UNIVERSITY OF CALIFORNIA
RIVERSIDE

Multicore Scheduling for Network Applications Based on Highest Random Weight

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Danhua Guo

June 2010

Dissertation Committee:
Dr. Laxmi N. Bhuyan, Chairperson
Dr. Chinya Ravishankar
Dr. Rajiv Gupta

The Dissertation of Danhua Guo is approved:

Committee Chairperson

University of California, Riverside

Acknowledgements

Looking back on my journey of research and completing this dissertation, I feel lucky and grateful for the endless support from many people. Had it not been for their help, I would not have been able to reach this point.

First and foremost, I owe the greatest debt of gratitude to my advisor, Dr. Laxmi N. Bhuyan, for his guidance and continuous support and belief in me over the past five years. Dr. Bhuyan exemplifies a distinguished scholar, a motivating advisor, and a true friend. He guided me to embark on the journey of research, challenged me to think analytically, and encouraged me to be exceptional. The time span of my Ph.D. is limited, but the influence of his scholarship attitude will be my lifetime assets. I am also grateful to have the privilege of having Dr. Chinya Ravishankar and Dr. Rajiv Gupta to serve on my dissertation committee. Their constructive suggestions and support help to improve the quality of this dissertation.

I would like to thank Dr. Jason Ding for his guidance during my internship at Cisco. The enlightening conversation with him in research and in life made my internship memorable and helped me with my career decision.

I would also like to thank former and current members of the architecture lab at UCR, Jia Yu, Jiani Guo, Jingnan Yao, Satya Mohanty and Guangdeng Liao, for their helpful feedback on my research, and their companionship during my stay at UCR.

Last but certainly not the least, I dedicate my accomplishment to my parents. Without their endless love and support throughout my life, I will not be what I am today.

ABSTRACT OF THE DISSERTATION

Multicore Scheduling for Network Applications Based on Highest Random Weight

by

Danhua Guo

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June 2010
Dr. Laxmi N. Bhuyan, Chairperson

The widening spectrum of network applications incurs increasing stress on physical resources for both the network infrastructure and the web servers. Meanwhile, the emergence of faster Ethernet has shifted the bottleneck of network performance to the processing capability of the web servers. This trend has driven the prevalence of Chip Multiprocessors (CMP, a.k.a. multicore). However, even running on the state of the art multicore web servers, the network performance still falls short of expectations.

In this study, we optimize multicore scheduling in both the OS kernel and userspace for three legacy network applications, i.e. Deep Packet Inspection (DPI), multimedia transcoding and SPECweb2005. In the OS kernel, we propose an interrupt affinity based scheduler to prevent starvation by separating interrupt handlers from userspace application. In the userspace, we first parallelize the network application and then propose an affinity based scheduler that affinitizes all the packets in the same connection to the same core. However, this scheduler is oblivious of load balancing, which can offset the cache benefits. We therefore propose several hash based schedulers to strike a balance between connection locality and load balancing. While the baseline Highest Random Weight (HRW) hash balances workload at the connection level, our Adjusted HRW (AHRW) achieves packet level load balancing by comparison of runqueue length of each core. In addition, we enable cache awareness of AHRW by means of a communication matrix in Cache-Aware AHRW (CA-HRW), and propose a hierarchical version, H-CAHRW, for different core/cache topologies. To incorporate QoS concerns, we also develop a Proportional Share HRW scheduler, PS-HRW, by allocating cores to each connection based on connection buffer size. We implement and verify all of our schedulers using real application measurements.

With the resurgent interest in system virtualization, we present a performance characterization of a virtualized multicore server under consolidated network workloads and show that L2 cache misses are the major bottleneck. We therefore optimize the virtual CPU migration policy to take advantage of the cache topology. Then, we port all our schedulers developed in the native system to a virtualized multicore server, and observe minimum performance degradation.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The growing variety of network applications has demanded scaling physical resources not only in network bandwidth but also in the processing capability of the backend servers. For example, Deep Packet Inspection (DPI), widely applied in ISP devices to monitor spiraling volume of network traffic, requires significant computing and memory resources to complete pattern matching of the packet payload [4, 81] at wire speed. On the other hand, multimedia applications have real time "on-demand" characteristic of transcoding that requires significant processing capability. In addition, as more and more backend processing has been moving to the "cloud" network [18], it is necessary to distribute the workload and utilize the shared network bandwidth and processing resources efficiently. The distribution must also ensure that Quality of Service (QoS) is guaranteed.

Fortunately, innovations and technology have evolved in different directions to fulfill the demand of network applications. At the infrastructure level, the emergence of high speed Internet resolves the performance limit on the network bandwidth. We have seen the prevalence of Gigabit Ethernet for home users, 10 Gbps and 40 Gbps uplinks for ISPs, and the forthcoming 100 Gbps Ethernet ready to be shipped from the hardware vendors. Consequently, the performance bottleneck has shifted from the network to the processing end. Chip-level multiprocessors (CMPs), i.e., multicore processors, have arguably become the *de facto* platform for modern web servers. The reasons behind this trend are two folds. On the one hand, the increase in CPU clock rate can no longer keep up linearly with the growing network bandwidth [70]. On the other hand, the cap of Instruction Level Parallelism (ILP) in network applications drives the target of hardware development from more powerful single core chip to multiple slower cores on chip.

The increasing processing power of multicore servers has not only satisfied the resource demand of network applications in native systems, it also shows great potential in running more OS boxes on the same physical machine. Despite the long envisioned benefits, Virtual Machine (VM) technology has regained its popularity only recently, when multicore chips can provide enough processing capability to for performance isolation, manageability and scalability. VM technology implements a management layer, Virtual Machine Monitor (VMM), sitting on top the native operating system. VMM monitors and schedules all guest VMs, giving them an illusion to be running directly on the physical devices. As the processing power no longer stays the bottleneck of network processing, virtualized multicore servers are expected to run applications that are more diverse and complicated. Essentially, we see a trend toward server consolidation, i.e. running different types of applications on the same physical machine. While this additional VMM layer makes VM a promising solution to server consolidation, the I/O processing overhead gets more complicated in the new context of consolidated workloads and multicore servers.

Despite the fact that multicore servers provide powerful processing capabilities for various network applications, the overall performance still falls below the requirement to satisfy the ever increasing network bandwidth. In the following sections, we will discuss some major challenges facing the deployment of multicore web servers.

1.1 Challenges in Modern Network Applications

Among the wide spectrum of network applications, an interesting characteristic is the hierarchical relationship between packets and connections - packets in the same connection usually share the header data. By exploiting the above locality between packets belonging to the same connection, we can schedule them efficiently in multicore servers to boost throughput and reduce latency. We pick two popular representatives of the network applications, which have high computational demand, are widely studied and productized and describe their locality characteristics below.

L7-filter is a Deep Packet Inspection (DPI) program for Linux and is becoming increasingly popular in both academia [10, 25, 26, 29, 43, 85] and industry [17, 30, 38] as a complement to the traditional header based packet inspection. It classifies packets based on application layer payload and is of great importance in traffic monitoring when port numbers are purposely hidden or require dynamically allocation. The major bottleneck of DPI is pattern matching, which lags far behind wire speed. Most of the current research focuses on the algorithm of pattern matching, leaving the deployment architecture untouched. As the processing capability of web servers evolves at an unprecedented speed, we believe studies in the scheduling of multithreaded DPI programs complements well with the algorithm studies.

Multimedia transcoding is a promising solution to customize the size of objects and distribute the available network bandwidth among Internet clients with various hardware resources, software sophistication and quality of connectivity [2, 23, 24, 27, 28, 63, 90]. A traditional solution to this problem is to store multiple copies of the source stream on the media server and select a copy according to some initial negotiation with the client. While this solution works for small scale multimedia requirements, it is infeasible as the demand has moved to the real time scale which requires faster processing and higher accuracy at a finer grain. In transcoding, based on the network bandwidth of each client, the transcoding algorithm responds to the client request accordingly. On-demand transcoding has been proposed to transform media streams in the active routers to adapt media streams to fluctuating network conditions [27, 28]. It is also widely adopted to satisfy real time requests by users using embedded systems, such as cell phone and netbooks. Because these devices usually requires a video version of small frame size and low bit rate, the video needs to be transcoded on an active router to fit the client's device.

As servers possess more processing capability, we see a trend of running multiple applications concurrently to fully explore the server. SPECweb2005 is a consolidated benchmark merging computation, database as well as I/O operations. The performance of SPECweb2005 is a direct test of the capability of web server under real workload pressure. No previous work has reported any thorough analysis at the architectural level of server consolidation under real measurement.

1.2 Challenges in Multicore Scheduling

The goal of scheduling is to achieve high throughput and low latency while maintaining QoS requirements if necessary. Unfortunately, while more multicore processors are shipped to web server vendors to provide increasing processing density on chip, the performance is not as good as expected. Experiments showed that the 8-core test server running a modified SPECweb2005 [82] workload achieved only a 4.8X speedup in throughput (compared to the ideal 8X) [88]. In another experiment on a Sun Niagara 2 based web server, which resides 16 independent pipelines, the system throughput is only increased by a factor of 10 instead of the ideal 16 [26]. The major challenges of multicore scheduling include:

1) Lack of parallelism. Most of legacy network applications such as DPI or multimedia were developed using single thread. Therefore, these applications cannot fully explore the core resources. In our study, both L7-filter and multimedia transcoding are originally written as a single thread program.

2) Ignorance of application characteristics. The intuitive data partitioning at the packet level in parallelization fails to consider the packet affinity within the same connection. The architecture of each multicore processor usually carries both logical and physical core topology. The location of the cache hierarchy is mapped onto the core topology to ameliorate the expensive memory access. Failing to schedule workload affinity to a right cache affinity causes unnecessary waste of miss penalties in the cache.

3) Load imbalance. While favoring packet affinity benefits from reusing shared data in the cache, the workload distribution can be significantly uneven when each connection

has a different number of packets. Paper [26] shows that the load imbalance on a Sun Niagara 2 processor that has a large number of threads, the benefits of cache affinity can be offset by load imbalance. On the other hand, a non-work-conserving QoS scheduling mechanism inevitably restrains processes from using idle CPUs because of the allocation cap.

4) QoS requirement. QoS scheduling allocates a proportional share of the processing resources to each process according to the weight of the process. General Processor Sharing (GPS) [19, 42, 64, 65] was a theoretically ideal scheduler that provides QoS guarantee. The GPS scheduling was also extended to multiple links scenario [9]. While GPS and its extension are a theoretically ideal scheduling policy to ensure fairness in multicore scheduling based on the weight of each process, it is impractical to implement in real systems. In addition, QoS guarantee usually sacrifices cache locality and load balancing [27, 28]. We have not seen any previous studies that consider this trade off.

5) Tradeoff between overhead and heuristics. While schedulers can achieve better performance with additional computation for more heuristics, its overhead should not contend with other processes for physical resources. Specifically in the network application domain, the scheduling frequency is an important factor that affects the system performance. Given this performance constraint, the multicore scheduling should balance between application characteristics, QoS requirement and load balancing.

1.3 Challenges in System Virtualization

While the emergence of virtualization has been a promising solution towards sever consolidation and cloud computing, the virtualized network performance lags significantly behind the performance in native systems operating directly on physical devices. The lack of performance characterization of multicore virtualized networking clouds the study of VM performance and delays the production of useful optimizations for the problem.

Most of the previous studies [45, 50, 51] in VM are limited to some simplistic benchmarks such as *Iperf* [34] or *Netperf* [58]. Their experiment results are therefore valid in a limited context. On the other hand, multicore servers possess extensive processing power that can be exploited for parallelization within an application or among multiple applications. In addition, shared memory multicore servers usually include a multi-layer cache topology to alleviate the memory wall problem.

Quite a few research groups have proposed different optimizations for virtualized network I/O [16, 47, 50, 51, 61, 69, 91]. Even though these optimizations claimed to improve virtualized I/O in their own context, none of them focuses their research on multicore systems, especially the cache topology. Since VM introduces the additional mapping between virtual to physical devices, it complicates the research results in multicore scheduling in native operating systems. This is not saying we should start a brand new area of VM study, but rather, we need to pinpoint the bottleneck of VM performance and build VM optimizations on top of those in native systems.

1.4 Overview of the Research

The goal of this study is to build a generic scheduling mechanism that improves system throughput by fully utilizing the parallelization resources on multicore web servers. To achieve this goal, we propose solutions to all of the challenges mentioned above in the following manner: scheduling optimizations in the kernel, parallelizing userspace network applications and affinity based scheduling, userspace hash based scheduling optimizations, and scheduling optimizations in virtualization.

In the **first** aspect of this study, we find that userspace processes can be starved when the interrupt rate is high due to the heavy traffic in high speed network. We further observe that the bottom half of interrupt handler is nonpreemptable, causing userspace processes to wait on interrupt handlers and will be delayed until all the interrupts are processed. Therefore, we enable the kernel to schedule interrupt handlers and application layer processes to different cores to prevent resource competition. We propose to separate interrupt processing from application processing and statically affinitize them to cores that share a last level cache. This idea is similar to the TCP onloading [70, 71] technique in the OS. However, in our onloading proposal, it is not necessary to modify the entire TCP stack to separate userspace applications from kernel processing.

The **second** aspect of our study is to parallelize userspace network applications by partitioning incoming traffic at the packet level. Among the three applications we choose in this study, L7-filter and transcoding were both originally single threaded. As both applications require a significant amount of computation and memory, parallelization is necessary to fully utilize all the core resources. We develop a trace driven model for both applications to decouple noises in the network from the performance bottleneck. This model helps the study of processing requirements in a well-controlled environment without noises from network infrastructure and kernel stack. Then, we observe that naive parallelism does not fully utilize all the computing resource on a multicore server. We therefore propose a processor affinity based scheduler to focus on connection locality. Essentially, we assign packets belonging to the same connection to the runqueue of the same thread, which is dispatched to a dedicated core in multicore server. Similar to Receive Side Scaling (RSS) [73] for the NIC, our scheduler works in software and provides faster packet classification with good scalability in multicore servers.

However, maintaining connection locality sacrifices the load balancing, core topology as well as QoS requirement in workload scheduling. Therefore, in the **third** aspect of this study, we design several hash based schedulers to balance between these concerns. The baseline hash is a robust hash function called Highest Random Weight (HRW) [74, 86], which only guarantees load balance at the connection level. A major benefit of hash is the uniform distribution of the input key to the hash space. HRW uses connection ID based on 4-tuple information, and hashes it with server ID to generate a weight for each server. The server with the highest weight will be the host of selection for the current packet. However, when the connection size varies, HRW scheduling inevitably incurs load imbalance. As a result, we propose an Adjusted HRW (AHRW) to balance the workload at the packet level by considering the runqueue length and maintaining the connection locality based on the Minimum Disturbance Rate property of HRW. A similar technique was adopted to develop an adaptive load balance technique for network processing based on processor utilization [40, 41]. However, designing a system based on processor utilization is difficult to implement. Additionally, our AHRW is a robust hash that maintains the connection locality by the famous minimum disruption property [40, 41, 74, 86].

Albeit the advantages of AHRW, two essential problems still remain untouched: 1) cache-awareness in multicore architecture with heterogeneous inter-core communication cost and 2) a generalized scheduler, for different core/cache topologies with both cache awareness and adaptive load balancing mechanisms. Therefore, we propose CA-AHRW, a Cache-Aware AHRW hash scheduler. This novel scheduler is designed for different core/cache topologies and is able to maintain both the connection and the packet level load balancing with cache-awareness while maintaining connection locality. Specifically, we construct a communication matrix for any given multicore architecture characterizing the heterogeneous inter-core communicating overhead. Then, we obtain a weighted queue vector for each core based on this matrix and apply the weighted queue vector as a multiplier to the original AHRW to adjust the hash value.

In addition to single layer hash schedulers, we extend our solutions to a hierarchical environment, where the extensive parallelism resources on multicore servers 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 $O(lgN)$, where N is the number of scheduling candidates. We verify our Hierarchical CA-AHRW (H-CAHRW) scheduler on real web servers using different multicore architectures, including Intel Xeon [33], AMD Opteron [1] and Sun Niagara 2 [79]. We show that the system throughput can be improved by 50.7% on average compared to the connection locality heuristic and by 19.8% on average compared to the linear AHRW scheduler. H-CAHRW can be applied to different multicore servers with hierarchical locality.

As to the QoS requirement, we propose a two-level proportional share scheduler, Proportional Share HRW, PS-HRW, based on H-CAHRW. In the first level, we use H-CAHRW to pick integral number of cores with the highest weights for the incoming traffic. The residual fractional request (smaller than one core) is handled in the second level. In the next level, we apply an influential vector $X = (x_1, x_2, \dots, x_N)$ to H-CAHRW so that the scheduled request on each core follows the capacity vector $P = (p_1, p_2, \dots, p_N)$. To derive the influential vector, we assign the cores selected in the first level with capacity $p_i = 1/N$, and the rest of the cores with capacity $p_i' = residual\_value/N$. Our method follows a proof in [74] for heterogeneous robust hash mapping. Compared to [27, 28], PS-HRW incorporates cache locality and load balancing factors into QoS decision.

The **fourth** aspect of our study is to verify the performance of our scheduler in a virtualized environment. Because of the additional mapping between virtual to physical resources, what proven to be correct in the native system might not be as effective in virtualization. In our VM study, we first conduct a thorough performance evaluation of multicore virtualized network performance and then propose two scheduling optimizations in the VMM. We choose the SPECweb2005 [82] benchmark suite. The three different types of workloads in SPECweb2005 simulate the realistic network traffic processed by web servers in the real world. We present the performance evaluation for a consolidation scenario where database transaction, computations and I/O processing applications run concurrently on the server. Our profiling breaks down the overhead at the architectural level, i.e. CPI, TLB, L1 instruction and data cache, and L2 cache. We then relate the collected hardware performance to the VMM scheduler by profiling the

VMM software at a detailed function level. We apply a unique "Life-of-Packet" analysis to break down the processing latency of a packet along the reception path. To the best of our knowledge, this study carries the most in-depth hardware profiling of a virtualized multicore server running consolidated workloads. We find that the current VMM scheduler is oblivious to the underlying cache topology and hence incurs a significant overhead. Our findings are in line with a previous study [3] that calls for attentions in the cache deployment for VM. However, instead of advocating a new (asymmetric) cache architecture, we propose an optimization for the current VM scheduler to be aware of the underlying cache topology in addition to all the schedulers we proposed in native OS.

Specifically for the interrupt affinity scheduler, similar to the kernel study in native OS, we apply the interrupt affinity scheduling in virtualization by binding all the virtual CPUs that handles interrupt processing to one core and the rest of the VCPUs to another core. We observe an average of 12% improvement in throughput for the *Iperf* benchmark compared to the default system scheduler (in OS or VMM) that works in a round-robin fashion. For the consolidated SPECweb2005 benchmark that consumes more than two cores for execution, we propose to modify the VMM scheduler to dynamically favor cores in the same cache domain when Virtual CPUs (VCPUs) are migrated. Results show an average of 15% improvement in supported concurrent sessions for the three benchmarks in SPECweb2005 suite. We further verify our optimization by showing the reduction of cache misses in the benchmark execution. We believe these findings will be useful for VMM and platform architects as they provide insights and solutions to the performance bottleneck and possible optimizations of virtualized I/O in multicore architectures.

1.5 Outline and Contributions

This study addresses the challenges in multicore scheduling for network applications and provides several effective solution to improve the system performance in terms of throughput, load balancing as well as QoS concerns. The contributions of this study can be summarized as follows:

- We break down the packet processing into kernel and userspace segments and propose several schedulers in both segments. In the kernel, we propose an interrupt affinity based scheduler to save userspace processes from starvation due to the original non-preemptive interrupt processing. The research is presented in Chapter 3.
- In the userspace, we design and implement parallel algorithms for L7-filter and transcoding at the packet level. We also develop a trace-driven model for network applications to decouple the performance bottleneck from noises. This model helps the study of processing requirements in a well-controlled environment without noises from network infrastructure and kernel stack. In addition, we propose an affinity based scheduler to take advantages of cache benefits and then The research is presented in Chapter 4.
- In the userspace, we propose several HRW hash based schedulers, such as AHRW, CA-AHRW, and H-CAHRW to improve system throughput by balancing between

cache benefits, load balancing, core topology in different multicore architectures, and system overhead. The research is presented in Chapter 5 and 6.

- We propose a QoS aware proportional share scheduler, PS-HRW, based on the previous HRW schedulers. It provides QoS guarantees for incoming network requests following the assigned weight. Compared to research [27, 28], PS-HRW incorporates cache locality and load balancing factors into QoS decisions by using H-CAHRW. The research is presented in Chapter 7.
- We present the first performance evaluation for virtualized multicore system under a consolidated SPECweb2005 network workload. We profile the benchmark execution by collecting hardware events and break down the overhead of virtualization at a function level. We relate the performance overhead to the obliviousness of the cache topology of the current VMM scheduler, and propose an additional dynamic VCPU scheduler in the VMM to migrate a VCPU to the PCPU in the same cache domain as the original PCPU. The research is presented in Chapter 8.
- We verify our proposed schedulers in both native system and virtualization. In our experiments, we choose a wide variety of multicore platforms including Intel Xeon, AMD Opteron and Sun Niagara 2. Cross platform/OS experiments give us confidence in the efficacy of our proposals. We have released all our source code online and maintain a website for further development and collaboration at: http://www.cs.ucr.edu/~dguo/.

Chapter 2

Background and Related Work

2.1 Research in DPI

The intensity of network resource competition increases as a greater number of applications demand high bandwidth and more computing capability. Consequently, the Quality of Service (QoS) in the network domain requires faster and scalable classifiers to distribute resources based on application priori-ties. 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 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, Deep Packet Inspection (DPI) plays a key role in bandwidth management and traffic reshaping, and is increasingly used to augment network security and underpin service creation and service management tools.

Fig. 2.1 illustrates the structure of a Linux networking system with L7-filter in an OSI model context. 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.

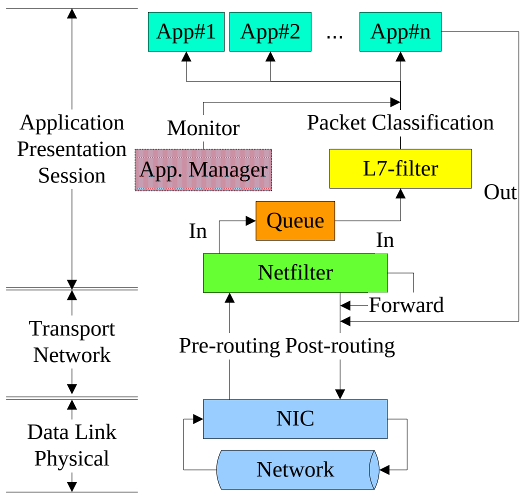


Fig. 2.1: L7-filter in OSI Model

By matching against signature fields of various protocols, L7-filter uses GNU regular expression 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 [4, 10, 43, 85]. Thus, optimization effort is needed.

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 [10]; 2) increasing throughput by processing multiple input characters per clock cycle [29, 43]; and 3) balancing between the memory bandwidth and memory size requirement [93]. Another direction of the research studies the deployment of DPI programs, i.e. how to use hardware accelerators. In this domain, both FPGA [53] and Network Processor [40, 41] solutions have been proposed to explore the packet level parallelism inside DPI program. Tan [85] proposed a "bit-splitting" architecture to explore the internal parallelism inside of the state machines.

As multicore web servers become the mainstream plat-forms for network appliances, research has increasingly pro-poses to deploy DPI programs using general purpose multicore servers. Authors in paper [89] discussed the possibility of parallelizing SNORT [81] using multicore servers with a 3-level feedback system. We [26] designed a multithreaded L7-filter on an 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 multicore 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.

2.2 Research in Multimedia Transcoding

Online transcoding is a popular web service to provide various clients with the media source according to their demand. Many researchers [13, 14, 23, 24, 63, 90] have addressed how to customize the multimedia contents to match user preferences or the diversity of network conditions and display devices. Chandra [13] used JPEG transcoding techniques to customize the object size, thus allowing a web server to dynamically allocate available bandwidth among different classes. Fox [23] dynamically distilled the web page content on active proxies. They also implemented a cluster-based web distillation proxy called TranSend [24]. However, their scheduling schemes emphasize fault tolerance more than load balancing, and parallel processing of a single stream is not considered. Researchers [63] and [90] proposed the concept of CLuster-based Active Router Architecture (CLARA), where a computing cluster is attached to a dedicated router. With CLARA, the multimedia transcoding tasks are processed in parallel. More recently, multicore web servers have become increasingly popular in processing multimedia applications. As a result, a plethora of scheduling strategies has been proposed to efficiently use the bountiful resources.

Simple static policies, such as random [75] and Round-Robin (RR) [39] are adopted in practice because they are easy to implement. Guo [28] evaluates three load sharing schemes, namely RR, Stream-based Mapping (SM) and Adaptive Load Sharing (ALS). They showed that RR is simple and fast, but it suffers from out-of-order problem. SM achieves good performance in terms of both throughput and video quality, but its advantage is limited to evenly distributed flow. Based on [41], ALS achieves better unit order in output streams, but it also involves higher overhead to map the media unit. Guo [28] further proposed two prediction-based schedulers, namely Prediction-based Least Load First (P-LLF) and Prediction-based Adaptive Partitioning (P-AP). P-LLF is based on [44], which always chooses a least loaded server to process the next media unit. Although the load balancing is well maintained, no flow locality is preserved. P-AP, however, strikes a good balance between flow locality and packet level load balancing by partitioning the servers into several subsets and establishes a good mapping between flows and servers. However, it does not consider core/cache topology on multicore servers, limiting its performance improvement.

2.3 Optimizations in Multicore Architecture for Network I/O

TCP onloading [70, 71] uses part of the computation resource in a cluster server or special Processing Engines (PE) to provide TCP/IP processing service, while the host CPU is left to continue executing application processes. Although the idea of TCP onloading sounds intuitive, most of such available designs require a large amount of change in the operating system, particularly in the TCP/IP protocol stack. The interconnection between host and PE is the key to this problem. Solutions like polling and Direct Transport Interface (DTI) [71] proven to be efficient although such designs require non-negligible modifications to the operating system. Intel's Embedded Transport Acceleration (ETA) project partitions the system between the host and packet processing engine using DTI. DTI is used at the place where Linux kernel manages processes and kernel context. Paper [94] designed an onloading model using Network Processor (NP) as the packet processing engine, and the interconnection between Mirco-Engines (ME) works as the communication channel between host and PE.

Because TCP/IP stack processing switches at a certain frequency between interrupt and process context, it is not easy to find the cutting point for binding. All the previous solutions simply work around this problem by moving the whole TCP/IP stack to a separate PE, which introduces modification to the original OS. With more in-depth studies in the operating system, particularly the TCP/IP stack and the scheduling mechanism on multicore systems, TCP onloading remains a promising solution to the network-CPU speed mismatch problem.

2.4 Load Balancing 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 web servers, both techniques have been well studied. Load balance guarantees no single bottleneck in the multicore environment and was recently evaluated using Nash equilibrium [15] and ranked elections [36] models. A detailed survey of general load balancing algorithms was provided in [40]. On the other hand, the flow-based scheduling reuses the common data in the packet headers and was productized by Microsoft in their Receive Side Scaling (RSS) NIC technology [73, 88]. However, our approach explores opportunities to strike a balance between load balance and connection locality. In addition, the load balanced provided in our scheduler is at the packet level rather than the traditional connection level. This is particularly useful when network traffic follows the "packet train" arrival pattern, as described in paper [37].

Hash functions generate independent, uniformly distributed variables. It provides theoretical load balance over the input key-value mappings. Hash functions have been widely adopted in the scheduling domain. Researchers [12, 68, 78] have applied CRC hashing in scheduling packets in parallel network systems. Among all the hash functions, HRW stands out due to its proven flow-level load balancing, flow locality and minimum flow remapping. There have been many HRW-based scheduling schemes [26, 74, 86]. It is originally proposed in [74, 86]. [40, 41] applies it to map client requested objects into the web cache in the client-server model. In the client-server model, hash functions are a favorable choice to map client requested objects into the web cache [74]. HRW and Toeplitz [73] hash are very popular in both academic studies and industrial products. The major drawback of hash mappings is that they are not adaptive to real time performance feedback and therefore are potentially vulnerable to traffic locality. In our proposed scheduler, it takes advantages of the randomness of the HRW hash, meanwhile adjusting the weight function by a feedback vector to provide load balance at the packet level.

2.5 QoS based Scheduling

Provisioning of a shared server with guarantees is an important scheduling task that has led to significant work in a number of areas including packet scheduling. The choice of an appropriate service discipline at the nodes of the network is the key to providing effective flow control. A good scheme should allow the network to treat users differently, in accordance with their desired Quality of Service (QoS), i.e. predefined CPU/network bandwidth requirement. However, this flexibility should not compromise the fairness of the scheme, i.e. one user should not be able to affect the performance to another user in the network, to the extent that the performance guarantees are violated.

Among this class of scheduling, Generalized Processor Sharing (GPS) provides an ideal QoS guaranteed scheduling in theory [19, 42, 64, 65]. The basic idea is to assign a portion of total outgoing bandwidth to each request class and ensure that each class does not overuse its share. Demers [19] introduced a Packetized GPS (PGPS, a.k.a. WFQ) that for the first time realized the theory in real practice. Parekh et al. proved in [64, 65] that PGPS closely approximates the ideal GPS in terms of packet delay and cumulative per-flow service. In [8], Bennet et al. observed that the service provided to a flow under PGPS may unboundedly exceed the amount of service received under GPS and indicated that this could lead to unfairness. Later, Blanquer [9] extended the application of GPS from a singer server system to multiple links and proposed a M2FQ scheduler that balances between fairness and service stability. Guo [28] and Chandra [14] proposed a partition based QoS scheduler that utilized the concept of proportional share in GPS. However, none of the available QoS scheduler considers the architecture of multicore servers, i.e. cache/core topology, or load balancing issues under the fairness constraint defined in [19]. In addition, the implementation difficulty of GPS-based schedulers have never been thoroughly discussed, let alone provisioning of a feasible solution. In this dissertation, we try to answer all these questions.

2.6 Network Virtualization

Network virtualization was invented and implemented in IBM’s System/360 and System/370 [80]. Each virtual machine in these initial virtualized architectures was exclusively assigned a particular set of physical devices. Data transfer relied on channel programs executing in the VMM, which ensured resource isolation.

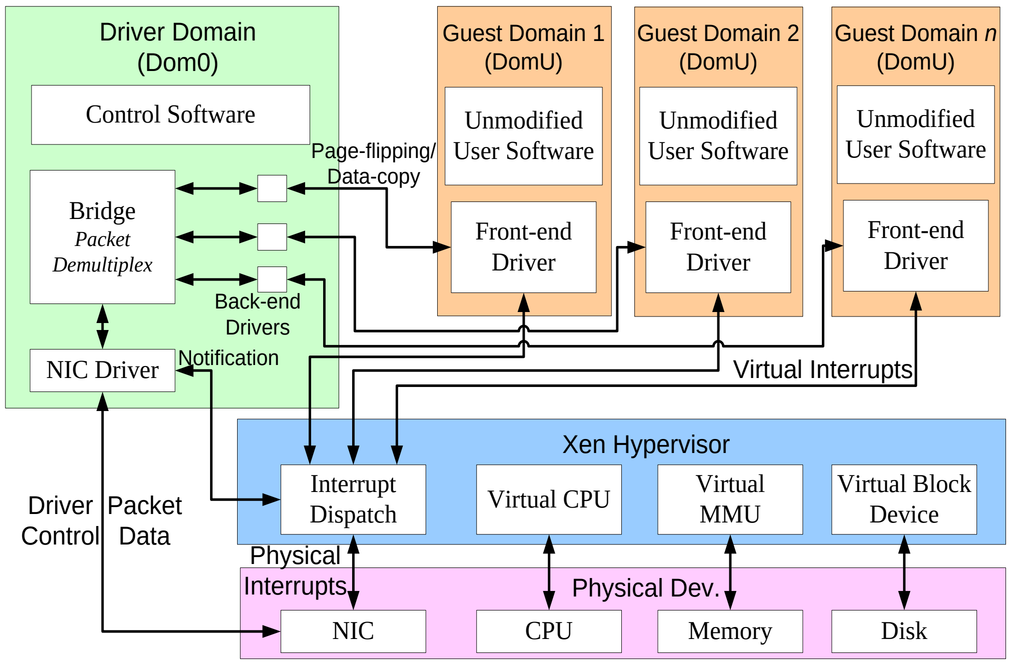


Fig. 2.2: Virtualized Network I/O in Xen

Despite the high performance through private I/O access, the costly replication of physical devices for each virtual machine limited per domain utilization. As a result, research in Xen [5] and VMware [84] designed shared access to devices and relied on a dedicated software entity to perform physical device management. This paper focused on the most popular open source virtualized system Xen.

Fig. 2.2 is an illustration of the Xen VMM. The VMM provides an abstraction layer between the VMs and the actual hardware, leaving each guest VM an illusion of running independently on native hardware. A privileged VM (driver domain or Dom0) runs a modified version of Linux that uses native Linux device drivers to manage physical I/O devices. Other VMs (guest domain or DomU) transmit and receive packets by communicating with Dom0 through shared memory I/O channels.

Once a packet arrives at the NIC, it generates an interrupt. The VMM then forwards the interrupt to the Dom0. When Dom0 acquires CPU, it DMAs the packet into the reception I/O ring. After demultiplexing the packet through the nested Ethernet Bridge to an appropriate back-end driver, Dom0 employs a data copy mechanism by default to directly copy data from the back-end driver to the front-end driver in the corresponding DomU. Once the packet reaches the front-end driver in DomU, back-end driver requests the VMM to send a virtual interrupt to notify the target domain of the new packet. Then the packet is processed from the kernel space to the user space of DomU as if it had come directly from the physical NIC.

Since the birth of VM, research in improving virtualized I/O performance never faded away. We summarize previous works into three categories: VMM architecture, VMM scheduling and optimizations in physical devices, i.e. cache.

For VMM architecture, authors in papers [50, 51] initiated an architectural profiling in Xen and introduced a faster I/O channel for inter-domain packet transfer. Compared to their works, our profiling provides more architectural metrics, leading to a more thorough understanding of Xen. In addition, Liu [47] 's VMM-bypass model and Willmann [92]'s Concurrent Direct Network Access (CDNA) architecture bypassed the VMM and overcome many of the bottlenecks in software multiplexing and demultiplexing. Researchers [69] also proposed using multi-queue in the NIC and a grant-table-reuse mechanism to improve the performance of the grant table in Xen. However, our work requires modest changes to the VMM architecture, and focus on attacking the performance issue at an algorithm level.

For VMM scheduling, Cherkasova et al. [16] discovered and enhanced the CPU scheduling residing in VMM to favor I/O domain without sacrificing fairness. Ongaro et al. [61] sorted the domains with the same states in the runqueue based on their remaining credits rather than arbitrarily insert the new domain at the end of each state section. However, with the same optimizations on our experiment environment under 10 GbE, we find that the blocking of scheduler tickle adversely glooms the I/O performance by a factor of 100 and the runqueue sort does not make any difference for I/O performance. We believe such a contrast is the result of the increased number of hardware interrupts, which cumulates a non-negligible overhead introduced by domain migration and therefore context switches. Our proposals differentiate from these angles by promoting the importance of cache topology. Liao et al. [45] proposed a credit-stealing mechanism for the VMM scheduler. However, the validity of their optimization is limited to the simplistic TCP streaming benchmark - Iperf. In our paper, the VM migration policy is proved efficient for more realistic workloads in SPECweb2005.

For optimizations in physical devices, Apparao et al.'s work [3] presented a workload characterization using vConsolidated benchmark and pinpointed the influence of cache in a virtualized multicore server. However, their proposal advocated a new asymmetric cache, whereas our work targets internal optimizations in Xen to use the cache more efficiently.

Chapter 3

Interrupt Affinity based Scheduling in the Kernel

Research in network I/O processing can be classified into three categories: 1) offload support on Network Interface Controller (NIC), such as TCP-IP Offloading Engine (TOE) [54], TCP Segmentation Offload (TSO) [70], TCP checksum offload, etc.; 2) changing network protocol stack by replacing TCP/IP with other O/S bypass protocols [21]; 3) running TCP/IP processing on an independent computation resource, which is tightly coupled with the application processor. Instead of using Network Processor (NP) to process network traffic [94] or offload the whole TCP/IP stack onto NIC, one of the cores on a multicore CPU can be bound to network processing, while other cores can run applications such as http requests and/or scientific computations. To distinguish from TOE, the last category is named "TCP Onloading" [70, 71]. Although the idea of TCP onloading sounds intuitive, most of such available designs require a large amount of change in the operating system, particularly in the TCP/IP protocol stack. Also, open problems like inter-core communication, mutual influences of processes for different applications still remain unsolved.

In this section, we implement a technique similar to TCP onloading technique for a heterogeneous workload on a multicore system with limited modifications to the kernel. We use 1) a kernel patch to change the frequency of scheduling between software interrupt processing and computation processes on the same core, 2) processor and interrupt affinity provided by Linux Kernel for SMP platform to affinitize the TCP/IP processing on the receive side to one separate core in the multicore system under a 10GbE environment, and 3) the Receive Side Scaling (RSS) [73] feature from the NIC, and show that our solutions are more cost-effective. Our experiments are based on a user-configurable modified version of *Iperf* [34], which considers the influence of a computational-intensive workload on I/O processing. We believe that our workload modification provides a more realistic scenario in the real server application. Our results illustrate on average a 10% and 12% improvement in throughput with corresponding reduction in cache misses, compared to the default round-robin fashion scheduling in a native system and VM, respectively.

A recent paper [57] contributed extensively to core scheduling in multicore processors with respect to I/O performance. However, we introduce different new approaches, namely, interrupt scheduling, onloading, and Receive Side Scaling (RSS) to boost the I/O performance. Also, while the previous paper measured the MPI latency at a high level, we base our experiments on a modified configurable *Iperf* workload, through which we are able to measure the impact of very low level scheduling decisions. Finally, our measurements are obtained using two dual-quad-core machines, which contains the state-of-the-art multicore processors today.

3.1 Traditional network I/O optimization

While the efficiency of core utilization remains an open problem in a multicore system, most of the research in network I/O has been focusing on new designs for hardware and support in Network Interface Controller (NIC), revolutionary fast connection network designs such as Myrinet [56], Infiniband [31], RDMA-iSCSI [72], etc. and TCP onloading. Although the performance gain of these techniques draws some attention in industry, each has its own counter-arguments.

TOE would work for high bandwidth, low latency applications, particularly IP storage network with RDMA support. However, the technique itself has been somewhat controversial because of the overhead in its software interface as well as security and extensibility concerns.

Fast connection aims at replacing traditional TCP/IP protocol stack with some other more efficient protocols and hence improves connection speed. However, two arguments point out its shortcomings. First of all, the price of such networking facility like Infiniband, iWarp [35] is higher compared to the ubiquitous Ethernet. With the already available 10GbE and availability of 40GbE in the near future, more interests are shown on protocols that support this traditional networking standard. In addition, research [22] shows that protocol processing itself is not the major source of I/O processing overhead. Instead, OS processing mechanism such as interrupt handlers, multi-copy of packets in-between Network Interface Controller (NIC) buffer and memory buffer as well as inside kernel buffers contribute more to the slow down.

TCP onloading uses part of the computation resource in a cluster server or special Processing Engines (PE) to provide TCP/IP processing service, while the host CPU is left to continue executing application processes. The interconnect between host and PE is the key to this problem. Solutions like polling and Direct Transport Interface (DTI) proved to be efficient although such designs require non-negligible modifications to the operating system. Intel’s Embedded Transport Acceleration (ETA) project [70, 71] partitions the system between the host and packet processing engine using DTI. DTI is used at the place where Linux kernel manages process and kernel context. Paper [57] designed an onloading model using Network Processor (NP) as the packet processing engine. Because TCP/IP stack processing switches at a certain frequency between interrupt and process context, it is not easy to find out the cutting point for the binding, all the previous solutions simply work around this problem by moving the whole TCP/IP stack to a separate processing engine, which therefore introduces redundancy in modification to the original OS. With more in-depth studies in the operating system, particularly the TCP/IP stack and the scheduling mechanism on multicore systems, TCP onloading remains a promising solution to the network-CPU speed mismatch problem even without changing the OS to a large extent.

3.2 Kernel core scheduling API

Since kernel 2.6, Linux provides affinity configuration for SMP platform, the user (with root access) could change the CPU affinity to interrupt and user space applications. Veal [88] shows the efficiency of CPU affinity with multiple NICs. Compared to previous works [70, 71, 94], we believe tuning up affinity set up for Linux is an easier and more reliable solution to onloading, especially when kernel modification becomes cumbersome in a virtualized environment.

3.2.1 Interrupt Affinity

Interrupt affinity is a Linux kernel feature that provides a user interface to bind hardware interrupts to certain CPU(s) based on the IRQ number of the interrupt. Because transmission side of I/O processing is relatively straightforward compared to the receive path, we only consider receive side. To initiate the receive process, NIC raises the interrupt when packets arrive at the kernel receive buffer. Later on, the device driver in the OS registers an entry for the interrupt in the IRQ translation table.

The file, `/proc/irq/<irq num>/smp_affinity`, contains the interrupt mask of interrupt number `<irq num>`. Each bit in the number represents a group of 4 CPUs, with the rightmost group being the least significant. "f" is the hexadecimal representation for the decimal number 15 (fifteen) and the binary pattern of "1111". Each of the places in the binary pattern corresponds to a CPU in the server, which means we can use the Table 3.1 to represent the CPU bit patterns:

It is noted that the kernel load balances all the hardware interrupts across the multicore system by default. As a result, the interrupt affinity could only take effect when the kernel feature irqbalance is disabled at boot time.

Table 3.1 Affinity Settings

Settings	Binary	Hex
CPU #0	00000001	1
CPU#1	00000010	2
CPU #0 and CPU #1	00000011	3
CPU #7	10000000	128

3.2.2 Processor Affinity

Processor affinity takes advantage of the fact that some remnants of a process may remain in one processor's state (in particular, in its cache) from the last time the process ran, so scheduling the process to run on the same processor for the next time could be more efficient than running on another processor.

Without explicit processor affinity set up, OS will load-balance user processes among cores to achieve fairness. However, the user space part of a network I/O process that reads packets is always bound (by default in OS) to the CPU that processes the interrupt.

Linux kernel 2.6, `set_cpus_allowed()` and `sched_setaffinity()` syscalls are provided to change the CPU affinity of processes. A user space utility, `taskset`, can be used to achieve the affinity between a CPU mask and the process PID.

3.3 Interrupt Affinity based Scheduling

In this section, we explain our affinity-based TCP/IP onloading technique in details. We start with a revisit on the classic Linux mechanism for I/O interrupts, particularly the receive side. Based on this mechanism, we show which processes are to be bound and how.

3.3.1 Linux interrupt handling mechanism

Once an interrupt is raised from a physical device, such as NIC, OS freezes user process and calls the corresponding interrupt handler to take over the CPU and service the device request in the interrupt context. Depending on the criticality, the job of the interrupt handler can be separated into two halves: top half, those time-critical jobs, such as acknowledging an interrupt to the PIC, reprogramming the PIC or the device controller. Bottom half, jobs that are deferrable, such as data copy from NIC buffer to kernel socket buffer. In Linux kernel 2.4, the bottom half is done in terms of system function `softirq()` by the OS. In this paper, we refer to the bottom half as interrupt context/processing unless stated otherwise.

Traditional Linux kernel services interrupt in interrupt context, in which the kernel is non-preemptable. No user process can be context-switched into CPU before the interrupt processing terminates. This mechanism is fine with limited interrupt requests on a simple host. However, under 10GbE network, where thousands of interrupts are generated in small intervals, the host CPU will be saturated by servicing interrupts, and therefore starving user space processes. On a consolidated server, where user space applications

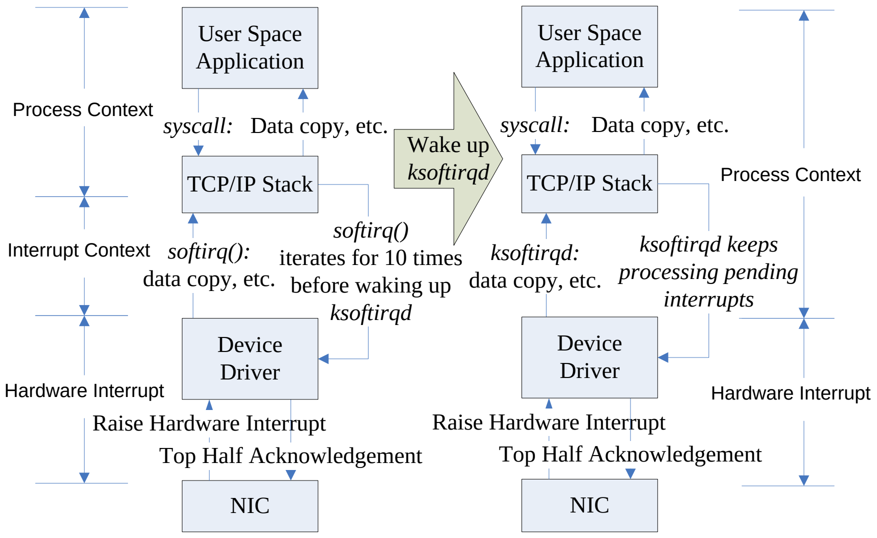


Fig. 3.1: Interrupt Handling Mechanism in Linux

generate tons of computational-intensive processes that require as much as, if not more, CPU power as opposed to those necessary for I/O processing, such a design would adversely influence the overall system performance to a large extent.

In order to solve this problem, `softirq()` was re-designed to handle the bottom half in a more flexible method. As shown in Fig. 3.1, before reaching to a pre-defined threshold (`MAX_SOFTIRQ_RESTART`=10), system function `softirq()` keeps handling all the pending requests. Notice that `softirq()` is interruptible, which means while it is running, more new interrupts might pop up. Without a threshold, `softirq()` falls back to the same mechanism as the traditional bottom half, where user mode processes could be starved. This threshold sets the maximum number of iterations `softirq()` runs before a kernel thread, `ksoftirqd`, is waken up to service the rest of pending interrupts.

The kernel thread `ksoftirqd` potentially optimizes the software interrupt processing. Since a thread can be scheduled, it is preemptible if another process with higher priority acquires the CPU that is currently running the first process instance, or the time slice of this process simply runs out. In fact, `ksoftirqd` has a low priority, so user programs have a very good chance to run; but if the machine is idle, the pending interrupts are executed quickly. This guarantees a critical tradeoff between interrupt processing and user process service.

3.3.2 Affinity configuration

In 10 GbE, traditional Linux kernel balances physical interrupts from PCI bus among all the available cores. We disable this feature by adding "`nonirqbalance`" to the bootloader so that interrupt and user application processing will be scheduled based on our affinity set up.

Previous section mentions the importance of the threshold-related kernel process `ksoftirqd`. By reducing the threshold, `ksoftirqd` is expected to be waken up at a higher frequency. In an extreme case, when `ksoftirqd` is exclusively used to service interrupt rather than using the system function in interrupt context, the whole interrupt handling becomes no more than a regular process which can be scheduled.

Most of previous research on affinity chooses to bind interrupts (with different IRQ#s) to different CPUs. More importantly, like [21], they claim that to affinitize interrupt processing and other userspace packet processing to the same CPU enhances the throughput by 20%.

Our research differs from those works in two areas. First, all of our experiments run in 10 GbE environment instead of 1GbE, with scalability issues involved. Secondly, we choose one NIC to handle all the interrupts instead of using multiple NICs. From our system utilization profiling (presented later), it is shown that device driver processing is not the bottleneck for I/O processing compared to the time spent in TCP/IP stack. Therefore, the number of hardware interrupts is no longer a significant benefactor to the performance overhead. In addition, even with just one NIC in 10GbE, we are guaranteed to have more hardware interrupts compared to those generated with multiple 1 GbE NICs. As a result, our work focuses on reducing overhead introduced by the upper half of operating system as opposed to the low level device driver and NICs.

There is also another reason why we tried to affinitize these interrupts with the same IRQ # to one CPU rather than splitting them across the multicore system. During our experiments, we found out the infeasibility of interrupt affinity while applied to the case in which 1-IRQ is mapped to multiple CPUs.

We bind the `ksoftirqd` process to one CPU for I/O processing, while keeping the other computational intensive workload on another core. By such configurations, the computation and I/O process run somewhat like a pipeline.

3.3.3 Interrupt Affinity Scheduling in VM

Similar to the set up in native system, as illustrated in Fig. 3.2, we boot 8 VCPUs for Dom0 and 2 VCPUs for DomU. We use `vcpu-pin` utility to pin all the VCPUs in Dom0 to PCPU#0 because Dom0 processes all the packets in the initial steps, while the rest of the processing is done in VCPU #0 that processed the interrupt (virtual) in DomU. Therefore ,VCPU#0 in DomU is mapped to PCPU#0 and VCPU#1 which runs the userspace application is affinitized to other PCPU #2 that shares the L2 cache with PCPU #0.

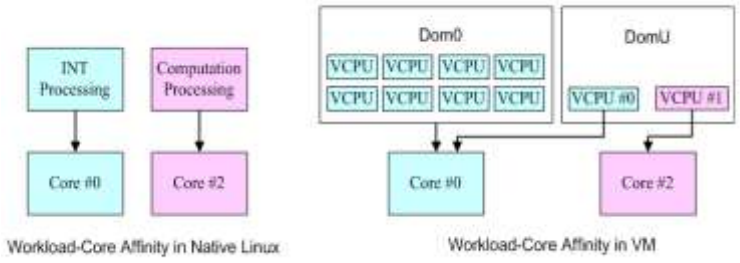


Fig. 3.2: Workload-Core Affinity

3.4 Experimental Results

Note that Fig. 3.3 solely illustrates how to separate out interrupt processing from userspace applications. Even though only 2 VCPUs are used in DomU, it is convenient to extend this set up for DomU with more VCPUs.

As shown in Fig. 3.3 and Fig. 4, we observe an average improvement of 10% and 12% for *Iperf* throughput in native system and VM, respectively. The biggest improvement is

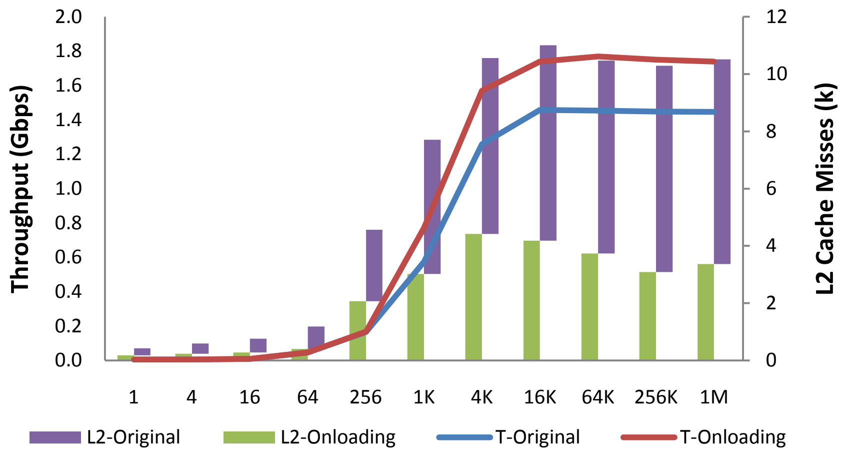


Fig. 3.3: Impact of Onloading on Throughput in for Iperf in Native Linux

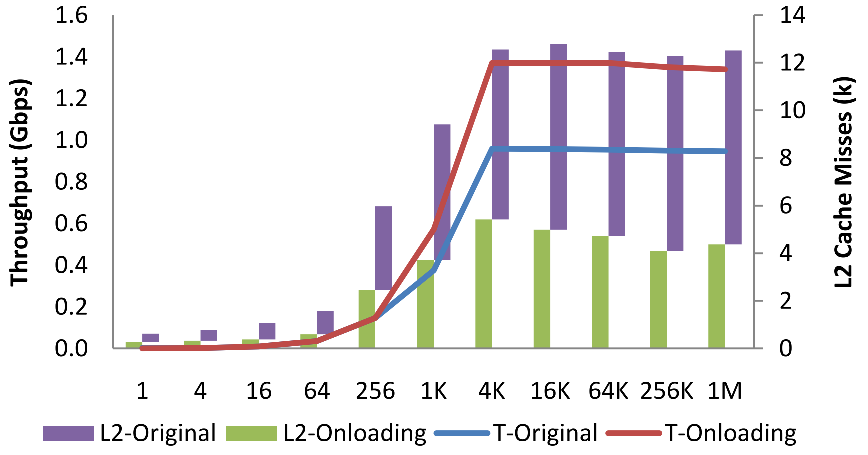


Fig. 3.4: Impact of Onloading on Throughput in for Iperf in VM

37% when the message size is greater than 4KB. Meanwhile, the L2 cache misses are also significantly reduced. The additional benefits of onloading in VM is due to the fact that the interrupt rate in VM is much higher than native system due to the virtual interrupts when guest domains communicates with each other and when guest domain request to access physical devices through the hypervisor. In this case, separating interrupt processing from userspace applications gives more time to the userspace application, and therefore achieves a higher throughput.

3.5 Summary

In this chapter, we proposed an interrupt affinity based scheduler to improve the system throughput on multicore servers by enabling interrupt scheduling. We used a kernel thread *ksoftirqd* to replace the syscall `softirq()` so that the bottom half of interrupt handler is changed from the interrupt context to the process context. We affinitized all the interrupt processing to one core, and leaving the rest of the userspace processing to the other core in the same cache domain.

Our proposed onloading method separates out interrupt processing irrespective to the intensity of the application workload. The limitation of this optimization is that it does not fully explore all the core resources (used only 2 out of 8). When more cores are involved for onloading, the cache affinity benefits will be offset by the core communication within the functionality group, whether it is for interrupt processing or userspace applications. In the following sections, we propose several schedulers in both native and VM systems that fully explores all the available core resources. They are particularly useful for heavy workloads such as L7-filter, multimedia transcoding and SPECweb2005.

Chapter 4

Affinity based Scheduling in Userspace

In Chapter 3, we proposed an interrupt affinity based scheduling mechanism in the kernel that works well for a microbenchmark. While the improvement in system throughput is impressive, we also realize that it only uses two cores on the multicore web server. This limit significantly affects the practicality of the scheduler because the computing cost of network applications such as L7-filter and transcoding is increasingly expensive. An effective scheduler should be able to use all the available cores on the web server to increase the system throughput.

In this chapter, we propose an affinity based scheduler in the userspace that fully utilize the computing resources on the multicore server. We discuss two important computationally intensive applications - L7-filter and transcoding. These applications are widely adopted in modern web servers. These legacy applications were originally designed as single threaded application, which cannot be running at full speed on multicore web servers. Among all the factors that contribute to this deficit, we focus on the parallelization of these applications. To focus on the study of the performance bottleneck, we propose an offline trace-driven model to decouple the program. This model cancels the processing noise coming from the network and provides a pure environment to study the scheduling problems of multithreaded programs on multicore servers. It is the foundation for the scheduler studies in the following sections.

4.1 Parallelization of 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 multicore server. As an extension, we also propose a processor mapping mechanism in Xen hypervisor and discuss the impact of virtualization on our new model.

4.1.1 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 [4, 10, 25, 26, 43, 67]. 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* [46] 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.

4.1.2 Modeling Single-Threaded L7-filter

Once a packet is processed by *libnids*, L7-filter classifies the packet in the steps described in Fig. 4.1. Packets are fed into the system at the optimal speed (as in a TCP connection with no packet dropped).

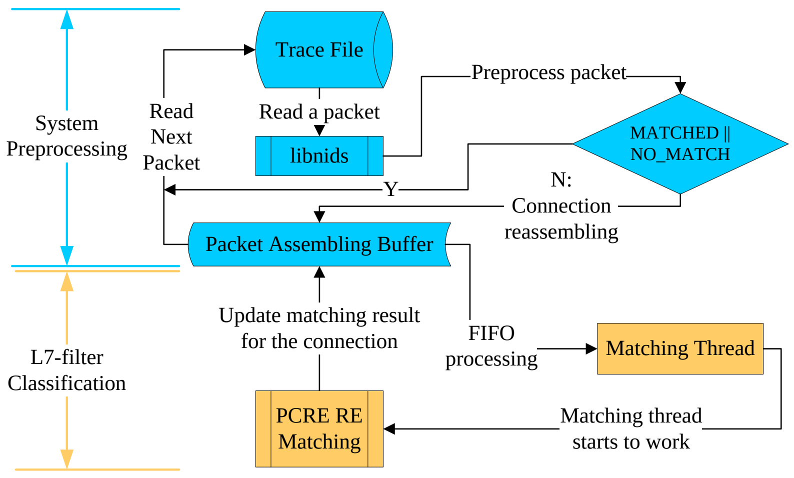


Fig. 4.1: Trace-driven L7-filter data flow.

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

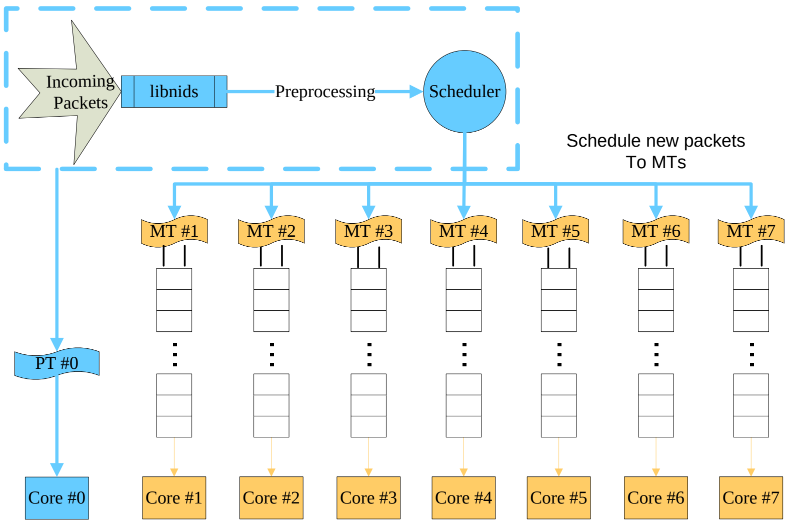


Fig. 4.2: Affinity-based Mulitthreaded L7-filter Architecture.

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 Fig. 4.1, one MT is handling the computation-heavy pattern matching operation. More MTs should be deployed to handle this operation, especially on multicore based systems.

4.1.3 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 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 [55, 60]. 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 [76, 77] 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 multicore server, as shown in Fig. 4.2. We affinitize the PT in core #0 and bind an MT for each of the cores left in the multicore 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.

Fig. 4.3 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 Fig. 4.2, 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

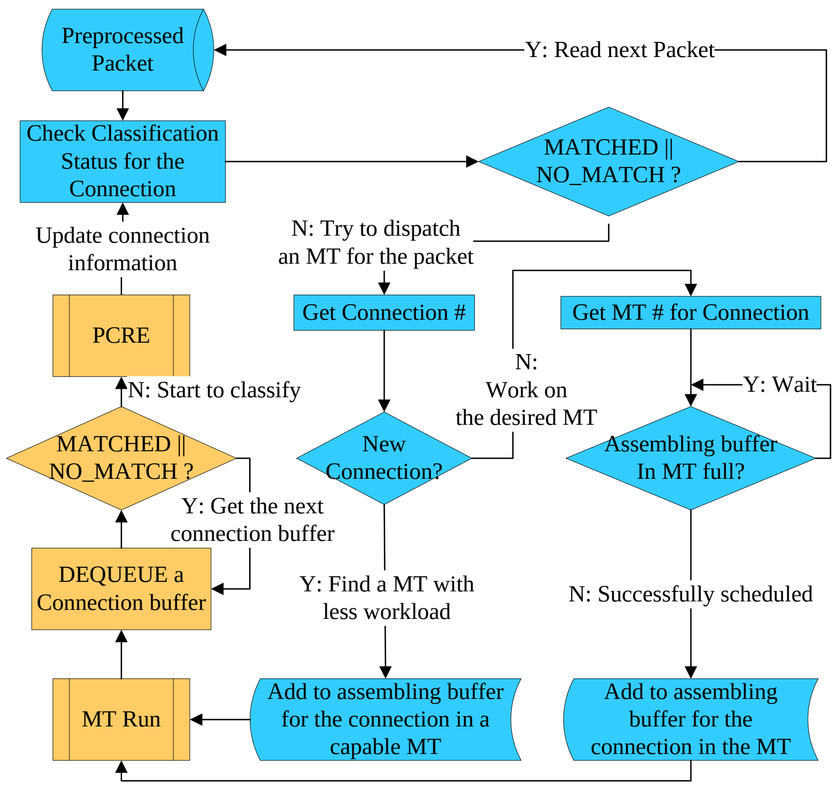

Preprocessed Packet
Y: Read next Packet
Check Classification Status for the Connection
MATCHED || NO_MATCH ?
N: Try to dispatch an MT for the packet
Update connection information
PCRE
Get Connection #
Get MT # for Connection
N: Work on the desired MT
Y: Wait
N: Start to classify
MATCHED || NO_MATCH ?
New Connection?
Assembling buffer In MT full?
Y: Get the next connection buffer
DEQUEUE a Connection buffer
Y: Find a MT with less workload
N: Successfully scheduled
MT Run
Add to assembling buffer for the connection in a capable MT
Add to assembling buffer for the connection in the MT

Fig. 4.3: Data flow in Scheduler.

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.

4.2 Parallelization of Transcoding

4.2.1 System Overview

We consider a scalable distributed web server architecture shown in Fig. 4.4, where the major functionalities required in the Internet servers (e.g., SSL, HTTP, script and cryptographic processing, database management, multimedia processing) are partitioned into parallel tasks. We focus on the media server with transcoding in this paper. The media web server adopts multicore architecture to accelerate the processing speed and accommodate the high bandwidth.

Fig. 4.5 illustrates the overview of the scheduling process in FFmpeg transcoding [20]. All the incoming streams are first stored in a global queue in the FIFO order. In the context of transcoding, each stream consists of many Groups of Pictures (GOPs), which are the minimal scheduling units and can be scheduled to any core independently. The relationship between stream and GOP can be equivalently understood as that of flow and

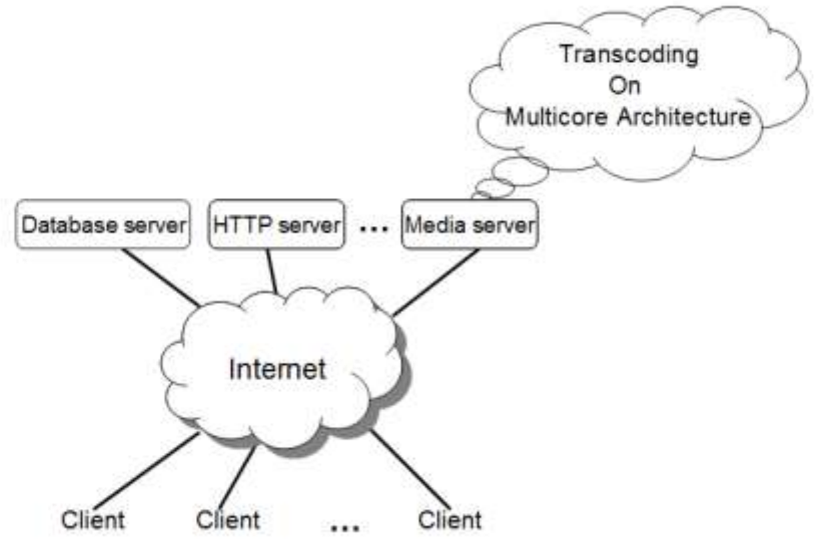


Fig. 4.4: Targeting web server architecture. Transcoding is deployed on media server.

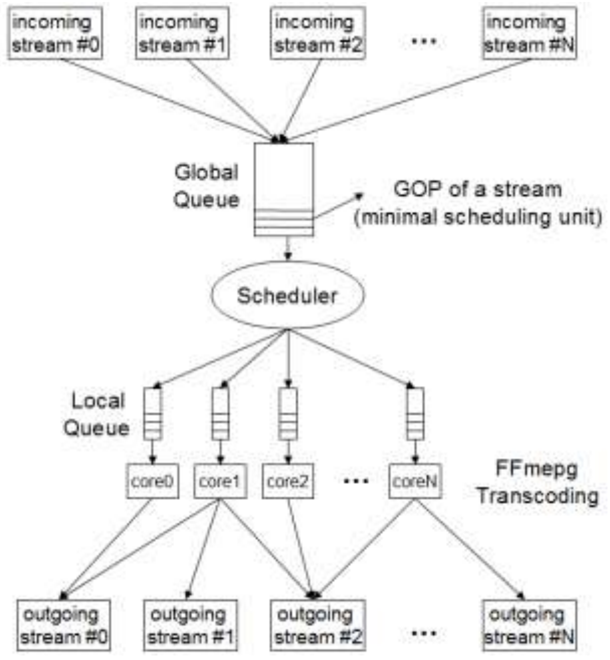


Fig. 4.5: Overview of the scheduling process in FFmpeg transcoding

packet. For each scheduling cycle, the scheduler fetches a GOP from the FIFO global queue, makes the scheduling decision based on the scheduler, and then dispatches the

GOP into the local queue of the scheduled core. In case of the affinity based scheduler described before for L7-filter, we just need to substitute "packet" with "GOP, and "connection" with "stream". Each core runs a transcoding thread, which iteratively fetches a GOP from its local queue and executes the task.

4.2.2 *FFmpeg* Transcoding Process

During the past decade, a number of video-coding standards have been developed for communicating video data, such as MPEG-1 for VCD, MPEG-2 for DVD and MPEG-4 for Blu-ray. For all these video-coding standards, four video resolution criteria are used in commercial products as listed in Table 4.1. Table 4.2 shows four movies used in our experiment with their original specifications. Depending on different client requirements, the transcoding operations can be categorized into frame size scaling, frame rate and bit rate alteration.

Fig. 4.6 illustrates the transcoding process in detail. *FFmpeg* is a powerful multimedia processing tool that can convert multimedia files between formats. The original movie is first decoded into raw frames by appropriate decoder (e.g., MPEG-1). Meanwhile, given the transcoding requirement for this movie in terms of video size,

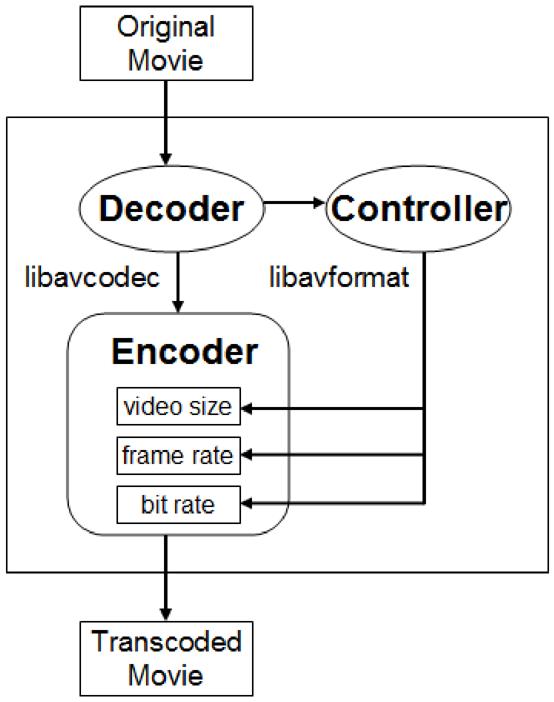


Fig. 4.6: FFmpeg transcoding process.

Table 4.1 Four resolution criteria in MPEG specification

Rate	Frame Size	Frame Rate (fps)	Bit Rate (kbps)
SQCIF	128 × 96	15	50
QCIF	176 × 144	15	70
CIF	352 × 288	26	100
4CIF	704 × 576	30	200

Table 4.2 Four MPEG-1 movies for transcoding

Rate	Frame Size	Frame Rate (fps)	Bit Rate (kbps)
Kung Fu Panda	352 × 240	29.97	920
Get Smart	352 × 240	29.97	920
Little Miss Sunshine	352 × 240	29.97	920
The Legend of 1900	352 × 240	29.97	920

frame rate and bit rate, the controller will specify those parameters used by encoder. Then, the encoder will start transcoding accordingly. The *libavcodec* is a library containing decoders and encoders for audio/video codecs and the *libavformat* is a library containing `demuxers` and `muxers` for multimedia container formats.

4.3 Direct Mapping in Virtualization

In this section, we extend our multithreaded L7-filter and *FFmpeg* to a virtualization environment. We expect our optimized programs to be integrated into the driver domain in Xen on a consolidated server. Incoming network traffic will be classified and demultiplexed to corresponding guest domains for further actions. The mapping between virtual and physical devices is transparent to userspace applications in the virtual machine. By default, Xen enumerates physical CPUs in a 'depth first' fashion. For a system with multiple cores, virtual CPUs would be allocated first to core #0. After core #0 is saturated, they are allocated to cores on the same processor socket, and then to cores in other sockets [95]. Such an allocation causes load imbalance that suffers from inefficient utilization of the multicore resources. Furthermore, it ignores application characteristics, which may introduce potential parallelism as in L7-filter and *FFmpeg*.

Hence, we propose a direct mapping technique for the multithreaded L7-filter and *FFmpeg* in virtualization. In direct mapping, we initialize the same number of virtual CPUs as the number of physical cores, and hot-plug each virtual CPU to the physical core using the `vcpu-pin` command. Such a mechanism eliminates the processor mapping overhead in the hypervisor and keeps the benefits of balanced utilization of the multicore server and warm cache as in the native environment.

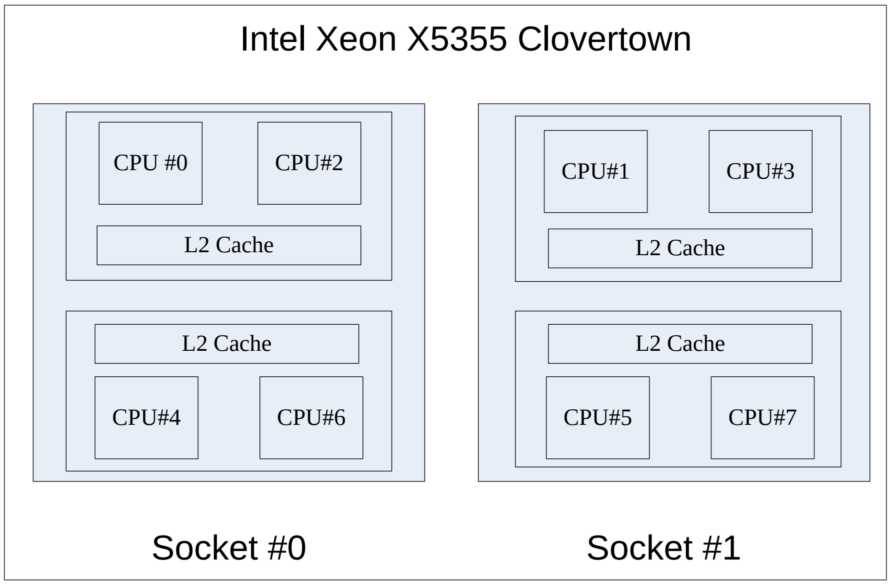


Fig. 4.7: CPU layout of Intel Xeon X5355 Clovertown

4.4 Experimental Results

4.4.1 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 Fig. 4.7. As we can see from the Fig., each socket has two 4MB shared L2 caches.

We use Linux kernel 2.6.18 as our default OS and Xen 3.1.3 with vanilla Linux 2.6.18 as the VM environment. 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 [66] 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 [52].

4.4.2 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 L7-filter. For *FFmpeg*, we define throughput as the number of GOPs processed per second. 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.

4.4.3 Throughput and Core Utilization

Fig. 4.8 and 4.9 illustrates the throughput and CPU utilization of L7-filter and FFmpeg in a native system, respectively. 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

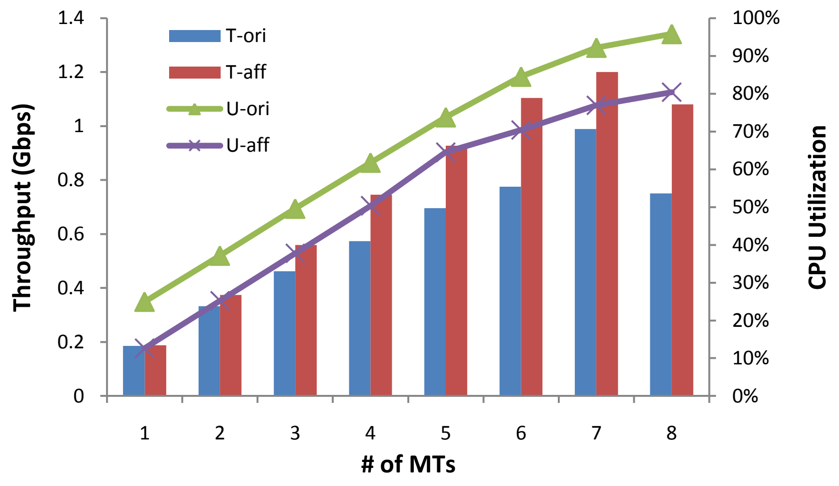


Fig. 4.8: Throughput and CPU Utilization for L7-filter in Native OS

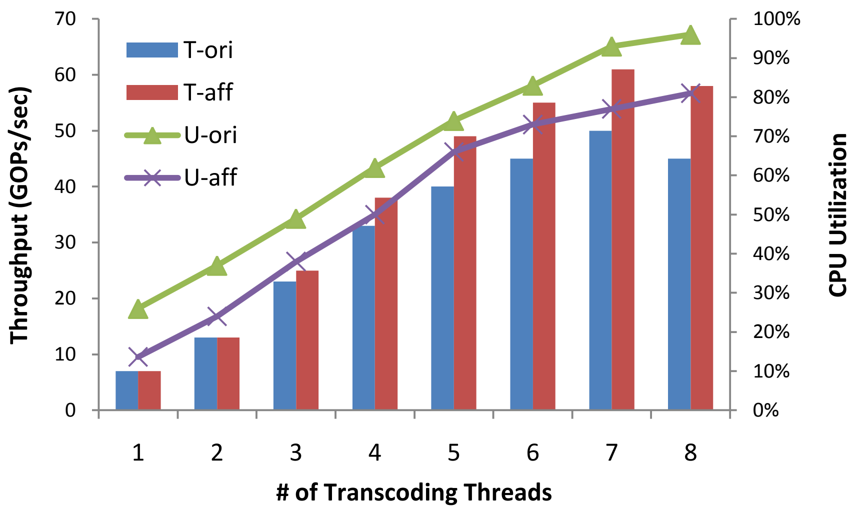


Fig. 4.9: Throughput and CPU Utilization of for FFmpeg in Native OS

as represented by the X-axis in the Fig. 4.8 and Fig. 4.9. 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. 4.8 and Fig. 4.9. 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.

We find similar throughput and CPU utilization improvement for the virtualization scenario in Fig. 4.10 and 4.11. For each group of bars, the first two (*T-ori* and *T-aff*) demonstrate the performance without and with the affinity based scheduler. The last bar (*T-aff w/pin*) shows the system throughput when "direct mapping" is applied. The legends for CPU utilization follow the same pattern. Our first observation is that the benefits of our affinity-based scheduling are somewhat offset by the default hypervisor behavior. Compared to the native case (51% increase in throughput), we can only achieve 12% increase in throughput. However, when direct mapping is applied, the system performance is improved by another 30%. In fact, the throughput in this case is only slightly lower than the native case for L7-filter (1.19Gbps V.S. 1.3Gbps). The VM utilization is higher than native (95.8% V.S. 89%). The difference is not significant for two reasons: 1) Unlike virtualized I/O where an additional data copy incurs due to packet transfer from the driver domain to guest domain, our offline model is ported to the driver domain without additional data copies. 2) Our direct mapping reduces the overhead in the

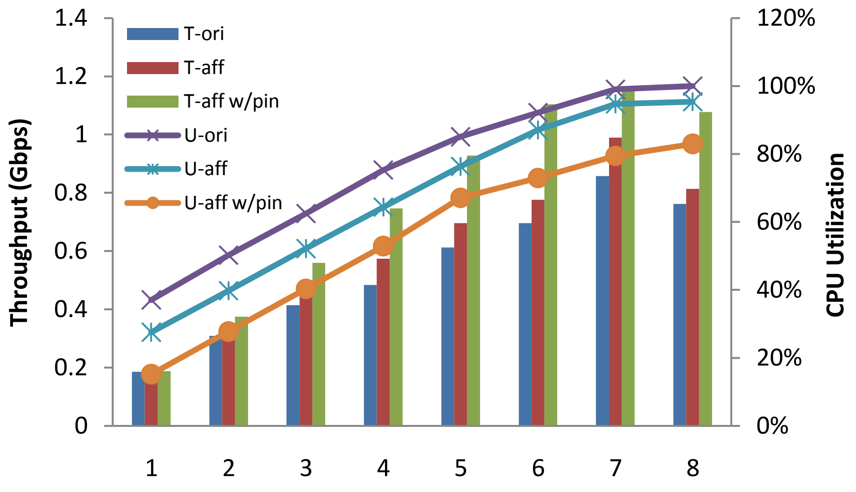


Fig. 4.10: Throughput and CPU Utilization for L7-filter in VM.

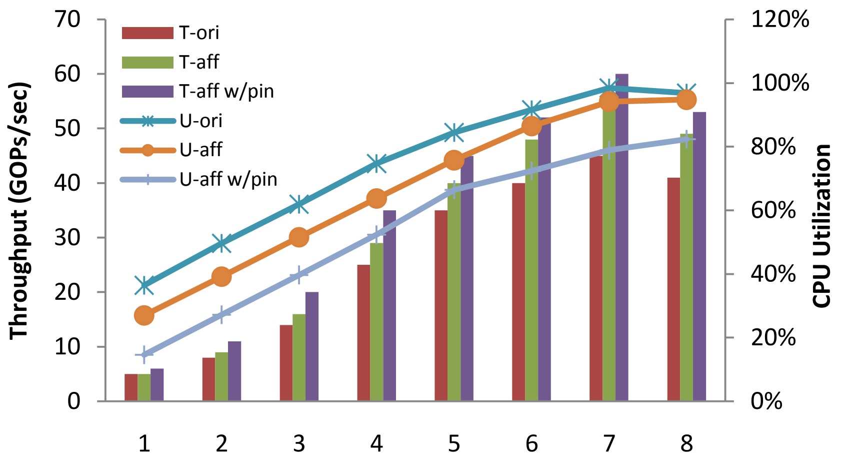


Fig. 4.11: Throughput and CPU Utilization for FFmpeg in VM

hypervisor by replacing virtual CPU scheduling with the native core scheduling. Notice that using L7-filter in driver domain has much broader utilization than its residence in guest domain. In server consolidation, each guest domain is likely to handle network traffic with the same functionality. Consequently, adding L7-filter in the driver domain helps improve the efficiency of packet demultiplexing and reduce the workload in guest OS. However, the implementation and design of guest domain infrastructure in virtualization is out of the scope of this section. In Chapter 7, we will introduce an additional optimization in the hypervisor to improve the system performance even more for a consolidated workload.

4.4.4 Cache Performance

As we discussed in section 3, the affinity-based multithreaded L7-filter is expected to provide performance improvement by ensuring efficient cache utilization. Fig. 4.12 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.

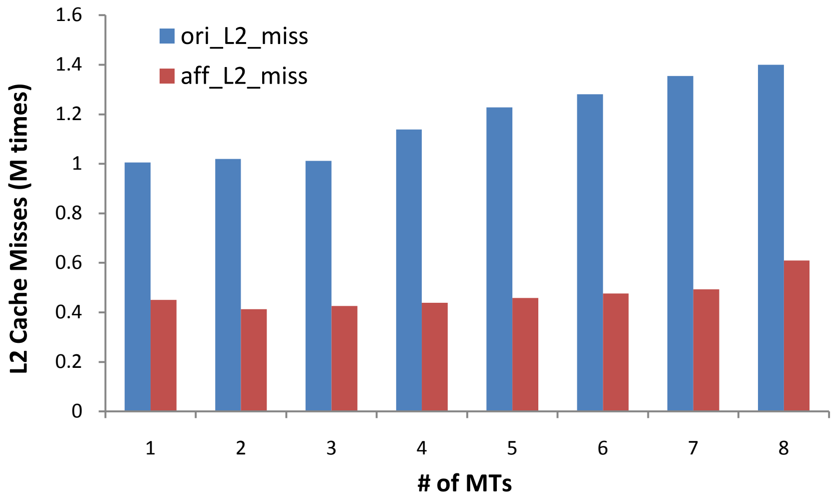


Fig. 4.12: L2 Cache Miss of the Offline Model using different # of MTs.

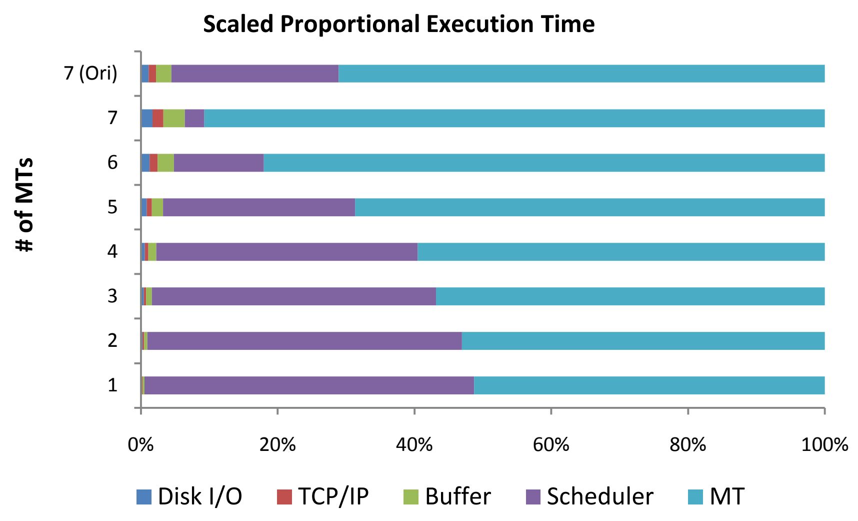


Fig. 4.13: A Life-of-Packet Analysis.

4.4.5 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 Fig. 4.13 and Fig. 4.14 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 Fig. 4.13 and only add the statistics for 7 MTs in the original non-affinity-based L7-filter as a comparison.

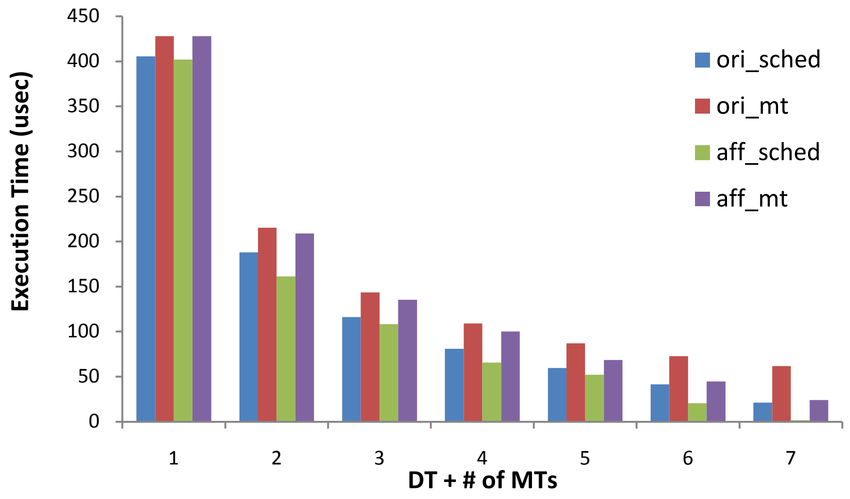


Fig. 4.14: Per-packet Component Execution Time Comparison Between the Scheduler and Matching Thread

Our first observation from Fig. 4.13 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 Fig. 4.13 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. 4.14 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

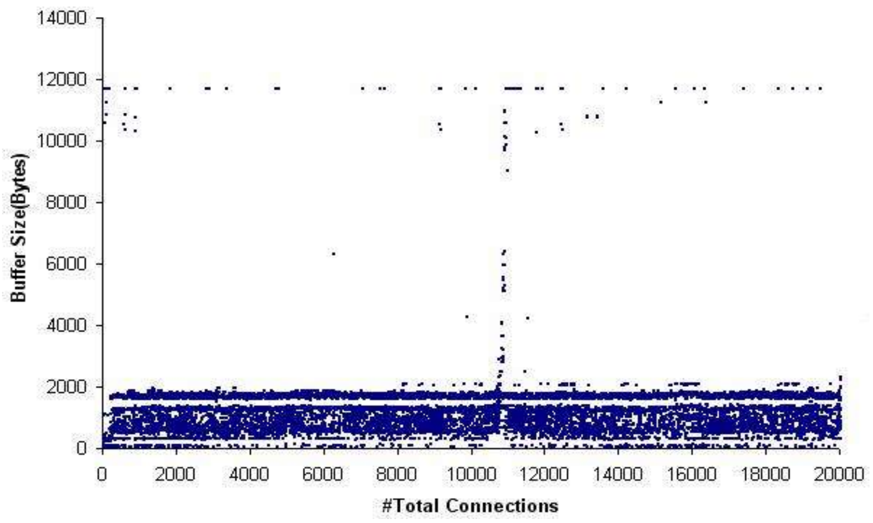


Fig. 4.15: Memory Requirement Analysis

scheduling stall. This observation is in line with the throughput results demonstrated in Fig. 4.8.

As Fig. 4.13 only presents results in percentage, we give the absolute execution time for the two largest contributors in the system in Fig. 4.13. A consistent observation from the Fig. 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 Fig. 4.13 to the system throughput, as demonstrated earlier in Fig. 4.8.

4.4.6 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 Fig. 4.15 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.

4.5 Summary

In this paper, we developed a multithreaded programming model for L7-filter and transcoding to exploit the connection level parallelism in packets. We proposed a thread scheduling technique on a multicore 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 multicore processor architectures. We also plan to make our multithreaded L7-filter software available to the open source community.

Chapter 5

Hash based Scheduling

In Chapter 4, we introduced a connection affinity based scheduler that improves system performance by reusing packet data in the same connection from the cache. However, the benefits of connection locality can be offset by the following challenges from the packet distribution in the network traffic to the core topology in different multicore architectures.

First of all, maintaining connection locality sacrifices the load balancing in workload scheduling at both the connection level and the packet level. Because connections usually outnumber cores, a naive scheduler might dispatch different number of connections to different cores. To achieve load balance at the connection level, hash based scheduler use the connection ID as an input [73, 74, 86]. The uniform distribution of keys in a hash function guarantees that each core shares a similar number of different connections. However, if the network traffic is unevenly distributed, packets in some connections might outnumber those in the others and therefore causes a jam on the core where the connection with more packets is 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.

Secondly, 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 8 SPARC cores. We show in paper [26] that with all the 64 threads enabled, the L7-filter system throughput can only be increased at most by a factor of 10.1X rather than the ideal 16X+. Note that we conservatively choose 16X to be the maximum speed up for "ideal" because the 64 threads only share 16 pipelines. Results in that paper also illustrate 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 multicore chip becomes a major concern in scheduler designs.

In this chapter, we propose an AHRW scheduler balance the workload at the packet level. In addition, we optimize the AHRW scheduler for a hierarchical multicore architecture. Specifically, we propose a hierarchical AHRW, H-AHRW, the recursively calls AHRW to balance the workload progressively. Instead of computing the HRW weight linearly core cores in one dimension, hash-tree scheduler computes the weighted queue vector and hash values hierarchically by traversing a tree. Not only does the hash-tree scheduler inherit the benefits from AHRW, such as load balancing at both the flow and the packet level, flow locality, minimal flow remapping and cache-awareness, it also 1) reduces the scheduling overhead from $O(N)$ to $O(lgN)$, assuming N is the total number of nodes and 2) reduces hash collision and achieve more effective load balancing through multiple hashing.

5.1 Baseline Highest Random Weight (HRW) Hash Function

Highest Random Weight (HRW) is a robust hash function and was originally proposed to map object requests to a cluster of servers [74, 86]. HRW has been popular in the areas of web servers, web caching and clustered digital libraries [26, 74, 86]. 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.

5.1.1 Formulation of HRW

In software scheduling scenario, we define HRW as follows:

Let $g(\vec{c}, j)$ be a pseudo-random weight function $g: C \times \{1,2,\dots,N\} \to (0, M-1]$, i.e. we assume $g(\vec{c}, j)$ to generate a random variable (weight) in $(0, M-1]$ with uniform distribution. The value of M 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. the set of connection IDs. The mapping $f(\vec{c})$ is then computed as follows:

$$f(\vec{c}) = j \Leftrightarrow g(\vec{c}, j) = \max_{k \in [1,N]} g(\vec{c}, k)$$

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 $g(\vec{c}, j)$, as proposed in paper [86].

The HRW hash function:

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

where $\mathrm{A} = 1103515245$ 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 i^{th} server in the cluster. This function generates a pseudo-random weight in the range $[0..2^{31} - 1]$. 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.

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 [86]:

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 [40, 41, 82], which guarantees that properties 1) and 2) hold true.

5.1.2 Minimum Disturbance Rate Property

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 is 1. However, if we partition the servers (hash space) into N sets, assuming that these sets are contiguous and of equal size:

The original N partitions:

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

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 N+1.

The new N+1 partitions:

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

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

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

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 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 [74].

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 case for example: for connection $c_1$andc_2, the incoming packets distribute over $c_1$and c_2 following probability $p_1$and p_2, respectively. Under the original HRW, only 2 PUs can be used to process c_1 and c_2, leaving the other 6 PUs idle. $p_1$and p_2 are usually not identical for L7-filter, causing further load imbalance between c_1 and c_2. It is necessary to break the connection locality to provide packet level load balance so that no PU is idling while runqueue of other PUs are non-empty.

5.2 Adjusted HRW (AHRW)

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 3, 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 try to put as many connection buffers with the same connection ID to the same PU as possible to reduce packet reordering and to increase the reuse of shared packet data in the cache. Now let us present the AHRW hash function.

Definition 5.1: Adjusted HRW Hash $\boldsymbol{h(\vec{c},p,r)} \rightarrow \boldsymbol{Weight}$: Let $g(\vec{c},p)$ be the original HRW hash function $g: C \times \{1,2,\dots,N\} \rightarrow [0..2^{31}-1]$, 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. g is a pseudo-random function that generates a random variable in $[0..2^{31}-1]$ 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 $r_p = \frac{\min_{p\in[1,N]} Queue\ Lent\,h_p}{Queuelent\ \ h_i}$.

The AHRW hash 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.

Definition 5.2: The scheduler $\boldsymbol{sched(\vec{c}) \rightarrow p}$:

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

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 before the runqueue length ratio r_p is applied by the scheduler. When r_p is applied, connection locality maintenance is only affected to the 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 is equivalent to the case when $DC = 0$. Because it is impossible to maintain perfect connection locality while applying a feedback system, we only need to justify that the relaxation is minimal for all the connections. Therefore, to prove the optimized connection locality property, we only need to show that DC is minimal. When $r_p = 1$, the AHRW falls back to the original HRW, whose property of minimal DC value is proved in paper [74]. In paper [40, 41], 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 r_p to gradually approximate the value 1 after the system becomes stabilized. Therefore, when $r_p \in (0,1)$, it falls into this case, i.e. 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 and fine-grained levels. For the coarse-grained connection level load balance, we simply need to consider the original HRW. Since the original HRW hash function g is a pseudo-random function that generates a random variable in $[0..2^{31}-1]$ 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 warm-up. 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$[0..r_p \cdot (2^{31}-1)]$. Thus, when r_p approximates to 1, we can see AHRW as a theoretical load balanced mapping model.

5.3 Hierarchical AHRW

In this section, we enhance the AHRW-based scheduler into a hash-tree scheduler, so that it can be more properly deployed on multicore servers with cache or thread localities. The theory behind this enhancement is that the 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 shorter compared to the number of leaf nodes as in a linear search.

5.3.1 Problem Statement and Motivation

Although the proposed AHRW-based scheduler provides load balance and connection locality, it requires a non-negligible amount of scheduling overhead, which delays the DPI processing and potentially causes undesirable packet drops. Meanwhile, the mainstream multicore servers usually possess extensive parallelization and sharing of resources that naturally form a hierarchical structure. For example, the highly threaded multicore chip Sun Niagara 2 in Fig. 5.2 (a), usually have multiple hardware threads organized hierarchically, forming a core-pipeline-thread architecture. 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 that accumulates the workload imbalance at each level of the parallelization.

5.3.2 Solution

Definition 5.3: The hash-tree scheduler $s_{tree}\,(\vec{c}) \rightarrow p_L$:

$$s_{tree}\,(\vec{c}) \rightarrow p_L$$
$$\Longleftrightarrow$$

$$\begin{cases} p_L = \arg \max\limits_{d_L \in \{leafs\ of\ node\ P_{L-1}\ at\ depth\ lgN\}} h(\vec{c}, d_L, r_{d_L}) \\ \cdots \\ p_1 = \arg \max\limits_{d_1 \in \{children\ of\ node\ p_0\ at\ depth\ 1\}} h(\vec{c}, d_1, r_{d_1}) \\ p_0 = \arg \max\limits_{d_0 \in \{PUs\ at\ depth\ 0\}} h(\vec{c}, d_0, r_{d_0}) \end{cases}$$

Given a hierarchical multicore architecture, we can apply the AHRW hash scheduler in Definition 5.2 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.

$$s_{tree}(\vec{c}) \to p_L \xLeftrightarrow{e.g.\ Niagara} \begin{cases} p_L = T_{P_c} = \arg \max\limits_{i \in \{threads\ on\ P_c\}} h(\vec{c}, i, r_i) \\ p_1 = P_c = \arg \max\limits_{j \in \{pipelines\ on\ c\}} h(\vec{c}, j, r_j) \\ p_0 = c = \arg \max\limits_{k \in \{cores\ in\ set\ C\}} h(\vec{c}, k, r_k) \end{cases}$$

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 3-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.

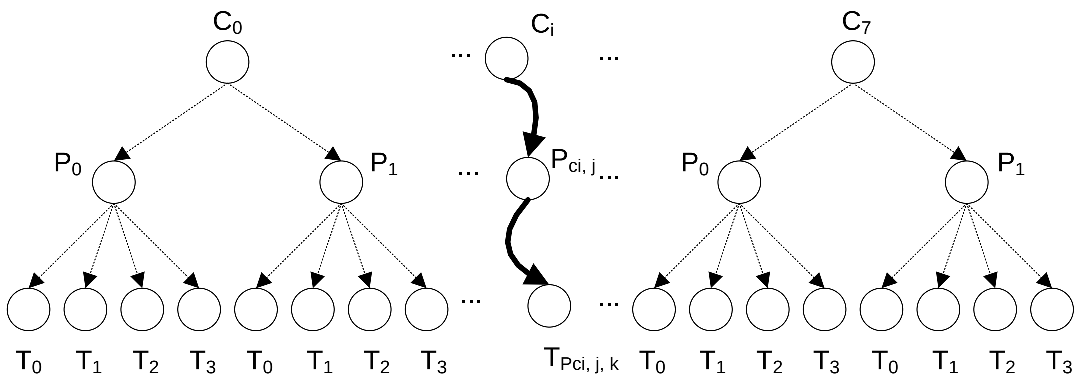


Fig. 5.1: The hierarchy of PUs on a Sun Niagara 2 chip. Each of the 8 cores (C) on chip contains 2 pipelines (P), with 4 threads (T) in each pipeline. The solid arrows represent a scheduler-selected path.

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 benefits:

Reduced computation cost. Suppose the complexity of HRW hash and the adjustment ratio computation is a constant number 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 $O(H \cdot lgN \cdot n)$. Note that the number of PUs N could be large, if the user defines more threads (virtual PUs) than the number of physical PUs.

More effective overall load balance. Note that while the hash space remains $[0..2^{31}-1]$, the hash-tree scheduler reduces the schedule space from N to lgN. 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 the experimental section, we will validate our model based on the performance measurement of our scheduler.

5.4 Experimental Set up

5.4.1 Experimental Platform

For the hash based scheduler, we chose the Sun SPARC and Niagara based web servers are one of the most popular choices for high performance network processing [49]. In the experiments, we used a Sun Niagara 2 based T5120 web server as our testbed. 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. The eight cores are connected to share a 4MB L2 cache through an 8X8 crossbar switch. Our testbed server 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 libraries in Solaris.

5.4.2 Sun Niagara 2 and the Solaris Scheduler

Fig. 5.2 (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. On the other hand, 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 [49]) 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.

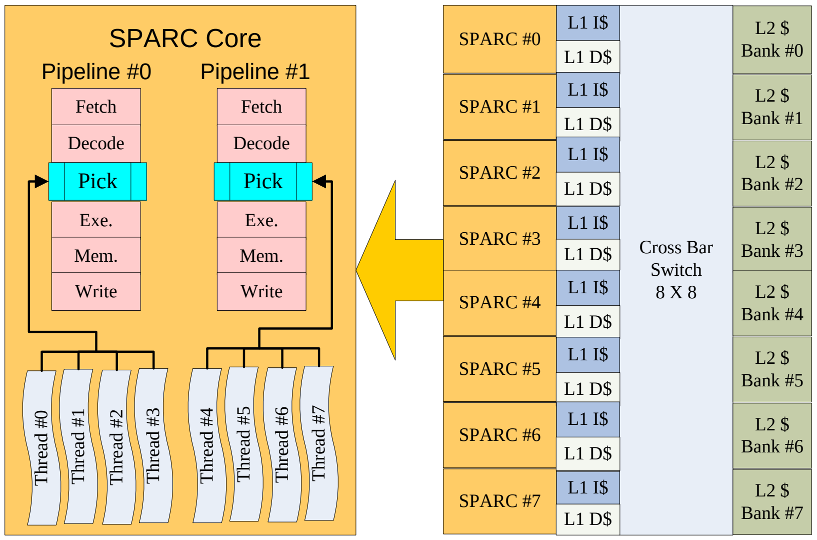


Fig. 5.2 (a): The Sun Niagara 2 Chip architecture and the parallelism inside each SPARC core

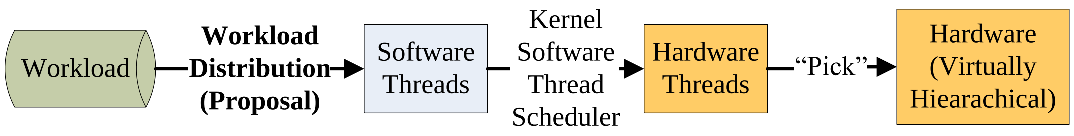

Workload
Workload Distribution (Proposal)
Software Threads
Kernel Software Thread Scheduler
Hardware Threads
"Pick"
Hardware (Virtually Hiearachical)

Fig. 5.2 (b): The scheduler topology in the Niagara 2-Solaris system

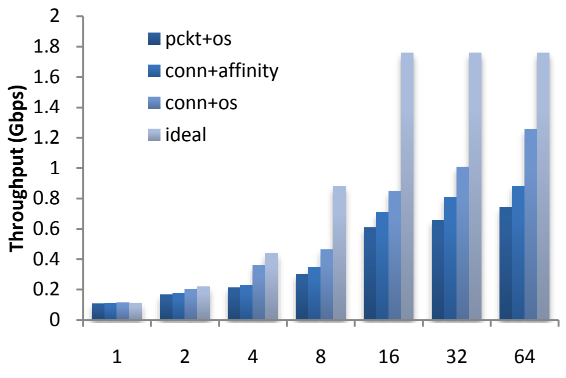
Fig. 5.2 (b) demonstrates the scheduler topology in the Niagara 2-Solaris system architecture. At every clock cycle, the hardware strand scheduler (the "Pick" unit) grants one of the four threads in a pipeline exclusive access to use the pipeline resource. Essentially, when one thread stalls for memory 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 [79]. 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.

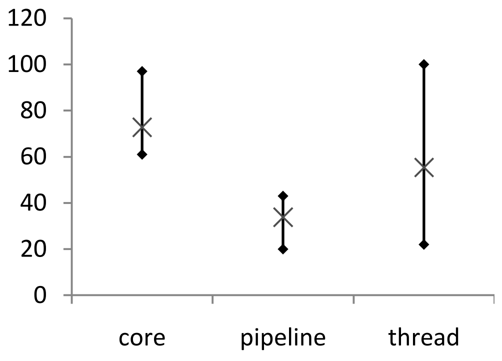
However, neither of these two schedulers distributes the incoming network traffic to the software thread. This kind of scheduling is defined in the application by the programmer. A Round-Robin distribution of the workload to the software threads is a common and simple default implementation. The scheduler 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.

5.4.3 Challenges in Sun Niagara 2

Previously, we have introduced a connection locality based scheduler for L7-filter on a general Intel Xeon web server [25]. However, we observed that the benefits of connection locality are offset by two major challenges on highly threaded hierarchical multicore servers.



(a) Throughput inefficiency



(b) Workload imbalance at different levels (%)

Fig. 5.3: 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 [ancs08]; "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 (%).

First of all, 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 8 SPARC cores. We show in Fig. 5.3 (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. 5.3 (b) illustrates the imbalanced system utilization at each level in the Niagara 2 system. This observation raises the concern of load balancing in addition to cache locality.

Maintaining connection locality sacrifices the fairness in workload scheduling when packet distribution is non-uniform. In an extreme case, if there may be more cores than connections giving rise to some idle servers. A load balanced system should be able to

Table 5.1Key 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

use all the cores by relaxing the connection locality The problem is how to balance the trade-off between the connection locality and load balance to maximize the throughput.

5.4.4 A Trace Driven Model

We adopt the same trace driven model proposed in paper [25]. 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 [46] 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 our experiment, we used three different packet trace files: 1) a 4-hour tcpdump file from MIT ("*MIT*") [52]; 2) a tcpdump file from Tsinghua University which records a section of 4 minutes and 19 seconds internet traffic ("*TU*"); and 3) a segmentation of tcpdump from NY Poly University ("*NYP*"). The key features of the traces are summarized in Table 5.1. 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.

5.4.5 Performance Metrics

In our experiments, we provide measurements from a real machine rather than simulators. We compare the AHRW hash-tree (H-AHRW) scheduler with 1) pure connection locality technique proposed in paper [25]; 2) Solaris OS scheduler; 3) pure HRW hash function which provides connection locality and load balance at the connection level; 4) our prototype AHRW scheduler which is more efficient compared to the idea proposed in paper [40, 41]. 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.

5.5 Experiment Results

In this section, we present the experiment results using different schedulers. Interested audiences may refer to paper [25, 26] for more details. The performance evaluation verifies the benefits of the proposed optimizations.

5.5.1 System Throughput and Scheduler Overhead

In Fig. 5.4, we show that the H-AHRW scheduler improves the system throughput by an average of 50.7% compared to using connection locality alone ("*conn*") [25], 19.8% compared to the single layer AHRW scheduler ("AHRW"), and 22.1% compared to the baseline HRW. We also find that for different traces, the system throughput increases as

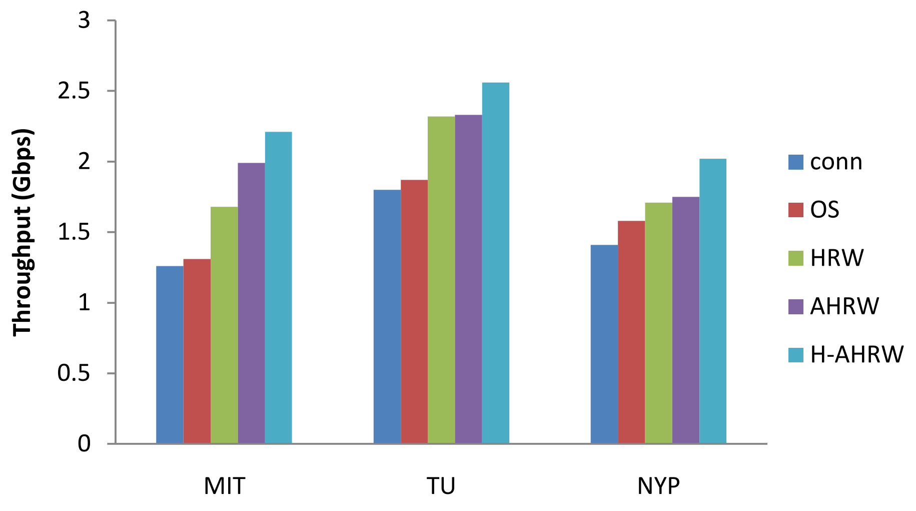
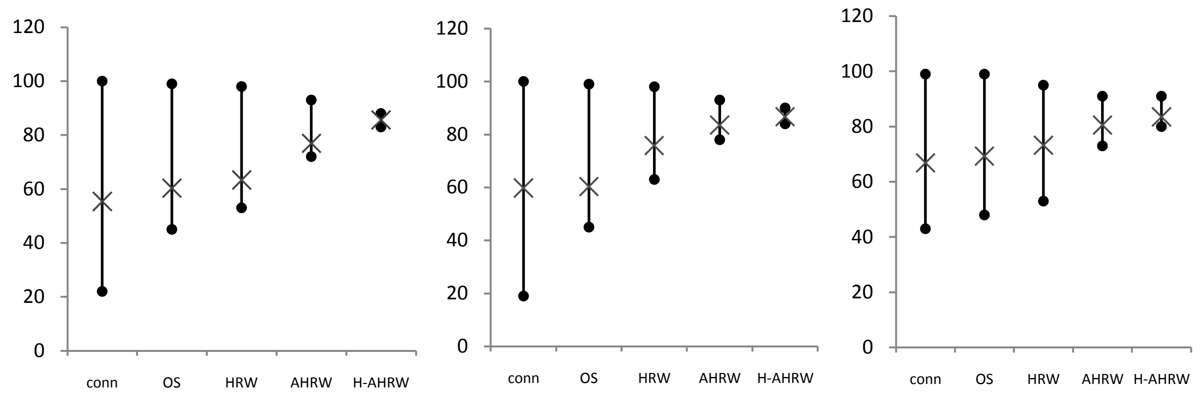


Fig. 5.4: System throughput comparison using different schedulers for all the traces

the average connection length increases. This is because L7-filter only processes the first eight packets in a connection. When connection length is large, more packets could be directly marked by L7-filter without going through pattern matching. Another observation is that the AHRW hash-tree scheduler is more efficient for a larger disparity between the connection distribution and packet distribution.

5.5.2 Load Balance

Fig. 5.6 verifies that AHRW hash-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 hierarchical AHRW, 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 significant. Specifically, the TU trace, in Fig. 5.6 (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



(a) MIT (b) TU (c) NYP

Fig. 5.6: System Utilization (%)

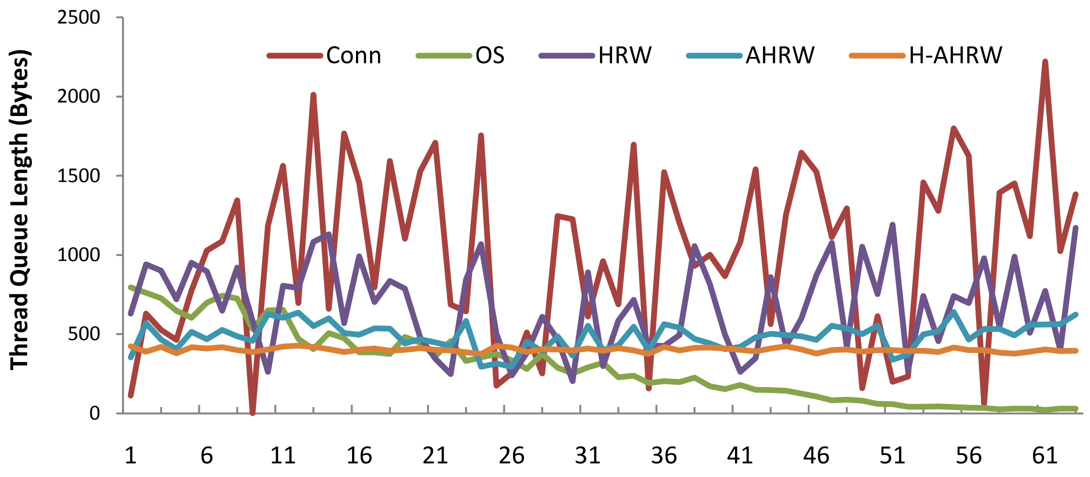


Fig. 5.5: Runqueue length on all the 63 matching threads. Note that thread #0 runs the preprocessing thread exclusively.

workload becomes more evenly distributed among the PUs, and 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. 5.4.

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. 5.5, it is quite straightforward that the runqueue length becomes much smoother when our scheduling optimizations that are applied. Another observation from the same Fig 5.5 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. 5.4.

5.5.3 Life of Packet Analysis (Scheduler 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. 5.7 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 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. 5.7 can be caused by either a reduced matching cost in the MT or an increased computation overhead in the scheduler.

Fig. 5.8 shows the absolute execution time for a packet. Clearly shown from this Fig. 5.8, each matching thread runs longer than the scheduler does. Therefore, the reduced MT execution percentage in Fig. 5.7 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

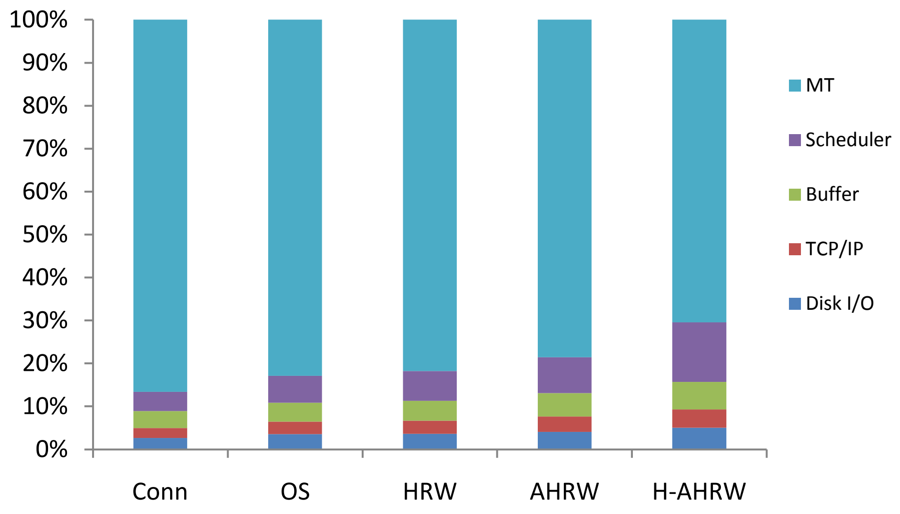


Fig. 5.7: Scaled execution time percentage for each component.

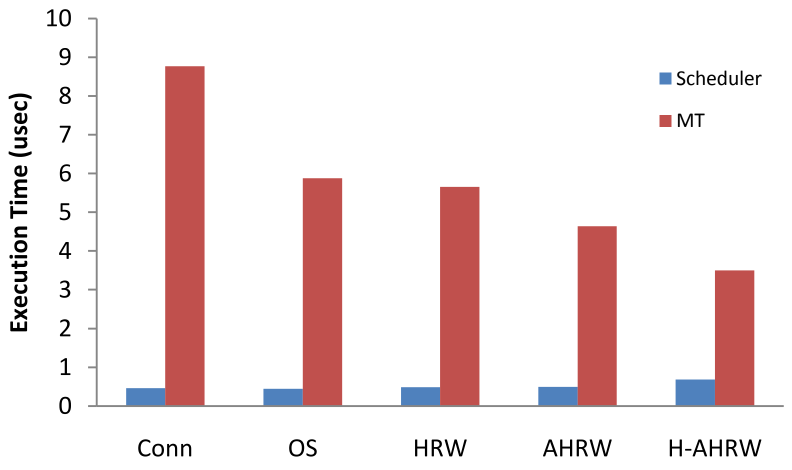


Fig. 5.8: Absolute execution time comparison between the scheduler and matching thread.

our theoretical analysis in Section 5.3. 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.

5.6 Summary

In this chapter, we propose a scheduler for L7-filter on a highly threaded hierarchical Sun Niagara 2 multicore server. In addition to maintaining the benefits from the connection locality of the network traffic like some previous proposed schedulers [25, 40, 41, 89], our scheduler also adaptively relaxes the locality constraint to achieve load balance at the packet level. 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 AHRW multilayer scheduler can improve the L7-fitler throughput by 59.2% compared to a previous work.

Chapter 6

Cache Aware AHRW (CA-AHRW)

In the previous chapter, AHRW strikes a balance between connection locality and load balancing at the packet level. Nonetheless, two essential problems still remain untouched: 1) cache-awareness in multicore architecture with heterogeneous inter-core communication cost and 2) a generalized HRW scheduler, for different core/cache topologies with both cache awareness and adaptive load balancing mechanisms.

In this chapter, we propose CA-AHRW, a Cache-Aware AHRW hash scheduler. This novel scheduler is designed for different core/cache topologies and is able to maintain both the flow and the packet level load balancing with cache-awareness while maintaining flow locality. Specifically, we first construct a communication matrix for a given multicore architecture characterizing the heterogeneous inter-core communicating overhead. Then, we obtain a weighted queue vector for each core based on that matrix and apply the weighted queue vector and an adjustment multiplier to the original HRW to adjust the hash value. Lastly, we propose a generalized CA-AHRW hash scheduler that can be applied to different core/cache topologies.

Similar to H-AHRW described in Chapter 5, we further design a hash-tree scheduler based on CA-AHRW, H-CAHRW, to enhance its performance for mainstream multicore web servers with a tree-based core/cache topology. The key difference between H-

CAAHRW and H-AHRW is that the former is generalized to accommodate different core topologies.

We implement and evaluate the H-CAHRW scheduler on two different tree based multicore servers, namely Intel Xeon E5335 [33] and AMD Opteron 2350 [1]. To do the transcoding, we parallelize *FFmpeg* software [20] and choose four recent movies with 20 concurrent streams as in [28]. The performance of our scheme is compared with five other schedulers, including Round Robin (RR), Stream-based Mapping (SM) [27, 28], original HRW (HRW) [74, 86], H-AHRW [Chapter 5]. Based on the results, we show that our scheduler improves the throughput and video quality for a variety of workloads with little scheduling overhead, better load balancing and fewer cache misses.

6.1 AHRW for Transcoding

Adaptive HRW Hash Function: We introduce an adjustment multiplier to the original HRW in our new hash function. Suppose the total number of processors is N, $\vec{c}$ is an identifier vector for the incoming flow and p is the Processing Unit (PU) ID. Let $g(\vec{c}, p)$ be the original HRW function, where$g: C \times [1, N] \longrightarrow (0, 2^{31} - 1]$, N is the number of PUs. We denote $r_p \in (0,1]$ as the ratio between the non-zero minimum queue length Q_{min} of all the PUs and the queue length of PU p, Q_p, at the point of scheduling. Then, the AHRW can be defined using Eq. 6.1:

$$h(\vec{c}, p, r_p) = r_p \cdot g(\vec{c}, p) = \frac{Q_{min}}{Q_p} \cdot g(\vec{c}, p) \tag{6.1}$$

In transcoding, the execution time of GOP can be modeled as linearly proportional to the GOP size [7]. Therefore, the queue length, which is measured by the total bytes of all

GOPs in the queue, is a good indicator of the workload on each processor. The adaptive HRW hash function dynamically adjusts the load balancing by reducing the weight on heavy loaded processor. Therefore, it decreases the chance to schedule the next packet to this processor. When the queue length of p is 0, the current processor p is idle. In this case, we force r_p to be 1, which results in the maximum possible weight for this particular processor. As a result, the chance to schedule the next packet to this processor is increased. The corresponding scheduler based on the adaptive HRW hash function is given in Eq. 6.2.

$$f(\vec{c}) = p \Leftrightarrow h\left(\vec{c}, p, r_p\right) = \max_{k \in [1,N]} h(\vec{c}, k, r_k) \qquad (6.2)$$

6.2 Weighted Queue Length Derivation

In the AHRW hash function, the adjustment multiplier r_p will strike a good balance between the connection and packet level load balancing. However, this multiplier is unaware of the underlying core/cache topology because it simply takes into account the actual queue length on each PU. In fact, since the communication overhead between cores in a multicore architecture is heterogeneous, especially with hierarchical memory structure, we should distinguish the inter-core relationship by applying weighted queue lengths in the AHRW hash function.

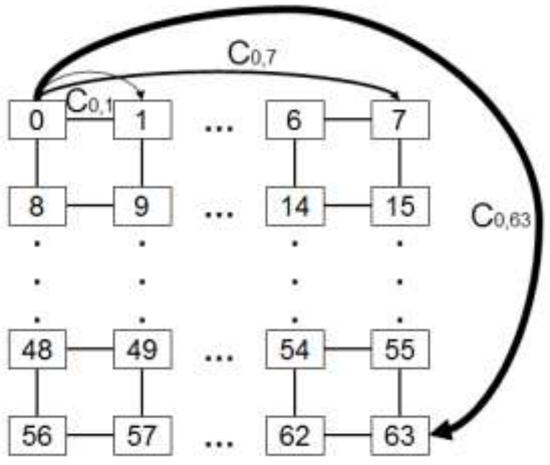


Fig. 6.2: A mesh-based multicore architecture: Tilera TILE64 Mesh Processor. $C_{0,1} < C_{0,7} < C_{0,63}$

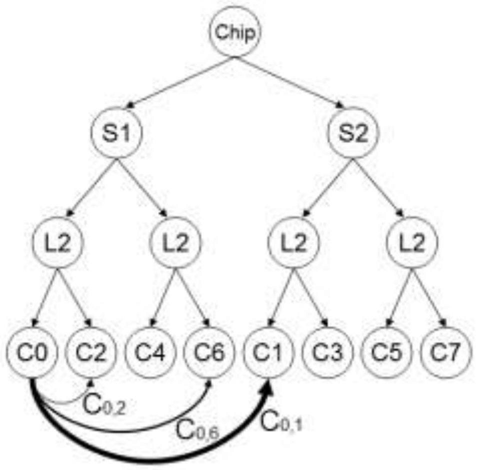


Fig. 6.1: A tree-based multicore architecture: Intel Xeon E5335. $C_{0,2} < C_{0,6} < C_{0,1}$

Fig. 6.1 and Fig. 6.2 show two typical types of multicore architectures. The Tilera's TILE64 mesh processor [87] is based on the mesh topology, where each core has its own L1 and L2 cache. The Intel Xeon E5335 has a tree hierarchy. From bottom up, a group of two cores share the same L2 cache. Two of these groups (4 cores) share the same core socket (S1 and S2). Two of these sockets (8 cores) share the same processor chip. Obviously, the communication cost between cores is asymmetric as illustrated by the arrow thicknesses, which is proportional to the communication delay.

We propose a matrix representation of inter-core relationship to characterize this communication heterogeneity. We consider a simple example as shown in Fig. 6.3, where 4 cores are connected in a mesh topology. We define the communication overhead ratio between core i and core j as $c_{i,j}$, which is defined by Eq. 6.3, assuming

$Comp_j$, Mem_j, $Comm_{j,i}$ represent the computation time on node j, memory access time on node j and the communication time from node j to node i, respectively.

$$c_{i,j} = \frac{Comp_j + Mem_j + Comm_{j,i}}{Comp_i + Mem_i} \tag{6.3}$$

We define the "homenode" for a flow as the scheduled node by the original HRW hash scheduler, which is responsible for preserving packet departure order and is determined only by the identifier vector $\vec{c}$. Since every packet has to communicate with its homenode during transcoding, $c_{i,j}$ implies the communication overhead when a packet with homenode i is scheduled on core j for packet level load balancing. $c_{i,j}$ is always larger than 1, because the sum of $Comp_j$ and Mem_j should be no less than that of $Comp_i$ and Mem_i in Eq. 6.3. Intuitively, the larger the $c_{i,j}$, the more overhead this scheduling decision will incur. Thus, our goal is to schedule a connection to those cores that are as close to its homenode as possible.

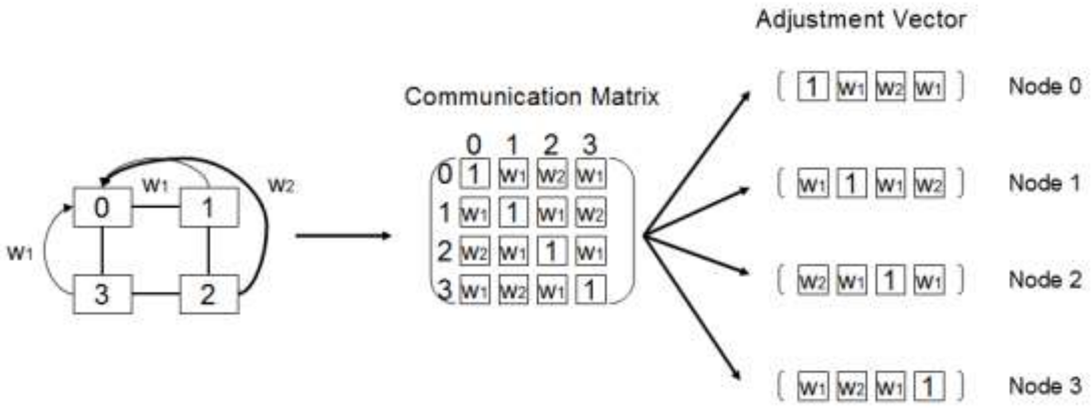
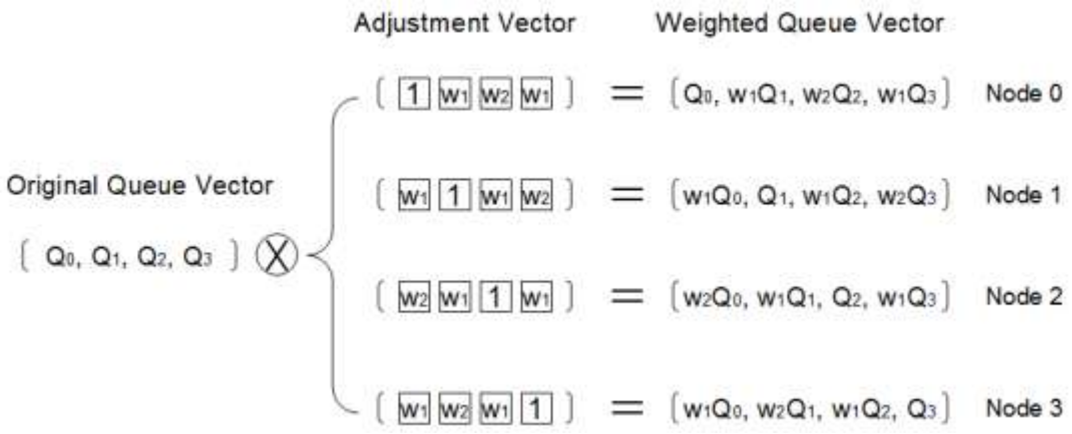


Fig. 6.3: An example of communication matrix and adjustment vector from a multicore architecture.

In Fig. 6.3, we have two different communication overhead ratio w_1 and w_2. Given this parameter, we can construct a communication matrix to represent the inter-core relationship. Suppose the total number of nodes (PUs) is M, then the $M \times M$ matrix is formed by filling the position (i, j) with $c_{i,j}$. Obviously, this is a symmetric matrix with the diagonal elements being all 1. We further define an adjustment vector for each node as follows: for node i, its adjustment vector is the i^{th} row vector in the communication matrix. Fig. 6 shows the communication matrix and the derived adjustment vector for each node.

Now we address how to derive the weighted queue length for each node. Suppose we have an original queue vector (Q_1, Q_2, Q_3, Q_4), which records the real queue length. For each node, by multiplying this queue vector with the adjustment vector from this node, we can obtain the weighted queue vector. Here, we define a new vector multiplication operation following Eq. 6.4.



Adjustment Vector
Weighted Queue Vector
Original Queue Vector
[Q0, Q1, Q2, Q3]
[1 w1 w2 w1] = [Q0, w1Q1, w2Q2, w1Q3] Node 0
[w1 1 w1 w2] = [w1Q0, Q1, w1Q2, w2Q3] Node 1
[w2 w1 1 w1] = [w2Q0, w1Q1, Q2, w1Q3] Node 2
[w1 w2 w1 1] = [w1Q0, w2Q1, w1Q2, Q3] Node 3

Fig. 6.4: Weighted queue vector derivation from the original queue vector and the adjustment vector.

$$\vec{A} \otimes \vec{B} = \vec{C}, where\ c_i = a_i \cdot b_i \qquad (6.4)$$

Suppose $\vec{A}$ is the original queue vector, $\vec{B}$ is the adjustment vector and $\vec{C}$ is the weighted queue vector. For instance, as shown in Fig. 6.4, node 1 obtains the weighted queue vector $(w_1Q_0, Q_1, w_1Q_2, w_2Q_3)$, which reflects the core/cache topology and communication heterogeneity in the multicore architecture.

6.3 Generalized CA-AHRW Hash Scheduler

The justification of the above definition for queue length is easy to understand. On one hand, if all the children nodes are busy, then the queue length for the parent node is simply the sum of all children queue length. On the other hand, if one child queue is empty, we set the parent queue length to 0, which increases the chance to schedule the next packet to the parent node and then finally to the node with empty queue. This scheme facilitates load balancing in a cache-aware way by considering possible resource sharing. The justification of the above definition for queue length is easy to understand. On one hand, if all the children nodes are busy, then the queue length for the parent node is simply the sum of all children queue length. On the other hand, if one child queue is empty, we set the parent queue length to 0, which increases the chance to schedule the next packet to the parent node and then finally to the node with empty queue.

Based on the weighted queue length, we present the CA-AHRW hash function in Eq. 6.5 and the generalized CA-AHRW hash scheduler in Eq. 6.6. Notice that the weighted adjustment multiplier w_p substitutes the original r_p in Eq. 6.1 and Eq. 6.2. $Q_p^{'}$ is the weighted queue length on node p chosen from the weighted queue vector for the homenode of $\vec{c}$ in Eq. 6.4.

$$h'(\vec{c}, p, w_p) = w_p \cdot g(\vec{c}, p) = \frac{Q_{min}}{Q'_p} \cdot g(\vec{c}, p) \tag{6.5}$$

$$f'(\vec{c}) = p \Leftrightarrow h(\vec{c}, p, w_p) = \max_{k \in [1,N]} h'(\vec{c}, k, w_k) \tag{6.6}$$

Our generalized CA-AHRW hash scheduler consists of the following four steps:

- Step 1: Construct the communication matrix for the targeting multicore architecture and derive the adjustment vector for each node.
- Step 2: For a given packet of a flow whose homenode is i, obtain the current weighted queue vector for node i.
- Step 3: Apply the weighted queue length in CA-AHRW hash function following Eq. 6.5.
- Step 4: Choose the node with the maximum weight resulted from the hash function $h'(\vec{c}, k, w_k)$ as the scheduling node according to Eq. 6.6.

The four properties in the CA-AHRW hash scheduler are listed below:

- **Flow Locality**: Initially, all the packets in the same flow are scheduled to the same processor before the adjustment multiplier w_p takes effect based on the original HRW function. When w_p is applied in the scheduler, flow locality is only affected to the minimum extent. Because although the generated weight for each processor changes, the selection of the candidate node with maximum weight is affected only when system load imbalance is substantial.

- **Load Balancing**: On the one hand, for the connection level load balancing, since the original HRW hash function is a pseudo-random function that generates a random variable in the rage of $[0,2^{31}-1]$ with uniform distribution, the load balancing at the flow level is intuitively proved when the flow is evenly distributed. On the other hand, w_p dynamically adjusts the load imbalance problem when the flow is not evenly distributed. As the system approaches the stable state, w_p approximates to 1, which indicates that $h^{'}(\vec{c},k,w_k)=w_p\cdot g(\vec{c},p)$ also generates a random variable with uniform distribution in the range of $[0,w_p\cdot(2^{31}-1)]$. Thus, CA-AHRW also achieves packet level load balancing after the system is stabilized.
- **Cache Awareness**: CA-AHRW uses the weighted queue length instead of the real queue length to reflect the communication heterogeneity in multicore architectures. The weighted queue vector distinguishes the distance between cores, which is able to characterize the underlying multicore architecture. Our scheme favors to schedule the packet as close as possible to its homenode, which can improve the system throughput by reducing cache misses. In most general cases, CA-AHRW's weighted queue vector reduces the unnecessary long-distance communication between cores that are far apart. The idea of localization in scheduling reduces the scheduling overhead and accelerates the processing speed.
- Architecture Independence: CA-AHRW is a generalized hash scheduler with both cache- awareness and adaptation, which can be used in different core/cache topologies. In fact, the topology difference only affects the communication matrix

and is independent of the later scheduling process. As long as the matrix is constructed, CA-AHRW can work effectively for any architecture it applies to. In addition, the feedback mechanism can also be carried out on any platform, because we only care about the local queue length on each processor. Therefore, CA-AHRW hash scheduler is architecture independent with its full features.

6.4 Hierarchical CA-AHRW (H-CAHRW)

Although CA-AHRW is a generalized HRW hash scheduler applicable for different core/cache topologies, it requires a non-negligible amount of computing overhead due to the hash computation for each node, which may delay the transcoding processing and potentially cause undesirable throughput. Meanwhile, as mainstream multicore web servers usually adopt a tree-based topology, such as Intel Xeon E5335 and AMD Opteron 2350, we further enhance our CA-AHRW into a hash-tree scheduler as in [14, 26].

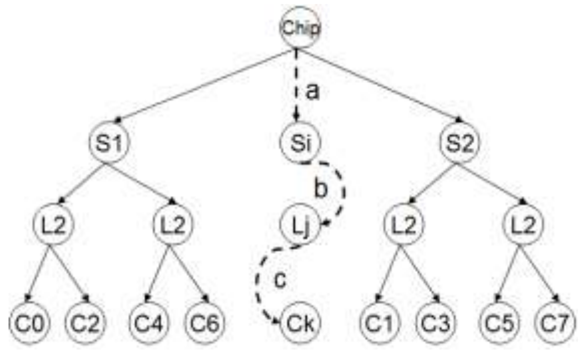


Fig. 6.5 (a): Hash-tree scheduling process on Intel Xeon E5335 server. The dashed arrows represent a scheduler-selected path: a→b→c.

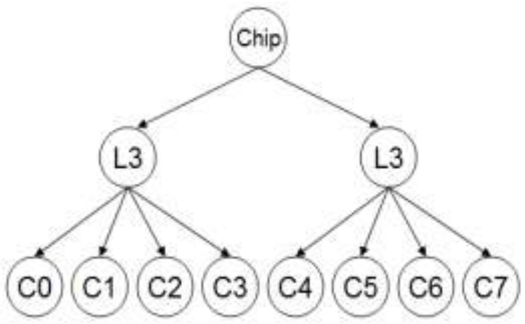


Fig. 6.5 (b): The hierarchy of PUs on a AMD Opteron 2350 server. Four cores share a L3 cache and the server consists of eight cores

6.4.1 Hierarchical Queue Length for Tree-based Topology

Similar to H-AHRW, the key idea for H-CAHRW is to take advantage of the inherent tree hierarchy. Traversing down from the root, we make scheduling decision at each level until reaching a leaf node. At each level, we apply our CA-AHRW hash scheduler. However, the introduction of hierarchy complicates the weighted queue length derivation as defined in Eq. 6.4. In H-AHRW, we naively sum up queue length of children nodes as the queue length of the parent node, which is not aware of the core/cache topology. To integrate the CA-AHRW into H-CAHRW, we propose a generic algorithm for hierarchical multicore architectures to derive the weighted queue length at each level.

We define each tree node as Processing Unit (PU) in Fig 6.2. Fig.6.5 (a) has demonstrated the hierarchy of PUs on Intel Xeon E5335. To illustrate another tree structure that will be used in our experiment, we present Fig. 6.5 (b) to show the hierarchy of PUs on AMD Opteron 2350.

Now we present the queue length derivation for each tree node. First, for all the leaf nodes, their queue lengths are decided by the weighted queue vector itself as in Fig. 6.4. Second, for any non-leaf node i, suppose it has m children nodes with queue length $n_j, j \in [0 - m - 1]$. Then, the queue length Q_i is derived in Eq. 6.7.

$$Q_i = \begin{cases} \sum_{j \in [0,m-1]} n_j \text{ , } if\ \forall n_j, n_j \neq 0 \\ 0 \text{ , } \quad if\ \forall n_j, n_j = 0 \end{cases} \tag{6.7}$$

The justification of the above definition for queue length is easy to understand. On one hand, if all the children nodes are busy, then the queue length for the parent node is simply the sum of all children queue length. On the other hand, if one child queue is empty, we set the parent queue length to 0, which increases the chance to schedule the next packet to the parent node and then finally to the node with empty queue. This scheme derives weighted queue length at each level and facilitates load balancing across all the cores.

6.4. 2 H-CAHRW scheduler and Properties

Given a hierarchical multicore architecture, we can apply our CA-AHRW hash scheduler repeatedly along the traversal of the tree, except that we replace the weighted queue length with the queue length obtained according to Eq. 6.7 for all non-leaf nodes. For nodes at the same depth of the tree, we can pick an internal node by the CA-AHRW hash scheduler and continue the traversal from that chosen node. The ultimate goal of the hash-tree scheduler is to select a candidate node among the leaf nodes by traversing through the three as shown in Eq. 6.8, assuming the tree height is L with the root being level 0 and the leaf being level L.

$$f_{tree}(\vec{c}) = p_n$$

$$\Leftrightarrow$$

$$\begin{cases} h'(\vec{c}, p_1, w_1) = \max h'(\vec{c}, k_1, w_{k_1}), k_1 \in children\ of\ root\ in\ level\ 1 \\ h'(\vec{c}, p_2, w_2) = \max h'(\vec{c}, k_2, w_{k_2}), k_2 \in children\ of\ P_1\ in\ level\ 2 \\ \dots \\ h'(\vec{c}, p_L, w_L) = \max h'(\vec{c}, k_L, w_{k_L}), \quad k_L \in children\ of\ P_{n-1}\ in\ level\ L \end{cases} \tag{6.8}$$

Fig. 9 shows the deployment of H-CAHRW on Intel Xeon E5335 web server. For any given packet, the hash- tree scheduler first picks a socket S_i with the maximum weight generated by the CA-AHRW hash function at the socket level (path a). Then we apply the CA-AHRW hash function at the L2 cache level for the selected socket S_i , and pick a L2 cache L_j (path b). Finally, at the core level, CA-AHRW picks the desired core C_k from the selected L2 cache L_j (path c). The properties of connection locality, load balancing and cache-awareness hold true at each level in the tree because the corresponding hash functions at each level are CA-AHRW. In addition, H-CAHRW also provides the following properties.

- **Reduced computation cost**. Suppose the complexity of the original HRW hash function and the adjustment multiplier computation is H. The complexity of CA-AHRW hash scheduler with flat-level linear search is $O(H \cdot N \cdot M)$ for N nodes and M scheduling units. On the other hand, the hash-tree scheduler selects a node by tree traversal. Thus, with the same denotation, the complexity is reduced to $O(H \cdot lgN \cdot M)$. In general, the scheduling overhead is reduced from N to lgN under the same workload and platform.
- **More effective load balancing**. Note that while the hash space remains in the range of $[0,2^{31}-1]$, H-CAHRW reduces the search space from N to lgN, which leads to faster computation for load balancing at each level. In addition, the hash-tree scheduler essentially uses multiple hashes per input key. This behavior not only reduces the possibility of hash collision, but also progressively improves load balancing for the overall system.

6.5 Experimental Results

6.5.1 Experiment Setup

We implement and evaluate our scheduler on both Intel Xeon E5335 and AMD Opteron 2350. The Intel server has two Quad-core Xeon E5335 processors with 2.0GHz frequency. Each core has a 128KB dedicated L1 cache. Two cores share a 4M L2 and four cores share a socket with 8M L2 cache as shown in Fig. 6.5. The AMD server has two Quad-core AMD Opteron 2350 processors with 2.0GHz frequency. The cache hierarchy includes 64KB of dedicated L1 cache and 512KB of dedicated L2 cache per core, with a 2MB L3 cache shared by four cores as in Fig. 8. Both systems are running Linux-2.6.18 OS.

The major implementation issue of the H-CAHRW scheduler is 1) how to provide a fast computable pseudo-random HRW hash function, and 2) how to construct the communication matrix. To solve the first issue, we follow the definition of HRW hash function proposed in Eq. 6.2. The identifier vector $\vec{c}$ is the stream ID and each S_i represents a PU ID. For H-CAHRW, S_i is decided by the level of the tree. In regard to the communication matrix, since there are only three different communicating overhead parameters for Intel machine (core, L2 and socket level) and two for AMD machine (core and L3 level), we adopt a trial-and-error approach. We test a large number of communicating overhead combinations for both machines and choose a group with the best throughput performance to construct the matrix. For Intel machine, we choose 2, 4, 12 as the core, L2 and socket level communication overhead ratio. For AMD machine, we choose 1.5 and 3 for that ratio of the core and L3 level, respectively.

Table 6.1 Four resolution criteria in MP3s

Rate	Frame Size	Frame Rate (fps)	Bit Rate (kbps)
SQCIF	128 × 96	15	50
QCIF	176 × 144	15	70
CIF	352 × 288	26	100
4CIF	704 × 576	30	200

Table 6.2 Four resolution criteria movies

Rate	Frame Size	Frame Rate (fps)	Bit Rate (kbps)
Kung Fu Panda	352 × 240	29.97	920
Get Smart	352 × 240	29.97	920
Little Miss Sunshine	352 × 240	29.97	920
The Legend of 1900	352 × 240	29.97	920

Table 6.1 and Table 6.2 show the transcoding specifications and four used movies. We assume 20 concurrent streams as suggested in [28] for a full loaded system. Each stream requires a different transcoding operation for a chosen movie, including frame size scaling, frame rate and bit rate alteration. We parallelized the *FFmpeg* software and implement our H-CAHRW scheduler along with five other scheduling schemes, including Round Robin (RR), Stream-based Mapping (SM), HRW, H-AHRW, CA-AHRW. The performance shows that our scheduler improves the throughput and video quality for a variety of workloads with little scheduling overhead, better load balancing and fewer cache misses.

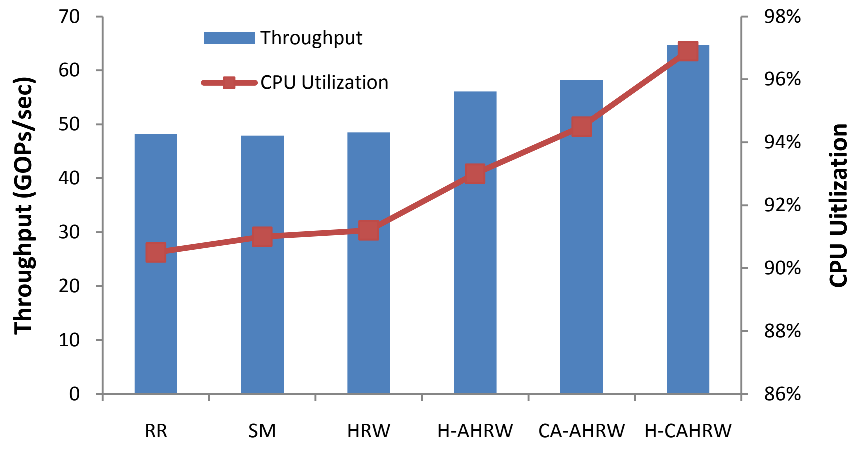


Fig. 6.6: Throughput and CPU utilization performance on Intel Xeon E5335.

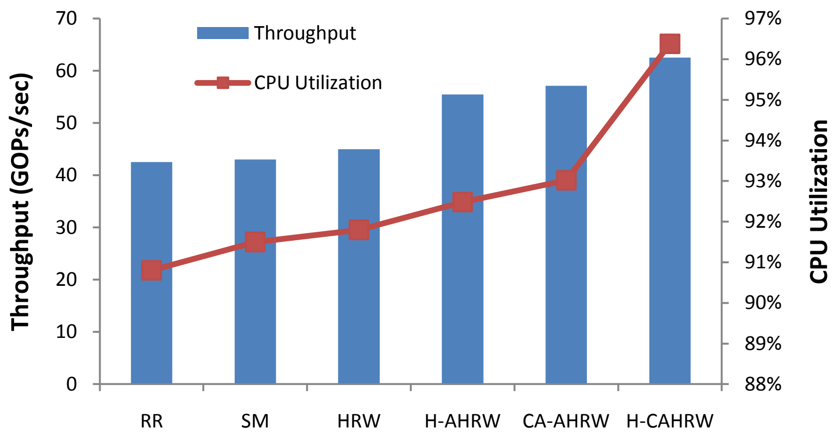


Fig. 6.7: Throughput and CPU utilization performance on AMD Opteron 2350.

6.5.2 Throughput Performance

Fig. 6.6 and Fig. 6.7 present the throughput and average CPU utilization for six schedulers. We measure the throughput in terms of processed GOPs per second for the whole system. From both Figures, we observe that throughput and CPU utilization follow the same trend, where H-CAHRW achieves the best performance. H-CAHRW shows an average of 25.7% and maximum of 35.1% throughput improvement over other schemes on Intel machine, as well as an average of 30.5% and maximum of 47.0% on AMD machine. Although RR has relatively even distribution of GOPs among all cores, it suffers from poor cache performance and substantial synchronization overhead in preserving GOP order due to its lack of stream locality. SM and HRW produce similar throughput compared to RR, because their strict stream locality cause load imbalance when the stream distribution is not even. As regard to H-AHRW and CA-AHRW, although they perform better than HRW because of their adaptive feedback mechanism, their improvement is limited without considering the underlying core/cache topology. H-CAHRW, taking advantage of the flow locality, adaptive load balancing and cache awareness, outperforms all other schemes.

6.5.3 Load Balancing Performance

Fig. 6.8 and Fig. 6.9 visualize the load balancing results on two platforms by the CPU utilization for each core. We can make three observations from these two figures. First, H-CAHRW, H-AHRW and RR expose better load balancing than other three schemes, with H-CAHRW performing the best of all, which is in line with the corresponding overall CPU utilization performance in the previous section. Second, the two stream-locality schemes, namely SM and HRW, suffer imbalanced load distribution, especially on core 3. This is because the workload scheduled on core 3 is far less compared to other cores. Third, as opposed to SM and HRW, H-AHRW and CA-AHRW are capable of relaxing the rigid stream locality for better load balancing. However, their overall CPU is still less utilized as shown in Fig. 6.6 and Fig. 6.7 with more imbalanced workload compared to H-CAHRW.

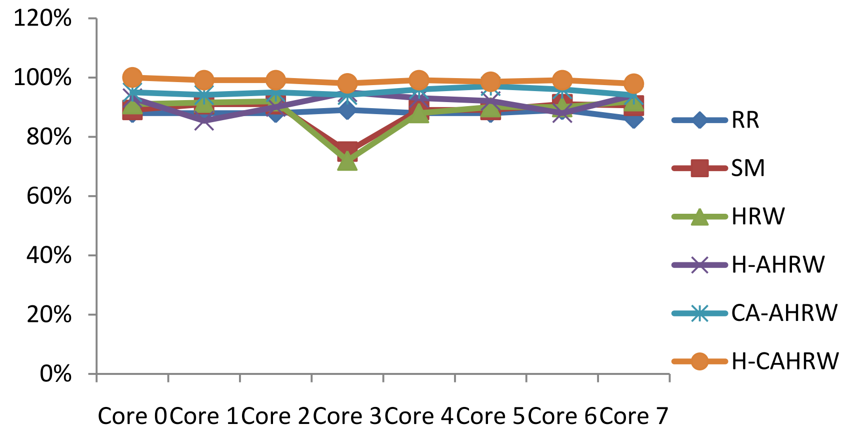


Fig. 6.8: CPU utilization for each core on Intel.

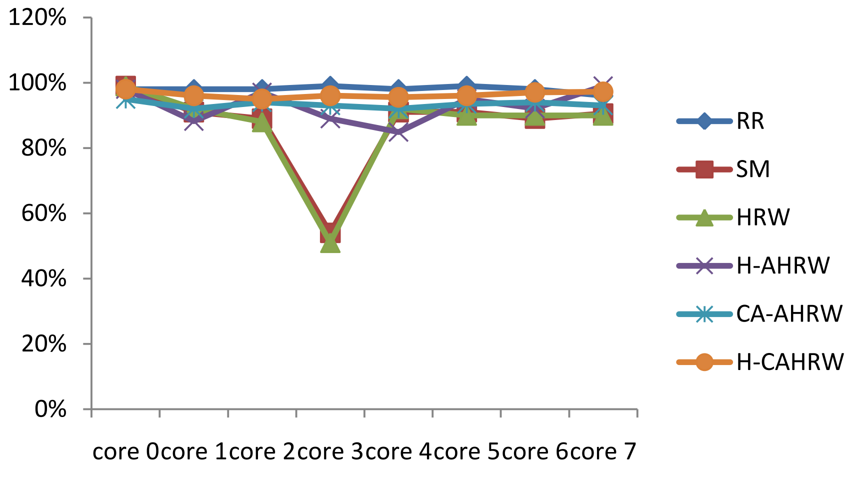


Fig. 6.9: CPU utilization for each core on AMD.

6.5.4 Cache Performance

In Fig. 6.10, we compare the cache performance in terms of L2 data cache miss rate measured by PAPI-3.7.0 [66]. On both machines, CA-AHRW incurs relatively lower miss rate compared to other load balancing schemes such as RR, H-AHRW and CA-AHRW. It reduces the miss rate by as much as 42.9% on Intel machine and 22.9% on AMD machine in comparison with RR because of the cache-awareness adaptive

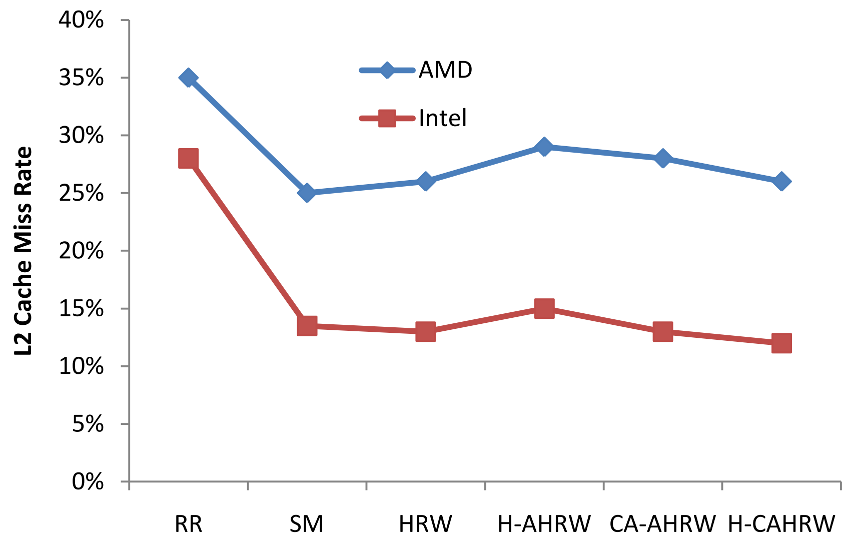


Fig. 6.10: L2 data cache miss rate on Intel and AMD

scheduling design, which fully utilizes the shared cache for different core/cache topologies. In addition, we observe that SM and HRW have slightly better cache performance than H-CAHRW due to their stream locality feature. However, that margin is very narrow, especially for AMD machine, where each core has its own dedicated L2 cache. Considering the poor throughput performance and inherent load imbalance problem in SM and HRW, we believe H-CAHRW is the only outstanding scheduler that is able to attain both high throughput and good load balancing with comparable cache performance.

6.5.5 Scheduling Overhead

Fig. 6.11 and Fig. 6.12 exhibit the average transcoding time per GOP and the associated scheduling overhead on Intel and AMD servers. We obtain the average transcoding time per GOP from the inverse of the throughput performance shown in Fig. 6.6 and Fig. 6.7. Notice that the average time to process a GOP is tens of milliseconds. The corresponding scheduling overhead represents how much time is spent during the computation of decision making. From both figures, we see that RR and SM involve the least overhead due to their simple schemes. With respect to HRW-based schemes, the scheduling overhead is proportional to the hash function complexity. Surprisingly, H-CAHRW only incurs slightly more time than other adaptive schedulers. Nonetheless, the scheduling overhead falls in the magnitude of microseconds, which is negligible compared to GOP processing time. Therefore, it is believed that H-CAHRW's sophistication and advantage do not incur substantial scheduling overhead increase at all.

6.5.6 Video Quality

Lastly, we measure the video quality by the out-of-order departure rate, which describes how many GOPs in a stream depart out of order on average, and delay jitter, which is the standard deviation of the GOP inter-departure time, as shown in Fig. 6.13 and Fig. 6.14. These metrics reflect how smooth a stream will be played. It is clear that the two metrics follow the same pattern in both platforms, where H-CAHRW performs comparably well among other adaptive HRW-based schedulers. In both figures, RR suffers the most due to its random distribution of GOPs without stream locality, which results in large number of out-of-order GOPs, and high delay jitter, accordingly. On the contrary, SM and HRW schemes are able to maintain a very good video quality by stream locality, although they have load imbalance problem. The remaining three adaptive HRW-based schedulers perform between the former two extreme cases with a reasonable video quality. All three schemes relax stream locality for better load balancing at both the stream and GOP level, thus causing some degrade in video quality. However, H-

CAHRW, strengthening in both throughput and video quality, still stands out among all other schedulers.

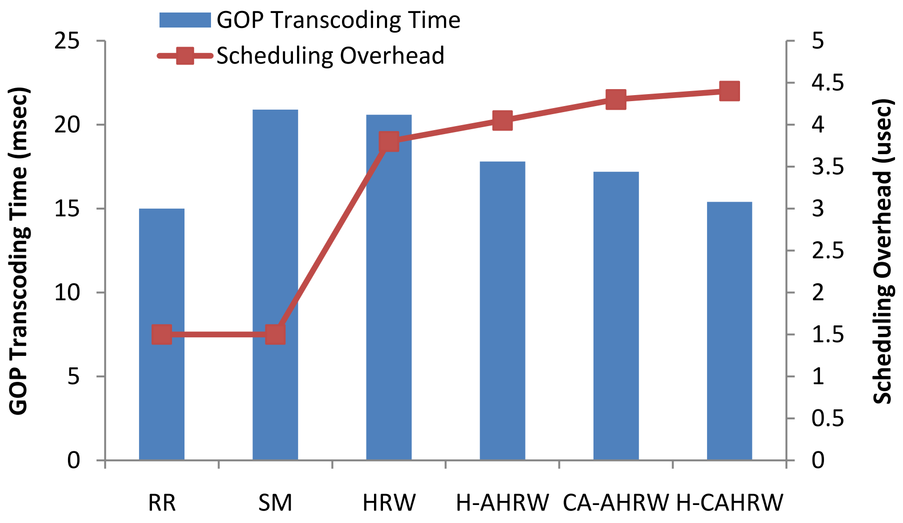


Fig. 6.11: Scheduling overhead and GOP transcoding time on Intel Xeon E5335.

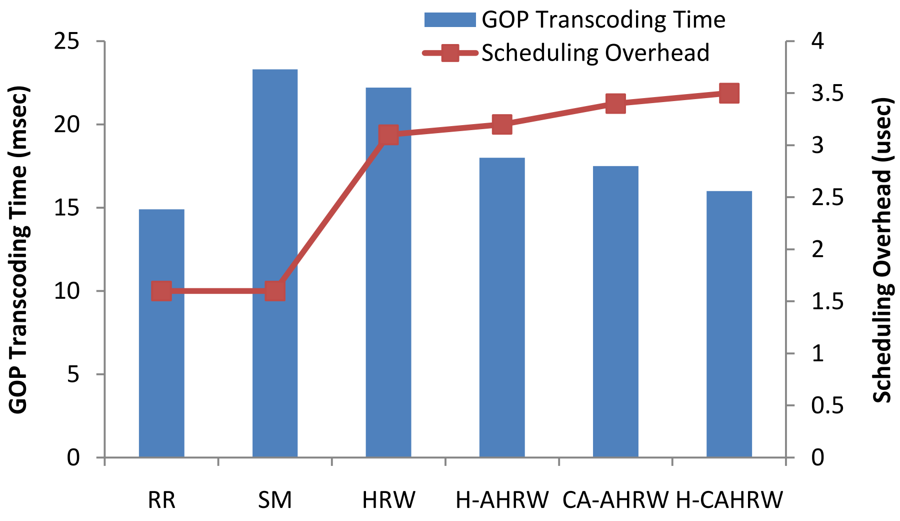


Fig. 6.12: Scheduling overhead and GOP transcoding time on AMD Opteron 2350.

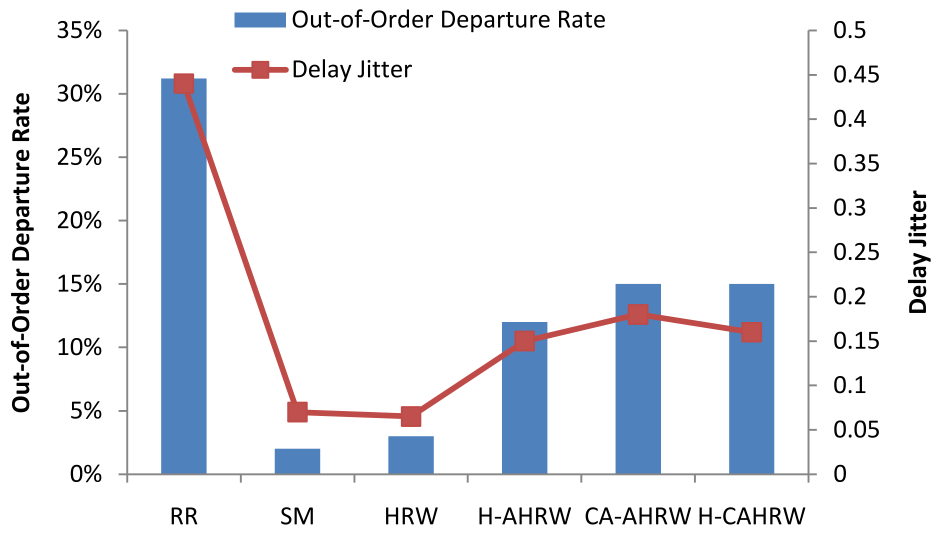


Fig. 6.13: Out-of-order departure rate and delay jitter on Intel Xeon E5335.

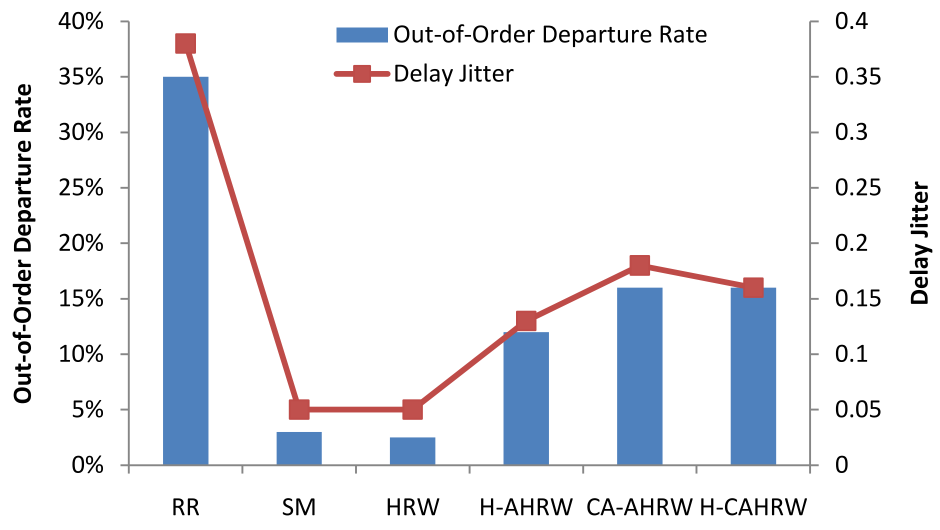


Fig. 6.14: Out-of-order departure rate and delay jitter on AMD Opteron 2350.

6.6 Summary

HRW-based hash schedulers have not considered cache-awareness in multicore architectures nor have been applicable for different core/cache topologies. In this chapter, we proposed CA-AHRW, a Cache-Aware AHRW scheduler. Our scheduler is capable of achieving both flow locality and load balancing in a cache-aware manner by adaptively changing the hashing decision based on real time workload difference and inter-core communication. Based on the generalized CA-AHRW, we further designed, implemented and evaluated a hash-tree scheduler, H-CAHRW, aiming at mainstream tree-based multicore architectures. The performance of H-CAHRW hash-tree scheduler was compared with five other scheduling schemes to show that this scheduler improves the throughput and video quality for a variety of workloads with little scheduling overhead, better load balancing and fewer cache misses.

Chapter 7

QoS Aware Proportional Share Hash Scheduler (PS-HRW)

From Chapter 4 to Chapter 6, we have introduced several schedulers that balances connection locality, load balancing and core/cache topology. In this chapter, we consider another important aspect of network I/O scheduling - Quality of Service (QoS).

QoS scheduling allocates a proportional share of the processing resources to each process according to the weight of the process. General Processor Sharing (GPS) [8, 19, 42, 64] was a theoretically ideal scheduler that provides QoS guarantee. GPS scheduling was also extended to multiple links scenario [9, 65]. While GPS and its extension provide theoretical QoS guarantee to ensure fairness in multicore scheduling based on the weight of each process, it is impractical to implement in real systems. In addition, QoS guarantee usually sacrifices connection locality, load balancing and core topology [27]. We have not seen any previous studies that consider this trade off.

In this chapter, we propose a proportional share hash scheduler, PS-HRW, based on our previous studies on hash based scheduling and incorporate QoS concerns into the scheduling decision. The PS-HRW consists of three steps. In the first step, PS-HRW calculates the proportional share of each service request, φ_i, based on the runqueue length of the request. Then, it allocates an integral number of cores $\left\lfloor g_i = \frac{\varphi_i}{\sum_j \varphi_j} r \right\rfloor$ based on H-CAHRW. In the third step, the residual request $g_i - \lfloor g_i \rfloor$ is allocated using a partitioning theory for routing requests to heterogeneous caches [74]. This theory generates a weight vector for H-CAHRW that follows strictly to the capacity of each PU. H-CAHRW then picks the proper core to host the residual request. The property of H-CAHRW guarantees that only the core that balances connection locality, load balancing and cache/core topology is picked.

We implement PS-HRW on an Intel Xeon web server running both L7-filter and FFmpeg and compare the performance with H-CAHRW and the partition-based scheduler proposed in paper [27]. We show that PS-HRW achieves the best balance among many performance aspects including system throughput, and scheduling overhead.

7.1 Definition of QoS scheduling

Modern web servers should be able to provide QoS to each client that subscribes to the required service. Generally, the QoS requirements can be assessed in terms of users' subjective wishes or satisfaction with the quality of the application performance, cost, and so forth. The assessment results are then mapped onto measurable QoS parameters to which the router needs to guarantee. For example, the QoS parameters can be the message response time, the end-to-end delay or the long term message processing rate. When performing QoS aware message scheduling, the router needs to relate the resource consumed by each message type with its QoS requirements. Basically, we classify the service provided by a web server into several service classes, each corresponding to a specification of the resource requirement and the QoS parameter. The QoS aware scheduling algorithm aims to provide differentiated service as follows:

- **Fairness**. The system resource is allocated proportionally among the service classes.
- **Independent allocation**. Given sufficient incoming traffic, a service class receives at least as much resources as were assigned to it irrespective of the traffic of other service classes.
- **Work conserving**. Resources not used by some service class may be distributed among other service classes.

The term "fairness" has colloquial meanings. In QoS, we define "fairness" following the General Processor Sharing (GPS) theory. A GPS server is work conserving and operates at a fixed rate r. By work conserving, we mean that the server must be busy if there are packets waiting in the system. It is characterized by positive real numbers $\varphi_1, \varphi_2, \ldots, \varphi_N$. At any given time, a flow is either backlogged or idle. Let $S_i(\tau, t)$ be the amount of session itraffic served in an interval $(\tau, t]$. A session is backlogged at time t if a positive amount of that session's traffic is queued at time t. The a GPS server is defined as one for which

$$\frac{S_i(\tau,t)}{S_j(\tau,t)} \geq \frac{\varphi_i}{\varphi_j}. j = 1,2, \ldots N \tag{7.1}$$

for any session i that is continuously backlogged in the interval $(\tau, t]$.

Summing over all sessions j:

$$\sum_{j=1,2,\ldots,N} S_j(\tau, t) = \frac{1}{(t - \tau) \cdot r}$$

$$S_j(\tau, t) \sum_j \varphi_j \geq (t - \tau) \cdot r \cdot \varphi_i$$

And session i is guaranteed a rate of

$$g_i = \frac{\varphi_i}{\sum_j \varphi_j} r \tag{7.2}$$

We propose a quantitative definition of the QoS requirements for L7-filter and FFmpeg. Following [2, 13, 27], the computation time of each request directly influences the QoS performance of network applications. In L7-filter, connection buffers are feed into the classification engine on a per byte basis for the state machine, the computation time is linear to the size of the connection buffer. In FFmpeg, the transcoding process is also strictly proportional to the size of the message size. Therefore, we can further simplify the QoS requirement for both applications to the size of runqueue in terms of byte for each processing request.

In the HRW scheduling case, let $\varphi_1, \varphi_2, \ldots, \varphi_N$ be the runqueue size for each request, and r be the overall number of cores on a server. Then following GPS, each request should be assigned $g_i = \frac{\varphi_i}{\sum_j \varphi_j} r$ number of cores.

In [27, 28], we see a partitioning algorithm that satisfies this constraint in a cluster of servers. However, that algorithm allocates the proportional share of clusters in a round robin fashion, which fails to consider core/cache topology and load balancing in a multicore system. In addition, the system scheduler handles the fractional allocation in that paper automatically. Therefore, the partitioning algorithm is oblivious to the scheduling decisions for the fractional part. Based on our previous studies in HRW schedulers, we know that HRW can achieve high system throughput by satisfying connection affinity, load balancing and core/cache topology. Our problem now is how to incorporate QoS concerns into HRW.

7.2 PS-HRW

Objective: Given a service request C_i, with weight φ_i, overall system service rate r, the allocated system resource g_i should be proportional to its weight assignment, $g_i = \frac{\varphi_i}{\sum_j \varphi_j} r$. Meanwhile, system throughput, load balancing, core/cache topology and scheduling overhead should be balanced.

Solution:

We propose a take a 3-step solution to achieve the objective. In the first step, PS-HRW calculates the weight of each service request, φ_i, based on the runqueue length of the request. Then, it allocates an integral number of cores $\left\lfloor g_i = \frac{\varphi_i}{\sum_j \varphi_j} r \right\rfloor$ based on H-CAHRW. In the third step, the residual request $g_i - \lfloor g_i \rfloor$ is allocated using a partitioning theory for routing requests to heterogeneous caches [74]. This theory generates a weight vector for H-CAHRW that follows strictly to the capacity of each PU. H-CAHRW then picks the proper core to host the residual request. The property of H-CAHRW guarantees that only the core that balances connection locality, load balancing and cache/core topology is picked.

7.2.1 Weight Calculation

For both L7-filter and FFmpeg, we can obtain the size of each request in terms of bytes by measuring the length of the connection buffer and stream size. The desired weight for request i at time t, $\varphi_i(t)$, can be calculated based on Eq. 7.3.

$$\varphi_i(t) = size_i(t) \tag{7.3}$$

Then the proportional allocation of cores for request i, g_i, can be calculated based on Eq. 7.4.

$$g_i(t) = \frac{\varphi_i(t)}{\sum_{j=1}^{M} \varphi_i(t)} \cdot r = \frac{size_i(t)}{\sum_{j=1}^{M} size_j(t)} \cdot r \tag{7.4}$$

where M is the number of concurrent requests being processed in the system and r is the number of cores.

7.2.2 Integral Core Allocation

For each $g_i(t)$, we use H-CAHRW to allocate the greatest integer number $\lfloor g_i(t) \rfloor$ that is smaller than the required cores $g_i(t)$. Since $g_i(t)$ can be greater than 1, the algorithm will recursively call H-CAHRW to allocate cores until the residual value $g_i(t) - \lfloor g_i(t) \rfloor$ is smaller than 1.

7.2.3 Residual Core Allocation

In this step, we use a partitioning theory for routing requests to heterogeneous caches [74]. We can apply that theory to our packet scheduler, where incoming packets function as URL requests and the cores function as caches. Let $p_1, \ldots, p_N$ be given arbitrary target probabilities for each of all the N cores. If a core has target probability p_i we desire the fraction p_i of requests to be mapped to it.

In the robust hashing scheme, for a given request $\vec{c}$ we calculate a hash value $h = h(\vec{c}, p_i)$ based on H-CAHRW for each of core i. We then map the request to the core that has the highest h. This scheme will map l/N of the connection buffers to each core.

To deal with target capacities, we introduce multipliers $x_1, \dots, x_N$ and multiply each h with the respective x_i , we then map the request to the core that has the largest $Z_i = x_i \cdot h_i$ value. If the multipliers are different, the fractions of requests routed to the PU will no longer all be the same. Specifically, we want to assign all the cores that has been used in step 2 with the capacity value $1/N'$ and the rest of the cores with $\frac{1}{N'}(g_i(t) - \lfloor g_i(t) \rfloor)$. (Suppose c cores have been allocated in step 2, $N^{'} = c + (N - c) \cdot (g_i(t) - \lfloor g_i(t) \rfloor)$). The intuition behind this algorithm is to use H-CAHRW and the heterogeneous partitioning theory to pick the residual proportional share of cores that balances between connection locality, load balancing and core/cache topology.

Theorem 7.1 - Let $p_1, \dots, p_N$ be given target capacities for each core. Reorder the cores so that $p_1 \leq \cdots \leq p_N$. Let $x_1 = (N \cdot p_1)^{1/N}$ and let $x_2, \dots, x_N$ be recursively calculated as follows: $x_n = \left[\frac{(N-n+1)(p_n - p_{n-1})}{\prod_{i=1}^{n-1} x_i} + x_{n-1}^{N-n+1}\right]^{\frac{1}{N-n+1}}$. Then the robust hash algorithm, H-CAHRW with multipliers $x_1, \dots, x_N$ will be allocate the fraction p_i of requests to the i^{th} core for $i = 1, \dots, N$.

Proof - Let $x_1, \dots, x_N$ be an arbitrary set of nonnegative multipliers satisfying $x_1 \leq x_2 \dots \leq x_N$. Let $h_1, \dots, h_N$ be the H-CAHRW hash values associated with each of the N cores. Because the outputs of a hash function can be taken to be independent, uniformly distributed random variables, without loss of generality, we take each h_i to be uniformly distributed over $[0,1]$. Let $Z_i = x_i h_i$ be the i^{th} multiplied hash value. Note that the Z_is are independent and that Z_i is uniformly distributed over $[0, x_n]$. Let

$Z_{(i)} = \max(Z_1, \dots, Z_{i-1}, Z_{i+1}, \dots, Z_N)$. Let q_i be the probability that core i has the largest multiplied hash value, that is, $q_i = P(Z_{(i)} \le Z_i)$. Conditioning on $Z_i = x$, we obtain

$q_i = P\left(Z_{(i)} \le Z_i\right) =$

$\frac{1}{x_i}\int_0^{x_i} P\left(Z_{(i)} \le x\right)dx = \frac{1}{x_i}\int_0^{x_i} \prod_{j \ne i} P(Z_j \le x)\, dx = \frac{1}{x_i}\int_0^{x_i} \frac{\prod_{j=1}^{N} P\left(Z_j \le x\right)}{P(Z_i \le x)}\, dx =$

0xi1xi=1NPZi≤xdx=j=1Nxj−1xj1xk=1iP(Zk≤x)dx , (7.5)

where $x_0 = 0$. We must now get an explicit expression for the product in the above expression.

For $x_{j-1} \le x \le x_j$,

$$P(Z_i \le x) = \begin{cases} 1, & i \le j-1 \\ \dfrac{x}{x_j}, & i \ge j \end{cases}$$

Thus, for $x_{j-1} \le x \le x_j$,

$$\prod_{i=1}^{N} P(Z_i \le x) = \left[\prod_{i=1}^{j-1} P(Z_i \le x)\right]\left[\prod_{i=j}^{N} P(Z_i \le x)\right] = \prod_{i=j}^{N} \frac{x}{x_j} = \frac{x^{N-j+1}}{\prod_{i=j}^{N} x_i} \qquad (7.6)$$

Insert Eq. 7.5 into Eq. 7.6 gives

$$q_i = \sum_{j=1}^{N} \int_{x_{j-1}}^{x_j} \frac{x^{N-j}}{\prod_{i=j}^{N} x_i}\, dx = \sum_{j=1}^{i} \frac{1}{\prod_{k=j}^{N} x_k} \frac{1}{N-j+1}\left(x_j^{N-j+1} - x_{j-1}^{N-j+1}\right)$$

$$= \left(\prod_{k=1}^{N} x_k\right)^{-1} \sum_{j=1}^{i} \frac{\left(\prod_{k=1}^{j-1} x_k\right)\left(x_j^{N-j+1} - x_{j-1}^{N-j+1}\right)}{N-j+1}$$

We have one degree of freedom. Set $\prod_{k=1}^{N} x_k = 1$. Then

$$q_i = \sum_{j=1}^{i} \frac{\left(\prod_{k=1}^{j-1} x_k\right)\left(x_j^{N-j+1} - x_{j-1}^{N-j+1}\right)}{N-j+1} \qquad (7.7)$$

From Eq. 7.7 we have

$$q_i = q_{i-1} + \frac{(\prod_{j=1}^{i-1} x_j)\left(x_i^{N-i+1} - x_{i-1}^{N-i+1}\right)}{N-i+1} \quad (7.8)$$

The desired result follows by setting $q_i = p_i, for\ i = 1, \ldots, N$,and solving for x_i in Eq. 7.8. End of Proof

The robust hash function with multipliers $x_1, \ldots, x_N$ in the above theorem is part of CARP [74]. Our algorithm is shown in Table 7.1

Discussion

Because the computation in a QoS scheduler is intensive, it can only be calculated and updated at a realistic rate. An additional data structure is required to register the current connection-to-core mapping.

We only apply this algorithm periodically, rather than on a per packet basis. Only when the processing cost of a connection φ_i varies by more than a tolerable percentage, say β should we recalculate the mapping.

We choose different value of β to update the weight vector $(\varphi_1, \varphi_2, \ldots, \varphi_m)$. A greater value of β reduces the recomputation but loses scheduling accuracy.

We also force the scheduler to update the weight vector $\varphi(\varphi_1, \varphi_2, \ldots, \varphi_m)$ any time when a connection finishes classification or a new connection arrives at the system.

Table 7.1 PS-HRW Algorithm

```
Input:      $\varphi_i, i = 1, \dots, N$

            $\overrightarrow{c_i}, \mathrm{i} = 1, \dots, \mathrm{N}$
Output:     $setc_i, \mathrm{i} = 1, \dots, \mathrm{N}$
Algorithm:
    for each $i = 1, \dots, N$
        if $\varphi_i$ varies by more than $\beta$ then
            $setc_i$={}                                   // Step 1
            $g_i(t) = \frac{\varphi_i(t)}{\sum_{j=1}^{M} \varphi_i(t)} \cdot r = \frac{size_i(t)}{\sum_{j=1}^{M} size_j(t)} \cdot r$
            while $g_i(t) > 0$ then                       // Step 2
                $setc_i = setc_i \cup HCAHRW(\overrightarrow{c_i})$
                $g_i(t) -= 1$
                                                          // Step 3
            for each core $j$
                if $j \in setc_i$ then
                    $p_j = 1/N'$
                else
                    $p_j = \frac{1}{N'}(g_i(t) - \lfloor g_i(t) \rfloor)$
            Recursively calculate vector $X = (x_1, \dots, x_N)$
                using Theorem 7.1
            reset $h(\overrightarrow{c_i})$ to $X \cdot h(\overrightarrow{c_i})$ in H-CAHRW
            $setc_i = setc_i \cup HCAHRW'(\overrightarrow{c_i})$
```

7.3 Experimental Results

We implement and evaluate our scheduler Intel Xeon E5355. The Intel server has two Quad-core Xeon E5335 processors with 2.0 GHz frequency. Each core has a 128KB dedicated L1 cache. Two cores share a 4M L2 and four cores share a socket with 8M L2 cache. We compare our results with all the previously proposed schedulers. In addition, we also develop a QoS scheduler based purely on HRW, instead of H-CAHRW. The trace files for L7-filter has been given in Table 5.1 and the Mpeg files has been given in Table 4.1 and 4.2.

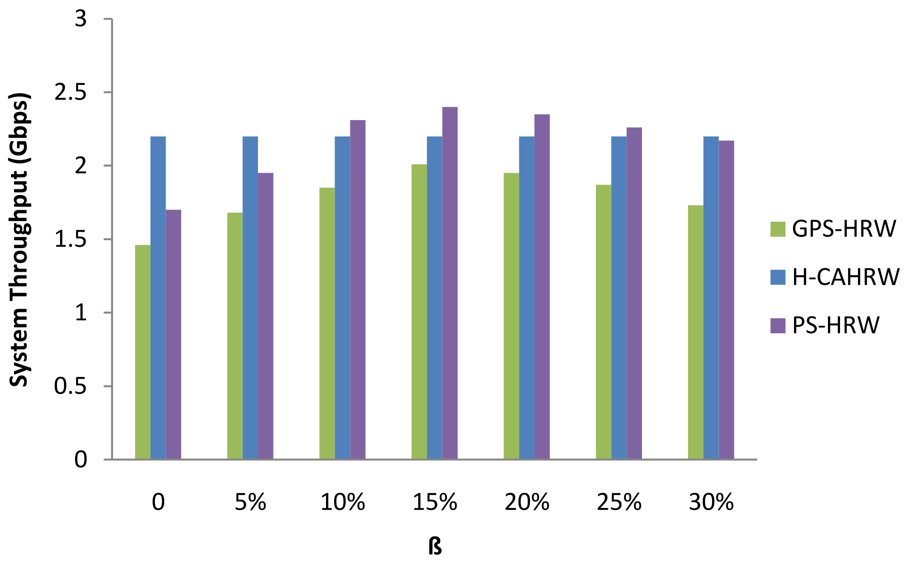


Fig. 7.1: Impact of scheduling frequency $\boldsymbol{\beta}$ on system throughput

7.3.1 Selection of Scheduling Frequency

In this experiment, the impact of different scheduling frequency is shown. A greater value of β reduces the accuracy of the QoS, because it requires less update to the scheduler and hence less remapping of the service requests to the core. However, it also reduces the system overhead due to less computation. Fig. 7.1 shows that a value of 16% gives the best result of performance-overhead trade off. For the rest of the experiments, we choose this value as the test bed.

7.3.2 System Throughput

Fig. 7.2 and 7.3 shows the system throughput of PS-HRW compared to previously proposed schedulers. To address the importance of the selection of the auxiliary scheduler, we also show the result of a pure QoS extension to the original HRW, GPS-HRW, in the figures. We observe a minimum of 10% degradation in system throughput using GPS-HRW for both L7-filter and FFmpeg. In addition, GPS-HRW, although more effective than AP, provides less throughput than PS-HRW and H-CAHRW based schedulers. On the other hand, PS-HRW generates throughput comparable to that of those throughput-centric schedulers. Therefore, we can see the importance of core/cache topology in the scheduler.

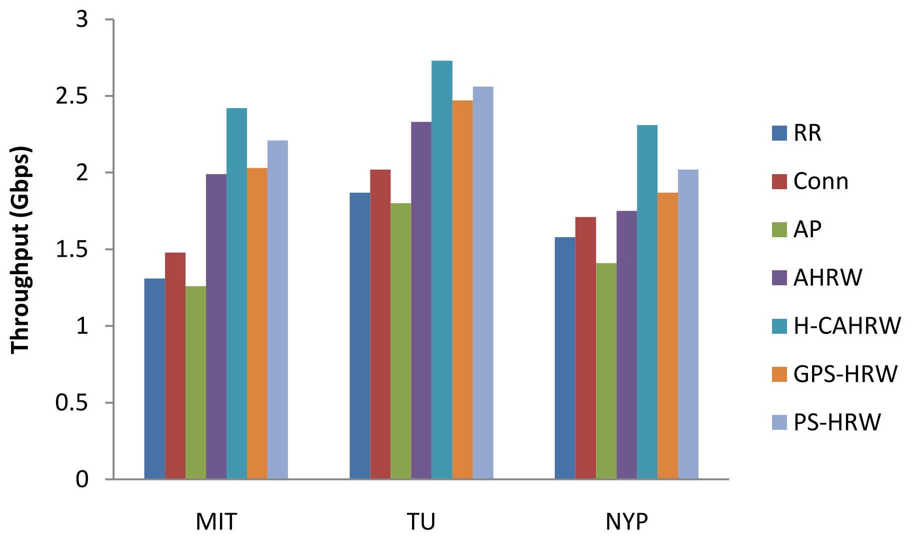


Fig. 7.2: System throughput of L7-filter

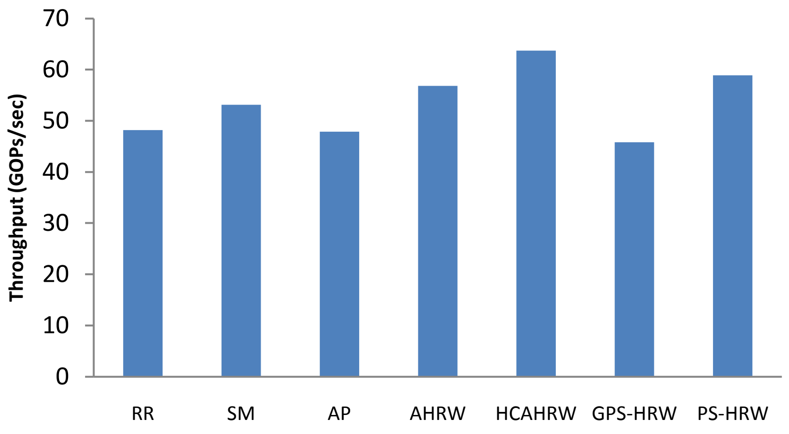


Fig. 7.3: System throughput of FFmpeg

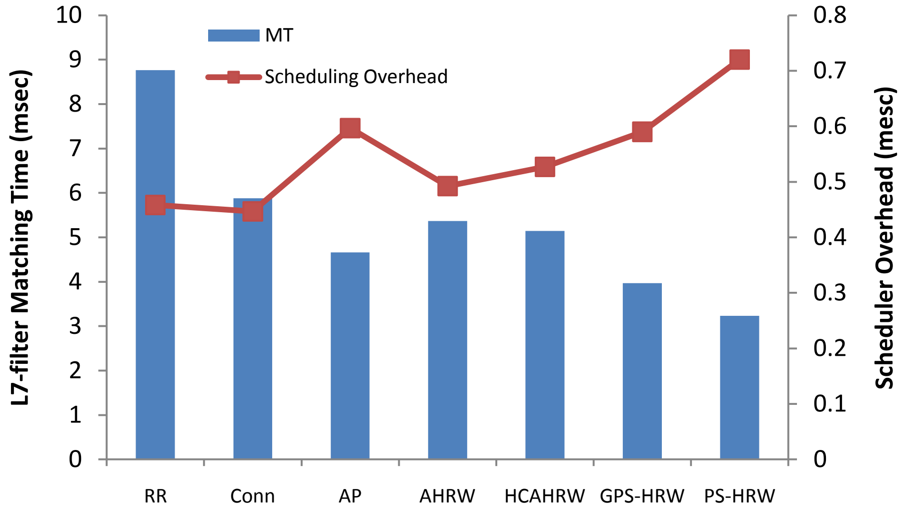


Fig. 7.4: Scheduling Overhead of L7-filter

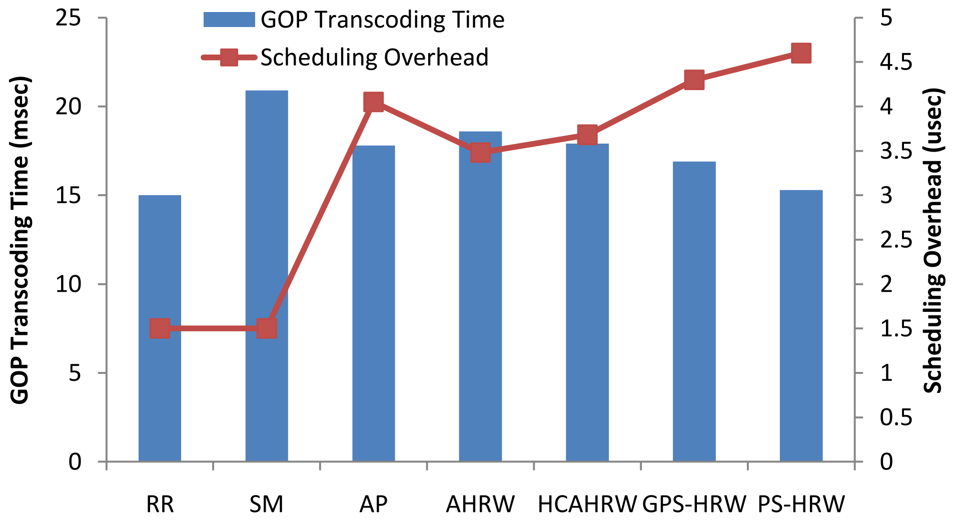


Fig. 7.5: Scheduling overhead of FFmpeg

7.3.3 Scheduling Overhead

As one of the major drawbacks of QoS based scheduling, the overhead incurred by the additional computation is a key concern for our design. Because PS-HRW is applied on a

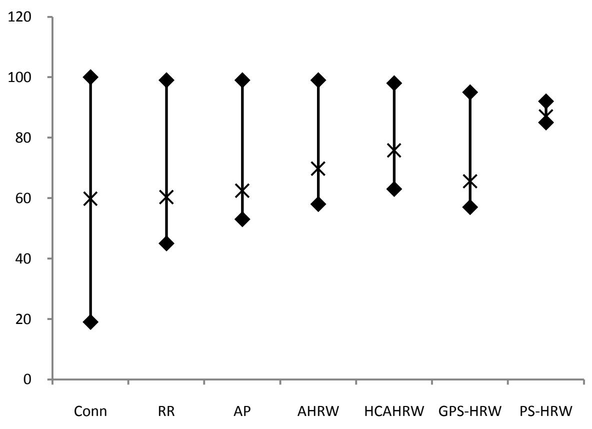


Fig. 7.6: Load balancing of L7-filter

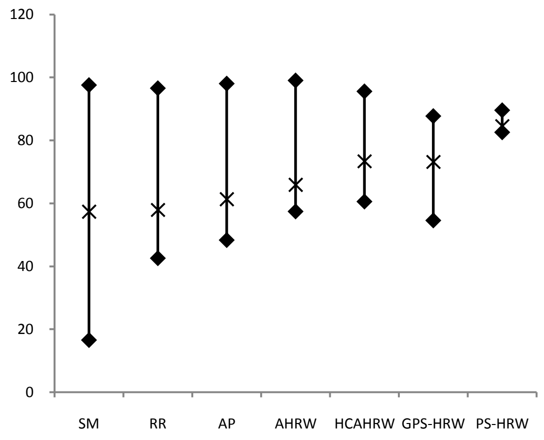
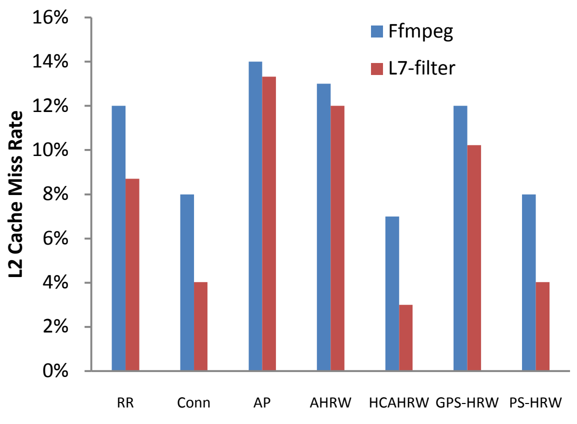


Fig. 7.7: Load balancing of FFmpeg

connection granularity, it is expected to have a lower overhead compared to the other schedulers that schedule on a per packet basis. On the other hand, the heuristic computation for QoS requirements are more complicated than throughput-centric schedulers. Therefore, we insert timestamps along the processing path of a packet and obtain the execution time for each component. Fig. 7.4 and 7.5 show the overhead of different schedulers. As expected, AP, GPS-HRW and PS-HRW incurs additional overhead compared to their baseline scheduler. Because of the calculation used to satisfy the QoS requirements, such overhead is inevitable. However, we also find that this overhead takes less than 9% of the overall system execution time for L7-filter and less than 0.2% for FFmpeg.

7.3.4 Load Balancing

Fig. 7.6 and Fig. 7.7 visualize the load balancing results by the range of CPU utilization for each scheduler. The cross ("X") represents the average CPU utilization. We make three observations from these two figures. First, hash based schedulers expose better load



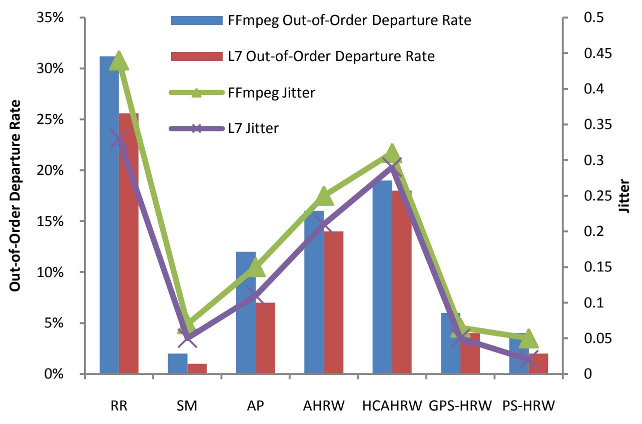


Fig. 7.8: L2 Cache Miss Rate

Fig. 7.9: Out-of-Order Departure and Jitter

balancing than pure connection locality based schemes. PS-HRW performs the best, due to the hierarchical packet level load balancing technique adopted by H-CAHRW. Second, AHRW and PS-HRW achieves better balancing due to the packet level adjustment. Third, compared to PS-HRW, AP and GPS-HRW incurs a higher degree of imbalance due to the obliviousness of load balancing. This observation shows that H-CAHRW balances the workload better with the hierarchical steps than the original linear HRW scheduler does.

7.3.5 Cache Misses

In Fig. 7.8, we compare the cache performance in terms of L2 data cache miss rate measured by PAPI-3.7.0 [66]. Our first observation is that AP and AHRW incurs high cache misses due to the obliviousness of cache/core locality. Both "conn" and "HRW" cause less cache misses because of the strict connection locality constraint. H-CAHRW causes the least cache misses due to the communication matrix. PS-HRW inherits the cache awareness from H-CAHRW, and incurs the least cache misses compared to GPS-HRW.

7.3.6 Out-of-order Departure and Jitters

Lastly, we measure the video quality by the out-of-order departure rate and delay jitter. The former metric describes how many GOPs/connection buffers in a stream depart out of order on average, while the latter is defined the standard deviation of the interdeparture time among GOPs/connection buffers. High jitter is detrimental to the playback quality for FFmpeg and wasting computing resources for L7-filter, which is the main concern of media clients. The results are illustrated Fig. 7.9. These metrics reflect the playing quality of streams is clear that the two metrics follow the same pattern, where RR suffers the most due to its random distribution of scheduling units without stream locality, which results in large number of out-of-order scheduling units, and high jitter. On the contrary, SM is able to maintain a very good video quality by stream locality, although it has throughput and load imbalance deficiency illustrated in Fig. 7.2, 7.3, 7.6 and 7.7. On the other hand, despite the throughput advantage using AP [28], we observe that this group of schedulers sacrifices video quality. Therefore, it is not the best choice when QoS requirement is the first performance priority. PS-HRW, strengthening both throughput and video quality, stands out among all other schedulers.

7.4 Summary

In this chapter, we propose a Proportional Share HRW based scheduler, PS-HRW. This scheduler balances system performance by provisioning QoS guarantees. Compared with previous works, PS-HRW also inherits connection locality, packet level load balancing, core/cache topology from H-CAHRW. We implement PS-HRW on a real web server and the results show that PS-HRW provides the best QoS guarantee in terms of out-of-order departure rate and delay jitter. Meanwhile, it provides comparable system throughput compared to existing throughput-centric schedulers. In addition, we also show that PS-HRW incurs only an additional 2% of scheduling overhead compared to the hash based heuristic scheduling, and causes little last level cache misses, which becomes increasingly important in multicore development.

In the future, we plan to deploy the proposed algorithm on Cisco's UCS blade on both the physical devices and the virtualized router and switches. We believe the trend of QoS aware high performance scheduling has a great future in the era of cloud computing.

Chapter 8

Schedulers in Virtualization

Virtual Machine (VM) technology is experiencing a resurgent interest as the ubiquitous multicore processors have become the de facto configuration on modern web servers. Multicore servers potentially provide sufficient physical resources to realize VM's benefits including performance isolation, manageability and scalability. However, the network performance of virtualized multicore servers falls short of expectation. It is therefore important to understand the overhead implications.

In this Chapter, we evaluate the network performance of a virtualized multicore server using a TCP streaming microbenchmark (*Iperf*) and SPECweb2005. We first motivate our research by presenting the performance gap between native and virtualized environment. We then break down the overhead from an architectural viewpoint and show that the cache topology greatly influences the performance. We also profile the Virtual Machine Monitor (VMM) at a function level to illustrate that functions in the current version of the Xen scheduler are the major contributors to the poor utilization of cache topology. Consequently, we implement a static onloading scheme to separate interrupt handling from application processes and execute them on cores with cache affinity. Based on the observed benefits, we modify the Xen scheduler to migrate virtual

CPUs dynamically to exploit the cache topology. Our results show that the VM performance improves by an average of 12% for *Iperf* and 15% for SPECweb2005.

8.1 Experimental Testbed and Methodology

8.1.1 Benchmarks

In this chapter, we use a microbenchmark, *Iperf*, and the SPECweb2005 benchmark suite to simulate the real world server workload. *Iperf* [6, 91] is widely chosen as an I/O intensive benchmark that measures TCP and UDP bandwidth performance. We instrument *Iperf* to run a user-configurable computational-intensive workload. As a result, every time a packet is received, the next packet has to wait until the computation is finished before it gets a chance to be processed. It simulates a real time network application, where the inter-packet latency is directly affected by the computation workload. With such modifications, we could bind *Iperf* to one core with processor affinity for the computation part, while binding interrupt processing to another core with interrupt affinity to service the I/O. SPECweb2005 [82] is a suite of synthetic workloads: Banking (HTTPS), E-commerce (HTTP and HTTPS), and Support (HTTP); agreed to by major players in the WWW market. Many well recognized papers [11, 83] have used it to emulate server behavior of large numbers of independent web clients with heterogeneous requests. Each of the three workloads consolidates web request processing, database transactions and dynamic workload calculation. We increase operating system limits where necessary to scale. We also set TCP options to values similar to those used on high-performance commercial web servers.

8.1.2 Server Software

The native test server ran the Linux kernel 2.6.18, which was compiled with support for the Intel 64 (x86 64) architecture. The virtualization environment is hosted by Xen 3.3.1 and a Vanilla Linux kernel 2.6.18. We chose software with freely-available source code to ease profiling. We used Apache HTTP Server 2.2.9—a multithreaded server that provides one thread per flow and scales at least to tens of thousands of concurrent sessions [48]. We also used the PHP 5.2.6 script engine for the SPECweb2005 PHP script and configured the Apache HTTP Server to improve scalability. The mpm worker module was used to provide a combined multi-process and multi-thread server with one thread per connection. Table 8.1 shows the settings used with the module. We also set `EnableSendfile` to “on” to allow the server to transmit file data without first copying it to userspace.

Table 8.1 Web Server Threads Settings

Settings	**Value**
ServerLimit	900
StartServers	10
MaxClients	28800
MinSpareThreads	25
MaxSpareThreads	75
ThreadsPerChild	32
MaxRequestsPerChild	0

8.1.3 Testbed Configuration

Both the server and client are Intel Clovertown machines, which is a two-processor platform based on the quad-core Intel Xeon processor 5300 series [33] delivering 8-thread, 32- and 64-bit processing capabilities with 8 MB of L2 cache per processor. It is equipped with 16GB DRAM and 1333 MHz system bus. To conduct virtualization experiments using 10GbE networks, we connected these two Clovertown machines end-to-end via two Intel 10GbE networks [32]. Since the virtualized system has not supported TCP/IP offloading engine (TOE) and Jumbo Frames yet, all the experiments in this paper were conducted without TOE support and with a MTU of 1500 bytes. We retained the default settings in the Intel's device driver without specific performance tuning on interrupt coalescing, write combining and network buffer sizes etc.

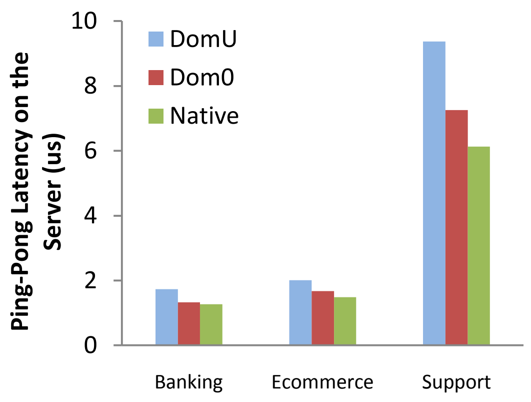
We use a single 10GbE NIC for all the VMs on the server machine. Despite the assumed scalability advantage of multiple NICs (1GbE) for VMs, it also introduces a significant overhead in managing traffic flows, i.e. packet demultiplexing, interrupt dispatching, etc. In addition, it is not practical to expect using one NIC for each individual VM. On the other hand, bandwidth at 10GbE level is more than enough to handle the network requirement of all the VMs. Therefore, optimizing network performance in a single 10GbE NIC is still a valid and very important research topic.

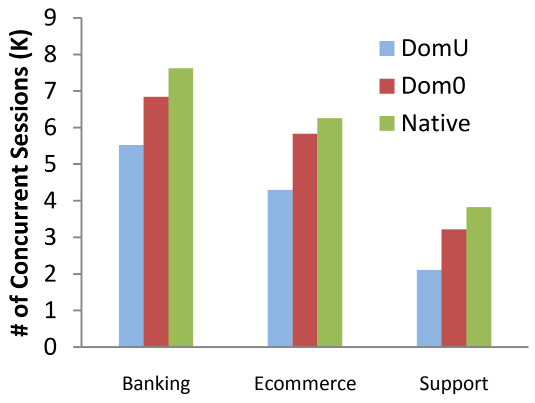
SPECweb2005 also requires at least one backend database simulation server (Besim) as part of the testbed. For simplicity, we configured the client machine to double as a Besim. During our experiments, we found that putting "Client"' and "Besim" on the same physical machine does not saturate its CPU (maximum 240% core utilization). Therefore, it was guaranteed that the performance bottleneck did not exist on the client side.

8.1.4 Measuring Performance

The throughput of SPECweb2005 benchmark was measured by the number of concurrent sessions supported by the server while meeting the QoS requirements of the benchmark workload, i.e. error rate. We manually booted up the SPECweb2005 performance in step of 100 (concurrent sessions) until the benchmark fails the QoS requirement, i.e. error rate < 3%. The ping-pong latency was measured by running the benchmark NetPIPE [59]. It marks the round trip delay of a ping-pong request from the client to the server and reflects how busy the server is. The greater the latency, the busier the server is running the major workloads.

To ease and expedite our web server experiments, we made several modifications to SPECweb2005's default settings. As such, our results are not suitable for comparison to formally published SPEC results. We made the most significant change to the DIRSCALING option, which we changed from 0.25 to 0.00625. This option adjusts the total number of files served by the web server, but it does not change the probability distribution of the files received by the clients. We made this change so that the file set would fit in the test web server's 16GB of main memory. We warmed the server's file cache before running experiments by sequentially reading through all the files. Without this change, our web server became heavily bound by I/O wait time due to reading from a single disk drive. To overcome I/O wait time, production web servers generally use large disk arrays. A recent official SPECweb2005 result for a similar system used 44 disk





(a) (b)

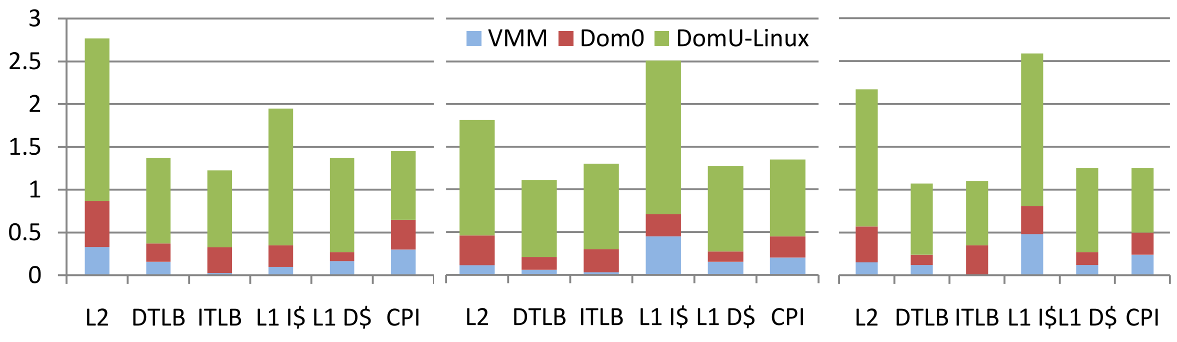
Fig. 8.1 Performance Comparison of SPECweb 2005 in Native OS and VM

(a) Ping-Pong Latency. (b) # of Concurrent Sessions

drives for the file set [88], which allowed this result to exceed the maximum number of users that our server supported. Thus, a sufficiently provisioned disk array would remove the I/O wait bottleneck.

We used Xenoprof [96] to collect the architectural event information for different domains. Xenoprof was developed based on Oprofile 0.9.4 [62], a system-wide profiler for Linux systems, capable of profiling all running code at low overhead. We treated Dom0 and DomU equally as active domains in Xenoprof and collect measurement results separately.

In order to analyze the functional level overhead along packet processing, we developed a tool to anatomize the life of a packet and quantify performance from the architecture perspective. It instruments the VMM, Dom0, DomU and network protocol stack along the packet processing path. We adopted a performance counter based approach, where a small piece of code is manually inserted into the points of interest.



(a) Support (b) Banking (c) Ecommerce

Fig. 8.2: Architectural events for different workload. It also illustrates the individual contribution of VMM, Dom0 and DomU-Linux

Those code records the current time-stamp of the measuring points into a buffer using the corresponding Intel Performance Counter [33]. The overhead of the instrumental code is small (only 90 CPU cycles for a timestamp read and 70 cycles for a performance counter read) and is subtracted from the measurement.

8.2 Virtualization Analysis Under 10GbE Network

8.2.1 Virtualization Performance Overhead

Fig. 8.1 presents the ping-pong latency and the number of concurrent sessions for the three workloads of SPECweb2005 at different level of the virtualized environment. The maximum core utilization among all cases is 530%, which means the performance is not bounded by processing power.

As we go up the virtualization layer, more overhead occurs. Among the three workloads, we find Support generates the biggest performance gap between native and virtualized environment - a degradation of 45% in DomU. Banking and Ecommerce incur 31% and 38% performance degradation respectively in VM compared to native OS. Compared to the other two workloads, Support has no encrypted connections and emulates file downloads from Internet clients. It provided the simplest possible abstraction of how a realistic application interacts with network processing and produced the largest networking load, so it was best suited to understand the performance of the network stack under heavy load. Therefore, compared to database transaction and other computational-intensive operations, the virtualized network stack incurs the most significant overhead.

8.2.2 Architectural Characterization

As we observed, virtualization incurs significant overhead for packet processing. In order to understand the loss in performance from architectural standpoint, we have gathered performance data from the architectural event counter for SPECweb2005. We use Xenoprof to read the count of the hardware events available on Xeon processor, which can monitor various micro-architectural activities, including cache miss, TLB miss, branch prediction rate, etc. All counter values are normalized with respect to native Linux (as "1"). We also present the overhead contributed by VMM and Dom0 to the overall performance overhead. Note that the DomU overhead in Fig. 8.2 is the overhead introduced by guest OS alone, excluding that of Dom0 and VMM. Because the delay of domain switching (context switches between Dom0 and DomU) is also counted for DomU, the guest OS, albeit almost identical to a native OS, is shown to incur more overhead compare to the native case.

For all the 3 workloads, Fig.8.2 shows that CPI increases in virtualized environment, with 50%, 37% and 30% for Support, Banking and Ecommerce, respectively. This observation explains the difference in the number of supported client sessions between the three workloads, as shown in Fig. 8.1.

We find that L1 ICache and L2 cache misses are the most sensitive components to the virtualization environment for all the three workloads. Particularly for Support, the L2 cache misses incurred in VM are 2.7X compared that of its value in native OS. We will show later in Fig. 8.3 that L2 cache misses takes a much heavier weight compared to L1 ICache misses in the overall execution time, despite of the difference in the sensitivity to virtualization.

Fig. 8.2 also shows that TLB and cache misses (both Instruction and Data) increase between 1.3X to 2.7X compared to the native Linux. This is consistent with the higher CPI in the virtualized environment. System gets a higher penalty due to the increased cache and TLB misses. We show the breakdown of the total execution time for both native and virtualized environment into various architectural components in Fig. 8.3. We scale the execution time to 100% and show the percentage of each component, i.e. architectural event. In our testbed, L1 and

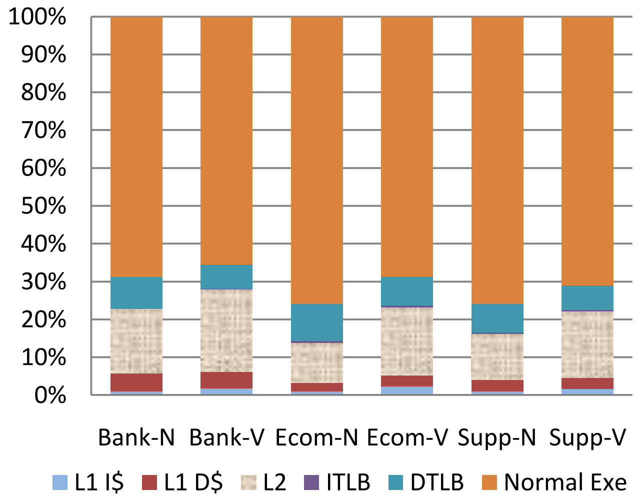


Fig. 8.3: Scaled Weight of Each Architectural Event towards Execution Time in SPECweb2005.

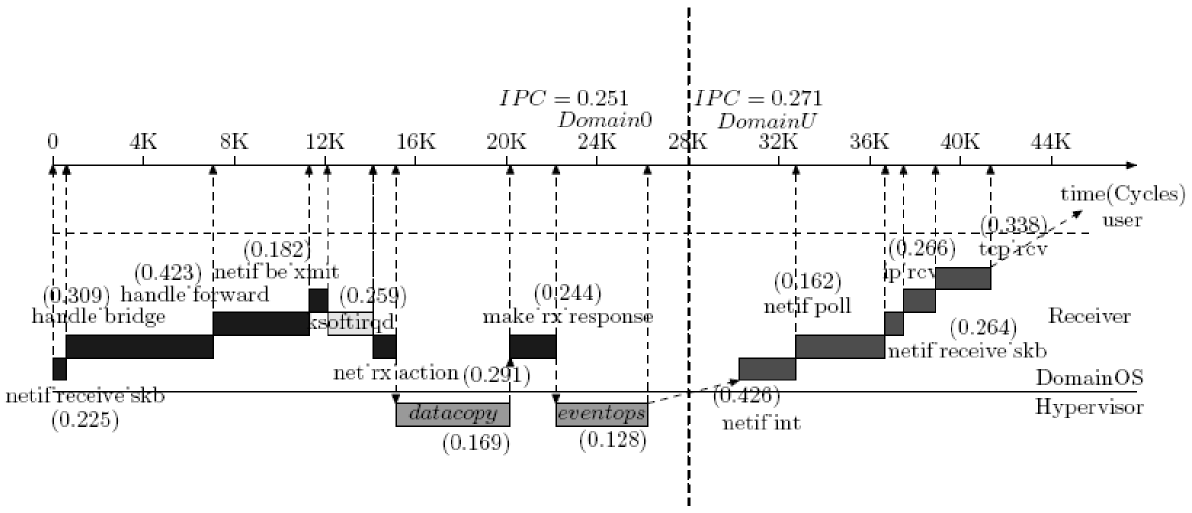


Fig. 8.4: Life-of-Packet Analysis on Receive Side

L2 cache misses incur 14 and 240 cycles penalty, respectively, and the average TLB miss penalty is 40 cycles [33]. We observe that: 1) normal instruction execution fraction for native Linux is higher than the virtualized instruction fraction because more time is spent in TLB and cache misses in a virtualized environment. 2) The increased cost from L2 miss penalty is the main contributor for performance loss in both native and VM Linux. In order to improve virtualized packet processing performance, L2 cache should be used more efficiently.

8.2.3 The Life-of-Packet Analysis

In order to pinpoint the contributors of the performance overhead in virtualization, we need to identify the functional level timing and bottleneck functions along the path of packet processing in a virtualized environment. In this section, we anatomize the life of a network packet based on an architectural characterization. We use the Iperf benchmark to present the life of a received packet in the web server because SPECweb2005 workloads introduce complications in userspace due to extensive use of PHP scripts. Paper [97] adopted a similar life of a packet for UDP on a 100 Mb Ethernet without virtualization. Due to the page limit, we only provide the life-of-packet analysis for the receive side in this section.

Receive side processing starts from the NIC interrupt. The VMM asynchronously passes all interrupts to Dom0 through an event channel mechanism. As shown in Fig. 8.4, the Ethernet driver in Dom0 takes over the received packets and passes them to the upper layer through `netif_receive_skb` method. Currently Linux Bridge module is used to switch the received packets to the corresponding guest domain, which consists of two main routines `handle_bridge` and `handle_forward`. According to destination MAC address in the received packet, it passes the packets to the corresponding Back-End Server (BE) via the Jhash algorithm. Once BE receives packet in the data copy mode, it calls the VMM to directly move packets between domains, and then notifies the guest domain through a lightweight inter-domain event channel (`eventops`) to pass on notifications. DomU gets scheduled next to execute interrupt handler netif_int and then acquires the received packet. Then the received packets are passed to the upper network stack layer through netif_receive_skb method. Finally the `ip_rcv` and `tcp_v4_rcv` in TCP/IP network stack are called sequentially as is done in Linux.

In Fig. 8.4, the bottom line is the border of the VMM and domain kernel space, and the horizontal dashed line divides the user/kernel space for domains. Dom0 and DomU are differentiated by the vertical dashed line. And the number near to functions is IPC (Instruction Per Cycle) for the corresponding function. The timeline scale in Fig. 8.4 is

4K CPU cycles. We also found that a latency of 1.6 μs is introduced by the scheduler in the VMM. It means that the current VMM scheduler is concerned about fairness more and is less latency sensitive. To verify this claim for a more general case (SPECweb2005), we use Xenoprof to list the major functions in the VMM that contributes to the L2 cache misses, which has been proved in Fig. 8.3 to be the main contributor to the overall performance delay.

Table 8.2 presents the results for all the three workloads in SPECweb2005. We find that two group of functions related to the Xen grant table mechanism (`do_grant_table_op` and `gnttab_copy`) and the credit scheduler (`csched_schedule`, `schedule_vcpu_wake`, and `vcpu_kick`) contribute in around 60% of the system L2 misses, independent of the workload. Other functions like `context_switch`, `__copy_to_user_II` are more generally used in the current Xen version for data copy. A recent work [92] has proposed using multi-queue and grant reuse mechanism to attack the deficiency in the grant table mechanism. In this paper, we focus on optimizing the credit scheduler.

Table 8.2 Major Contributors in VMM for L2 Misses

Support		**Banking**		**Ecommerce**	
Function	Percentage	Function	Percentage	Function	Percentage
do_grant_table_op	21.10%	do_grant_table_op	15.42%	do_grant_table_op	18.30%
csched_schedule	17.00%	csched_schedule	13.30%	csched_schedule	15.21%
csched_vcpu_wake	15.60%	vcpu_kick	11.60%	gnttab_copy	12.41%
evtchn_set_pending	13.65%	csched_vcpu_wake	9.35%	csched_vcpu_wake	9.60%
gnttab_copy	9.42%	context_switch	8%	__copy_to_user_II	7.32%
…	…	…	…	…	…

VM migration in Xen causes a lot of data copy. The current VMM Scheduler in Xen ignores the underlying cache topology of the multicore server. To achieve load balance, the scheduler moves the workload from the current core to another core with the least intensive workload. On modern multicore chips, cores are located in different groups with corresponding cache sharing topology. The communication cost between cores that share the last level cache is significantly lower than that between cores with different last level caches. The obliviousness of such important locality information incurs more time in scheduling. As a result, the algorithms in the scheduler need to be aware of the cache topology.

8.3 Dynamic Cache-Aware Scheduling in Credit Scheduler

8.3.1 The Credit Scheduler in Xen

The VMM in Xen functions as an abstraction layer of the real physical devices. As a result, scheduling in virtualization is based on Virtual CPUs (VCPU) because Physical CPUs (PCPU) are transparent to the guest VMs. Each guest VM can be arbitrarily allocated with multiple VCPUs. The Credit Scheduler is a proportional fair share CPU scheduler built from the ground up to be work-conserving on SMP hosts [83]. Its objective is to allocate the processor resources fairly, weighted by the predefined credits for each domain.

The credit-based scheduler in Xen organizes all the online VCPUs in a runqueue and always picks a workload (VCPU) from the head of the queue to run on the PCPU. In order to decide the position of VCPUs in the runqueue, the scheduler marks each of the online VCPU with one of the ternary states: OVER, UNDER and BOOST. Based on the remaining credits, a VCPU will be labeled OVER if it runs out of credits; UNDER if the remnant is positive; and BOOST if it meets the standard of UNDER and has a frequent wake-and-sleep behavior. All the VCPUs in BOOST state are placed in front of those in UNDER state in the runqueue, while those with OVER state are kept in the tail portion.

When it comes to multicore architecture, there are a few twists while the scheduler functions. First of all, before a PCPU goes idle, it looks on other PCPUs to find any run-able VCPU that might potentially use its own resource. A VCPU is run-able on all PCPUs as long as it is not affinitized to a certain PCPU. This guarantees that no PCPU idles when there is run-able work in the system.

In addition, domain migration might happen based on priority difference for each event notification. Whenever an event is notified to a target domain, the scheduler tickles the runqueue on the designated PCPU and re-evaluates the runqueue to see if the target VCPU should preempt the running VCPU and take over the processing resource. If this PCPU has at least two run-able VCPUs when such preemption occurs, they would be migrated into the idlers in the system when these idlers check to guarantee load balance as explained in the previous paragraph.

Moreover, the scheduler checks the state of the current running domain during each scheduler interrupt and redistributes the PCPU if necessary. If the processing VCPU has been BOOST since the last check, the state is reset to UNDER because other VCPUs in the runqueue might be more latency-sensitive than the current one. The original domain is then migrated to another online neighbor PCPU with the most idling neighbors in its grouping, if there is one. Otherwise, it is simply reinserted back into the runqueue. Meanwhile, if the current domain has been wakening and sleeping at a high frequency, it keeps BOOST and remains working. This ensures that latency-sensitive domains such as those processing I/O benefit from its PCPU share.

Table 8.3 Dynamic VM Migration Policy in Credit Scheduler

```
IF an event notification occurs THEN
    IF the target VCPU should preempt the current VCPU THEN
        Find the PCPU in the same cache domain;
        Move the current VCPU to the new PCPU;
    ENDIF
ENDIF
IF there is a timer interrupt THEN
    IF the current VCPU is in BOOST THEN
        Reset priority to UNDER;
        Find the highest priority VCPU in runqueue;
        Find the PCPU in the same cache domain;
        Move the current VCPU to the new PCPU;
    ENDIF
ENDIF
```

8.3.2 Dynamic Cache-Aware Scheduling

The VCPU migration in the current credit scheduler occurs when a VCPU remains UNDER for a while and there are other PCPUs that are idle. The scheduler chooses the target PCPU with the largest number idling neighbors in its grouping. If more than one PCPUs have the same number of idling neighbors, this policy distributes work across distinct sockets first and then distinct cores in the same socket.

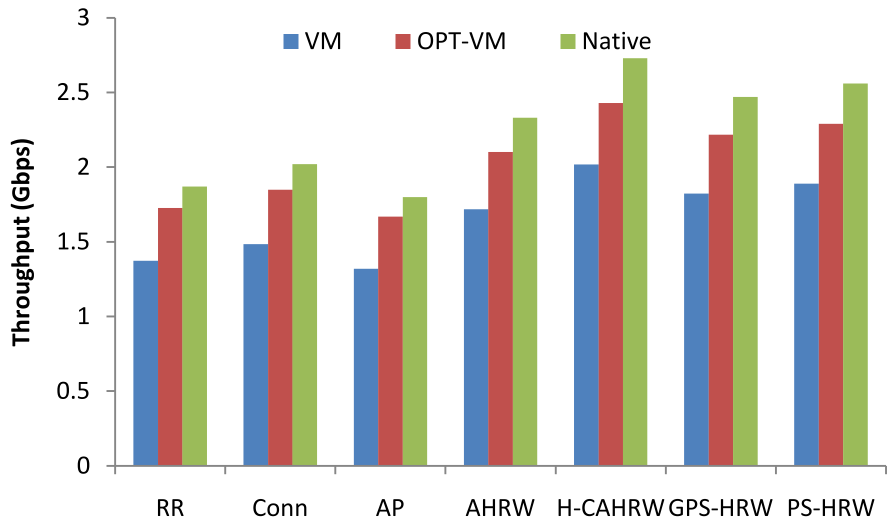
A drawback of the current migration policy is that it does not consider the influence of cache behaviors on the multicore system. This policy focuses on load balancing and guarantees scheduling fairness, which has been acknowledged to be beneficial to heavy workload. However, when there are not enough VCPUs competing for physical processors, the selection of migration destination affects the system performance. In this case, the lack of efficient utilization of the shared cache deteriorates the already limited network performance in VM.

Based on our core layout of Fig. 4.7, we propose the following VCPU migration scheme in Xen Credit Scheduler as shown in Table 8.3: When necessary (same condition as original case), the VCPU should be moved to the PCPU that shares the last level cache with the source PCPU. E.g. if VCPU is originally running on Core #0, then once migration occurs, if Core #2 is idle, it becomes the new vehicle to run this VCPU when it is waken up for the next time, otherwise the scheduler picks the core with the largest group of idlers in its neighborhood.

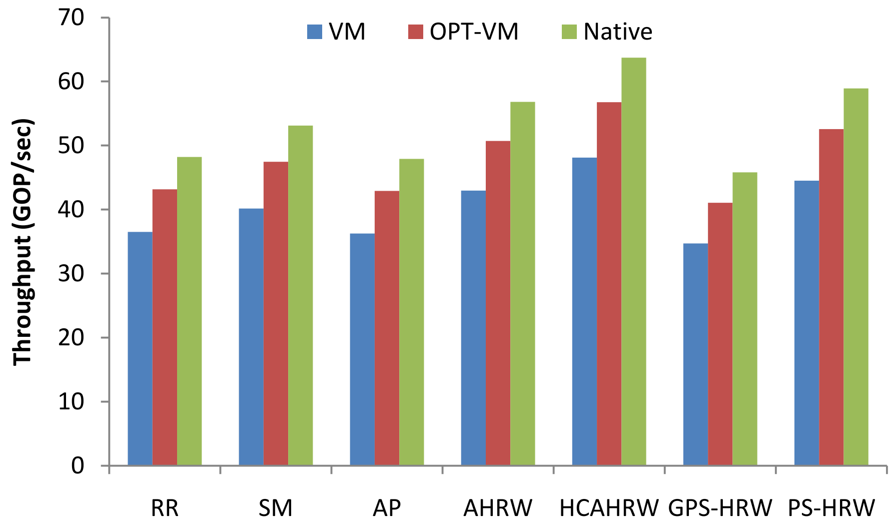
Essentially, we optimize the original migration policy to be cache-aware. That said, the temporal cache locality provided by the shared last level cache reduces cache miss in migration.

8.3.3 Optimization Results for L7-filter and FFmpeg

We test the efficacy of the dynamic cache-aware scheduling using L7-filter and FFmpeg and observe on average a 9.5% improvement for all our previous schedulers compared to applying them using the default credit scheduler. As shown in Figure 8.5, while the improvement is not as significant as the schedulers achieves in the native system, the importance is that it enables schedulers in the userspace in native systems to be developed in virtualization and sacrifices minimum performance overhead.



(a)



(b)

Fig. 8.5: Impact of Dynamic Cache Aware Scheduling for (a) L7-filter and (b) FFmpeg

8.3.4 Optimization Results for SPECweb2005

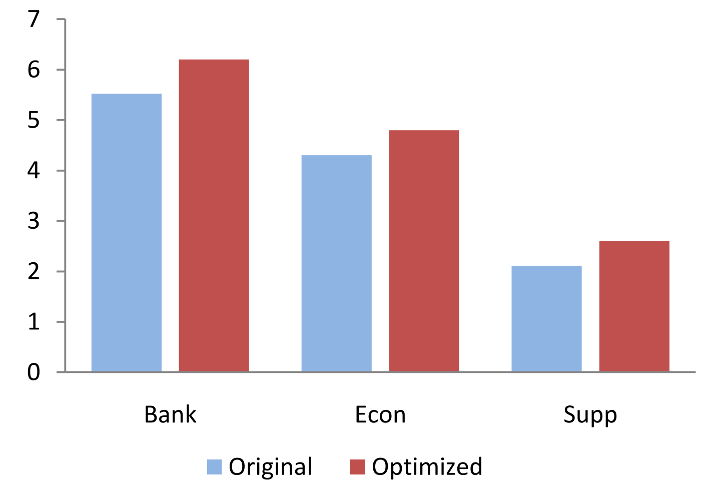
We use two nodes of 8 cores connected back to back with two 10GbE NICs. Connections are being made from one such node (with 8 cores) to the other node. The cache-aware scheduling is running on one node. The numbers shown in Fig. 8.5 (a) is the number of concurrent sessions supported by the server while meeting the QoS requirements of the benchmark workload, i.e. error rate. It is analogous to "Throughput" in the Iperf measurement. The aggregate throughput numbers can be calculated by Eq. 8.1

$$Throughput = \frac{Size\ of\ Transmitted\ \ Packet\ per\ Connection}{System\ \ Up\ Time\ (300\ ms)} \times (\#\ of\ Concurrent\ Sessions) \tag{8.1}$$

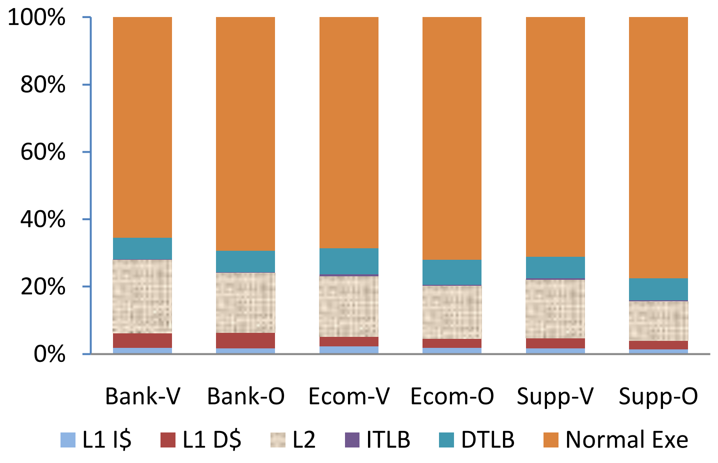
As Fig. 8.6 (a) shows, the number of supported concurrent sessions increases by an average of 15%. In order to verify our policy, we redo the experiment in Fig. 5 and present the results in Fig. 8.5 (b). In order to make a fair comparison, we tune down the performance of our optimized scheduler, so that the original case and the optimized case supports the same number of concurrent sessions. We can see that for all the three workloads, the fraction of execution time spent in L2 misses has been reduced and a proportional improvement is added to normal execution.

8.4 Summary

This paper carried out an exhaustive performance evaluation of a virtualized multicore server under 10GbE network with *Iperf* and SPECweb2005 benchmarks. We found that virtualization adds significant performance overhead to network packet processing especially running heavy network workload that frequently accessing the virtualized network stack. We designed a profiling methodology and presented a Life-of-Packet analysis and architectural characterization along the packet receive path. Our anatomy of life-of-packet provided functional and quantitative explanations of the observed extra virtualization overhead in latency. We discovered that the cache topology of the multicore server is one of the important architectural components for optimization in virtualized packet processing. To tackle this problem, we designed a dynamic cache-aware VCPU migration policy for scheduling virtual machines on physical CPUs under heavy workload like SPECweb2005. The results showed that the server performance can be improved by an average of 15%. In the future, we would like to further optimize packet processing performance by using emerging hardware technology and redesigning VMM architecture.



(a)



(b)

Fig. 8.6: Impact of Dynamic Cache Aware Scheduling for SPECweb2005:

(a) Improvement in the # of supported concurrent sessions.
(b) Reduction of L2 Cache Miss Cycles and Improvement in Normal Execution Cycles

Chapter 9

Conclusion and Future work

9.1 Conclusion

The widening spectrum of network applications drives the increasing demand of physical resources on both the network infrastructure and the web servers. As the unfolding of faster Ethernet shifts the bottleneck of network performance to the server end, both software and hardware vendors call for efficient solutions to deploy network applications on mainstream multicore web servers. Among many contributors to the network performance problem on multicore web servers, a primary concern is the scheduling mechanism of multithreaded programs to fully explore the available physical resources. In this domain, the scheduling mechanism should consider and balance between different influential factors including OS behavior on multicore architecture, network application characteristics, load balancing, core/cache topology and QoS requirement. In this dissertation, we discussed all the factors above and proposed several schedulers to solve the scheduling problem.

We broke down the processing of network application at the system level and analyzed the problem from both the kernel and the userspace level. Inside the OS kernel, we enabled the scheduling of interrupt processing by waking up a kernel thread in the process context to replace the original syscall in the interrupt context. This operation solves the starvation problem for the userspace application in the original case, when the bottom half of interrupt handler is non-preemptable. To take advantage of the cache topology in multicore web servers, the interrupt affinity based scheduler bind the interrupt processing on each core done through the kernel threads to one dedicated core, and bind the userspace processing to another core that shares the last level cache. Realizing the benefit of this scheduler is limited by the partial utilization of the cores, we moved to the userspace and investigated different optimization potentials.

In the userspace, we first parallelized two important network applications on modern web servers, L7-filter, a DPI extension of the Netfilter in Linux, and FFmpeg, a multimedia transcoding program. To decouple the performance bottleneck for the multithreaded applications from the noises of network stack, we developed a trace driven model that reads packet directly from the trace file instead of the network. Based on this model, we proposed an affinity based scheduler that schedules all the packets in the same connection to be processed on the same core. This scheduler guarantees the best utilization of shared data in the cache between packets in the same connection.

However, the performance gain from the affinity based scheduler can be offset by workload imbalance at different levels. At the connection level, the affinity based scheduler can scheduler an arbitrary number of connections to each core, oblivious to the per core workload. While this problem can be solved with a hash based scheduler that evenly distribute connections across all the cores, workload imbalance at the packet level remains unsolved. We therefore proposed AHRW based on HRW to relax connection locality to balance the workload at the packet level. AHRW schedules packets following the runqueue length per core, and guarantees that the workload difference among all the cores does not exceed a threshold. In addition, we developed a Hierarchical AHRW, H-AHRW, to progressively balances the workload corresponding to the topology of the core architecture.

Due to the variety of multicore architectures, we further generalize H-AHRW to H-CAHRW to incorporate the difference between inter-core communication costs. In each level of H-CAHRW, we adopted a communication matrix to reflect the core topology and multiplied it to the results of AHRW. The idea behind the additional multiplication is to increase the likelihood of scheduling packets in the same connection to cores in the same cache domain. Because of the complication of this matrix, we proposed a bottom-up weight calculation algorithm to reflect workload at different multicore levels.

We also proposed PS-HRW, a Proportional Share HRW scheduler, to take QoS requirement into scheduling consideration. PS-HRW maps connections to cores in three steps. It first calculates the weight of each connection by measuring the connection buffer size. It then assigns an integral number of cores to the connection according to its weight based on H-CAHRW. In the third step, the residual value is assigned following a partitioning theory that generates heterogeneous distribution for H-CAHRW. PS-HRW provides QoS guarantees and relies on H-CAHRW to balance between connection locality, load balancing and core/cache topology. In a cross comparison of all of our proposed schedulers implemented using real web servers, PS-HRW achieves the highest throughput with only negligible additional performance overhead.

As virtualization has gained resurgent interest since the prevalence of multicore servers, a common question is: how to transplant development in native systems to virtualization with minimal performance degradation. In order to answer this question, we conducted a thorough performance evaluation of a virtualized multicore web server running a consolidated network workload. This is the first report in its class that provides great potential for future development in VM. From this report, we observed the performance bottleneck in the VMM scheduler, and proposed a VCPU migration heuristic to reduce the last level cache misses. This scheduler was verified using L7-filter, FFmpeg as well as SPECweb2005. The experimental results show that our schedulers developed in native systems can be successfully ported onto a virtualized web server.

9.2 Future Work

In this dissertation, we proposed several schedulers in both the OS kernel and the userspace to optimize network applications for both native and virtualized multicore web servers. Based on the current results, we foresee at least three research directions to extend this work.

Firstly, multicore scheduling should incorporate power concerns. As DVFS and clock gating becomes increasingly popular as part of the on chip module in hardware, we see the trend of power-aware multicore chips being widely adopted by web servers. As a result, not only should the scheduler consider throughput performance of network applications, it should also aim at saving power whenever necessary. Essentially, the scheduler should co-exist with the DVFS module and direct network flows to comply with core frequency. The performance metric should be redefined as "throughput per watt", and the scheduling objective should be to maximize this value.

Secondly, hardware assistance can accelerate the performance of hash based schedulers. Although hash function calculation is expensive in software, it has a great potential to be deployed using hardware acceleration, such as additional heterogeneous assisting cores, FPGA or GPU. While these hardware accelerators can provide significant speed up, they also introduce a new area of study - software/hardware partitioning, i.e. when and where in the software program should be sent off to the hardware. If successful, the scheduler can be offload entirely to a dedicated hardware unit on chip.

Thirdly, abstraction of the scheduler can increase the practicality of our research. While scheduling problems cannot be perfectly solved as new applications keep calling for additional service requirements, i.e. more heuristics, the research in this dissertation should aim at an abstraction of the scheduling mechanism irrespective to the chosen application. We tend to deploy the proposed schedulers into Cisco routers and switches as part of the Cisco appliance. As the data center research is moving towards "cloud" computing, the virtualization studies in this dissertation is of great value to future development in this domain.

Bibliography

[1] AMD Opteron Machine, *http://www.amd.com/opteron*.

[2] E. Amir, S. McCanne, and R. Katz, An active service framework and its application to real-time multimedia transcoding, in *Proc. of ACM SIGCOMM,* Sept. 1998.

[3] P. Apparao, et al., Implications of cache asymmetry on server consolidation performance, *IEEE IISWC*, 2008.

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

[5] P. Barham, et al., Xen and the art of virtualization, *SOSP*, Oct 2003.

[6] J. Bass, TOE performance improvement for Linux 10Gbps Ethernet, *NCSU Whitepaper, http://www.chelsio.com/assetlibrary/pdf/redhat-chelsio-toe-final_v2.pdf*

[7] A. Bavier, A. Montz, and L. Peterson, Predicting mpeg execution times, in *Proc. of ACM SIGMETRICS*, June 1998.

[8] J. C. R. Bennett and H. Zhang. WF2Q: Worst-case Fair Weighted Fair Queuing. In *Proceedings of the IEEE INFOCOM*, San Francisco, March 1996.

[9] J. Blanquer and B. Ozden, Fair queuing for aggregated multiple links, *SIGCOMM* 2001.

[10] B. Brodie, et al., A Scalable Architecture for High-Throughput Regular-Expression Pattern Matching, *ISCA*, 2006.

[11] H. Cain, et al., An Architectural Evaluation of Java TPC-W, *HPCA*, 2001.

[12] Z. Cao, Z. Wang, and E. Zegura, Performance of hashing-based schemes for internet load balancing, in *Proc. of IEEE INFOCOM,* Apr. 2000.

[13] S. Chandra, C. S. Ellis, and A. Vahdat, Differentiated multimedia web services using quality aware transcoding, in *Proc. of IEEE INFOCOM* , Mar. 2000.

[14] A. Chandra, and P. Shenoy, Hierarchical Scheduling for Symmetric Multiprocessors, *IEEE Transaction on Parallel and Distributed Systems, Vol. 19, No. 3*, March 2008.

[15] Ho-Lin Chen, et al., On the impact of heterogeneity and back-end scheduling in load balance designs, *INFOCOM*, 2009.

[16] L. Cherkasova, et al., Measuring CPU overhead for I/O processing in the Xen virtual machine monitor, *USENIX*, 2005.

[17] Cisco SCE 2000 Series Service Control Engine, *http://www.cisco.com/en/US/products/ps6151/.*

[18] Cloud computing, *http://www.cisco.com/en/US/netsol/ns976/index.html*.

[19] A. Demers, S. Keshav, and S. Shenkar, Analysis and simulation of a fair queuing algorithm, *Internet. Res. and Exper., vol. 1*, 1990.

[20] FFmpeg, *http://ffmpeg.org/*.

[21] 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.

[22] Annie P. Foong , et. al., TCP performance re-visited, *IEEE International Symposium on Performance Analysis of Systems and Software,* 2002.

[23] A. Fox, S. Gribble, and Y. Chawathe, Adapting to network and client variation using active procies: Lessons and perspectives, *IEEE Perfonal Communication on Adaptation, special issue,* 1998.

[24] A. Fox, S. Gribble, Y. Chawathe, E. Brewer, and P. Gauthier, Cluster-based scalable network services, in *Proc. of Symposium on Operating Systems Principles*, Oct. 1997.

[25] D. Guo, et al., A scalable multithreaded L7-filter design for multicore servers, *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2008.

[26] D. Guo, et al., An adaptive hash-based multilayer scheduler for L7-filter on a highly threaded hierarchical multicore server, *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2009.

[27] J. Guo, F. Chen, L. Bhuyan, and R. Kumar, A cluster-based active router architecture supporting video/audio stream transcoding service, in *Proc. of th 17th Intl. Parallel and Distributed Processing Symposium (IPDPS),* Apr. 2003.

[28] J. Guo and L. Bhuyan, Load balancing in a cluster-based web server for multimedia applications, *IEEE Transactions on Parallel and Distributed Systems, No. 11*, Nov. 2006.

[29] N. Hua, et al., Variable-stride multi-pattern matching for scalable deep packet inspection, *IEEE INFOCOM*, 2009.

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

[31] Infiniband Trade Association. *http://www.infinibandta.org/*.

[32] Intel 10 GbE Server Adapters, *http://www.intel.com/Products/Server/Adapters/10-GbE-XFSR-Adapters/10-GbE-XFSR-Adapters-overview.htm*.

[33] Intel Xeon Machine, *http://www.intel.com/design/intarch/xeon/xeon.htm*.

[34] iperf, *http://sourceforge.net/projects/iperf*.

[35] iWarp, *http://www.iol.unh.edu/services/testing/iwarp/*.

[36] S. Jagabathula, et al., Fair scheduling through packet election", *IEEE INFOCOM*, 2008.

[37] 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.

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

[39] E. Katz, M. Butler, and R. McGrath, A scalable http server: the ncsa prototype, *Computer Networks and ISDN Systems*, 1994.

[40] Lukas Kencl, Jean-Yves Le Boudec, Adaptive load sharing for network processor, *IEEE INFOCOM*, 2002.

[41] Lukas Kencl, Jean-Yves Le Boudec, Adaptive load sharing for network processors, *IEEE Transactions on Networking, No. 2,* April 2008.

[42] L. Kleinrock, *Queueing SystemVol. 2: Computer applications*, New York: Wiley, 1976.

[43] S. Kumar, et al., Algorithms to accelerate multiple regular expressions matching for deep packet inspection, *SIGCOMM*, 2006.

[44] C. Li, G. Peng, K. Gopalan, and T. Chiueh, Performance garantees for cluster-based internet services," in *Proc. of IEEE ICDCS*, May 2003.

[45] G. Liao, et al., Software techniques to improve virtualized I/O performance on multi-core systems, *ANCS 2008: 161-170*.

[46] libnids, *http://libnids.sourceforge.net/*.

[47] J. Liu, et al., High performance vmm-bypass I/O in virtual machines, *USENIX*, 2006.

[48] C. MacCarthaigh, Scaling apache 2.x beyond 20,000 concurrent downloads, *ApacheCon*.

[49] 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*.

[50] A. Menon, et al., Diagnosing Performance overheads in the Xen Virtual Machine Environment, *VEE*, 2005.

[51] A. Menon, et al., Optimizing Network Virtualization in Xen, *USENIX*, 2006.

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

[53] A. Mitra, W. Najjar and L. Bhuyan, Compiling PCRE to FPGA for Accelerating SNORT IDS, *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2007.

[54] Jeffrey Mogul, TCP offload is a dumb idea whose time has come, *Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems*, 2003.

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

[56] Myricom. *Myrinet home page. http://www.myri.com/*.

[57] G. Narayanaswamy, An analysis of 10-Gigabit Ethernet protocol stacks in multicore environments, *A Symposium on High Performance Interconnects (HOT Interconnects)*, 2007.

[58] Netperf, *http://www.netperf.org/netperf/*.

[59] NetPipe benchmark, *http://www.scl.ameslab.gov/netpipe/*.

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

[61] D. Ongaro, et al., Scheduling I/O in Virtual Machine Monitors, *VEE*, 2008.

[62] Oprofile, *http://oprofile.sourceforge.net/*.

[63] M. Ott, G. Welling, S. Mathur, D. Reininger, and R. Izmailov, The journey active network model, *IEEE Journal of Selected Areas in Communications, No. 3*, Mar. 2001.

[64] Abhay K. Parekh and Robert G. Gallager, A generalized processor sharing approach to flow control in integrated services networks: the single-node case, *IEEE/ACM Transaction on Networking, Vol. 1, No. 3*, June 1993.

[65] A. K. Parekh and R. G. Gallager, A generalized processor sharing approach to flow control--- the multiple node case, *Tech. Rep. 2076, Lab. for Inform. and Decision Syst., M.I.T.*, 1991.

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

[67] P. Piyachon and Y. Luo, Efficient memory utilization on network processors for deep packet inspection, *ANCS*, 2006.

[68] Y. Qi, B. Xu, F. He, B. Yang, J. Yu, and J. Li, Towards high-performance flow-level packet processing on multi-core network processors, in *Proc. of ACM Symposium on Arch. for Networking and Comm. Systems (ANCS)*, Dec. 2007.

[69] K. Ram, et al., Achieving 10Gb/s using Safe and Transparent Network Interface Virtualization, *VEE*, 2009.

[70] Greg Regnier et.al., ETA: Experience with an Intel Xeon processor as a packet processing engine”, *A Symposium on High Performance Interconnects (HOT Interconnects)*, 2003.

[71] Greg Regnier et.al., TCP onloading for data center servers”, *IEEE Computer*, 2004.

[72] RDMA, *RDMA consortium, http://www.rdmaconsortium.org/*.

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

[74] K. W. Ross, Hash routing for collections of shared web caches, *IEEE Network, Vol. 11, No. 6*, November-December 1997.

[75] M. Satyanarayanan, Scalable, secure, and highly available distributed file access, *IEEE Computer, No. 5,* May 1990.

[76] W. Shi, et al., Load balancing for parallel forwarding, *IEEE Transactions on Networking, 13(4):790-801*, Aug. 2005.

[77] W. Shi, M. H. MacGregor, and P. Gburzynski, A scalable load balancer for forwarding internet traffic: Exploiting flow-level burstiness, *in Proc. of Symposium on Arch. for Networking and Communications Systems (ANCS),* Oct. 2005.

[78] W. Shi, et al., Sequence-preserving adaptive load balancers, *ANCS*, 2006.

[79] 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*.

[80] J. E. Smith and Ravi Nair, Virtual machines: architectures, implementations and Applications, *Morgan Kaufmann Publishers*, 2004.

[81] SNORT Network Intrusion Detection System, *http://www.snort.org/*.

[82] SPECweb 2005, *http://www.spec.org/web2005/*.

[83] L. Sprecklen, et al., Chip multithreading: opportunities and challenges, *HPCA*, 2005.

[84] J. Sugerman, et al., Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor", *USENIX*, 2001.

[85] L. Tan, et al., "A high throughput string matching architecture for intrusion detection and prevention, *ISCA*, 2005.

[86] 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.

[87] Tilera TILE64 mesh processor, *http://www.tilera.com/products/TILE64.php*.

[88] B. Veal, et al., Performance scalability of a multi-core web werver, *ANCS*, 2007.

[89] J. Verdu, et al., " MultiLayer processing - an execution model for parallel stateful packet processing ", *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS),* 2008.

[90] G. Welling, M. Ott, and S. Mathur, A cluster-based active router architecture, *IEEE Micro, No. 1,* Jan. 2001.

[91] J. Wiegart, et al., Challenges for scalable networking in a virtualized server, *IEEE MICRO*, 2007.

[92] P. Willmann, et. al., Concurrent direct network access for virtual machine monitors, *HPCA*, 2007.

[93] T. Woo, A modular approach to packet classification: algorithms and results, *IEEE INFOCOM*, 2000.

[94] Benjamin Wun, et al., Network I/O acceleration in heterogeneous multicore processors, *A Symposium on High Performance Interconnects (HOT Interconnects)*, 2006.

[95] Xen User Manual Version 3.0, *http://tx.downloads.xensource.com/downloads/docs/user/*.

[96] Xenoprof, *http://xenoprof.sourceforge.net/*.

[97] X. Zhang, L. Bhuyan and W. Feng, Anatomy of UDP and M-VIA for cluster communication, *Journal of Parallel and Distributed Computing (JPDC), Special issue on Design and Performance of Networks for Super-, Cluster-, and Grid-Computing, Vol. 65, Issue 10, October 2005, pp. 1290-1298*