An Adaptive Hash-based Multilayer Scheduler for L7-filter on a Highly Threaded Hierarchical Multi-Core Server

Danhua Guo[1], Guangdeng Liao[1], Laxmi N. Bhuyan[1] and Bin Liu[2]

[1]University of California, Riverside
900 University Ave.
Riverside, CA 92521
{dguo, gliao, bhuyan}@cs.ucr.edu

[2]Tsinghua University
Beijing 100084
P.R. China
liub@tsinghua.edu.cn

ABSTRACT

Ubiquitous multi-core-based web servers and edge routers are increasingly popular in deploying computationally intensive Deep Packet Inspection (DPI) programs. Previous work has shown the benefits of connection locality-based scheduling on multi-core servers to improve L7-filter performance. However, we show that highly threaded hierarchical multi-core processors, such as the Sun Niagara 2 processor, accumulate imbalanced workload at each resource layer. This workload imbalance potentially offsets the benefits from connection locality. In addition, connection-locality-based load balance fails to work when network traffic is unevenly distributed.

In this paper, we propose an adaptive hash-based multilayer scheduler for a highly threaded hierarchical Sun Niagara 2 server. Our scheduler maintains connection locality and adaptively adjusts the scheduling to balance the real time workload. The original Highest Random Weight (HRW) hash guarantees the connection locality but only balances the workload over the number of different connections. We enhance the original single layer HRW into a hierarchical "hash tree" scheduler to balance the connection workload in accordance with the hierarchical processor architecture. We then optimize our multilayer scheduler to adaptively adjust scheduling decisions based on service time at each level, further improving the system load balance. Our scheduler is shown to increase the system throughput by 59.2% compared to the previously proposed connection locality optimization.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ANCS'09, October 19–20, 2009, Princeton, New Jersey, USA.

Copyright 2009 ACM 978-1-60558-630-4/09/0010...$10.00.

Keywords

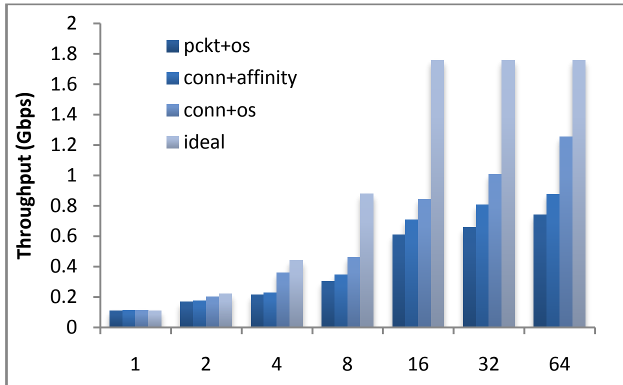
Connection Locality, Deep Packet Inspection, L7-filter, Load Balance, Multicore, Multithreading, Packet Classification, Parallelism, QoS, Scalability, Scheduling.

1. INTRODUCTION

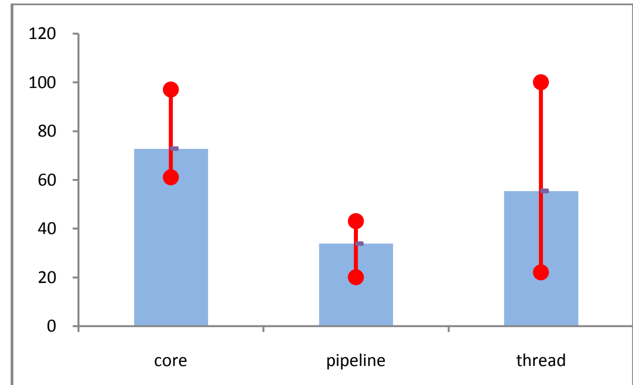
The prevalence of multi-core chips in scientific computations has enlightened researchers in the network domain to use these chips to bridge the gap between the ever-increasing network bandwidth and the relatively unfulfilling processing speed. In the network QoS domain, Deep Packet Inspection (DPI) is an important functionality in most of the major routers/switches on the market [3, 9, 11]. Maintaining connection locality has been proven beneficial in improving the performance of DPI programs, e.g. L7-filter, on some multi-core-based web servers [7]. For L7-filter, the classification of one connection might require multiple connection buffers (with different number of packets). Therefore, maintaining connection locality warms up the cache with reusable packet data, so that future classifications of the same connection benefit from the data in the local cache without accessing the remote memory, which is an order of magnitude more expensive. However, the benefits of connection locality are offset by two major challenges on highly threaded hierarchical multi-core servers.

First of all, a highly threaded hierarchical multi-core server suffers from accumulative workload imbalance when connection locality is applied. The hierarchical Sun Niagara 2 multi-core processor features 64 hardware threads on 16 independent pipelines across 8 SPARC cores. We show in Fig. 1(a) that with all the 64 threads enabled, the L7-filter system throughput can only be increased at most by a factor of 10.1X ("conn+os"-64 VS "pckt+os"-1) rather than the ideal 16X+. Note that we conservatively choose 16X to be the maximum speedup for "ideal" because the 64 threads only share 16 pipelines. Fig. 1(b) illustrates the imbalanced system utilization at each level in the Niagara 2 system. Therefore, how to schedule the extensive thread resource more efficiently on such a multi-core chip becomes a major concern in scheduler designs.

Secondly, maintaining connection locality sacrifices the fairness in workload scheduling when packet distribution is disproportional to the connection distribution. Connection locality-based load balance guarantees that each core shares a similar number of different connections. But if the network traffic is unevenly distributed, packets in some connections might



(a) Throughput inefficiency



(b) Workload imbalance at different levels (%)

Fig. 1 L7-filter performance on a Sun Niagara 2 chip. (a) "pckt+os" is the default set up without any optimization; "conn+affinity" applies the connection locality and thread affinity optimizations proposed in paper [7]; "conn+os" substitutes the thread affinity option to use the default Solaris kernel software thread scheduler, which is discussed in the section 2.2. "ideal" is the ideal throughput based on a linear expectation to the number of independent processing units. (b) The bars show the average utilization (%) at each level in the core; the lines represent the range of peak high and peak low values (%).

outnumber those in the others and therefore causes a jam on the core where the connections with more packets are affinitized. In an extreme case, if there are more cores than the number of different connections being processed at a certain point in the system, a load balanced system should be able to use all the cores by relaxing the connection locality instead of wasting the idling cores and blocking the busy ones. The problem is now clear: how to balance the trade-off between the maintenance of connection locality and load balance, subject to throughput constraint.

In this paper, we propose an adaptive hash-based multilayer scheduler that relaxes connection locality for load balance based on real time statistical feedback. We choose Highest Random Weight (HRW) [28] as the baseline hash function, which intuitively guarantees load balance over the request space, i.e. in our case, the number of different connections. We implement HRW at all the three levels: the core, the pipeline and the thread, respectively, corresponding to the hierarchical architecture on the Niagara 2 chip. Our intention is to balance the workload progressively. However, in the aforementioned scenario when the network traffic follows an uneven distribution, the workload should be balanced at the packet level instead of the connection level, which HRW fails to do genetically. As a result, we propose to adaptively change the HRW hash decision when it deteriorates the packet level workload balance. Our experiment results show that the adaptive hash-based multilayer scheduler achieves close to ideal load balance and the system throughput can be increased by 59.2% compared to the previously proposed connection-based scheduling in paper [7]. Authors in paper [12] also adopted a feedback system for HRW based on CPU utilization. Our work differs from theirs in that we choose a less expensive real time metric and implement the scheduler hierarchically for a much more complicated highly threaded server architecture.

To summarize, we make the following contributions in this paper:

- We motivate our research by pointing out that the connection locality alone does not guarantee performance benefits in a highly threaded hierarchical multi-core system like Niagara 2.
- We adopt HRW hash function to guarantee connection locality while maintaining load balance over the number of different connections.
- We implement a multilayer HRW scheduler hierarchically, corresponding to the Niagara 2 architecture so that connection workload is balanced progressively at the core, the pipeline and the thread level, respectively.
- We optimize the multilayer HRW scheduler to adaptively change scheduling decisions based on real time workload distribution at the packet level to provide better load balance.

The rest of the paper is organized as follows: In section 2, we review the background information about the Niagara 2 system architecture, its default system scheduler and the HRW hash. We describe our implementation in details in section 3. In section 4, we describe our experiment environment and we present our result in section 5. In section 6, we conclude our paper.

2. BACKGROUND

2.1 L7-filter and Connection Locality

L7-filter [1] is an important QoS component in Linux that classifies network traffic based on packet payload. It classifies the network traffic based on connections as opposed to packets. The connection-based DPI programs are gaining more publicity in both academic studies [7, 13, 14, 27, 31] and industrial products [3, 9, 11]. 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 8 packets for each connection. If a connection cannot be classified with 8 packets in the buffer [1], it is marked as "NO_MATCH", and any further packets for this connection will be excluded from matching.

As multi-core processors have become the *de facto* server platforms, a recent trend is moving towards the deployment of multithreaded DPI programs on multi-core servers. Paper [7] showed that a multithreaded L7-filter program could achieve a speedup of 7.6X in TCP throughput using an 8-core Intel Clovertown server. The reason behind this gain is to maintain the connection locality for incoming traffic to benefit from cache locality. Their research result is in line with the widespread Receive Side Scaling (RSS) technique implemented in NIC [21] as well as findings from an Intel Research group [29]. However, we observe from Fig. 1 that on a highly threaded hierarchical multi-core server using a Sun Niagara 2 processor, connection locality alone does not guarantee desired system performance. In Fig. 1(b), we show that the hierarchical parallelization resource accumulatively incurs load imbalance, which offsets the performance gain from connection locality. In addition, the uneven traffic distribution also introduces a challenge to the connection locality optimization.

2.2 The Sun Niagara 2 and the Solaris Scheduler

Fig. 2(a) illustrates the system architecture of a Sun Niagara 2 processor. The eight cores connect through a crossbar switch to eight banks of 16-way set associative L2 cache, totaling 4 MB. The Sun Niagara chipset series differ from other high-end server processors not merely in degree but also in kind. The Niagara 2 processor uses eight simple in-order SPARC cores rather than the more complicated out-of-order x86 cores. Each core on the Niagara 2 chip runs at a relatively lower (1.2 GHz [16]) frequency. However, the low frequency cores are complemented with two independent integer pipelines, each residing 4 hardware threads. Naturally, the Niagara chip forms a virtual hierarchical structure, with the cores at the first level, the pipelines inside each core at the second level, and the threads running on each pipeline at the third level.

Fig. 2(b) demonstrates the scheduler topology in the Niagara 2-Solaris system architecture. At every clock cycle, the hardware strand scheduler (the "Pick" unit) grants one of the four threads in a pipeline exclusive access to use the pipeline resource. Essentially, when one thread stalls for memory access, the "Pick" unit on chip chooses from the other 3 idling threads on the same core to hide the latency. Note that the scheduling done by "Pick" is a hardware implementation that runs at a clock cycle granularity, which cannot be modified in software. It is at a different level from the thread/pipeline/core scheduling discussed in this paper.

In addition to the "Pick" scheduler, there is also a kernel software thread scheduler that maps software threads to hardware threads. In Solaris 10, 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 8 hardware threads on the core's two integer pipelines [23]. This kernel software thread scheduler works at a higher level (closer to the application layer). The "thread affinity" system calls exist in both Linux and Solaris to overwrite the decisions made by this scheduler.

[1] The number of packets allowed in the buffer of a connection is a system configurable parameter.

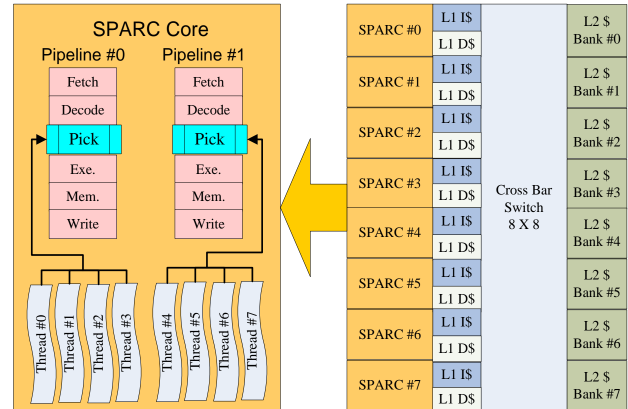


Fig. 2(a). The Sun Niagara 2 Chip architecture and the parallelism inside each SPARC core

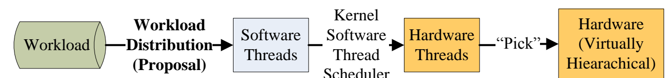


Fig. 2(b). The scheduler topology in the Niagara 2-Solaris system

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.

2.3 The Highest Random Weight (HRW) Hash Function

HRW is a popular choice for web servers [12]. It is later adopted by the CARP distributed caching scheme [22]. Benefits of HRW are proven [28] to include low overhead, load balance, high hit rate and minimal disruption. HRW always maps a given object name to the same server within a given cluster, which guarantees cache locality.

In practice, HRW assigns a weight to each server based on the requested object space. Every time a scheduling decision needs to be made, the incoming request triggers an update of the weight on each server. The server with the highest weight gets chosen to service the request. It is important to note that HRW provides load balance over the requested object space, i.e. in our case, the number of different connection IDs. In contrast, the actual loads due to the actual traffic received at the router/web server input ports may by no means be distributed uniformly over this request object space, but rather will exhibit certain locality patterns, as described in the "Packet Train" paper [10].

3. DESIGN AND OPTIMIZATION OF AN ADAPTIVE HASH-BASED MULTILAYER SCHEDULER FOR L7-FILTER

In this section, we propose a multilayer scheduler based on HRW hash that adaptively adjusts connection locality decisions for workload balance at the core, the pipeline and the thread level, respectively. In our scheduler, the workload is balanced over the length of the runqueue (i.e. service time) at each level, instead of over the number of different connections as in the original hash.

3.1 A Basic Implementation of HRW-based Scheduler on Niagara 2

In our software scheduling scenario, we define HRW as follows:

Let $g(\vec{c}, j)$ be a pseudo-random weight function $g : C \times \{0,1,2,...,63\} \to (0,n)$, i.e. we assume $g(\vec{c}, j)$ to generate a random variable (weight) in $(0,n)$ with uniform distribution. The value of n is different, depending on the selection of the weight function $g(\vec{c}, j)$. Let a packet arrive at an input i, carrying an identifier vector $\vec{c}$ belonging to C, i.g. connection IDs. The mapping $f(\vec{c})$ is then computed as follows:

$$f(\vec{c}) = j \Leftrightarrow g(\vec{c}, j) = \max_{k \in \{0,...,63\}} g(\vec{c}, k)$$

Because packets of the same connection share the same connection ID, our HRW function guarantees connection locality. Now we want to make sure this function also balances the workload upon the selection of the weight function. In our paper, we follow the random variable generation hash function g, as proposed in paper [28]:

$$g(k, S_i) = (A \cdot ((A \cdot S_i + B) \oplus D(k)) + B) \bmod 2^{31}$$

where $A = 1103515245$ and $B = 12345$. $D(k)$ is a 31-bit digest of the object name k and S_i is the ID of the ith server in the cluster. This function generates a pseudo-random weight in the range $[0..2^{31} - 1]$. In our case, the object name k is the connection ID of an incoming packet. Each S_i is represented by a software thread ID.

If we define a random variable q_i as the probability that a request will be sent to S_i, and another random variable l_i as the amount of processing done by server S_i, we claim the following two properties of our hash function, when the number of requests is infinite, as in paper [28]:

1) The coefficient of variation of q_i is zero, i.e. each software thread has an equal probability of being chosen to service the request to classify the connection buffer.

2) The coefficient of variation of l_i is zero, i.e. each software thread services the same amount of requests/connections.

These two properties guarantee that our HRW function balances different types of connections across the software thread pool, after enough connections pass through the system. A typical real network link usually contains more than 10K connections [12, 20], which guarantees that properties 1) and 2) hold true.

Implementation Details:

As the network traffic comes into the server, L7-filter checks the connection table and decides further processing for the packet. Each connection could be in one of the three states: "MATCHED", i.e. all packets of this connection are and will be marked with the corresponding protocol ID; "NO_MATCH", i.e. all packets of this connection are and will be directly forwarded to upper layer programs, bypassing L7-filter without being classified; and "NO_MATCH_YET", i.e. all packets of this connection are necessary for the classification until the state changes to one of the other two states. For the third case, L7-filter places the incoming packets into their corresponding entries in the connection reassembling buffer based on the 4-tuple (Source IP, Destination IP, Source Port # and Destination Port #) information in the packet header. These parts of the program can be seen as a preprocessing stage, which can be handled in a trace driven model [7]. For every newly assembled connection buffer, L7-filter calls the HRW hash to generate a weight for each and every one of the 64 threads. The scheduler picks the thread with the maximum weight, and assigns the connection buffer to the runqueue of the selected thread. The selected thread classifies each connection buffer in its runqueue on a First In First Out (FIFO) fashion.

Computation Cost analysis:

Intuitively, computing the weight function for 64 threads upon each new packet arrival is expensive. However, we can preprocess the computation of weight functions offline for different (connection ID, software thread ID) combinations, and then load the result table into memory. Essentially, the hash process requires only 1 additional memory access to check the result table, given we use array data structure to guarantee random memory access. Now, let us calculate the size of the result table:

For an example of 10K different connections, we need 10K * 64 threads * 4-byte weight value = 2.5MB. As we will present in the result section, our scheduling incurs very little overhead.

3.2 A 3-Level Hierarchical HRW-based Hash-Tree Scheduler

In section 3.1, we presented the baseline HRW-based scheduler. It guarantees connection locality while maintaining load balance over the number of different connections, independent of the underlying architecture and the parallelization implementation. In this section, we redesign the baseline HRW-based scheduler for Sun Niagara 2. Specifically, the redesigned HRW-based scheduler takes the hierarchical concerns into consideration, and further leverages the potential load imbalance.

In the background section, we already show that Niagara 2 is a highly threaded hierarchical multi-core processor. It has eight cores on chip, and each of the cores contains two independent pipelines running eight hardware threads. From Fig. 2, it is clear that the parallelization resources on a Niagara 2 chip naturally form a virtual "dendrogram/tree" architecture, with eight cores being the first level children; sixteen pipelines the second level; and 64 threads the lowest level.

With the understanding of the chip architecture, let us reevaluate the load balance property in the baseline HRW-based scheduler. It is coarse-grained in the sense that only the thread level parallelism is considered. A better way of load balance should take the entire parallelism hierarchy into consideration.

We redesign the baseline hash-based scheduler into a 3-level hierarchical scheduler. Essentially, we have a hash function to balance the workload at the core level, then another hash function at the pipeline level in the selected core, and finally a hash function to choose from one of the four threads in the selected pipeline. Our idea is formulated as follows:

We want to select the software thread j for every newly assembled connection buffer of connection $\vec{c}$,

$f(\vec{c}) = j$

$\Leftrightarrow$

$g(\vec{c}, j) = \max_{k \in \{0,\ldots,3\}} g(\vec{c}, k)$, where thread k belongs to pipeline j', such that:

$g(\vec{c}, j') = \max_{k' \in \{0,1\}} g(\vec{c}, k')$, where pipeline k' belongs to core j'', such that:

$g(\vec{c}, j'') = \max_{k'' \in \{0,\ldots,7\}} g(\vec{c}, k'')$, where k'' is one of the eight cores.

This redesigned scheduler uses the HRW hash at all the three levels and therefore balances the workload over the number of connections hierarchically. Fig. 3 shows how the algorithm works. The ultimate destination thread is selected progressively from top down in the tree structure. At each layer, the node with the maximum weight is selected. Scheduling at a lower layer would only choose from children nodes of the selected parent node.

Implementation details:

In addition to the weight vector for threads, now we need weight vectors for the cores and pipelines as well. We can use multi-dimensional arrays to reflect the hierarchical relationship between different level elements. The index system is described as follows:

- core_weight[core_ID] core_ID $\in$ [0,7]
- pipeline_weight[core_select][pipeline_ID] pipeline_ID $\in$ [0,1]
- thread_weight[core_select][pipeline_select][thread_ID] thread_ID $\in$ [0,3]

Computation cost analysis:

Compared to the baseline HRW-based hash scheduler, the hierarchical scheduler can also be precomputed offline. The only difference is that it requires two additional accesses to the weight result table. Moreover, the *core_weight* and *pipeline_weight* array requires additional memory to store. For an example of 10K connections, we need 10K * 8 cores * 4-byte weight value = 320KB for the *core_weight* array and 10K * 2 pipelines * 4-byte weight value = 80K for the *pipeline_weight* array. Therefore, we need 400KB of extra memory to store the hierarchical result table.

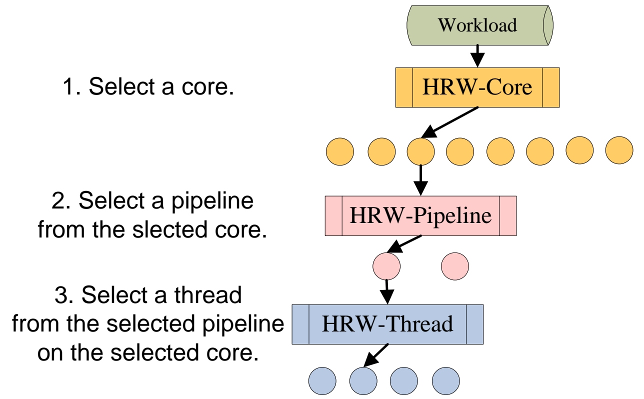


Fig. 3. An illustration of the HRW-based multilayer scheduler.

3.3 An Adaptive Feedback System for the HRW-based Hash Scheduler

Load sharing over the number of different connections might be problematic when the network traffic does not evenly distribute across connections. Consider the following example: suppose we have two connections c1 and c2, and 80% of the network traffics are of packets belonging to c1 while only 20% of the packets belong to c2. Both of the HRW-based schedulers can only use two of all the eight available cores (one for each distinguished connection, because of connection locality), leaving the rest of the cores idle.

This example shows that connection locality and load balance over the number of different connections cannot guarantee real workload balance over the system utilization, especially when the packet distribution does not follow the connection distribution. Under such circumstance, the scheduler should be able to observe the difference in utilization between each thread/pipeline/core, and relax the connection locality to adjust the workload accordingly.

In order to measure the real system utilization, an intuitive solution is to read from the kernel to obtain core/CPU utilization information [12]. However, we do not adopt this approach for two reasons: 1) reading kernel information and reporting back to the userspace program for every schedule incurs too much overhead for high speed networks. For a 10GbE network, packet intervals could be as short as 1.25 µsec (for MTU=1.5KB sized packets). The cost of interrupts and system calls (more than 0.5 µsec on our testbed machine) would be non-negligible in this case. 2) core/CPU utilization information is a real time value, which means it changes overtime. From the point when the values are measured to the point when they are reported to the program, the values are already outdated. We need a metric that accurately reports the run time values, and that is less affected by the reporting delay.

In our feedback system, we choose an application layer metric that is a lot less costly. We measure the overall runqueue length in terms of bytes at the core, the pipeline and the thread level, respectively. All the runqueues are filled with connection buffers with different lengths. The length of a connection buffer directly reflects the number of cycles needed for the DPI matching engine, because for the most widely used DFA representation, each input character (1 byte) requires exactly one clock cycle to process.

TABLE I
AN ADAPTIVE HRW-BASED MULTILAYER SCHEDULER

```
SCHEDULER (char * conn_buff)
Find core_select w. MAX HRW weight;
IF core_len[core_select] + conn_buff->len() -
  core_len[core_min] > THRESHOLD THEN
    core_select = core_min;
ENDIF
UPDATE the queue length of cores;
Find pipe_select w. MAX HRW weight;
IF pipe_len[core_select][pipe_select]
  +conn_buff->len()-core_len[pipe_select][pipe_min]
  >THRESHOLD THEN
    pipe_select = pipe_min;
ENDIF
UPDATE the queue length of pipelines in the selected core;
Find thread_select w. MAX HRW weight;
IF thread_len[core_select][pipe_select][thread_select]
  + conn_buff->len() -
  thread_len[core_select][pipe_select][thread_min]
  > THRESHOLD THEN
    core_select = core_min;
UPDATE the queue length of threads in the selected
pipeline of the selected core;
ENDIF
```

Therefore, the runqueue length is equivalent to the processing time in terms of CPU cycles. Table I summarizes the algorithm in details.

With this runqueue length measurement, we can adaptively change the HRW-based hash scheduling if the decision causes packet level workload imbalance. If the summation of the runqueue length between the HRW selected node at each level (core/pipeline/thread) and the current buffer length at the same level is greater than a THRESHOLD compared to the node with the shortest runqueue at the same level, the decision made by the HRW should be overruled, and the connection buffer should be scheduled to the node with the shortest runqueue. Two points needed to be noted: 1) the THRESHOLD value is a balance between HRW decision (when it is low) and Minimum-load Mapping [4, 24] (when it is high), i.e. a balance between connection locality and load balance. A high value for THRESHOLD makes HRW decisions more powerful and the feedback system less responsive, while a low value for THRESHOLD provides better load balance by overruling HRW decisions, incurring a higher scheduler overhead. Based on our measurements, we choose THRESHOLD at each level to be 10% of the shortest runqueue length at the same level in our experiments. 2) The adaptive feedback system works at all the three levels where HRW-based hash works. Essentially, our feedback system itself provides hierarchical feedback information to the schedulers at different levels.

Implementation details:

We use similar data structures and index system as those described in section 3.2 to record the runqueue lengths for the cores, the pipelines and the threads. The runqueue length array at each level is used to overrule the decision made by the HRW-based function at the same level, if and only if the summation between the runqueue of the selected node at each level (core/pipeline/thread) and the current connection buffer length exceeds the shortest runqueue length of this level by a THRESHOLD (10%). If the condition is satisfied, the current connection buffer should be scheduled to the node with the shortest runqueue of this level.

Computation cost analysis:

In addition to the three memory accesses to all the three result tables for the weight of the nodes at each level, the feedback system needs to check and compare among the length of runqueues at all the three levels. The runqueues are dynamic data structures, whose contents change as the program runs. Therefore, they cannot be precomputed. However, finding the minimum element in the runqueue length array with a fixed size (8 for the core level, 2*8=16 for the pipeline level and 4*2*8=64 for the thread level) only requires constant time. Therefore, the overall scheduling overhead including 1) checking the weight table; 2) checking the minimum runqueque length; and 3) comparing between 1) and 2) is only a constant factor for each incoming connection buffer. As we will present in the experimental results section later, the overhead of this scheduling mechanism is negligible compared to the cost of the pattern matching processing for the classification.

4. EXPERIMENTAL RESULTS

4.1 Experiment Platform

We use a Sun Niagara 2-based T5120 machine as our testbed server. The layout of the cores is presented in Fig. 2. The hierarchical processor architecture contains 8 in-order cores (1.2 GHz). Each of the eight cores embeds 2 independent integer pipelines that enable real multithreading without causing resource contention. Each pipeline is shared by 4 hardware threads, totaling 64 hardware threads in the system. As we can see from Fig. 2, the eight cores are connected to share a 4MB L2 cache through an 8X8 crossbar switch. Our testbed server also installs 16GB of 667MHz DDR2 memory.

We use Solaris 10 as our default OS. The baseline userspace sequential L7-filter is of version 0.6 with protocol definition updated by 05/19/2009. 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 library in Solaris. To keep our work a consistent reference and contrast with the paper [7], we remain using pthread as the multithreading library instead of the" thread" library provided by Solaris.

4.2 The Trace Driven Model

We adopt the same trace driven model proposed in paper [7]. The decoupled model proposed in that work separates the packet processing in the kernel stack from the pattern matching operations at the application layer. We choose the most recent version 1.23 libnids [15] 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. For the packet trace, we select an intrusion detection evaluation data set from the MIT DARPA project [17]. It contains about 340K packets from more than 40K connections.

4.3 Performance Metrics

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 core/CPU utilizations, we present results for both software threads (using "prstat" command) and physical core utilization (using a Perl script "corestat"). More specifically, we break down the utilization to show the workload distribution at the core, the pipeline and the thread level. We also present the maximum queue length for each thread to correspond to the software thread utilization. This gives us a better idea about the workload balance situation.

We additionally profile the life of a packet in the system to illustrate the overhead of scheduling versus the cost of pattern matching.

4.4 Throughput and Core Utilization

Fig. 4 illustrates the throughput and CPU utilization results obtained from different optimizations. The "conn+aff." reflects the basic connection locality + thread affinity optimization, as proposed in their paper [7]. We adopt this optimization as our baseline set up. It is clearly presented that our adaptive multilayer hash scheduler ("3-HRW+Adp") increases the system throughput by 130% (0.87 Gbps VS 1.99Gbps). It is arguably reasonable to question the fairness of this comparison because the testbed in that paper was an Intel dual quad-core Clovertown machine, whereas we use a 64-thread 8-core Sun Niagara 2 machine. Therefore, we did a simple optimization ("conn+os") for the connection locality technique by using the default software scheduler on Solaris, which provides a better load balance compared to the thread affinity set up. This optimization can increase the throughput by 43% (1.25 Gbps VS 0.87 Gbps). To keep our result report reasonable, we choose the “conn+os” case as the default case which our optimizations are compared to. We also observe that HRW alone only increases the throughput by 3.2%, while the multilayer HRW achieves a throughput of 1.54 Gbps, an additional improvement of 20%. The ultimate system throughput can be increased by 59.2% compared to "conn+os" using the adaptive multilayer scheduler.

The CPU utilization shows a pattern of growth as throughput increases. This is because better load balance reduces CPU idle time. Therefore, more CPU time is spent in matching connection buffers. If the per core CPU workload is unevenly distributed, some of the cores might be idling after they finish the workload in their runqueue, while those cores with higher workloads keep running blindly, blocking workload deeper in the runqueue. In the next subsection, we will present the workload distribution situation at the core, the pipeline and the thread level.

4.5 Workload Distribution

Fig. 5 shows the utilization at different levels in the system using different optimizations. We obtain the results for all the three figures at the same timestamp after system warm-up. The bars in each figure show the average utilization across all the processing units at the same level, e.g. the bars in Fig. 5.(b) are the average pipeline utilization across all the 16 pipelines in the

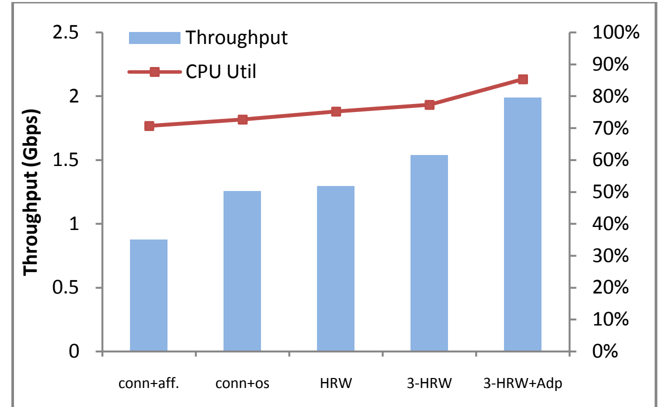


Fig. 4 System Throughput and Utilization.

system. The vertical lines in each figure represent the range between the highest and lowest utilization for each optimization. Note that the value for a node at a higher level in the hierarchical architecture is not simply equivalent to the sum of its children's values. Especially for the thread utilization, Solaris counts the block I/O time as part of the non-idling statistics for a thread. Therefore, even if all the four threads show 100% utilization, their pipeline/core utilization numbers could still be low [23]. However, what we really care about in Fig. 5 is the load balance impact of each optimization, which is not affected by the structural relations.

All the three figures show that HRW-based scheduling optimizations consistently provide narrow ranges of utilization, independent of the depth of the hierarchy. At the same time, a narrow range of utilization means each node at the same level shares a more similar workload. Therefore, our adaptive multilayer hash scheduler provides the best load balance and consequently the most efficient CPU utilization. It should also be noted that as we go down the hierarchical tree, the benefits of the multilayer hash increase. This is understandable because the genetic workload discrepancy grows accumulatively, as we go down the hierarchy. The utilizations at a higher level are less different due to the accumulative effect. At the thread level, the workload difference is reduced from 89% to 10%. An increased average utilization further verifies the gain in efficiency.

Here we also present the runqueue length at the thread level to directly illustrate the changes in workload balance. As we can see from Fig. 6, it is quite straightforward that the runqueue length becomes much smoother when our scheduling optimizations that are applied. Another observation from the same figure shows the average runqueue length of the thread decreases as we further optimizes our scheduler. This observation means the overall matching time is reduced in the system, which is in line with the observation in Fig. 4.

4.6 A Life-of-Packet Analysis (Overhead)

In this subsection, we discuss the overhead of our hash-based scheduler by conducting a life-of-packet analysis, which profiles the execution time for each component along the processing path of one packet instead of the entire packet trace.

Fig. 7 scales the execution time to 100% for all the five different optimizations. We would like to present the impact of scheduling overhead on the overall packet processing. It shows that preprocessing components take about 5% of the overall packet processing time. The cost of scheduler increases as more

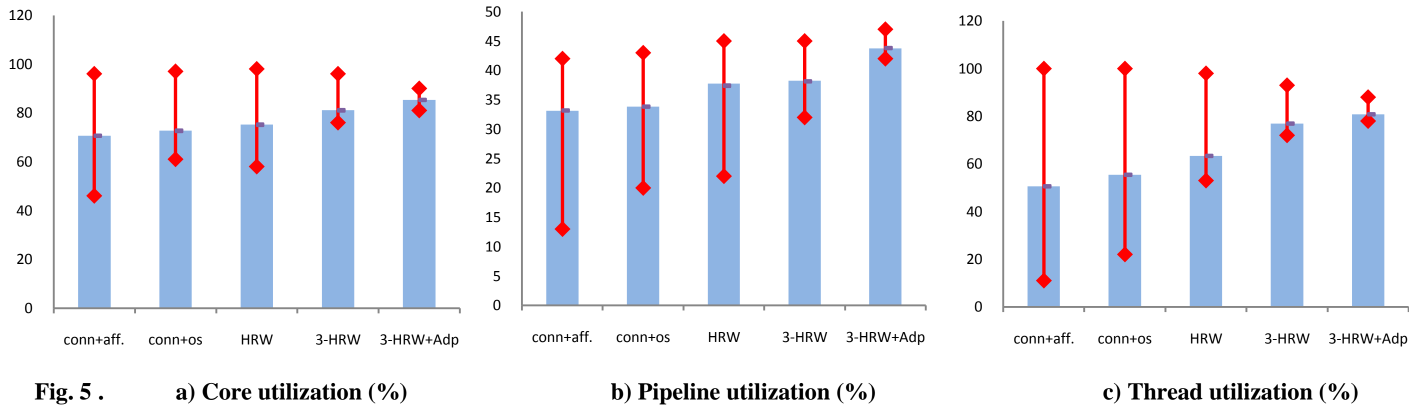


Fig. 5 . **a) Core utilization (%)** **b) Pipeline utilization (%)** **c) Thread utilization (%)**

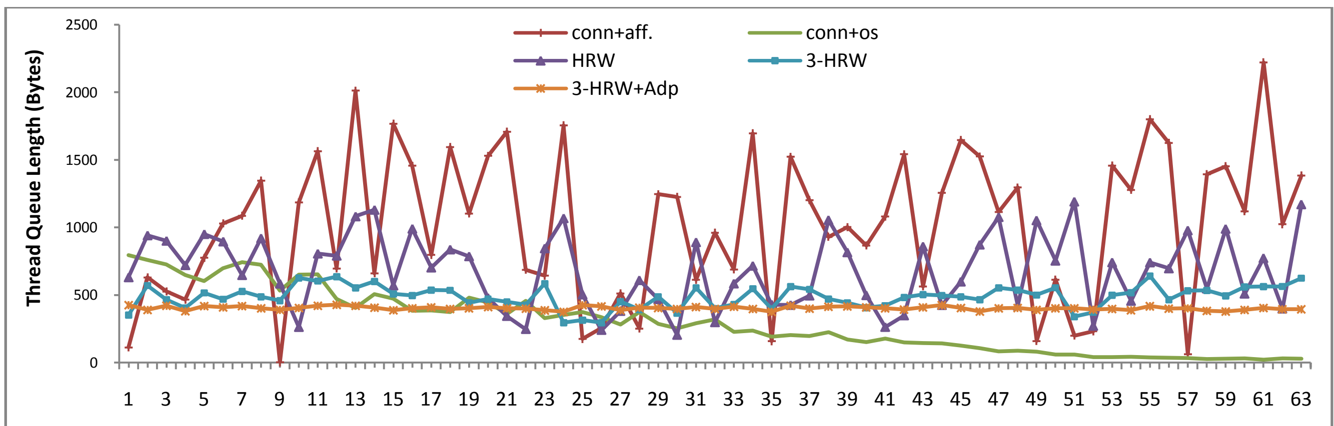


Fig. 6. Runqueue length on all the 63 matching threads. Note that thread #0 runs the preprocessing thread exclusively.

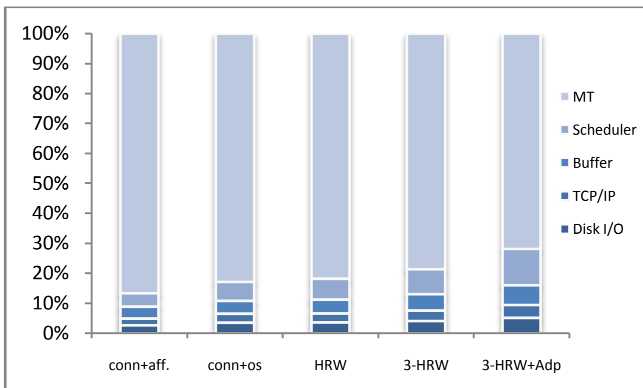


Fig. 7 Scaled execution time percentage for each component.

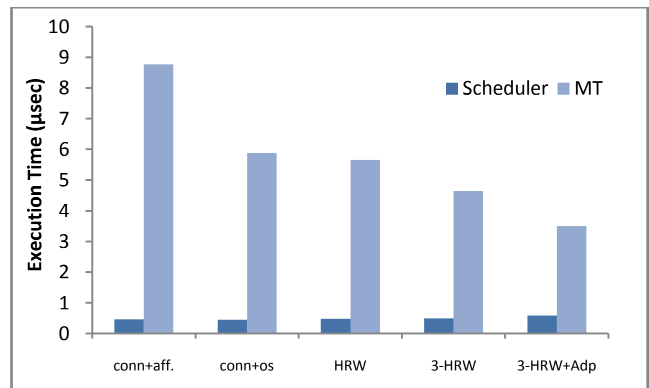


Fig. 8 Absolute execution time comparison between the scheduler and matching thread.

heuristics are applied. For the adaptive multilayer hash, it takes about 10% of the overall packet processing. Compared to the 76% execution time spent in pattern matching, we believe this overhead is still acceptable. We also observe a decreased time share of Matching Thread (MT) when more optimizations are applied. A smaller time-share for the MT in Fig. 7 can be caused by either a reduced matching cost in the MT or an increased computation overhead in the scheduler.

Fig. 8 shows the absolute execution time for a packet. Clearly shown from this figure, each matching thread runs longer than the scheduler does. Therefore, the reduced MT execution percentage in Fig. 7 is due more to the reduction in MT execution time than the increased scheduler cost (from 0.49 µsec to 0.57 µsec). This observation verifies our theoretical analysis in section 3. The average per packet execution time for the MTs is reduced because workloads are more balanced on the available threads. The workload balance 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.

5. RELATED WORK

5.1 Optimizations for DPI

The costly 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 [2]; 2) increasing throughput by processing multiple input characters per clock cycle [8, 14]; and 3) balancing between the memory bandwidth and memory size requirement [13, 31].

Another direction of the research studies the deployment of DPI programs, i.e. how to use hardware accelerators. In this domain, both FPGA [18] and Network Processor [19] solutions have been proposed to explore the packet level parallelism inside DPI programs. [27] proposed a "bit-splitting" architecture to explore the internal parallelism inside of the state machines.

As multicore-based web servers become the mainstream platforms for network appliances, research has increasingly proposes to deploy DPI programs using general purpose multi-core servers. Authors in paper [30] discussed the possibility of parallelizing SNORT [25] using multi-core servers with a 3-level feedback system. Another research group [7] designed a multithreaded L7-filter on a Intel Xeon server. It showed good performance by using connection locality and thread affinity. However, we found that simply applying connection locality optimization alone does not guarantee good performance on a highly threaded hierarchical multi-core processor like Sun Niagara 2. Our analysis showed that load imbalance across the extensive parallelism resources offsets the benefits achieved from connection locality. Therefore, we adopted a hash-based technique and a feedback system to consider load balance while maintaining connection locality.

5.2 Multi-core Scheduling and Hash

The key issue in multi-core scheduling is how to balance the workload across available processing resources. Previously proposed works [5, 6, 26] mainly achieve balanced workload based on real time thread migration. The advantage of research in this domain is that the locality of running threads can be adjusted to shorten the blocking delay incurred by uneven workload distribution based on real time statistics. The downside of migration-based load balance algorithms is cache thrashing, i.e. old cache data might be replaced by new data for the recently migrated thread.

Hash functions have been widely adopted in the network domain. In the client-server model, hash functions are a favorable choice to map client requested objects into the web cache [22]. One of the popular hash functions is HRW, which is proposed in paper [28]. Although the algorithm provides load balance over the request object space, it is not adaptive and therefore potentially vulnerable to traffic locality. In paper [12], the authors presented a feedback system for the traditional HRW hash. However, our research differs from theirs in two folds: 1) we implemented a multilayer HRW using a highly threaded hierarchical multi-core server instead of a network processor simulator; 2) we chose 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.

6. CONCLUSION AND FUTURE WORK

In this paper, we propose a scheduler for L7-filter on a highly threaded hierarchical Sun Niagara 2 multi-core server. In addition to maintaining the benefits from the connection locality of the network traffic like some previous proposed schedulers [7, 12, 30], our scheduler also adaptively relaxes the locality constraint to achieve better load balance. Based on the hierarchical architecture of the Sun Niagara 2 processor, our scheduler works at the core, the pipeline and the thread level, respectively. We choose the HRW hash as our baseline hash function that guarantees connection locality and load balance over the number of different connections. We apply a low overhead adaptive feedback system to balance the workload over real time queue length at each level. Our experimental results show that the adaptive hash-based multilayer scheduler can improve the L7-fitler throughput by 59.2% compared to a previous work.

As to future work, we are in the process of developing a hash function that encloses the feedback system into the hash itself. We believe a self-adaptive hash function can further reduce the system overhead incurred by the additional feedback control.

7. ACKNOWLEDGEMENT

The authors would like to thank anonymous reviewers for their valuable comments. We also thank Intel for providing the Sun Niagara 2 based web server. The research was supported in part by NSF grant CNS 0832108 and NSFC (60625201, 60873250).

REFERENCES

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

[2] B. Brodie, et al., "A Scalable Architecture for High-Throughput Regular-Expression Pattern Matching", ISCA '06.

[3] Cisco IOS Netflow, http://www.cisco.com/en/US/products/ps6601/products_ios_protocol_group_home.html

[4] Cisco Systems. Scaling the world wide web. Available from http://cio.cisco.com/warp/public/751/advtg/swww_wp.htm.

[5] A. C. Dusseau, et al., "Effective Distributed Scheduling of Parallel Workloads", SIGMETRICS 1996.

[6] A. Fedorova, et al., "Cache-Fair Thread Scheduling for Multicore Processors", OSDI '06.

[7] D. Guo, et al., “A Scalable Multithreaded L7-filter Design for Multi-core Servers”, ANCS 2008.

[8] N. Hua, et al., "Variable-Stride Multi-Pattern Matching for Scalable Deep Packet Inspection", IEEE INFOCOM '09.

[9] Huawei MSCG Hierarchical DPI Solution, http://www.huawei.com/products/datacomm/catalog.do?id=1219

[10] Raj Jain and Shawn A. Routheir, "Packet trains - measurements and a new model for computer network traffic", IEEE Journal on Selected Areas in Communications, 4(6):986-995, September 1986.

[11] Juniper M Series Multiservice Edge Routers, http://www.juniper.net/us/en/local/pdf/datasheets/1000042-en.pdf

[12] Lukas Kencl, Jean-Yves Le Boudec, "Adaptive Load Sharing for Network Processor", IEEE INFOCOM 2002.

[13] S. Kumar, et al., “Advanced Algorithms for Fast and Scalable Deep Packet Inspection”, ANCS 2006.

[14] S. Kumar, et al., "Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection", SIGCOMM '06.
[15] libnids, http://libnids.sourceforge.net/
[16] Harlan McGhan, "Niagara 2 Opens the Floodgates - Niagara 2 Design is the Closest thing Yet to a True Server on a Chip", The Insider's Guide to Microprocessor Hardware, 11/6/06-01.
[17] MIT DARPA Intrusion Detection Data Sets, http://www.ll.mit.edu/IST/ideval/data/2000/2000_data_index.html.
[18] A. Mitra, et al., “Compiling PCRE to FPGA for Accelerating SNORT IDS”, ANCS 2007.
[19] P. Piyachon and Y. Luo, “Efficient Memory Utilization on Network Processors for Deep Packet Inspection”, ANCS 2006.
[20] SPECweb 2005 Published Results, http://www.spec.org/web2005/results/
[21] Receive Side Scaling (RSS), http://www.microsoft.com/whdc/device/network/NDIS_RSS.mspx/.
[22] K. W. Ross, "Hash Routing For Collections of Shared Web Caches", IEEE Network, Vol. 11, No. 6 November-December 1997.
[23] Steve Sistare, "The UltraSparc T2 Processor and the Solaris Operating System", Oct 09, 2007, http://blogs.sun.com/sistare/entry/the_ultrasparc_t2_processor_and
[24] Reid G. Smith, "The contract Net Protocol: High-level Communication and Control in a distributed Problem Solver", ACM Transactions on Computers, pages 1104-1113, December 1980.
[25] SNORT Network Intrusion Detection System, http://www.snort.org/
[26] D. Tam, et al., "Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors", EuroSys '07.
[27] L. Tan, et al., "A High Throughput String Matching Architecture for Intrusion Detection and Prevention", ISCA '05.
[28] D. G. Thaler, C. V. Ravishankar, "Using name-based mappings to Increase Hit Rates", IEEE/ACM Transactions on Networking, Vol. 6 No. 1 pp. 1-14, Feburary 1998.
[29] B. Veal, et al., “Performance Scalability of a Multi-core Web Server”, ANCS 2007.
[30] J. Verdu, et al., " MultiLayer processing - an execution model for parallel stateful packet processing ", ANCS 2008.
[31] F. Yu, et al., Fast and memory-efficient regular expression matching for deep packet inspection, ANCS 2006.