A Scalable Multithreaded L7-filter Design for Multi-Core Servers

Danhua Guo[1, 3], Guangdeng Liao[1], Laxmi N. Bhuyan[1], Bin Liu[2], Jianxun Jason Ding[3]

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

[2]Tsinghua University
Beijing, China
lmyujie@gmail.com

[3]Cisco Systems, Inc.
San Jose, CA
{dannguo,jiding}@cisco.com

ABSTRACT

L7-filter is a significant component in Linux's QoS framework that classifies network traffic based on application layer data. It enables subsequent distribution of network resources in respect to the priority of applications. Considerable research has been reported to deploy multi-core architectures for computationally intensive applications. Unfortunately, the proliferation of multi-core architectures has not helped fast packet processing due to: 1) the lack of efficient parallelism in legacy network programs, and 2) the non-trivial configuration for scalable utilization on multi-core servers.

In this paper, we propose a highly scalable parallelized L7-filter system architecture with affinity-based scheduling in a multi-core server. We start with an analytical study of the system architecture based on an offline design. Similar to Receive Side Scaling (RSS) in the NIC, we develop a model to explore the connection level parallelism in L7-filter and propose an affinity-based scheduler to optimize system scalability. Performance results show that our optimized L7-filter has superior scalability over the naive multithreaded version. It improves system performance by about 50% when all the cores are deployed. In addition, this model also ameliorates the performance degradation in a virtualized environment due to our direct mapping mechanism.

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

ANCS'08, November 6–7, 2008, San Jose, California, USA.

Keywords

Affinity, L7-filter, Multi-Core, Packet Classification, Parallelism, QoS, Scalability, Scheduling.

1. INTRODUCTION

Network resource competition becomes increasingly intense as more and more applications demand high bandwidth and more computing capability. The Quality of Service (QoS) in network domain requires powerful classifiers to distribute resources based on application priorities. Traditional packet classifications make the decision based on packet header information. But many applications, such as P2P and HTTP, hide their application characteristics in the payload. They require dynamic assignment of port numbers during connection establishment. Under such circumstances, Deep Packet Inspection (DPI) is used to identify the protocol/application information and improve QoS.

A layer-7 classifier requires massive processing capability. The emergence of high speed networks, such as 10 Gigabit Ethernet (10GbE), increases the intensity of network traffic, which consequently elevates the demand for fast DPI processing. Traditional single core server is insufficient to satisfy DPI functionality. However, multi-core architectures can boost packet processing by executing concurrent processes/threads on different cores. Therefore, they are widely deployed as server platforms to provide more computing power and to match I/O processing required by high bandwidth network. Cisco's Application Oriented Network (AON) technology also employs multi-core architectures [4].

In spite of its enhanced processing power, efficient core utilization in a multi-core architecture remains a challenge. The lack of efficient program level parallelism in legacy network applications greatly limits the utilization of multi-core architectures, and they often perform no better than single core systems. Furthermore, inter-processor communication in multi-core architectures hampers system scalability. Experiments [25] showed that the 8-core test server running a modified SPECweb2005 workload achieved only a 4.8X speedup in throughput (compared to the ideal 8X).

To address the issues above, we propose a highly scalable parallelized L7-filter system architecture with affinity-based scheduling for a multi-core server. The original L7-filter is a sequential DPI program that identifies protocol information in a given connection. It reads in network traffic through Netfilter's *QUEUE* in the kernel, and was first designed as a component in Linux's QoS framework. However, we parallelize the L7-filter operations based on a userspace version. Our technique is in line with many recent products which have moved Netfilter processing from the kernel level to application level [3, 4]. Previous research from both academia and industry [2, 9, 11] have demonstrated that the performance of L7-filter is bounded by the cost of pattern matching. Therefore, we have developed a decoupled model to separate the packet arrival handling and focus on optimizing the pattern matching operations at the application layer.

Based on the understanding of this model, we explore potential connection level parallelism in pattern matching, and propose an affinity-based scheduler to enhance the scalability of multithreading. Essentially, we assign packets belonging to the same connection to the runqueue of the same thread, which is dispatched to a dedicated core in multi-core server. Similar to Receive Side Scaling (RSS) [20] for the NIC, our scheduler works in software and provides faster packet classification with good scalability in the multi-core architecture.

Considering that DPI engines always reside in a back end server or a content-aware router/switch, one should be concerned with the memory limit for the matching process. We measure the number of connections under processing and decide the minimum buffer size with each thread running independently on a separate core. The measurement results demonstrate storage efficiency (bounded by KByte scale) and improved practicality of our multithreaded L7-filter architecture.

Our contributions in this paper can be summarized as follows:

- Developed a trace-driven offline L7-filter DPI system model to study its processing requirements in a well-controlled environment without noises from network infrastructure and kernel stack.
- Designed and implemented L7-filter parallel algorithm at the connection level and proposed an affinity-based scheduler to achieve higher performance scalability.
- Performed a unique "life-of-a-packet" analysis to illustrate the L7-filter performance bottleneck and demonstrate the optimization advantages of our design.

The remainder of the paper is organized as follows: In section 2, we provide background information and motivation of the paper. In section 3, we demonstrate the system architecture in a progressive fashion. We present the performance measurements in section 4. In section 5, we discuss related research. Finally, in section 6, we conclude the paper and propose future research directions.

2. BACKGROUND AND MOTIVATION

2.1 Packet Inspection and L7-filter

The universal intention of packet inspection is to provide Quality of Service (QoS) in the network domain. According to various priorities of applications, network resources must be distributed in a manner such that critical applications are guaranteed better services in terms of bandwidth and latency. In retrospect, packet inspection has evolved from simple matching of packet header information to intensive computation with payload data.

In spite of the simplicity and practicality of the classical port-based packet inspection, DPI is more and more widely used to meet the requirement of classifying file sharing services such as P2P in recent years. DPI checks packet payload information where service characteristics are deliberately hidden in some malicious applications. An accurate and efficient classification of packet payload prevents these malicious applications from acquiring large bandwidth, and therefore guarantees network QoS.

Among many signature-based protocol parsing software, L7-filter [2] is most widely used for connection-based DPI in Linux. By matching against signature fields of various protocols, L7-filter uses GNU Regular Expression (RE) matching to obtain protocol type associated with the application layer data in the packet. Different from signature-based Intrusion Detection Systems (IDS), 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, and thus can be easily implemented and deployed in software without any specific hardware accelerators. That said, the computation cost of the L7-filter software still remains high for real-time processing of packets [2, 9, 19]. Thus, optimization effort is needed.

2.2 Multi-core Architecture

Multi-core architecture duplicates hardware resources such as ALU, L1 cache, etc. on the same die, and hence allows multiple processes to run concurrently on different cores. Intel multi-core architecture, as shown in Figure 1, provides shared L2 cache for cores in the same group. This simplifies inter-core communication by eliminating cache-coherency protocols as used in multi-cache systems.

Multi-core architectures are deployed more and more to satisfy the requirement of faster packet processing. In addition, multithreading directives such as Pthread and MPI have explored program level parallelism. [8, 15, 21, 22] propose OS level heuristics to distribute heterogeneous workloads among cores to improve efficiency in system utilization. Unfortunately, most of such available designs

require a large amount of change in the OS.

Since kernel 2.6, Linux provides affinity configuration API for SMP platform. The user (with root access) could change the CPU affinity to interrupt and userspace applications. Foong [6] shows the efficiency of CPU affinity with multiple NICs. Compared to previous works, we believe tuning up affinity settings for Linux is easier and more reliable, especially when kernel modification becomes cumbersome in a virtualization environment.

2.3 From Kernel to Userspace

Most of the DPI software operations are conducted in the OS kernel space. Granted that operations in the kernel sometimes provide performance benefits, computation intensive operations are not suitable in the kernel space. The developers of L7-filter realized this problem and admitted in 2006 that, "By Dec 2006, we had realized that working anywhere in kernel space was not the brightest idea, so we released a version that runs in userspace and gets its data through Netfilter's *QUEUE*." [2]

That userspace version of Linux L7-filter is single-threaded. Even though the computation-intensive operations are moved out from the kernel space to the userspace, this single-threaded L7-filter cannot utilize the multi-core processing resources. Hence, a multithreaded L7-filter that applies proper core affinity can better utilize multi-core server resources and significantly improve L7-filter performance.

2.4 Connection-based Parallelism

Connection-based parallelism was productized by Microsoft in their Receive Side Scaling (RSS) NIC technology. In that technology, a particular core in the system is assigned to process all the packets in the same connection. Compared to the default load-balance scheduling in Linux kernel (with *irqbalance* enabled), RSS balances the workload on a connection rather than packet basis [20].

The same idea can be applied to our L7-filter parallelization effort. Because packets in the same connection share not only IP layer information in the header, but also potential common sections in the payload, processing packets of the same connection in the same CPU can improve I/O performance by maintaining a better cache locality. Since L7-filter classifies packets based on connections, an implementation of connection-based parallelism is expected to provide efficiency in multi-core utilization.

3. DESIGN AND OPTIMIZATION OF AN MULTITHREADED L7-FILTER

In this section, we present our optimized L7-filter system architecture to address the issues raised above. We first develop a decoupled offline model to focus on optimizing the performance bottleneck in the L7-filter. We then propose a connection level multithreaded L7-filter system architecture and an affinity-based scheduler to efficiently utilize a multi-core server. As an extension, we also propose a processor mapping mechanism in Xen hypervisor and discuss the impact of virtualization on our new model.

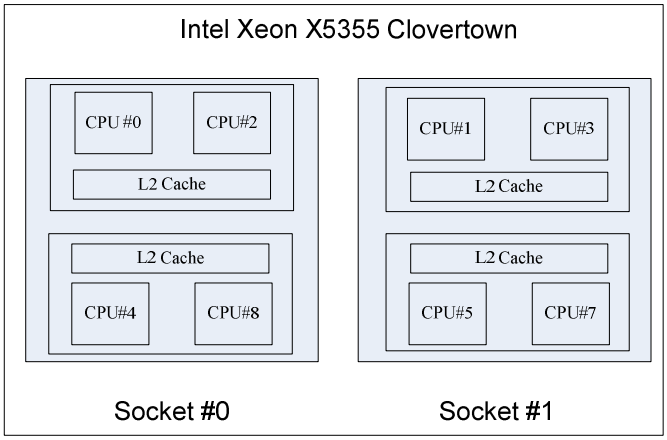

Figure 1. CPU layout of Intel Xeon X5355 Clovertown machine.

3.1 Decoupling Linux L7-filter Operations

Network traffic in 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. A registered callback function is then used for every packet that traverses the respective hook within the network stack. Inside the network stack of the kernel, a series of operations are executed to establish a connection buffer based on 5-tuple connection information in the packet header. After such a preprocessing stage including TCP/IP packets checksum verification, TCP/IP reassembling, IP refragmentation, etc., L7-filter starts to match all the application layer data of the arriving packets in the same connection against the protocol database in a sequential fashion.

It is known that the pattern matching operation at the application level consumes most of the time in DPI system [2, 9, 19]. We expect the same to be true in an L7 filter, both intuitively and by the experiment data to be presented in section 4.5.

To concentrate on optimizing the pattern matching operation, we developed an offline trace-driven model in our study. We choose libnids [10] as a userspace module. Libnids reads tcpdump trace files and simulates kernel network stack behaviors in userspace. In the real-world situation, packet arrival and pattern matching operations are tightly coupled. However, in our study, we use an offline trace input to replace the handling of network packets arrival. 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

hot-spot 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.

3.2 Modeling Single-Threaded L7-filter

Once a packet is processed by libnids, L7-filter classifies the packet in the steps described in Figure 2. Packets are fed into the system at the optimal speed (as in a TCP connection with no packet dropped).

The original online L7-filter is substituted by a combination of a Preprocessing Thread (PT) and a Matching Thread (MT). The PT functions as a real network stack in the kernel and schedules the packets. 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 the 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*".

As shown in Figure 2, one MT is handling the computation-heavy pattern matching operation. More MTs should be deployed to handle this operation, especially on multi-core based systems.

3.3 Parallelizing L7-filter at Connection Level with an Affinity-based Scheduler

A straightforward optimization to the single-threaded L7-filter is to create more MTs in the thread pool. Theoretically, multithreading can improve system performance in proportion to the number of additional processing units. However, in real practice, the scalability issue of multithreading depends heavily on the OS scheduler and the design of multithreading.

In Linux kernel 2.6, an O(1) scheduler takes the place of the O(n) scheduler in kernel 2.4 in order to improve performance on highly threaded workloads. Processes created in an SMP system are placed on a given CPU's runqueue. In the general case, it is impossible to predict the life of a process. Therefore, the initial allocation of

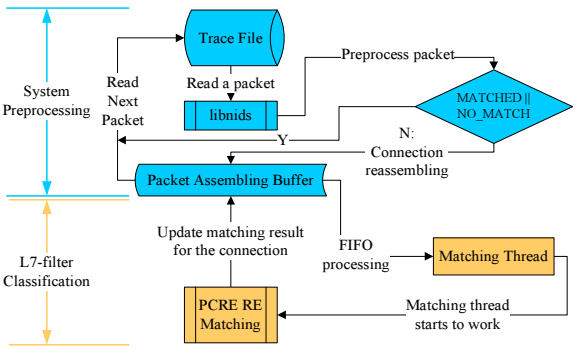

Figure 2. Trace-driven L7-filter data flow.

processes to CPUs is very likely to be suboptimal. To maintain a balanced workload across CPUs, every 200ms, Linux 2.6 scheduler checks to see whether a cross-CPU balancing of tasks is necessary. On the other hand, when a thread blocking for I/O is signaled, it will be awakened on the core (migration occurs, if necessary) where the event occurred [13, 16]. This ensures that application processing of a flow's packets is likely to be executed on the same core as its network protocol processing.

Despite the advantage of resolving load imbalances and implementing I/O affinity, the O(1) scheduler introduces an undesirable overhead for periodic CPU statistic collection and additional cache misses due to inter-core data copies. Studies [23, 24] have shown load-balancing to be an issue for edge (e.g. routing) workloads. As a result, we need to find an alternative to solve the multithreading scheduling problem.

Once more MTs are created, each MT executes on a connection buffer basis. When a new packet is reassembled for a connection, randomly selecting a non-empty runqueue of a thread introduces additional cache overhead by copying packets of the same connection to different cores. In addition, it also wastes the thread resources. Consider the case when MT #*t* is matching for connection #*c* with *p* packets in the buffer. During execution of MT #*t*, another packet of connection #*c* arrives. Since no matching result is reported yet, this new packet is reassembled with all the *p* packets in MT #*t*'s buffer and the *p+1* packets of connection #*c* is dispatched to another MT #*t+1*'s buffer for further action. When MT #*t+1* starts to classify for connection #*c*, MT #*t* might return that connection #*p* belongs to protocol #*q*. Since MT #*t+1* is unaware of the status, it has to go through the same process and thus wastes valuable computation resources and incurs cache pollution if it is load-balanced to a different CPU from where MT #*t* executes originally.

To attack the challenges discussed above, we propose an affinity-based scheduling mechanism for our multi-core server, as shown in Figure 3. We affinitize the PT in core

#0* and bind an MT for each of the cores left in the multi-core server. On the one hand, multiple MTs ameliorate the lack of processing power for pattern matching. Even though OS scheduler can balance the workload to all the cores with only one MT, we believe dispatching an independent thread to a dedicated core saves the cost of scheduling overhead and reduces cache misses introduced by live migrations of unbalanced workloads. On the other hand, in order to avoid cold-cache-line effect, we develop our own scheduler for thread dispatching, which will be discussed shortly. With our scheduler, the cache and resource efficiency as previously discussed are both greatly improved.

Figure 4 illustrates the data flow in our scheduler. Essentially, all the assembling buffers for the same connection (with different number of packets) will be scheduled to the same MT. Similar to the baseline model in Figure 2, the optimized scheduler initially decides whether an incoming packet needs classification based on the status of its host connection. If no previous result has been reported for the connection, the scheduler tries to append the new packet to the connection buffer and add this new buffer to the MT runqueue that already contains assembling buffers of the same connection. In case the desired MT runqueue is full, the scheduler will sleep until the runqueue is available for new entries.

There are two heuristics in the optimized multithreaded model for resource contention. First, if an incoming packet belongs to a new connection, the scheduler will try to balance the workload by looking for an MT that has the shortest runqueue. This load balance mechanism incurs no extra overhead compared to the default OS scheduler because the new connection has to suffer from cold cache line anyway. In addition, before classifying each entry in the runqueue, the MT checks whether the connection status has been changed. If L7-filter successfully classifies the connection with an assembly buffer of less number of packets, the MT will give up further attempt for this connection and get the next connection to be classified from its runqueue.

By serving connection buffer to the same MT in a FIFO fashion, wastage of MT resources due to information asymmetry of the classification status is avoided. Moreover, when previous instance fails to classify the connection, the used packets are still kept in the cache that hosts the MT. Consequently, further connection classification could enjoy the warm cache and executes more efficiently compared to the original case.

* This is an arbitrary selection. PT can be affinitized to any core.

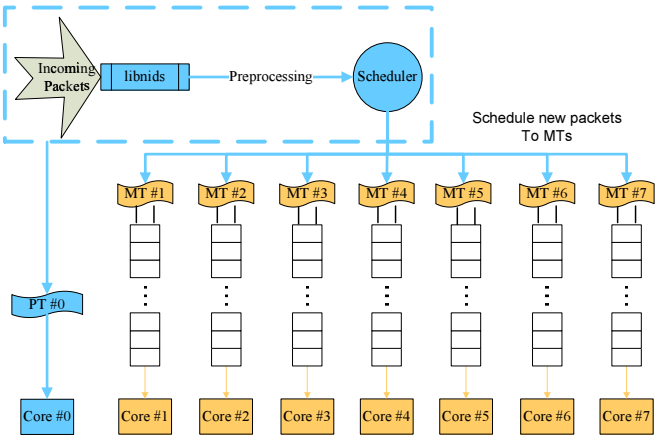

Figure 3. Affinity-based Mulit-threaded L7-filter Architecture.

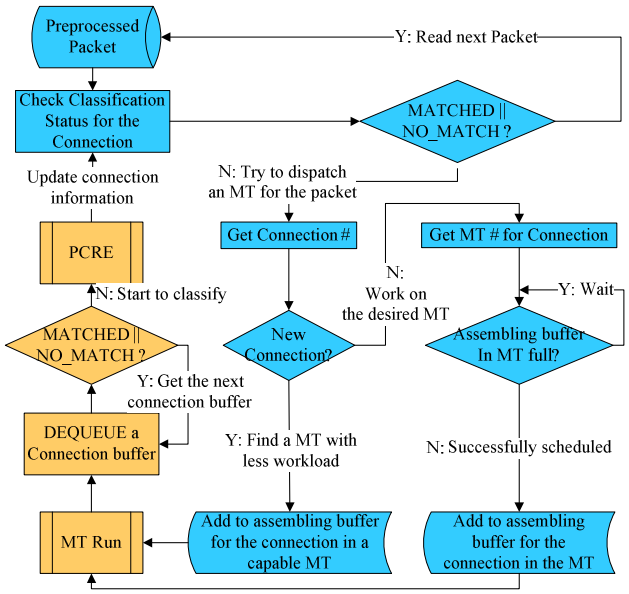

Figure 4. Data flow in Scheduler.

4. EXPERIMENTAL RESULTS

4.1 Experiment Platform

We choose Intel Clovertown (Dell Poweredge 2900) as our test bed server machine. This server system has two CPU sockets, each embeds a quad-core Xeon X5355 2.66GHz processors, and 16GB of 667MHz DDR2 SDRAM. The layout of the cores is presented in Figure 1. As we can see from the figure, each socket has two 4MB shared L2 caches.

We use Linux kernel 2.6.18 as our default OS. The baseline userspace sequential L7-filter is of version 0.6 with protocol definition updated by 04/23/2008. We choose the most recent version 1.23 libnids as the preprocessing component. We also use PAPI 3.6 [17] to measure cache misses. We hard-code time stamps into the L7-filter source

to obtain eclipsed time of different components of the system.

For the packet trace, we select an intrusion detection evaluation data set from the MIT DARPA project [11].

4.2 Performance Metrics

In order to verify the performance of our model, we compare our optimizations for multithreading scalability with the default OS scheduler in terms of throughput, CPU utilization and cache misses. We define the system throughput as the size of the total packets in the trace file divided by the execution time. In addition, we also provide a life-of-packet analysis to measure various overhead during processing. As an important metric for real router/switch, memory requirement of our model is also discussed.

We present below the evaluation of our optimized L7-filter in comparison to the original version. For the original case, we use the default OS scheduler to dispatch MTs with periodic load balancing. Our affinity-based scheduling for connection level parallelism is then implemented and compared with the original scheduler. The experimental results prove the efficiency of our design.

4.3 Throughput and Core Utilization

Figure 5 illustrates the throughput and CPU utilization of L7-filter in native Linux. For all the experiments, we use one thread for preprocessing, including disk I/O, TCP/IP reassembling and defagmentation, connection buffer reassembling and scheduling. The number of threads for pattern matching is varied as represented by the X-axis in the figure. The bars represent the throughput and curves represent the CPU utilization. *T-ori* and *T-aff* illustrate throughputs of the original OS scheduling and our affinity-based scheduling, respectively. Similarly, *U-ori* and *U-aff* demonstrate CPU utilization without and with our optimization. The affinity-based multithreading shows its superiority in scalability compared to the default OS scheduler. With 7 concurrent threads, the system throughput increases by 51% compared to the naive OS scheduling. The system scales near linearly (a speedup of 6.5X when 7 threads are applied.) to the number of MTs. Additionally, the CPU utilization is also less in the affinity-based technique with more efficient cache performance. A significant reduction of last level cache misses is observed and will be presented in the next subsection. We observe the effect of providing more than 7 MTs in the 8-core machine in the last set of bars in Fig. 7. Since core #0 is always running PT, the extra MT has to compete resource with the PT on core #0. The system performance is therefore degraded.

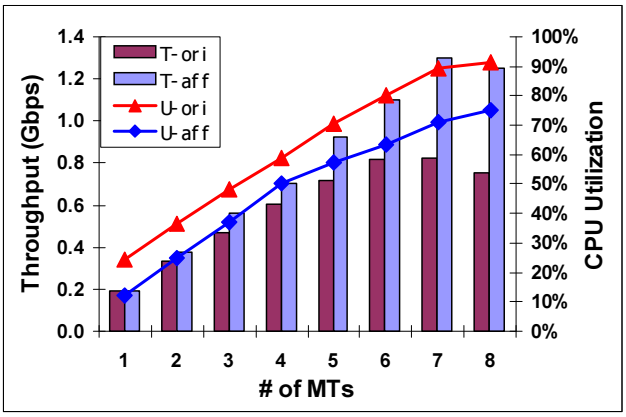

Figure 5. Throughput and Utilization in Native Kernel.

4.4 Cache Performance

As we discussed in section 3, the affinity-based multi-threaded L7-filter is expected to provide performance improvement by ensuring efficient cache utilization. Figure 6 shows that the connection-based multithreading mechanism with the proposed scheduler reduces L2 cache misses by about 50%. By default, threads are migrated by the OS scheduler to avoid imbalanced load at the price of cold cache misses. Our scheduler dispatches all the packets of the same connection to the same thread, which is affinitized to a designated core, keeping the cache warm for pattern matching.

4.5 A Life-of-Packet Analysis

In this subsection, we decompose the tuned L7-filter to study the behavior of each component with different number of matching threads. The execution time for each experiment is scaled to 100% to better represent the fractional contribution. Note that all the measurements in Figures 7 and 8 are based on the lifetime of one packet rather than the complete trace file because of the timing overlap in preprocessing thread and matching thread. While the PT runs the libnids routines, it also dispatches packets to the proper MT runqueue. In the mean time, the desired MT is also classifying connections in its runqueue in a FIFO manner. Therefore, it is necessary to use per packet profiles to explain the inter-relations among different components in the system. Due to page limitation, we provide the life-of-packet for the affinity-based system in Figure 7 and only add the statistics for 7 MTs in the original non-affinity-based L7-filter as a comparison.

Our first observation from Figure 7 is that preprocessing incurs very limited overhead, while most of the execution time is for the scheduler and pattern matching. The execution times include queuing times at individual components. With a limited number of MTs, the scheduler is very likely to stall due to the capacity limitation of the runqueues, hence takes a large proportion of the execution. On the other hand, when the number of MTs increases, more runqueues are provided to the scheduler. A packet is consequently reassembled into its connection buffer sooner, while more time is spent in MT. A large time share for MT in Figure 7 means it either gets more opportunity for connection classification, or it sleeps frequently because it

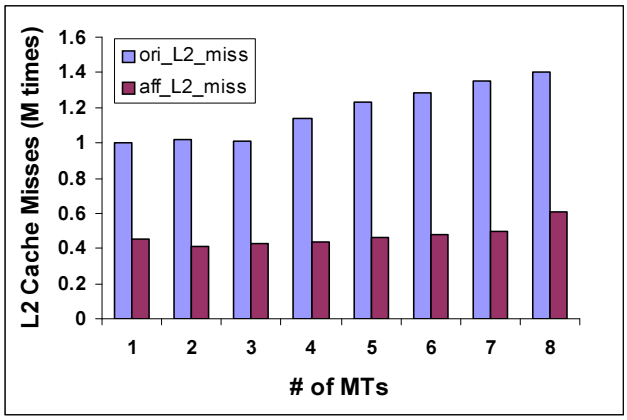

Figure 6. L2 Cache Miss.

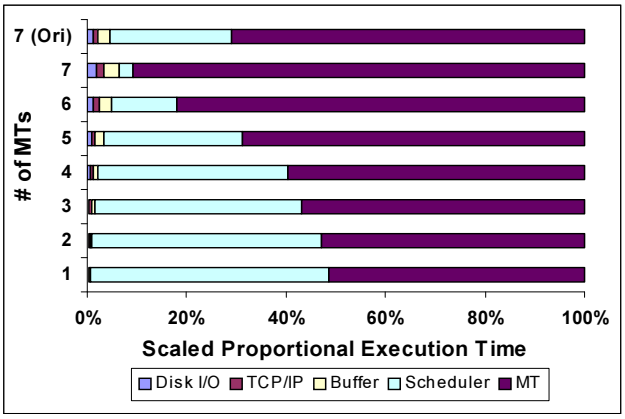

Figure 7. A Life-of-Packet Analysis.

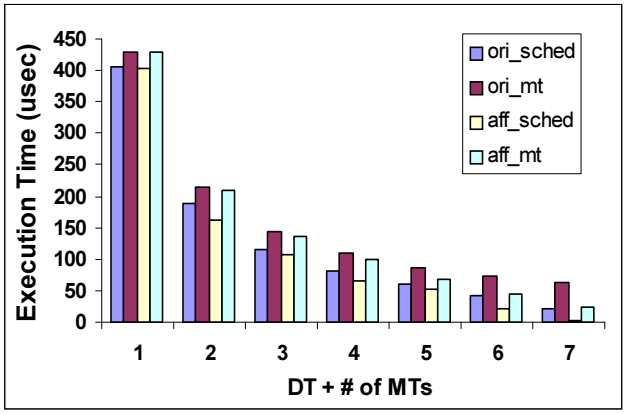

Figure 8. T-sched V.S. T-mt

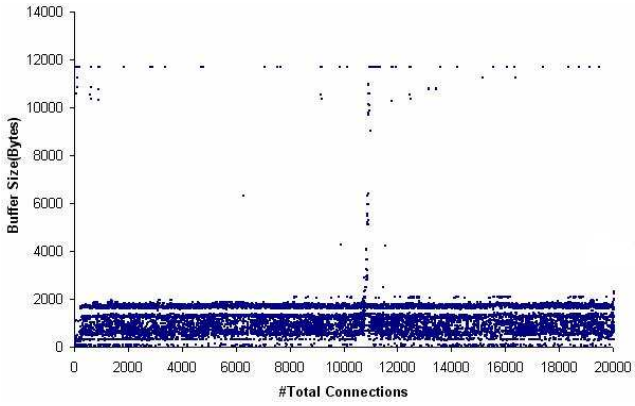

Figure 9. Memory Requirement of L7-filter.

runs so fast that its runqueue is empty very often. Figure 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 Figure 5.

As Figure 7 only presents results in percentage, we give the absolute execution time for the two largest contributors in the system in Figure 8. A consistent observation from the figure is that affinity-based L7-filter scales better than the default case, on a per packet basis. Recall that packet processing by libnids incurs very limited overhead to L7-filter and that PT reads in a packet as soon as the previous packet was scheduled to a runqueue, with no unnecessary inter-packet latency introduced. We can therefore project the data pattern in Figure 8 to the system throughput, as demonstrated earlier in Figure 5.

4.6 Memory Requirement (8 threads): Assembling Buffer Size

A significant concern about application performance in a router/switch is memory requirements. A large memory requirement not only incurs overhead for intensive memory accesses, it also costs extra software and/or hardware unit to provide memory management. We therefore conduct experiments to measure the memory bound for our affinity-based multithreading system.

In the experiment, we measure the total size of connection buffers every time a new connection buffer is scheduled to a runqueue of an MT. We used all the core resources with 1 core specified for PT, and the other 7 cores each running 1 MT. The last section of the trace is pruned to simulate the real scenario in the network (packets keep coming, so that there is no system down time). As Figure 9 illustrates, despite some scarce growth to 12 KB, the required memory is around 2 KB. Since a magnitude of KB in memory is acceptable for a router application, we believe our model is very practical to be implemented in a router.

5. RELATED WORK

5.1 Optimizations for DPI

There are a rich set of literature on accelerating DPI for edge (e.g. router/switch) nodes. Most of these literatures focus on the algorithm of regular expression pattern matching, from both software and hardware perspectives.

For software solutions, [7, 18, 26] focus on optimizing the representation of regular expression. They developed different techniques to take advantage of the spatial locality of Non-deterministic Finite Automaton (NFA) and temporal locality of Deterministic Finite Automaton (DFA). For hardware solutions, For hardware solutions, [12] implements an NFA-based regular expression engine on an SGI Altix 4700 workstation with FPGA support, and significantly improves the throughput of NFA while maintaining a compact memory requirement. Piyachon et al. [19] chose Network Processor (NP) as the deployment platform for fast DPI.

However, our work focuses on the efficient utilization of multi-core servers to improve overall system performance and scalability. L7 Filter uses NFA-based GNU Regular Expression engine. The stateless NFA requires duplicated packet copies to maintain the status of pattern matching for a given connection. Hence, the efficiency of cache utilization directly influences system performance. Our optimizations are based on the understanding of program characteristics and OS support for multi-core architectures. The solutions we have proposed (connection-level parallelism plus the core affinity) can be easily deployed on industrial products, independent of pattern matching mechanisms.

5.2 Scheduling in OS

Aside from our baseline O(1) scheduler, coscheduling [1, 5, 14] also favors SMP performance. With coscheduling, processes of a parallel job run at the same time across processors in either an explicit or implicit manner, depending on whether synchronization is conducted globally. The latency of inter-process synchronization and communication is the major bottleneck for coscheduling. Despite that implicit coscheduling alleviates this problem with suboptimal local knowledge, it sacrifices the accuracy of scheduling decisions.

Compared to coscheduling, our connection affinity based scheduling mechanism incurs very little performance overhead. This is because our scheduler only idles when MTs are busy. When our scheduler runs, it only polls for the depth of the runqueue in each MT. In fact, the more MTs run for classification, the less overhead the scheduler introduces.

6. CONCLUSION AND FUTURE WORK

In this paper, we developed a multithreaded programming model for L7-filter to exploit the connection level parallelism in packets. We proposed a thread scheduling technique on a multi-core architecture based on cache affinity and showed that the throughput is increased by 51% and core utilization is reduced by 15% compared to native Linux. Our experimental results were based on the configuration with one preprocessing thread running on core #0 and one matching thread on each of the remaining core. Our maximum throughput is close to linear speedup compared to the sequential version. We also conducted a life-of-a-packet analysis to analyze latencies due to different stages of L7-filter processing. This unique analysis showed that CPU time is more effectively distributed to the pattern matching threads in our design, which consequently eliminates the latency of scheduling stalls.

In the future, we plan to apply our design to the real-world network products. Our immediate deployment candidates will be on a Cisco switch and a Cisco appliance. We will further optimize our multithreaded L7-filter on non-x86 based multi-core processor architectures. We also plan to make our multithreaded L7-filter software available to the open source community.

7. AKNOWLEDGEMENT

This project is partially supported by a Cisco University Research Grant, NSFC (60573121, 60625201), and 863 high-tech (2007AA01Z216, 2007AA01Z468).

8. REFERENCES

[1] C. Anglano, "A Comparative Evaluation of Implicit Coscheduling Strategies for Networks of Workstations", HPDC 2000.

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

[3] Cisco Internetworking Operating System (IOS) IPS Deployment Guide, http://www.cisco.com.

[4] Cisco Systems, Inc., “Application-Oriented Networking: Products and Services”, http://www.cisco.com/en/US/products/ps6692/Products_Sub_Category_Home.html.

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

[6] A. P. Foong , et al., “An In-depth Analysis of the Impact of Processor Affinity on Network Performance”, IEEE International Conference on Networks (ICON), 2004.

[7] C. Hayes and Y. Luo, “DPICO: A High Speed Deep Packet Inspection Engine using Compact Finite Automata”, ANCS 2007.

[8] L. Kencl, et al., “Adaptive load sharing for network processors”. INFOCOM 2002.

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

[10] Libnids, http://libnids.sourceforge.net/.

[11] MIT DARPA Intrusion Detection Data Sets, http://www.ll.mit.edu/IST/ideval/data/2000/2000_data_index.html.

[12] A. Mitra, W. Najjar and L. Bhuyan, “Compiling PCRE to FPGA for Accelerating SNORT IDS”, ANCS 2007.

[13] I. Molnar, “Goals, Design and Implementation of the New Ultra-Scalable O(1) Scheduler”, Linux Kernel, April 2002. Docomentation/sched-design.txt.

[14] S. Nagar, "A Closer Look At Coscheduling Approaches for a Network of Workstations", SPAA 1999.

[15] G. Narayanaswamy, et al, “An Analysis of 10-Gigabit Ethernet Protocol Stacks in Multicore Environments”, HOT Interconnects 2007.

[16] O(1) scheduler, http://www.ibm.com/developerworks/linux/library/l-scheduler/.

[17] Performance Application Programming Interface (PAPI), http://icl.cs.utk.edu/papi/.

[18] P. Piyachon and Y. Luo, “Compact State Machines for High Performance Pattern Matching”, DAC 2007.

[19] P. Piyachon and Y. Luo, “Efficient Memory Utilization on Network Processors for Deep Packet Inspection”, ANCS 2006.

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

[21] G. Regnier et al, “ETA: Experience with An Intel Xeon Processor As A Packet Processing Engine”, HOT Interconnects 2003.

[22] G. Regnier et al, “TCP Onloading for Data Center Servers”, IEEE Computer 2004.

[23] W. Shi, et al., “Load Balancing for Parallel Forwarding”, Transactions on Networking, 13(4):790-801, Aug. 2005.

[24] W. Shi, et al., “Sequence-preserving Adaptive Load Balancers”, ANCS 2006.

[25] B. Veal, et al., “Performance Scalability of a Multi-core Web Server”, ANCS 2007.

[26] F. Yu, et al., Fast and memory-efficient regular expression matching for deep packet inspection, ANCS 2006.