes for all network captures and there were at most 14 threads pulling packets from the RQ. A smaller average packet size or more threads could cause the RQ to become the bottleneck. In addition to the RQ, CQ contention can limit performance when a large percentage of the packets are handed off to a single thread.

3.3 Last Flow Bundle (LFB)

Last Flow Bundle (LFB) is a flow burst scheduling algorithm we designed with the goal of maximizing cache affinity. It schedules packets similar to the way Last Processor (LP) schedules tasks [28]. The idea is to process all of the packets in the system that map to the same flow on a single thread and not interleave the processing of packets that map to other flows. LFB uses two types of queues, a single receive queue (RQ) and a table of flow bundle queues (FBQ), to distribute packets.

As in the DH method, when a packet arrives it is processed by the packet scheduler to extract the FID that will be used as the

input to the hash function. However, instead of using the result of the hash function to select the thread that will process the packet, it is used to map the packet to a FBQ in the FBQ table. If the selected FBQ is empty, then the packet is enqueued in it and the RQ. On the other hand, if the selected FBQ is not empty, the packet is only enqueued in it.

When the system is initialized, all PAM threads proceed to the RQ to get their first packet. After a packet is processed by a PAM thread, it is dequeued from the associated FBQ (i.e. the FBQ the packet scheduler mapped the packet to) by the PAM thread. If the FBQ is not empty after the newly processed packet is dequeued, the PAM thread selects the packet at the head of the FBQ to process next, but does not dequeue it. This process repeats until the FBQ is empty. Once the FBQ is empty the PAM thread returns to the RQ for its next packet.

By enqueuing the packet in both the RQ and the selected FBQ, when the selected FBQ is empty, the packet scheduler permits any non busy PAM thread (i.e. a PAM thread not currently associated with a FBQ) to process it. However, by enqueuing the packet only in the selected FBQ when it is not empty, the packet scheduler ensures that only the PAM thread that is currently associated with that FBQ will process it. PAM threads dequeue packets from the RQ before they process them. Contrarily, PAM threads dequeue packets from the FBQ after they have fully processed them. Leaving a packet in the FBQ during processing informs the packet scheduler that a PAM thread is currently associated with that FBQ. The FBQ association occurs when a PAM thread dequeues a packet from the RQ. A PAM thread is associated with a FBQ until it is empty. A packet is placed in the RQ only when there are no other packets in the system mapping to the same FBQ. Therefore, at any given time, there is at most one packet in the RQ that maps to a given FBQ.

The advantages of LFB are:

- Per-flow Ordering: Packets on the same flow hash to the same FBQ. One PAM thread is assigned to a FBQ until all of the packets in the system that map to it have been processed. Thus, packets on a flow cannot be reordered because they are processed serially in the order they arrived in the system.
- Cache Affinity: Packets that map to the same FBQ and are in the system at the same time are processed by a single thread. So packet trains will be processed on the same processor increasing the likelihood of a cache hit on the data structures associated with the flow. In addition, a thread will choose to process a packet that maps to the previous packet's FBQ over packets that arrived earlier, but did not map to the same FBQ. This increases the cache hit rate by eliminating interleaved flow processing that could evict cache lines associated with the previous packet's flow.
- Load Balancing: The FBQ table is over two orders of magnitude larger than the number of PAM threads. So, the number of flow collisions is small compared to DH. FBQ assignments are dynamic. A PAM thread is only assigned to a FBQ while there are packets that map to it in the system. When a thread finishes processing the last packet in the system on a FBQ, it goes to the RQ to get its next packet. If there are *n* threads processing packets and there are at least *n* packets in the system that map to unique FBQ, no thread will be idle.

The disadvantages of LFB are:

- Header Parsing: Like direct hash, the packet scheduler must be complex enough to skip over the preceding protocol layers and extract the FID.
- Increased Packet Delay Variation (PDV) [21]: Packets on the same flow are intentionally processed together for improved efficiency. However, this causes packets in a packet train to jump ahead of older packets in the queue resulting in higher latency for those packets. If enough packets jump ahead the PDV will increase.
- High Distribution Overhead: In addition to the overhead described in the DH section, LFB requires the packet scheduler to enqueue a packet in two places if it maps to an empty FBQ.

4. PERFORMANCE EVALUATION

This section describes our testing methodology, the data used for the evaluation, and the results. Our goal was to understand the impact each scheduling algorithm has on throughput and latency. Therefore, for each packet scheduling algorithm we measured the following:

- Maximum raw throughput: This measures how fast a packet scheduling algorithm can process a network capture and ignores packet timing information.
- Maximum scaled throughput: This measures how fast a packet scheduling algorithm can process a packet capture using the timing information in the capture.
- Average packet latency: The average amount of time a packet spends in the system.
- Maximum packet latency: The maximum time that any packet spent in the system.

4.1 Testing Methodology

We designed a test harness that takes a network capture as input and collects the throughput and latency measurements described above. The test harness begins by loading all of the packets from the capture into memory to eliminate disk I/O from the measurements. The packets are parsed by the packet scheduler during load to extract the FID and calculate the hash. All of the PAM threads wait on a barrier during this phase. When all of the packets are in memory they proceed through the barrier and record the value of the cycle counter. After the packet scheduler enqueues the last packet in the capture, it enqueues sentinel packets to let the PAM threads know they are done. When a PAM thread dequeues a sentinel packet it calculates the difference between the value of the current cycle counter and the recorded start value. The largest cycle count recorded by a PAM thread is then used in the time and bandwidth calculations.

4.1.1 Measuring Throughput

To measure the maximum raw throughput, the packet scheduler enqueues packets as fast as it can and only pauses when a queue is full. The largest cycle count recorded by a PAM thread is then used to calculate throughput. Measuring the maximum scaled throughput is more complicated. After the packets have been loaded into memory, the packet scheduler scans the timestamps recorded for each packet during capture to calculate the average and maximum bandwidth (we define the maximum bandwidth to be the maximum bit rate sustained for a minimum of one millisecond). Then the packet scheduler scales the timestamp of each packet so that the new average bandwidth matches a

program argument provided to the test harness. When processing begins, the packet scheduler uses the cycle counter and the adjusted timestamp of each packet to enqueue them at their new specified time. Periodically, the packet scheduler will check on the number of outstanding packets. If that number exceeds a specified maximum value it considers those packets dropped. In an actual DPI appliance, there are a fixed number of packet buffers. Once that value is exceeded, incoming packets are dropped. We consider PAM to be oversubscribed if it drops a packet; thus, unable to perform at that bandwidth. We chose 4096 as the maximum number of outstanding packets because latency starts to increase significantly when it is exceeded.

4.1.2 Measuring Latency

Packet latency is measured by recording the value of the cycle counter when a packet is enqueued and calculating the difference in the cycle counter after a PAM thread processes it. The latency associated with receiving and forwarding of packets that would be present in a DPI appliance is not measured by our test harness. However, our goal is to measure the latency relative to each scheduling algorithm so we believe our measurement to be sufficient. Our latency measurements were taken at 80% of maximum scaled throughput.

4.1.3 Evaluation Platform

We evaluated the packet scheduling algorithms on a system running a Linux 2.6 kernel with two 2.53 GHz Intel Quad-Core Xeon E5540 processors [12] with 4 gigabytes of RAM. Hyper-threading was enabled and processor affinity was set so that each PAM thread and the packet scheduler executed on different hardware threads. In addition, software threads were assigned so that no two executed on the same core unless there were more software threads than cores. Furthermore, the priority of all software threads was set to real-time.

The Xeon E5540 processors are based on Intel's Nehalem microarchitecture. They have a memory controller per processor and are connected via the Quick Path Interconnect (QPI) to create a ccNUMA architecture. Each processor has 4 cores and each core has its own L1 (32 KB instruction/32 KB data) and L2 (256 KB) cache. All of the cores on a processor share an inclusive L3 (8 MB) cache. The cache line size is 64 bytes. Cache coherency between processors is maintained by the MESIF protocol.

4.2 Network Captures

We used three network captures to evaluate the performance of the packet scheduling algorithms. They consist of real network traffic from locations in networks where a DPI appliance is deployed. They contain the entire payload of every packet so that we can determine the number of clock cycles PAM requires to process each packet. The clock cycle measurements in this section were taken by recording the clock cycle counter just before entering PAM's packet processing function and then recording the difference when PAM returned. Table 1 shows the following information for each capture:

- Packets: The total number of packets.
- Connections (Conns): The total number of unique TCP and UDP connections.
- Average Mb/s: The average bandwidth of the network traffic during capture.
- Maximum Mb/s: The maximum bandwidth of the network traffic that was sustained for a minimum of one millisecond during capture.
- Average Cycles Per-Packet (CPP): The average number of clock cycles PAM expends processing a packet.
- Connection Density (CD): The average number of unique connections per one hundred packets.

Table 1: Network Capture Attributes

Caps	Packets	Conns.	Avg. Mb/s	Max. Mb/s	Avg. CPP	CD
Dominant	454,988	2,224	13	38	4,814	7
Many	1,567,397	105,691	27	74	5,658	50
Balanced	1,236,710	43,357	57	75	5,203	29

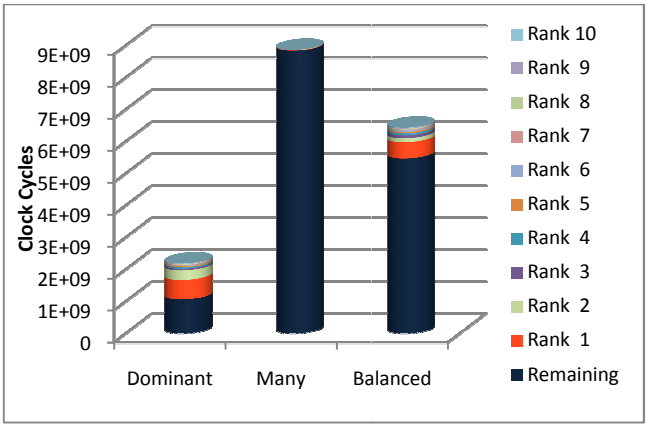

Figure 1: Top Ten Connections by Clock Cycles

We chose these three captures because we wanted to evaluate the packet scheduling algorithms under varying network conditions using real network traffic. Figure 1 shows the clock cycle distribution of the top ten connections from each capture. The top ten connections in the Dominant capture are responsible for over 50% of the total clock cycles. The traffic in the capture is dominated by a single TCP connection. It is responsible for over 50% of the packets and 27% of the clock cycles. The top ten connections in the Many capture are responsible for less than 0.4%. This means there is a much lower variance in clock cycles spent on each flow than in the other captures. In addition, packets on the different connections are spread throughout the capture having an average 50 unique connections per 100 packets. The Balanced capture's top ten connections contribute about 15% of the total clock cycles. The dominant connection in the Balanced capture is responsible for 8%. This places the Balanced capture between the other two in terms of connection clock cycle distribution.

4.3 Throughput Results

4.3.1 Dominant Capture Raw Throughput

Figure 2 shows the maximum raw throughput results for the Dominant capture. The LFB algorithm has the highest throughput at every thread count tested. It shows near linear scalability from 1 to 4 threads. After that, as more threads are added, the throughput increases cease. This is because of the dominating TCP connection that is responsible for the majority of the packets. At 5 threads and above, a single thread processes all of the packets on that connection and nothing else. So, 3.8 gigabits per second is how fast a single thread can process that connection. Adding additional threads only decreases the average latency of the packets on the other connections. At 14 threads the

performance decreases. This is due to the fact that the hardware thread on the core that inspected the dominating TCP connection shared cycles with another hardware thread that processed 11,803 packets on other connections.

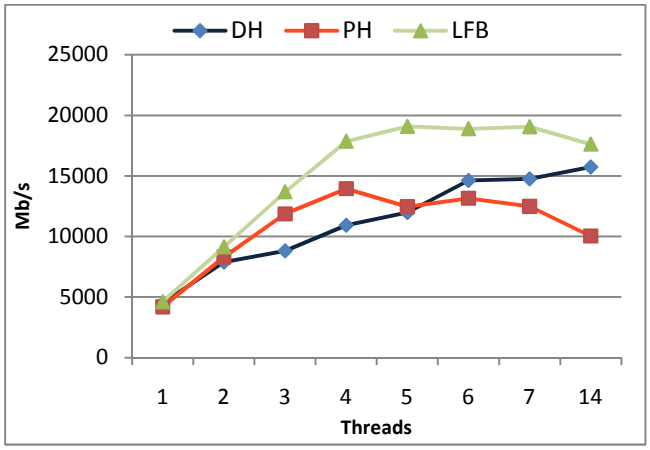

Figure 2: Dominant Capture Raw Throughput

The LFB algorithm performed well on this capture for two reasons. First the work was optimally balanced in the sense that a single thread only processed packets on the dominating TCP connection. The work balance cannot be improved because PAM requires that packets on the same TCP connection be processed sequentially. Second the work was optimally scheduled for performance. Every packet on the dominating TCP connection was processed on the same core and no packets from other connections were processed on the core. Therefore, the data associated with the dominating TCP connection stayed in cache, improving the cache hit percentage, resulting in fewer cycles per instruction.

The DH algorithm also received the cache benefit of processing the dominating TCP connection on the same core. However, the thread that processed it also had to process packets on other connections. The dips and peaks in the graph show the thread numbers that resulted in more or less packets mapping to the thread that processed the dominating TCP connection. At 14 threads, no other packets mapped to the thread processing the dominating TCP connection. However, the hardware thread sharing the core ultimately limited the throughput to approximately 12% lower than LFB.

The PH algorithm had the worst performance. This is because at 14 threads over 67% of packets were enqueued (i.e. handed off) to other threads. As the number of threads increased so did the number of handoffs. Handing off a packet to another thread is expensive because the data link, internet, and transport layer information is copied and enqueued with the packet to avoid processing those layers again. Also, enqueuing a packet to another thread requires an atomic operation since there could be multiple producers (i.e. multiple packets on the same connection being enqueued by different PAM threads). For these reasons the performance decreased with more than 4 threads.

4.3.2 Many Capture Raw Throughput

The Many capture does not contain a single TCP connection that dominates the bandwidth or the packet processing cycles. In fact, the fastest TCP connection is responsible for less than 0.4% of the total packets. Figure 3 shows the maximum raw throughput results for the Many capture. LFB has the highest throughput at every thread count tested. This is because of the cache benefits each thread gets from preferring to process a packet on the flow bundle it just processed over packets that may have arrived earlier. Even when there is just 1 thread, LFB performs better than the other algorithms because of this. At 14 threads LFB is around 25% faster than DH.

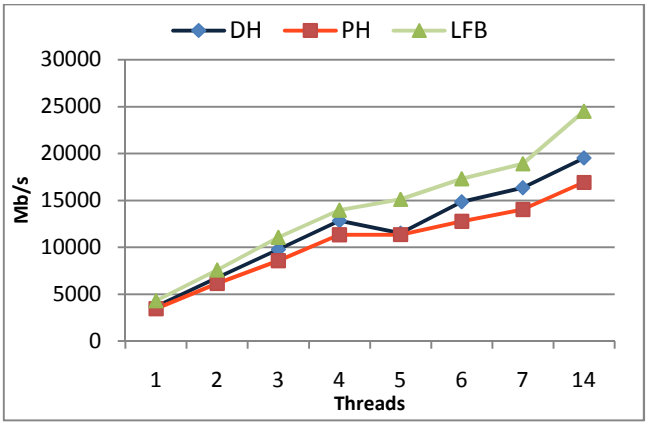

Figure 3: Many Capture Raw Throughput

DH performs better than PH because the workload is almost perfectly balanced and it, like LFB, gets the cache benefits of having the same thread process every packet on a flow. Nevertheless, because there are more interleaving packets on different flows, it does not perform as well as LFB. PH is about 15% slower than DH even with less than 10% of the packets being handed off to other threads. The slower performance is due to cache misses from the data associated with the packet's connection entry not being in cache.

4.3.3 Balanced Capture Raw Throughput

The throughput results for the Balanced capture are presented in Figure 4. Again, LFB has the highest throughput at every thread count tested because of the cache benefits of the scheduling algorithm. At 14 threads it is 38% faster than DH. DH performs well at 5 threads, but does not do as well at 4, 6, 7, and 14 due to load imbalances. In fact, PH outperforms DH at 4 threads. This shows that a load imbalance can outweigh the cache benefits if it is large enough. However, cache misses limit PH's scalability above 4 threads.

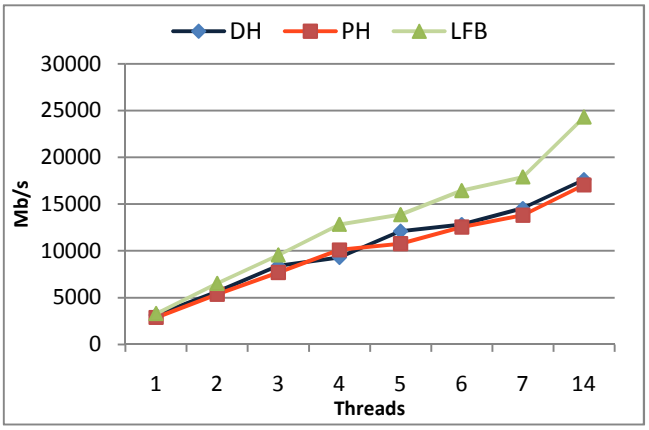

Figure 4: Balanced Capture Raw Throughput

4.3.4 Scaled Throughput

The maximum scaled throughput results for all captures and algorithms are shown in Figure 5. The throughput for all

algorithms decreased in comparison to the maximum raw throughput because of bandwidth spikes that are from 1.5 to 3 times the average and last for several milliseconds.

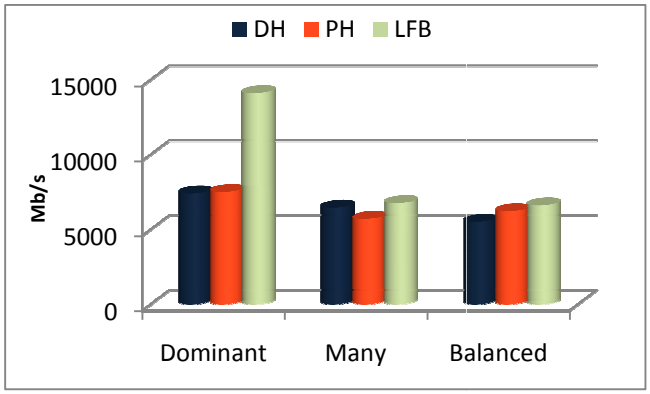

Figure 5: Scaled Throughput

Notice that the throughput decline of the LFB algorithm for the Dominant capture is significantly less than the others. We suspect this is due to the fact that it processes packets in a different order than the rest. For example, assume there is a three millisecond burst of packets. If the packets that are least expensive, in terms of clock cycles, are processed first then more packet buffers will be available for arriving packets than if they were processed in their original order; thus, increasing the maximum scaled bandwidth. Another interesting observation is that PH performed better than DH on the Dominant capture and the Balanced capture. This is due to load imbalances during packet bursts resulting in dropped packets; therefore, reduced throughput.

4.4 Latency Results

Figure 6 shows the average latency of all three packet scheduling algorithms for each capture. These numbers represent the mean time taken to process a packet at 80% of the maximum scaled throughput for the packet scheduling algorithm. The latency numbers are all below 100 microseconds. DH has the highest average latency for all three captures. This is because of work imbalances causing packets to wait in queues even when the system is lightly loaded.

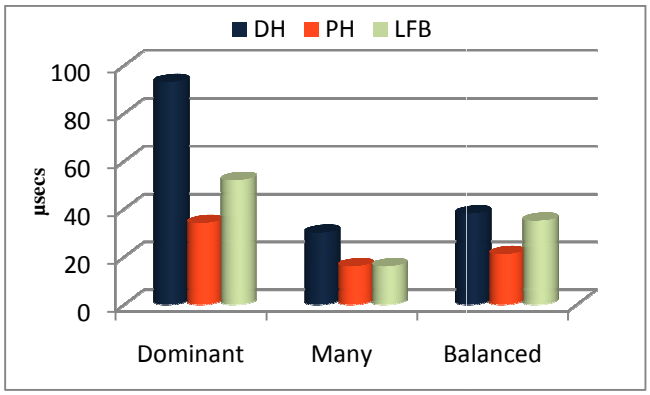

Figure 6: Average Latency

The maximum latency results are presented in Figure 7. LFB has the highest maximum latency for the Dominant and Balanced captures. This is due to the order the packets are processed. During a burst of traffic, packets in the RQ can starve while packets on an index in the FBQ are processed. This causes the packets sitting in the RQ to accumulate latency while waiting to be processed. This does not happen with the Many capture because of the high connection density and the large number of connections that prevent RQ starvation.

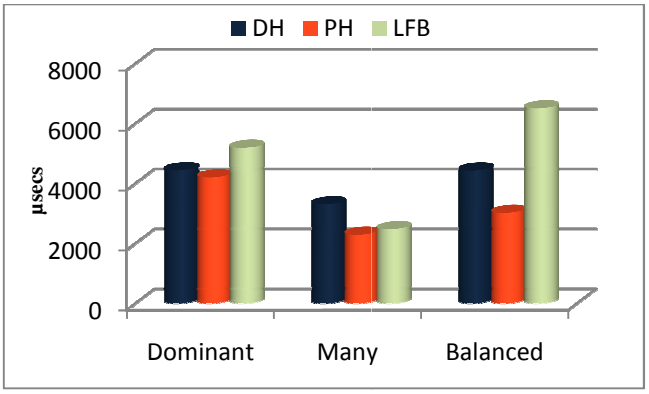

Figure 7: Maximum Latency

4.5 Cache Measurements

Figure 8 shows the average number of L1, L2, and L3 cache misses per packet with seven PAM threads for the three packet scheduling algorithms and network captures. For the Dominant capture you can see the considerable number of cache misses for PH resulting from the large number of packet handoffs. As for DH and LFB, the average number of cache misses per packet is almost identical for the Dominant capture; yet, LFB had 29% higher throughput. This is because DH processed around 8% more packets on the core that processed the dominant connection. These additional packets were more expensive to process due to their associated state not being in cache causing the average cache misses per packet to be higher on the dominant core for DH (see figure 9). So, more packets plus a higher average number of cache misses per packet caused DH to be slower.

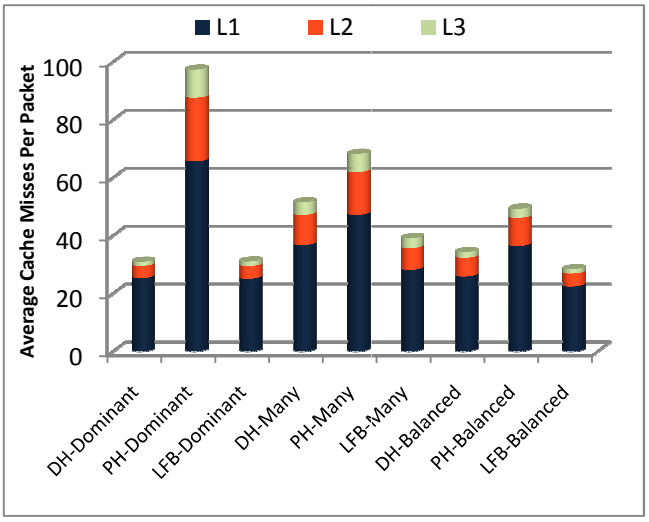

Figure 8: Average Cache Misses Per Packet

The Many capture produced the highest number of cache misses per packet for DH and LFB. This is due to the large number of concurrent connections (i.e. connection density) in the capture that results in more interleaved connection processing. As for the Balance capture, LFB had the fewest cache misses, as it did for all of the network captures. DH, in general, had more cache misses than LFB because of load imbalances and more connection interleaving.

In order to determine how the system's cache size impacted the performance of the three scheduling algorithms, we measured their throughput with half of the L3 cache available. Since there is no way to disable half of the L3 cache on the processor, we created a cache clobbering thread to run on one core of each processor. The cache clobbering thread allocates 4MB of cache aligned memory and reads the first byte of each cache line in a loop; thus, half of the L3 is invalidated with each pass. Figure 10, 11 and 12 show the results of these experiments.

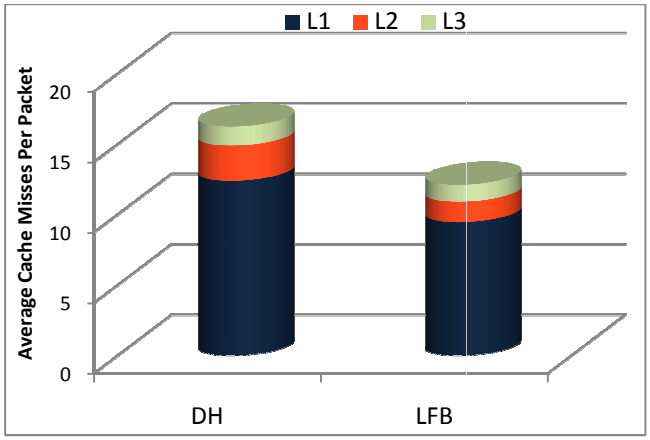

Figure 9: Dominant Core Average Cache Misses Per Packet

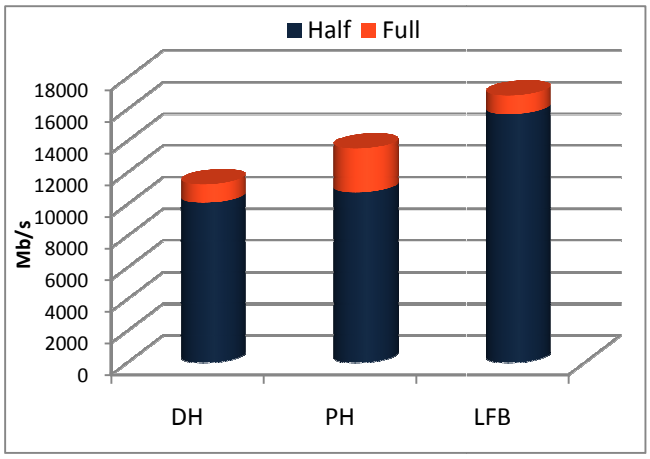

Figure 10: Dominant Capture with ½ L3 Cache

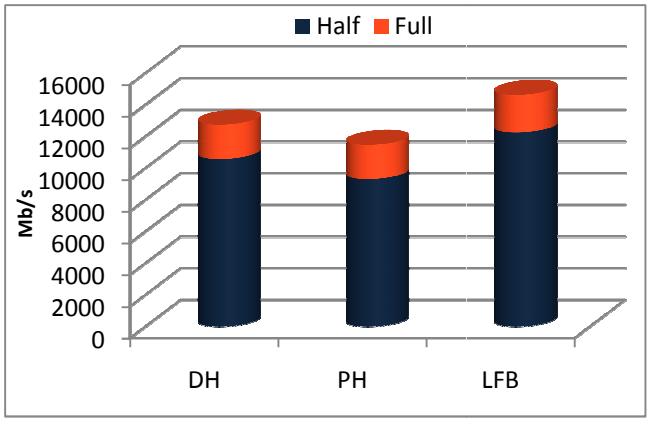

Figure 11: Many Capture with ½ L3 Cache

For the Dominant capture, LFB performance only declined by around 7%. L3 cache misses did increase, but only by a small amount on the core processing the dominant connection. This is because the state for the dominant connection easily fits in the smaller L3 cache and it is accessed enough to keep it in cache. As for DH, it experienced a higher percentage decrease. This is due to an increase in cache misses on other connections that were being processed on the same core as the dominant connection.

The results for the Many capture show a similar percentage throughput decrease for all three packet scheduling algorithms. This is because of the large number of connections (i.e. many connections) that are processed concurrently resulting in the need of a larger L3 to cache all of the state. In the case of the Balanced capture, DH had the lowest decrease in throughput. This is the result of the uneven distribution of work across the cores. The core assigned the most work by DH did not increase its number of L3 cache misses by the same proportion as the other cores. Thus, even though its increase in L3 cache misses is slightly higher than LFB, it did not have as much of an impact on its performance.

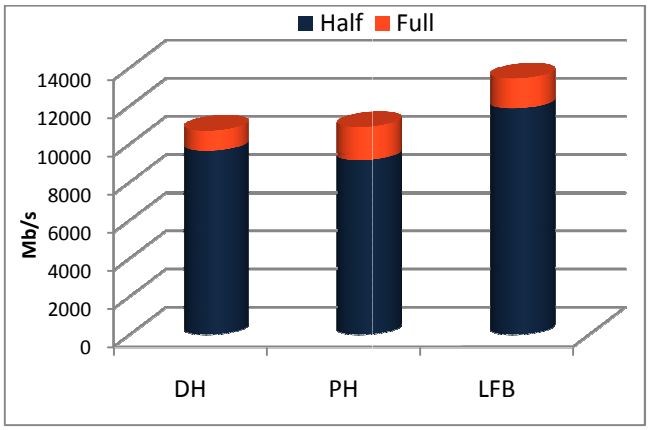

Figure 12: Balanced Capture with ½ L3 Cache

4.6 Discussion

The LFB algorithm is the best performer in terms of throughput. This is because of the temporal locality cache benefit gained by preferring to process packets that map to the FBQ of the last packet over packets that may have arrived earlier. But, reordering packets for optimal temporal locality can have a negative impact on latency. In fact, we observed this effect on the Balanced capture. The DH algorithm also benefits from temporal locality because packets on the same flow bundle are always processed by the same thread. In addition, the DH algorithm does not reorder packets so a high maximum latency is less of an issue. However, interleaving packets on different flows and load imbalances reduce its maximum throughput. The PH algorithm produces the most balanced load in terms of packets processed by threads; yet, its throughput is typically lower than the other algorithms. This is because it does not exploit the cache benefits of temporal locality in the network traffic.

5. CONCLUSION AND FUTURE WORK

In this paper, we discussed the design and implementation of two packet scheduling algorithms we invented. Each one was designed to maximize a different attribute of our ideal scheduler. We compared our two packet schedulers against DH, an algorithm commonly used to schedule packets in network applications. The results show the importance of cache affinity in packet scheduling. In fact, LFB, our packet scheduler that maximizes cache affinity, outperformed the other two schedulers in terms of throughput for all network captures. For the Balanced network capture, LFB's throughput was 38% faster than the next best

scheduler. All in all, our results show that scheduling packets for cache affinity is more important for throughput than balancing the workload evenly.

In the future, we plan to modify the LFB scheduling algorithm so that thread cache affinity information is maintained even when all packets on a FBQ have exited the system. One way to do this is to record the thread and packet number associated with the last packet processed on a FBQ. Then, when a new packet on a FBQ arrives, the packet scheduler can use that information to decide which thread should process it (the last or least loaded thread). In addition, when we finish integrating LFB with IBM's packet driver, we plan to compare the performance of LFB and RSS [20] on a live network. Furthermore, we plan to evaluate our packet scheduling algorithms on platforms that have larger core and hardware thread counts as they become available to us. We suspect that scheduling for cache affinity will be as important, if not more important, on these systems.

6. ACKNOWLEDGEMENTS

The authors would like to thank Hubertus Franke, Ada Gavrilovska, and Paul Palmer for reading early versions of this paper and providing invaluable feedback. In addition, we would like to thank Andrew Baumann (this paper's "shepherd") and the anonymous ANCS reviewers for their comments and suggestions that improved this paper. Finally, we want to thank the IBM PAM 2.0 team for their support and feedback.

7. REFERENCES

[1] Application Layer Packet Classifier for Linux (L7-filter), http://l7-filter.sourceforge.net.

[2] Bennett, J. C., Partridge, C., and Shectman, N. 1999. Packet reordering is not pathological network behavior. *IEEE/ACM Trans. Netw.* 7, 6 (Dec. 1999), 789-798. DOI= http://dx.doi.org/10.1109/90.811445.

[3] Blanton, E. and Allman, M. 2002. On making TCP more robust to packet reordering. *SIGCOMM Comput. Commun. Rev.* 32, 1 (Jan. 2002), 20-30. DOI= http://doi.acm.org/10.1145/510726.510728.

[4] Cao, Z., Wang, Z., and Zegura, E. 2000. Performance of hashing-based schemes for Internet load balancing. *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol.1, no., pp.332-341 vol.1, 2000. DOI= 10.1109/INFCOM.2000.832203.

[5] Dittmann, G. and Herkersdorf, A. 2002. Network processor load balancing for high-speed links. *Proc. Int. Symp. Performance Evaluation of Computer and Telecommunication Systems* (SPECTS'2002), San Diego, CA, Jul. 2002, pp. 727-735.

[6] Duffy, J. 2009. 100 gigabit ethernet: bridge to terabit Ethernet. *Network World*, http://www.networkworld.com/news/2009/042009-terabit-ethernet.html/.

[7] Franke, H., et al. 2010. Exploiting heterogeneous multi-core processor systems for high-performance network processing. *IBM Journal of Research and Development, Network–Optimized Computing*, Vol. 54, No. 1.

[8] Franke, H., et al. 2010. Introduction to the wire-speed processor and architecture. *IBM Journal of Research and Development, Network–Optimized Computing*, Vol. 54, No. 1.

[9] Guo, D., et al. 2009. An adaptive hash-based multilayer scheduler for L7-filter on a highly threaded hierarchical multi-core server, In *Proceedings of the 5th ACM/IEEE Symposium on Architectures For Networking and Communications Systems* (Princeton, New Jersey, October 19 - 20, 2009). ANCS '09.

[10] Guo, D., et al. 2008. A scalable multithreaded L7-filter design for multi-core servers. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures For Networking and Communications Systems* (San Jose, California, November 06 - 07, 2008). ANCS '08. ACM, New York, NY, 60-68. DOI= http://doi.acm.org/10.1145/1477942.1477952.

[11] IEEE Std 802.1AX-2008 IEEE Standard for Local and Metropolitan Area Networks — Link Aggregation. IEEE Standards Association, http://standards.ieee.org/getieee802/download/802.1AX-2008.pdf.

[12] Intel® Xeon® Processor E5540, http://ark.intel.com/Product.aspx?id=37104.

[13] Jain, R. and Routhier, S. 1986. Packet trains: measurement and a new model for computer network traffic. *IEEE Journal of Selected Areas in Communications*, SAC-4(6):986-995, September 1986.

[14] Li, C., Peng, G., Gopalan, K., and Chiueh, T. 2003. Performance Guarantees for Cluster-Based Internet Services. In *Proceedings of the 23rd international Conference on Distributed Computing Systems* (May 19 - 22, 2003). ICDCS. IEEE Computer Society, Washington, DC, 378.

[15] Kandula, S., et al. 2007. Dynamic load balancing without packet reordering. *SIGCOMM Comput. Commun. Rev.* 37, 2 (Mar. 2007), 51-62.

[16] Martin, R., et al. 2006. Accuracy and dynamics of hash-based load balancing algorithms for multipath Internet routing. *Broadband Communications, Networks and Systems, 2006. BROADNETS 2006. 3rd International Conference*, vol., no., pp.1-10, 1-5 Oct. 2006.

[17] NSS Labs. IBM ISS GX6116 Intrusion Prevention System achieves NSS labs gold award and certification. http://nsslabs.com/2008/ibm-iss-gx6116-intrusion-prevention-system-achieves-nss-labs-gold-award-and-certification.html/.

[18] Paxson, V. 1999. Bro: a system for detecting network intruders in real-time. *Computer Networks*. 31(23-24), pp. 2435-2463, 14 Dec. 1999.

[19] Proventia Network Intrusion Prevention System, http://www-935.ibm.com/services/us/index.wss/offerfamily/iss/a1030570/.

[20] Receive Side Scaling (RSS), http://www.microsoft.com/whdc/device/network/ndis_rss.mspx/.

[21] RFC 3393, http://tools.ietf.org/html/rfc3393.

[22] Roesch, M. 1999. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX Conference on System Administration* (Seattle, Washington, November 07 - 12, 1999).

[23] Shi, W., MacGregor, M. H., and Gburzynski, P. 2004. Load Balancing for Parallel Forwarding. *IEEE/ACM Transactions on Networking*, 2004.

[24] Shi, W., MacGregor, M. H., and Gburzynski, P. 2005. A scalable load balancer for forwarding internet traffic: exploiting flow-level burstiness. In *Proceedings of the 2005 ACM Symposium on Architecture For Networking and Communications Systems* (Princeton, NJ, USA, October 26 - 28, 2005). ANCS '05. ACM, New York, NY, 145-152. DOI= http://doi.acm.org/10.1145/1095890.1095911.

[25] Shi, W. and Kencl, L. 2006. Sequence-preserving adaptive load balancers. In *Proceedings of the 2006 ACM/IEEE Symposium on Architecture For Networking and Communications Systems* (San Jose, California, USA, December 03 - 05, 2006). ANCS '06. ACM, New York, NY, 143-152. DOI= http://doi.acm.org/10.1145/1185347.1185367.

[26] Snort 3.0, http://www.snort.org/snort-downloads/snort-3-0/.

[27] Sommer, R., Paxson, V., and Weaver. 2009. An architecture for exploiting multi-core processors to parallelize network intrusion prevention. *Concurr. Comput. : Pract. Exper.* 21, 10 (Jul. 2009), 1255-1279.

[28] Squiillante, M. S., and Lazowska, E. D. 1993. Using processor-cache affinity information in shared-memory multiprocessor scheduling. *IEEE Trans. Parallel Distrib. Syst.* 4, 2 (Feb. 1993), 131-143.

[29] Yu, H., et al. 2008. Stateful hardware decompression in networking environment. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (San Jose, California, November 06 - 07, 2008). ANCS '08. ACM, New York, NY, 141-150. DOI= http://doi.acm.org/10.1145/1477942.1477968.

[30] Zipf, G. K. 1949. *Human Behavior and the Principle of Least-Effort*, Addison-Wesley, Cambridge, MA, 1949.