An Efficient Parallelized L7-Filter Design for Multicore Servers

Danhua Guo, Laxmi Narayan Bhuyan, *Fellow, IEEE, ACM*, and Bin Liu, *Member, IEEE*

***Abstract*—L7-filter is a significant deep packet inspection (DPI) extension to Netfilter in Linux's QoS framework. It classifies network traffic based on information hidden in the packet payload. Although the computationally intensive payload classification can be accelerated with multiple processors, the default OS scheduler is oblivious to both the software characteristics and the underlying multicore architecture. In this paper, we present a parallelized L7-filter algorithm and an efficient scheduler technique for multicore servers. Our multithreaded L7-filter algorithm can process the incoming packets on multiple servers boosting the throughput tremendously. Our scheduling algorithm is based on Highest Random Weight (HRW), which maintains the connection locality for the incoming traffic, but only guarantees load balance at the connection level. We present an Adapted Highest Random Weight (AHRW) algorithm that enhances HRW by applying packet-level load balancing with an additional feedback vector corresponding to the queue length at each processor. We further introduce a Hierarchical AHRW (AHRW-tree) algorithm that considers characteristics of the multicore architecture such as cache and hardware topology by developing a hash tree architecture. The algorithm reduces the scheduling overhead to $O(\log N)$ instead of $O(N)$ and produces a better balance between locality and load balancing. Results show that the AHRW-tree scheduler can improve the L7-filter throughput by about 50% on a Sun-Niagara-2-based server compared to a connection locality-based scheduler. Although extensively tested for L7-filter traces, our technique is applicable to many other packet processing applications, where connection locality and load balancing are important while executing on multiple processors. With these speedups and inherent software flexibility, our design and implementation provide a cost-effective alternative to the traffic monitoring and filtering ASICs.**

***Index Terms*—Cache topology, connection locality, deep packet inspection (DPI), L7-filter, load balance, multicore processors, multithreading.**

Manuscript received April 14, 2010; revised May 21, 2011 and November 09, 2011; accepted November 22, 2011; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor T. Wolf. Date of publication December 12, 2011; date of current version October 11, 2012. This work was supported by the NSF under Grants CNS-0832108 and CSR-091285, and supported in part by the NSFC under Grants 60873250 and 61073171, the Tsinghua University Initiative Scientific Research Program, and the Specialized Research Fund for the Doctoral Program of Higher Education of China (20100002110051). Parts of this work were presented in papers [7] and [8].

D. Guo is with Microsoft Corporation, Mountain View, CA 94043 USA (e-mail:danguo@microsoft.com).

L. N. Bhuyan is with the Computer Science and Engineering Department, University of California, Riverside, Riverside, CA 92508 USA (e-mail: bhuyan@cs.ucr.edu).

B. Liu is with the Computer Science and Technology Department, Tsinghua University, Beijing 100084, China (e-mail: liub@tsinghua.edu.cn).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNET.2011.2177858

I. INTRODUCTION

DEEP packet inspection (DPI) is being increasingly required in network devices to identify and process sophisticated application-specific network flows. Among much DPI software, L7-filter is most widely used as an extension to Netfilter in Linux [1], [10], [11], [14], [20]. It was initially deployed to control spiraling volumes of P2P traffic, but its use has widened because of its proven relevancy in a number of other contexts, including network security, service packaging, and service management. Despite DPI's increasing popularity, research in both academia [14], [20], [32], [35], [40] and industry [9], [15], [18] has shown that its performance terribly lags behind wire speed. The emergence of high-speed networks, such as 10 Gigabit Ethernet (10 GbE), increases the intensity of network traffic, which further elevates the demand for faster DPI processing.

Extensive research has been conducted to accelerate DPI processing using special-purpose hardware, such as Network Processor [19], [20], FPGA [24], [27], TCAM [23], [28] and ASIC [3], [7], [8], [14]. While hardware solutions provide satisfactory performance, software techniques are more flexible and cost-effective. They can adapt easily to changes in algorithm and application characteristics. Also, development of hardware including testing, fabrication, and reconfiguration is expensive and time-consuming. Some existing research [20], [32], [38], [40] optimizes signature representations, i.e., regular expression, for fast and memory-efficient DPI. However, none of these approaches addresses DPI optimization in the context of general-purpose multicore-based servers, which are being increasingly employed in the mainstream network devices.

Developing efficient multithreaded programs for efficient use of multicore servers involves studies in both the software algorithm and the underlying hardware architecture. The design of the parallelized DPI algorithm should create multiple threads to explore the intrinsic data sharing in the DPI program and schedule these threads efficiently considering both architecture and communication characteristics. Packet processing is a natural candidate for multiprocessing because abundant packet-level parallelism is available. However, a multithreaded program with necessary synchronization must be designed to take advantage of this parallelism.

Previous research [1], [3], [10], [11], [14], [20] has demonstrated that the performance of DPI is bounded by the cost of pattern matching. We developed a decoupled model to separate the packet arrival handling from pattern matching and focused

on optimizing the pattern matching operations at the application layer [11]. For L7-filter, the classification of one connection might require multiple connection buffers, with different numbers of packets. Because packets in the same connection share the packet header information, it is intuitively beneficial to keep the processing of the same connection on the same processing unit (PU). Maintaining such connection locality warms up the cache with reusable packet data, so that future classifications of the same connection benefit from reusing the shared data. This results in a cache hit without accessing the memory, which is an order of magnitude more expensive. Our previous results showed that this affinity-based multithreading mechanism significantly increases performance scalability for the L7-filter on an Intel Xeon-based server [11].

The benefits of connection locality may, however, be offset by load imbalance due to network flow characteristics, where the connections have different number of packets. On highly threaded hierarchical multicore servers, e.g., Sun Niagara 2 or Intel Xeon, the accumulative load imbalance further deteriorates the performance. As shown in Section V, the benefits of connection locality can be outweighed by load imbalance in this case. Previous approaches have been proposed to balance the workload at the connection level so that each core shares a similar number of connections [30], [36]. In this realm of work, a robust hash function called Highest Random Weight (HRW) was proposed [36], which strikes a balance between locality and connection-level load balancing. Given an object name and a set of servers, HRW assigns a weight to each server and maps the request to the server with the maximum weight. HRW is a robust hash function and was originally proposed to map object requests to a cluster of servers. However, given the commercial multicore processors today, the mapping needs to consider the asymmetrical distribution and topology of cores. Furthermore, the load balance problem cannot be addressed at the connection level alone due to the difference in connection length in L7-filter. The extreme case happens when there are more cores than connections. Thus, a good load-balanced system should be able to use all the cores by distributing workload at *packet level* instead of *connection level*, while also trying to maintain the connection locality.

To strike a balance between connection locality and packet-level load balancing, we design an Adapted Highest Random Weight (AHRW) hash scheduling mechanism and verify our scheduler using real machine measurements. AHRW enhances HRW by introducing an additional feedback vector that is multiplied by the weight generated by HRW for each PU. Our hash adjustment technique follows the heterogeneous robust hash theory described in a previous work [30] to maintain the HRW *minimum disruption property*. This theory guarantees a good balance between connection locality and load balancing by going through a recursive computation. However, because of the high complexity of the computation, we propose and verify a simplified but effective alternative technique. In our solution, the HRW weight for each PU is updated directly based on the runqueue length of each PU. Our results show that the loss of accuracy is negligible, while the improvement in performance is significantly enhanced. A similar technique was adopted to develop an adaptive load balance technique for URL requests based on processor utilization [19], but was verified through simulation. Also, designing a feedback system based on processor utilization is difficult to implement on a per-packet basis in a real system. Next, we develop a solution for a hierarchical environment, where the resources on a multicore server form a virtual "tree" structure. By extending our scheduler into these tree structures rather than running a linear scheduling, we can reduce the scheduling overhead from $O(N)$ to $O(\log N)$, where N is the number of scheduling candidates. The above hierarchical scheduling technique is a major difference of our contribution compared to the previous work [13], [19], [30], [36].

We verify our AHRW hash tree scheduler by executing some real traces on a Sun Niagara 2 server. We show that the system throughput can be improved by about 50% compared to the heuristic based on pure connection locality. Although we experiment only with the Niagara 2 machine, our hash tree scheduler can be extended to different multicore servers with hierarchical cache locality, such as Intel Xeon servers.

The major contributions of this paper can be summarized as follows:

- design and implementation a parallelized L7-filter algorithm at the connection and packet levels;
- development of AHRW for L7-filter to strike a balance between connection locality and packet-level load balance;
- design of a Hierarchical HRW (AHRW-tree) scheduler as suitable for the underlying multicore architecture with different thread or cache topologies;
- verification the results through actual execution of real traces on a highly threaded hierarchical multicore server, i.e., Sun Niagara 2.

The remainder of the paper is organized as follows: In Section II, we provide background information on DPI and L7-filter and related work on multicore scheduling. In Section III, we design a parallelized L7-filter and present the details of AHRW and its theoretical background. In Section IV, we provide the enhanced tree version of this schedule, called AHRW-tree. In Section V, we describe experimental methodology based on real traces and implementation on SUN Niagara. In Section VI, we present the performance measurements and various sensitivity studies to show the superiority of our design. In Section VII, we discuss the scope of our scheduler for general packet processing on other architectures. Finally, in Section VIII, we conclude the paper and propose future research directions.

II. Related Work

A. L7-Filter and DPI Optimizations

The intensity of network resource competition increases as more applications demand higher bandwidth and more computing capability. Traditional packet classification software, such as Netfilter in Linux, identifies and controls packet flows based on layer-3 and layer-4 information, i.e., IP addresses

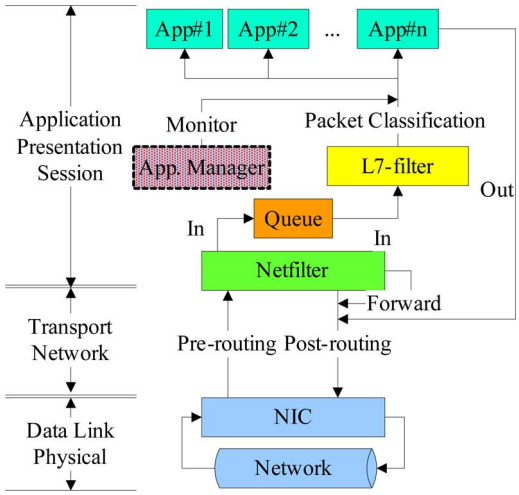


Fig. 1. L7-filter in OSI model.

and port numbers. Many recent applications, such as P2P and HTTP, however, hide their port numbers in the payload or require dynamic allocation for port numbers during connection establishment. Under such circumstances, DPI plays a key role in bandwidth management and traffic reshaping. It is increasingly used to augment network security and underpin service creation and service management tools.

Fig. 1 illustrates the structure of a Linux networking system with L7-filter in an OSI model. It sits on top of the transport layer, monitoring network traffic based on protocol features represented in packet payloads. While Netfilter relies on *iptables* to accept/forward/drop incoming packets, L7-filter further marks all the accepted packets with their protocol IDs. Potential process/application managers can easily pick up the packet protocol IDs and reshape the traffic for security and management concerns.

By matching against signature fields of various protocols, L7-filter uses GNU regular expression matching to obtain the protocol type associated with the application layer data in the packet. Different from signature-based intrusion detection systems (IDSs), which have thousands of complex network security regulation sets, signature-based protocol parsing schemes are comparatively simple with only hundreds of protocol matching rule sets. They can be easily implemented and deployed in software without any specific hardware accelerators. Even then, the computation cost of the L7-filter software still remains high for real-time processing of packets [1], [20] in high-bandwidth networks.

The pattern matching in DPI programs has been studied extensively at the sequential program level. Major research in this domain falls into three categories: 1) reducing the alphabet size [3]; 2) increasing throughput by processing multiple input characters per clock cycle [14], [20]; and 3) balancing between the memory bandwidth and memory size requirement [39]. Another direction of the research studies the deployment of DPI programs, i.e., how to use hardware accelerators. In this domain, both FPGA [27] and Network Processor [19] solutions have been proposed to explore the packet-level parallelism inside DPI program. For example, [34] proposed a "bit-splitting" architecture to explore the internal parallelism inside of the state machines.

However, there is no work on parallel execution of L7-filter using off-the-shelf multicore processors.

B. Load Balance and Flow-Based Scheduling

Load balance and flow-based packet scheduling are two orthogonal directions in packet scheduling because of their inherent incompatibility with each other. Since the advent of multicore-based servers, both techniques have been well studied. Load balance guarantees no single bottleneck in the multicore environment and was recently evaluated using Nash equilibrium [6] and ranked elections [16] models. As link speed reaches the multigigabit era, the server is loaded with more simultaneous connection requests. If all the CPUs on the servers are constantly saturated, we would not worry about workload imbalance as no CPU could be idle. However, load balance has been proven to be a significant problem in network packet scheduling [9], [36]. This is because under most practical circumstances, servers receive traffic in bursts, i.e., at times, the servers are not fully utilized. In this paper, we choose to discuss how to resolve the load imbalance issue from the workload scheduling point of view and propose a scheduler that uses given resources more efficiently.

The connection-based DPI programs are gaining publicity in both academic studies [11], [20], [34], [40] and industrial products [7], [15], [18]. In L7-filter, incoming packets are preprocessed and then placed in a reassembling buffer. Each connection has a registered entry in the reassembling buffer. A preprocessed packet is appended to the corresponding connection entry in the buffer, and the entire new entry triggers the matching engine for classification. Upon receiving the classification result, any further packets of the current connection will be marked with the matched protocol ID and bypass the classification engine. If the matching engine cannot find a match, the classification for this connection will be triggered every time a new packet of this connection comes in, and this new packet is reassembled into a new connection buffer. An entry in the reassembling buffer can hold up to eight packets for each connection. If a connection cannot be classified with eight packets in the buffer, any further packets for this connection will be excluded from matching.

As multicore processors have become the *de facto* server platforms, a recent trend is moving toward the deployment of multithreaded DPI programs on multicore servers. As we presented in our previous paper [11], a multithreaded L7-filter program could achieve a speedup of 7.6 $\times$ in TCP throughput using an 8-core Intel Xeon server. The reason behind this gain is to maintain the connection locality for incoming traffic to benefit from cache locality. Our research result is in line with the widespread Receive Side Scaling (RSS) technique implemented in NIC [29] as well as findings from an Intel Research group [37].

However, this paper explores opportunities to strike a balance between load balance and connection locality. In addition, our scheduler provides packet-level load balancing instead of the traditional connection-level balancing. This is particularly useful when network traffic follows the "packet train" arrival pattern, as described in [17].

C. Hash-Based Packet Scheduling

Hash functions generate independent, uniformly distributed variables. They provide theoretical load balance over the input key-value mappings. In the client–server model, hash functions are a favorable choice to map client-requested objects into the Web cache [36]. HRW [19], [30], [36] and Toeplitz [29] hash have been shown to be are very suitable for network applications.

HRW is a robust hash function and was originally proposed to map object requests to a cluster of servers [30], [36]. HRW has been popular in the areas of servers, Web caching, and clustered digital libraries [10], [17], [30], [36]. Given an object name and a set of servers, HRW assigns a weight to each server and maps the request to the server with the maximum weight. Because connection buffers of the same connection share the same connection ID, HRW guarantees the connection locality when the object name is represented by the connection ID. The term "robust" refers to the efficient maintenance of connection locality, i.e., it requires only a minimum amount of remapping when the connection locality is relaxed for packet-level load balance.

The major drawback of hash mappings is that they are not adaptive to real-time performance variation and therefore are potentially vulnerable to traffic imbalance. In our proposed scheduler, it takes advantage of the randomness of the HRW hash, meanwhile adjusting the weight function by a feedback vector to provide load balance at the packet level. The work presented in paper [19] is the closest to the scheduler in this paper. However, our research differs from that paper in that: 1) we choose a low-overhead feedback metric, runqueue length, to provide better load balance rather than to poll values from the hardware counter, which is infeasible to do at a per-packet basis in a high-speed network; and 2) we design and implement a hash-tree scheduler that is well suited to work in a hierarchical general-purpose multicore environment, which is increasingly popular in server deployment. The architecture of the underlying multicore processor is considered while designing our scheduling algorithm.

III. Multithreaded L7-Filter Design With AHRW Scheduler

In this section, we first present the design of a multithreaded L7-filter that can classify the packets in parallel using a multiprocessor, analyze a robust hash function HRW [30], [36] that forms the basis of scheduling, and then propose the new hash function AHRW.

A. Parallelized L7-Filter Algorithm

In order to accelerate the costly pattern matching in L7-filter as we have discussed in Section II-A, we proposed the first parallelized algorithm for L7-filter. This algorithm has been accepted as an important show case for DPI optimizations using multicore architecture [4]. Algorithm 1 illustrates the data flow in the parallelized L7-filter algorithm. In this algorithm, the original online L7-filter is substituted by a combination of a preprocessing thread (PT) and a set of matching threads (MTs). The PT works as a real network stack in the kernel and schedules the packets. MTs, on the other hand, focus on the classification of packets. Each MT has a runqueue that carries connection buffers yet to be classified. The scheduler decides whether an incoming packet needs to be classified based on current classification result of its connection. If the connection has not been classified, the incoming packet will be used to assemble a new connection buffer for the connection, and the scheduler will schedule the connection buffer to an MT accordingly.

Algorithm 1: The Multithreaded L7-Filter Algorithm

```
Main Thread (Preprocessing Thread)
pthread_create() ;                    // create MT pool
FOR each incoming packet P in QUEUE
  Check header of P in connection table;
  Decide its belonging connection C;
  IF C is a new connection THEN
   Build a new Cbuffer with P;
   Add C to connection table;
  ELSE IF Cmark==MATCHED
        || Cmark==NO_MATCH THEN
    CONTINUE;
  ELSE  Attach P to Cbuffer;          // Old connection
  ENDIF
  Schedule(Cbuffer)                   // Call scheduler
END LOOP
pthread_join();
=======================================
Schedule(Cbuffer)
IF C is a new connection THEN
  // AHRW is explained in section 3.4
  dst_core = Find a core using AHRW;
  Append Cbuffer to MT[dst_core]'s runqueue;
ELSE MT[dst_core] has a full runqueue
  Wait();
ENDIF
=======================================
MT thread                  // Procedure for each MT
Affinitize itself to CPU[MT_ID];  // Thread affinity
WHILE runqueue is not empty
  Dequeue(runqueue);
  // Classification Synchronization
  Recheck Cmark in connection tracking table;
  IF Cmark == NO_MATCH_YET THEN
    Cmark=Classify(Cbuffer);          // GNU syscall
  ELSE
    CONTINUE;
  ENDIF
  IF Cmark==NO_MATCH_YET
       AND Cbuffer_size==8 THEN
       Cmark = NO_MATCH;
  ENDIF
END LOOP
```

Specifically, we maintain a connection table (to record classification status for each connection) and a reassembly buffer (to save up to eight packets of any given connection) for the RE matching. It is a stateless matching in the sense that any connection that has not been classified will be reclassified from the beginning of the connection buffer, as long as a new packet of the same connection arrives. Of course, the new packet will be reassembled to the buffer for the connection. At any point of processing, a connection can only have one of the three statuses: 1) *MATCHED*; 2) *NO_MATCH*; and 3) *NO_MATCH_YET*. For any incoming packet, L7-filter first decides the host connection

based on the 5-tuple of this packet. It is then preprocessed based on the connection status in one of the following two ways.

For 1) or 2): L7-filter already marks a final result to the connection. No further action is necessary.

For 3): This packet is appended to the corresponding connection in the assembling buffer, and the new buffer is placed in the runqueue of an MT.

For both cases, the PT goes back to fetch the next packet from the trace file only after the current packet has been preprocessed. On the other hand, the MT keeps matching the connection in its runqueue until the queue is empty. If the number of packets in a connection exceeds a predefined threshold before the connection is classified, the connection is marked as "*NO_MATCH*." Note that the first packets in the connection buffer can be classified multiple times, which increases processing in MT and has impact to the measured throughput. As a result, the throughput with stateful matching, i.e., matching first packet of a connection only once and record the matching state, could probably be higher. However, the additional data structure for storing temporary matching states is not only a separate complicated issue to discuss; it might introduce higher memory requirement.

Interested readers can refer to our previous paper [11] for more details of the parallelized algorithm.

B. Baseline Robust Hash Function HRW

To measure the maintenance of connection locality, let us define *disruption coefficient* (DC) as the fraction of updated mappings, i.e., the fraction of packets that loses connection locality. The reasoning behind minimum remapping in HRW is to reduce the DC by dividing the server cluster into small partitions. For simple modulo-M hash functions, the entire mapping set needs to be changed whenever the number of servers changes, and hence the DC value is close to one. However, we partition the servers (hash space) into N sets, assuming that these sets are contiguous and of equal size.

The original N partitions are

$$\left[0, \frac{1}{N}\right), \left[\frac{1}{N}, \frac{2}{N}\right), \left[\frac{2}{N}, \frac{3}{N}\right), \ldots, \left[\frac{N-1}{N}, 1\right].$$

Without loss of generality, assume that the hash space is the interval [0, 1]. Now, suppose we add a server to the space, increasing the number of servers to $\mathrm{N}+1$.

The new $\mathrm{N}+1$ partitions are

$$\left[0, \frac{1}{N+1}\right), \left[\frac{1}{N+1}, \frac{2}{N+1}\right), \left[\frac{2}{N+1}, \frac{3}{N+1}\right), \ldots, \left[\frac{N-1}{N+1}, 1\right].$$

The overlapping intervals represent a fraction of (1-DC) new mappings that agree with the original mappings

$$\left[0, \frac{1}{N+1}\right], \left[\frac{1}{N}, \frac{2}{N+1}\right], \left[\frac{2}{N}, \frac{3}{N+1}\right], \ldots, \left[\frac{N-1}{N}, \frac{N}{N+1}\right].$$

Adding the lengths of these intervals gives

$$\sum_{i=0}^{N-1} \frac{N-i}{N(N+1)} = \frac{1}{N(N+1)} \sum_{i=1}^{N} n = \frac{1}{2}.$$

Therefore, by partitioning the hash space, the DC is reduced by almost half. In HRW, each server ID is combined with the requested object for a mapping to the hash space, further reducing the DC value. Interested readers may find a formal proof in paper [30].

In addition to minimum DC, i.e., efficient maintenance of the connection locality, HRW also provides load balance at the connection level. However, in case of traffic-to-PU mappings, coarse-grained load balance at the connection level is not enough to guarantee system performance. Take the following extreme example. Assume that the incoming packets are distributed over *connections* c_1 and c_2 with probabilities p_1 and p_2, respectively. Under the original HRW, only two PUs can be used to process c_1 and c_2, leaving the other six PUs idle. Again, p_1 and p_2 are not the same, or the connections do not contain the same number of packets for L7-filter, causing further load imbalance. It is necessary to relax the connection locality requirement to provide packet-level load balance so that no PU is idling while runqueues of other PUs are nonempty.

C. Adaptive HRW Theory

In [30], the author addressed the modification of the HRW hashing schemes so as to map connection buffers to various servers with the desired target probabilities. We can directly apply that theory to our packet scheduler on servers based on multicore chips, where packets correspond to URL requests. Let $p_1, \ldots, p_n$ be given arbitrary target probabilities for each PU for $p_1 >= 0$ and $\sum_{i=1}^{N} p_i = 1$. If a PU has target probability p_i, we desire the fraction p_i of connection buffers in the L7-filter to be mapped to it.

In the robust hashing scheme, for a given connection buffer $\vec{c}$, we calculate a hash value $h = h(\vec{c}, p_i)$ for each of i PUs. We then map the connection buffer to the PU that has the highest h value. This scheme will map $1/N$ of the connection buffers to each PU. To deal with target probabilities, we introduce multipliers $x_i, \ldots, x_N$ and multiply each h with the respective x_i and map the connection buffer to the PU that has the largest$Z_i = x_i \cdot h_i$ value. If the multipliers are different, the fractions of connection buffer routed to the PU will no longer be the same.

Given the capacity/probability p_i for PU i, the weight multiplier x_i can be derived recursively as follows:

$$\begin{cases} x_1 = (N \cdot p_1)^{1/N} \\ x_n = \left[\frac{(N-n+1)(p_n - p_{n-1})}{\prod_{i-1}^{n-1} x_i} + X_{n-1}^{N-n+1} \right]^{\frac{1}{N-n+1}} \end{cases}. \quad (1)$$

Then, the robust hash algorithm with multipliers $x_1, \ldots, x_N$ will route the fraction p_i of the connection buffers to the ith PU for $i = 1, \ldots, N$. Interested readers can find a formal proof of (1) in a previous paper [30].

The question now is how to determine p_i to reflect different probability of each PU. Recall that the goal of adaptive scheduling is to relax connection locality for better load balancing. Therefore, an intuitive way to incorporate "load" information into the multiplier is to measure the CPU utilization, as done in a previous work [19]. However, such a technique is complicated and costly to implement and needs a syscall mechanism. Rather, we choose the runqueue length as the feedback metric because it is based on the lightweight data structure in the application layer. Our method is in line with [13], which applies the runqueue metrics for many adaptive scheduling algorithms. Note that $\sum_{i=1}^{N} p_i = 1$, so we need to get a fraction of the measurement results for each node. In case of the runqueue metrics, we need to calculate the ratio between the runqueue length of a node over the total runqueue length of all nodes and use that as the p_i.

This computation is very intensive on a per-packet basis. When we feed real-time measurement results for p_i and calculate the corresponding x_i for each server, the scheduler stalls at the iterative update process. Hence, we propose an alternative feedback mechanism that relaxes the influence vector accuracy, but significantly reduces the scheduler overhead. As our result later shows, our simplified AHRW scheme achieves comparable performance in terms of load balance and throughput to the theoretical derivation.

D. AHRW Scheduler

We introduce a multiplier vector as an adjustment to the original HRW. The new AHRW has properties to maintain the connection locality and only relaxes it for packet-level load balance. Before starting to discuss the validity of AHRW, let us recall the scheduling requirement for L7-filter. From Section III, we know that an incoming packet is first "preprocessed" in the connection reassembly buffer based on its 4-tuple (Source IP, Destination IP, Source port #, Destination port #) information. Therefore, the output of preprocessing, which is also the input to the scheduler, is in the form of a connection buffer, distinguished by the connection ID. At any given point, there could be multiple connection buffers for the same connection because classification for some connection requires multiple packets, leading to multiple connection buffers of different sizes. The scheduler should evenly distribute the connection buffers over the available PU resources and put as many connection buffers with the same connection ID to the same PU as possible. This will preserve connection locality to reduce packet reordering and increase the reuse of shared packet data in the cache. We present the AHRW hash function as follows.

Adaptive HRW Hash $h(\vec{c}, p, r) \rightarrow$ Weight*:* Let $g(\vec{c}, p)$ be the original HRW hash function $g : C \times \{1, 2, \ldots, N\} \rightarrow [0 \ldots 2^{31} - 1]$, where C is the set of possible identifier vectors, i.e., connection IDs; $p \in [1, N]$ is the ID of a PU; and N is the number of PUs. Here g is a pseudorandom function that generates a random variable in $[0 \ldots 2^{31} - 1]$ with uniform distribution for each incoming connection buffer with identifier vector $\vec{c} \in C$ and the PU ID p. We denote $r_p \in (0, 1]$ as the ratio between the minimum runqueue length of all the PUs and the runqueue length of PU p at the point of scheduling. Then

$$h(\vec{c}, p, r_p) = r_p \cdot g(\vec{c}, p)$$

where $r_p = \frac{\min_{p \in [1,N]} \text{QueueLenth}_i}{\text{Queuelenth}_i}$.

As we discussed in Section III-B, the theoretical multiplier based on runqueue length should be derived from the recursive algorithm in (1), where

$$p_i = \frac{\frac{\min_{i \in [1,N]} \text{QL}_i}{\text{QL}_i}}{\sum_{i=1}^{N} \frac{\min_{i \in [1,N]} \text{QL}_i}{\text{QL}_i}} \qquad \text{s.t.} \sum_{i=1}^{N} p_i = 1.$$

The AHRW hash function always reduces the weight on the PU whose runqueue length is greater than the minimum runqueue length. The weight adjustment r_p becomes more aggressive as the runqueue length difference increases. When $r_p = 1$, the current PU has the shortest runqueue length, the AHRW function falls back to the traditional HRW. When the runqueue length of p is 0, we set $r_p = 1$. In this case, p is idle, and it should be assigned the original HRW weight. Thus, the connection locality will not be sacrificed.

Scheduler $\text{sched}(\vec{c}) \rightarrow p$*:*

$$\text{sched}(\vec{c}) \rightarrow \Leftrightarrow p = \arg \max_{j \in [1,N]} h(\vec{c}, j, r_j).$$

For any given connection buffer, the scheduler decision performs load balance at the packet level by relaxing connection locality. We apply adaptations to all the PUs rather than a subset of them to guarantee fairness. The AHRW-based scheduler possesses the following properties:

Optimized Connection Locality: Connection buffers with the same connection ID are initially scheduled to the same PU. Then the runqueue length ratio r_p is applied by the scheduler. The connection locality maintenance is only affected to a minimum extent, i.e., although the generated weight for each PU changes, the selection of the maximum weight PU is affected only when necessary. Note that connection locality and the DC value are directly related: maintaining perfect connection locality as defined in our model. It is equivalent to the case when $\text{DC} = 0$. Because it is impossible to maintain perfect connection locality while applying a feedback system, we only need to justify that the relaxation coefficient DC is minimal for all the connections. When $r_p = 1$, the AHRW falls back to the original HRW, whose property of minimal DC value is proved in [36]. In [19], the authors proved that for a single constant multiplier α, the minimal DC property holds true for an adjustment $\alpha \cdot g$ to the original HRW weight. Because our adjustment is a dynamic process, we expect queue lengths to be similar to achieve load balance. As a result, r_p gradually approximates the value 1 after the system becomes stabilized. When $r_p \in (0, 1)$, our AHRW scheduler also possesses the minimum DC property.

Load Balance: If necessary, the scheduler relaxes the connection locality by applying the runqueue length ratio r_p. It is necessary to discuss the load balance at both the coarse-grained

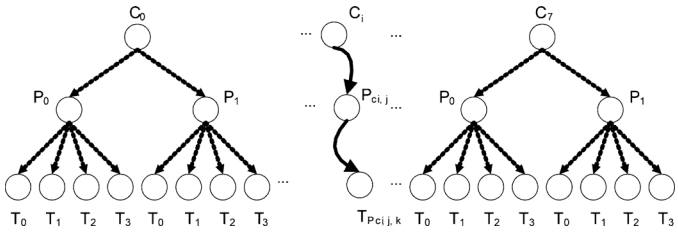


Fig. 2. Hierarchy of PUs on a Sun Niagara 2 chip. Each of the eight cores (C) on chip contains two pipelines (P), with four threads (T) in each pipeline. The solid arrows represent a scheduler-selected path.

and fine-grained levels. For the coarse-grained connection-level load balance, we simply need to consider the original HRW. The original HRW hash function g is a pseudorandom function that generates a random variable in $[0 \ldots 2^{31}-1]$ with independent uniform distribution. The load balance at the connection level is intuitively proved. For the fine-grained packet-level load balance, we should consider the impact of r_p. At the conceptual level, r_p fixes the load imbalance caused by HRW to maintain the connection locality, i.e., minimum DC. Thus, hash function h provides better load balance on top of g. As we pointed out in the previous proof, the value of r_p gradually approximates 1 after system warmup. If r_p is a constant multiplier, then $h(\vec{c}, p, r_p) = r_p \cdot g(\vec{c}, p)$ generates approximately random variables that are independent and uniformly distributed over $[0 \ldots r_p \cdot (2^{31}-1)]$. Thus, we can see AHRW as a theoretical load-balanced mapping model when r_p approximates to 1.

IV. Hierarchical AHRW Hash-Tree Scheduler (AHRW-Tree)

Although the proposed AHRW-based scheduler provides load balance and connection locality, it does not exploit the architectural locality. For example, the communication time between two threads that do not share the same cache will be much higher compared to when they share a cache. Also, the AHRW scheduler requires a nonnegligible amount of scheduling overhead, which delays the DPI processing and potentially causes undesirable packet drops. The mainstream multicore servers usually possess extensive parallelization and sharing of resources that naturally form a hierarchical structure. The highly threaded multicore chip Sun Niagara 2 usually has multiple hardware threads organized hierarchically, forming a core-pipeline-thread architecture, as shown in Fig. 2. Similarly, the Xeon server can be represented as a tree consisting of L2 caches at intermediate level and cores at the leaf level. Thus, a scheduler based on a blindly linear hash for all the PUs is not only inefficient due to the large candidate pool, but also unsuitable for a hierarchical multicore server due to the workload imbalance at each level of the parallelization. The theory behind the AHRW-tree scheduler is that traversal through a tree structure will guide the processing to the proper core, where the locality can be exploited. Also, the scheduling time is bounded by the depth of the tree (logarithmic), which is much shorter compared to the number of leaf nodes as in a linear search.

A. AHRW-Tree Scheduler Design

Hash-Tree Scheduler $s_{\text{tree}}(\vec{c}) \rightarrow p_L$*:*

$$s_{\text{tree}}(\vec{c}) \rightarrow p_L \Leftrightarrow \begin{cases} p_l = \arg \max_{d_L \in \{\text{leafs of node} P_{L-1} \text{at depth } lgN\}} h\left(\vec{c}, d_L, r_{d_L}\right) \\ p_1 = \arg \max_{d_1 \in \{\text{children of node } p_o \text{at depth } 1\}} h\left(\vec{c}, d_1, r_{d_1}\right) \\ p_0 = \arg \max_{d_0 \in \{\text{PUs at depth } 0\}} h\left(\vec{c}, d_0, r_{d_0}\right). \end{cases}$$

Given a hierarchical multicore architecture, we can apply the AHRW hash scheduler repeatedly along the traversal of the tree hierarchy. For nodes at the same depth of the tree, we can pick an internal node by the AHRW scheduler at that depth and continue the traversal from that node. The ultimate goal of the hash-tree scheduler is to select a candidate PU among the leave nodes by traversing through the tree

$$s_{\text{tree}}(\vec{c}) \rightarrow p_L \overset{\text{e.g. Niagara}}{\Longleftrightarrow} \begin{cases} p_L = T_{P_c} = \arg \max_{i \in \{\text{threads on } P_c\}} h\left(\vec{c}, i, r_i\right) \\ p_1 = P_c = \arg \max_{j \in \{\text{pipelines on } c\}} h\left(\vec{c}, j, r_j\right) \\ p_0 = c = \arg \max_{k \in \{\text{cores in set } c\}} h\left(\vec{c}, k, r_k\right). \end{cases}$$

As an example, for any given connection buffer and the SUN Niagara architecture, the hash-tree scheduler first picks a core c with the maximum weight generated by the AHRW at the core level. Then, we apply the AHRW at the pipeline level for the selected core c and pick a pipeline P_c. Finally, at the thread level, on the selected pipeline P_c, the AHRW picks the desired thread T_{P_c} with the maximum weight. We can use a three-dimensional array indexed by the core ID, pipeline ID, and thread ID, respectively. This data structure clearly manages the internal hierarchical relationship between each PU.

The properties of connection locality and packet-level load balance hold true at each level in the tree because the corresponding hash functions at each level are the same as the linear case. In addition to these two properties, the hash-tree scheduler also provides the following benefit.

Reduced Computation Cost: Suppose the complexity of HRW hash and the adjustment ratio computation is a constant H. The complexity of the original linear AHRW hash scheduler is $O(H \cdot N \cdot n)$ for n connection buffers and N PUs. On the other hand, the hash-tree scheduler selects a PU by a three-level tree traversal. Thus, with the same denotation, the complexity is reduced to $O(H \cdot \log N \cdot n)$. Note that the number of PUs N could be large if the user defines more threads (virtual PUs) than the number of physical PUs. While the hash space remains the same as $[0 \ldots 2^{31}-1]$, the hash-tree scheduler reduces the schedule space from N to $\log N$. Thus, the randomness of the hash function remains the same, but the size of the adjustment target set is reduced, leading to faster computation for the adjustments at each layer. In addition, the hash-tree scheduler essentially uses multiple hashes per key

input. This behavior not only reduces the possibility of hash collision, but also progressively improves the effect of load balance for the overall system performance.

As to this point, we have presented our scheduler in full detail. Recall that the hash values are used as a baseline to serve our Markov mesh model. In Section V, we will validate our model based on the performance measurement of our scheduler.

B. Hash Parameters

The major implementation issue of the AHRW-based hash-tree scheduler is how to provide a fast computable pseudorandom function $g(\vec{c}, p)$. In our experiment, we follow the definition of HRW hash function proposed in [35] as follows.

The HRW hash function:

$$g(\vec{c}, S_i) = (A \cdot ((A \cdot S_i + B) \text{ XOR } D(\vec{c})) + B) (\text{mod } 2^{31})$$

where A $=$ 1103515245 and $B = 12345$. $D(\vec{c})$ is a 31-bit digest of the object name $\vec{c}$, and S_i is the ID of the ith server in the cluster. This function generates a pseudorandom weight in the range $[0 \ldots 2^{31} - 1]$. In our case, the object name $\vec{c}$ is the 4-tuple header information of a connection buffer. Each S_i is represented by a PU ID. In the AHRW-based hash-tree case, S_i represents the cores, where $i \in [0, 7]$; the pipelines, where $i \in [0, 1]$; and the hardware threads, where $i \in [0, 3]$, respectively for the hash at each level of the tree.

V. Experimental Setup

A. Trace-Driven Offline Model for Linux L7-Filter Operations

Network traffic in the original L7-filter is captured by Netfilter, which consists of a set of hooks inside the Linux kernel that allows kernel modules to register callback functions with the network stack.

To concentrate on optimizing the pattern matching operation, we developed an offline trace-driven model in our study. We chose *libnids* [22] as a userspace module to read tcpdump trace files and simulated kernel network stack behaviors in userspace. In the real world, packet arrival and pattern matching operations are tightly coupled. However, in our study, we used an offline trace input to replace the handling of network packet arrival. Once a packet is processed by *libnids*, L7-filter classifies the packet following the steps described in Fig. 3. Packets are fed into the system at the optimal speed (as in a TCP connection with no packet drop).

This decoupled model has the following advantages.

1) It frees us from dealing with complex and corner case operations in the lower-layer networking and kernel stacks so that we can concentrate on optimizing the hotspot pattern matching operations.
2) It provides repeatable and well-controlled research environment, enabling testing and validation on various approaches.
3) It also allows us to simulate and measure L7-filter performance on reliable connections without any packet loss or retransmission.

The baseline userspace sequential L7-filter is of version 0.6 with protocol definition updated by May 19, 2009. Because the original L7-filter was written for Linux OS, we make some changes in the Makefile and header files to direct the program to link to the corresponding libraries in Solaris.

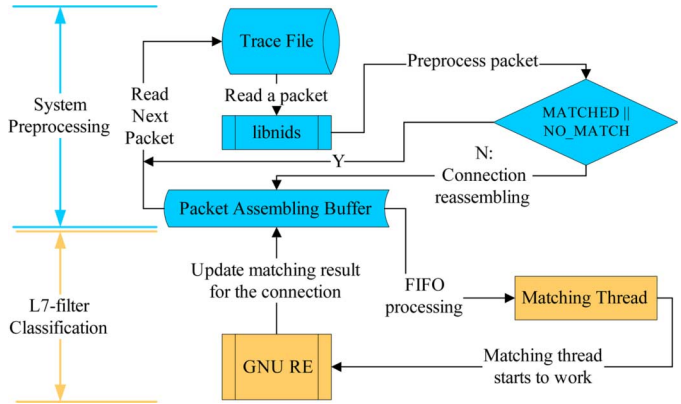


Fig. 3. Trace-driven L7-filter data flow.

B. Traces for the Experiment

In this paper, we adopt the same trace driven model proposed in [11]. The decoupled model proposed in that work separates the packet processing from the pattern matching operations at the application layer. We choose the most recent version 1.23 libnids [22] as the preprocessing component, which parses the 4-tuple information in the incoming packet and places it into the corresponding entry in the connection reassembling buffer.

In regards to the packet trace, we selected the traces that were used for L7-filter classification in previous publications [2], [21] and that we could get hold of. The traces that we selected are representative of different connection lengths, packet sizes, and packet rate. Our trace-driven model feeds packets as fast as they can be processed regardless of the interpacket latency in the packet trace. Thus, the original packet rate in the trace does not matter for the evaluation of our scheduler. Our throughput is determined by the packet processing rate instead of the original rate of the traces.

We would like to point out here that previous papers on HRW have only considered synthetic workload and simulated using a generic multiprocessor model (M/M/m) without considering any particular application or architecture. We therefore chose L7-filter as a real application and actually implemented on a multicore architecture SUN Niagara 2 chip to obtain experimental results.

In our experiment, we used three different packet trace files: 1) a 4-h tcpdump file from the Massachusetts Institute of Technology, Cambridge ("MIT") [2], [21], [26]; 2) a tcpdump file from Tsinghua University, Beijing, China, which records a section of 4 min and 19 s Internet traffic ("TU"); and 3) a segmentation of tcpdump from New York Polytechnic University, New York ("NYP"). The key features of the traces are summarized in Table I. The "Conn. Length" column shows the average number of packet in a connection. The "Distro. Disparity" column shows the degree of difference of the number of packets among different connections. As this value increases, we see the disparity among the number of packets in the connections varies more in the trace file.

TABLE I
KEY FEATURES OF THE TRACE FILES

Trace Name	# of Pkt.	# of Conn.	Conn. Length	Distro. Disparity	Trace Size
MIT	340K	40K	8.5	Medium	286MB
TU	1.32M	110K	12	Large	1GB
NYP	590K	61K	9.6	Small	500MB

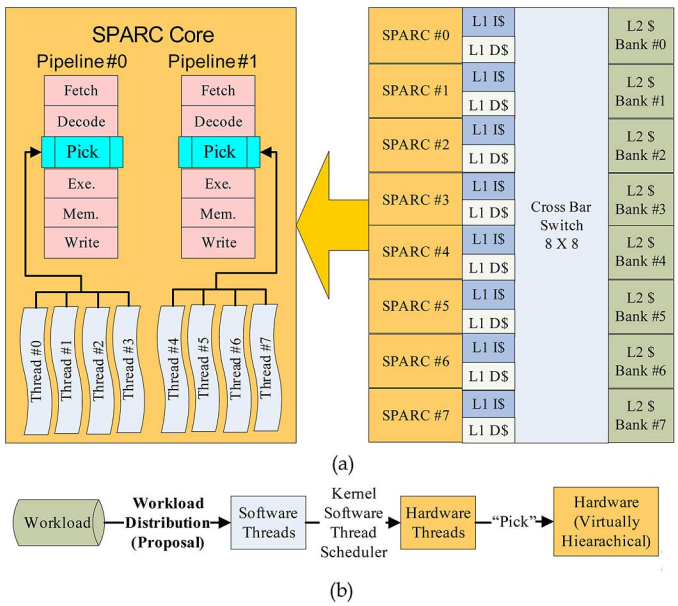


Fig. 4. (a) Sun Niagara 2 Chip architecture and the parallelism inside each SPARC core. (b) Scheduler topology in the Niagara 2-Solaris system.

C. Multicore Platform

In the experiments, we used a Sun-Niagara-2-based T5120 server as our testbed. The hierarchical processor architecture contains eight in-order cores (1.2 GHz). Each of the eight cores embeds two independent integer pipelines that enable real multithreading without causing resource contention. Each pipeline is shared by four hardware threads, totaling 64 hardware threads in the system. The eight cores are connected to share a 4-MB L2 cache through an 8×8 crossbar switch. Our testbed server installs 16 GB of 667-MHz DDR2 memory. Fig. 4(a) illustrates the system architecture of a Sun Niagara 2 processor.

We use Solaris 10 as our default OS. Fig. 4(b) demonstrates the scheduler topology in the Niagara 2-Solaris system architecture. The kernel software thread scheduler spreads software threads first across cores, one thread per core until every core has one, then two threads per core until every core has two, and so on. Within each core, the kernel software thread scheduler balances the software threads onto the eight hardware threads on the core's two integer pipelines [31].

However, neither of these two schedulers distributes the incoming network traffic to the software thread. This kind of scheduling is defined in the application by the programmer. A round-robin distribution of the workload to the software threads is a common and simple default implementation. The scheduler proposed in this paper belongs to this category. The hierarchical architecture of the Niagara 2 is a virtual organization of the software threads. In order to avoid the influence of the kernel software thread scheduler, we use a system call (processor_bind) to affinitize each software thread to a hardware thread. By doing this 1-to-1 pinning, we can focus on the scheduling of workload distribution at the software level.

VI. PERFORMANCE RESULTS

In this section, we present the experimental results using different schedulers. The performance evaluation verifies the benefits of the proposed optimizations. In our experiments, we provide measurements from a real machine rather than simulators.

We compare the AHRW hash-tree scheduler to the following: 1) pure connection locality technique proposed in [8]; 2) pure HRW hash function that provides connection locality and load balance at the connection level; 3) our prototype AHRW scheduler, which is more efficient compared to the idea proposed in [19]; and 4) the hierarchical AHRW-based hash tree scheduler (AHRW-tree).

Throughput is a direct reflection of any packet processing system. We calculate the throughput in our system by dividing the overall packet length (bytes) by the execution time of our trace driven model. For system utilizations, we present results for physical core utilization (using a Perl script "corestat"). We additionally profile the life of a packet in the system to illustrate the overhead of scheduling versus the cost of pattern matching.

A. System Throughput for Connection-Based Scheduling

Here, we show that the applicability of a purely connection-based scheduling technique may vary depending on the particular multicore architecture. Previously, we have introduced a connection locality with cache affinity-based scheduler for L7-filter on a general-purpose Intel Xeon server [11]. We modified the cache affinity to be replaced by thread affinity, but observed that the benefits of connection locality are offset by two major challenges on highly threaded hierarchical multicore server Sun Niagra 2.

Fig. 5(a) shows the L7-filter system throughput as a function of the number of threads. We plot results for four existing scheduling policies.

1) "pckt+os" is the default setup without any optimization.
2) "conn+affinity" applies the connection locality and thread affinity optimizations proposed in [11].
3) "conn+os" substitutes the thread affinity option to use the default Solaris kernel software thread scheduler, which redistributes the threads to pipelines on different cores to improve load balancing. This is because two pipelines share the same core and each core has its own first-level cache.
4) "ideal" is the ideal throughput based on a linear expectation to the number of independent processing units.

The throughput can only be increased at most by a factor of $10.1\times$ ("conn+os"-64 versus "pckt+os"-1) rather than the ideal $16\times+$. Note that we conservatively choose $16\times$ to be the maximum speedup for "ideal" because the 64 threads only share 16 pipelines. Fig. 5(b) illustrates the imbalanced system utilization at each level in the Niagara 2 system. The cross on each vertical bar shows the average utilization (%) at each level in the core topology; the lines represent the range of peak high

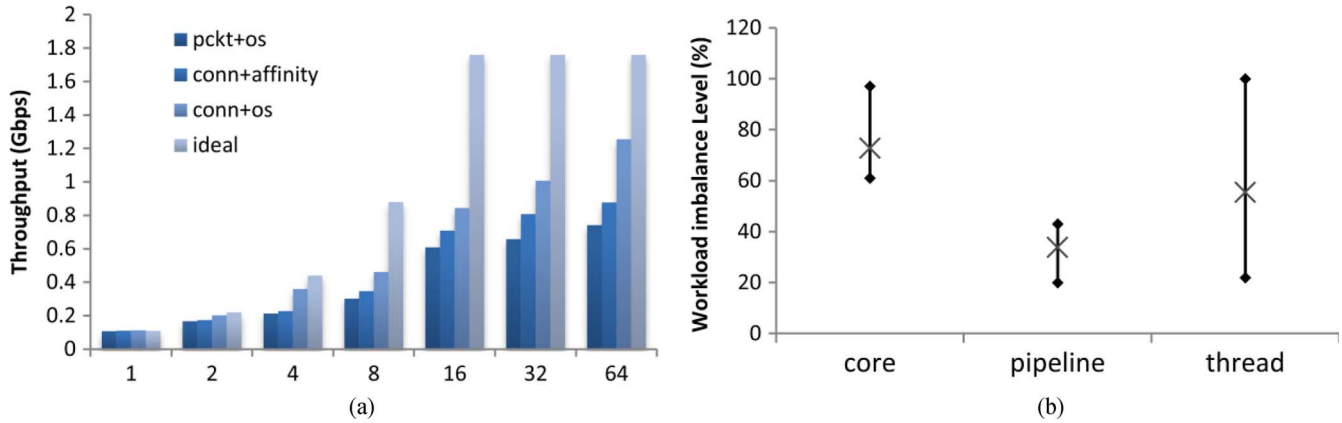


Fig. 5. L7-filter performance on a Sun Niagara 2 chip. (a) Throughput inefficiency. (b) Workload imbalance at different levels (%).

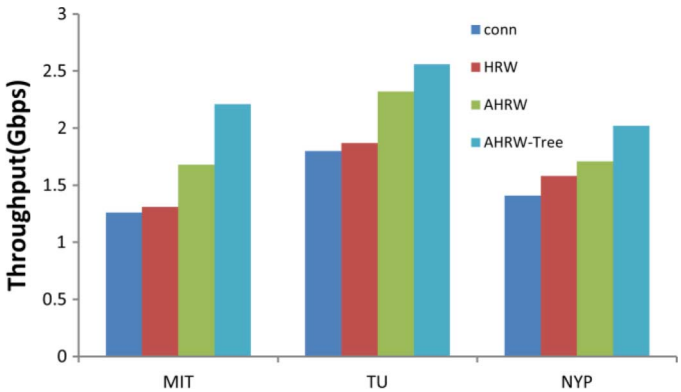


Fig. 6. System throughput comparison using different schedulers for all the traces.

and peak low values (%). This observation raises the concern of load balancing in addition to cache locality.

A highly threaded hierarchical multicore server suffers from accumulative workload imbalance when connection locality is applied. The hierarchical Sun Niagara 2 multicore processor features 64 hardware threads on 16 independent pipelines across eight SPARC cores. Each pipeline is shared between four threads, which needs proper distribution of software threads to balance the workload at the pipeline level. Maintaining connection locality sacrifices the fairness in workload scheduling when packet distribution is nonuniform. In an extreme case, there may be more cores than connections giving rise to some idle servers. A load-balanced system should be able to use all the cores by relaxing the connection locality. It is clear from Fig. 5 a that the Solaris OS (conn+os) performs better than (conn+affinity) by considering load balancing among threads by placing them in appropriate cores because of its knowledge of the underlying architecture. The problem is how to balance the tradeoff between the connection locality and load balance to maximize the throughput.

B. System Throughput for HRW Based Schedulers

Fig. 6 compares the throughput of the baseline connection-based scheduler and various versions of the HRW scheduler. It is observed that the AHRW-tree scheduler improves the system throughput by an average of 50% compared to the connection locality alone ("conn") [11], 20% compared to the single-layer AHRW scheduler ("AHRW"), and 22% compared to the baseline HRW. We also find that for different traces, the system throughput increases as the average connection length increases. This is because L7-filter only processes the first eight packets in a connection. When the connection length is large, more packets could be directly marked by L7-filter without going through pattern matching. Another observation is that the AHRW-tree scheduler is more efficient for a larger disparity between the connection distribution and packet distribution. Specifically, we saw the AHRW and AHRW-tree schedulers have the best results for the TU trace. Recall in Table I that the number of packets in each connection in the TU traces varies the most.

C. Life-of-Packet Analysis

Here, we decompose the processing of L7-filter to different components to study the individual contribution. For the rest of the paper, unless otherwise noted, we show the results for only one trace (TU) due to space limit. The execution time for each experiment is scaled to 100% to better represent the fractional contribution. Note that all the measurements in Figs. 7 and 8 are based on the lifetime of one packet rather than the complete trace file because of the timing overlap in PT and MT. While the PT runs the *libnids* routines, it also dispatches packets to the proper MT runqueue. In the meantime, the desired MT is also classifying connections in its runqueue in a first-in–first-out (FIFO) manner. Therefore, it is necessary to use per-packet profiles to explain the interrelations among different components in the system. We present the average values obtained from all the three trace files.

Fig. 7 shows the contribution of the major processing components. From the point a packet arrives in the system to the packet being processed, we divide the processing of a packet into: 1) reading trace file from the disk ("disk I/O"); 2) libnids stack operations ("TCP/IP"); 3) buffer management including reassemble packet buffers ("buffer"); 4) scheduling cost ("scheduler"); and 5) pattern matching ("MT"). We observe that the overhead of the scheduler is permissible (under 9% on average), even in the more complicated AHRW hash-tree scheduler. The overhead in AHRW hash-tree scheduler is greater than the single-layer AHRW by only a negligible amount. We also observe a decreased timeshare of MT execution time when more

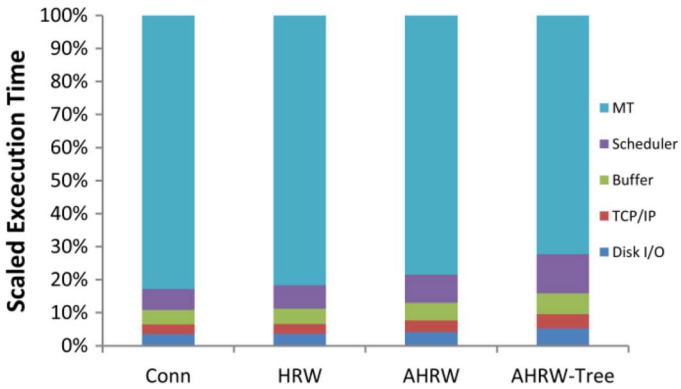


Fig. 7. Life-of-a-packet analysis for the TU trace.

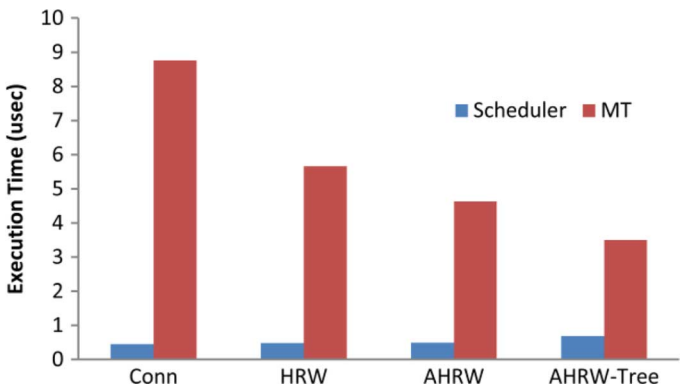


Fig. 8. Absolute execution time comparison between the scheduler and matching thread for the TU trace.

optimizations are applied. A large timeshare for MT in Fig. 7 means it either gets more opportunity for connection classification, or it sleeps frequently because it runs so fast that its runqueue is empty very often. Fig. 8 shows that for each packet, it always takes longer to match than scheduling, which dismisses the chance of MT sleeping. Therefore, we conclude that the system distributes more time for pattern matching in MT and reduces the latency of scheduling stall. This observation is in line with the throughput results demonstrated in Fig. 6.

D. Scheduler Overhead

Fig. 8 shows the scheduler overhead in terms of the absolute execution time for a packet. Clearly, each matching thread runs much longer than the scheduler does, even though the scheduling time increases as we move from simple scheduling strategy (connection-based) to complex AHRW-tree scheduler. Therefore, the reduced MT execution percentage in Fig. 7 is due to the reduction in MT execution time than the increased scheduler cost (from 0.49 to 0.57 μs). This observation verifies our theoretical analysis in Section III. The average per-packet execution time for the MTs is reduced because workloads are more balanced on the available threads. Also, the execution time is reduced because of the cache hits and preservation of locality while allocating threads to the pipelines in the Sun Niagara architecture. The workload balance also reduces blocking time by scheduling those connection buffers from a deeper location in a busy thread to a relatively free thread, hence increasing the overall system throughput.

E. Load Balance

Fig. 9 verifies that AHRW-tree scheduler provides the best load balance for all the three traces. The star on each vertical bar represents the average system utilization, and the vertical bars represent the range (min-max) of the system utilization. With AHRW-tree, the load imbalance among all the cores is reduced from 89% to 7%. As the distribution disparity increases, the benefits of our scheduler become more pronounced. Specifically, the TU trace, in Fig. 9(b), has the best load balancing among all the traces. This result shows that our scheduler can efficiently balance the uneven workload across the multicore platform. Another interesting observation is that as the workload becomes more evenly distributed among the PUs, the system utilization increases. This is because PUs have less opportunity to be idle in a more balanced environment. As a result, more PU time is dedicated to the DPI processing, leading to higher system throughput as presented in Fig. 6.

We also present the runqueue length at the thread level in Fig. 10 to directly illustrate the changes in workload balance. As we can see from Fig. 10, it is quite straightforward that the runqueue length becomes much smoother when our scheduling optimizations are applied. Another observation from the same figure shows the average runqueue length of the thread decreases as we further optimize our scheduler. This observation means the overall matching time is reduced in the system, which is in line with the observation in Fig. 9.

F. Comparison to Theoretical Adjustment

Recall that, in theory, we should use (1) to derive the weight on each thread in our scheduler, and its hierarchical heuristic is a tradeoff between effectiveness and efficiency. In this section, we compare the cost-effectiveness of AHRW and the theoretical solution. Again, we only show the results for the TU trace due to space limit.

In Fig. 11(a), we show that AHRW-tree achieves a 3% improvement in system throughput compared to the theoretical adjustment using (1). This is quite surprising because the theory derivation provides a more accurate weight calculation, which guarantees perfect load balancing. In Fig. 11(b) and (c), we further investigate the load balancing and overhead comparison. As expected, the theoretical method balances the workload better than our simplified version. However, as we have shown in the previous section, the AHRW-tree already achieves good load balancing performance (7% imbalanced workload). Therefore, this gain from the accurate derivation is negligible. In addition, when we separate the scheduling time from the normal execution time for each packet, we find that the scheduler using theoretical adjustment incurs 50% more time on the calculation inside the scheduler. In conclusion, the AHRW and AHRW-tree are cost-effective alternatives to the theoretical adjustment for robust hash algorithms.

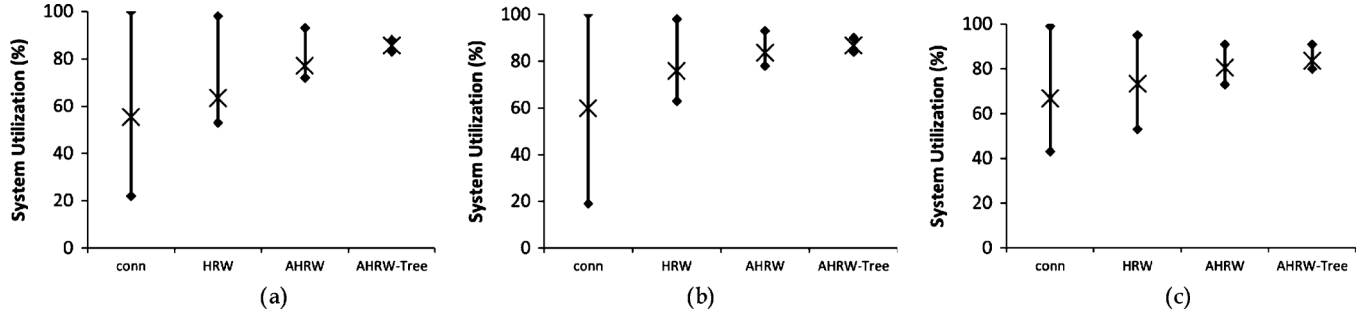


Fig. 9. System utilization (%): (a) MIT; (b) TU; (c) NYP.

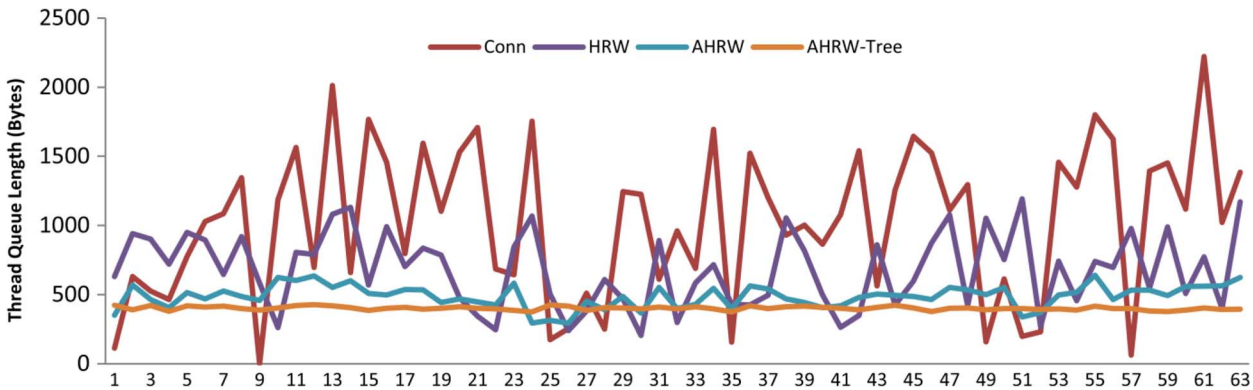


Fig. 10. Runqueue length on all the 63 matching threads for the TU trace. Note that thread #0 runs the preprocessing thread exclusively.

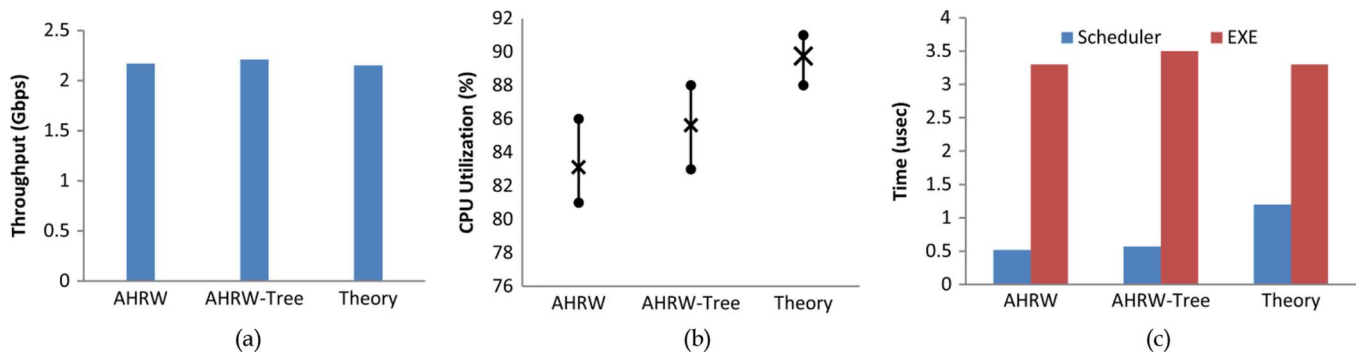


Fig. 11. Comparison to theoretical adjustment for the TU trace.

VII. Discussion on the Scope of Our Scheduler

In this paper, we proposed a hierarchical scheduler for L7-filter on a multicore CPU with extensive parallel thread resources. In this section, we would like to discuss the potential applicability of the scheduler on other network applications and other CPU architectures.

A. Applicability to Other Applications

A key observation of scheduling in L7-filter is the interdependency between packets belonging to the same connection. In addition to the packet headers (source and destination IP addresses, source and destination port numbers, etc.) that are shared by all the packets, L7-filter classifies a buffer of packets. The minimum scheduling unit is not just one packet or one connection, but a connection buffer filled with several packets. During the scheduling, the in-flight connections can be scheduled to different cores depending on the workload on each core.

Fig. 12 summarizes the multicore scheduling characteristics of our L7-filter system. It can be interpreted as a generic model for a multicore scheduler for packet processing—any application with a packet-connection interdependency can potentially be scheduled using our proposed scheduler. All the incoming request streams are first stored in a global queue in the FIFO order. Each stream consists of many buffers of packets, which are the minimal scheduling units and can be scheduled to any core independently. For each scheduling cycle, the scheduler fetches a buffer from the FIFO global queue, makes the scheduling decision, and then dispatches the buffer into the local queue of the scheduled core for processing.

All streaming applications including multimedia FFmpeg transcoding falls into this category. Some scheduling techniques of multimedia packets on a workstation cluster were presented in [13], where the QoS aspects were emphasized. This paper develops a hash-based scheduling technique consid-

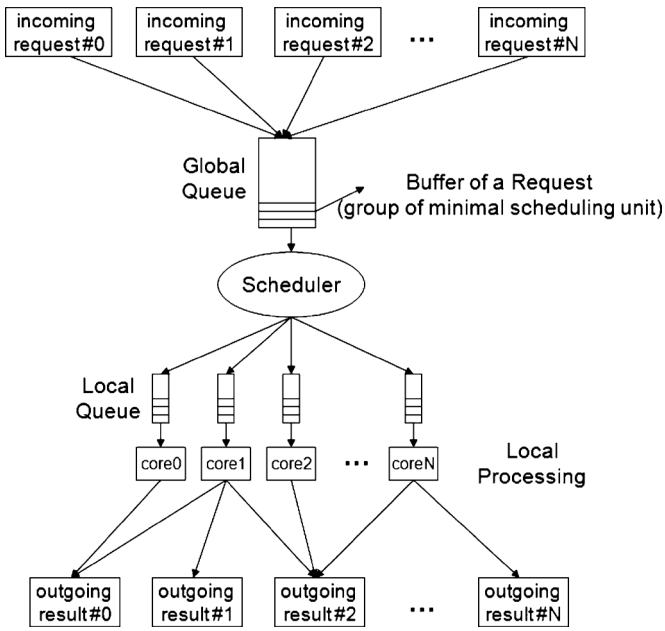


Fig. 12. Generic model of the scheduling process.

ering the cache and thread locality in a multicore architecture. Recently, we developed a QoS-based HRW scheduler for FFmpeg application [12]. Compared to L7-filter, FFmpeg has the same hierarchy—we just need to substitute "packet buffer" with "GOP" and "request stream" with "stream". Each core runs a transcoding thread, which iteratively fetches a GOP from its local queue and executes the task. In fact, the technique here is applicable to any packet processing system.

B. Applicability to Other Architectures

By applying a scheduler at each level of the topology, it is expected to achieve the global optimal scheduling decision. In Fig. 2, we have illustrated the hierarchical AHRW scheduler and how it works on the Sun Niagara 2 CPU. With some minor changes of the scheduler at each layer, we can easily port our scheduler onto servers with different CPU architectures. Our technique is also applicable to a modern multicore CPU that generally has a hierarchical core/cache topology. For example, Intel's Xeon series (Nehalem, Westmere, Sandy Bridge, etc.) all possess a hierarchical cache topology, although the detailed architectures could be different for different models. The cache topology saves hardware area by sharing last-level cache among different core groups. It also leaves potential optimization opportunities to fully utilize the cache topology. We have presented the results for a QoS-aware hierarchical scheduler on an Intel Xeon server in a recent paper [12]. Finally, the scheduler can be extended to any cluster-based multiprocessors, where the CPUs inside a cluster have much lower communication cost compared to the intercluster communication. Basically, we are exploiting the locality in communication while scheduling the packets of the same connection instead of distributing them randomly.

VIII. Conclusion and Future Work

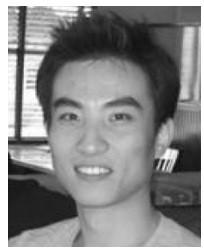
In this paper, we designed a multithreaded L7-filter algorithm to execute on an off-the-shelf multicore architecture. Our parallelized L7-filter design can efficiently explore the resources on a highly threaded hierarchical multicore server, such as Sun Niagra 2. We first showed that the benefits of pure connection locality-based scheduling are offset by load imbalance in the server. After thorough analysis, we found that the load balance should be enforced at the packet level rather than connection level. We proposed an adaptive hash-based scheduler AHRW using runqueue length to strike a balance between connection locality and load balancing at the packet level. We also designed a Hierarchical AHRW (HAHRW) scheduler to be applied recursively in the hierarchical CPU architecture at the core, the pipeline and the thread level, respectively. The hierarchical design reduces the scheduling complexity to $O(\log N)$ from $O(N)$, where N is the number of processors. Our experimental results showed that the AHRW-tree scheduler can improve the L7-fitler throughput by about 50% compared to the existing algorithms. We also described how our scheduler can be easily extended to other applications and other architectures.

We plan to deploy our L7-filter software on the Cisco Unified Computing System (UCS). The optimized L7-filter can provide fast deep packet inspection function in the control plane of a UCS chassis, where QoS ability is of great importance for client deployment in virtualization. We believe our design and implementation can provide insights to the DPI solutions in networking appliances. In the future, we also plan to design a generalized hash function considering various communication topologies.

References

[1] "Application layer packet classifier for Linux (L7-filter)," 2009 [Online]. Available: http://l7-filter.sourceforge.net
[2] N. Borisov, D. Brumley, H. J. Wang, J. Dunagan, P. Joshi, and C. Guo, "A generic application-level protocol analyzer and its language," in *Proc. NDSS*, 2007, pp. 216–231.
[3] B. C. Brodie, R. K. Cytron, and D. E. Taylor, "A scalable architecture for high-throughput regular-expression pattern matching," in *Proc. ISCA*, 2006, pp. 191–202.
[4] "Cavium network lectures," Cavium, 2010 [Online]. Available: http://university.caviumnetworks.com/download_center.html#lectures
[5] A. Chandra and P. Shenoy, "Hierarchical scheduling for symmetric multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, no. 3, pp. 418–431, Mar. 2008.
[6] H.-L. Chen, J. R. Marden, and A. Wierman, "On the impact of heterogeneity and back-end scheduling in load balancing designs," in *Proc. IEEE INFOCOM*, 2009, pp. 2267–2275.
[7] "Cisco Internetworking Operating System (IOS) IPS deployment guide," Cisco, San Jose, CA, 2011 [Online]. Available: http://www.cisco.com/en/US/products/ps6601/products_ios_protocol_group_home.html
[8] "Cisco SCE 2000 series service control engine," Cisco, San Jose, CA, 2011 [Online]. Available: http://www.cisco.com/en/US/products/ps6151/
[9] "Application-oriented networking: Products and services," Cisco, San Jose, CA, 2008 [Online]. Available: http://www.cisco.com/en/US/products/ps6692/Products_Sub_Category_Home.html
[10] D. Guo, G. Liao, L. N. Bhuyan, and B. Liu, "An adaptive hash-based multilayer scheduler for L7-filter on a highly threaded hierarchical multicore server," in *Proc. ACM/IEEE ANCS*, 2009, pp. 50–59.
[11] D. Guo, G. Liao, L. N. Bhuyan, B. Liu, and J. Ding, "A scalable multithreaded L7-filter design for multi-core servers," in *Proc. ACM/IEEE ANCS*, 2008, pp. 60–68.

[12] D. Guo and L. Bhuyan, "A QoS aware multicore hash scheduler for network applications," in *Proc. IEEE INFOCOM*, Shanghai, China, 2011, pp. 1089–1097.
[13] J. Guo and L. Bhuyan, "Load balancing in a cluster-based web server for multimedia applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 11, pp. 1321–1334, Nov. 2006.
[14] N. Hua, H. Song, and T. V. Lakshman, "Variable-stride multi-pattern matching for scalable deep packet inspection," in *Proc. IEEE INFOCOM*, 2009, pp. 415–423.
[15] "MSCG hierarchical DPI solution," Huawei, Shenzhen, China, 2010 [Online]. Available: http://www.huawei.com/products/datacomm/catalog.do?id=1219
[16] S. Jagabathula, V. Doshi, and D. Shah, "Fair scheduling through packet election," in *Proc. IEEE INFOCOM*, 2008, pp. 301–305.
[17] R. Jain and S. A. Routheir, "Packet trains—Measurements and a new model for computer network traffic," *IEEE J. Sel. Areas Commun.*, vol. 4, no. 6, pp. 986–995, Sep. 1986.
[18] "Juniper M series multiservice edge routers," Juniper Networks, Sunnyvale, CA, 2011 [Online]. Available: http://www.juniper.net/us/en/local/pdf/datasheets/1000042-en.pdf
[19] L. Kencl and J.-Y. Le Boudec, "Adaptive load sharing for network processors," *IEEE/ACM Trans. Netw.*, vol. 16, no. 2, pp. 293–306, Apr. 2008.
[20] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in *Proc. ACM SIGCOMM*, 2006, pp. 339–350.
[21] Z. Li, G. Xia, H. Gao, Y. Tang, Y. Chen, B. Liu, J. Jiang, and Y. Lv, "NetShield: Massive semantics-based vulnerability signature matching for high-speed networks," in *Proc. ACM SIGCOMM*, New Delhi, India, 2010, pp. 279–290.
[22] R. Wojtczuk, "libnids," 2010 [Online]. Available: http://libnids.sourceforge.net/
[23] A. X. Liu, C. R. Meiners, and Y. Zhou, "All-match based complete redundancy removal for packet classifiers in TCAMs," in *Proc. IEEE INFOCOM*, 2008, pp. 111–115.
[24] J. W. Lockwood, N. Naufel, J. S. Turner, and D. E. Taylor, "Reprogrammable network packet prReprogrammable network packet processing on the field programmable port extender FPXocessing on the field programmable port extender FPX," in *Proc. ACM FPGA*, Feb. 2001, pp. 87–93.
[25] H. McGhan, "Niagara 2 opens the floodgates—Niagara 2 design is the closest thing yet to a true server on a chip," *Microprocess. Rep.*, 2006, 11/6/06-01.
[26] "MIT DARPA intrusion detection data sets," MIT Lincoln Laboratory, Lexington, MA, 2000 [Online]. Available: http://www.ll.mit.edu/IST/ideval/data/2000/2000_data_index.html
[27] A. Mitra, W. Najjar, and L. Bhuyan, "Compiling PCRE to FPGA for accelerating SNORT IDS," in *Proc. ACM/IEEE ANCS*, 2007, pp. 127–136.
[28] I. Papaefstathiou and V. Papaefsthathiou, "Memory-efficient 5D packet classification at 40 Gbps," in *Proc. IEEE INFOCOM*, 2007, pp. 1370–1378.
[29] "Receive Side Scaling (RSS)," Microsoft, Redmond, WA, 2008 [Online]. Available: http://www.microsoft.com/whdc/device/network/NDIS_RSS.mspx/
[30] K. W. Ross, "Hash routing for collections of shared web caches," *IEEE Netw.*, vol. 11, no. 6, pp. 37–44, Nov.–Dec. 1997.
[31] S. Sistare, "The UltraSparc T2 processor and the Solaris operating system," Oracle, Redwood Shores, CA, Oct. 09, 2007 [Online]. Available: http://blogs.sun.com/sistare/entry/the_ultrasparc_t2_processor_and
[32] R. Smith, C. Estan, S. Jha, and S. Kong, "Deflating the big bang: Fast and scalable deep packet inspection with extended finite automata," in *Proc. ACM SIGCOMM*, 2008, pp. 207–218.
[33] "SNORT network intrusion detection system," Sourcefire, Columbia, MD [Online]. Available: http://www.snort.org/
[34] L. Tan and T. Sherwood, "A high throughput string matching architecture for intrusion detection and prevention," in *Proc. ISCA'05*, 2005, pp. 112–122.
[35] D. E. Taylor and J. S. Turner, "ClassBench: A packet classification benchmark," in *Proc. IEEE INFOCOM*, 2005, vol. 3, pp. 2068–2079.
[36] D. G. Thaler and C. V. Ravishankar, "Using name-based mappings to increase hit rates," *IEEE/ACM Trans. Netw.*, vol. 6, no. 1, pp. 1–14, Feb. 1998.
[37] B. Veal and A. Foong, "Performance scalability of a multicore Web server," in *Proc. ACM/IEEE ANCS*, 2007, pp. 57–66.
[38] J. Verdu, M. Nemirovsky, and M. Valero, "MultiLayer processing—An execution model for parallel stateful packet processing," in *Proc. ACM/IEEE ANCS*, 2008, pp. 79–88.
[39] T. Y. C. Woo, "A modular approach to packet classification: Algorithms and results," in *Proc. IEEE INFOCOM*, 2000, vol. 3, pp. 1213–1222.
[40] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *Proc. ACM/IEEE ANCS*, 2006, pp. 93–102.



Danhua Guo received the B.S. degree in computer science from Beijing University of Technology, Beijing, China, in 2005, and the Ph.D. degree in computer science from the University of California, Riverside, in 2010.

He has been with Microsoft Corporation, Mountain View, CA, since 2011. Previously, he was a System Engineer with the Server Access and Virtualization BU, Cisco Systems, Inc., San Jose, CA, where he worked on the performance evaluation and feature development for Ethernet drivers on the next-generation virtual network adapter for Cisco's UCS systems. His research interests include server architecture for fast network I/O processing, with focus on deep packet inspection (DPI). He has also been working on both native and virtualized OS scheduling on multicore architecture.

Laxmi Narayan Bhuyan (S'81–M'82–SM'87–F'98) received the Ph.D. degree in computer engineering from Wayne State University, Detroit, MI, in 1982.

He is a Distinguished Professor and Chairman of the Computer Science and Engineering Department with the University of California, Riverside (UCR). He has also worked as a consultant to Intel and HP Labs. His current research interests are in the areas of network computing, multiprocessor architectures, router and Web server architectures, parallel and distributed processing, and performance evaluation. He has published more than 150 papers in these areas in the IEEE TRANSACTIONS ON COMPUTERS, IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, the *Journal of Parallel and Distributed Computing*, and many refereed conference proceedings.

Prof. Bhuyan is a Fellow of the Association for Computing Machinery (ACM) and the American Association for the Advancement of Science (AAAS).

Bin Liu (M'03) received the M.S. and Ph.D. degrees in computer science and engineering from Northwestern Polytechnical University, Xi'an, China, in 1988 and 1993, respectively.

He is a Professor with the Computer Science and Technology Department, Tsinghua University, Beijing, China. He is currently leading the Lab of Broadband Network Switching Technology and Communication, Tsinghua University. His research interests include parallel computing architecture, high-performance switches/routers, traffic measurement and management, and high-speed network security.